REASONING ABOUT FLOATING POINT IN REAL-WORLD SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Wonyeol Lee
August 2023

Abstract

Continuous computations, which involve continuous data and operations on them, are ubiquitous in diverse areas such as machine learning and scientific computing. In theoretical studies of such computations, we typically use real numbers and exact operations. In practice, however, we often substitute floating-point numbers for the reals and apply inexact floating-point operations, which presents a clear discrepancy between the theory and practice of continuous computations.

In this dissertation, we aim at better understanding this discrepancy, especially for three different classes of real-world computations. First, for computations that implement math libraries using floats, we present automatic techniques to formally verify their correctness. Next, for computations that calculate derivatives of neural networks at floating-point inputs, we show theoretical results on their correctness. Lastly, for computations that train deep neural networks using floats, we present a systematic way to accelerate them using lower-precision floats.

Acknowledgments

Without the help of many great people around me, I could not have achieved anything I did during my Ph.D. studies. I deeply thank my advisor, Alex Aiken, not only for teaching me how to do research (i.e., how to find a good problem, how to approach a technical problem, how to write a good paper, and how to give a good talk), but also for encouraging me all the time especially when I went through difficulties. I thank Clark Barrett and Fredrik Kjolstad for serving on my reading/oral committee, and Thomas Icard and Caroline Trippel for serving on my oral committee. I also thank all the collaborators I worked with during my Ph.D. studies (and my military service in South Korea): Gwonsoo Che, Hyoungjin Lim, Sejun Park, Xavier Rival, Rahul Sharma, Hongseok Yang, and Hangyeol Yu. Finally, I thank all my friends and family members for supporting me unconditionally.

Contents

Abstract **iv**

Acknowledgments **v**

Contents **ix**

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Publications . . . 3

2 Background **5**
2.1 Floating-Point Formats and Numbers . . . 5
2.2 Floating-Point Operations and Rounding Errors . . . 6

3 Correctness of Highly Optimized Math Libraries **9**
3.1 Introduction . . . 9
3.2 Motivation . . . 11
3.3 Algorithm . . . 15
3.3.1 Core Language . . . 15
3.3.2 Symbolic Abstractions . . . 16
3.3.3 Construction of Symbolic Abstractions . . . 17
3.3.4 Computation of Precision Loss . . . 21
3.4 Case Studies . . . 23
3.4.1 The `sin` Implementation . . . 26
3.4.2 The `tan` Implementation . . . 27
3.4.3 The `log` Implementation . . . 28
3.5 Related Work . . . 29
3.6 Discussion . . . 31
3.7 Conclusion . . . 31

4 Correctness of Highly Accurate Math Libraries **32**
4.1 Introduction . . . 32
4.2 Motivation . . . 34
4.3 Abstraction . . . 37
4.3.1 Core Language . . . 37
4.3.2 Sound Abstractions . . . 38
4.3.3 Construction of Sound Abstractions . . . 40
4.4 Exploiting Exactness Properties . . . 43
4.4.1 Simple Exact Operations . . . 43
4.4.2 Sterbenz's Theorem . . . 44
4.4.3 Dekker's Theorem . . . 46
4.4.4 Nonzero Significand Bits . . . 49
4.4.5 Refined $(1+\varepsilon)$-property . . . 52
4.4.6 Ulp Error Bound . . . 55
4.5 Implementation . . . 56
4.6 Case Studies . . . 57
4.6.1 The `exp` Implementation . . . 60
4.6.2 The `sin` Implementation . . . 60
4.6.3 The `tan` Implementation . . . 60
4.6.4 The `log` Implementation . . . 61
4.7 Related Work . . . 61
4.8 Conclusion . . . 63

5 Correctness of Automatic Differentiation **64**
5.1 Introduction . . . 64
5.2 Preliminaries . . . 67
5.2.1 Notation and Definitions . . . 67
5.2.2 Neural Networks . . . 67
5.2.3 Automatic Differentiation . . . 68
5.2.4 Incorrect and Non-Differentiable Sets . . . 69
5.3 Neural Networks with Bias Parameters . . . 70
5.3.1 Characterization of the Incorrect Set . . . 71
5.3.2 Characterization of the Non-Differentiable Set . . . 71
5.3.3 Connection to Clarke Subderivatives . . . 73
5.4 Neural Networks without Bias Parameters . . . 74
5.4.1 Bounds for Non-Differentiable and Incorrect Sets . . . 75
5.4.2 Bounds for the Incorrect Set . . . 76
5.4.3 Conditions for Computing Standard Derivatives and Clarke Subderivatives . . . 77

5.5 Related Work . . . 78
5.6 Discussion . . . 78
5.7 Conclusion . . . 79

6 Acceleration of Deep Neural Network Training **81**
6.1 Introduction . . . 81
6.2 Problem . . . 84
6.2.1 Low-Precision Floating-Point Training . . . 84
6.2.2 Memory-Accuracy Tradeoff Problem . . . 86
6.2.3 NP-Hardness of the Problem . . . 87
6.3 Algorithm . . . 87
6.3.1 Precision Demotion for Saving Memory . . . 87
6.3.2 Precision Promotion for Handling Overflows . . . 88
6.4 Experiments . . . 90
6.4.1 Implementation . . . 90
6.4.2 Experiment Setups . . . 91
6.4.3 Comparison with Existing Precision Assignments . . . 92
6.4.4 Ablation Study: Precision Demotion and Promotion . . . 94
6.4.5 Choosing the Value of r . . . 96
6.5 Related Work . . . 97
6.6 Conclusion . . . 98

7 Conclusion **99**

A Appendix for Chapter 4 **100**
A.1 Complete Definitions and Rules . . . 100
A.1.1 Definition of Operations on Abstractions . . . 100
A.1.2 Rules for Constructing Abstractions . . . 101

B Appendix for Chapter 5 **105**
B.1 Formal Setup . . . 105
B.1.1 Piecewise-Analytic Functions . . . 105
B.1.2 Neural Networks . . . 107
B.1.3 Automatic Differentiation . . . 109
B.2 Upper Bounds on $|\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L)|$ . . . 112
B.2.1 Lemmas (Basic) . . . 112
B.2.2 Lemmas (Technical: Part 1) . . . 113
B.2.3 Theorem 5.7 (Main Lemmas) . . . 115
B.2.4 Theorem 5.7 (Main Proof) . . . 115

B.2.5 Lemmas (Technical: Part 2) . . . 116
B.2.6 Theorem 5.12 (Main Lemmas) . . . 119
B.2.7 Theorem 5.12 (Main Proof) . . . 121
B.3 Upper Bounds on $|\mathsf{inc}_\Omega(z_L)|$ . . . 122
B.3.1 Lemmas (Basic) . . . 122
B.3.2 Lemmas (Technical: Part 1) . . . 123
B.3.3 Lemmas (Technical: Part 2) . . . 125
B.3.4 Theorem 5.6 (Main Lemmas) . . . 129
B.3.5 Theorem 5.6 (Main Proof) . . . 132
B.3.6 Lemmas (Technical: Part 3) . . . 132
B.3.7 Theorem 5.14 (Main Lemma) . . . 135
B.3.8 Theorem 5.14 (Main Proof) . . . 135
B.4 Lower Bounds on $|\mathsf{ndf}_\Omega(z_L)|$ and $|\mathsf{inc}_\Omega(z_L)|$ . . . 136
B.4.1 Theorem 5.8 (Main Proof) . . . 136
B.4.2 Theorem 5.13 (Main Proof) . . . 138
B.4.3 Theorem 5.15 (Main Proof) . . . 140
B.5 Computation of Standard Derivatives . . . 142
B.5.1 Lemmas (Basic) . . . 142
B.5.2 Lemmas (Technical: Part 1) . . . 143
B.5.3 Lemmas (Technical: Part 2) . . . 144
B.5.4 Theorems 5.9 and 5.16 (Main Lemmas) . . . 146
B.5.5 Theorems 5.9 and 5.16 (Main Proofs) . . . 147
B.6 Computation of Clarke Subderivatives . . . 148
B.6.1 Lemmas (Basic) . . . 148
B.6.2 Lemmas (Technical) . . . 150
B.6.3 Theorems 5.10 and 5.17 (Main Lemmas) . . . 151
B.6.4 Theorems 5.10 and 5.17 (Main Proofs) . . . 154

C Appendix for Chapter 6 **156**
C.1 Problem: Deferred Proof . . . 156
C.2 Experiments: Deferred Details . . . 162
C.3 Experiments: Deferred Results . . . 163
C.3.1 Comparison with Existing Precision Assignments . . . 163
C.3.2 Ablation Study: Precision Demotion and Promotion . . . 163

Bibliography **173**

List of Tables

3.1 Important statistics of each implementation for case studies. 24
3.2 Summary of results. 24

4.1 Summary of results. 58

List of Figures

2.1 Bit representation of 64-bit double-precision floats. 6

3.1 The x86 assembly code of `exp` that ships with S3D. 12
3.2 The x86 assembly code of $\texttt{exp}_{\texttt{opt}}$ automatically generated by STOKE. 12
3.3 The syntax of the core language. 16
3.4 The rules for constructing a symbolic abstraction. 19
3.4 The rules for constructing a symbolic abstraction (continued). 20
3.5 Bounds on precision loss between implementations and mathematically exact results. 25

4.1 The computation of $r(x)$ in Intel's implementation of the log function. 36
4.2 The abstract syntax of our core language 37
4.3 Rules for constructing an abstraction of an expression 40
4.4 Rules for simple exact operations 44
4.5 Rules for applying Sterbenz's theorem 46
4.6 Rules for applying Dekker's theorem 47
4.7 Rule for using $\sigma(\cdot)$ 50
4.8 Rules for applying the refined $(1+\varepsilon)$-property 53
4.9 The ulp error of each implementation over an input interval. 59

6.1 Training trajectory of various models on CIFAR-100. 82
6.2 A diagram showing the tensors and operators used in a gradient computation. . . . 85
6.3 Results of training ShuffleNet-v2 on ImageNet with π_{fp32}, π_{unif}, π_{op}, $\pi_{\text{op}'}$, and $\pi_{\text{ours},r}$. 94
6.4 Memory-accuracy tradeoffs of π_{unif}, π_{op}, $\pi_{\text{op}'}$, and $\pi_{\text{ours},r}$ for four models and their smaller variants on CIFAR-10 and CIFAR-100. 95
6.5 Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$, $\pi_{\text{ours[inc]},r}$, and $\pi_{\text{ours[rand]},r}$ for three models on CIFAR-100. 96
6.6 Training ShuffleNet-v2 on ImageNet with $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. 96
6.7 Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$ for ShuffleNet-v2 on ImageNet-200-i ($i \in [5]$). . 97

C.1 The model network $\mathcal{M}$ and the loss network $\mathcal{L}$ used in the proof of Theorem 6.2. . . 157

C.2 A zoomed-in version of Figure 6.3 (left). 164
C.3 Results continued from Figure 6.4. 165
C.4 Training trajectories for the configurations shown in Figure 6.4. 166
C.5 Training trajectories for the configurations shown in Figure C.3. 167
C.6 Results corresponding to Figure 6.3. 168
C.7 Results corresponding to Figure 6.4. 169
C.8 Results corresponding to Figure C.3. 170
C.9 Results continued from Figure 6.5. 171
C.10 Results continued from Figure 6.6. 172

Chapter 1

Introduction

Continuous computations, which involve continuous data and operations on them, are becoming ever more prevalent in computer science (e.g., in machine learning, graphics, and security) as well as outside computer science (e.g., in scientific computing, statistics, and finance). Rigorous reasoning about these computations, therefore, is increasingly important as it can guarantee that desired results are computed and they are computed in a desired amount of time.

Doing this reasoning, however, is challenging due to a fundamental discrepancy between what is considered in theory and what is used in practice for continuous computations. In theory, we usually work with real numbers and arithmetic to model continuous values and operations, while in practice, even representing all the real numbers is impossible because actual computers have only finitely many bits. Hence, real numbers and arithmetic are approximated in practical computations, and floating-point numbers and arithmetic are "by far the most widely used" approximation [109, page 3] among many others (e.g., fixed-point numbers [157], posit numbers [59], and computable real numbers [156]); see Chapter 2 for background on floating point. We point out that the discrepancy between reals and floats is fundamental and not easily reconcilable: the set of all real numbers is uncountable and complete (i.e., does not have any "missing points" between the numbers), while the set of all floating-point numbers is finite and incomplete; further, real arithmetic is exact by definition and enjoys many nice properties (such as associativity), while floating-point arithmetic is mostly inexact due to rounding errors and lacks some of those properties (including associativity). These fundamental differences between reals and floats make it difficult to rigorously reason about continuous computations performed in practice.

To better understand the discrepancy between the theory and practice of continuous computations, considerable efforts have been made in several different areas. For example, rigorous rounding error analysis appeared as early as the 1940s in the work of John von Neumann, Alan Turing, and others [150, 153]; since then it has been actively studied, with an emphasis on numerical linear algebra, by numerical analysts [68, 157, 158]. As another example, many floating-point algorithms to compute elementary functions (e.g., exp, log, and sin) have been proposed, theoretically analyzed, and further

improved for more than a half century by computer scientists [64, 100, 108].

Despite such efforts, our theoretical understanding of practical continuous computations is still limited, for at least three reasons. First, even if some computations (e.g., algorithms for elementary functions) have been thoroughly studied from a theoretical perspective, what we actually use in practice is often not these computations, but rather their variants (e.g., implementations of the algorithms) that deviate from the original computations (e.g., to achieve better performance). This makes already established theoretical results inapplicable to practically used computations. Second, some computations are just beyond what we can analyze with existing theoretical tools. Donald Knuth stated this point in [82, page 229]: "Many serious mathematicians have attempted to analyze a sequence of floating point operations rigorously, but have found the task so formidable that they have tried to be content with plausibility arguments instead." Third, some computations (e.g., those from deep learning) started to be considered only recently, relative to the long history of floating point. Hence, simply not enough time has been given to us to develop a full understanding of these computations.

1.1 Contributions

Given this status, we aim to better understand and characterize the discrepancy between the theory and practice of continuous computations. In particular, we focus on three different classes of real-world computations that make use of floating point: computations that implement math libraries using floats; computations that calculate derivatives of neural networks at floating-point inputs; and computations that train deep neural networks using floats.

First, in Chapters 3 and 4, we consider floating-point implementations of math libraries, such as Intel's implementation of the C math library `math.h`. Due to the use of floats, these implementations inevitably have some errors in the outputs with respect to their theoretical specifications (e.g., $\exp : \mathbb{R} \to \mathbb{R}$), and these errors can be unexpectedly large as illustrated in [45, 46]. It is therefore important to formally verify the correctness of these implementations (i.e., bound their maximum errors), ideally automatically. This verification problem, however, has two challenges. First, highly optimized implementations of math libraries often intermix floating-point operations with bit-level operations (e.g., bit-shifts), and reasoning about such "mixed" code is challenging because floating-point operations behave "smoothly" while bit-level operations behave "discretely." Second, industry standard implementations of math libraries are often claimed to have extremely small error bounds, and proving such small bounds is in general challenging even for human beings. We present automatic techniques for the verification problem that address these two challenges based on abstraction, analytical optimization, and testing. We apply these techniques to Intel's implementations of `math.h` and prove correctness automatically.

Second, in Chapter 5, we consider automatic differentiation (AD), a class of algorithms for

computing the derivative of a given program, which includes the well-known backpropagation algorithm. Recent work has shown that AD over the reals is almost always correct in a mathematically precise sense [18, 71, 92, 101]. However, actual programs work with floating-point numbers (or machine-representable numbers in general), not reals. To better understand this gap, we study the correctness of AD when applied to a neural network with machine-representable parameters. In particular, we analyze two sets of parameters on which AD can be incorrect: the incorrect set on which the network is differentiable but AD does not compute its derivative, and the non-differentiable set on which the network is non-differentiable. For a neural network with "bias parameters", we first prove that the incorrect set is always empty. We then prove a tight bound on the size of the non-differentiable set, which is linear in the number of non-differentiabilities in activation functions, and give a simple necessary and sufficient condition for a parameter to be in this set. We further prove that AD always computes a Clarke subderivative even on the non-differentiable set. We also extend these results to neural networks possibly without bias parameters.

Third, in Chapter 6, we consider the training of deep neural networks. When training a network, keeping all tensors in high precision (e.g., 32-bit or 16-bit floats) is often wasteful, while keeping all tensors in low precision (e.g., 8-bit floats) can lead to unacceptable accuracy loss [78, 104, 147, 155]. It is therefore important to use a precision assignment—a mapping from all tensors to precision levels (high or low)—that keeps most of the tensors in low precision and leads to sufficiently accurate models. However, how to explore this memory-accuracy tradeoff in a systematic way has not been well-understood even empirically. We provide a heuristic technique that achieves the tradeoff by generating precision assignments that (i) use less memory and (ii) lead to more accurate models at the same time, compared to existing precision assignments. We evaluate our technique on convolutional networks for image classification tasks, and show that our method typically provides $> 2\times$ memory reduction over a baseline precision assignment while preserving training accuracy, and gives further reductions by trading off accuracy. Compared to other baselines which sometimes cause training to diverge, our method provides similar or better memory reduction while avoiding divergence.

1.2 Publications

Some parts of this dissertation have appeared in the following papers, which I wrote in collaboration with Alex Aiken, Sejun Park, and Rahul Sharma.

- Wonyeol Lee, Rahul Sharma, Alex Aiken. Verifying Bit-Manipulations of Floating-Point. In *Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, June 2016. Cited as [90].
- Wonyeol Lee, Rahul Sharma, Alex Aiken. On Automatically Proving the Correctness of math.h Implementations. In *Principles of Programming Languages (POPL)*, Los Angeles, CA, January 2018. Cited as [91].

- Wonyeol Lee, Sejun Park, Alex Aiken. On the Correctness of Automatic Differentiation for Neural Networks with Machine-Representable Parameters. In *International Conference on Machine Learning (ICML)*, Honolulu, HI, July 2023. Cited as [93].
- Wonyeol Lee, Rahul Sharma, Alex Aiken. Training with Mixed-Precision Floating-Point Assignments. *Transactions on Machine Learning Research (TMLR)*, June 2023. Cited as [94].

Chapter 2

Background

In this chapter, we introduce some basic definitions and properties related to floating point.

2.1 Floating-Point Formats and Numbers

We start with the definition of a floating-point format.

Definition 2.1. A tuple $(e_{\min}, e_{\max}, p) \in \mathbb{Z}^3$ is a *floating-point format* if $e_{\min} \leq e_{\max}$ and $p \geq 1$. We call $e_{\min}$ (or $e_{\max}$) the minimum (or maximum) exponent and p the precision of the format.

The most popular floating-point formats include: the 64-bit double-precision format defined as $(-1022, 1023, 53)$; the 32-bit single-precision format defined as $(-126, 127, 24)$; the 16-bit half-precision format defined as $(-14, 15, 11)$; and the 16-bit bfloat format defined as $(-126, 127, 8)$. The first three formats have been specified by the IEEE 754 standard since 1985 [73], while the last format has been introduced by Google Brain around 2016 [2].

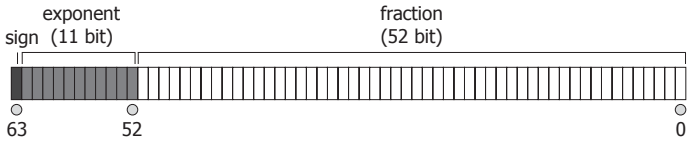
Let $(e_{\min}, e_{\max}, p)$ be a floating-point format. We define the floating-point numbers (or *floats*) of the format as follows. First, *finite* floating-point numbers are defined.

Definition 2.2. A real number $x \in \mathbb{R}$ is a *finite floating-point number* if

$$x = (-1)^s \times 2^e \times f_1.f_2 \cdots f_{p\,(2)}$$

for some $s \in \{0, 1\}$, $e \in [e_{\min}, e_{\max}] \cap \mathbb{Z}$, and $f_i \in \{0, 1\}$ such that (i) $f_1 = 1$, or (ii) $f_1 = 0$ and $e = e_{\min}$. We call s the *sign*, e the *exponent*, and $f_1.f_2 \cdots f_{p\,(2)}$ the *significand* (or fraction, or mantissa) of x. We call $x \in \mathbb{R}$ a *normal* number if $f_1 = 1$ or $x = 0$, and a *subnormal* (or denormal) number otherwise. We write $\mathbb{F} \subseteq \mathbb{R}$ as the set of all finite floating-point numbers.

In addition to finite ones, *non-finite* floating-point numbers are considered in many formats (e.g., those specified by the IEEE 754 standard) and usually defined as three values: $+\infty$, $-\infty$, and NaN



Type	Exponent part (`expn`)	Fraction part (`frac`)	Value
Zero	0	0	$(-1)^{\texttt{sign}} \times 0$
Subnormal	0	$\neq 0$	$(-1)^{\texttt{sign}} \times 2^{-1022} \times 0.\texttt{frac}_{(2)}$
Normal	$[1, 2046]$	unconstrained	$(-1)^{\texttt{sign}} \times 2^{\texttt{expn}-1023} \times 1.\texttt{frac}_{(2)}$
Infinity	2047	0	$(-1)^{\texttt{sign}} \times \infty$
NaN	2047	$\neq 0$	$(-1)^{\texttt{sign}} \times \perp$

Figure 2.1: Bit representation of 64-bit double-precision floats. The 11-bit exponent part, when interpreted as an unsigned integer, takes a value between 0 and 2047; it is converted to the exponent of a float by subtracting some integer, which is 1023 (called the exponent bias) in the case of normal floats. The 52-bit fraction part does not contain the leading fraction bit; it is converted to the significand of a float by adding the leading fraction bit, which is 1 in the case of normal floats.

("not a number"). These numbers can be generated by floating-point operations when the output is mathematically infinite (e.g., one divided by positive zero), too large in magnitude (e.g., one divided by 2^{-130} in the single-precision format), or mathematically undefined (e.g., zero divided by zero).

When representing floats in actual computers, we usually follow the IEEE 754 standard. Figure 2.1 illustrates the bit representation of 64-bit double-precision floats defined in the standard. In this representation, the most significant bit denotes the sign part, the next 11 bits denote the exponent part, and the remaining 52 bits denote the fraction part. The figure shows the values represented by different bit patterns. The bit representations of different floating-point formats can be defined similarly.

2.2 Floating-Point Operations and Rounding Errors

We now define floating-point operations. To do so, we first introduce the rounding function which converts a real number to a float.

Definition 2.3. The *rounding function* $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ is defined as

$$\mathrm{fl}(r) \triangleq \arg\min_{x \in \mathbb{F}} |r - x|$$

where ties are broken by choosing the x with 0 in the least significant position.

Here we use the "rounding to nearest even" instead of other rounding modes (e.g., "rounding toward 0"), since it is the default rounding mode in the IEEE 754 standard. Using the rounding function, we define floating-point operations.

Definition 2.4. For a real-valued operation $* \in \{+, -, \times, /\}$, the corresponding *floating-point operation* $\circledast$ is defined as

$$x \circledast y \triangleq \mathrm{fl}(x * y)$$

for operands $x, y \in \mathbb{F}$ with $|x * y| \leq \max \mathbb{F}$.

In this dissertation, we assume that each floating-point operation is left-associative.

Floating-point operations often introduce rounding errors, yet the error from each operation is rigorously modeled by the following $(1 + \varepsilon)$-property.

Theorem 2.5 ([109, Section 2.3.1])**.** *For any $* \in \{+, -, \times, /\}$ and $x, y \in \mathbb{F}$ with $|x * y| \leq \max \mathbb{F}$,*

$$x \circledast y = (x * y)(1 + \delta) + \delta'$$

for some $|\delta| < \varepsilon$ and $|\delta'| \leq \varepsilon'$, where $\varepsilon \triangleq 2^{-p}$ and $\varepsilon' \triangleq 2^{-p+e_{\min}}$ are constants.

The $(1 + \varepsilon)$-property states that each floating-point operation can introduce two kinds of errors, a multiplicative error modeled by δ and an additive error modeled by δ', but that each type of error is always bounded by very small constants ε and ε' respectively, regardless of the operands x and y. Here the constant ε is often called the machine epsilon.

To measure rounding errors, we define three metrics: the absolute error, the relative error, and the ulp (units in last place) error.

Definition 2.6. Let $r \in \mathbb{R}$ be an exact value and $r' \in \mathbb{R}$ be an approximation to r. The *absolute error*, *relative error*, and *ulp error* of r' with respect to r are defined as

$$\mathrm{ErrAbs}(r, r') \triangleq |r - r'|, \quad \mathrm{ErrRel}(r, r') \triangleq \left|\frac{r - r'}{r}\right|, \quad \mathrm{ErrUlp}(r, r') \triangleq \frac{|r - r'|}{\mathrm{ulp}(r)},$$

where

$$\mathrm{ulp}(r) \triangleq \begin{cases} 2^{e-p+1} & \text{if } |r| \in [2^e, 2^{e+1}) \text{ with } e \in [e_{\min}, e_{\max}] \cap \mathbb{Z}, \\ 2^{e_{\min}-p+1} & \text{if } |r| \in [0, 2^{e_{\min}}). \end{cases}$$

Here, $\mathrm{ulp}(r)$ represents the gap between the two adjacent floats $x, y \in \mathbb{F}$ that surround r (i.e., $x \leq r < y$ if $r \geq 0$, and $x < r \leq y$ if $r < 0$). For example, the ulp error of a floating-point operation with respect to the corresponding exact operation is always bounded by 0.5 ulps:

$$\mathrm{ErrUlp}(x * y, x \circledast y) \leq \frac{1}{2}$$

for any $x, y \in \mathbb{F}$ with $|x * y| \leq \max \mathbb{F}$. Although the absolute and relative errors are widely known, the ulp error is more commonly used than the other two when measuring the rounding error of math libraries. The absolute error can be large even when the result is incorrect only in the least significant bit. The relative error does not suffer from this problem but it is undefined at $r = 0$. On the other

hand, the ulp error is proportional to the relative error (Theorem 2.7), is always defined, and hence is preferable. Therefore, Chapters 3 and 4 focus on the ulp error of floating-point implementations.

The ulp error is closely related to the relative error as follows.

Theorem 2.7 ([109, Section 2.3.3]). *For any* $|r| \leq \max \mathbb{F}$ *and* $r' \in \mathbb{R}$*,*

$$\begin{aligned} \mathrm{ErrUlp}(r, r') &\leq \mathrm{ErrRel}(r, r') \cdot \frac{1}{\varepsilon} \qquad \textit{if} \quad r \neq 0, \\ \mathrm{ErrRel}(r, r') &\leq \mathrm{ErrUlp}(r, r') \cdot 2\varepsilon \qquad \textit{if} \quad |r| \geq 2^{e_{\min}}. \end{aligned}$$

As a corollary, for any $|r| \in [2^{e_{\min}}, \max \mathbb{F}]$ and $r' \in \mathbb{R}$, we have

$$\frac{\mathrm{ErrRel}(r, r')}{2\varepsilon} \leq \mathrm{ErrUlp}(r, r') \leq \frac{\mathrm{ErrRel}(r, r')}{\varepsilon}.$$

In the following chapters, we will make use of the definitions and results presented above. More details on floating point can be found in the following books and survey articles: [16, 52, 68, 109, 114] and [82, Chapter 4.2].

Chapter 3

Correctness of Highly Optimized Math Libraries

3.1 Introduction

Highly optimized implementations of math libraries rely on intermixing floating-point and bit-level code. Even though such code is widely used, automatic formal verification of these implementations has remained an open challenge [137]. Although it has been demonstrated that it is possible to construct machine-checkable proofs of correctness by hand for floating-point algorithms of the level of sophistication we are interested in [61, 62], no existing automated verification technique is capable of analyzing these implementations. In this chapter, we present a first step towards addressing this challenge.

Bit-precise floating-point reasoning is hard: floating-point is an approximation to real arithmetic, but floating-point numbers do not obey the algebraic axioms of real numbers due to rounding errors. The situation becomes even more difficult in the presence of bit-level operations, such as bit-manipulations of the floating-point representation. To illustrate *mixed* code, consider an implementation that computes the 64-bit floating-point number 2^n from a small integer n. A naive implementation would first compute the integer representing 2^n and then perform the computationally expensive operation of converting an integer to a 64-bit floating-point number. Alternatively, the same result can be obtained by bit-shifting $n + 1023$ left by 52 bits (Figure 2.1). Existing static analyses for floating-point arithmetic would be stumped by the bit-shift operation and would fail to prove the functional correctness of this trick. Moreover, such tricks are routine in real code [54, 106].

Before explaining our solution, it is important to understand why existing automated techniques (based on testing, model checking, and abstract interpretation) are inadequate. The simplest verification technique is exhaustive testing of all possible inputs. This approach is feasible for a function like `expf` that computes the exponential of a 32-bit single-precision floating-point number. However,

the number of 64-bit double-precision floating-point numbers (or *doubles*) is too large for brute force enumeration to be tractable.

A plausible verification strategy involves encoding correctness as the validity of a SMT formula [35]. However, the specifications of interest here are transcendentals and these (e^x, $\sin(x)$, etc.) cannot be encoded precisely in existing SMT theories. Verifiers based on abstract interpretation, such as ASTREE and FLUCTUAT, use pattern matching to handle certain bit-trick routines in commercial floating-point avionics code [54, 106]. Our goal is a general technique.

Our approach to the problem is to divide and conquer. For a given floating-point implementation, we consider non-overlapping intervals that are subsets of the possible range of inputs. We require each interval I to satisfy the following property: if we statically know that the inputs are restricted to I, the bit-level operations can be removed from the implementation by partial evaluation. Then, for each interval, we have a specialized implementation that is composed exclusively of floating-point operations and thus amenable to abstraction-based techniques. Our main contribution is to devise a procedure to construct such intervals (§3.3). There is one significant subtlety: The intervals do not always fully cover the space and we must deal with potential "gaps" between intervals. Commercial tools such as FLUCTUAT [43, 54] also subdivide the input range (with no gaps) to improve precision, and our technique can be seen as a systematic method to construct these subdivisions. We analyze the implementations specialized for each interval and report the maximum error between the implementation and the ideal mathematical specification.

We make the following contributions.

- We describe the first general technique for verification of mixed floating-point and bit-level code. We are unaware of any automatic or semi-automatic verification technique that can prove the functional correctness of the production grade benchmarks we consider. Prior to this work, formal verification of such benchmarks required manual construction of machine-checkable proofs [61, 62].

- We reduce the problem of computing bounds on numerical errors to an optimization problem and leverage state-of-the-art techniques for analytical optimization. While our method is not fully automatic, these techniques automate one of the most difficult aspects of the problem and make verification of complex implementations feasible.

- Our technique performs verification at the binary level, not on source code or a model of the program. Thus, the derived bounds apply to the actual code that executes directly on the hardware.

We evaluate our technique on three implementations of transcendental functions from Intel's libraries: a bounded periodic function (`sin`, §3.4.1), an unbounded discontinuous periodic function (`tan`, §3.4.2), and an unbounded continuous function (`log`, §3.4.3). We are able to successfully bound the difference between the result computed by these implementations and the exact mathematical result. For each of these functions, we also trade precision for performance and create significantly

more efficient variants that produce approximately correct results. Using our technique, we are able to provide a bound on the difference between the approximate variants and the mathematical specifications. These results demonstrate the generality of our technique and address some of the drawbacks of manually constructed proofs: modifying the manual proofs to prove even slightly different theorems is difficult [61, 62]. To quote Harrison [61],

> *Nontrivial proofs, as are carried out in the work described here, often require long and complicated sequence of rules. The construction of these proofs often requires considerable persistence. Moreover, the resulting proof scripts can be quite hard to read, and in some cases hard to modify to prove a slightly different theorem.*

The rest of this chapter is organized as follows. §3.2 provides an overview of our verification technique through an example, and §3.3 presents the technique in detail which combines abstraction, analytical optimization, and testing. §3.4 discusses evaluation and §3.5 surveys prior work. Finally, §3.6 gives a discussion of future work and §3.7 concludes.

3.2 Motivation

S3D [23] is a combustion chemistry simulation that is heavily used in research on developing more efficient and cleaner fuels for internal combustion engines. The performance of the exponential function is so important for this task that the developers ship a hand-coded x86 assembly implementation `exp` (Figure 3.1), which is inspired by the implementation of the exponential function present in CUDA libraries for GPUs. There is no source code as it has been implemented directly in assembly. There is also no documentation available regarding `exp` except that it is supposed to compute e^x for $x \in [-2.6, 0.12]$. As is characteristic of highly optimized floating-point implementations, `exp` contains bit-level operations (rounding to integer on line 3, converting double to integer on line 4, bit-vector addition on line 5, bit-shift on line 6, and bit-shuffle on line 7).

The main algorithm used by `exp` first computes an integer N and a reduced value d:

$$N = \text{round}\left(x \cdot \log_2 e\right), \; d = x - (\log 2)N, \tag{3.1}$$

where $\log(\cdot)$ without a base denotes the natural logarithm. Then it uses the following identity to compute e^x:

$$e^x = e^{(\log 2)N} \cdot e^d = 2^N \cdot e^d.$$

The implementation `exp` computes 2^N (a double in IEEE representation) from N (a 64-bit integer in 2's complement representation) using bit-level operations. It computes e^d using a Taylor series expansion of degree 12:

$$e^d \approx \sum_{i=0}^{12} \frac{d^i}{i!}.$$

```
 1  vmovddup    %xmm0, %xmm0
 2  vmulpd      L2E,   %xmm0, %xmm2
 3  vroundpd    $0,    %xmm2, %xmm2
 4  vcvtpd2dqx  %xmm2, %xmm3
 5  vpaddd      B,     %xmm3, %xmm3
 6  vpslld      $20,   %xmm3, %xmm3
 7  vpshufd     $114,  %xmm3, %xmm3
 8  vmulpd      C1,    %xmm2, %xmm1
 9  vmulpd      C2,    %xmm2, %xmm2
10  vaddpd      %xmm1, %xmm0, %xmm1
11  vaddpd      %xmm2, %xmm1, %xmm1
12  vmovapd     T1,    %xmm0
13  vmulpd      T12,   %xmm1, %xmm2
14  vaddpd      T11,   %xmm2, %xmm2
15  vmulpd      %xmm1, %xmm2, %xmm2
16  vaddpd      T10,   %xmm2, %xmm2
17  vmulpd      %xmm1, %xmm2, %xmm2
18  vaddpd      T9,    %xmm2, %xmm2
19  vmulpd      %xmm1, %xmm2, %xmm2
20  vaddpd      T8,    %xmm2, %xmm2
21  vmulpd      %xmm1, %xmm2, %xmm2
22  vaddpd      T7,    %xmm2, %xmm2
23  vmulpd      %xmm1, %xmm2, %xmm2
24  vaddpd      T6,    %xmm2, %xmm2
25  vmulpd      %xmm1, %xmm2, %xmm2
26  vaddpd      T5,    %xmm2, %xmm2
27  vmulpd      %xmm1, %xmm2, %xmm2
28  vaddpd      T4,    %xmm2, %xmm2
29  vmulpd      %xmm1, %xmm2, %xmm2
30  vaddpd      T3,    %xmm2, %xmm2
31  vmulpd      %xmm1, %xmm2, %xmm2
32  vaddpd      T2,    %xmm2, %xmm2
33  vmulpd      %xmm1, %xmm2, %xmm2
34  vaddpd      %xmm0, %xmm2, %xmm2
35  vmulpd      %xmm1, %xmm2, %xmm1
36  vaddpd      %xmm0, %xmm1, %xmm0
37  vmulpd      %xmm3, %xmm0, %xmm0
38  retq
```

Figure 3.1: The x86 assembly code of `exp` that ships with S3D [23]. Instructions have been reordered to aid understanding, without affecting the output.

```
 1  vmulpd      L2E,   %xmm0, %xmm2
 2  vroundpd    C, %xmm2, %xmm2
 3  // C = $0xfffffffffffffffe
 4  vcvttpd2dq  %xmm2, %xmm3
 5  vpaddw      B,     %xmm3, %xmm3
 6  vpsllq      $0x14, %xmm3, %xmm3
 7  vpshufd     $0x3,  %xmm3, %xmm3
 8  vmulpd      C1,    %xmm2, %xmm1
 9  vaddpd      %xmm1, %xmm0, %xmm1
10
11
12  vmovapd     T1,    %xmm0
13  vlddqu      T8,    %xmm2
14  vmulpd      %xmm1, %xmm2, %xmm2
15  vaddpd      T7,    %xmm2, %xmm2
16  vmulpd      %xmm1, %xmm2, %xmm2
17  vaddpd      T6,    %xmm2, %xmm2
18  vmulsd      %xmm1, %xmm2, %xmm2
19  vaddpd      T5,    %xmm2, %xmm2
20  vmulpd      %xmm1, %xmm2, %xmm2
21  vaddpd      T4,    %xmm2, %xmm2
22  vmulpd      %xmm1, %xmm2, %xmm2
23  vaddpd      T3,    %xmm2, %xmm2
24  vmulsd      %xmm1, %xmm2, %xmm2
25  vaddpd      T2,    %xmm2, %xmm2
26  vmulsd      %xmm1, %xmm2, %xmm2
27  vaddsd      %xmm0, %xmm2, %xmm2
28  vmulpd      %xmm1, %xmm2, %xmm1
29  vaddsd      %xmm0, %xmm1, %xmm0
30
31
32
33
34
35
36
37  vmulpd      %xmm3, %xmm0, %xmm0
38  retq
```

Figure 3.2: The x86 assembly code of $\texttt{exp}_{\texttt{opt}}$ automatically generated by STOKE [137]. This code is less precise but has better performance compared to `exp` (Figure 3.1).

Next, we relate this algorithm with Figure 3.1. Our description below elides several details that are important for performance (such as the use of vector instructions), and focuses on functionality. The calling convention used by `exp` includes storing the first argument and the return value of a function in the register `xmm0`. We omit details about the x86 syntax and describe the implementation at a high level. The code is divided into several blocks by the horizontal lines in Figure 3.1 and the instructions within a block compute a value of interest.

- The first block (lines 1-3) computes N from the input x. The second instruction multiplies x by `L2E` (the double closest to $\log_2 e$) and the third instruction rounds the result.
- The second block (lines 4-7) computes 2^N from N, using bit-vector addition (line 5), bit-shift (line 6), and bit-shuffle (line 7). The bit-vector represented by `B` is `0x000003ff000003ff` (in hex). Recall that `0x3ff` in hex is 1023 in decimal, which is the bias used for the exponent in doubles (Chapter 2).
- The third group (lines 8-11) computes d from x and N. This computation uses $\log 2$ which is a transcendental number (Eq. (3.1)). To maintain accuracy, $\log 2$ is represented as the sum of two doubles: $c_1 = -0.69315\cdots$ (line 8) and $c_2 = -2.31904\cdots\times 10^{-17}$ (line 9) where $\log 2 \approx -c_1 - c_2$. This representation effectively provides more than a hundred bits to represent $\log 2$ with the desired accuracy. Using c_1 and c_2, we can compute $d \approx (x + c_1 N) + c_2 N$ (lines 10-11).
- The fourth block (lines 12-36) computes e^d from d, using a Taylor series expansion of degree 12. The constant `T`i represents $\frac{1}{i!}$.
- The last group (line 37-38) returns $e^x = 2^N \cdot e^d$. Recall that 2^N is computed exactly by the second block and e^d is computed approximately by the fourth block.

Given this non-trivial implementation, a valid question is: What does it achieve? Fundamentally, since there are only a finite number of floating-point numbers, no implementation can compute e^x exactly. All floating-point implementations provide only an approximate result which has finite precision. Therefore, one possible measure of a correctness of such implementations is given by *precision loss*: the deviation of the computed answer from the mathematically exact result. Our goal in this chapter is to provide sound bounds on the maximum precision loss of such implementations.

The first challenge associated with floating-point implementations is that of rounding errors. As is standard, we model rounding error using non-determinism. For example, line 2 of Figure 3.1 multiplies the input x and a constant `L2E`, and we model its output as an element chosen non-deterministically from the set $\{(x \times \texttt{L2E})(1+\delta) : |\delta| < \varepsilon\}$, where $+$ and $\times$ denote real-valued addition and multiplication, respectively.[1] The quantity δ models the rounding error and the *machine epsilon* ε provides a bound on the rounding errors of floating-point multiplication (Chapter 2).

[1]This modeling is sound because in this chapter we do not consider subnormals for simplicity. It is straightforward to incorporate them in our verification techniques, e.g., by adding δ' (with $|\delta'| < \varepsilon'$) to the modeling as discussed in Chapter 2.

The next challenge, which has not been systematically addressed previously, is associated with bit-level operations. We use a divide-and-conquer approach to address the challenge. We denote the set of possible inputs by the interval X and create a set of intervals $\mathcal{I} = \{I_k : k \in \mathbb{Z} \text{ and } I_k \subseteq X\}$ such that the following property holds:

$$\forall x \in I_k.\ \{(x \times \texttt{L2E})(1+\delta) : |\delta| < \varepsilon\} \subseteq \left(k - \frac{1}{2}, k + \frac{1}{2}\right). \tag{3.2}$$

This decomposition is useful because it ensures that for each input $x \in I_k$ the rounded output N of line 3 is k. We show how to obtain this decomposition in §3.3.

Given such a decomposition $\mathcal{I}$, we run $|\mathcal{I}|$ separate analyses, where the k^{th} analysis restricts the inputs to I_k. In the k^{th} analysis, the result N of rounding (line 3) is always k which is the only input to the bit-level operations of the second block (lines 4-7). Hence, it is sound to replace the bit-level operations by an instruction that moves the double 2^k to the register `xmm3`. This specialized code consists exclusively of floating-point operations.

Next, each analysis generates a separate *symbolic representation* that over-approximates the possible outputs of its specialized code. The symbolic representation $\mathcal{A}_{\vec{\delta}}(x)$ is a function of the input x and the rounding errors $\vec{\delta}$. For example, the symbolic representation of the multiplication of x and `L2E` (line 2) is given by $(x \times \texttt{L2E})(1+\delta)$. In general, if an expression e_1 has symbolic representation $\mathcal{A'}_{\vec{\delta}}(x)$ and e_2 has symbolic representation $\mathcal{A''}_{\vec{\delta}}(x)$ then the symbolic representation of $e_1 *_{\texttt{f}} e_2$ (where $*_{\texttt{f}}$ is floating-point addition or multiplication) is given by $(\mathcal{A'}_{\vec{\delta}}(x) * \mathcal{A''}_{\vec{\delta}}(x))(1+\delta')$, where $*$ is real-number addition or multiplication and δ' is a new variable representing the rounding error of the operation $*_{\texttt{f}}$.

Finally, each analysis uses analytical optimization to maximize the difference between the symbolic representation and the ideal mathematical result in the interval of interest. Several representations of precision loss are possible. If we measure precision loss using *absolute error* then the k^{th} analysis solves the following optimization problem:

$$\max_{x \in I_k, \vec{\delta}} \left| \mathcal{A}_{\vec{\delta}}(x) - e^x \right|.$$

For `exp`, we use the set of inputs $X = [-4, 4]$ that results in 13 intervals, i.e., $|\mathcal{I}| = 13$ and the symbolic representations have a maximum of 29 δ variables for rounding errors. The maximum absolute error reported across all intervals is 5.6×10^{-14}.

There is one remaining issue. The division of the entire input range X into intervals $\mathcal{I} = \{I_k\}$ may result in some $x \in X$ where $x \notin I_k$ for any k. For example, consider the input $\frac{1}{2 \cdot \texttt{L2E}}$. Due to rounding errors, it is not clear whether, for this input, the result N of line 3 would be 0 or 1. Therefore, this input is not included in any $I_k \in \mathcal{I}$ (Eq. (3.2)). For `exp`, there are 36 such doubles that are not included in any interval. For these inputs, we simply execute the program on each one and directly measure the precision loss. The maximum absolute error for these 36 inputs is 2.1×10^{-14} (which is close to but less than 5.6×10^{-14}).

Another common representation for precision loss is ulp error that signifies the number of floating-point numbers between the computed and the mathematically exact result. We show how to compute ulp error (for each interval) in §3.3.4. For `exp`, the maximum ulp error over X is 14, that is, there are at most 14 doubles between the output of `exp` and e^x. We are unaware of any previous automated analysis that can prove this result.

It is possible to further improve the performance of `exp` by sacrificing more precision. Despite its heavy use of the exponential function, S3D loses precision elsewhere and does not require precise results from `exp` to maintain correctness. One possible strategy involves asking a developer to create a customized implementation that has better performance at the cost of less precise results (§3.4). Such implementations can also be automatically generated using stochastic optimizers such as STOKE [137]. At a high level, STOKE makes random changes to binaries until they get faster and remain "approximately correct" on some tests.

The binary $\texttt{exp}_{\texttt{opt}}$ (Figure 3.2) is automatically generated by making random changes to `exp` using STOKE. The differences between $\texttt{exp}_{\texttt{opt}}$ and `exp` are that $\texttt{exp}_{\texttt{opt}}$ computes d as $x + c_1 N$ (without using c_2) and it uses a Taylor series expansion of degree 8 (instead of 12). The authors of [137] claim that $\texttt{exp}_{\texttt{opt}}$ has an ulp error below 10^7 from `exp` and improves the performance of the diffusion task of S3D by 27%. However, the correctness guarantees provided by STOKE are statistical and there is no formal guarantee that $\texttt{exp}_{\texttt{opt}}$ cannot produce results with an ulp error much greater than 10^7 for some unobserved input.

We use our analysis to bound the precision loss between `exp` and $\texttt{exp}_{\texttt{opt}}$. Our analysis reports that the maximum absolute error between `exp` and $\texttt{exp}_{\texttt{opt}}$ is 1.2×10^{-8} and the maximum ulp error is 1.9×10^6 which though large is well below the desired bound of 10^7 ulps. Therefore, we have proven formally that the STOKE generated code respects the ulp bound of 10^7 for all inputs $x \in [-4, 4]$. This verification task was left as a challenge in [137].

3.3 Algorithm

We now describe our procedure for estimating the precision loss of an implementation with respect to its mathematical specification. The procedure has two main steps. First, we construct a symbolic abstraction that soundly over-approximates the implementation. Next, we use an analytical optimization procedure to obtain a precise bound on the deviations the symbolic abstraction can have from the mathematical specification. We start by defining the syntax of a core language that we use for the formal development.

3.3.1 Core Language

Recall that the implementations of interest are highly optimized x86 binaries. The x86 ISA has more than two thousand different instruction variants, so we desugar x86 programs into a core expression

$$
\begin{array}{rrcl}
\text{expression} & e & ::= & c \mid \chi \mid \mathsf{L}[e] \mid \\
 & & & e *_{\mathsf{b}} e \mid e *_{\mathsf{s}} e \mid \\
 & & & e *_{\mathsf{i}} e \mid e *_{\mathsf{f}} e \mid *_{\mathsf{c}} e \\
\text{bitwise operators} & *_{\mathsf{b}} & \in & \{\mathsf{AND}, \mathsf{OR}, \cdots\} \\
\text{shift operators} & *_{\mathsf{s}} & \in & \{\ll, \gg, \cdots\} \\
\text{bit-vector arithmetic operators} & *_{\mathsf{i}} & \in & \{+_{\mathsf{i}}, -_{\mathsf{i}}, \times_{\mathsf{i}}, \cdots\} \\
\text{floating-point operators} & *_{\mathsf{f}} & \in & \{+_{\mathsf{f}}, \times_{\mathsf{f}}, /_{\mathsf{f}}, \cdots\} \\
\text{casting operators} & *_{\mathsf{c}} & \in & \{\mathsf{i2f}, \mathsf{f2i}, \mathsf{round}, \cdots\}
\end{array}
$$

Figure 3.3: The syntax of the core language.

language that is the subject of our analysis.

Figure 3.3 shows the grammar representing the expressions in this language. An *expression* e in the core language can be a 64-bit constant c, a 64-bit input χ, the result of a table lookup $\mathsf{L}[\cdot]$, or the application of a unary or a binary operator to subexpression(s). All expressions in this language evaluate to 64-bit values. The x86 implementations often use 128-bit SSE registers and SIMD instructions for efficiency. However, these operations can be expressed in our language by desugaring them to multiple operations applied to multiple 64-bit expressions. For brevity, we also restrict our presentation to settings with only a single input and a single lookup table. It is straightforward to generalize our results to implementations with multiple inputs and multiple tables (but see discussion in §3.6 of scaling issues in handling multiple inputs).

The operators relevant for our benchmarks are shown in Figure 3.3. The bitwise operators include bitwise-and (AND) and bitwise-or (OR) that are used for masking bits. The left shift ($\ll$) and the right shift ($\gg$) operators are "logical" shifts (as opposed to "arithmetic" shifts) and introduce zeros. The bit-vector arithmetic operators are "signed" operators that interpret the argument bits as 64-bit integers in the 2's complement notation. The floating-point operators interpret the argument bits as 64-bit double-precision floating-point numbers (or doubles). The casting operator i2f consumes a bit-string, interprets it as an integer written in the 2's complement notation (e.g., 42), and generates a bit-string that when interpreted as a double represents a value equal to the integer (e.g., 42.0). The round operator rounds to the nearest integer (e.g, 42.1 is rounded to 42.0) and the f2i operator first rounds and then converts the rounded double to a 64-bit integer.

For any expression e, the concrete semantics is denoted by $\mathcal{E}(e) : \{0,1\}^{64} \to \{0,1\}^{64}$. That is, the value obtained by evaluating e with an input x is given by $\mathcal{E}(e)(x)$. The definition of $\mathcal{E}(\cdot)$ is standard and we omit it.

3.3.2 Symbolic Abstractions

Our goal is to compute a symbolic representation that over-approximates the behaviors of an expression e. Constructing this abstraction is difficult due to the interplay between floating-point and *bit-level* (bitwise, shift, bit-vector arithmetic, and casting) operations. Therefore, we define this abstraction

piecewise. We restrict the inputs to small enough intervals—thus ensuring that the bit-level operations are amenable to static analysis—and we construct multiple abstractions, one for each interval.

The description of our abstractions requires some operators relating real numbers and doubles. The function $\mathrm{d2R} : \{0,1\}^{64} \to \mathbb{R} \cup \{\pm\infty, \mathrm{NaN}\}$ is the natural map from doubles to real numbers. It handles infinity and not-a-number by mapping them to $\{\pm\infty, \mathrm{NaN}\}$. This mapping is also extended to sets of doubles in the obvious way. If $\mathcal{P}(\cdot)$ denotes the power set of a given set then the inverse map $\mathrm{R2d} : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\{0,1\}^{64})$ maps a subset S of real numbers to the largest set of doubles such that

$$\forall x \in \mathrm{R2d}(S).\ \mathrm{d2R}(x) \in S.$$

For brevity, we use the abbreviation $\widehat{S} \triangleq \mathrm{R2d}(S)$. We use X to denote the interval over real numbers over which we want to compute the precision loss. Therefore, the input χ ranges over the set $\widehat{X} = \mathrm{R2d}(X)$.

We next define a symbolic abstraction. Let $\mathcal{I} = \{[l_1, r_1], \cdots, [l_n, r_n]\}$ denote a set of intervals in $\mathbb{R}$, where $[l_i, r_i] \subseteq X$ for all i. The *symbolic representation* $\mathcal{A}_{\vec{\delta}} : \mathbb{R} \to \mathbb{R} \cup \{\bot\}$ is a function of $x \in \mathbb{R}$ and $\vec{\delta} = (\delta_1, \cdots, \delta_m)$, where each $\delta_i \in \mathbb{R}$ represents a rounding error. The fact that $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ is an abstraction of e is defined as follows.

Definition 3.1. $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ is *a symbolic abstraction* of e if for all intervals $I \in \mathcal{I}$, for all doubles $x \in \widehat{I}$,

$$\mathrm{d2R}(\mathcal{E}(e)(x)) \in \mathcal{A}(\mathrm{d2R}(x))$$

where $\mathcal{A}(y) \triangleq [\min_{\vec{\delta}} \mathcal{A}_{\vec{\delta}}(y), \max_{\vec{\delta}} \mathcal{A}_{\vec{\delta}}(y)]$.

We discuss a procedure to construct a symbolic abstraction $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ of an expression e next.

3.3.3 Construction of Symbolic Abstractions

The expressions consist of floating-point and bit-level parts. To reason about bit-level operations, we need to keep track of subexpressions that are constant or are determined by a small subset of the bits of the input χ. To this end, we define a *book-keeping map* $\mathcal{B} : \{0,1\}^{64} \to \{0,1\}^{64} \cup \{\bot\}$. If $\mathcal{B}(x) \neq \bot$ then for all $i = 0, \cdots, 63$, the i^{th} bit of $\mathcal{B}(x)$, denoted by $\mathcal{B}(x)[i]$, is either 0, or 1, or a boolean function of the bits $x[63], \cdots, x[0]$. For instance, consider an expression $e = (x \ \mathsf{AND}\ \texttt{0x80}^{15})\ \mathsf{OR}\ \texttt{0x3fe0}^{13}$ which computes a double $\mathrm{sgn}(x) \times 0.5$, where b^i is the string with i b's, and $\mathrm{sgn}(\cdot)$ is the sign function. The book-keeping map for e is $\mathcal{B}(x) = x[63]01^9 0^{53}$, and it represents that the sign bit of e is that of x and the rest of the bits of e are the same as that of $\texttt{0x3fe0}^{13}$.

Just like symbolic abstractions, the book-keeping map is also defined piecewise over different intervals of $\mathcal{I}$. We give the formal definitions of definedness next.

Definition 3.2.

$$(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \text{ defined} \iff \forall I \in \mathcal{I}.\ \forall x \in I.\ \mathcal{A}_{\vec{\delta}}(x) \neq \bot,$$
$$(\mathcal{I}, \mathcal{B}) \text{ defined} \iff \forall I \in \mathcal{I}.\ \forall x \in \widehat{I}.\ \mathcal{B}(x) \neq \bot.$$

Next, we relate these abstractions with concrete semantics.

Definition 3.3. $(\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$ is *consistent* with e if the following hold:

$$(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \text{ defined} \implies (\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \text{ is a symbolic abstraction of } e,$$
$$(\mathcal{I}, \mathcal{B}) \text{ defined} \implies \forall I \in \mathcal{I}.\ \forall x \in \widehat{I}.\ \mathcal{E}(e)(x) = \mathcal{B}(x).$$

An important case is when every bit of $\mathcal{B}$ is a constant over each interval and we define it separately.

Definition 3.4.

$$(\mathcal{I}, \mathcal{B}) \text{ constant} \iff \forall I \in \mathcal{I}.\ \exists n \in \{0,1\}^{64}.\ \forall x \in \widehat{I}.\ \mathcal{B}(x) = n.$$

Figure 3.4 lists the rules for construction of symbolic abstractions. These rules have the following form: $e \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$, read as e is provably consistent with $(\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})$. These rules use the following notations. For a set S, $\mathrm{Id}_S : S \to S$ denotes the identity function. For a function $f : T \to U$ and a subset $S \subseteq T$, $f\restriction_S : S \to U$ is a restriction of f on S.

The rules for atomic expressions are direct (CONST and INPUT). For a table lookup, the index should be a constant over each interval (LOOKUP). Here, the result of the lookup can be obtained via the concrete semantics. For example, consider a table lookup $\mathsf{L}[e]$ where e represents the bits of the exponent of χ. Also assume that $\mathcal{I} = \{I_k : \forall x \in \widehat{I_k}.\ \text{exponent of } x = k\}$. Then, the lookup corresponding to the k^{th} interval provides the bits $\mathsf{L}[k]$, which are recorded in the book-keeping map and the symbolic representation. The rule for i2f is similar. Similarly, if all the arguments to a binary operator $*$ are constant over each interval then the resulting values can be obtained by evaluation (CONSTARG). For example, suppose that for all $I_k \in \mathcal{I}$, the book-keeping map says that for all $x \in \widehat{I_k}$, the expression e_1 evaluates to k and e_2 evaluates to $k+1$. Then the book-keeping map for $e_1 +_{\mathsf{i}} e_2$ maps $x \in \widehat{I_k}$ to $2k+1$. Note that for bit-vector arithmetic operations, only the rule CONSTARG applies.

The rule for bitwise operations, when $\mathcal{B}$ is defined but the bits are not constant (BITOP), requires a refinement of two sets of intervals. This operation is defined as follows:

$$\text{refine}(\mathcal{I}_1, \mathcal{I}_2) \triangleq \{I_1 \cap I_2 \neq \emptyset : I_1 \in \mathcal{I}_1, I_2 \in \mathcal{I}_2\}.$$

The refinement operation is necessary because the intervals over which $\mathcal{B}_1$ (or $\mathcal{A}_{1,\vec{\delta}}$) is defined piecewise can be different from those intervals for $\mathcal{B}_2$ (or $\mathcal{A}_{2,\vec{\delta}}$). For instance, for $\mathcal{I}_1 = \{[-3,0),(0,3]\}$ and $\mathcal{I}_2 = \{[-3,-1),(-1,1),(1,3]\}$, $\text{refine}(\mathcal{I}_1, \mathcal{I}_2) = \{[-3,-1),(-1,0),(0,1),(1,3]\}$. Note that the refined intervals do not necessarily cover the original intervals: there can be inputs $x \in I_1$ where $I_1 \in \mathcal{I}_1$ and x is absent from all intervals of $\text{refine}(\mathcal{I}_1, \mathcal{I}_2)$. The shift operation also uses $\text{refine}(\cdot,\cdot)$ and

Rules for atomic expressions:

$$\frac{}{c \rhd (\{X\}, \mathrm{d2R}(c), c)}\ \text{CONST} \qquad \frac{}{\chi \rhd (\{X\}, \mathrm{Id}_{\mathbb{R}}, \mathrm{Id}_{\{0,1\}^{64}})}\ \text{INPUT}$$

Rules for operators (with all constant arguments):

$$\frac{e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \mathsf{constant} \quad \forall I \in \mathcal{I}.\ \mathcal{B}'\restriction_{\widehat{I}} = \mathcal{E}(\mathsf{L}[\mathcal{B}\restriction_{\widehat{I}}]) \wedge \mathcal{A}'_{\vec{\delta}}\restriction_I = \mathrm{d2R}(\mathcal{B}'\restriction_{\widehat{I}})}{\mathsf{L}[e] \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{LOOKUP} \qquad \frac{e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \mathsf{constant} \quad \forall I \in \mathcal{I}.\ \mathcal{A}'_{\vec{\delta}}\restriction_I = \mathrm{d2R}(\mathcal{B}\restriction_{\widehat{I}})}{\mathsf{i2f}(e) \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B})}\ \text{I2F}$$

$$\frac{e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \mathsf{constant} \quad \forall I \in \mathcal{I}.\ \mathcal{B}'\restriction_{\widehat{I}} = \mathcal{E}(\mathsf{f2i}(\mathcal{B}\restriction_{\widehat{I}})) \wedge \mathcal{A}'_{\vec{\delta}}\restriction_I = \mathrm{d2R}(\mathcal{B}'\restriction_{\widehat{I}})}{\mathsf{f2i}(e) \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{F2I}_{\text{CONST}}$$

$$\frac{e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \mathsf{constant} \quad \forall I \in \mathcal{I}.\ \mathcal{B}'\restriction_{\widehat{I}} = \mathcal{E}(\mathsf{round}(\mathcal{B}\restriction_{\widehat{I}})) \wedge \mathcal{A}'_{\vec{\delta}}\restriction_I = \mathrm{d2R}(\mathcal{B}'\restriction_{\widehat{I}})}{\mathsf{round}(e) \rhd (\mathcal{I}, \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{RND}_{\text{CONST}}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \_, \mathcal{B}_1) \quad (\mathcal{I}_1, \mathcal{B}_1)\ \mathsf{constant} \\ e_2 \rhd (\mathcal{I}_2, \_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \mathsf{constant} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \mathcal{B}'\restriction_{\widehat{I}} = \mathcal{E}(\mathcal{B}_1\restriction_{\widehat{I}} * \mathcal{B}_2\restriction_{\widehat{I}}) \wedge \mathcal{A}'_{\vec{\delta}}\restriction_I = \mathrm{d2R}(\mathcal{B}'\restriction_{\widehat{I}}) \end{array}}{e_1 * e_2 \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{CONSTARG}$$

Rules for operators (with one or more non-constant argument(s)):

$$\frac{\mathsf{round}(e) \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B})\ \mathsf{defined}}{\mathsf{f2i}(e) \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \mathcal{B})}\ \text{F2I} \qquad \frac{e \rhd (\mathcal{I}, \mathcal{A}_{\vec{\delta}}, \_) \quad (\mathcal{I}, \mathcal{A}_{\vec{\delta}})\ \mathsf{defined} \quad \mathcal{I}' = \mathrm{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \quad \forall I_k \in \mathcal{I}'.\ \mathcal{A}'_{\vec{\delta}}\restriction_{I_k} = k \wedge \mathcal{B}'\restriction_{\widehat{I_k}} = \widehat{k}}{\mathsf{round}(e) \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \mathcal{B}')}\ \text{RND}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \_, \mathcal{B}_1) \quad (\mathcal{I}_1, \mathcal{B}_1)\ \mathsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \mathsf{defined} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in \widehat{I}.\ \mathcal{B}'\restriction_{\widehat{I}}(x) = \mathcal{B}_1\restriction_{\widehat{I}}(x) *_{\mathsf{b}} \mathcal{B}_2\restriction_{\widehat{I}}(x) \end{array}}{e_1 *_{\mathsf{b}} e_2 \rhd (\mathcal{I}', \bot, \mathcal{B}')}\ \text{BITOP}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \_, \mathcal{B}_1) \quad (\mathcal{I}_1, \mathcal{B}_1)\ \mathsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2)\ \mathsf{constant} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in \widehat{I}.\ \mathcal{B}'\restriction_{\widehat{I}}(x) = \mathcal{B}_1\restriction_{\widehat{I}}(x) *_{\mathsf{s}} \mathcal{B}_2\restriction_{\widehat{I}} \end{array}}{e_1 *_{\mathsf{s}} e_2 \rhd (\mathcal{I}', \bot, \mathcal{B}')}\ \text{SHIFT}$$

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}}, \_) \quad (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}})\ \mathsf{defined} \\ e_2 \rhd (\mathcal{I}_2, \mathcal{A}_{2,\vec{\delta}}, \_) \quad (\mathcal{I}_2, \mathcal{A}_{2,\vec{\delta}})\ \mathsf{defined} \\ \mathcal{I}' = \mathrm{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in I.\ \mathcal{A}'_{\vec{\delta}}\restriction_I(x) = (\mathcal{A}_{1,\vec{\delta}}\restriction_I(x) * \mathcal{A}_{2,\vec{\delta}}\restriction_I(x))(1+\delta'), \\ \text{where } \delta' \text{ is a fresh variable with a condition } |\delta'| < \varepsilon \end{array}}{e_1 *_{\mathsf{f}} e_2 \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \bot)}\ \text{FLOP}$$

Figure 3.4: The rules for constructing a symbolic abstraction.

Rule to produce a book-keeping map that is constant on each interval:

$$\frac{e \rhd (\mathcal{I}, \_, \mathcal{B}) \quad (\mathcal{I}, \mathcal{B}) \text{ defined} \quad \mathcal{I}' = \text{splitB}(\mathcal{I}, \mathcal{B}) \quad \forall I \in \mathcal{I}'.\ \mathcal{A}'_{\vec{\delta}}\restriction_I = \text{d2R}(\mathcal{B}\restriction_{\widehat{I}})}{e \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \mathcal{B})} \text{ SPLIT}$$

Rule for masking:

$$\frac{\begin{array}{c} e_1 \rhd (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}}, \_\_) \quad (\mathcal{I}_1, \mathcal{A}_{1,\vec{\delta}}) \text{ defined} \\ e_2 \rhd (\mathcal{I}_2, \_\_\_, \mathcal{B}_2) \quad (\mathcal{I}_2, \mathcal{B}_2) \text{ constant} \\ \exists n \in \mathbb{N}.\ \forall I \in \mathcal{I}_2.\ \mathcal{B}_2\restriction_{\widehat{I}} = 1^{12+n}0^{52-n}, \\ \mathcal{I}' = \text{refine}(\mathcal{I}_1, \mathcal{I}_2) \quad \forall I \in \mathcal{I}'.\ \forall x \in I.\ \mathcal{A}'_{\vec{\delta}}\restriction_I(x) = (\mathcal{A}_{1,\vec{\delta}}\restriction_I(x))(1+\delta'), \\ \text{where } \delta' \text{ is a fresh variable with a condition } |\delta'| < 2^{-n} \end{array}}{e_1 \text{ AND } e_2 \rhd (\mathcal{I}', \mathcal{A}'_{\vec{\delta}}, \bot)} \text{ BAND}$$

Figure 3.4: The rules for constructing a symbolic abstraction (continued).

requires the shift amount to be a constant for each interval (SHIFT). These are in contrast to the rule CONSTARG that requires both of the arguments of a binary operator to be constant for each interval.

The symbolic representation is undefined for bit-level operations (BITOP and SHIFT). A floating-point operation results in an undefined book-keeping map and the symbolic representation is updated by applying the corresponding operation over real numbers and introducing a new error term δ' (FLOP)[2].

Remark. The rule FLOP describes an abstraction step: we are over-approximating the result obtained from a floating-point operator by introducing the δ variables that model rounding errors. Floating-point operations can be composed in non-trivial ways to generate exactly rounded results [52], whereas in the symbolic representations that we use, the rounding errors only increase with the number of operations. For example, there are no rounding errors if a floating-point number is multiplied by a power of two. However, the rule FLOP introduces a new δ variable for this operation.

The rounding operation requires an auxiliary function $\text{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ (RND). This function splits the intervals in $\mathcal{I}$ further so that all doubles in each sub-interval round to the same integer:

$$\text{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) \triangleq \bigcup_{I \in \mathcal{I}} \Big\{ I_k \neq \emptyset : k \in \mathbb{Z} \Big\},$$

where each $I_k \subseteq I$ is an interval such that $\mathcal{A}(I_k) \subseteq (k-\frac{1}{2}, k+\frac{1}{2})$, and $\mathcal{A}(I) \triangleq [\min_{x \in I, \vec{\delta}} \mathcal{A}_{\vec{\delta}}, \max_{x \in I, \vec{\delta}} \mathcal{A}_{\vec{\delta}}]$. For instance, for $\mathcal{I} = \{[-3, 3]\}$ and $\mathcal{A}_{\vec{\delta}}(x) = (0.25 \times x)(1+\delta_1)$, $\text{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}}) = \{[-3, -\frac{2}{1-\varepsilon}], [-\frac{2}{1+\varepsilon}, \frac{2}{1+\varepsilon}], [\frac{2}{1-\varepsilon}, 3]\} = \mathcal{I}'$. The intervals created by $\text{splitA}(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ are not guaranteed to include all doubles that belong to the intervals of $\mathcal{I}$ (e.g., no interval of $\mathcal{I}'$ includes 2).

The rule SPLIT uses an auxiliary function $\text{splitB}(\mathcal{I}, \mathcal{B})$ to split the intervals in $\mathcal{I}$ further so that

[2] Modifying the rule FLOP from $\mathcal{A}' = (\mathcal{A}_1 * \mathcal{A}_2)(1+\delta')$ to $\mathcal{A}' = (\mathcal{A}_1 * \mathcal{A}_2)(1+\delta') + \delta''$ enables us to fully support subnormals, where δ'' is a fresh variable with $|\delta''| < 2^{-1075}$.

$\mathcal{B}$ is constant on each sub-interval. For an interval I, we define

$$M(I,\mathcal{B}) \triangleq \max\Big\{i : \forall x \in \{0,1\}^{64-i}.\ \exists n \in \{0,1\}^{64}.\ \forall y \in \{0,1\}^{i}.\ (x,y) \in \widehat{I} \Rightarrow \mathcal{B}((x,y)) = n\Big\}.$$

Intuitively, $M(I,\mathcal{B})$ is the maximum bit position i such that for each choice of bits $x[63],\cdots,x[i]$ we have $\mathcal{B}(x[63],\cdots,x[0]) = n$ for some constant n regardless of the choice of bits $x[i-1],\cdots,x[0]$. By using $M(I,\mathcal{B})$, we define splitB$(\mathcal{I},\mathcal{B})$ as

$$\text{splitB}(\mathcal{I},\mathcal{B}) \triangleq \bigcup_{I\in\mathcal{I}} \Big\{[\text{d2R}(l),\text{d2R}(r)] \cap I \neq \emptyset : x \in \{0,1\}^{64-i}, \\ \text{where } i = M(I,\mathcal{B}),\ l = (x,0^i),\ \text{and } r = (x,1^i)\Big\}.$$

Here, l and r are 64-bit vectors. For instance, for $\mathcal{B}(x) = x[63]01^90^{53}$, $I = [-3,3]$, and $\mathcal{I} = \{I\}$, we have $M(I,\mathcal{B}) = 63$ and splitB$(\mathcal{I},\mathcal{B}) = \{[-\infty,-0]\cap I, [+0,+\infty]\cap I\} = \{[-3,0],[0,3]\}$. Unlike refine$(\cdot,\cdot)$ and splitA$(\cdot,\cdot)$, the intervals created by splitB$(\mathcal{I},\mathcal{B})$ include all the doubles that belong to the intervals of $\mathcal{I}$. This rule is useful to create intervals over which expressions evaluate to constant values, which are required for the rules LOOKUP, I2F, F2I$_{\text{CONST}}$, RND$_{\text{CONST}}$, CONSTARG and SHIFT.

If we mask out the lower order bits of the significand using a bitwise-and operation (BAND) then its affect on the symbolic representation can be modeled by an error term using the following result:

Lemma 3.5. *Let $c \in \{0,1\}^{64}$ and $d = 1^{12+n}0^{52-n} \in \{0,1\}^{64}$ for some $n \in \mathbb{N}$. Then* d2R$(c$ AND $d) \in \{\text{d2R}(c)(1+\delta') : |\delta'| < 2^{-n}\}$.

This result bounds the difference between the masked output and the input using error terms.

Our main result follows by induction on the derivation tree:

Theorem 3.6. *If $e \rhd (\mathcal{I},\mathcal{A}_{\vec{\delta}},\mathcal{B})$, then $(\mathcal{I},\mathcal{A}_{\vec{\delta}},\mathcal{B})$ is consistent with e, and thus $(\mathcal{I},\mathcal{A}_{\vec{\delta}})$ is a symbolic abstraction of e.*

From the rules to construct symbolic abstractions, intervals in $\mathcal{I}$ cannot overlap with each other because each interval $I \in \mathcal{I}$ has the property that for all values in $\widehat{I}$, the value of some subexpression is guaranteed to be a constant, and because distinct intervals correspond to distinct constants. In constructing symbolic abstractions, the only step that requires manual intervention is the computation of splitA$(\cdot,\cdot)$, which could be automated for our benchmarks in §3.4.

Next, we use this symbolic abstraction to bound the absolute error or the ulp error of an implementation e from its mathematical specification.

3.3.4 Computation of Precision Loss

We describe a procedure to bound the precision loss of an implementation. Let e be an expression that implements a mathematical specification $f : \mathbb{R} \to \mathbb{R}$. The aim is to compute a bound Θ such

that on any input the outputs of e and f differ by at most Θ. More formally,

Definition 3.7. $\Theta_a \in \mathbb{R}$ is *a sound absolute error bound* for e and f over the interval I if for all $x \in \widehat{I}$,

$$\mathrm{ErrAbs}\Big(f(\mathrm{d2R}(x)), \mathcal{E}(e)(x))\Big) \leq \Theta_a.$$

We have a similar definition for ulps.

Definition 3.8. $\Theta_u \in \mathbb{R}$ is *a sound ulp error bound* for e and f over the interval I if for all $x \in \widehat{I}$,

$$\mathrm{ErrUlp}\Big(f(\mathrm{d2R}(x)), \mathcal{E}(e)(x)\Big) \leq \Theta_u.$$

We first present a procedure to compute Θ_a. Let $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ be a symbolic abstraction of e. We compute

$$\Theta_1 = \max\left\{\max_{x \in I, \vec{\delta}} \left|f(x) - \mathcal{A}_{\vec{\delta}}(x)\right| : I \in \mathcal{I}\right\}. \tag{3.3}$$

This computation is an analytical optimization problem that can be solved by computer algebra systems. The time and memory required to solve the optimization problem increases with the number of variables. Moreover, $\mathcal{A}_{\vec{\delta}}(x)$ has many variables, because a new δ variable is created for each floating-point operation (the rule FLOP of Figure 3.4). Hence off-the-shelf solvers fail to solve Eq. (3.3) as is.

For tractability, we simplify Eq. (3.3). In our evaluation, the symbolic representations are polynomials with small degrees and the inputs are restricted to small ranges (§3.4). In this setting, we can prove that the terms in $\mathcal{A}_{\vec{\delta}}(x)$ that involve a product of multiple δ variables are negligible compared to other values. Informally, if the coefficients and input ranges of a polynomial are bounded by a constant c then the rounding error introduced by all δ terms with degree > 1 is bounded by $C = \frac{(4c)^n \varepsilon}{1-\varepsilon}$. The proof uses standard numerical analysis techniques and is omitted.

With this simplification, the expression $\left|f(x) - \mathcal{A}_{\vec{\delta}}(x)\right|$ can be rearranged such that $|f(x) - \mathcal{A}_{\vec{\delta}}(x)| \leq |M_0(x)| + \sum_i |M_i(x)||\delta_i|$, where $M_0(x)$ is C plus all the terms of $f(x) - \mathcal{A}_{\vec{\delta}}(x)$ having no δ, and $M_i(x)$ is the coefficient of δ_i in $f(x) - \mathcal{A}_{\vec{\delta}}(x)$. We use optimization techniques to maximize $|M_i(x)|$ for each interval $I \in \mathcal{I}$ (which is tractable because $M_i(x)$ is a function of a single variable) and report

$$\max_{x \in I} |M_0(x)| + \sum_i \left(\max_{x \in I} |M_i(x)|\right)\left(\max_{\delta_i} |\delta_i|\right) \tag{3.4}$$

as a bound on the absolute error over the interval I. The number of total optimization tasks required to obtain Θ_1 (Eq. (3.3)) is proportional to the product of the number of δ variables and the number of intervals in $\mathcal{I}$. In our evaluation, we observe that the number of optimization tasks is tractable and the time taken by each task is less than a second (§3.4).

Recall that X represents the interval of interest and we may have $\left(\bigcup_{I \in \mathcal{I}} I\right) \neq X$ due to the operations used in computing symbolic abstractions (i.e., refine$(\cdot,\cdot)$ and splitA$(\cdot,\cdot)$). Therefore, it

is possible that $\Theta_a \neq \Theta_1$ if the input that results in the maximum absolute error belongs to the set $H = X \setminus \left(\bigcup_{I \in \mathcal{I}} I\right)$. So we compute

$$\Theta_2 = \max\left\{ |\mathrm{d2R}(\mathcal{E}(e)(x)) - f(\mathrm{d2R}(x))| : x \in \widehat{H} \right\}. \tag{3.5}$$

In our experiments, $|\widehat{H}|$ is small enough that it is feasible to compute Θ_2 by brute force testing, i.e., by evaluating e and f on every double in $\widehat{H}$. Note that it is not feasible to perform brute force on $\widehat{X}$ as it contains an intractable number of doubles and thus symbolic abstractions are necessary. A sound absolute error bound is then given by $\Theta_a = \max\{\Theta_1, \Theta_2\}$.

Next, we compute a sound ulp error bound Θ_u. It can be computed from either a bound on absolute error or a bound on relative error (Chapter 2). The latter is the maximum of two quantities: the maximum relative error observed during testing on inputs in $\widehat{H}$ and the result of the following optimization problem:

$$\Theta_r = \max\left\{ \max_{x \in I, \vec{\delta}} \left| \frac{f(x) - \mathcal{A}_{\vec{\delta}}(x)}{f(x)} \right| : I \in \mathcal{I} \right\}. \tag{3.6}$$

In our evaluation, we compute bounds on the ulp error using both the relative and the absolute error and report the better bound.

3.4 Case Studies

We evaluate our technique on Intel's implementations of three widely used transcendental functions: the sine function, the tangent function, and the natural logarithm function. We prove formal bounds on how much each implementation (`sin`, `tan`, and `log`) deviates from the corresponding mathematical specification ($\sin(\cdot)$, $\tan(\cdot)$, and $\log(\cdot)$). These implementations are available as (undocumented) x86 binaries included in `libimf`, which is Intel's implementation of the C numerics library `math.h`. The library contains many different implementations for these functions that have differing performance on different processors and input ranges. We choose the implementations used in the benchmark set of [137] and perform the verification for these implementations.

Next, we manually modify these binaries to create implementations that have better performance at the cost of less precise results (Table 3.1). Using our technique, we are also able to prove formal bounds on the precision loss of these variants. The results are summarized in Table 3.2 and Figure 3.5.

We use Mathematica to solve Eq. (3.4), though this is largely a matter of convenience. Mathematica provides functions to solve global optimization problems *analytically*, namely the `MaxValue`[·] and the `MinValue`[·] functions which find *exact* global optima of a given optimization problem (using several different algorithms, e.g., cylindrical algebraic decompositions).[3] The analytical optimization that we use is in contrast to typical *numerical* optimization that uses finite-precision arithmetic

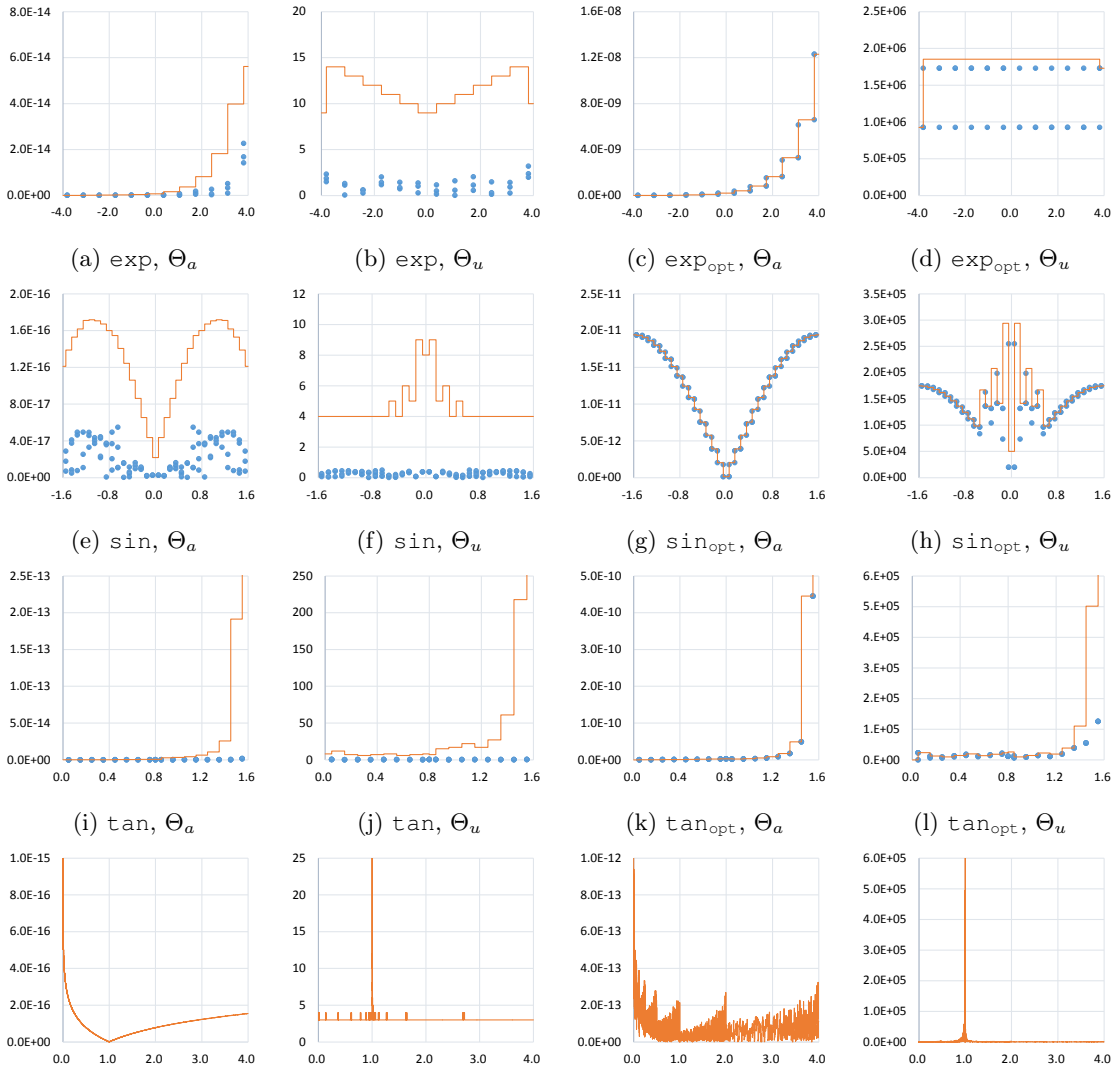
[3] https://reference.wolfram.com/language/tutorial/ConstrainedOptimizationExact.html (titled "Exact Global Optimization"; accessed on July 2023)

f	LoC of `f`	LoC of $\texttt{f}_{\texttt{opt}}$	Reduction in size	Speedup
exp	38	28	10	1.6×
sin	66	42	24	1.5×
tan	107	89	18	1.1×
log	67	54	13	1.3×

Table 3.1: Important statistics of each implementation for case studies (LoC is lines of x86 assembly). The manually/automatically created variants have smaller number of instructions (column 4) and are faster (column 5) than their production counterparts.

	Interval	Θ_a	Θ_u	$\lvert\widehat{H}\rvert$	$\lvert\vec{\delta}\rvert$	$\lvert\mathcal{I}\rvert$
`exp`	$[-4, 4]$	6×10^{-14}	14	36	29	13
$\texttt{exp}_{\texttt{opt}}$	$[-4, 4]$	1×10^{-8}	2×10^{6}	36	19	13
`sin`	$\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	2×10^{-16}	9	110	53	33
$\texttt{sin}_{\texttt{opt}}$	$\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	2×10^{-11}	3×10^{5}	110	26	33
`tan`	$\left[0, \frac{17\pi}{64}\right)$	7×10^{-16}	12	89	84	10
	$\left[\frac{17\pi}{64}, \frac{31\pi}{64}\right)^{\dagger}$	2×10^{-13}	218	89	85	7
	$\left[\frac{31\pi}{64}, \frac{\pi}{2}\right)^{\dagger}$	2×10^{15}	9×10^{18}	89	85	1
$\texttt{tan}_{\texttt{opt}}$	$\left[0, \frac{17\pi}{64}\right)$	3×10^{-12}	3×10^{4}	89	69	10
	$\left[\frac{17\pi}{64}, \frac{31\pi}{64}\right)^{\dagger}$	4×10^{-10}	5×10^{5}	89	69	7
	$\left[\frac{31\pi}{64}, \frac{\pi}{2}\right)^{\dagger}$	2×10^{16}	9×10^{18}	89	69	1
`log`	$(0, 4) \setminus \left[\frac{4095}{4096}, 1\right)$	8×10^{-14}	21	0	25	2^{21}
	$\left[\frac{4095}{4096}, 1\right)$	9×10^{-19}	1×10^{14}	0	25	1
$\texttt{log}_{\texttt{opt}}$	$(0, 4) \setminus \left[\frac{4095}{4096}, 1\right)$	6×10^{-11}	5×10^{5}	0	12	2^{21}
	$\left[\frac{4095}{4096}, 1\right)$	1×10^{-18}	1×10^{14}	0	12	1

Table 3.2: Summary of results: For each implementation (column 1), for all inputs in the interval (column 2), Θ_a shows the bound on maximum absolute error and Θ_u shows the bound on maximum ulp error between the implementation and the exact mathematical result. The number of inputs that require testing ($\lvert\widehat{H}\rvert$), the number of δ variables ($\lvert\vec{\delta}\rvert$), and the number of intervals considered ($\lvert\mathcal{I}\rvert$) in the symbolic abstraction are also shown. The values of Θ_a and Θ_u on the rows with † need not to be sound.



(a) exp, Θ_a (b) exp, Θ_u (c) $\texttt{exp}_{\texttt{opt}}$, Θ_a (d) $\texttt{exp}_{\texttt{opt}}$, Θ_u

(e) sin, Θ_a (f) sin, Θ_u (g) $\texttt{sin}_{\texttt{opt}}$, Θ_a (h) $\texttt{sin}_{\texttt{opt}}$, Θ_u

(i) tan, Θ_a (j) tan, Θ_u (k) $\texttt{tan}_{\texttt{opt}}$, Θ_a (l) $\texttt{tan}_{\texttt{opt}}$, Θ_u

(m) log, Θ_a (n) log, Θ_u (o) $\texttt{log}_{\texttt{opt}}$, Θ_a (p) $\texttt{log}_{\texttt{opt}}$, Θ_u

Figure 3.5: Each graph shows a bound on precision loss between an implementation and the mathematically exact result. For example, (a) plots a bound on absolute error between $\texttt{exp}(x)$ and e^x as a function of x. The brown lines represent the bounds obtained from symbolic abstractions and the blue dots signify the errors observed during explicit testing on inputs in $\widehat{H}$.

without providing formal guarantees. Other techniques, such as interval arithmetic or branch and bound optimization [143], can be used (instead of Mathematica) to solve Eq. (3.4) soundly.

To compute Θ_2 in Eq. (3.5), for each $x \in \widehat{H}$, we compare the floating-point result $\mathcal{E}(e)(x)$ computed by an evaluation of e on x, with the exact result $f(\mathrm{d2R}(x))$ computed by Mathematica.

Even though Mathematica claims soundness guarantees, it does not produce a certificate of correctness with the solution. In the future we would like to replace Mathematica with a solver that produces machine-checkable certificates.

3.4.1 The `sin` Implementation

Intel's `sin` implementation uses the following procedure to compute $\sin(x)$. It first computes an integer N and a reduced value d:

$$N = \mathrm{round}\left(\frac{32}{\pi}x\right),\ d = x - \frac{\pi}{32}N. \tag{3.7}$$

Then it uses the following trigonometric identity:

$$\sin(x) = \sin(d)\cos\left(\frac{\pi}{32}N\right) + \cos(d)\sin\left(\frac{\pi}{32}N\right).$$

The terms $\sin(d)$ and $\cos(d)$ are computed by a Taylor series expansion:

$$\sin(d) \approx d - \frac{d^3}{3!} + \frac{d^5}{5!} - \frac{d^7}{7!} + \frac{d^9}{9!}, \qquad \cos(d) \approx 1 - \frac{d^2}{2!} + \frac{d^4}{4!} - \frac{d^6}{6!} + \frac{d^8}{8!}.$$

The Taylor series expansion includes an infinite number of terms. However, since d is small in magnitude, a small number of Taylor series terms shown above are sufficient to provide a good approximation of $\sin(d)$ and $\cos(d)$. The remaining terms, $\sin\left(\frac{\pi}{32}N\right)$ and $\cos\left(\frac{\pi}{32}N\right)$, are obtained by table lookups. A table in memory stores precomputed values $\sin\left(\frac{\pi}{32}i\right)$ and $\cos\left(\frac{\pi}{32}i\right)$ for each $i = 0, \cdots, 63$ and the index $i = (N)$ AND `0x3f` is used to retrieve the correct value. The `sin` implementation uses bit-level operations to compute the index i and the final memory address for the lookup.

We modify `sin` to obtain a more efficient but less precise implementation $\mathtt{sin}_{\mathtt{opt}}$. These modifications include removing all subcomputations that have only a small effect on the final output. In particular, $\mathtt{sin}_{\mathtt{opt}}$ uses a Taylor series expansion of degree 5 (instead of 9). The *compensation* terms are also omitted. These are terms such as $(c_1 -_{\mathsf{f}} (c_1 +_{\mathsf{f}} c_2)) +_{\mathsf{f}} c_2$ that are 0 in the absence of rounding errors but are important for maintaining accuracy of floating-point computations (see the remark in §3.3.3). Moreover $\mathtt{sin}_{\mathtt{opt}}$ replaces some very small constants (e.g., 7.9×10^{-18}) by 0.

For $X = [-\frac{\pi}{2}, \frac{\pi}{2}]$, we compute a symbolic abstraction $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ of `sin` by applying the technique described in §3.3.3. In the final abstraction $|\mathcal{I}| = 33$ and there are 53 δ's in $\mathcal{A}_{\vec{\delta}}$ (Table 3.2). The main step in the construction of a symbolic abstraction for `sin` (as well as $\mathtt{sin}_{\mathtt{opt}}$) is the application of the rule RND so that each of the resulting intervals contains inputs that map to the same N

(Eq. (3.7)). A total of $|\widehat{H}| = 110$ inputs do not belong to any interval of $\mathcal{I}$, which is easily a small enough set that we can compute Θ_2 (Eq. (3.5)) via testing on each of these inputs.

We then use the procedure described in §3.3.4 to compute Θ_a and Θ_u over the interval X for `sin` and $\texttt{sin}_{\texttt{opt}}$ (Table 3.2 and Figure 3.5). Our main result for `sin` is a proof that for all inputs in $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, the computed result differs from $\sin(x)$ only by 9 ulps, that is, there are at most 9 doubles between the computed result and the mathematically exact result. For $\texttt{sin}_{\texttt{opt}}$, we have successfully traded a small loss in precision (bearable for many applications) for over 50% improvement in performance (Table 3.1).

3.4.2 The `tan` Implementation

Intel's `tan` implementation uses the following procedure to compute $\tan(x)$. To focus on the distinctive parts of the implementation, we assume that the input $x \in X = \left[0, \frac{\pi}{2}\right)$. The first step is to compute an integer N and a reduced value d:

$$N = \left\lfloor \frac{32}{\pi} x + \frac{1}{2} \right\rfloor, \; d = x - \frac{\pi}{32} N.$$

Then we compute

$$\tan(x) \approx bR(x) + T_{15}(d), \; R(x) = \frac{1}{\frac{\pi}{2} - x} + \frac{c}{(\frac{\pi}{2} - x)^2}.$$

Here, $b = 0$ if $x \in \left[0, \frac{17\pi}{64}\right)$ and $b = 1$ if $x \in \left[\frac{17\pi}{64}, \frac{\pi}{2}\right)$, $c = 8.84372 \cdots \times 10^{-29}$, and $T_{15}(d) = \sum_{i=0}^{15} q_i d^i$ is a polynomial approximation of $\tan(x) - bR(x)$ of degree 15 where the polynomial coefficients depend on N. The coefficients $q_0, \cdots, q_{15}$ of $T_{15}(\cdot)$ are retrieved from a lookup table based on the index N. Note that, unlike `exp` and `sin`, `tan` includes rational terms $1/\left(\frac{\pi}{2} - x\right)$ and $1/\left(\frac{\pi}{2} - x\right)^2$ in order to minimize the precision loss near $x = \frac{\pi}{2}$.

The hand-optimized implementation $\texttt{tan}_{\texttt{opt}}$ uses a polynomial approximation of degree 12 (instead of 15) and omits an AND operation used by `tan` to mask out lower order bits of the significand of an intermediate result. We omit the compensation terms and obtain a total speed up of 1.1× (Table 3.1).

To construct a symbolic abstraction of `tan` (and $\texttt{tan}_{\texttt{opt}}$), we apply the rules RND, SPLIT, and BAND. Again, we use Mathematica to compute an absolute error bound. However, the analytical optimization routines of Mathematica time out while computing Θ_1 in the presence of rational terms (i.e., if $x \geq \frac{17\pi}{64}$). Therefore, we use numerical optimization routines `NMaxValue`[·] and `NMinValue`[·] for the intervals that are subsets of $\left[\frac{17\pi}{64}, \frac{\pi}{2}\right)$. Since numerical optimization routines have no correctness guarantees, the computed error bounds for $x \geq \frac{17\pi}{64}$ need not to be sound. However, soundness is still maintained for $x < \frac{17\pi}{64}$ as these intervals are optimized analytically (using `MaxValue` and `MinValue`).

In Table 3.2, we observe that the ulp error over $\left[0, \frac{17\pi}{64}\right)$ is small (12 ulps), and the ulp error over $\left[\frac{17\pi}{64}, \frac{31\pi}{64}\right)$ is slightly higher (218 ulps). For the interval $J = \left[\frac{31\pi}{64}, \frac{\pi}{2}\right)$, we do not obtain good bounds. The absolute error is large on the interval J as the rational terms grow quickly near $\frac{\pi}{2}$. The relative

error is large on J as one of the optimization task is the following:

$$\max_{x \in J} \left| \frac{1}{(\frac{\pi}{2} - x)^2 \tan(x)} \right|.$$

For x close to $\frac{\pi}{2}$, the optimization objective is unbounded so the obtained bounds on relative error are large. Because neither the absolute error nor the relative error provides good bounds on ulp error, the bounds on ulp error are large for $x \in J$. The results for $\texttt{tan}_{\texttt{opt}}$ are similar (Table 3.2 and Figure 3.5).

3.4.3 The `log` Implementation

Intel's `log` implementation uses the following procedure to compute $\log(x)$ for $x > 0$. Let us denote the exponent of x by p and $f = f_1, f_2, \ldots, f_{52}$ denotes the bits of the significand of x. The implementation first constructs a single-precision floating-point number $g = 1.f_1 \ldots f_{23}$. Next, the result $\frac{1}{g}$ obtained from a single-precision reciprocal operation is converted to a double d'. Using $x = 1.f \times 2^p$, we have the following identity:

$$\begin{aligned} \log(x) &\approx \log(2^p) + \log(g) + \log(d' \times 1.f) \\ &\approx p \log(2) + \log\left(\frac{256}{i + 128}\right) + \log(1 + d), \end{aligned}$$

where $i = \text{round}(256d' - 128)$ and $d = d' \times 1.f - 1$. The quantity p is computed exactly by bit-level operations that extract the exponent of x. The quantity $\log\left(\frac{256}{i+128}\right)$ is computed by table lookups based on the index i. The lookup table stores the value $\log\left(\frac{256}{i+128}\right)$ for $i = 0, \cdots, 128$. Finally, since d is small in magnitude, $\log(1 + d)$ is computed using a Taylor series of degree 7. The `log` implementation uses bit-level operations to compute p, g, and d'.

We hand-optimize `log` to create an implementation $\texttt{log}_{\texttt{opt}}$ that uses a Taylor series of degree 4 (instead of 7) and ignores some bit-manipulations of the significand. We also remove the compensation terms and obtain a total speedup of 1.3× (Table 3.1).

To construct a symbolic abstraction of `log` (as well as $\texttt{log}_{\texttt{opt}}$), we apply the rule SPLIT to ensure that, for each interval, d' is a constant. Using $X = (0, 4)$, we obtain a symbolic abstraction $(\mathcal{I}, \mathcal{A}_{\vec{\delta}})$ of `log` with $|\mathcal{I}| = 2^{21}$, $|\widehat{H}| = 0$, and 25 δ variables (Table 3.2). Unlike the other benchmarks, $|\widehat{H}| = 0$ as splitA$(\cdot, \cdot)$ is not used in constructing the symbolic abstraction. Since the number of intervals, $|\mathcal{I}|$, is large, we run 32 different instances of Mathematica in parallel. The optimizations are performed analytically and the obtained error bounds are sound. Moreover, each optimization task takes less than a second and the total wall clock time for `log` (resp. $\texttt{log}_{\texttt{opt}}$) was 16 hours (resp. 5 hours).

In our opinion, the analysis of the log function best illustrates the power of our technique: We are able to automatically reduce a very complex problem to millions of tractable subproblems, and the total time required compares favorably with the only alternative available today, which is an

expert using an interactive theorem prover to construct the proof.

We present the most interesting results. Except for the interval $J = \left[\frac{4095}{4096}, 1\right)$, the ulp error is small: 21 ulps (Table 3.2). For the interval J, the absolute error is very small: 9×10^{-19}. This small absolute error is expected as $\log(x)$ is close to zero on this interval ($\log(1) = 0$). However, due to the proximity to zero, even this small absolute error leads to a large ulp error. For `sin` and `tan`, we are able to get good bounds on ulp error near zero by using bounds on relative error. However, the relative error of `log` is large on the interval J as one of the optimization tasks is the following:

$$\max_{x \in J} \left| \frac{x - \frac{255}{256}}{\log x} \right|.$$

For x close to one, the optimization objective is unbounded and the obtained bounds on relative error are large. Therefore, the bounds on ulp error are large for $x \in J$. The results for $\texttt{log}_{\texttt{opt}}$ are similar (Table 3.2 and Figure 3.5).

3.5 Related Work

Due to their mathematical sophistication and practical importance, transcendental functions are used as benchmarks in many verification studies. However, prior studies have either focused on the verification of algorithms (and not implementations) or the verification of comparatively simple implementations that contain no bit-level operations.

In the absence of bit-level operations, a variety of techniques can be used to bound rounding errors: Gappa uses interval arithmetic [39], Fluctuat uses affine arithmetic [43, 54], MathSAT is an SMT solver that uses interval arithmetic for floating-point [60], and Rosa combines affine arithmetic with SMT solvers [35, 36]. Interval arithmetic does not preserve dependencies between variables and affine arithmetic fits poorly with non-linearities. Hence, these approaches lead to imprecise results (see the evaluation in [143]). To address this issue, other techniques have been used as well to bound rounding errors: FPTaylor uses optimization [143, 144], Real2Float uses semidefinite programming [99], and Satire uses automatic differentiation [37].

The problem that our technique solves is slightly different from the problems that previous methods do. FPTaylor bounds the difference between interpretations of an expression over the reals and over the floating-point numbers. Formally speaking, given a function $g : \mathbb{R} \to \mathbb{R}$, FPTaylor computes a bound on $|\mathrm{fp}(g)(x) - g(x)|$, where $\mathrm{fp}(g)$ is a function (from floating-point numbers to floating-point numbers) obtained from g by replacing all real-valued operations with the corresponding floating-point operations. In contrast, our technique bounds the difference between the polynomials obtained from symbolic abstractions of binaries and the true mathematical transcendentals. Due to the relationship between $\mathrm{fp}(g)$ and g, it is impossible to have that $\mathrm{fp}(g)$ is a polynomial *and* g is a transcendental, so FPTaylor and our technique solve different problems.

Similarly, Gappa, Fluctuat, and Rosa also aim to bound the difference between two interpretations of the same expression: over the reals and over the floating-point numbers. Moreover, they do not encode transcendentals while our method requires encoding transcendentals.

Commercial tools such as Fluctuat and Astree [15, 106] provide some support for mixed floating-point and bit-level code. These tools use abstract domains tailored to specific code patterns and reason soundly about low-level C implementations. In contrast, our approach is general and systematic. Fluctuat supports subdivision of input ranges, but its subdivision strategy is generic and requires no static analysis (e.g., repeatedly halving an input interval until the desired error bound on each subdivision is reached). The paper [95] subdivides according to the exponent bits to improve precision. This chapter provides a general algorithm to construct these subdivisions; the need for this automatic interval construction to be sound leads to a distinctive characteristic of our subdivisions, the presence of floating-point numbers that are not covered by any interval.

Intel's processors contain instructions (e.g., `fsin`, `fcos`, etc.) that evaluate transcendental functions. The manual[4] for Pentium processors stated that a "rigorous mathematical analysis" had been performed and that the "worst case error is less than one ulp" for the output of these instructions. These claims were discovered to be incorrect and the instructions actually have large errors even for small arguments [45]. This incident indicates that obtaining accurate implementations of transcendental functions is non-trivial. Intel's latest manual[5] acknowledges the mistake ("the ulp error will grow above these thresholds") and recommends developers use the software implementations in Intel's Math Library. We are interested in verification of these implementations. A description of implementations that are similar to the ones that we verify can be found in [63].

Harrison describes machine-checkable proofs of algorithms that compute 64-bit $\sin(\cdot)$ [62] and 32-bit $\exp(\cdot)$ [61]. The verified algorithms are fairly close to the one employed by the implementations described in §3.4. Harrison's main result is a proof in HOL Light that the algorithms compute results within 0.57 ulps of the mathematically exact result. The proof requires an extensive mathematical apparatus that seems necessary for such a precise bound. Harrison reports that the manual effort required for each such proof can vary from weeks to months [61].

In contrast, we have a general verification technique that can be readily applied to a variety of functions and their automatically or manually produced variants. This generality and better automation is achieved at the expense of analysis precision and the bounds we infer, while sharp enough to be useful, are loose compared to Harrison's manual proofs. Moreover, we have evaluated our technique only on small input ranges, whereas Harrison proves bounds for large ranges.

Techniques that provide statistical (as opposed to formal) guarantees include [107, 111, 137]. Analyses described in [6, 11, 25, 87] do not have any formal or statistical guarantees.

[4] Appendix G of "Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual" (available at `http://datasheets.chipdb.org/Intel/x86/Pentium/24143004.PDF`; accessed on July 2023)

[5] Section 8.3.10 of "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture" (available at `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`; accessed on July 2023).

3.6 Discussion

Our technique is directly applicable to trading precision for performance. Most programs do not require all bits of precision and imprecise implementations can have better performance [131]. We believe that our verification technique significantly lowers the barrier to entry for developers who want to explore such trade-offs but are afraid of the subtleties associated with floating-point. The developers can create imprecise but efficient implementations either manually or by using tools such as [131, 137] and prove their correctness formally using our technique. Conversely, our technique can also be used to formally prove the correctness of transformations that improve precision [115]. These transformations are currently validated via sampling.

While we believe our approach is promising, there are important limitations that would be desirable to improve or remove altogether. Ideally, we would like to achieve bounds within 1 ulp. We have taken a step in this direction by demonstrating the first technique that can prove sound bounds on the ulp error for these implementations, which in some cases, are close to the desired result (e.g., 9 ulps for `sin`). However, automatically proving that these routines are accurate to within 1 ulp requires further advances in verification techniques for floating-point that can address the imprecision introduced by the abstractions for rounding errors (see the remark in §3.3.3). Next, the inferred bounds are sound only for small ranges of inputs. Removing this restriction introduces new challenges: if the inputs belong to a large range then the intermediate values could include NaNs or infinities. Moreover, the simplifications of the optimization problem described in §3.3.4 would no longer be sound. Orthogonally, our technique cannot handle all bit-level tricks. For example, a good approximation to the inverse square root of a single-precision floating-point number x is given by $\texttt{0x5f3759df} - (x \gg 1)$. During verification of this approximation using our technique, the rule SPLIT of Figure 3.4 creates an intractable number of intervals. Therefore, this task requires a different technique (e.g., exhaustive enumeration).

In general, the number of intervals can grow quickly with the number of arguments on which bit-level operations are performed. However, the functions in `math.h` have at most three arguments. Therefore, this situation does not arise in our particular application. We expect the number of intervals to be tractable even for multivariate implementations. Finally, since the optimization problems are independent, parallelization can improve scalability significantly (§3.4.3).

3.7 Conclusion

In this chapter, we present a systematic technique for verifying the behavior of mixed binaries (i.e., those mixing floating-point and bit-level operations), which combines abstraction, analytical optimization, and testing. We demonstrate that our technique is directly applicable to Intel's highly optimized implementations of transcendental functions and it proves formal error bounds of these widely used routines.

Chapter 4

Correctness of Highly Accurate Math Libraries

4.1 Introduction

Industry standard math libraries, such as Intel's implementation of `math.h`, have very strict correctness requirements. In particular, Intel guarantees that the maximum *precision loss*, i.e., the difference between the computed floating-point value and the actual mathematical result, is very small. However, to the best of our knowledge, this claim is not backed by formal proofs. Establishing the correctness of these implementations is non-trivial: the error bounds are tight (see below), floating-point operations have rounding errors that are non-trivial to reason about, and these high performance libraries are full of undocumented code optimization tricks. We describe a novel automatic verification technique capable of establishing the correctness of these implementations.

For example, consider the `sin` function of `math.h`. Since it is a transcendental, for most floating-point inputs, $sin\, x$ is an irrational number inexpressible as a 64-bit double-precision floating-point number. Most standard libraries guarantee that the maximum precision loss is strictly below one ulp, i.e., if the exact mathematical result $sin\, x$ is in the interval (d_1, d_2) where d_1 and d_2 are two consecutive floating-point numbers, then the computed result is generally either d_1 or d_2. This requirement is difficult to meet because of floating-point rounding errors. Consider, for example, what happens if we implement x^4 using $x \otimes x \otimes x \otimes x$. For some inputs the precision loss of this implementation is more than one ulp.

An algorithm for computing $sin\, x$ was verified to be correct, i.e., meeting the one ulp bound, for any $x \in [-2^{63}, 2^{63}]$ by Harrison using the proof assistant HOL Light [62]. Constructing such machine-checkable proofs requires a Herculean effort and Harrison remarks that each proof can take weeks to months of manual effort [61]. In Chapter 3 (which was published as [90]), we proved an

error bound of 9 ulps for Intel's `sin` implementation over the input interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ automatically, i.e., in most cases there can be at most 9 floating-point numbers between the computed and the mathematically exact result for any input in $[-\frac{\pi}{2}, \frac{\pi}{2}]$. In this chapter, we focus on automatically proving much tighter error bounds. We describe an analysis that is fully automatic and, for example, proves that the maximum precision loss of `sin` is below one ulp over the input interval $[-\pi, \pi]$.

The main source of imprecision in Chapter 3 stems from modeling every floating-point operation as having a rounding error about which worst-case assumptions must be made. However, floating-point operations do not always introduce rounding errors. In particular, there are several *exactness* results that describe conditions under which floating-point operations are exact, i.e., the floating-point operation gives the mathematical result. For example, although $2^{100} \ominus 1 = 2^{100}$ (due to rounding errors), $0.5 \ominus 0.25$ is exactly equal to 0.25 in floating-point arithmetic. An important example of such an exactness result is Sterbenz's theorem [146], which says that when two floating-point numbers are close to each other then their subtraction is exact. Our approach to improving the provable error bounds is to identify floating-point computations that are exact and thus avoid introducing unneeded potential rounding errors into the modeling of those computations. Our main technical contribution is in reducing the problem of checking whether an exactness result applies to a set of mathematical optimization problems that can be solved soundly and automatically by off-the-shelf computer algebra systems. For example, our analysis checks the closeness conditions in Sterbenz's theorem by solving four optimization problems.

We apply this analysis to the benchmarks of [90, 137], i.e., Intel's `sin`, `tan`, and `log`. For `log`, we prove that for all valid inputs the precision loss is below one ulp. We are not aware of any other formal correctness proof of this implementation. Previous to this work, the best known provable error bound for `log` was 10^{14} ulps presented in Chapter 3 (i.e., [90]), which says the implementation is provably correct only up to two decimal digits. We note that our proof is computationally intensive and took more than two weeks of computation time on 16 cores (but is also highly parallelizable). Next, we prove the correctness of `sin` for inputs between $-\pi$ and π. Recall that $sin\, x$ is periodic and our results can be extended to all valid inputs at the expense of more computational resources. For `tan`, we proved correctness only for a part of the input interval. In particular, our abstractions lose precision and the inferred bounds, though sound, are imprecise for inputs near $\frac{\pi}{2}$ (§4.6). The previously known bounds for `tan`, such as those presented in Chapter 3 (i.e., [90]), were loose (up to several orders of magnitude greater than the bounds we prove) and sometimes not guaranteed to be sound.

Our main contributions are as follows:

- We show a novel automatic analysis that systematically uses exactness results about floating-point arithmetic. In particular, the analysis verifies that the result of a floating-point operation is exact by solving several mathematical optimization problems soundly.

- We describe the first technique that automatically proves the correctness of transcendental functions in industry standard math libraries. Prior to this work, these implementations could

only be verified with significant manual effort.

- We present the properties of floating-point used in these proofs. Some of these properties are only well-known to floating-point experts, and others are new in the sense that they have not been stated explicitly in the literature.

The rest of this chapter is organized as follows. §4.2 motivates our analysis using an example. §4.3 and §4.4 present the two major components of our method: An abstraction of floating-point is described in §4.3 and proven to be sound; the analysis is described in §4.4. The analysis uses some well-known results (§4.4.1, 4.4.2, 4.4.3, 4.4.6) and other results about floating-point that we have proven (§4.4.4, 4.4.5) and found useful. §4.5 mentions some interesting implementation details and §4.6 evaluates the analysis on a number of functions from `math.h`. Finally, §4.7 discusses related work and §4.8 concludes.

4.2 Motivation

We discuss an example on which standard analyses produce very imprecise bounds and show how the precision can be recovered by applying exactness results. Consider Intel's `log` implementation of the natural logarithm function over the input interval $X = [4095/4096, 1)$. The complete `log` implementation is quite complicated but if we restrict the inputs to the small interval X, it can be significantly simplified. For an input $x \in X$, `log` first computes the following quantity:

$$r(x) = \left(\left((2 \otimes x) \ominus \frac{255}{128}\right) \otimes \frac{1}{2}\right) \oplus \left(\left(\frac{255}{128} \otimes \frac{1}{2}\right) \ominus 1\right) \tag{4.1}$$

where $\circledast$ denotes the floating-point operation corresponding to the real-valued operation $* \in \{+, -, \times, /\}$. Then `log` returns $v(x) = v_3 \oplus v_2 \oplus v_5 \oplus v_4 \oplus v_1$, where $v_1, \cdots, v_5$ are computed as:

$$\begin{aligned}
v_1 &= (d_1 \otimes n) \oplus t_1 \oplus r, \\
v_2 &= (d_1 \otimes n) \oplus t_1 \ominus v_1 \oplus r, \\
v_3 &= (d_2 \otimes n) \oplus t_2, \\
v_4 &= [c_2 \oplus (c_3 \otimes r) \oplus (c_4 \otimes (r \otimes r))] \otimes (r \otimes r), \\
v_5 &= [((c_5 \oplus (c_6 \otimes r)) \otimes r) \oplus (c_7 \otimes r \otimes (r \otimes r))] \otimes ((r \otimes r) \otimes (r \otimes r)).
\end{aligned}$$

Here every floating-point operation is assumed to be left-associative, r denotes $r(x)$, and the floating-point constants c_i, d_i, t_i, and n are $c_i \approx (-1)^{i+1}/i$ $(i = 2, \cdots, 7)$, $d_1 \approx (log\, 2)/16$, $d_2 \approx (log\, 2)/16 - d_1$, $t_1 \approx log\, 2$, $t_2 \approx log\, 2 - t_1$, and $n = -16$, where $log\, x$ is the natural logarithm function.

A standard technique to automatically bound the maximum precision loss of such a floating-point implementation is the well-known $(1 + \varepsilon)$-property (Chapter 2). The property states that for any

mathematical operator $* \in \{+, -, \times, /\}$, the result of a floating-point operation $a \circledast b$ is $(a * b)(1 + \delta)$ for some $|\delta| < 2^{-53}$. By applying the $(1 + \varepsilon)$-property to each floating-point operation of $r(x)$, we obtain the following abstraction $\mathcal{A}(x)$ of $r(x)$:

$$\mathcal{A}(x) \triangleq \left[\left(\left((2x)(1+\delta_0) - \frac{255}{128}\right)(1+\delta_1) \times \frac{1}{2}\right)(1+\delta_2) + \left(\left(\frac{255}{128} \times \frac{1}{2}\right)(1+\delta_3) - 1\right)(1+\delta_4)\right](1+\delta_5)$$
$$= (x-1) + \left(x - \frac{255}{256}\right)\delta_1 + \cdots \tag{4.2}$$

where each δ_i ranges over $(-2^{-53}, 2^{-53})$. We call $\mathcal{A}(x)$ an abstraction as it over-approximates $r(x)$, i.e., $\forall x \in X. \exists \delta_0, \cdots, \delta_5.\ r(x) = \mathcal{A}(x)$. Observe that the rounding errors accumulate with each floating-point operation, and the maximum precision loss of the final result $v(x)$ (a polynomial in $r(x)$) is at least as large as the maximum precision loss of $r(x)$. Using the abstraction $\mathcal{A}(x)$ of $r(x)$, the maximum relative error of $r(x)$ is bounded by:

$$\max_{x \in X, |\delta_i| < 2^{-53}} \left| \frac{\mathcal{A}(x) - (x-1)}{x-1} \right|.$$

Because of the term $(x - \frac{255}{256})\delta_1$ in the abstraction $\mathcal{A}(x)$ of $r(x)$, this error is at least

$$\max_{x \in X} \left| \frac{x - \frac{255}{256}}{x-1} \right| \varepsilon. \tag{4.3}$$

Unfortunately the objective function in Eq. (4.3) is unbounded for $x \in X$, and thus, using this analysis, we are unable to bound the maximum relative error of the result.

A more precise analysis can bound the maximum relative error of $r(x)$. The key insight is that some floating-point operations in Eq. (4.1) are exact and do not introduce any rounding errors. In particular, the subtraction operations in Eq. (4.1) are exact according to Sterbenz's theorem: $a \ominus b$ is exact whenever a is within a factor of 2 of b (§4.4.2). Here, $x \in [4095/4096, 1)$ and hence $\frac{1}{2} \cdot \frac{255}{128} \leq 2x \leq 2 \cdot \frac{255}{128}$ holds. Moreover, multiplication and division by 2 are also exact (§4.4.1). Using this information, we can construct a more precise abstraction of $r(x)$. In particular, for an exact operation $a \circledast b$, we have $a \circledast b = a * b$ and we do not need to introduce δ variables. Since all the operations except $\oplus$ in Eq. (4.1) are exact, we have $r(x) = \left(\left(2x - \frac{255}{128}\right) \times \frac{1}{2}\right) \oplus \left(\left(\frac{255}{128} \times \frac{1}{2}\right) - 1\right)$. Therefore, by applying the $(1 + \varepsilon)$-property only once to the operation $\oplus$, we obtain the following abstraction $\mathcal{A}'(x)$ of $r(x)$ which is more precise than Eq. (4.2):

$$\mathcal{A}'(x) \triangleq \left[\left(\left(2x - \frac{255}{128}\right) \times \frac{1}{2}\right) + \left(\left(\frac{255}{128} \times \frac{1}{2}\right) - 1\right)\right](1+\delta') = (x-1) + (x-1)\delta'$$

where δ' ranges over $(-2^{-53}, 2^{-53})$. We use this more precise abstraction to find a better bound on

1: Let $x = 2^p \times 1.g_2 \cdots g_{53\,(2)}$.
2: Compute $s = 1.g_2 \cdots g_{53\,(2)}$ and $s' = 1.g_2 \cdots g_8 0 \cdots 0\,_{(2)}$.
3: Compute $s_{inv} = 2^q \times 1.h_2 \cdots h_8 0 \cdots 0\,_{(2)}$ such that $s_{inv} \approx 1/s$.
4: Compute $r(x) = (s \ominus s') \otimes s_{inv} \oplus (s' \otimes s_{inv} \ominus 1)$.

Figure 4.1: The computation of $r(x)$ in Intel's implementation `log` of the natural logarithm function.

the maximum relative error of $r(x)$:

$$\max_{x \in X, |\delta'| < 2^{-53}} \left| \frac{\mathcal{A}'(x) - (x-1)}{x-1} \right| \leq 2^{-53}. \tag{4.4}$$

Note that $\mathcal{A}'(x)$ does not contain the term $(x - \frac{255}{256})\delta_1$ unlike $\mathcal{A}(x)$ (Eq. (4.2)), and we do not need to solve the optimization problem of Eq. (4.3) that has an unbounded objective. In our analysis of `log`, this step is the key to improving the error bound from the previously published bound of 10^{14} ulps to 0.583 ulps (§4.6).

In general, for any 64-bit double-precision floating-point number (or any *double*) $x \geq 2^{-1022}$, `log` computes the quantity $r(x)$ as described in Figure 4.1. The `log` implementation first extracts the exponent p and the 53-bit significand $1.g_2 \cdots g_{53\,(2)}$ of the double x (line 1), and constructs two doubles s and s' that represent the significand of x and the result of masking out the 45 least significant bits of the significand, respectively (line 2). It then computes a double s_{inv} that is close to $1/s$ while having only an 8-bit significand (line 3). Using doubles s, s', and s_{inv}, `log` computes $r(x)$ that approximates $s \times s_{inv} - 1$ (line 4). Note that, if we restrict inputs to $[4095/4096, 1)$, $s = 2 \otimes x$, $s' = 255/128$, $s_{inv} = 1/2$, and line 4 becomes Eq. (4.1).

By using additional properties of floating-point arithmetic, we can show that all the operations except the addition $\oplus$ in line 4 of Figure 4.1 are exact for any input $x \geq 2^{-1022}$. The operation $\ominus$ in $s \ominus s'$ is exact according to the Sterbenz's theorem, because $s'/2 \leq s \leq 2s'$ for any $x \geq 2^{-1022}$. The multiplication $\otimes$ in $(s \ominus s') \otimes s_{inv}$ is also exact according to the following property: $a \otimes b$ is exact whenever $\sigma(a) + \sigma(b) \leq 53$, where $\sigma(d)$ for a double d denotes the number of significand bits of d that are not trailing zeros (§4.4.4). Note that $\sigma(s \ominus s') \leq 45$ because the 8 most significant bits $1, g_2, \cdots, g_8$ of s and s' are canceled out during the subtraction $s \ominus s'$, and that $\sigma(s_{inv}) \leq 8$ by the definition of s_{inv}; thus, we have $\sigma(s \ominus s') + \sigma(s_{inv}) \leq 53$ for any $x \geq 2^{-1022}$. Similarly, we can show that the two operations in $s' \otimes s_{inv} \ominus 1$ are also exact, using Sterbenz's theorem and the property of $\sigma(\cdot)$.

Based on the above exactness results, we can tightly bound the maximum relative error of $r(x)$ for any $x \geq 2^{-1022}$. Since all the operations except $\oplus$ in line 4 of Figure 4.1 are exact, the following is an abstraction of $r(x)$ for any $x \geq 2^{-1022}$ by the $(1+\varepsilon)$-property:

$$\mathcal{A}''(x) \triangleq [(s - s') \times s_{inv} + (s' \times s_{inv} - 1)](1 + \delta'') = (s \times s_{inv} - 1) + (s \times s_{inv} - 1)\delta''$$

where δ'' ranges over $(-2^{-53}, 2^{-53})$. Using the abstraction $\mathcal{A}''(x)$ of $r(x)$, the maximum relative

$$\begin{array}{rrcl} \text{expression} & e & ::= & c \mid x \mid e \circledast e \mid \textit{bit-mask}(e, B) \\ \text{floating-point constant} & c & \in & \mathbb{F} \\ \text{floating-point operation} & \circledast & \in & \{\oplus, \ominus, \otimes, \oslash\} \\ \text{bit-mask constant} & B & \in & \{1, 2, \cdots, 52\} \end{array}$$

Figure 4.2: The abstract syntax of our core language

error of $r(x)$ for any $x \geq 2^{-1022}$ is bounded by

$$\max_{x \geq 2^{-1022}, |\delta''| < 2^{-53}} \left| \frac{\mathcal{A}''(x) - (s \times s_{inv} - 1)}{s \times s_{inv} - 1} \right| \leq 2^{-53}.$$

This analysis generalizes the previous result (Eq. (4.4)) that the maximum relative error of $r(x)$ is bounded by 2^{-53} for $x \in [4095/4096, 1)$ to the larger input interval $x \geq 2^{-1022}$. Note that we cannot obtain such tight bounds on the maximum relative error without proving the exactness of floating-point operations.

In the next three sections, we describe an analysis that automatically exploits such non-trivial properties of floating-point arithmetic to tightly bound the maximum precision loss of floating-point implementations. After defining metrics for precision loss, we present all the properties of floating-point used in our analysis. Some of these are well-known to floating-point experts but not to others, and some properties are new in the sense that they have not been stated explicitly in the literature. Our main contribution is a reduction from the problem of automatically applying these properties to mathematical optimization problems. For example, optimization problems are used to check preconditions of the properties we use above (e.g., whether two values are within a factor of 2) and to compute relevant quantities (e.g., $\sigma(\cdot)$).

As in the previous chapter, this chapter focuses on 64-bit `math.h` implementations and not on 32-bit ones, because the latter can be verified by exhaustive testing while the former cannot be. For this reason, $\mathbb{F}$ in this chapter denotes the set of all finite doubles.

4.3 Abstraction

In this section, we describe the syntax of the floating-point expressions we consider, the abstraction that over-approximates behaviors of an expression, and the abstraction process.

4.3.1 Core Language

Figure 4.2 defines the abstract syntax of the core language for our formal development. An elementary expression e can be a 64-bit floating-point constant c, a 64-bit floating-point input x, an application of a floating-point operation $\circledast$ to subexpressions, or an application of the bit-mask operation $\textit{bit-mask}(\cdot, \cdot)$ to a subexpression. The bit-mask operation $\textit{bit-mask}(e, B)$ masks out B

least significant bits of e's significand ($1 \leq B \leq 52$). For brevity, we describe our techniques for elementary or uni-variate expressions, but they can easily be extended to expressions with multiple inputs. Let $X \subseteq \mathbb{R}$ denote the input interval of an expression, i.e., the floating-point input $x \in X$. And let $\mathcal{E}(e) : X \cap \mathbb{F} \to \mathbb{F}$ denote the concrete semantics of the language, i.e., the result of evaluating e over an input $x \in X \cap \mathbb{F}$ is given by $\mathcal{E}(e)(x)$.

4.3.2 Sound Abstractions

In the remaining parts of this chapter, we use the following abstraction to over-approximate the behaviors of an expression e:

$$\mathcal{A}_{\vec{\delta}}(x) = a(x) + \sum_i b_i(x)\delta_i \qquad \text{where } |\delta_i| \leq \Delta_i.$$

The abstraction $\mathcal{A}_{\vec{\delta}} : X \to \mathbb{R}$ is a function of $x \in X$ and $\vec{\delta} = (\delta_1, \cdots, \delta_n) \in \mathbb{R}^n$, where each δ_i represents a rounding error. $\mathcal{A}_{\vec{\delta}}(x)$ consists of two parts: $a(x)$ and the sum over $b_i(x)\delta_i$. The first part $a(x)$ represents the exact result of e on an input x, which is obtained by replacing every floating-point operation in e with its corresponding real-valued operation and by ignoring every bit-mask operation. In particular, for our benchmarks $a(x)$ is non-linear, i.e., composed of polynomials and rational functions in x. In the second part $b_i(x)\delta_i$ represents an error term that arises from the rounding error of one or more floating-point/bit-mask operation(s). Here the variable $\delta_i \in [-\Delta_i, \Delta_i]$, where $\Delta_i \in \mathbb{R}_{\geq 0}$ is a constant. This abstraction is similar to the one described in [143].

We next define *sound* abstractions of expressions as follows:

Definition 4.1. $\mathcal{A}_{\vec{\delta}}(x)$ is a *sound* abstraction of e if $\forall x \in X \cap \mathbb{F}$. $\mathcal{E}(e)(x) \in \{\mathcal{A}_{\vec{\delta}}(x) : |\delta_i| \leq \Delta_i\}$.

The abstractions form a partial order: $\mathcal{A}_{\vec{\delta}}(x) \sqsubseteq \mathcal{A}'_{\vec{\delta}'}(x)$ if $\forall x \in X \cap \mathbb{F}$. $\{\mathcal{A}_{\vec{\delta}}(x) : |\delta_i| \leq \Delta_i\} \subseteq \{\mathcal{A}'_{\vec{\delta}'}(x) : |\delta'_i| \leq \Delta'_i\}$. The abstractions higher up in the order are more over-approximate. The goal of the next subsection is to construct a sound abstraction of a given expression.

Before describing how to construct such an abstraction, we define the four elementary operations on abstractions that over-approximate their real-valued counterparts. They are defined as:

$$\begin{aligned}
\mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) + \mathcal{A}'_{\vec{\delta}}(x),\\
\mathcal{A}_{\vec{\delta}}(x) \boxminus \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) - \mathcal{A}'_{\vec{\delta}}(x),\\
\mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathit{linearize}(\mathcal{A}_{\vec{\delta}}(x) \times \mathcal{A}'_{\vec{\delta}}(x)),\\
\mathcal{A}_{\vec{\delta}}(x) \boxslash \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathit{inv}(\mathcal{A}'_{\vec{\delta}}(x)).
\end{aligned}$$

Observe that $\boxplus$ and $\boxminus$ are defined simply as $+$ and $-$. On the other hand, the real-valued multiplication of two abstractions may not be an abstraction because of $\delta_i\delta_j$ terms. To soundly remove

such quadratic δ terms, we introduce a new operation $linearize(\cdot)$:

$$linearize\left(a(x)+\sum_i b_i(x)\delta_i+\sum_{i,j} b_{i,j}(x)\delta_i\delta_j\right) \triangleq a(x)+\sum_i b_i(x)\delta_i+\sum_{i,j} b_{i,j}(x)\delta'_{i,j} \quad \text{where } |\delta'_{i,j}| \le \Delta_i\Delta_j.$$

Here $\delta'_{i,j}$ is a fresh variable ranging over $[-\Delta_i\Delta_j, \Delta_i\Delta_j]$. Using the new operation, $\boxtimes$ is defined as the application of $linearize(\cdot)$ to the real-valued multiplication of two abstractions. Note that this abstraction is more precise than the ones considered in prior work that either bound all the quadratic error terms by one ulp [90] or bound the coefficients of the quadratic terms by constants obtained via interval analysis [143]. These constants can lead to imprecise ulp error bounds when $a(x) \approx 0$ and we give an example at the end of the next subsection.

To define $\boxslash$, it is enough to define the operation $inv(\mathcal{A}_{\vec{\delta}}(x))$ that over-approximates the inverse of $\mathcal{A}_{\vec{\delta}}(x)$. We first over-approximate $\mathcal{A}_{\vec{\delta}}(x) = a(x) + \sum_i b_i(x)\delta_i$ to obtain a simpler abstraction $a(x) + a(x)\delta'$ that has only one δ term, and then over-approximate the inverse of the simplified abstraction, $\frac{1}{a(x)+a(x)\delta'} = \frac{1}{a(x)} \cdot \frac{1}{1+\delta'}$, to obtain the final abstraction. This is formalized as:

$$inv\left(a(x)+\sum_i b_i(x)\delta_i\right) \triangleq \frac{1}{a(x)}+\frac{1}{a(x)}\delta'' \qquad \text{where } |\delta''| \le \frac{\Delta'}{1-\Delta'} \ \text{ (assumes } \Delta' < 1).$$

Here δ'' is a fresh variable and Δ' is obtained by solving the following optimization problem:

$$\Delta' = \sum_i \max_{x\in X} \left|\frac{b_i(x)}{a(x)}\right| \cdot \Delta_i$$

Throughout this chapter, for any function $f(x)$ and $g(x)$, the value of $|f(x_0)/g(x_0)|$ at x_0 with $g(x_0) = 0$ is defined as 0 if $f(x_0) = 0$, and ∞ if $f(x_0) \neq 0$. Note that Δ' bounds the relative error of $\mathcal{A}_{\vec{\delta}}(x)$ with respect to its exact term $a(x)$, i.e., $\mathrm{ErrRel}(a(x), \mathcal{A}_{\vec{\delta}}(x)) \le \Delta'$ for all $x \in X$ and $|\delta_i| \le \Delta_i$. Technically, the above definition of $inv(\cdot)$ assumes $\Delta' < 1$, but it can be extended to work even when $\Delta' \ge 1$. However, the condition $\Delta' < 1$ holds for all applications of $inv(\cdot)$ in our benchmarks.

Next, we show that the four operations $\circledast$ defined above over-approximate their real-valued counterparts:

Lemma 4.2. *$\mathcal{A}_{\vec{\delta}}(x) * \mathcal{A}'_{\vec{\delta}}(x) \sqsubseteq \mathcal{A}_{\vec{\delta}}(x) \circledast \mathcal{A}'_{\vec{\delta}}(x)$ for any $* \in \{+, -, \times, /\}$.*

Proof. We sketch the argument that $linearize(\cdot)$ and $inv(\cdot)$ over-approximate their arguments. The main observation is that $\{\delta_i\delta_j : |\delta_i| \le \Delta_i, |\delta_j| \le \Delta_j\} \subseteq [-\Delta_i\Delta_j, \Delta_i\Delta_j]$, and $\{1/(1+\delta') - 1 : |\delta'| \le \Delta'\} \subseteq [-\Delta'/(1+\Delta'), \Delta'/(1-\Delta')] \subseteq [-\Delta'/(1-\Delta'), \Delta'/(1-\Delta')]$ if $\Delta' < 1$. The proof of Lemma 4.5 shows $a(x) + a(x)\delta'$ with $|\delta'| \le \Delta'$ over-approximates $\mathcal{A}_{\vec{\delta}}(x)$. □

$$\frac{e \in dom(\mathcal{K}) \qquad \mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, \_)}{(\mathcal{K}, e) \rhd (\mathcal{K}, \mathcal{A}_{\vec{\delta}})}\ \textsc{Load}$$

$$\frac{}{(\mathcal{K}, c) \rhd (\mathcal{K}[c \mapsto (c, \texttt{false})], c)}\ \text{R1} \qquad \frac{}{(\mathcal{K}, x) \rhd (\mathcal{K}[x \mapsto (x, \texttt{false})], x)}\ \text{R2}$$

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) \qquad * \in \{+, -, \times, /\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = fresh(\varepsilon),\ \delta'' = fresh(\varepsilon') \\ \mathcal{A}'_{\vec{\delta}} = compress((\mathcal{A}_{1,\vec{\delta}} \circledast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta') \boxplus \delta'') \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false})] \end{cases}}\ \text{R3}$$

$$\frac{(\mathcal{K}, \textit{bit-mask}(e_1, B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}})}{(\mathcal{K}, \textit{bit-mask}(e_1, B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = fresh(2^{-52+B}),\ \delta'' = fresh(2^{-1074+B}) \\ \mathcal{A}'_{\vec{\delta}} = compress(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1 + \delta') \boxplus \delta'') \\ \mathcal{K}' = \mathcal{K}_1[\textit{bit-mask}(e_1, B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false})] \end{cases}}\ \text{R4}$$

Figure 4.3: Rules for constructing an abstraction of an expression

4.3.3 Construction of Sound Abstractions

The rules for constructing a sound abstraction of e are given in Figure 4.3. In the rules, $\mathcal{K}$ (and $\mathcal{K}'$) represents a *cache*, a mapping from expressions to tuples of size 2, which stores already computed analysis results. A cache is defined to be *sound* if for any $e \in dom(\mathcal{K})$ with $\mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, b)$, $\mathcal{A}_{\vec{\delta}}$ is a sound abstraction of e and $b = \texttt{true}$ implies that e is not atomic and the last operation of e is exact. The judgment $(\mathcal{K}, e) \rhd (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ denotes that given a sound cache $\mathcal{K}$, our analysis of e constructs a provably sound abstraction $\mathcal{A}_{\vec{\delta}}$ of e and a sound cache $\mathcal{K}'$ that stores both previous and new analysis results. The function $fresh(\Delta)$ returns a fresh variable δ with the constraint $|\delta| \leq \Delta$. The operations $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta')$ and $\mathcal{A}_{\vec{\delta}}(x) \boxplus \delta'$ are defined as a special case of $\mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x)$ and $\mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x)$:

$$\begin{aligned} \mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta') &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) \qquad \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 1 + 1 \cdot \delta', \\ \mathcal{A}_{\vec{\delta}}(x) \boxplus \delta' &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x) \qquad \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 0 + 1 \cdot \delta'. \end{aligned}$$

For now, let us ignore the operation $compress(\cdot)$. The rule Load is applied first whenever applicable; other rules are applied only when the rule Load is not applicable (i.e., an analysis result of an expression is not found in the current cache). The rule R3 is based on the $(1 + \varepsilon)$-property[1], and the rule R4 is based on the following lemma about abstracting the bit-mask operation:

Lemma 4.3. *Given $x \in \mathbb{F}$ and $B \in \{1, 2, \cdots, 52\}$, let $y \in \mathbb{F}$ be the result of masking out B least*

[1]For $* \in \{+, -\}$, we can soundly remove the term $\boxplus\delta''$ from the rule R3 by Theorem 4.14 in §4.4.5 (see R14 in Figure 4.8).

significant bits of x's significand. Then for some $|\delta| < 2^{-52+B}$ and $|\delta'| \leq 2^{-1074+B}$,

$$y = x(1+\delta) + \delta'.$$

The rules in Figure 4.3 (with *compress*$(\cdot)$ erased) can be used to construct a sound abstraction, but the final abstraction can potentially have a huge number of δ variables. Specifically, for $\mathcal{A}_{i,\vec{\delta}}(x)$ with k_i δ variables $(i = 1, 2)$, $\mathcal{A}_{1,\vec{\delta}}(x) \boxtimes \mathcal{A}_{2,\vec{\delta}}(x)$ has $(k_1+1)(k_2+1)-1$ δ variables. Using this fact, we can prove that the abstraction of e can potentially have more than 2^k δ variables, where k is the number of floating-point/bit-mask operations in e. This property holds because for each floating-point/bit-mask operation, we need to apply either the rule R3 or R4 both of which introduce new $(1+\delta')$ terms and perform the operation $(\cdots) \boxtimes (1+\delta')$. Thus, constructing an abstraction based on the rules in Figure 4.3 without *compress*$(\cdot)$ is intractable.

To address this issue, we re-define the operation $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ as:

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta') \triangleq a(x) + a(x)\delta' + \sum_i b_i(x)\delta_i' \qquad \text{where } |\delta_i'| \leq \Delta_i(1+\Delta').$$

Here a given variable δ' ranges over $[-\Delta', \Delta']$ and δ_i' is a fresh variable. Note that under the new definition, $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ now has $k+1$ (instead of $2k+1$) δ variables for any $\mathcal{A}_{\vec{\delta}}(x)$ with k δ variables, and it still over-approximates $\mathcal{A}_{\vec{\delta}}(x) \times (1+\delta')$:

Lemma 4.4. $\mathcal{A}_{\vec{\delta}}(x) \times (1+\delta') \sqsubseteq \mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$.

Proof. For $\{\delta_i(1+\delta') : |\delta_i| \leq \Delta_i, |\delta'| \leq \Delta'\} \subseteq [-\Delta_i(1+\Delta'), \Delta_i(1+\Delta')]$. □

This revised definition of $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ resolves the issue to some extent, but not completely because the number of δ variables is still exponential in the number of multiplications in e. To this end, we define a new operation *compress*$(\mathcal{A}_{\vec{\delta}}(x))$ that significantly reduces the number of δ variables in $\mathcal{A}_{\vec{\delta}}(x)$, as follows:

$$\textit{compress}(\mathcal{A}_{\vec{\delta}}(x)) \triangleq a(x) + a(x)\delta' + \sum_{i \notin S} b_i(x)\delta_i \qquad \text{where } |\delta'| \leq \sum_{i \in S} \gamma_i.$$

Here δ' is a fresh variable, and $\gamma_i \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ and the set S are computed as

$$\gamma_i = \max_{x \in X} \left| \frac{b_i(x)}{a(x)} \right| \cdot \Delta_i \qquad \text{and} \qquad S = \left\{ i : \frac{\gamma_i}{\varepsilon} \leq \tau \right\}. \tag{4.5}$$

The operation *compress*$(\mathcal{A}_{\vec{\delta}}(x))$ can remove some δ variables in $\mathcal{A}_{\vec{\delta}}(x)$, and how aggressively it removes the δ variables is determined by a user-given constant $\tau \in \mathbb{R}_{\geq 0}$ ($\tau = 10$ in our experiments): if τ is big, *compress*$(\mathcal{A}_{\vec{\delta}}(x))$ would have a small number of δ variables but can be too over-approximate, and if τ is small, *compress*$(\mathcal{A}_{\vec{\delta}}(x))$ would capture most behaviors of $\mathcal{A}_{\vec{\delta}}(x)$ precisely but can have

many δ variables. Note that γ_i is computed by solving the optimization problem (of a single variable) that also appears in the computation of $inv(\cdot)$. The quantity γ_i represents the contribution of the i-th error term $b_i(x)\delta_i$ to the overall relative error of $\mathcal{A}_{\vec{\delta}}(x)$ with respect to $a(x)$; thus, from Theorem 2.7, γ_i/ε represents how much the error term $b_i(x)\delta_i$ contributes to the overall ulp error of $\mathcal{A}_{\vec{\delta}}(x)$. Thus, the set S represents the indices of the error terms whose contribution to the overall ulp error is small enough, i.e., $\leq \tau$ ulps. The $compress(\cdot)$ procedure merges all the error terms of $\mathcal{A}_{\vec{\delta}}(x)$ that have such small contribution to the total ulp error, into a single error term $a(x)\delta'$, and leaves all the other error terms of $\mathcal{A}_{\vec{\delta}}(x)$ as is. Conceptually, we can set $\tau = \infty$ and merge all δ terms into a single term. But for some cases, e.g., Theorem 4.8 in §4.4.3, this merging can lead to very imprecise abstractions.

Like all the previous operations on abstractions, $compress(\cdot)$ over-approximates its argument:

Lemma 4.5. $\mathcal{A}_{\vec{\delta}}(x) \sqsubseteq compress(\mathcal{A}_{\vec{\delta}}(x))$.

Proof. First, we show that for any i with $\gamma_i < \infty$ we have $b_i(x)\delta_i \sqsubseteq a(x)\delta_i'$, where $|\delta_i'| \leq \gamma_i$. Consider any i with $\gamma_i < \infty$ and any $x \in X$. If $a(x) \neq 0$,

$$|b_i(x)\delta_i| = |a(x)| \cdot \left|\frac{b_i(x)}{a(x)}\delta_i\right| \leq |a(x)| \cdot \left|\frac{b_i(x)}{a(x)}\right| \Delta_i \leq |a(x)| \cdot \gamma_i = |a(x)\gamma_i|$$

which implies $\{b_i(x)\delta_i : |\delta_i| \leq \Delta_i\} \subseteq \{a(x)\delta_i' : |\delta_i'| \leq \gamma_i\}$. If $a(x) = 0$, $\gamma_i < \infty$ implies $b_i(x) = 0$, so we have $\{b_i(x)\delta_i : |\delta_i| \leq \Delta_i\} = \{0\} = \{a(x)\delta_i' : |\delta_i'| \leq \gamma_i\}$.

Next, it is easy to see that $\sum_{i\in S} a(x)\delta_i' \sqsubseteq a(x)\delta'$, where $|\delta'| \leq \sum_{i\in S}\gamma_i$. Combining the two facts implies $\sum_{i\in S} b_i(x)\delta_i \sqsubseteq a(x)\delta'$. Hence $\sum_i b_i(x)\delta_i = \sum_{i\in S} b_i(x)\delta_i + \sum_{i\notin S} b_i(x)\delta_i \sqsubseteq a(x)\delta' + \sum_{i\notin S} b_i(x)\delta_i$. □

We remark that the previous work on similar abstractions, [53, 143], does not use the $compress(\cdot)$ operation.

Let us revisit the rules in Figure 4.3. The rules R3 and R4 use the re-defined operation $(\cdots) \boxtimes (1+\delta')$ and the newly defined operation $compress(\cdot)$, to reduce the number of δ variables in the abstractions of $e_1 \circledast e_2$ and $bit\text{-}mask(e_1, B)$. Using these two operations, the rules can be applied to expressions of practical sizes. We note that these two operations will be re-defined again in §4.4.3. Finally, we establish the soundness of the rules:

Theorem 4.6. *If $(\cdot, e) \triangleright (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ then $\mathcal{A}_{\vec{\delta}}$ is a sound abstraction of e.*

Proof. We generalize the above statement as: if $(\mathcal{K}, e) \triangleright (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ and $\mathcal{K}$ is a sound cache, then $\mathcal{K}'$ is a sound cache and $\mathcal{A}_{\vec{\delta}}$ is a sound abstraction of e. We can prove this by induction on the derivation tree of $(\mathcal{K}, e) \triangleright (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ using Theorem 2.5 and Lemmas 4.2, 4.3, 4.4, and 4.5. □

We conclude this subsection by demonstrating that the abstraction used in [143] is not sufficient to prove tight ulp error bounds. For example, consider Intel's `sin` implementation of the sine function over the input interval $X = [2^{-252}, \frac{\pi}{64}]$. Since `sin` computes $x - \frac{1}{6}x^3 + \cdots$ for an input $x \in X$,

applying the $(1+\varepsilon)$-property produces an abstraction of `sin` that contains the error term $-\frac{1}{6}x^3\delta_1\delta_2\delta_3$, where $|\delta_i| < \varepsilon$ $(i = 1, 2, 3)$. In the abstraction of [143], the cubic error term is over-approximated by a first-order error term $C\delta'$, where $|\delta'| < \varepsilon$ and $C = \varepsilon^2 \max\{|-\frac{1}{6}x^3| : x \in X\} \approx 2 \times 10^{-37}$. Hence with this abstraction, a bound on the maximum ulp error of `sin` over X (with respect to $\sin x$) is at least $\frac{1}{\varepsilon}\max\{|C\delta'/\sin x| : x \in X, |\delta'| < \varepsilon\} \approx 2 \times 10^{39}$ ulps, which is too loose. The culprit is that this abstraction has constant coefficients for higher-order error terms and these terms become significant for inputs near zero.

4.4 Exploiting Exactness Properties

Although the rules in Figure 4.3 can be used to construct a sound abstraction of an expression e, the resulting abstraction can over-approximate the behaviors of e too imprecisely and fail to prove a tight error bound of e. Consider a part of the implementation of `log` discussed in §4.2: $e = (2 \otimes x) \ominus \frac{255}{128}$ with $X = [\frac{4095}{4096}, 1)$. As already explained, the operations $\otimes$ and $\ominus$ in e are exact, i.e., introduces no rounding errors due to the exactness of multiplication by 2 and Sterbenz's theorem (Theorem 4.7), so $\mathcal{A}_{\vec{\delta}}(x) = 2x - \frac{255}{128}$ is a sound abstraction of e. However, the rules in Figure 4.3 generate $\mathcal{A}'_{\vec{\delta}}(x) = (2x - \frac{255}{128}) + (2x - \frac{255}{128})\delta_1 + \cdots$ as an abstraction of e by simply applying the $(1+\varepsilon)$-property to the $\otimes$ and $\ominus$ operation. The proof then uses $\mathcal{A}'_{\vec{\delta}}$ as an abstraction of e, instead of using the more precise abstraction $\mathcal{A}_{\vec{\delta}}$. This imprecision leads to the imprecise error bound of 10^{14} ulps for `log` over X, presented in Chapter 3.

To prove a tighter error bound, we construct a more precise abstraction by avoiding the application of the $(1+\varepsilon)$-property whenever possible while maintaining soundness (Theorem 4.6). For each floating-point operation $e_1 \circledast e_2$, we first determine whether the operation $\circledast$ is exact or not using some properties of floating-point arithmetic. If the particular operation $\circledast$ is exact, we simply use $\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}$ as a sound abstraction of $e_1 \circledast e_2$, where $\mathcal{A}_{i,\vec{\delta}}$ is a sound abstraction of e_i $(i = 1, 2)$. In contrast, the $(1+\varepsilon)$-property instead yields the less precise abstraction $(\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta') \boxplus \delta''$. The key to this approach is automatically determining whether a given floating-point operation is exact, i.e., produces no rounding errors.

Some of the properties of floating-point that we use are well-known to floating-point experts (§4.4.1, 4.4.2, 4.4.3, 4.4.6) and some are new (i.e., haven't appeared explicitly in the literature) to the best of our knowledge (§4.4.4, 4.4.5). We remark that it was challenging for us to rediscover these properties and infer how to use them in automatic proofs of error bounds for practical floating-point implementations.

4.4.1 Simple Exact Operations

We start with the simplest situation where a floating-point addition/subtraction or a floating-point multiplication/division is exact: for any $x, y \in \mathbb{F}$ with $y = 2^n$ for some $n \in \mathbb{Z}$, we have $x \circledast 0 = x$ if $* \in \{+, -\}$, and $x \circledast y = x * y$ if $* \in \{\times, /\}$. In other words, floating-point addition by 0 is always

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, 0) \qquad * \in \{+, -\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}_{1,\vec{\delta}}, \texttt{true})], \mathcal{A}_{1,\vec{\delta}})} \text{ R5}$$

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, 2^n)\ (n \in \mathbb{Z}) \qquad * \in \{\times, /\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}_{1,\vec{\delta}} * 2^n, \texttt{true})], \mathcal{A}_{1,\vec{\delta}} * 2^n)} \text{ R6}$$

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, c_1) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, c_2) \qquad * \in \{+, -, \times, /\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}_2[e_1 \circledast e_2 \mapsto (c', c' \texttt{ == } c_1 * c_2)], c'),\ \text{where } c' = c_1 \circledast c_2} \text{ R7}$$

Figure 4.4: Rules for simple exact operations

exact, and floating-point multiplication by an integer power of 2 is always exact because multiplying x by a power of 2 changes only the exponent of x, not its significand.[2]

Figure 4.4 presents the rules based on the above property. The rule R5 considers the addition/-subtraction case and the rule R6 considers the multiplication/division case[3]; their commutative counterparts are omitted. The rule R7 considers the case where the evaluation result of e_1 and e_2 are exactly known as c_1 and c_2. In such a case, though the floating-point operation $\circledast$ in $e_1 \circledast e_2$ may not be exact, we can know the exact evaluation result of the operation, $c_1 \circledast c_2$, by partial evaluation. Note that all the rules in the figure do not use the $(1 + \varepsilon)$-property.

4.4.2 Sterbenz's Theorem

The next situation where a floating-point addition/subtraction is exact is described in Sterbenz's theorem [146]:

Theorem 4.7 ([146]). *Let $x, y \in \mathbb{F}$ with $x, y \geq 0$. Then*

$$\frac{x}{2} \leq y \leq 2x \quad \Longrightarrow \quad x \ominus y = x - y.$$

The theorem says that the floating-point subtraction of $x \geq 0$ and $y \geq 0$ is exact whenever y is within a factor of two of x.

Typical examples that make use of Sterbenz's theorem are from range reduction steps that reduce the computation of $f(x)$ to the computation of $g(r)$ such that the range of r is much smaller than that of x. A range reduction step used to compute $log\, x$ has been discussed in §4.2 and at the beginning

[2] Technically, $x \otimes 2^n = x \times 2^n$ may not hold if $x \times 2^n$ is very small (e.g., $x = 2^{-1074}$ and $n = -1$) since the exponent of a double cannot be smaller than -1022.

[3] Strictly speaking, the rule R6 is unsound according to Footnote 2. For a sound version of the rule R6, refer to the rule R6′ (Appendix A.1.2) which uses quantities $\sigma(e)$ and $\mu(e)$ introduced in §4.4.4 and §4.4.5.

of this section. Another common range reduction is:

$$n = \text{round}(K_{inv} \otimes x), \quad r = x \ominus (K \otimes n),$$

where n is an integer, and $K_{inv}, K \in \mathbb{F}_{>0}$ have a relationship that $K_{inv} \approx 1/K$. For example, if $K = \text{fl}(\pi)$ then this range reduction can reduce the computation of $sin\, x$ to $sin\, r$ for $r \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. This range reduction relies on Sterbenz's theorem to guarantee that the operation $\ominus$ in the computation of r is exact.

Before explaining how to exploit Sterbenz's theorem in our framework, we point out that Theorem 4.7 considers only the case $x, y \geq 0$. To cover the case $x, y \leq 0$ as well, we extend the theorem in the following way: for any $x, y \in \mathbb{F}$, if they satisfy

$$\frac{x}{2} \leq y \leq 2x \qquad \text{or} \qquad 2x \leq y \leq \frac{x}{2}, \tag{4.6}$$

then $x \ominus y = x - y$. From now on, we refer to this extended theorem (instead of Theorem 4.7) as Sterbenz's theorem.

Next, we derive optimization problems, based on Sterbenz's theorem, that can check whether an operation $\ominus$ between two expressions e_1 and e_2 is exact. As e_1 and e_2 are functions of x, we would like to check if the operation $\mathcal{E}(e_1)(x) \ominus \mathcal{E}(e_2)(x)$ is exact for all $x \in X \cap \mathbb{F}$. According to Sterbenz's theorem (Eq. (4.6)), the operation is exact for all $x \in X \cap \mathbb{F}$ if

$$\forall x \in X \cap \mathbb{F}. \left(\frac{1}{2}\mathcal{E}(e_1)(x) \leq \mathcal{E}(e_2)(x) \leq 2\mathcal{E}(e_1)(x)\right) \vee \left(2\mathcal{E}(e_1)(x) \leq \mathcal{E}(e_2)(x) \leq \frac{1}{2}\mathcal{E}(e_1)(x)\right). \tag{4.7}$$

However, we do not know $\mathcal{E}(e_i)(x)$ statically; rather we can construct its abstraction as described in §4.3. Let $\mathcal{A}_{i,\vec{\delta}}$ be a sound abstraction of e_i $(i = 1, 2)$. From the definition of a sound abstraction, for any $x \in X \cap \mathbb{F}$, we have $\mathcal{E}(e_1)(x) = \mathcal{A}_{1,\vec{\delta}}(x)$ and $\mathcal{E}(e_2)(x) = \mathcal{A}_{2,\vec{\delta}}(x)$ for some $\vec{\delta} \in \vec{\Delta}$, where $\vec{\Delta} = [-\Delta_1, \Delta_1] \times \cdots \times [-\Delta_n, \Delta_n]$. Using $\mathcal{A}_{1,\vec{\delta}}$ and $\mathcal{A}_{2,\vec{\delta}}$, we strengthen Eq. (4.7) to Eq. (4.8):

$$\forall x \in X. \forall \vec{\delta} \in \vec{\Delta}. \left[\left(\frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x) \leq \mathcal{A}_{2,\vec{\delta}}(x) \leq 2\mathcal{A}_{1,\vec{\delta}}(x)\right) \vee \left(2\mathcal{A}_{1,\vec{\delta}}(x) \leq \mathcal{A}_{2,\vec{\delta}}(x) \leq \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x)\right)\right]. \tag{4.8}$$

Eq. (4.8) is stronger than Eq. (4.7) in two aspects: it is quantified over x that ranges over X (instead of over $X \cap \mathbb{F}$), and additionally over $\vec{\delta}$. The first change is motivated by the fact that checking inequalities (and solving optimization problems) over $X \subseteq \mathbb{R}$ is easier than over the discrete set $X \cap \mathbb{F}$. The second change is necessary since we do not know statically which $\vec{\delta} \in \vec{\Delta}$ would satisfy $\mathcal{E}(e_1)(x) = \mathcal{A}_{1,\vec{\delta}}(x)$ and $\mathcal{E}(e_2)(x) = \mathcal{A}_{2,\vec{\delta}}(x)$ for each x. Although Eq. (4.8) is easier to handle than Eq. (4.7), transforming it directly into optimization problems is still difficult because of the $\vee$ within

$$\frac{\begin{array}{cc}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \geq 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \leq 0\end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true})] \end{cases}} \text{ R8}$$

$$\frac{\begin{array}{cc}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \geq 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \leq 0\end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true})] \end{cases}} \text{ R9}$$

Figure 4.5: Rules for applying Sterbenz's theorem

the quantifiers. We strengthen it further to obtain Eq. (4.9):

$$\left(\forall x.\forall\vec{\delta}.\ \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x) \leq \mathcal{A}_{2,\vec{\delta}}(x) \leq 2\mathcal{A}_{1,\vec{\delta}}(x)\right) \vee \left(\forall x.\forall\vec{\delta}.\ 2\mathcal{A}_{1,\vec{\delta}}(x) \leq \mathcal{A}_{2,\vec{\delta}}(x) \leq \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x)\right), \tag{4.9}$$

where x ranges over X and $\vec{\delta}$ over $\vec{\Delta}$. The left clause of Eq. (4.9) is logically equivalent to

$$\left(\min_{x\in X,\vec{\delta}\in\vec{\Delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \geq 0\right) \wedge \left(\max_{x\in X,\vec{\delta}\in\vec{\Delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \leq 0\right) \tag{4.10}$$

involving two optimization problems that are sufficient to ensure $e_1 \ominus e_2$ is exact.

The rules shown in Figure 4.5 are based on the above derivation. The rule R8 does not apply the $(1+\varepsilon)$-property if Eq. (4.10) holds. The rule R9 is based on the counterpart of Eq. (4.10) derived from the right clause of Eq. (4.9). Note that in Figure 4.5 the rules for $e_1 \oplus e_2$ are omitted: they can be obtained from the rules for $e_1 \ominus e_2$ by negating $\mathcal{A}_{2,\vec{\delta}}$ as $x \oplus y = x \ominus (-y)$ for any x and y.

4.4.3 Dekker's Theorem

The next property of floating-point arithmetic that we use to construct a more precise abstraction is Dekker's theorem [42]. The theorem suggests a way to compute the rounding error, $(x \oplus y) - (x + y)$, of an operation $x \oplus y$. It is well-known that the rounding error $r = (x \oplus y) - (x + y)$ is in fact a double for any $x, y \in \mathbb{F}$, and Dekker's theorem provides a way to recover r using only floating-point operations on x and y:

Theorem 4.8 ([42])**.** *Let $x, y \in \mathbb{F}$ with $|x + y| \leq \max \mathbb{F}$ and $r = x \oplus y \ominus x \ominus y$. Then*

$$|x| \geq |y| \quad \Longrightarrow \quad r = (x \oplus y) - (x + y).$$

The double $r = x \oplus y \ominus x \ominus y$ in the theorem represents the rounding error of $x \oplus y$.

Let us start with the rule R11 of Figure 4.6 that constructs a tighter abstraction of an expression

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) \qquad \mathsf{hasDekker}(e_1 \oplus e_2)\end{array}}{(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\varepsilon, \texttt{true}) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \oplus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}\langle\delta'\rangle)] \end{cases}} \text{R10}$$

$$\frac{\begin{array}{ccc}(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}_1, \_) & \mathcal{K}_1(e_1) = (\mathcal{A}_{1,\vec{\delta}}, \_) & \mathcal{K}_1(e_1 \oplus e_2) = (\_, \texttt{false}\langle\delta'\rangle) \\ & \mathcal{K}_1(e_2) = (\mathcal{A}_{2,\vec{\delta}}, \_) & \min_{x,\vec{\delta}} |\mathcal{A}_{1,\vec{\delta}}| \geq \max_{x,\vec{\delta}} |\mathcal{A}_{2,\vec{\delta}}|\end{array}}{(\mathcal{K}, e_1 \oplus e_2 \ominus e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes \delta') \\ \mathcal{K}' = \mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false})] \end{cases}} \text{R11}$$

$$\frac{(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}_1, \_) \qquad \mathcal{K}_1(e_1 \oplus e_2) = (\_, \texttt{true})}{(\mathcal{K}, e_1 \oplus e_2 \ominus e_1 \ominus e_2) \rhd (\mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (0, \texttt{true})], 0)} \text{R12}$$

Figure 4.6: Rules for applying Dekker's theorem

$e_r = e_1 \oplus e_2 \ominus e_1 \ominus e_2$ based on Dekker's theorem. The rule first checks whether Dekker's theorem is applicable to the expression e_r, i.e., whether the condition $P_1 \triangleq \forall x \in X \cap \mathbb{F}.\ |\mathcal{E}(e_1)(x)| \geq |\mathcal{E}(e_2)(x)|$ is satisfied. However, P_1 cannot be checked statically and the rule actually checks a stronger condition $P_2 \triangleq \min_{x,\vec{\delta}} |\mathcal{A}_{1,\vec{\delta}}(x)| \geq \max_{x,\vec{\delta}} |\mathcal{A}_{2,\vec{\delta}}(x)|$ by solving optimization problems on a sound abstraction $\mathcal{A}_{i,\vec{\delta}}$ of e_i $(i = 1, 2)$. The derivation of P_2 from P_1 is similar to the derivation of Eq. (4.10) from Eq. (4.7) in §4.4.2. Once the rule successfully checks that P_2 is true, it constructs a sound abstraction of e_r not by applying the $(1+\varepsilon)$-property, but by applying Dekker's theorem which says e_r is the rounding error of $e_1 \oplus e_2$. Note that the rule requires a new operation $\mathcal{A}_{\vec{\delta}}(x) \boxtimes \delta'$:

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes \delta' \triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) \qquad \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 0 + 1 \cdot \delta'.$$

Although the rule R11 constructs a tighter abstraction of $e_r = e_1 \oplus e_2 \ominus e_1 \ominus e_2$ than the rule R3, it does not fully capture the essence of Dekker's theorem: the possibility of the rounding error of $x \oplus y$ being exactly canceled out by $x \oplus y \ominus x \ominus y$. Such cancellation may not occur even with the rule R11 because some δ variables can be replaced with or merged into fresh ones by $\mathcal{A}_{\vec{\delta}} \boxtimes (1+\delta')$ and $\mathit{compress}(\cdot)$.

We re-define the operations $\mathcal{A}_{\vec{\delta}} \boxtimes (1+\delta')$ and $\mathit{compress}(\cdot)$ and introduce the rule R10 to ensure that δ variables related to Dekker's theorem are preserved (i.e., not replaced with or merged into fresh variables). As a first step, each variable δ_i is associated with the predicate $\mathsf{preserve}(\delta_i)$ which indicates whether δ_i should be preserved: $\mathsf{preserve}(\delta_i) = \texttt{true}$ denotes that δ_i should be preserved, whereas $\mathsf{preserve}(\delta_i) = \texttt{false}$ denotes that merging δ_i is allowed. Using $\mathsf{preserve}(\cdot)$, we re-define the following operations on abstractions:

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta') \triangleq a(x) + a(x)\delta' + \sum_{i \in R} b_i(x)\delta'_i + \sum_{i \notin R} (b_i(x)\delta_i + b_i(x)\delta''_i) \quad \text{where} \begin{array}{l} |\delta'_i| \leq \Delta_i(1+\Delta') \\ |\delta''_i| \leq \Delta_i\Delta' \end{array},$$

$$compress(\mathcal{A}_{\vec{\delta}}(x)) \triangleq a(x) + a(x)\delta' + \sum_{i \notin R \cap S} b_i(x)\delta_i \qquad \text{where } |\delta'| \leq \sum_{i \in R \cap S} \gamma_i.$$

Here δ', δ'_i, and δ''_i are fresh variables, S and γ_i are defined as before (Eq. (4.5)), and $R = \{i : \mathsf{preserve}(\delta_i) = \texttt{false}\}$. The re-defined operations preserve any δ_i with $\mathsf{preserve}(\delta_i) = \texttt{true}$. Note that the previous definition of $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ and $compress(\cdot)$ is obtained by setting $\mathsf{preserve}(\delta_i) = \texttt{false}$ for all i.

In the rule R10 of Figure 4.6, the predicate $\mathsf{hasDekker}(e_1 \oplus e_2)$ denotes that a given expression contains $e_r = e_1 \oplus e_2 \ominus e_1 \ominus e_2$ as a subexpression, and the value $\texttt{false}\langle\delta'\rangle$ denotes that the operation $\oplus$ in $e_1 \oplus e_2$ may not be exact and the rounding error from this $\oplus$ operation is modeled by the variable δ'. The function $fresh(\Delta, b)$ returns a fresh variable δ with the constraint $|\delta| \leq \Delta$ and $\mathsf{preserve}(\delta) = b$; the previous function $fresh(\Delta)$ now denotes $fresh(\Delta, \Delta > \varepsilon)$, which implies that by default only those δ variables from bit-mask operations are preserved. The antecedent of the rule R10 indicates that Dekker's theorem can possibly be applied to $e_1 \oplus e_2$ and e_r. In this case the rule sets $\mathsf{preserve}(\delta')$ to be $\texttt{true}$, where δ' denotes the rounding error of $e_1 \oplus e_2$, and prevents δ' from being removed. Note that the rule R11 uses this δ' in an abstraction of e_r to make the cancellation possible. The rule R10 does not add an absolute error term δ'' ($|\delta''| \leq \varepsilon'$) in its consequent by the refined $(1+\varepsilon)$-property (Theorem 4.14 in §4.4.5).

To illustrate how the rules R10 and R11 are applied, consider an expression $e = e_1 \ominus (e_2 \oplus e_3)$ over $X = [1, 2]$, where $e_1 = x \oplus 1$, $e_2 = x \oplus 1 \ominus x \ominus 1$, and $e_3 = 0.01 \otimes x \otimes x$. The expression e accurately computes $1 + x - 0.01x^2$ by subtracting e_2 (which evaluates exactly to the rounding error of e_1 by Dekker's theorem) from e_1. Analyzing e with the rules R10 and R11 produces the following derivation tree:

$$\dfrac{\dfrac{\cdots \quad \mathsf{hasDekker}(e_1)}{(\cdot, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}})}\,\text{R10} \qquad \dfrac{\dfrac{\dfrac{x \oplus 1 \in dom(\mathcal{K}_1)}{(\mathcal{K}_1, x \oplus 1) \rhd (\mathcal{K}_1, \_)}\,\textsc{Load} \qquad \begin{array}{c}\mathcal{K}_1(x) = (x, \_)\\ \mathcal{K}_1(1) = (1, \_)\\ \mathcal{K}_1(x \oplus 1) = (\_, \texttt{false}\langle\delta_1\rangle)\\ \min_{x,\vec{\delta}} |x| \geq \max_{x,\vec{\delta}} |1|\end{array}}{(\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}})}\,\text{R11} \qquad (\mathcal{K}_2, e_3) \rhd \cdots}{(\mathcal{K}_1, e_2 \oplus e_3) \rhd \cdots}\,\text{R3}}{(\cdot, e_1 \ominus (e_2 \oplus e_3)) \rhd (\cdots, \mathcal{A}'_{\vec{\delta}})}\,\text{R3}$$

Here $\mathcal{A}_{1,\vec{\delta}}(x) = (x+1) + (x+1)\delta_1$, $\mathcal{K}_1 = [1 \mapsto (1, \texttt{false}), x \mapsto (x, \texttt{false}), e_1 \mapsto (\mathcal{A}_{1,\vec{\delta}}, \texttt{false}\langle\delta_1\rangle)]$, $\mathcal{A}_{2,\vec{\delta}}(x) = (x+1)\delta_1$, $\mathcal{K}_2 = \mathcal{K}_1[e_2 \mapsto (\mathcal{A}_{2,\vec{\delta}}, \texttt{false})]$, and $\mathcal{A}'_{\vec{\delta}}(x) = (1 + x - 0.01x^2) + (1 + x - 0.01x^2)\delta'$, where $|\delta_1| \leq \varepsilon$ and $|\delta'| \leq 1.041\varepsilon$. The above derivation tree states that an abstraction of e_1 and of e_2 are constructed as $\mathcal{A}_{1,\vec{\delta}}$ and $\mathcal{A}_{2,\vec{\delta}}$. Note that the abstraction $\mathcal{A}_{2,\vec{\delta}}$ of e_2 is $(x+1)\delta_1$, the error term of $\mathcal{A}_{1,\vec{\delta}}$ which models the rounding error of $x \oplus 1$. Hence the final abstraction $\mathcal{A}'_{\vec{\delta}}$ of e does not contain the error term $(x+1)\delta_1$ due to its cancellation, which is what we desired.

The rule R12 in Figure 4.6, based on Lemma 4.9, deals with the specific case when $e_1 \oplus e_2$ is exact. The lemma says that if $x \oplus y$ is exact then $x \oplus y \ominus x \ominus y = 0$ regardless of the ordering between x and y.

Lemma 4.9. *Let $x, y \in \mathbb{F}$ and $r = x \oplus y \ominus x \ominus y$. Then*

$$x \oplus y = x + y \qquad \Longrightarrow \qquad r = 0.$$

Note that there are several variants of Theorem 4.8 and Lemma 4.9, and Figure 4.6 omits the corresponding variants of the rules R10, R11, and R12 for brevity. For instance, one variant of Theorem 4.8 is: $|y| \geq |x|$ implies $r = -((x \ominus y) - (x - y))$ where $r = x \ominus (x \ominus y \oplus y)$. The rules based on each such variant of Theorem 4.8 and Lemma 4.9 can be designed analogously to the rules R10, R11, and R12 by focusing on different expressions (e.g., $x \ominus y$ and $x \ominus (x \ominus y \oplus y)$) and extending the definition of $\mathsf{hasDekker}(\cdot)$ accordingly.

4.4.4 Nonzero Significand Bits

The next floating-point property we exploit is based on $\sigma(d)$, the number of the significand bits of $d \in \mathbb{F}$ that are not trailing zeros. To formally define $\sigma(\cdot)$ over a subset of $\mathbb{R}$, we define the exponent function $\mathrm{expnt}(\cdot)$ as:

$$\mathrm{expnt}(r) \triangleq \begin{cases} k & \text{for } |r| \in [2^k, 2^{k+1}) \quad \text{where } k \in [-1022, 1023] \cap \mathbb{Z} \\ -1022 & \text{for } |r| \in [0, 2^{-1022}). \end{cases}$$

Using $\mathrm{expnt}(\cdot)$, we define the function $\sigma : [-\max \mathbb{F}, \max \mathbb{F}] \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ as follows: for $r \neq 0$,

$$\sigma(r) \triangleq \begin{cases} \max\{i \in \mathbb{Z}_{\geq 1} : f_i \neq 0\} & \text{if defined} \\ \infty & \text{otherwise} \end{cases}$$

where $f_1.f_2f_3\cdots_{(2)}$ is the binary representation of $r/2^{\mathrm{expnt}(r)}$ ($f_i \in \{0, 1\}$ for all i), and $\sigma(0) \triangleq 0$. For example, $\sigma(5/8) = 3$ since $\mathrm{expnt}(5/8) = -1$ and $5/8 = 2^{-1} \times 1.01_{(2)}$, and $\sigma(1/5) = \infty$ since $1/5 = 2^{-3} \times 1.10011001\cdots_{(2)}$.

The following theorem uses $\sigma(\cdot)$ to determine if a floating-point operation is exact:

Theorem 4.10. *Let $x, y \in \mathbb{F}$ with $|x * y| \leq \max \mathbb{F}$ where $* \in \{+, -, \times, /\}$. Then*

$$\sigma(x * y) \leq 53 \qquad \Longrightarrow \qquad x \circledast y = x * y.$$

To make use of this theorem, we must compute $\sigma(x * y)$. The following two lemmas can be used to bound $\sigma(x * y)$, given $\sigma(x)$ and $\sigma(y)$ (or their upper bounds). First, Lemma 4.11 handles multiplication:

$$\frac{\begin{array}{ccc}\mathcal{K}_1(e_1) = (\_, \_, \sigma_1) & & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2) & * \in \{+, -, \times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \sigma' = \textit{bound-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \sigma' \leq 53\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A'}_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A'}_{\vec{\delta}} = \textit{compress}(\mathcal{A}_{1,\vec{\delta}} \circledast \mathcal{A}_{2,\vec{\delta}}) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A'}_{\vec{\delta}}, \texttt{true}, \sigma')] \end{cases}} \text{ R13}$$

Figure 4.7: Rule for using $\sigma(\cdot)$

Lemma 4.11. *Let $x, y \in \mathbb{F}$ with $|xy| \leq \max \mathbb{F}$. Assume $x, y \neq 0$. Then*

$$\sigma(x \times y) \leq \sigma(x) + \sigma(y) + k$$

where $k = \min\{n \in \mathbb{Z}_{\geq 0} : |xy| \geq 2^{-1022-n}\}$.

Second, Lemma 4.12 handles addition and subtraction:

Lemma 4.12. *Let $x, y \in \mathbb{F}$ with $|x + y| \leq \max \mathbb{F}$. Assume $x, y > 0$ and $\text{expnt}(x) \geq \text{expnt}(y)$. Then*

$$\begin{aligned}\sigma(x + y) &\leq \max\{\sigma(x), \sigma(y) + \Delta e\} + k, \\ \sigma(x - y) &\leq \max\{\max\{\sigma(x), \sigma(y) + \Delta e\} - \min\{l, \text{expnt}(x) + 1022\}, 0\},\end{aligned}$$

where $\Delta e = \text{expnt}(x) - \text{expnt}(y)$, $k = \min\{n \in \mathbb{Z}_{\geq 0} : |x + y| < 2^{\text{expnt}(x)+1+n}\}$, and $l = \max\{n \in \mathbb{Z}_{\geq 0} : |x - y| < 2^{\text{expnt}(x)+1-n}\}$.

The lemma says that when $\sigma(x)$ and $\sigma(y)$ are fixed, $\sigma(x + y)$ and $\sigma(x - y)$ decrease as x and y get closer to each other (since it makes Δe smaller and l larger). In the lemma, the integer k represents whether there is a carry-over during the addition $x + y$, as $k = 0$ if no carry-over and $k = 1$ otherwise. The integer l is subtracted from the upper bound on $\sigma(x - y)$ to consider the case when x and y are close: if they are close enough, some of x's most significant bits can be canceled out by y's corresponding significant bits during the subtraction $x - y$, thereby reducing $\sigma(x - y)$. The term $\min\{\cdots, \text{expnt}(x) + 1022\}$ is necessary to consider the case when $|x - y| < 2^{-1022}$.

Note that Lemma 4.12 is a generalization of Sterbenz's theorem. We are unaware of any previous work that proves this lemma. Moreover, this lemma is a general fact about floating-point that may have applicability beyond this dissertation.

We present the rule based on Theorem 4.10 in Figure 4.7. To apply Theorem 4.10, we need to track $\sigma(e)$ for each expression e, which is defined as:

$$\sigma(e) \triangleq \max\{\sigma(\mathcal{E}(e)(x)) : x \in X \cap \mathbb{F}\}.$$

The cache $\mathcal{K}$ is extended to store an upper bound of $\sigma(e)$ for each e. The rule R13 computes σ' that upper bounds $\sigma(e_1 * e_2)$ via Algorithm 1, and then checks whether $\sigma' \leq 53$. If the check passes, the

Algorithm 1 *bound-*$\sigma(*, f_1, f_2, \sigma_1, \sigma_2)$

Let $D = dom(f_1)$
if $* = +$ **then**
  Compute $\Delta e \in \mathbb{Z}_{\geq 0}$ such that $\forall \vec{x} \in D.\ |\text{expnt}(f_1(\vec{x})) - \text{expnt}(f_2(\vec{x}))| \leq \Delta e$
  if $\forall \vec{x} \in D.\ \text{expnt}(f_1(\vec{x})) \geq \text{expnt}(f_2(\vec{x}))$ **then**
    if $\forall \vec{x} \in D.\ f_1(\vec{x}) f_2(\vec{x}) \geq 0$ **then**
      Compute $k \in \mathbb{Z}_{\geq 0}$ such that $\forall \vec{x} \in D.\ |f_1(\vec{x}) + f_2(\vec{x})| < 2^{\text{expnt}(f_1(\vec{x}))+1+k}$
      return $\max\{\sigma_1, \sigma_2 + \Delta e\} + k$
    end if
    if $\forall \vec{x} \in D.\ f_1(\vec{x}) f_2(\vec{x}) < 0$ **then**
      Compute $\begin{cases} l \in \mathbb{Z}_{\geq 0} \text{ such that } \forall \vec{x} \in D.\ |f_1(\vec{x}) + f_2(\vec{x})| < 2^{\text{expnt}(f_1(\vec{x}))+1-l} \\ em_1 \in \mathbb{Z} \text{ such that } \forall \vec{x} \in D.\ \text{expnt}(f_1(\vec{x})) \geq em_1 \end{cases}$
      return $\max\{\max\{\sigma_1, \sigma_2 + \Delta e\} - \min\{l, em_1 + 1022\}, 0\}$
    end if
  end if
  if $\forall \vec{x} \in D.\ \text{expnt}(f_1(\vec{x})) \leq \text{expnt}(f_2(\vec{x}))$ **then**
    [symmetric to the above case]
  end if
  return $\max\{\sigma_1, \sigma_2\} + \Delta e + 1$
end if
if $* = -$ **then**
  [similar to the case $* = +$]
end if
if $* = \times$ **then**
  Compute $k \in \mathbb{Z}_{\geq 0}$ such that $\forall \vec{x} \in D.\ |f_1(\vec{x}) f_2(\vec{x})| \geq 2^{-1022-k}$
  return $\sigma_1 + \sigma_2 + k$
end if
if $* = /$ **then**
  return ∞
end if

rule constructs an abstraction of $e_1 \circledast e_2$ without applying the $(1+\varepsilon)$-property.

Next, we discuss Algorithm 1 in more detail. For brevity, the algorithm uses the notation f_i to represent an abstraction $\mathcal{A}_{i,\vec{\delta}}$ $(i = 1, 2)$; D represents $X \times [-\Delta_1, \Delta_1] \times \cdots \times [-\Delta_n, \Delta_n]$ and $\vec{x}$ represents $(x, \delta_1, \cdots, \delta_n)$. With this notation Algorithm 1 upper bounds $\sigma(f_1 * f_2)$, given upper bounds on $\sigma(f_1)$ and $\sigma(f_2)$. Formally, the algorithm meets the following specification:

Lemma 4.13. *Consider* $* \in \{+, -, \times, /\}$, $f_i : D \to \mathbb{R}$, *and* $\sigma_i \in \mathbb{Z}_{\geq 0}$ *for* $i \in \{1, 2\}$ *and* $D \subseteq \mathbb{R}^m$. *Let* $\sigma' = $ *bound-*$\sigma(*, f_1, f_2, \sigma_1, \sigma_2)$. *Then for any* $\vec{x} \in D$ *such that* $\forall i \in \{1, 2\}.\ \sigma(f_i(\vec{x})) \leq \sigma_i$,

$$|f_1(\vec{x}) * f_2(\vec{x})| \leq \max \mathbb{F} \qquad \Longrightarrow \qquad \sigma(f_1(\vec{x}) * f_2(\vec{x})) \leq \sigma'.$$

Algorithm 1 (conservatively) returns ∞ for division because the result of dividing two doubles often has no representation with finite binary digits (e.g., $1/5 = 2^{-3} \times 1.10011001\cdots_{(2)}$).

Algorithm 1 solves multiple optimization problems to compute the final bound. For example, to obtain Δe, we first compute $[em_i, eM_i]$ $(i = 1, 2)$, an interval bounding the range of $\text{expnt}(f_i(\vec{x}))$ over $\vec{x} \in D$, by solving optimization problems: $em_i = \text{expnt}(\min\{|f_i(\vec{x})| : \vec{x} \in D\})$ and $eM_i = \text{expnt}(\max\{|f_i(\vec{x})| : \vec{x} \in D\})$. Next, Δe is set to $\Delta e = \max\{eM_1 - em_2, eM_2 - em_1\}$. For another example, consider the **if** condition on line 5. The condition can be conservatively checked by deciding whether $(fm_1 \geq 0 \wedge fm_2 \geq 0) \vee (fM_1 \leq 0 \wedge fM_2 \leq 0)$ holds, where $fm_i = \min\{f_i(\vec{x}) : \vec{x} \in D\}$ and $fM_i = \max\{f_i(\vec{x}) : \vec{x} \in D\}$.

Now that the cache has been extended to store an upper bound of $\sigma(e)$, we need to extend all the previous rules accordingly and ensure that Theorem 4.6 holds. For instance, the rule R4 is extended to

$$\frac{(\mathcal{K}, \textit{bit-mask}(e_1, B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \qquad \mathcal{K}_1(e_1) = (\_, \_, \sigma_1)}{(\mathcal{K}, \textit{bit-mask}(e_1, B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \textit{fresh}(2^{-52+B}),\ \delta'' = \textit{fresh}(2^{-1074+B}) \\ \mathcal{A}'_{\vec{\delta}} = \textit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1+\delta') \boxplus \delta'') \\ \sigma' = \min\{\sigma_1, 53 - B\} \\ \mathcal{K}' = \mathcal{K}_1[\textit{bit-mask}(e_1, B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma')] \end{cases}} \text{R4}'$$

because $\textit{bit-mask}(e_1, B)$ masks out B least significant bits of e_1's significand. We can prove that Theorem 4.6 still holds, using the extended definition of a sound cache: a cache $\mathcal{K}$ is sound if for any $e \in \textit{dom}(\mathcal{K})$ with $\mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, b, \sigma)$, $\mathcal{A}_{\vec{\delta}}$ and b satisfy the previous condition and $\sigma(e) \leq \sigma$.

4.4.5 Refined $(1+\varepsilon)$-property

In some cases, the absolute error term δ' in Theorem 2.5 can be soundly removed according to the following *refined* $(1+\varepsilon)$-property:

Theorem 4.14 ([66, 109]). *Let $x, y \in \mathbb{F}$ and $* \in \{+, -, \times, /\}$. Assume $|x * y| \leq \max \mathbb{F}$. For any $* \in \{+, -\}$, and for any $* \in \{\times, /\}$ with either $x * y = 0$ or $|x * y| \geq 2^{-1022}$, we have*

$$x \circledast y = (x * y)(1 + \delta) \qquad \textit{for some } |\delta| < \varepsilon.$$

The theorem states that the absolute error term δ' is always unnecessary for addition and subtraction, and for the other operations it is unnecessary if the exact result of the operation is not in the subnormal range. The theorem is standard and follows from three properties of floating-point: $\text{ErrRel}(r, \text{fl}(r)) < \varepsilon$ for any $r \in \mathbb{R}$ not in the subnormal range (i.e., $0 < |r| < 2^{-1022}$), every double is a multiple of $\text{ulp}(0)$, and any multiple of $\text{ulp}(0)$ is a double if it is in the subnormal range.

To use Theorem 4.14 in constructing an abstraction of an expression e, we need to know whether e can evaluate to a number between 0 and $\pm 2^{-1022}$. To this end, define a function $\mu(e) > 0$ over

$$\frac{\begin{array}{lll} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma' = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu' = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & * \in \{+, -\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(\varepsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \circledast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \sigma'' = \min\{\sigma', 53\},\ \mu'' = \max\{\mathrm{fl}^-(\mu'), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma'', \mu'')] \end{cases}} \text{ R14}$$

$$\frac{\begin{array}{lll} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma' = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{\times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu' = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mu' \geq 2^{-1022} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(\varepsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \circledast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \sigma'' = \min\{\sigma', 53\},\ \mu'' = \max\{\mathrm{fl}^-(\mu'), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma'', \mu'')] \end{cases}} \text{ R15}$$

Figure 4.8: Rules for applying the refined $(1+\varepsilon)$-property

expressions, which denotes how close a non-zero evaluation result of e can be to 0:

$$\mu(e) \triangleq \min\{|\mathcal{E}(e)(x)| \neq 0 : x \in X \cap \mathbb{F}\}$$

where $\min \emptyset = \infty$. An important property related to $\mu(e)$ is the following lemma:

Lemma 4.15. *Let $\mu_1, \mu_2 > 0$. Consider any $d_1, d_2 \in \mathbb{F}$ such that $|d_i| \geq \mu_i$ $(i = 1, 2)$. Then*

$$d_1 + d_2 \neq 0 \qquad \Longrightarrow \qquad |d_1 + d_2| \geq \frac{1}{2}\mathrm{ulp}(\max\{\mu_1, \mu_2\}).$$

The lemma states that if the sum of two doubles is non-zero, then its magnitude cannot be smaller than some (small) number. The lemma holds because there is a finite gap between any two consecutive doubles. To illustrate an application of the lemma, consider $X = [0, 2]$ and $e = x \ominus 1$. Clearly e can evaluate to 0 (for an input $x = 1$). However, from the lemma we can conclude $\mu(e) \geq \frac{1}{2}\mathrm{ulp}(1) = 2^{-53}$, i.e., e can never evaluate to any value in $(0, 2^{-53})$.

The rules based on Theorem 4.14 are given in Figure 4.8. To keep track of $\mu(e)$, the cache $\mathcal{K}$ is extended to store a lower bound of $\mu(e)$ for each e. Both the rules R14 and R15 first compute a lower bound μ' on $\mu(e_1 * e_2)$ using Algorithm 2. The rule R15 then checks whether $\mu' \geq 2^{-1022}$; if the check passes, the rule constructs an abstraction of $e_1 \circledast e_2$ without adding an absolute error term δ'', based on Theorem 4.14. On the other hand, the rule R14 does not add the absolute error term δ'' regardless of whether $\mu' \geq 2^{-1022}$, also based on Theorem 4.14. Note that both rules set a lower bound of $\mu(e_1 \circledast e_2)$ to $\max\{\mathrm{fl}^-(\mu'), 2^{-1074}\}$ because the smallest positive double is 2^{-1074},

Algorithm 2 *bound-*$\mu(*, f_1, f_2, \mu_1, \mu_2)$

1: Let $D = dom(f_1)$
2: **if** $0 \notin \{f_1(\vec{x}) * f_2(\vec{x}) : \vec{x} \in D\}$ **then**
3: **return** $\mu' \in \mathbb{R}$ such that $0 < \mu' \leq \min\{|f_1(\vec{x}) * f_2(\vec{x})| : \vec{x} \in D\}$
4: **else**
5: **if** $* = \{+, -\}$ **then**
6: **return** $\min\{\frac{1}{2}\text{ulp}(\max\{\mu_1, \mu_2\}), \mu_1, \mu_2\}$
7: **else if** $* = \times$ **then**
8: **return** $\mu_1\mu_2$
9: **else if** $* = /$ **then**
10: Compute $M_2 \in \mathbb{R}$ such that $M_2 \geq \max\{|f_2(\vec{x})| : \vec{x} \in D\}$
11: **return** μ_1/M_2
12: **end if**
13: **end if**

where $\text{fl}^-(r) \triangleq \max\{d \in \mathbb{F} : d \leq r\}$.

Consider Algorithm 2. For brevity, the algorithm uses the notation of Algorithm 1 (e.g., f_i to represent an abstraction $\mathcal{A}_{i,\vec{\delta}}$). Given lower bounds on $\mu(f_1)$ and $\mu(f_2)$, the algorithm finds a lower bound on $\mu(f_1 * f_2)$ using Lemma 4.15:

Lemma 4.16. *Consider* $* \in \{+, -, \times, /\}$, $f_i : D \to \mathbb{R}$, *and* $\mu_i \in \mathbb{R}_{>0}$ *for* $i \in \{1, 2\}$ *and* $D \subseteq \mathbb{R}^m$. *Let* $\mu' =$ *bound-*$\mu(*, f_1, f_2, \mu_1, \mu_2)$. *Then for any* $\vec{x} \in D$ *such that* $\forall i \in \{1, 2\}$. $f_i(\vec{x}) = 0 \vee |f_i(\vec{x})| \geq \mu_i$,

$$f_1(\vec{x}) * f_2(\vec{x}) \neq 0 \qquad \Longrightarrow \qquad |f_1(\vec{x}) * f_2(\vec{x})| \geq \mu'.$$

Like Algorithm 1, Algorithm 2 requires solving optimization problems to obtain the final answer. For instance, the **if** condition on line 2 can be conservatively checked by deciding whether $0 \notin [fm, fM]$, where $fm = \min\{f_1(\vec{x}) * f_2(\vec{x}) : \vec{x} \in D\}$ and $fM = \max\{f_1(\vec{x}) * f_2(\vec{x}) : \vec{x} \in D\}$.

Since the cache has been extended to store a lower bound of $\mu(e)$, we need to extend all the previous rules and check Theorem 4.6 again. For example, the rules R1 and R2 are extended to

$$\frac{}{(\mathcal{K}, c) \rhd (\mathcal{K}[c \mapsto (c, \texttt{false}, \sigma(c), \mu(c))], c)}\ \text{R1}' \qquad \frac{}{(\mathcal{K}, x) \rhd (\mathcal{K}[x \mapsto (x, \texttt{false}, 53, \mu(x))], x)}\ \text{R2}'$$

We can prove that Theorem 4.6 is still true, by extending the definition of a sound cache: a cache $\mathcal{K}$ is sound if for any $e \in dom(\mathcal{K})$ with $\mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, b, \sigma, \mu)$, $\mathcal{A}_{\vec{\delta}}$, b, and σ satisfy the previous condition and $\mu(e) \geq \mu$. The complete rules appear in Appendix A.1.2.

4.4.6 Ulp Error Bound

The main goal of this chapter is to find an ulp error bound Θ_{ulp} of an expression e with respect to a mathematical specification $f(x)$, i.e., to find Θ_{ulp} such that

$$\text{ErrUlp}(f(x), \mathcal{E}(e)(x)) \leq \Theta_{\text{ulp}} \quad \text{for all } x \in X \cap \mathbb{F}. \tag{4.11}$$

To achieve the goal, we first construct a sound abstraction $\mathcal{A}_{\vec{\delta}}(x) = a(x) + \sum_i b_i(x)\delta_i$ of e by applying the rules discussed so far, and then compute a relative error bound Θ_{rel} of e with respect to $f(x)$ by solving the following optimization problems over $x \in X$:

$$\Theta_{\text{rel}} = \max_{x \in X} \left| \frac{f(x) - a(x)}{f(x)} \right| + \sum_i \max_{x \in X} \left| \frac{b_i(x)}{f(x)} \right| \cdot \Delta_i.$$

We can prove that Θ_{rel} is an upper bound on the relative error of e with respect to $f(x)$, i.e., $\text{ErrRel}(f(x), \mathcal{E}(e)(x)) \leq \Theta_{\text{rel}}$ for all $x \in X \cap \mathbb{F}$, using the triangle inequality and the soundness of $\mathcal{A}_{\vec{\delta}}(x)$. Finally, Theorem 2.7 enables us to obtain $\Theta_{\text{ulp}} = \Theta_{\text{rel}}/\varepsilon$ that satisfies Eq. (4.11).

However, the above approach often cannot prove an ulp error bound less than 1 ulp. To illustrate, consider $X = [1, 2]$, $e = x \oplus 1$, and $f(x) = x + 1$. Applying the rules R1′, R2′, and R14 to e gives an abstraction $\mathcal{A}_{\vec{\delta}}(x) = (x + 1) + (x + 1)\delta$ of e with $|\delta| \leq \varepsilon$, and we obtain $\Theta_{\text{rel}} = 0 + 1 \cdot \varepsilon = \varepsilon$ and $\Theta_{\text{ulp}} = \varepsilon/\varepsilon = 1$. But the tightest ulp error bound of e is in fact 0.5 ulps, as a single floating-point operation always has a rounding error of ≤ 0.5 ulps.

To obtain an ulp error bound less than 1 ulp, we use the following property about ulp errors which has been used to prove very precise ulp error bounds in [62]:

Theorem 4.17 ([62]). *Let* $r \in [-\max \mathbb{F}, \max \mathbb{F}]$, $d_1, d_2 \in \mathbb{F}$, *and* $* \in \{+, -, \times, /\}$. *Assume* $|d_1 * d_2| \leq \max \mathbb{F}$. *Then* $\text{ErrUlp}(r, d_1 * d_2) \leq 1$ *implies*

$$\text{ErrUlp}(r, d_1 \circledast d_2) \leq \text{ErrUlp}(r, d_1 * d_2) + \frac{1}{2}. \tag{4.12}$$

The theorem states that the ulp error of a floating-point operation is upper bounded by the ulp error of the corresponding exact operation plus 1/2. Note that the condition $\text{ErrUlp}(r, d_1 * d_2) \leq 1$ in the theorem is necessary; Eq. (4.12) may not hold if $\text{ErrUlp}(r, d_1 * d_2) > 1$. For the case when $\text{ErrUlp}(r, d_1 * d_2) > 1$, we can use the following similar statement: $\text{ErrUlp}(r, d_1 * d_2) \leq 2^{53}$ implies $\text{ErrUlp}(r, d_1 \circledast d_2) \leq \text{ErrUlp}(r, d_1 * d_2) + 1$.

Using Theorem 4.17, we compute an ulp error bound $\Theta_{\text{ulp,new}}$ of e tighter than Θ_{ulp} as follows. We first construct an abstraction $\mathcal{A}'_{\vec{\delta}}$ of e by applying the previous rules as before, but with the assumption that the last operation of e is exact. Then we compute an ulp error bound Θ'_{ulp} from $\mathcal{A}'_{\vec{\delta}}$ (not from $\mathcal{A}_{\vec{\delta}}$) by following the exactly same steps as above. Finally, we obtain a new, tighter ulp error bound of e by $\Theta_{\text{ulp,new}} = \Theta'_{\text{ulp}} + 1/2$ if $\Theta'_{\text{ulp}} \leq 1$, or by $\Theta_{\text{ulp,new}} = \Theta'_{\text{ulp}} + 1$ if $1 < \Theta'_{\text{ulp}} \leq 2^{53}$.

Using Theorem 4.17, we can show that $\Theta_{\text{ulp,new}}$ satisfies Eq. (4.11).

4.5 Implementation

We implement the techniques described in §4.3 and §4.4 using Mathematica 11.0.1. To solve optimization problems that appear in constructing abstractions and computing error bounds, we use Mathematica's built-in functions `MaxValue[···]` and `MinValue[···]` that find the global maximum/minimum of an objective function soundly using analytical optimization (not numerical optimization).

Most optimization problems occurring in our analysis involve abstractions, i.e., the minimization or the maximization of an abstraction $\mathcal{A}_{\vec{\delta}}$ or its magnitude $|\mathcal{A}_{\vec{\delta}}|$. However, these optimization problems are multi-variate and are difficult to solve in general. Hence, our implementation computes sound lower/upper bounds of these optimization objectives via Eq. (4.13)–(4.16), and uses these instead of the exact minimization/maximization results as in [90, 143].

$$\max_{x\in X,|\delta_i|\le\Delta_i} \mathcal{A}_{\vec{\delta}}(x) \le \max_{x\in X} a(x) + \sum_i \max_{x\in X} |b_i(x)| \cdot \Delta_i, \tag{4.13}$$

$$\min_{x\in X,|\delta_i|\le\Delta_i} \mathcal{A}_{\vec{\delta}}(x) \ge \min_{x\in X} a(x) - \sum_i \max_{x\in X} |b_i(x)| \cdot \Delta_i, \tag{4.14}$$

$$\max_{x\in X,|\delta_i|\le\Delta_i} |\mathcal{A}_{\vec{\delta}}(x)| \le \max_{x\in X} |a(x)| + \sum_i \max_{x\in X} |b_i(x)| \cdot \Delta_i, \tag{4.15}$$

$$\min_{x\in X,|\delta_i|\le\Delta_i} |\mathcal{A}_{\vec{\delta}}(x)| \ge \min_{x\in X} |a(x)| - \sum_i \max_{x\in X} |b_i(x)| \cdot \Delta_i. \tag{4.16}$$

In Eq. (4.13)–(4.16), the RHS represents a lower/upper bound of the LHS. As each RHS is a collection of uni-variate optimization problems, it is much easier to solve.

Our implementation checks that the evaluation of an expression e does not introduce $\pm\infty$ or NaNs, which arise from overflows or divide-by-zero errors. For proving the absence of overflows, the inequality $\max_{x,\vec{\delta}} |\mathcal{A'}_{\vec{\delta}}| \le \max \mathbb{F}$ is checked for every subexpression e' of e, where $\mathcal{A'}_{\vec{\delta}}$ is an abstraction of e'. For proving the absence of divide-by-zero errors, the inequality $0 \notin [\min_{x,\vec{\delta}} \mathcal{A'}_{\vec{\delta}}, \max_{x,\vec{\delta}} \mathcal{A'}_{\vec{\delta}}]$ is checked for every e' such that $e'' \oslash e'$ is a subexpression of e, where $\mathcal{A'}_{\vec{\delta}}$ is an abstraction of e'. Our implementation checks these conditions by solving additional optimization problems.

For some expressions e and input intervals $X = [l, r]$, our technique might produce imprecise results. In such scenarios, typically we can subdivide the interval into two (or more) subintervals $[l_1, r_1] \cup [l_2, r_2] = [l, r]$ such that separate analysis of the subintervals does yield tight bounds. This situation arises because the preconditions of different exactness properties are satisfied on different subintervals, but few or no such properties hold for the entire interval.

To prove tighter error bounds in such scenarios, our implementation works as follows. Let e be an expression, X be an input interval, and $\Theta_{\text{ulp,goal}}$ be an ulp error bound that we aim to prove (we use $\Theta_{\text{ulp,goal}} = 0.53$ in the evaluation). We first compute an ulp error bound Θ_{ulp} by applying

our technique to e and X. If $\Theta_{\text{ulp}} \leq \Theta_{\text{ulp,goal}}$ or if we are out of computation budget, return Θ_{ulp}. Otherwise, we bisect X into two subintervals X_1 and X_2, recursively compute an ulp error bound $\Theta_{\text{ulp,i}}$ of e over X_i $(i = 1, 2)$, and return the maximum of $\Theta_{\text{ulp},1}$ and $\Theta_{\text{ulp},2}$. Such approaches that bisect input intervals are well-known and are a part of existing commercial tools [43].

4.6 Case Studies

We evaluate our technique on the benchmarks of [90, 137] which consist of implementations (`exp`, `sin`, `tan`, and `log`) of four different transcendental functions (e^x, $sin\, x$, $tan\, x$, and $log\, x$). The code in `exp` is a custom implementation used in S3D [23], a combustion chemistry simulator; `sin`, `tan`, and `log` are taken from Intel® Math Library `libimf` which is Intel's implementation of the C library `math.h` and contains "highly optimized and very accurate mathematical functions."[4] All these implementations are loop-free programs, and have been written directly in x86 assembly for the best performance. We remark that analyzing these x86 implementations involves substantial engineering effort beyond what is described in this chapter or Chapter 3, as modeling the semantics of the more complex x86 instructions correctly and in detail is itself a significant undertaking and, at least so far, the overlap in instructions used among the benchmarks we have studied has not been as much as might be hoped.

We find an ulp error bound of each x86 implementation $P \in \{\texttt{exp}, \texttt{sin}, \texttt{tan}, \texttt{log}\}$ as follows. We first apply the technique from Chapter 3 that eliminates bit-level and integer arithmetic computations intermingled with floating-point operations using partial evaluation. The result is that for each P with an input interval X, the method yields k different expressions $e_1, \cdots, e_k$ in our core language (Figure 4.2), corresponding input intervals $X_1, \cdots, X_k$, and a (small) set H of individual doubles (typically $|H| < 250$) such that $\forall i \in \{1, \cdots, k\}. \forall x \in X_i \cap \mathbb{F}.\ P(x) = \mathcal{E}(e_i)(x)$ and $X \cap \mathbb{F} = \bigcup_{1 \leq i \leq k}(X_i \cap \mathbb{F}) \cup H$. Let us call $X_1, \cdots, X_k$ the *initial* input intervals from X. We then find an ulp error bound $\Theta_{\text{ulp},i}$ of e_i over X_i with respect to the exact mathematical function $f \in \{e^x, sin\, x, tan\, x, log\, x\}$ for each $1 \leq i \leq k$. Finally, we obtain an ulp error bound of P over X with respect to f by taking the maximum of $\max\{\Theta_{\text{ulp},i} : 1 \leq i \leq k\}$ and $\max\{\text{ErrUlp}(f(x), P(x)) : x \in H\}$. Although the procedure of Chapter 3 relies on procedures that may need manual intervention, these procedures have been automated for our benchmarks.

The results of applying our technique to these implementations are summarized in Table 4.1. Columns 1 and 2 represent P and X, and column 3 represents the proved ulp error bound of P over X with respect to f. Column 4 shows k, the number of the initial input intervals $X_1, \cdots, X_k$, while column 5 shows the number of disjoint intervals after bisecting these initial input intervals (§4.5). Column 6 shows the total wall clock time taken to obtain the bounds in column 3, and column 7 shows the maximum time taken to verify an initial input interval. In particular, the maximum time taken by our analysis (for a given expression e_i and an initial input interval X_i) is less than

[4] https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intel-c-compiler-classic-math-library.html (titled "Intel C++ Compiler Classic Math Library"; accessed on July 2023)

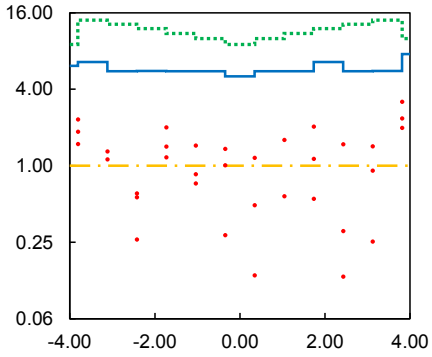
	Input interval	Ulp error bound	# of intervals before bisections	# of intervals after bisections	Verification time (m)	Max time per interval (s)
`exp`	$[-4, 4]$	7.552	13	13	0.52	2.5
`sin`	$[-\pi, \pi] \setminus (-2^{-252}, 2^{-252})$	0.530	66	142	68	446
`tan`	$[\frac{13}{128}, \frac{17\pi}{64})$	0.595	9	22	40	495
	$[\frac{17\pi}{64}, \frac{\pi}{2})$	13.33	8	8	10	81
`log`	$[2^{-1022}, \max \mathbb{F}]$	0.583	4.2×10^6	4.2×10^6	461 hrs†	24

Table 4.1: Summary of results. For each implementation (column 1) and for each input interval (column 2), column 3 shows the ulp error bound of the implementation over the input interval. Column 4 is the number of the initial input intervals from column 2, and column 5 is the number of disjoint intervals obtained by repeatedly bisecting the initial input intervals (until the ulp error bound of column 3 is obtained). Column 6 shows the total wall clock time taken to obtain the ulp error bound of column 3 (in minutes), and column 7 shows the maximum time taken to verify an initial input interval (in seconds). Here † denotes that we used 16 instances of Mathematica in parallel; by default we run only one instance of Mathematica.

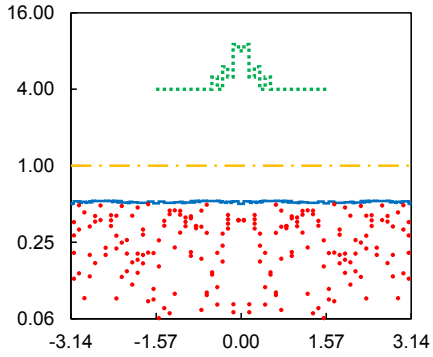
10 minutes. This table is discussed in detail in §4.6.1–§4.6.4.

Figure 4.9 shows our results graphically. For each graph, the x-axis represents the input values and the y-axis represents the bounds on ulp error (in log scale) between an implementation and the exact mathematical function. The ulp error bounds we prove are shown as solid blue lines, while the ulp error bounds from Chapter 3 are shown by dotted green lines. Dashed yellow lines in (b)-(e) denote the one ulp bound that we must prove to verify the correctness of `sin`, `tan`, and `log`. The actual ulp errors on concrete inputs (from the set H and random samples) are shown by the red dots. These ulp errors are calculated by comparing the output of an implementation with the exact result computed by Mathematica.

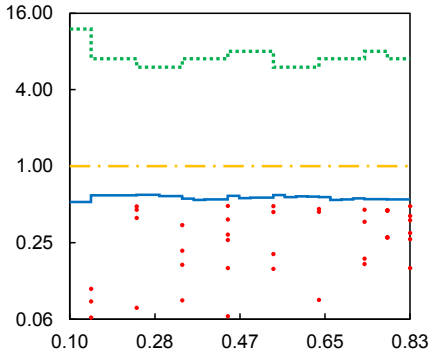
In these graphs, lower bounds are tighter and the bounds proven by our analysis are much better than that of Chapter 3 across the board. There are some inputs that we analyze but Chapter 3 does not, e.g., `sin` near $\pm\pi$ and `log`(x) for $x \geq 4$. For such inputs, the green dotted lines are missing. In (b), (c), and (e), our bounds are below one ulp and we successfully establish the correctness of these implementations. Except for (d), the bounds we infer are tight and the observed ulp errors on concrete inputs are close to the statically inferred bounds. In (d), although our bounds are not tight enough to establish correctness, they are still sound. However, the bounds from Chapter 3 in this graph are obtained by numerical optimization (as opposed to analytical optimization) and are not guaranteed to be sound. Finally, although we only compare our approach with Chapter 3 in Figure 4.9, other generic tools for floating-point verification such as [35, 43, 143] would meet a similar fate due to the absence of the relevant exactness results in their analyses.



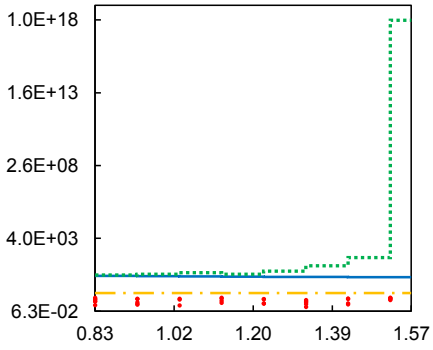
(a) `exp` on $[-4, 4]$



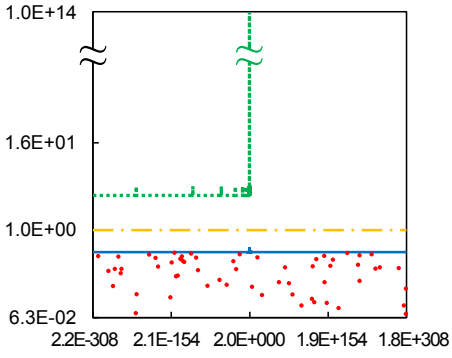
(b) `sin` on $[-\pi, \pi] \setminus (-2^{-252}, 2^{-252})$



(c) `tan` on $[\frac{13}{128}, \frac{17\pi}{64})$



(d) `tan` on $[\frac{17\pi}{64}, \frac{\pi}{2})$



(e) `log` on $[2^{-1022}, \max \mathbb{F}]$

Figure 4.9: Each graph shows the ulp error (y-axis in log scale) of each implementation over an input interval (x-axis). Solid blue lines represent our ulp error bounds (Table 4.1), dotted green lines represent the ulp error bounds from Chapter 3, and dashed yellow lines represent 1 ulp. Red dots represent actual ulp errors on concrete inputs. The x-axis in (a)-(d) is linear. Because of the large input interval, x-axis in (e) is log-scale.

4.6.1 The `exp` Implementation

For the `exp` implementation, our technique finds an error bound of 7.552 ulps over the interval $[-4, 4]$ in 31 seconds. An important step for proving this error bound is the application of rule R8/R9. The error bound of 7.552 ulps is larger than Intel's implementations as the developer sacrificed precision for performance in this custom implementation; even assuming every floating-point operation in `exp` is exact, the ulp error bound is 4.06 ulps over $[-4, 4]$, and indeed we typically observe error of 3-4 ulps on concrete inputs. The documentation of `exp` specifies that the implementation is supposed to compute e^x with "small errors" for inputs between -2.6 and 0.12, and we have successfully quantified the maximum precision loss formally.

4.6.2 The `sin` Implementation

In sharp contrast to custom implementations such as `exp`, standard math libraries such as `libimf` claim that the maximum precision loss is below one ulp. For `sin`, our technique finds an error bound of 0.530 ulps over the interval $X = [-\pi, \pi] \setminus (-2^{-252}, 2^{-252})$ in 68 minutes. We exclude the interval $(-2^{-252}, 2^{-252})$ because the `sin` implementation we analyze is executed only for $|x| \in [2^{-252}, 90112)$. For inputs outside this range, different implementations are used. To prove the error bound, rules R8 and R9 (related to Sterbenz's theorem, §4.4.2), rules R10, R11, and R12 (related to Dekker's theorem, §4.4.3), and rules R14 and R15 (related to the refined $(1+\varepsilon)$-property, §4.4.5) are crucial.

Proving the error bound of `sin` shown in Table 4.1 requires us to analyze `sin` over 142 disjoint intervals, the result of repeatedly bisecting the 66 initial input intervals. In particular, to verify `sin` over $X_{66} = [63\pi/64, \pi]$, we need to repeatedly bisect X_{66} to have 13 disjoint subintervals $[63\pi/64, y_1)$, $[y_1, y_2)$, $\cdots$, $[y_{12}, \pi]$, where $y_0 = 63\pi/64$, $y_{13} = \pi$, and $y_i = (y_{i-1} + y_{13})/2$ $(i = 1, \cdots, 12)$. We require many subintervals because the antecedents of the rules R11 and R12 are valid only over small intervals.

4.6.3 The `tan` Implementation

For the `tan` implementation, our technique finds an error bound of 0.595 and 13.33 ulps over the intervals $[13/128, 17\pi/64)$ and $[17\pi/64, \pi/2)$, respectively, in 50 minutes. We exclude the interval $[0, 13/128)$ because our benchmark implementation is supposed to compute $tan\, x$ precisely only for $|x| \in [13/128, 12800)$. To obtain the error bounds, it is crucial to apply all the rules used in verifying `sin` multiple times (R8, R9, R10, R11, R12, R14, and R15). Additionally, the rules R4′ and R13 (related to $\sigma(\cdot)$, §4.4.4) are used to precisely abstract bit-mask operations (which are absent in `sin`).

For `tan` over the input interval $X = [17\pi/64, \pi/2)$, we obtain the error bound of 13.33 ulps. The main culprit is the requirement that, though our abstractions can be non-linear in x, they must be linear in each δ variable. For example, consider the following expressions that appear in `tan`:

$$e_1 = \textit{bit-mask}(e_0, 18), \quad e_2 = \textit{bit-mask}(1 \oslash e_1, 35), \quad e = 1 \ominus e_1 \otimes e_2.$$

For simplicity, assume that the expression e_0 has no rounding error, i.e., $a(x)$ is a sound abstraction of e_0, and we suppress any δ variable with $|\delta| \leq \varepsilon$. First, by a precise manual analysis, we show that e computes the rounding error of the bit-mask operation in e_2: from Lemma 4.3, $g_1(x, \vec{\delta}) = a(x)(1+\delta_1)$ and $g_2(x, \vec{\delta}) = \frac{1}{a(x)(1+\delta_1)}(1+\delta_2)$ are over-approximations of e_1 and e_2, where $|\delta_1| \leq 2^{-34}$ and $|\delta_2| \leq 2^{-17}$; thus the function

$$g(x, \vec{\delta}) = 1 - g_1(x, \vec{\delta}) \times g_2(x, \vec{\delta}) = 1 - a(x)(1+\delta_1) \times \frac{1}{a(x)(1+\delta_1)}(1+\delta_2) = -\delta_2$$

is an over-approximation of e (the operations $\ominus$ and $\otimes$ of e are exact by Theorem 4.7 and 4.10). Note that the term $(1+\delta_1)$ in g_1 and g_2 is exactly canceled out in computing g. On the other hand, the analysis of e using our abstractions proceeds as follows. First, $\mathcal{A}_{1,\vec{\delta}}(x) = g_1(x, \vec{\delta})$ is a sound abstraction of e_1. However, g_2 is non-linear in δ_1; by the rules R3 and R4, $\mathcal{A}_{2,\vec{\delta}}(x) = \frac{1}{a(x)}(1+\delta_1')(1+\delta_2)$ is a sound abstraction of e_2, where $|\delta_1'| \leq 2^{-34}$. Given $\mathcal{A}_{1,\vec{\delta}}$ and $\mathcal{A}_{2,\vec{\delta}}$, the following is a sound abstraction of e:

$$\mathcal{A}_{\vec{\delta}}(x) = 1 - \mathcal{A}_{1,\vec{\delta}}(x) \times \mathcal{A}_{2,\vec{\delta}}(x) = 1 - a(x)(1+\delta_1) \times \frac{1}{a(x)}(1+\delta_1')(1+\delta_2) \approx -\delta_2 - (\delta_1 + \delta_1' + \delta_{1,2} + \delta_{1,2}')$$

where $|\delta_{1,2}| \leq 2^{-51}$ and $|\delta_{1,2}'| \leq 2^{-51}$. These additional δ terms (compared to g) contribute significantly to the error bound of 13.33 ulps for `tan` over X (since $\Delta_{1,2}$ and $\Delta_{1,2}'$ are $2^{-51} = 4\varepsilon$).

4.6.4 The `log` Implementation

For the `log` implementation, we apply our technique to its complete input interval $X = [2^{-1022}, \max \mathbb{F}]$ and obtain an error bound of 0.583 ulps. The error bound implies that `log` mostly returns the nearest double to the mathematically exact results. This verification requires all the rules presented in §4.4. For `log`, we used 16 instances of Mathematica in parallel and required 461 hours of wall clock time to verify all four million cases. We note that this verification is highly parallelizable as analyses over distinct input intervals can be run independently. From Table 4.1, we observe that the average time and the maximum time taken to verify `log` over each initial input interval are 6 seconds and 24 seconds, respectively.

4.7 Related Work

An obvious approach to obtain a provably correct mathematical library involves using verified routines for arbitrary precision arithmetic during computations and then rounding the results (e.g., the `libmcr` library by Sun). However, the resulting implementations have vastly inferior performance compared to the math libraries that exclusively use 64-bit arithmetic. Libraries such as CRLibm aim to keep the maximum error below 0.5 ulps while maintaining performance. The correctness is ensured by a mixture of "pen-and-paper" proofs [34] and machine-checkable proofs in Gappa [39, 41]

and Coq [102, 128]. The tightness of error bounds and the peculiar structure of rounding errors coupled with optimization tricks make such high performance libraries difficult to verify. Furthermore, industry standard libraries such as Intel's math library lose precision to have better performance. Harrison proved a tight error bound of an algorithm for computing $sin\, x$ for $|x| \leq 2^{63}$ (which is slightly different from Intel's `sin` implementation) in HOL Light [62]. In general, the `libimf` documentation claims, without any formal proofs, that the maximum error in the routines is always below one ulp[5]. In this chapter, we have validated this claim fully automatically for `log` (for all valid inputs), `sin` (for inputs between $-\pi$ and π), and `tan` (for inputs between $13/128$ and $17\pi/64$). We are unaware of any prior technique that can prove such tight bounds for math libraries automatically.

The existing work closest to this chapter is Chapter 3 (which was published as [90]), based on which this chapter is developed. In Chapter 3, error bounds were proven by first decomposing the `math.h` implementations into simple expressions and then proving error bounds of those expressions using Mathematica. The primary research contribution of Chapter 3 is the first step which performs the decomposition, and the chapter used a standard error analysis in the second step. The current chapter reuses the decomposition step and adds a novel automatic error analysis that leverages results about exact floating-point arithmetic systematically. No prior analysis, including the one in Chapter 3 and those mentioned below, uses all the exactness results we discussed in §4.4 and all of these would fail to prove that the error bounds are below one ulp for the benchmarks we consider.

Automatic tools that can provide formal guarantees on error bounds include Astree [15, 106], Fluctuat [43, 54], Gappa [39], MathSAT [60], Rosa [35, 36], FPTaylor [143, 144], Real2Float [99], and Satire [37]. In contrast to [90], none of these provide support to bound the error between expressions in our core language and exact transcendentals. For example, these techniques do not handle bit-masking. Although some of these can handle some exactness results about floating-point, they do not provide a general framework like ours. For example, Gappa automatically applies some of the exactness results described in §4.4, but not all of them (e.g., Dekker's theorem (§4.4.3) and the refined $(1+\varepsilon)$-property (§4.4.5) for $* \in \{\times, /\}$). Moreover, Gappa uses interval arithmetic to soundly bound the max/min of some expressions, when checking preconditions of exactness results. Interval arithmetic can often cause imprecision (because it does not preserve dependencies between variables) and fail to discharge the preconditions; our optimization-based technique is more precise.

There are techniques that check whether two floating-point programs produce exactly equivalent results [30, 112]. These do not produce any bound on the maximum deviation between the implementations. Debugging tools such as [6, 11, 25, 87] are complementary to this chapter and can help detect incorrect implementations. In particular, [50] use optimization to find inputs that achieve high branch coverage. Other techniques that provide statistical (and not formal) guarantees include [107, 111, 137].

[5] See `max-error=1.0` at https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/fimf-precision-qimf-precision.html (titled "fimf-precision, Qimf-precision"; accessed on July 2023)

4.8 Conclusion

A major source of imprecision in generic verification techniques for floating-point stems from modeling every floating-point operation as having a rounding error about which worst-case assumptions must be made. However, floating-point operations do not always introduce rounding errors. In this chapter, we identify floating-point computations that are exact and thus avoid introducing unneeded potential rounding errors into the modeling of those computations. Our main technical contribution is a reduction from the problem of checking whether an operation is exact to a set of mathematical optimization problems that are solved soundly and automatically by off-the-shelf computer algebra systems. We introduce transformations, also involving optimization problems, to control the size of our abstractions while maintaining precision. Our analysis successfully proves the correctness of x86 implementations from an industry standard math library.

Chapter 5

Correctness of Automatic Differentiation

5.1 Introduction

Forward- and reverse-mode automatic differentiation (AD) are popular algorithms for computing the derivative of a function represented by a program [56]. Diverse practical systems for AD have been developed for general-purpose programs [9, 65, 98, 120, 129, 141, 154], and particularly for machine-learning programs [12, 31, 76, 140, 149, 152], including TensorFlow [3], PyTorch [117], and JAX [49]. The development of such AD systems has been a driving force of the rapid advances in deep learning (and machine learning in general) in the past 10 years [10, 89, 138].

Recently, the correctness of AD has been actively studied for various types of programs. For programs that only use differentiable functions, AD is correct *everywhere*, i.e., it computes the derivative of a given program at all inputs [1, 7, 20, 47, 70, 83, 126, 142, 151]. On the other hand, for programs that use non-differentiable functions (e.g., ReLU[1]), AD can be incorrect at some inputs [77].

There are two cases where AD is incorrect. The first case is when the function f represented by a given program is differentiable at some x, but AD returns a value different from the derivative of f at x. For instance, consider a program[2] that represents the identity function, defined as $\mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$. If AD uses zero as a "derivative" of ReLU at $x = 0$, as is standard (e.g., in TensorFlow and PyTorch), it returns zero for this program at $x = 0$ while the true derivative is one. The second case is when f is non-differentiable at some x, but AD does not return a generalized notion of derivative (e.g., Clarke subdifferential) of f at x. For example, $\mathrm{ReLU}(x) - \frac{1}{2}\mathrm{ReLU}(-x)$ represents a function that is non-differentiable at $x = 0$ with the Clarke subdifferential $[\frac{1}{2}, 1]$, but AD outputs 0 at $x = 0$.

[1] $\mathrm{ReLU}(x) \triangleq \max\{x, 0\}$.
[2] It appeared in [77].

Although AD can be incorrect, recent works show that for a large class of programs using non-differentiable functions, AD is correct *almost everywhere*, i.e., it is incorrect at most on a Lebesgue measure-zero subset of the input domain of a program [17, 18, 71, 92, 101].

These prior works, however, have a limitation: they consider AD over the real numbers, but in practice, inputs to a program are always *machine-representable numbers* such as 32-bit floating-point numbers. Since the set of machine-representable numbers is countable (and usually finite), it is always a Lebesgue measure-zero subset of the real numbers. Hence, AD could be incorrect on *all* machine-representable inputs according to prior works, and this is indeed possible. Consider a program[3] for a function from $\mathbb{R}$ to $\mathbb{R}$, defined as

$$\sum_{c \in \mathbb{M}} \Big[\lambda x + \Big(\frac{1}{|\mathbb{M}|} - \lambda \Big) \Big(\mathrm{ReLU}(x - c) - \mathrm{ReLU}(-x + c) \Big) \Big],$$

where $\mathbb{M} \subseteq \mathbb{R}$ is a finite set of machine-representable numbers and $\lambda \in \mathbb{R} \setminus \{1\}$ is an arbitrary constant. Then, the program represents the affine function $x \mapsto x + a$ for $a = (\lambda - \frac{1}{|\mathbb{M}|}) \times \sum_{c \in \mathbb{M}} c$, but AD incorrectly computes its derivative at any $x \in \mathbb{M}$ as λ (the arbitrarily chosen value) if zero is used as a "derivative" of ReLU at 0 as before.[4]

Given these observations, we raise the following questions: for a program that represents a neural network, at which machine-representable inputs to the program (i.e., parameters to the network) can AD be incorrect, and how many such inputs can there be? In this chapter, we tackle these questions and present the first theoretical results. In particular, we study the two sets of machine-representable parameters of a neural network on which AD can be incorrect: the *incorrect set*, on which the network is differentiable but AD does not compute its derivative, and the *non-differentiable set*, on which the network is non-differentiable.

Summary of results. We focus on neural networks consisting of alternating analytic pre-activation functions (e.g., fully-connected and convolution layers) and pointwise continuous activation functions (e.g., ReLU and Sigmoid). The first set of our results (§5.3) is for such networks *with bias parameters* at every layer, and is summarized as follows.

- We prove that the incorrect set is *always empty*, not only over machine-representable parameters but also over real-valued ones. To our knowledge, this is the first result showing that the incorrect set can be empty for a class of neural networks using possibly non-differentiable functions; prior works only bounded the measure of this set.

- On the other hand, the non-differentiable set can be non-empty. We give a tight bound on its density over all machine-representable parameters, which has the form $n/|\mathbb{M}|$ where n is the *total number of non-differentiable points* in activation functions. This result implies that in

[3] Inspired by [18, 101].

[4] We can even make AD return different values at different $x \in \mathbb{M}$, by using a different λ_i for each $c_i \in \mathbb{M}$. Similarly, we can also construct a program such that at all machine-representable numbers $\mathbb{M}$, the program is non-differentiable and AD returns arbitrary values.

practice, the non-differentiable set often has a low density, especially if we use high-precision parameters (e.g., use 32-bit floating-point numbers for $\mathbb{M}$, where $|\mathbb{M}| \approx 2^{32}$).

- To better describe the non-differentiable set, we provide a simple, easily verifiable *necessary and sufficient condition* for a parameter to be in the non-differentiable set. Given that deciding the non-differentiability of a neural network is NP-hard in general [19], our result is surprising: having bias parameters is sufficient to efficiently decide the non-differentiability.

- Given that the non-differentiable set can be non-empty, a natural question arises: what does AD compute on this set? We prove that AD *always computes a Clarke subderivative* (a generalized derivative) even on the non-differentiable set. That is, AD is an efficient algorithm for computing a Clarke subderivative in this case.

The second set of our results (§5.4) extends the above results to neural networks possibly *without bias parameters* at some layers, and is summarized as follows.

- As we observed in the $\mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$ example, the incorrect set can be non-empty in this case. Thus, we prove tight bounds on the density of both the incorrect and non-differentiable sets, which have the form $n'/|\mathbb{M}|$ where n' is linear in the total number of non-differentiable points in activation functions as well as the total number of boundary points in activation functions' zero sets.

- We provide simple, easily verifiable sufficient conditions on parameters under which AD computes the standard derivative or a Clarke subderivative.

Our theoretical results carry two main practical implications: AD for neural networks is correct on most machine-representable parameters, and it is correct more often with bias parameters. For networks with bias parameters at all layers, our results further provide an exact characterization of when AD is correct and what it computes.

We remark that many of our results, especially all the results not about the density of certain sets, hold not only for machine-representable parameters but also for real-valued ones. On the other hand, our results may not be directly applicable to neural networks with non-analytic pre-activation functions or non-pointwise activation functions; we discuss such limitations in §5.6.

Organization. We first introduce notation and the problem setup (§5.2). We then present our main results for neural networks with bias parameters (§5.3) and extend them to neural networks possibly without bias parameters (§5.4). We conclude this chapter with related work and discussion (§5.5–5.7).

5.2 Preliminaries

5.2.1 Notation and Definitions

We use the following notation and definitions. Let $\mathbb{N}$ and $\mathbb{R}$ be the sets of positive integers and real numbers, respectively. For $n \in \mathbb{N}$, we use $[n] \triangleq \{1, 2, \ldots, n\}$ and $\vec{0}_n \triangleq (0, \ldots, 0) \in \mathbb{R}^n$, and often drop n from $\vec{0}_n$ when the subscript is clear from context. For $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, we use $x_{-i} \triangleq (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. We call $A \subseteq \mathbb{R}$ an *interval* if it is $[a, b]$, $[a, b)$, $(a, b]$, or (a, b) for some $a, b \in \mathbb{R} \cup \{\pm\infty\}$. For $A \subseteq \mathbb{R}^n$, $\mathbf{1}_A : \mathbb{R}^n \to \{0, 1\}$ denotes the indicator function of A. We say that $f : \mathbb{R}^n \to \mathbb{R}^m$ is *analytic* if it is infinitely differentiable and its Taylor series at any $x \in \mathbb{R}^n$ converges to f on some neighborhood of x. For any $f : \mathbb{R}^n \to \mathbb{R}^m$,

$$Df : \mathbb{R}^n \to \mathbb{R}^{m \times n} \cup \{\bot\}$$

denotes the standard derivative of f, where $f(x) = \bot$ denotes that f is non-differentiable at x. Lastly, for $f : \mathbb{R} \to \mathbb{R}$,

$$\mathsf{ndf}(f) \triangleq \{x \in \mathbb{R} \mid f \text{ is non-differentiable at } x\} \qquad \text{and} \qquad \mathsf{bdz}(f) \triangleq bd(\{x \in \mathbb{R} \mid f(x) = 0\})$$

denote the set of non-differentiable points of f and the boundary of the zero set of f, respectively.

5.2.2 Neural Networks

We define a neural network as follows. Given the number of layers $L \in \mathbb{N}$, let $N_0 \in \mathbb{N}$ be the dimension of input data, $N_l \in \mathbb{N}$ and $W_l \in \mathbb{N} \cup \{0\}$ be the number of neurons and the number of parameters at layer $l \in [L]$, and $N \triangleq N_1 + \cdots + N_L$ and $W \triangleq W_1 + \cdots + W_L$. Further, for each $l \in [L]$, let $\tau_l : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}^{N_l}$ be an analytic *pre-activation function* and $\sigma_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$ be a pointwise, continuous *activation function*, i.e.,

$$\sigma_l(x_1, \ldots, x_{N_l}) \triangleq \big(\sigma_{l,1}(x_1), \ldots, \sigma_{l,N_l}(x_{N_l})\big)$$

for some continuous $\sigma_{l,i} : \mathbb{R} \to \mathbb{R}$. Under this setup, we define a neural network as a function of model parameters: given input data $c \in \mathbb{R}^{N_0}$, a *neural network* $z_L(\,\cdot\,; c) : \mathbb{R}^W \to \mathbb{R}^{N_L}$ is defined as

$$z_L(w; c) \triangleq (\sigma_L \circ \tau_L^{\langle w_L \rangle} \circ \cdots \circ \sigma_1 \circ \tau_1^{\langle w_1 \rangle})(c), \tag{5.1}$$

where $w \triangleq (w_1, \ldots, w_L)$, $w_l \triangleq (w_{l,1}, \ldots, w_{l,W_l}) \in \mathbb{R}^{W_l}$, and $\tau_l^{\langle w_l \rangle}(x) \triangleq \tau_l(x, w_l)$. We say such z_L *has L layers, N neurons, and W parameters.*

We next define the *activation neurons* $z_l(\,\cdot\,; c) : \mathbb{R}^W \to \mathbb{R}^{N_l}$ and the *pre-activation values*

$y_l(\,\cdot\,; c) : \mathbb{R}^W \to \mathbb{R}^{N_l}$ at layer $l \in [L]$, as we defined z_L above:

$$z_l(w; c) \triangleq (\sigma_l \circ \tau_l^{\langle w_l \rangle} \circ \cdots \circ \sigma_1 \circ \tau_1^{\langle w_1 \rangle})(c), \qquad y_l(w; c) \triangleq \tau_l^{\langle w_l \rangle}(z_{l-1}(w; c)),$$

where $z_0(w; c) \triangleq c$. Since the input data c is fixed while we compute the derivative of z_L with respect to w (e.g., in order to train z_L), we often omit c and simply write $z_l(w)$ and $y_l(w)$ to denote $z_l(w; c)$ and $y_l(w; c)$, respectively.

For the set of all indices of neurons

$$\mathsf{Idx} \triangleq \{(l, i) \mid l \in [L], i \in [N_l]\}$$

and for each $(l, i) \in \mathsf{Idx}$, we use $y_{l,i}, z_{l,i} : \mathbb{R}^W \to \mathbb{R}$ and $\tau_{l,i} : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}$ to denote the functions that take only the i-th output component of y_l, z_l, and τ_l, respectively. Note that we defined $\sigma_{l,i}$ above in a slightly different way: its domain is not $\mathbb{R}^{N_l}$ (i.e., the domain of σ_l) but $\mathbb{R}$.

Finally, we introduce the notion of piecewise-analytic[5] to consider possibly non-differentiable activation functions.

Definition 5.1. A function $f : \mathbb{R} \to \mathbb{R}$ is *piecewise-analytic* if there exist $n \in \mathbb{N}$, a partition $\{A_i\}_{i \in [n]}$ of $\mathbb{R}$ consisting of non-empty intervals, and analytic functions $\{f_i : \mathbb{R} \to \mathbb{R}\}_{i \in [n]}$ such that $f = f_i$ on A_i for all $i \in [n]$.

Assumption. $\sigma_{l,i}$ is piecewise-analytic for all $(l, i) \in \mathsf{Idx}$.

The class of piecewise-analytic functions includes not only all analytic functions but also many non-differentiable functions widely used in neural networks such as ReLU, LeakyReLU, and HardSigmoid. Hence, our definition of neural networks includes a rich class of practical networks: τ_l can be any analytic function (e.g., a fully-connected, convolution, or normalization layer), and σ_l can be any pointwise continuous and piecewise-analytic function (e.g., ReLU, LeakyReLU, or HardSigmoid).

In practice, we often apply AD to the composition of a neural network z_L and a loss function ℓ (e.g., Softmax followed by CrossEntropy), to compute the derivative of the loss value of z_L with respect to its parameters. We emphasize that all of our results except for lower bounds (i.e., Theorems 5.8, 5.13 and 5.15) continue to hold even if we replace z_L in their conclusions by $\ell \circ z_L$ for any analytic $\ell : \mathbb{R}^{N_L} \to \mathbb{R}^m$. For simplicity, however, we state our results only for z_L and not for $\ell \circ z_L$.

5.2.3 Automatic Differentiation

Given a program that represents a neural network z_L as in Eq. (5.1), AD essentially computes the function

$$D^{\mathtt{AD}} z_L : \mathbb{R}^W \to \mathbb{R}^{N_L \times W}$$

[5] It is inspired by the notion of PAP in [92].

by applying the chain rule of differentiation to Eq. (5.1). That is, $D^{\mathtt{AD}}z_L$ is defined as the product of $D^{\mathtt{AD}}\tau_{l,i}$ and $D^{\mathtt{AD}}\sigma_{l,i}$ for $(l,i) \in \mathsf{Idx}$, where $D^{\mathtt{AD}}\tau_{l,i} : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}^{1\times(N_{l-1}+W_l)}$ and $D^{\mathtt{AD}}\sigma_{l,i} : \mathbb{R}^{N_l} \to \mathbb{R}^{1\times N_l}$ denote the "derivatives" of $\tau_{l,i}$ and $\sigma_{l,i}$ that AD uses in its computation (see Appendix B.1.3 for more details). Here $D^{\mathtt{AD}}z_L$, $D^{\mathtt{AD}}\tau_{l,i}$, and $D^{\mathtt{AD}}\sigma_{l,i}$ can be different from the standard derivatives Dz_L, $D\tau_{l,i}$, and $D\sigma_{l,i}$, partly because the former never return $\bot$ even at non-differentiable points while the latter always return $\bot$ at those points. We note that $D^{\mathtt{AD}}z_L$ expresses what practical AD systems (e.g., TensorFlow, PyTorch) essentially compute in *both* forward-mode and reverse-mode.

By definition, the output $D^{\mathtt{AD}}z_L$ of AD depends on the choice of $D^{\mathtt{AD}}\tau_{l,i}$ and $D^{\mathtt{AD}}\sigma_{l,i}$. To focus on the standard choices made by practical AD systems, we introduce the notion of an extended derivative.

Definition 5.2. A function $g : \mathbb{R}^n \to \mathbb{R}^{m\times n}$ is an *extended derivative* of $f : \mathbb{R}^n \to \mathbb{R}^m$ if for all $x \in \mathbb{R}^n$ with $Df(x) \neq \bot$, it holds that $g(x) = Df(x)$.

Assumption. $D^{\mathtt{AD}}f$ is an extended derivative of f for all $f \in \{\tau_{l,i}, \sigma_{l,i} \mid (l,i) \in \mathsf{Idx}\}$.

We note that a differentiable function f has a unique extended derivative which is the standard derivative Df of f. In contrast, a non-differentiable function f has (uncountably) many extended derivatives: e.g., $\mathbf{1}_{(0,\infty)} + c \cdot \mathbf{1}_{\{0\}}$ is an extended derivative of ReLU for all $c \in \mathbb{R}$, where $\mathbf{1}_A$ denotes the indicator function of a set A.

Among many extended derivatives, some of them are used more frequently in practice, which we characterize as consistency.

Definition 5.3. For $f : \mathbb{R}^n \to \mathbb{R}^m$, an extended derivative g of f is *consistent* if for all $x \in \mathbb{R}^n$ with $Df(x) = \bot$, it holds that $g(x) = \lim_{k\to\infty} Df(x_k)$ for some $x_k \to x$.[6]

For instance, $\mathbf{1}_{(0,\infty)}$ and $\mathbf{1}_{[0,\infty)}$ are consistent extended derivatives of ReLU but $\mathbf{1}_{(0,\infty)} + c \cdot \mathbf{1}_{\{0\}}$ is not for all $c \in \mathbb{R} \setminus \{0, 1\}$; among them, $D^{\mathtt{AD}}\mathrm{ReLU} = \mathbf{1}_{(0,\infty)}$ is typically used by popular AD systems (e.g., TensorFlow and PyTorch). Although $D^{\mathtt{AD}}f$ is usually consistent in practice, we do not assume it by default (and explicitly assume it only when necessary) to make our results as general as possible, and to study whether the values of extended derivatives at non-differentiable points matter to AD.

5.2.4 Incorrect and Non-Differentiable Sets

In practice, the parameters of a neural network cannot be arbitrary real numbers (as machines cannot represent them), but can only be machine-representable numbers $\mathbb{M} \subseteq \mathbb{R}$, where $\mathbb{M}$ is often chosen as the set of all 32-bit floating-point numbers. To this end, we consider

$$\Omega \triangleq \mathbb{M}^W \subseteq \mathbb{R}^W,$$

[6] Any consistent extended derivative of f is an element of the so-called Bouligand subdifferential of f [33]. But the converse does not hold in general.

the set of parameters that a neural network $z_L : \mathbb{R}^W \to \mathbb{R}^{N_L}$ can take in practice. We assume that $\mathbb{M}$ is an arbitrary finite subset of $\mathbb{R}$ throughout this chapter; e.g., it can be the set of n-bit floating-point (or fixed-point) numbers for any $n \in \mathbb{N}$.

To better understand the correctness of AD, we study the following two disjoint subsets of Ω on which AD can return an incorrect output.

Definition 5.4. For a neural network z_L, define the *incorrect set* and the *non-differentiable set* of z_L as

$$\mathsf{inc}_\Omega(z_L) \triangleq \{w \in \Omega \mid Dz_L(w) \neq \bot,\ D^{\mathtt{AD}} z_L(w) \neq Dz_L(w)\},$$
$$\mathsf{ndf}_\Omega(z_L) \triangleq \{w \in \Omega \mid Dz_L(w) = \bot\}.$$

These two sets correspond to the two cases when AD can be incorrect: on the incorrect set $\mathsf{inc}_\Omega(z_L)$, z_L is differentiable but AD does not compute its standard derivative; on the non-differentiable set $\mathsf{ndf}_\Omega(z_L)$, z_L is non-differentiable and AD may not compute a generalized notion of derivative (e.g., Clarke subdifferential). Here $\mathsf{ndf}_\Omega(z_L) \subseteq \Omega$ is different from $\mathsf{ndf}(f) \subseteq \mathbb{R}$, which was defined in §5.2.1 for $f : \mathbb{R} \to \mathbb{R}$.

5.3 Neural Networks with Bias Parameters

Our main objective is to understand the incorrect and non-differentiable sets. In particular, we focus on neural networks with bias parameters (defined below) in this section and consider more general neural networks in §5.4. For the former class of neural networks, we characterize the incorrect and non-differentiable sets in §5.3.1 and §5.3.2, and establish a connection between AD and Clarke subderivatives (a generalized notion of derivative) in §5.3.3.

We start by defining neural networks with bias parameters.

Definition 5.5. A pre-activation function $\tau_l : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}^{N_l}$ of a neural network *has bias parameters* if $W_l \geq N_l$ and there exist $f_1, \ldots, f_{N_l} : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l - N_l} \to \mathbb{R}$ such that

$$\tau_{l,i}(x, (u, v)) = f_i(x, u) + v_i$$

for all $i \in [N_l]$ and $(x, u, v) \in \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l - N_l} \times \mathbb{R}^{N_l}$. Here v_i is called the *bias parameter of* $\tau_{l,i}$. A neural network z_L *has bias parameters* if τ_l has bias parameters for all $l \in [L]$.

Many popular pre-activation functions are typically implemented with bias parameters. For example, fully-connected layers, attention layers (e.g., MultiheadAttention), and some normalization layers (e.g., LayerNorm) do so. Yet not all pre-activation functions have bias parameters in practice. For instance, convolutional layers and other normalization layers (e.g., BatchNorm) usually do not satisfy Definition 5.5: they do contain some bias terms, but each of these terms is used to compute multiple output values (instead of a single output value as in our definition).

5.3.1 Characterization of the Incorrect Set

We first show that the incorrect set of a neural network is always empty if the network has bias parameters, i.e., AD computes the standard derivative wherever the network is differentiable.

Theorem 5.6. *If a neural network z_L has bias parameters, then for all $w \in \mathbb{R}^W$ at which z_L is differentiable,*

$$D^{\mathsf{AD}} z_L(w) = D z_L(w). \tag{5.2}$$

This implies that $|\mathsf{inc}_\Omega(z_L)| = 0$.

It should be emphasized that Eq. (5.2) is not only for machine-representable parameters, but also for any real-valued parameters. Compared to existing results, this result is surprising. For instance, [18, 92] show that the incorrect set over $\mathbb{R}^n$ (not over $\mathbb{M}^n$) has Lebesgue measure zero for some classes of programs, but they do not give any results on whether the set can be empty. In contrast, Theorem 5.6 states that the incorrect set over $\mathbb{R}^n$ is empty for a smaller, yet still large class of programs, i.e., neural networks with bias parameters.

In Theorem 5.6, the condition that z_L has bias parameters plays a crucial role. Namely, Theorem 5.6 does not hold if this condition is dropped. For instance, consider a neural network $z_L : \mathbb{R} \to \mathbb{R}$ that is essentially the same as $f : \mathbb{R} \to \mathbb{R}$ with $f(w) = \mathrm{ReLU}(w) - \mathrm{ReLU}(-w)$ (which we discussed in §5.1). Then, z_L does not have bias parameters, and $\mathsf{inc}_\Omega(z_L)$ is non-empty if $D^{\mathsf{AD}}\mathrm{ReLU} = \mathbf{1}_{(0,\infty)}$ is used.

The proof of Theorem 5.6 consists of the following two arguments: for all $w \in \mathbb{R}^W$ with $D z_L(w) \neq \bot$,

(i) if $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$, then $\partial z_L / \partial z_{l,i} = \vec{0}$ at w, and

(ii) if (i) holds, then $D^{\mathsf{AD}} z_L(w) = D z_L(w)$.

That is, (i) if a pre-activation value $y_{l,i}$ touches a non-differentiable point of its activation function $\sigma_{l,i}$, then the derivative of z_L with respect to $z_{l,i}$ should always be zero; and (ii) Theorem 5.6 follows from (i). We point out that the proof of (i) relies heavily on the bias parameter condition. For more details, see Appendix B.3.

5.3.2 Characterization of the Non-Differentiable Set

We next show that if a neural network has bias parameters, then the density of the non-differentiable set in Ω is bounded by $n/|\mathbb{M}|$, where n is the total number of non-differentiable points in activation functions.

Theorem 5.7. *If a neural network z_L has bias parameters,*

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \leq \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in \mathsf{Idx}} |\mathsf{ndf}(\sigma_{l,i})|$$

where $\mathsf{ndf}(f)$ is the set of non-differentiable points of f.

In many practical settings, the bound in Theorem 5.7 is often small, especially under high-precision parameters. For example, $\mathbb{M}$ is frequently chosen as the set of 32-bit floating-point numbers so $|\mathbb{M}| \approx 2^{32}$, while $|\mathsf{Idx}|$ (the number of neurons) is often smaller than 2^{32} and $|\mathsf{ndf}(\sigma_{l,i})|$ is typically small (e.g., 0 for differentiable $\sigma_{l,i}$, 1 for ReLU, and 2 for HardSigmoid). This implies that in practice, the non-differentiable set often has a low density in Ω. We remark, however, that the bound in Theorem 5.7 can grow large in low-precision settings (e.g., when parameters are represented by $\leq$ 16-bit numbers).

Although the bound in Theorem 5.7 can be large in some cases (e.g., when $|\mathbb{M}|$ is small), we prove that the bound is in general tight up to a constant multiplicative factor.

Theorem 5.8. *For any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ with $1 \leq |\mathbb{M}| < \infty$, $n \geq 2$, and $\alpha \leq |\mathbb{M}|/(n-1)$, there is a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ with bias parameters that satisfies*

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{2} \cdot \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in \mathsf{Idx}} |\mathsf{ndf}(\sigma_{l,i})|$$

and the following: z_L has $n+1$ neurons and $|\mathsf{ndf}(\sigma_{1,i})| = \alpha$ for all $i \in [N_1]$.

In Theorem 5.8, the condition $\alpha \leq |\mathbb{M}|/(n-1)$ is for achieving the constant $1/2$ in the bound. A similar bound can be derived for a larger α (i.e., $\alpha > |\mathbb{M}|/(n-1)$) but with a constant smaller than $1/2$.

Theorems 5.7 and 5.8 describe how large the non-differentiable set $\mathsf{ndf}_\Omega(z_L)$ can be, but give no clue about exactly which parameters constitute this set. To better understand this, we present an easily verifiable necessary and sufficient condition for characterizing $\mathsf{ndf}_\Omega(z_L)$.

Theorem 5.9. *If a neural network z_L has bias parameters, then the following are equivalent for all $w \in \mathbb{R}^W$.*

- *z_L is non-differentiable at w.*
- *$y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ and $\partial^{\mathtt{AD}} z_L / \partial z_{l,i} \neq \vec{0}$ at w for some $(l,i) \in \mathsf{Idx}$.*

Here $\partial^{\mathtt{AD}} z_L / \partial z_{l,i}$ denotes the partial derivative of z_L with respect to $z_{l,i}$ that reverse-mode AD (e.g., backpropagation) computes as a byproduct of computing $D^{\mathtt{AD}} z_L$ (see Appendix B.5.2 for more details). Hence, Theorem 5.9 implies that we can efficiently[7] decide whether a neural network with

[7] in $\mathcal{O}(N_L T)$ time for a neural network $z_L : \mathbb{R}^W \to \mathbb{R}^{N_L}$ where T is the time to evaluate $z_L(w)$, because reverse-mode AD takes $\mathcal{O}(N_L T)$ time to compute $D^{\mathtt{AD}} z_L(w)$.

bias parameters is non-differentiable at a (real-valued) parameter or not. This result is surprising given a recent, relevant result that deciding such non-differentiability is NP-hard in general [19].

We now sketch the proof of Theorem 5.7, to explain how we obtain the bound in the theorem and where we use the bias parameter condition. First, we prove that if $y_{l,i}(w)$ does not touch any non-differentiable point of $\sigma_{l,i}$ for all $(l,i) \in \mathsf{Idx}$, then z_L is differentiable at w. In other words,

$$\mathsf{ndf}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in\mathsf{Idx}} \bigcup_{c\in\mathsf{ndf}(\sigma_{l,i})} \{w \in \Omega \mid y_{l,i}(w) = c\}. \tag{5.3}$$

Second, we prove that for all $(l,i) \in \mathsf{Idx}$ and $c \in \mathbb{R}$,

$$\left|\{w \in \Omega \mid y_{l,i}(w) = c\}\right| \leq |\mathbb{M}|^{W-1}. \tag{5.4}$$

This inequality is invalid in general, but is valid when τ_l has bias parameters. If the parameter w has a value $v = (v_1, \ldots, v_W)$ and its j-th entry v_j corresponds to the bias parameter of $\tau_{l,i}$, then $y_{l,i}(v) = f(v_{-j}) + v_j$ for some function f. Hence, for any $v_{-j} \in \mathbb{M}^{W-1}$, there is at most one $v_j \in \mathbb{M}$ achieving $y_{l,i}(v) = c$, and this implies the above inequality. Finally, we prove that Theorem 5.7 follows from the above two results. The full proofs of Theorems 5.7–5.9 are presented in Appendices B.2, B.4, and B.5, respectively.

5.3.3 Connection to Clarke Subderivatives

We have so far observed that with bias parameters, the incorrect set is always empty but the non-differentiable set may not be. A natural question is then: what does AD compute on the non-differentiable set? We answer this question by showing that AD computes a Clarke subderivative[8] everywhere (including on the non-differentiable set), if it uses *consistent* extended derivatives for activation functions.

Theorem 5.10. *If a neural network z_L has bias parameters and $D^{\mathsf{AD}}\sigma_{l,i}$ is consistent for all $(l,i) \in \mathsf{Idx}$, then for all $w \in \mathbb{R}^W$,*

$$D^{\mathsf{AD}} z_L(w) = \begin{cases} Dz_L(w) & \text{if } Dz_L(w) \neq \bot \\ \lim_{n\to\infty} Dz_L(w'_n) \text{ for some } w'_n \to w & \text{if } Dz_L(w) = \bot. \end{cases}$$

This implies that $D^{\mathsf{AD}} z_L$ is a Clarke subderivative of z_L.

Theorem 5.10 is not only a new result about AD, but also gives a positive answer to a long-standing open question about Clarke subgradients [19, 28, 77]: are there a sufficiently large class $\mathcal{F}$ of *scalar* functions and a deterministic algorithm $\mathcal{A}$ that computes a *Clarke subgradient* (i.e., subderivative)

[8] The *Clarke subdifferential* of $f : \mathbb{R}^n \to \mathbb{R}^m$ at $x \in \mathbb{R}^n$ refers to the convex hull of $\{\lim_{n\to\infty} Df(x_n) \mid x_n \to x, Df(x_n) \neq \bot\} \subseteq \mathbb{R}^{m\times n}$, and an element of the Clarke subdifferential is called a *Clarke subderivative* [29, 77].

of $f \in \mathcal{F}$ at $x \in \mathbb{R}^n$ efficiently (i.e., in time $\mathcal{O}(T)$ that is independent of n, where T is time to evaluate $f(x)$)? In other words, is there a so-called "Cheap Subgradient Principle"? For instance, [77] propose an efficient algorithm $\mathcal{A}'$ (for some $\mathcal{F}'$) but $\mathcal{A}'$ is not deterministic, whereas [8, 80] propose deterministic algorithms $\mathcal{A}''$ (for some $\mathcal{F}''$) but $\mathcal{A}''$ are not efficient. In contrast, Theorem 5.10 implies that for neural networks with bias parameters, a Clarke subgradient at any (real-valued) parameter can be computed deterministically and efficiently, even by the vanilla reverse-mode AD. In this sense, we provide a new understanding on the computational aspects of Clarke subgradients.

We note that Theorem 5.10 no longer holds without any of its conditions: having bias parameters and using consistent extended derivatives. One can confirm this using the following examples: $z_L(w) = \mathrm{ReLU}(w) - \mathrm{ReLU}(-w)$ with $D^{\mathtt{AD}}\mathrm{ReLU} = \mathbf{1}_{(0,\infty)}$ (in which z_L does not have bias parameters as observed in §5.3.1), and $\widehat{z}_L(w) = \mathrm{ReLU}(w)$ with $D^{\mathtt{AD}}\mathrm{ReLU} = \mathbf{1}_{(0,\infty)} + c \cdot \mathbf{1}_{\{0\}}$ for any $c \in \mathbb{R} \setminus [0,1]$ (in which $D^{\mathtt{AD}}\mathrm{ReLU}$ is not consistent). For the proof of Theorem 5.10, see Appendix B.6.

5.4 Neural Networks without Bias Parameters

In this section, we investigate the correctness of AD for neural networks that may or may not have bias parameters. For such general networks, however, considering only the properties of activation functions such as $\mathsf{ndf}(\sigma_{l,i})$ (as we did in §5.3) is insufficient to derive non-trivial bounds on the size of the incorrect and non-differentiable sets, as long as general pre-activation functions are used.

To illustrate this, consider neural networks $z_L, \widehat{z}_L : \mathbb{R} \to \mathbb{R}$ that are essentially the same as $f, \widehat{f} : \mathbb{R} \to \mathbb{R}$ with $f(w) = \mathrm{ReLU}(h(w)) - \mathrm{ReLU}(-h(w))$ and $\widehat{f}(w) = \mathrm{ReLU}(h(w))$, where $h : \mathbb{R} \to \mathbb{R}$ is some analytic pre-activation function satisfying $h(x) = 0$ and $Dh(x) = 1$ for all $x \in \mathbb{M}$. Suppose that $D^{\mathtt{AD}}\mathrm{ReLU} = \mathbf{1}_{(0,\infty)}$. Then, we have $\mathsf{inc}_\Omega(z_L) = \mathsf{ndf}_\Omega(\widehat{z}_L) = \Omega$ even though z_L and $\widehat{z}_L$ have only ≤ 2 non-differentiable points in their activation functions. The main culprit of having such large $\mathsf{inc}_\Omega(z_L)$ and $\mathsf{ndf}_\Omega(\widehat{z}_L)$, even with a tiny number of non-differentiable points in activation functions, is that z_L and $\widehat{z}_L$ use the unrealistic pre-activation function h which does not have bias parameters.

To exclude such extreme cases and focus on realistic neural networks, we will often consider *well-structured biaffine* pre-activation functions when they do not have bias parameters.

Definition 5.11. A pre-activation function $\tau_l : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}^{N_l}$ is *well-structured biaffine* if there are $M_i \in \mathbb{R}^{N_{l-1} \times W_l}$ and $c_i \in \mathbb{R}$ for all $i \in [N_l]$ such that

$$\tau_{l,i}(x, u) = x^\mathsf{T} M_i u + c_i$$

and each column of M_i has at most one non-zero entry.

Any fully-connected or convolution layers are well-structured biaffine when they do not have bias parameters. Thus, a large class of neural networks is still under our consideration even after we

impose the above restriction. Yet some pre-activation functions (e.g., normalization and attention layers) are not well-structured biaffine whether or not they have bias parameters.

We now present our results for neural networks possibly without bias parameters, extending Theorems 5.6–5.10.

5.4.1 Bounds for Non-Differentiable and Incorrect Sets

We first bound the density of the non-differentiable and incorrect sets in Ω, extending Theorem 5.7.

Theorem 5.12. *If a pre-activation function τ_l has bias parameters or is well-structured biaffine for all $l \in [L]$, then*

$$\frac{|\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L)|}{|\Omega|} \leq \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in \mathsf{Idx}} \Big|\mathsf{ndf}(\sigma_{l,i}) \cup \big(\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1}\big)\Big|,$$

where $\mathsf{bdz}(f)$ is the boundary of f's zero set (see §5.2.1), and

$$S_l \triangleq \begin{cases} \emptyset & \text{if } l > L \text{ or } \tau_l \text{ has bias parameters} \\ \mathbb{R} & \text{otherwise.} \end{cases}$$

We note that if z_L has bias parameters, Theorem 5.12 reduces to Theorem 5.7 since $\mathsf{inc}_\Omega(z_L) = \emptyset$ (by Theorem 5.6) and $S_l = \emptyset$ for all l (by its definition) in such a case.

As in Theorem 5.7, the bound in Theorem 5.12 is often small for neural networks that use practical activation functions, since $|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})|$ is typically small for those activation functions (e.g., 1 for ReLU and 2 for HardSigmoid).

We now show that the additional term $\mathsf{bdz}(\sigma_{l,i})$ in Theorem 5.12 is indeed necessary by providing a matching lower bound up to a constant factor.

Theorem 5.13. *For any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ with $1 \leq |\mathbb{M}| < \infty$, $n \geq 4$, and $\alpha \leq |\mathbb{M}|/(n-1)$, there is a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that satisfies*

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{9} \cdot \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in \mathsf{Idx}} \Big|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\Big|$$

and the following: (i) τ_l is well-structured biaffine without bias parameters for all $l < L$, and has bias parameters for $l = L$; (ii) z_L has $n+1$ neurons; and (iii) $|\mathsf{ndf}(\sigma_{1,i})| = \alpha$ and $|\mathsf{bdz}(\sigma_{1,i})| = 0$ for all i. We obtain the same result for (i), (ii'), and (iii'): (ii') z_L has $2n+1$ neurons; and (iii') $|\mathsf{ndf}(\sigma_{1,i})| = 0$ and $|\mathsf{bdz}(\sigma_{1,i})| = \alpha$ for all i.

We next give an intuition for why the zero set of $\sigma_{l,i}$ (from which the additional term $\mathsf{bdz}(\sigma_{l,i})$ is defined) appears in Theorem 5.12, by examining its proof. The proof consists of two main parts

that extend Eqs. (5.3) and (5.4) from the proof sketch of Theorem 5.7: we first show

$$\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in\mathsf{Idx},\, c\in\mathsf{ndf}(\sigma_{l,i})} \{w \in \Omega \mid y_{l,i}(w) = c\}$$

and then find a reasonable bound on $|\Lambda_{l,i,c}|$ for $\Lambda_{l,i,c} \triangleq \{w \in \Omega \mid y_{l,i}(w) = c\}$, the set of parameters on which the pre-activation value $y_{l,i}$ touches the non-differentiable point c of $\sigma_{l,i}$. Among the two parts, the zero set of $\sigma_{l,i}$ arises from the second part (i.e., bounding $|\Lambda_{l,i,c}|$), especially when τ_l does not have bias parameters and is well-structured biaffine. For simplicity, assume that τ_l is a fully-connected layer with constant biases, i.e., $y_{l,i}(w) = \sum_{j\in[N_{l-1}]} z_{l-1,j}(w) \cdot w_{j+a} + b$ for some constants a, b. Based on this, we decompose $\Lambda_{l,i,c}$ into $\Lambda' \cup \Lambda''$:

$$\Lambda' \triangleq \{w \in \Omega \mid y_{l,i}(w) = c,\ z_{l-1,j}(w) \neq 0 \text{ for some } j\},$$
$$\Lambda'' \triangleq \{w \in \Omega \mid y_{l,i}(w) = c,\ z_{l-1,j}(w) = 0 \text{ for all } j\}.$$

Then, we can show $|\Lambda'| \leq |\mathbb{M}|^{W-1}$ as in Eq. (5.4), since w_{j+a} acts like a bias parameter of $y_{l,i}$ for any j with $z_{l-1,j}(w) \neq 0$. To bound $|\Lambda''|$, however, we cannot apply a similar approach due to the lack of j with $z_{l-1,j}(w) \neq 0$. Instead, we directly count the number of parameters $w \in \Omega$ achieving $z_{l-1,j}(w) = 0$ for all j (i.e., $\sigma_{l-1,j}(y_{l-1,j}(w)) = 0$ for all j), and this requires the zero set of $\sigma_{l-1,j}$. For the full proofs of Theorems 5.12 and 5.13, see Appendices B.2 and B.4.

5.4.2 Bounds for the Incorrect Set

For the non-differentiable set, Theorems 5.12 and 5.13 provide tight bounds on its size. For the incorrect set, it turns out that we can further improve the upper bound in Theorem 5.12 and get a similar lower bound to Theorem 5.13.

Theorem 5.14. *If a pre-activation function τ_l has bias parameters or is well-structured biaffine for all $l \in [L]$, then*

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \leq \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in\mathsf{Idx}} \Big|\big(\mathsf{ndf}(\sigma_{l,i}) \cap S_l\big) \cup \big(\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1}\big)\Big|,$$

where S_l is defined as in Theorem 5.12.

Theorem 5.15. *For any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ with $1 \leq |\mathbb{M}| < \infty$, $n \geq 4$, and $\alpha \leq |\mathbb{M}|/(n-1)$, there is a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that satisfies*

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{13} \cdot \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in\mathsf{Idx}} \Big|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\Big|$$

and the following: (i) τ_l is well-structured biaffine without bias parameters for all $l < L$, and has

bias parameters for $l = L$; (ii) z_L has $2n + 1$ neurons; and (iii) $|\mathsf{ndf}(\sigma_{1,i})| = \alpha$ and $|\mathsf{bdz}(\sigma_{1,i})| = 0$ for all i. We obtain the same result for (i), (ii'), and (iii'): (ii') z_L has $3n + 1$ neurons; and (iii') $|\mathsf{ndf}(\sigma_{1,i})| = 0$ and $|\mathsf{bdz}(\sigma_{1,i})| = \alpha$ for all i.

We note that if z_L has bias parameters, Theorem 5.14 reduces to $|\mathsf{inc}_\Omega(z_L)| = 0$ as in Theorem 5.6 since $S_l = \emptyset$ for all l in the case. On the other hand, if z_L does not have bias parameters, then the incorrect set can be non-empty as discussed in §5.3.1, and more importantly, its size can be bounded by Theorem 5.14. To see why the bounds on $|\mathsf{inc}_\Omega(z_L)|$ depend on both $\mathsf{ndf}(\sigma_{l,i})$ and $\mathsf{bdz}(\sigma_{l,i})$, refer to the proofs of Theorems 5.14 and 5.15 in Appendices B.3 and B.4.

5.4.3 Conditions for Computing Standard Derivatives and Clarke Subderivatives

We extend Theorems 5.9 and 5.10 to general neural networks without the well-structured biaffinity restriction, by characterizing two sufficient conditions on parameters under which AD computes the standard derivative or a Clarke subderivative.

Theorem 5.16. *Let $w \in \mathbb{R}^W$. If $y_{l,i}(w) \notin \mathsf{ndf}(\sigma_{l,i})$ for all $(l,i) \in \mathsf{Idx}$ such that τ_l does not have bias parameters or $\partial^{\mathsf{AD}} z_L / \partial z_{l,i} \neq \vec{0}$ at w, then*

$$D^{\mathsf{AD}} z_L(w) = Dz_L(w) \neq \bot.$$

Theorem 5.17. *Let $w \in \mathbb{R}^W$. Assume that $D^{\mathsf{AD}}\sigma_{l,i}$ is consistent for all $(l,i) \in \mathsf{Idx}$. If $y_{l,i}(w) \notin \mathsf{ncdf}(\sigma_{l,i})$ for all $(l,i) \in \mathsf{Idx}$ such that τ_l does not have bias parameters, then*

$$D^{\mathsf{AD}} z_L(w) = \begin{cases} Dz_L(w) & \text{if } Dz_L(w) \neq \bot \\ \lim_{n\to\infty} Dz_L(w'_n) \text{ for some } w'_n \to w & \text{if } Dz_L(w) = \bot \end{cases}$$

and so $D^{\mathsf{AD}} z_L(w)$ is a Clarke subderivative of z_L at w. Here $\mathsf{ncdf}(f)$ denotes the set of real numbers at which $f : \mathbb{R} \to \mathbb{R}$ is not continuously differentiable.

The two sufficient conditions on w given in Theorems 5.16 and 5.17 are simple enough to be checked efficiently in practice; thus, we can use them to validate whether the output of AD is the standard derivative or a Clarke subderivative. If w does not satisfy either of the sufficient conditions, AD may not compute the standard derivative or a Clarke subderivative; the first example discussed in §5.3.3 illustrates both cases. We remark that the sufficient condition in Theorem 5.17 involves $\mathsf{ncdf}(\sigma_{l,i})$ (not $\mathsf{ndf}(\sigma_{l,i})$), since we use continuous differentiability (not differentiability) in the proof to properly handle the limit of derivatives $Dz_L(w'_n)$. For the proofs of Theorems 5.16 and 5.17, see Appendices B.5 and B.6.

5.5 Related Work

The correctness of AD has been extensively studied, especially in the past few years. When a program uses only differentiable functions, AD is shown to compute its standard derivative at all real-valued inputs [1, 7, 20, 47, 70, 83, 126, 142, 151]. In contrast, when a program uses non-differentiable functions, the program itself can be non-differentiable, and AD can return a value different from its standard derivative, at some real-valued inputs. Nevertheless, for a large class of programs, such inputs are shown to be in a Lebesgue measure-zero subset of the real-valued input domain [17, 18, 71, 92, 101]. All these works consider the case when inputs to AD are real-valued, while this chapter focuses on the case when the inputs are machine-representable.

The Clarke subdifferential and its connection to AD have been studied for decades. Some classes of functions (e.g., subdifferentially regular or strictly differentiable) are shown to admit exact chain rules for the Clarke subdifferential (e.g., Theorems 2.3.9, 2.3.10, and 2.6.6 of [29] and Theorem 10.6 of [130]), and this implies that AD always computes a Clarke subderivative for a certain class of programs. However, this class of programs is restrictive, excluding even simple neural networks (e.g., $(1-\mathrm{ReLU}(x))^2$) [40]. In contrast, our Theorem 5.10 shows that AD always computes a Clarke subderivative of neural networks with bias parameters. For piecewise differentiable functions, the Clarke subdifferential can be expressed in terms of the standard derivatives of underlying differentiable functions (e.g., Proposition 4.3.1 of [139]), but this result is not directly related to AD.

A variety of algorithms (other than AD) have been proposed to compute a Clarke subgradient of a scalar program, correctly and efficiently. For a large class of programs $f : \mathbb{R}^n \to \mathbb{R}$ and an input $x \in \mathbb{R}^n$, the algorithm by [77] computes a Clarke subgradient of f at x in time $\mathcal{O}(T)$ almost surely, while the algorithms by [8, 80] compute the quantity in time $\mathcal{O}(nT)$ deterministically, where T denotes time to evaluate $f(x)$. Our Theorem 5.10 provides a relevant result as described above, but we point out that this chapter is about analyzing the correctness of vanilla (forward/reverse-mode) AD, not about proposing a new algorithm.

Recently, [13] empirically studied how the choice of $D^{\mathtt{AD}}\mathrm{ReLU}(0)$ changes the output of AD and the training of neural networks. In contrast, this chapter theoretically studies the correctness of AD. Further connections between [13] and this chapter are discussed in §5.6.

5.6 Discussion

Connections to [13]. The paper [13] empirically studied the *bifurcation zone* of a neural network with ReLU, given an input dataset: the set of the network parameters on which the output of AD using $D^{\mathtt{AD}}\mathrm{ReLU}(0) = 0$ is different from that using $D^{\mathtt{AD}}\mathrm{ReLU}(0) = 1$ for some input data. The bifurcation zone is closely related to the non-differentiable and incorrect sets as follows: the bifurcation zone (over machine-representable parameters) is always a subset of the union of the non-differentiable set and two incorrect sets (one for $D^{\mathtt{AD}}\mathrm{ReLU}(0) = 0$ and the other for $D^{\mathtt{AD}}\mathrm{ReLU}(0) = 1$) over all

input data in the given dataset.

For various neural networks (MLP, VGG, ResNet) and datasets (MNIST, CIFAR10, SVHN, ImageNet), the paper [13] estimated the density of the bifurcation zone over 32-bit floating-point parameters (i.e., the number of 32-bit parameters in the bifurcation zone over the total number of 32-bit parameters) using Monte Carlo sampling. The paper reported two results among many others: when AD uses 64-bit precision in its computation, the estimated density is exactly 0 in all cases they considered; and when AD uses 32- or 16-bit precision, the estimated density is often large and even goes up to 1. The first result is consistent with our results: if we use 32-bit parameters, the non-differentiable and incorrect sets would often have small densities in practice. Meanwhile, the second result does not contradict our results, since our results assume that AD computes its output without any rounding errors. Given these observations, it would be an interesting direction to rigorously study the correctness of AD under floating-point operations.

Extensions. As mentioned in §5.2.2, all our theorems except for those on lower bounds (i.e., Theorems 5.8, 5.13 and 5.15) continue to hold even if we replace z_L in their conclusions by $\ell \circ z_L$ for any analytic $\ell : \mathbb{R}^{N_L} \to \mathbb{R}^m$. Among them, Theorems 5.7 and 5.12 are easily extended to a more general case with multiple input data: they remain valid even if we replace z_L in their conclusions by $\ell(z_L(\cdot; x_1), \dots, z_L(\cdot; x_k))$ for any $x_1, \dots, x_k \in \mathbb{R}^{N_0}$ and analytic $\ell : \mathbb{R}^{N_L} \to \mathbb{R}^m$, where we need to multiply k to the upper bounds in the theorems. The remaining theorems (i.e., Theorems 5.6, 5.9, 5.10, 5.14, 5.16 and 5.17), on the other hand, are not easily extended to the case with multiple input data, at least based on our current proofs. Studying such extensions could be another interesting future direction.

Limitations. Our results have some limitations. For example, all of our results are for a class of neural networks consisting of alternating analytic pre-activation functions and pointwise continuous activation functions. Hence, if a network contains non-pointwise activation functions (e.g., MaxPool) or a residual connection bypassing a non-analytic activation function (e.g., ReLU), then our results may not be directly applicable. Our results for general neural networks (e.g., Theorems 5.12 and 5.14) additionally assume pre-activation functions to have bias parameters or to be well-structured biaffine, which does not allow, e.g., BatchNorm layers and attention layers without bias parameters. Nevertheless, we believe that our results still cover a large class of neural networks, especially compared to prior works studying theoretical aspects of neural networks [75, 81, 88, 96, 116]. We believe that extending the work in this chapter to more general neural networks is an interesting direction for future work.

5.7 Conclusion

In this chapter, we theoretically study for the first time the correctness of AD for neural networks with machine-representable parameters. In particular, we provide various theoretical results on the

incorrect and non-differentiable sets of a neural network, as well as closely related questions such as when AD is correct and what it computes. Our results have two major practical implications: AD is correct at most machine-representable parameters when applied to neural networks, and it is correct more often if more layers of the network have bias parameters. Furthermore, our theoretical analyses suggest new applications of AD for identifying differentiability and computing Clarke subderivatives, not only for machine-representable parameters but also for any real-valued ones.

Chapter 6

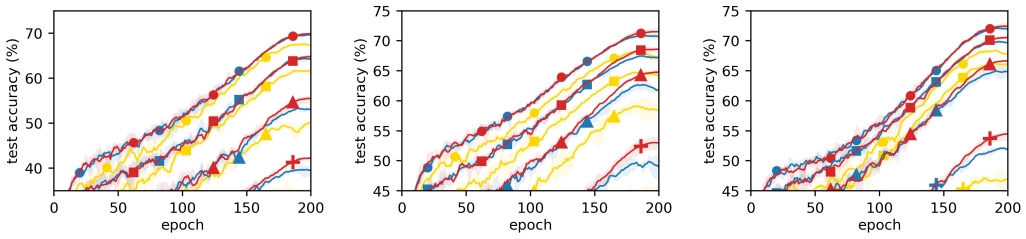
Acceleration of Deep Neural Network Training

6.1 Introduction

In deep neural network training, floating-point formats are usually used to represent tensors and it is worthwhile to use the smallest bitwidth format that gives acceptable results. For example, it is common to replace tensors using 32-bit floats with tensors that use 16-bit floats [78, 104]. The benefits are easy to understand: computations using lower-precision floats not only use less memory but are also faster (due to improved vector parallelism, locality, and reduced data movement). The downside is that there is generally some loss of training accuracy, and in the worst case training may not even converge.

For such *low-precision floating-point training*, the most common approaches use two floating-point formats—one for lower-precision floats (e.g., 8-bit floats) and the other for higher-precision floats (e.g., 16-bit floats)—and assign one of the two formats to each tensor (including weights, activations, and their gradients). The precision assignments studied in previous work fall into one of two assignment schemes (which both have several variants): the *uniform* assignment uses low precision for almost all tensors (often excepting those in the first and/or last few layers) [104], while the *operator-based* assignment limits low precision to the input tensors of certain operators (e.g., convolutions) [147]. Prior work has shown that both precision assignment schemes (with well-chosen low-bitwidth floating-point formats) can match the accuracy of 32-bit-float training [22, 26, 44, 48, 78, 104, 147, 155].

There is an important limitation in all prior approaches to low-precision floating-point training: they use very few precision assignments (most often just one) for a given set of models, but there are some other models and inputs where the chosen precision assignment (i) results in noticeably worse accuracy than 32-bit-float training, (ii) causes training to even diverge, or (iii) admits a more efficient assignment that achieves similar training accuracy (see Figures 6.1, 6.3, and 6.4).



(a) SqueezeNet (b) ShuffleNet-v2 (c) MobileNet-v2

Figure 6.1: Training trajectory of various models on CIFAR-100. Colors denote precision assignments: all-32-bit π_{fp32} (red), uniform π_{unif} (yellow), and operator-based π_{op} (blue) (see §6.2.1); the latter two use the 8-bit (and 16-bit) floats in [147] as low (and high) precision numbers. Markers denote the "width multiplier" of a model, which controls the capacity of the model (see §6.4.3): 1.0 (●), 0.5 (■), 0.25 (▲), and 0.1 (✚). Some lines of π_{unif} are missing as they converge to small values or diverge. Observe that neither π_{unif} nor π_{op} works best for all models: in some models, π_{op} has a similar accuracy to π_{fp32}; but in other (and all) models, the accuracy drop of π_{op} (and π_{unif}) from π_{fp32} are noticeably large (i.e., $>1\%$).

In this chapter, we present a new, automated method for choosing precision assignments that removes the limitations described above. To do so, we formally introduce the *memory-accuracy tradeoff problem* (§6.2.2): given a dataset, a model, and two floating-point precision levels (i.e., bitwidths; high and low), find a *mixed precision assignment* (a mapping from all tensors arising in training to high/low precision) for the model that maximizes training accuracy subject to a given upper bound on the *model aggregate* (i.e., the total number of bits of all tensors appearing in training). The model aggregate is a proxy for the memory and time required for training, as it is roughly proportional to memory footprint and also well-correlated with training time (since training is often dominated by data movement) [104].

We prove that the memory-accuracy tradeoff problem is theoretically difficult (namely NP-hard) partly due to the exponential number of possible mixed precision assignments (which we often refer to simply as precision assignments for brevity) (§6.2.3). The large number of possible assignments makes the problem difficult in practice as well: there is no known analytical method for predicting the training accuracy of a given precision assignment, and for any practical model there are far too many precision assignments to simply test them all.

We propose a simple (heuristic) approach to the tradeoff problem that prioritizes tensors for low-precision formats based on the tensor's size (with an additional step described below) (§6.3.1). More specifically, our algorithm takes as input a single parameter giving a desired upper bound on the model aggregate. Starting with the largest tensor in the model, tensors are assigned low precision in size order (from largest to smallest) until the model aggregate falls below the given upper bound; all remaining tensors are assigned high precision. Our main result is that this method discovers mixed precision assignments that use less memory while achieving higher training accuracy than previous approaches. While we cannot show that our method finds Pareto-optimal memory-accuracy tradeoffs, we do show that our results are closer to Pareto-optimal than prior methods.

Some precision assignments initially generated by our algorithm cause training to diverge due to an excessive number of overflows. To address this issue, we propose an overflow handling technique that promotes tensors causing too many overflows from low precision to high precision during training (§6.3.2). In our experiments, these promotions consume only a small amount of additional memory ($< 3\%$ of the maximum model aggregate) and prevent training from diverging. The overflow handling technique is not specific to our algorithm and can be applied to other precision assignment methods as well.

We evaluate a PyTorch implementation of our method on standard image classification tasks by training four convolutional networks (and their variants) on CIFAR-10, CIFAR-100, and ImageNet (§6.4). We first demonstrate that the precision assignments computed by our method alleviate the limitations of existing methods: they indeed explore the tradeoff between memory and accuracy and exhibit a better tradeoff than the uniform and operator-based assignments. We then show the two main components of our method (i.e., precision demotion of larger tensors and precision promotion of overflowing tensors) are both important to produce competitive precision assignments. We also provide some guidance on how users may apply our method to navigate the memory-accuracy tradeoff.

To summarize, this chapter makes four main contributions:

- We formally introduce the memory-accuracy tradeoff problem to explore better mixed precision assignments for low-precision floating-point training and prove the NP-hardness of the problem.
- We present a novel precision assignment technique, as a heuristic solution to the tradeoff problem, that proposes assignments based on a single parameter denoting a desired upper bound on the model aggregate.
- We present a novel technique that handles an excessive number of overflows arising in training while using a small amount of additional memory. The technique is applicable to any (not just our) precision assignments.
- We demonstrate that the mixed precision assignments found by our method do explore the tradeoff between memory and training accuracy, and outperform existing precision assignment methods.

We remark that this chapter focuses on low-precision *floating-point* training, not *fixed-point* training (which uses fixed-point formats), since we want to target upcoming (or very recent) hardware with native support for low-precision floats (e.g., 8-bit floats) and their operations (e.g., [4]). Also, this chapter focuses on low-precision *training* (which trains a model from scratch), not *inference* (which assumes a pre-trained model). More discussion is in §6.5. We further remark that in our experiments we simulate low-precision formats (e.g., 8-bit floats) with 32-bit floats as in prior works, since a hardware and software ecosystem that natively supports these formats does not yet exist. Similarly, we do not include other models (e.g., language models) in the experiments, since no current

software stacks support per-tensor precision assignments for certain operators that those models use. More details are in §6.4.1 and §6.4.2.

For image classification tasks and convolutional networks, our precision assignment method typically provides $> 2\times$ memory reduction over the operator-based assignment while maintaining similar training accuracy and gives further reductions by trading off accuracy. Our method also provides similar memory reduction to the uniform assignment, while avoiding the divergence of training often caused by a uniform assignment.

This chapter is organized as follows. We first define the memory-accuracy tradeoff problem and study its hardness (§6.2). We then describe our algorithm for the problem (§6.3) and our evaluation (§6.4). After discussing related work (§6.5), we conclude this chapter (§6.6).

6.2 Problem

In this section, we first provide background on low-precision floating-point training (§6.2.1), based on which the memory-accuracy tradeoff problem is introduced (§6.2.2). We then prove the NP-hardness of the problem (§6.2.3). Our approach in §6.2–6.3 is more formal than most related works for two reasons: (i) we show the problem is NP-hard, which has not been considered in prior work; and (ii) to clearly describe the precision assignments to be considered.

6.2.1 Low-Precision Floating-Point Training

Let $\mathbb{T}$ be the set of real-valued tensors and let $[n]$ denote the set $\{1, \ldots, n\}$. For a supervised learning task, we usually consider a *model network* $\mathcal{M} = (f_1, \ldots, f_n)$ parameterized by $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n) \in \mathbb{T}^n$, and a *loss network* $\mathcal{L} = (f_{n+1}, \ldots, f_m)$, where $f_i : \mathbb{T}^2 \to \mathbb{T}$ is a primitive operator on tensors (e.g., convolution, batch normalization, maxpool, and softmax). Given an input-output pair $(x, y) \in \mathbb{T}^2$, the model $\mathcal{M}$ computes a predicted output y' of x by iteratively applying $f_i(\cdot, \theta_i)$ to x ($i \in [n]$), and $\mathcal{L}$ computes a loss from y' by iteratively applying $f_{i'}(\cdot, y)$ to y' ($i' \in [m] \setminus [n]$). A standard way to train $\mathcal{M}$ is to minimize the loss value using the gradient descent algorithm: iteratively update $\boldsymbol{\theta}$ by following the gradient of the loss with respect to $\boldsymbol{\theta}$.

Floating-point training. In practice, we perform a gradient computation usually with tensors represented in floating-point formats. Let $\pi : \mathsf{TS} \to \mathsf{FP}$ be a *precision assignment* giving the floating-point format of each tensor, where $\mathsf{TS} \triangleq \{v_i, dv_i, \theta_j, d\theta_j \mid i \in [m+1], j \in [n]\}$ is the set of tensors arising in a gradient computation (explained below), and $\mathsf{FP} \triangleq \{\mathsf{fp}(e, m, b) \mid e, m \in \mathbb{N}, b \in \mathbb{Z}\}$ is the set of floating-point formats. Here $\mathsf{fp}(e, m, b)$ denotes a floating-point format that consists of a 1-bit sign, an e-bit exponent, and an m-bit mantissa, and has an (additional) exponent bias of $b \in \mathbb{Z}$. A common choice of π is $\pi_{\mathrm{fp32}}(t) \triangleq \mathsf{fp32}$ for all $t \in \mathsf{TS}$, where $\mathsf{fp32} \triangleq \mathsf{fp}(8, 23, 0)$ is the standard 32-bit floating-point format.

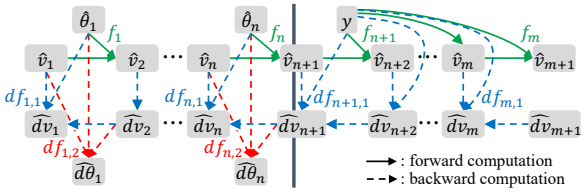


Figure 6.2: A diagram showing the tensors and operators used in a gradient computation; see Eq. (6.1) for details. For brevity, rounding functions $\mathrm{rnd}_{\pi(\cdot)}$ are omitted.

Given a precision assignment π, a gradient computation is typically performed by the backpropagation algorithm: with $\hat{v}_1 = \mathrm{rnd}_{\pi(v_1)}(x)$ and $\hat{dv}_{m+1} = \mathrm{rnd}_{\pi(dv_{m+1})}(1)$, compute

$$\begin{aligned} \hat{v}_{i+1} &= \mathrm{rnd}_{\pi(v_{i+1})}(f_i(\hat{v}_i, \hat{u}_i)), & \hat{\theta}_j &= \mathrm{rnd}_{\pi(\theta_j)}(\theta_j), \\ \hat{dv}_i &= \mathrm{rnd}_{\pi(dv_i)}(df_{i,1}(\hat{dv}_{i+1}, \hat{v}_i, \hat{u}_i)), & \hat{d\theta}_j &= \mathrm{rnd}_{\pi(d\theta_j)}(df_{j,2}(\hat{dv}_{j+1}, \hat{v}_j, \hat{\theta}_j)), \end{aligned} \tag{6.1}$$

for $i \in [m]$ and $j \in [n]$; see Figure 6.2 for a diagram. Here $\mathrm{rnd} : \mathsf{FP} \times \mathbb{T} \to \mathbb{T}$ is a function rounding a given input to a given floating-point format, $df_{i,1}, df_{i,2} : \mathbb{T}^3 \to \mathbb{T}$ are the backward operators of f_i with respect to its first and second arguments, respectively, and $\hat{u}_i = \hat{\theta}_i$ if $i \in [n]$ and y otherwise. We call v_i and θ_j the *forward* tensors, and dv_i and $d\theta_j$ the *backward* tensors. We put a hat over each tensor to emphasize that its value is the output of a rounding function to a possibly low-precision format; remark that such a rounding function is not used within f_i, $df_{i,1}$, and $df_{i,2}$, since they typically use large bitwidth floats (e.g., $\mathsf{fp32}$) and no low-precision floats internally [22, 78]. After the computation, $\hat{d\theta}_j$ stores the gradient of the loss value with respect to θ_j.

The overall picture of floating-point training is now described as follows. In each iteration of the gradient descent algorithm, we compute $\hat{d\theta}_j$ via Eq. (6.1) using a given precision assignment π, training data (x, y), and current weights $\boldsymbol{\theta}$. We then update each θ_j by $\theta_j \leftarrow \mathrm{rnd}_{\mathsf{fp32}}(\theta_j - \eta \cdot \hat{d\theta}_j)$ given a learning rate $\eta > 0$, and proceed to the next iteration until the training ends. Here we use $\mathsf{fp32}$ to represent θ_j by following the convention in low-precision floating-point training: a "master copy" of weights (i.e., θ_j) is stored separately from the weight values (i.e., $\hat{\theta}_j$) used in a gradient computation, and is usually represented by $\mathsf{fp32}$ [22, 78, 104]. The memory overhead of this master copy is very small compared to the memory required to store other tensors (e.g., activation tensors v_i) [104].

Low-precision floating-point training. In low-precision training, we use a precision assignment π where some tensors have a smaller bitwidth than $\mathsf{fp32}$. Particularly well-studied are π that use two predetermined floating-point bitwidths (which are different) and optionally vary the rest of the format from tensor to tensor. We call $\mathcal{C} : \mathsf{TS} \times \{\mathsf{lo}, \mathsf{hi}\} \to \mathsf{FP}$ a *precision-candidate assignment* if $\mathcal{C}(t, \mathsf{lo})$ has the same bitwidth for all $t \in \mathsf{TS}$, the same holds for hi, and the bitwidth for lo is smaller than that for hi. We define $\Pi(\mathcal{C}) \triangleq \{\pi : \mathsf{TS} \to \mathsf{FP} \mid \forall t \in \mathsf{TS}.\, \pi(t) \in \{\mathcal{C}(t, \mathsf{lo}), \mathcal{C}(t, \mathsf{hi})\}\}$ as the set of precision assignments that conform to $\mathcal{C}$.

Among various precision assignments in $\Pi(\mathcal{C})$, two have received the most attention: the uniform

assignment $\pi_{\mathrm{unif},\mathcal{C}}$ [104] and the operator-based assignment $\pi_{\mathrm{op},\mathcal{C}}$ [147]. The former assigns low-precision formats to all tensors uniformly[1], and the latter to (most of) the input tensors of GEMM operators (in both forward and backward passes):

$$\begin{aligned}\pi_{\mathrm{unif},\mathcal{C}}(t) &\triangleq \mathcal{C}(t,\mathsf{lo}) \text{ for all } t \in \mathsf{TS},\\ \pi_{\mathrm{op},\mathcal{C}}(t) &\triangleq \begin{cases}\mathcal{C}(t,\mathsf{lo}) & \text{if } t \in \{v_i, \theta_i, dv_{i+1}\} \text{ for some } i\\ & \quad \text{and } f_i \text{ is a GEMM operator (but not the first/last one)}\\ \mathcal{C}(t,\mathsf{hi}) & \text{otherwise,}\end{cases}\end{aligned} \tag{6.2}$$

where a GEMM operator refers to a general matrix multiplication operator which arises in, e.g., fully-connected or convolutional layers. A particular variant $\pi_{\mathrm{op}',\mathcal{C}}$ of $\pi_{\mathrm{op},\mathcal{C}}$ has received much attention as well [78], which assigns low-precision formats to (most of) the input and output tensors of GEMM operators: it is defined as $\pi_{\mathrm{op},\mathcal{C}}$ except that $\{v_i, \theta_i, dv_{i+1}\}$ in Eq. (6.2) is replaced by $\{v_i, \theta_i, v_{i+1}, dv_i, d\theta_i, dv_{i+1}\}$. We note that the precision assignments used in `apex.amp` and `torch.amp` [113, 121] correspond to $\pi_{\mathrm{op},\mathcal{C}}$ and $\pi_{\mathrm{op}',\mathcal{C}}$, respectively. For several choices of $\mathcal{C}$, these assignments have been shown to produce training accuracy similar to that by π_{fp32} on many datasets and models (see §6.1 and §6.5).

6.2.2 Memory-Accuracy Tradeoff Problem

We now introduce the following problem based on §6.2.1, to address the limitation of existing approaches for low-precision floating-point training discussed in §6.1:

Problem 6.1 (Memory-accuracy tradeoff)**.** Given training data $\{(x_i, y_i)\}$, a model and loss network $\mathcal{M}$ and $\mathcal{L}$, a precision-candidate assignment $\mathcal{C}$, and a lower bound $r \in [0, 1]$ on the low-precision ratio, find a precision assignment $\pi \in \Pi(\mathcal{C})$ that maximizes $\mathrm{acc}(\pi)$ subject to $\mathrm{ratio}_{\mathsf{lo}}(\pi) \geq r$. □

Here $\mathrm{acc}(\pi)$ denotes the accuracy of the model $\mathcal{M}$ when trained with π on $\{(x_i, y_i)\}$, and $\mathrm{ratio}_{\mathsf{lo}}(\pi)$ denotes the *low-precision ratio* of π, i.e., the portion of the tensors represented in low-precision under π, among all tensors appearing in a gradient computation:

$$\mathrm{ratio}_{\mathsf{lo}}(\pi) \triangleq \frac{\mathrm{size}(\{t \in \mathsf{TS} \mid \pi(t) = \mathcal{C}(t,\mathsf{lo})\})}{\mathrm{size}(\mathsf{TS})} \in [0, 1]$$

where $\mathrm{size}(T) \triangleq \sum_{t \in T} \mathrm{size}(t)$ denotes the total size (i.e., number of elements) of all tensors in $T \subseteq \mathsf{TS}$.[2] For instance, $\mathrm{ratio}_{\mathsf{lo}}(\pi_{\mathsf{hi}}) = 0$ and $\mathrm{ratio}_{\mathsf{lo}}(\pi_{\mathsf{lo}}) = 1$ for the all-high-precision assignment π_{hi} and the all-low-precision assignment π_{lo}. The problem asks for a precision assignment that maximizes training accuracy under a memory constraint, which is expressed as a fraction of the memory required to train the model using π_{hi}.

[1] For simplicity we define $\pi_{\mathrm{unif},\mathcal{C}}$ without the common exceptions for tensors near v_1 and/or v_{m+1}.

[2] As explained in §6.1, the low-precision ratio is a proxy for the reduction in memory as well as training time (because the low-precision ratio increases as the model aggregate decreases).

6.2.3 NP-Hardness of the Problem

We prove that the memory-accuracy tradeoff problem from §6.2.2 is NP-hard by showing that there is a polynomial-time reduction from the knapsack problem to this problem:

Theorem 6.2. *Problem 6.1 is NP-hard.*

Proof sketch. Recall the knapsack problem: given n items with weights $w_i \in \mathbb{N}$ and profits $p_i \in \mathbb{N}$ ($i \in [n]$), find a subset of the items that maximizes the total profit while its total weight does not exceed a given threshold $W \in \mathbb{N}$.

Given an instance (w, p, W) of the knapsack problem, we construct an instance of Problem 6.1 such that we get the following (informal) correspondence between the two: w_i corresponds to the size of the parameter tensor θ_i; p_i to the i-th component of the input data; W to the lower bound r on the low-precision ratio (in an inverse way); and selecting the i-th item corresponds to assigning a high-precision format to the tensor θ_i (and related tensors), which roughly decreases the low-precision ratio by w_i while increasing the accuracy of the model (after training) by p_i. Based on this informal correspondence, we formally prove that an optimal solution to the above instance of Problem 6.1 can be converted in linear time to an optimal solution to the given knapsack problem (w, p, W). That is, we have a linear-time reduction from the knapsack problem (which is NP-hard [79]) to Problem 6.1 which is therefore NP-hard. For a detailed proof, see Appendix C.1. □

Intuitively, the proof relies on two aspects of Problem 6.1: the size of the search space (i.e., $|\Pi(\mathcal{C})|$) is exponential in the size of the problem (especially $|\mathsf{TS}|$), and some values representable in a high-precision format underflow to 0 in a lower-precision format. Note that underflows are relevant in low-precision training: they frequently arise in practice, degrading the results of training [104]. The NP-hardness result indicates that it is unlikely any polynomial-time algorithm solves the problem exactly.

6.3 Algorithm

In this section, we propose a novel (heuristic) algorithm for the memory-accuracy tradeoff problem (§6.3.1), and a new technique to handle overflows arising in training (§6.3.2). We point out that the former algorithm finds an initial precision assignment *before* training starts, whereas the latter technique updates the current precision assignment *while* training proceeds.

6.3.1 Precision Demotion for Saving Memory

Consider an input to the memory-accuracy trade-off problem (Problem 6.1): a model and loss network $\mathcal{M} = (f_1, \ldots, f_n)$ and $\mathcal{L} = (f_{n+1}, \ldots, f_m)$, a precision-candidate assignment $\mathcal{C}$, and a lower bound r on the low-precision ratio. Given the input, our algorithm finds a precision assignment π in two steps (Algorithm 1).

Tensor grouping. We first *group* tensors in TS such that each group consists of all the tensors between two "adjacent" GEMM operators (see below for details). This grouping reduces the search space over precision assignments, from all of $\Pi(\mathcal{C})$ to a subset in which the same precision is assigned to the tensors in the same group. This specific grouping strategy is based on two observations: a majority of floating-point operations are carried out in GEMM operators, and it is standard (e.g., in PyTorch) to use the same precision for a forward tensor and its corresponding backward tensor.

Formally, we group tensors as follows. Let f_k and $f_{k'}$ $(k < k')$ be GEMM operators that are "adjacent", i.e., there is no GEMM operator in $\{f_{k+1}, \ldots, f_{k'-1}\}$. For each such $(f_k, f_{k'})$, we create a group $\{v_i, dv_i, \theta_j, d\theta_j \mid i \in (k, k'] \cap [m+1], j \in (k, k'] \cap [n]\}$. After that, we create two more groups for the remaining tensors: one for the tensors near v_1 and the other for tensors near v_{m+1}. As a result, we obtain a set of disjoint groups of tensors $\{T_1, T_2, \ldots\} \subseteq 2^{\mathsf{TS}}$.

Precision demotion. Given the groups of tensors, $T_1, T_2, \ldots$, we construct a precision assignment π as follows: initialize π to the all-high-precision assignment and update π by *demoting* the precision of all tensors in a group to low precision, one group at a time, until the low-precision ratio of π becomes greater than r. We demote the precision of groups in *decreasing* order of their sizes (i.e., the total number of elements in tensors); that is, the precision of a larger size group is demoted earlier. Formally, let $\{T'_1, T'_2, \ldots\}$ be the reordering of $\{T_1, T_2, \ldots\}$ such that $\mathrm{size}(T'_1) \geq \mathrm{size}(T'_2) \geq \cdots$. After initializing π by $\pi(t) = \mathcal{C}(t, \mathsf{hi})$ for all t, we iterate over $i \in \mathbb{N}$ and update π to $\pi(t) = \mathcal{C}(t, \mathsf{lo})$ for all $t \in T'_i$, until $\mathrm{ratio}_{\mathsf{lo}}(\pi) \geq r$ is first satisfied. The resulting π is the output of our algorithm.

The intuition behind using group size as the priority order for precision demotion is based on the fact that it is actually *optimal* in a very simplified setting. Suppose that an input x to the model $\mathcal{M}$ stores a quantity of information I and the forward computation of $\mathcal{M}$ is nothing but a process of extracting the information in the input into a small number of values, i.e., the tensor v_{n+1}. Assume that passing through each group $O_i = \{f_{k+1}, \ldots, f_{k'}\}$ of operators (corresponding to the group T_i of tensors) reduces the amount of information by a factor $\alpha_i \in (0, 1)$, and using low precision on the group T_i further reduces the amount of information by a constant factor $\beta \in (0, 1)$ for all i. Then, the amount of information left in v_{n+1} becomes $I \times (\alpha_1 \alpha_2 \cdots) \times \beta^l$, where l is the number of groups in low precision. In this simplified setting, maximizing the amount of information in v_{n+1} is equivalent to minimizing the number of groups in low precision, which is achieved precisely by demoting the largest groups first (when there is a constraint on the low-precision ratio). We show empirically (§6.4.4) that using the decreasing size order in precision demotion indeed produces better precision assignments than using other orders.

6.3.2 Precision Promotion for Handling Overflows

While our algorithm in §6.3.1 exerts a constraint on memory usage, it places no explicit constraint on training accuracy, and so not surprisingly for some models and datasets the resulting precision assignment causes training to *diverge*—accuracy decreases significantly and remains low after some

Algorithm 1: Computing π with precision demotion

Input: $(f_1, \ldots, f_n)$, $(f_{n+1}, \ldots, f_m)$, $\mathcal{C}$, r
/* Tensor grouping */
$k = 1$; $T_k = \emptyset$
for $i = 1$ **to** m **do**
  $T_k = T_k \cup \{v_i, dv_i\}$
  if $k \leq n$ **then** { $T_k = T_k \cup \{\theta_i, d\theta_i\}$ }
  if f_i is GEMM **then** { $k \mathrel{+}= 1$; $T_k = \emptyset$ }
end
/* Precision demotion */
$(T'_1, \ldots, T'_k) = \text{sort}(T_1, \ldots, T_k)$ by dec. size
$\pi(t) = \mathcal{C}(t, \mathsf{hi})$ for all $t \in \mathsf{TS}$
for $j = 1$ **to** k **do**
  if $\text{ratio}_{\mathsf{lo}}(\pi) \geq r$ **then** { **break** }
  $\pi(t) = \mathcal{C}(t, \mathsf{lo})$ for all $t \in T'_j$
end
return π

Algorithm 2: Training with precision promotion

Input: π, Θ, $\mathcal{C}$
/* Training loop */
Initialize weights $\boldsymbol{\theta}$
while training not finished **do**
  Compute the current gradient $\mathbf{d}\boldsymbol{\theta}$ using π
  Update $\boldsymbol{\theta}$ using $\mathbf{d}\boldsymbol{\theta}$
  /* Precision promotion */
  for forward $t \in \mathsf{TS}$ **do**
    if overflow_ratio$(t) > \Theta$ **then**
      $\pi(t) = \mathcal{C}(t, \mathsf{hi})$
    end
  end
end
return $\boldsymbol{\theta}$

point. We observe that when training begins to diverge (and a bit before that), many overflows occur in the rounding function of some tensors $\hat{v}_i$, i.e., an input tensor to the function $\text{rnd}_{\pi(v_i)}(\cdot)$ in Eq. (6.1) contains many elements whose magnitude is larger than the maximum representable number of the format $\pi(v_i)$ (Figure 6.6(a-b); §6.4.4). This rapid increase in overflows in individual tensors is a signal that training may diverge.

Precision promotion. Based on this observation, after each gradient computation we update the current precision assignment π by *promoting* to high precision (i.e., $\mathcal{C}(t, \mathsf{hi})$) any forward tensor t whose *overflow ratio* is greater than a given threshold $\Theta \in (0, 1)$; this updated precision assignment is used in the next gradient computation (Algorithm 2). Here the overflow ratio of $t \in \mathsf{TS}$ denotes the number of overflows arising in the rounding function of $\hat{t}$ in Eq. (6.1), divided by the number of elements in $\hat{t}$. We show empirically (§6.4.4) that training always converges using this technique and the additional memory cost of promotion is small (in our experiments, $< 3\%$ of the maximum model aggregate[3]). For the experiments, we use $\Theta = 0.01$; in fact we found that a wide range of values for Θ (0.1, 0.01, and 0.001) all work well. Note that this technique is not specific to our algorithm and can also be applied to other precision assignment methods.

We apply precision promotion only to forward tensors for two reasons. First, *dynamic loss scaling* [104, 113, 121, 147] already handles overflows in backward tensors, but not in forward tensors: loss scaling multiplies the backward loss tensor dv_{m+1} by a constant before performing backward computation, to scale up all backward tensors; the dynamic version adjusts the constant during training in a way that avoids overflows in backward tensors. Note that dynamic loss scaling does

[3]For each training where our methods (presented in §6.3) are used, we measure the model aggregate when the training starts and when it ends. We observe that the difference between the two values (averaged over four different runs) is at most 3% of the maximum model aggregate (i.e., the model aggregate when all tensors are in high precision).

not affect forward tensors at all. Second, we cannot use a similar idea to handle overflows in forward tensors, because forward tensors are not linear in the input tensor v_1 whereas backward tensors are linear in the backward loss tensor dv_{m+1} (by the linearity of differentiation).

Precision promotion incurs little if any computational overhead: checking whether a single rounding operation overflows is cheap, and we only apply rounding functions to the output tensor of an arithmetic-intensive operator (e.g., convolution and batch normalization), amortizing the cost of the overflow checks over a large number of other operations.

6.4 Experiments

In this section, we evaluate our precision assignment technique (developed in §6.3) on standard training tasks to answer three research questions:

- Does our technique explore the tradeoff between memory and accuracy and achieve a better tradeoff than existing (fixed) precision assignments (§6.4.3)?
- Are the two main components of our technique, precision demotion/promotion of larger/overflowing tensors, important for good performance (§6.4.4)?
- How can we choose the parameter r in our technique (i.e., a lower bound on the low-precision ratio) (§6.4.5)?

6.4.1 Implementation

We have implemented our precision assignment technique using PyTorch [118]. Given a model and loss network, and a dataset, our implementation takes as parameters a precision-candidate assignment $\mathcal{C}$ and a lower bound r on the low-precision ratio; it then automatically assigns precisions to tensors (appearing in training) according to our technique and uses those assigned precisions in gradient computations. To make these procedures automatic, our implementation works as follows:

- For each primitive operator in PyTorch (e.g., `torch.nn.Conv2d`), our implementation provides its wrapped version (e.g., `ext3.nn.Conv2d`) which records auxiliary information for our technique (e.g., floating-point format of input/output tensors) and applies proper rounding functions in forward/backward computations based on the auxiliary information. Models should now use the wrapped classes instead of the original ones.
- Our implementation first constructs a computation graph (of a given model and loss network) dynamically by running a forward computation on a minibatch of input data. The computation graph and other information (e.g., each tensor's size) are recorded in the wrapped classes.

- Using the auxiliary information just recorded, our implementation then constructs an initial precision assignment according to §6.3.1, and starts training with this assignment. During the training, our implementation uses the current precision in gradient computations, and updates it after each gradient computation according to §6.3.2. We record the precision assignment also in the wrapped classes to automatically apply proper rounding functions in gradient computations.

We simulate low-precision formats used in the experiments (e.g., 8-bit floats) and their operations, with 32-bit floats and 32-bit operations followed by rounding functions as described in Eq. (6.1); simulating low-precision formats is the standard methodology set by prior works on low-precision training [22, 48, 78, 105] and we simply follow this.[4] We implement the rounding functions based on the QPyTorch library [164], but a few extensions are required, e.g., to support exponent bias and signal overflows for dynamic loss scaling. We automatically apply these rounding functions after each primitive operator, by using PyTorch's hook feature (e.g., `nn.Module.register_*hook`).

6.4.2 Experiment Setups

Datasets and models. As benchmarks for our experiments, we use the image classification task and three datasets for the task: CIFAR-10 and CIFAR-100 [84], and ImageNet [133]; these task and datasets have been widely used in recent works on low-precision training as a standard choice [26, 127, 135, 155] and we simply follow this. For the task and datasets, we use four well-known models: SqueezeNet [72], ShuffleNet-v2 [97], MobileNet-v2 [136], and ResNet-18 [67]; they are chosen since models with relatively few weights, such as these, are generally known to be more difficult to train with low precision than those with more weights [147]. We considered other tasks (e.g., language modeling) and related models (e.g., RNN/transformer-based models) but did not include them in our experiments because substantial additional implementation effort orthogonal to our main contributions would be required: these models use some PyTorch operators that do not support per-tensor precision assignments,[5] so applying our technique to these models requires significant modifications to PyTorch internals.

Precision-candidate and precision assignments. For the experiments, we use the precision-candidate assignment $\mathcal{C}$ studied in [147], which uses 16-bit (and 8-bit) floats for high (and low) precision; in particular, $\mathcal{C}(t, \mathsf{hi}) = \mathsf{fp}(6, 9, 0)$ for all (forward/backward) tensors t, and $\mathcal{C}(t, \mathsf{lo}) = \mathsf{fp}(4, 3, 4)$ for all forward tensors t and $\mathsf{fp}(5, 2, 0)$ otherwise. We choose this particular $\mathcal{C}$ since it uses sub-32-bit floating-point formats for both low and high precision and the precision assignment $\pi_{\mathrm{op},\mathcal{C}}$ was shown

[4] The 8-bit formats used in our experiments (see §6.4.2) began to be supported natively in hardware very recently (by NVIDIA H100 GPU). But access to such hardware is still very limited (e.g., no major cloud services including AWS, Azure, and Google Cloud provide it), and these formats are not yet supported natively in software (e.g., PyTorch, TensorFlow, and cuDNN). Due to such lack of a hardware and software ecosystem natively supporting these formats, we chose to simulate them as in prior works.

[5] For instance, the PyTorch functions `nn.RNN` and `nn.MultiheadAttention` do not allow to change the precision of intermediate tensors (e.g., input/output tensors of each GEMM operator used in the functions) to user-defined formats (e.g., $\mathsf{fp}(4, 3, 4)$).

to achieve accuracy comparable to 32-bit training [147]. The three floating-point formats used in $\mathcal{C}$ have subnormals but no infinities and NaNs, which are rounded to the largest or smallest representable numbers. Since our technique is parameterized by a precision-candidate assignment, it is easily applied to other assignments as well.

We evaluate our technique by varying its parameter r (i.e., a lower bound on low-precision ratio) over deciles $r \in \{0, 0.1, 0.2, \ldots, 1\}$. We write $\pi_{\mathrm{ours},r}$ to denote the precision assignment chosen by our technique (described in §6.3) for a given r; e.g., $\pi_{\mathrm{ours},0}$ is the all-high-precision assignment, and $\pi_{\mathrm{ours},1}$ is the all-low-precision assignment equipped with our precision promotion technique (§6.3.2). Following [147], all precision assignments (including $\pi_{\mathrm{ours},r}$) in our experiments use high precision (i.e., 16 bits) for all backward weight tensors (i.e., $\hat{d\theta}_j$). For each precision assignment π, its low-precision ratio can change during training due to our precision promotion technique (when applied), so we compute the average of the ratio over all epochs and report this value as the low-precision ratio of π.

Other setups and compute time. All experiments were performed on NVIDIA V100 GPUs; total compute time for all experiments was 1,081 GPU days. We train all models in a standard way: we apply dynamic loss scaling (a standard technique used in low-precision floating-point training; see §6.3.2 for details) except for 32-bit training, and use standard settings (e.g., learning rate); see Appendix C.2 for details. Due to random variations in training, we perform four runs of training for each configuration and report the average and the range of measured quantities.

6.4.3 Comparison with Existing Precision Assignments

To compare our technique with existing precision assignments for floating-point training, we train each model with the following precision assignments: all-32-bit π_{fp32}, uniform π_{unif} [104], operator-based π_{op} [113, 147], its variant $\pi_{\mathrm{op}'}$ [78, 121], and ours $\pi_{\mathrm{ours},r}$ (see §6.2.1 and §6.4.2 for their definitions). We choose π_{unif}, π_{op}, and $\pi_{\mathrm{op}'}$ as baselines because existing precision assignments for floating-point training fall into one of the three assignments (or their variants) (see §6.1 and §6.5).

We train the four models mentioned in §6.4.2 on CIFAR-10 and CIFAR-100, and ShuffleNet-v2 on ImageNet. We also train smaller variants of the four models (which are more difficult to train with low precision) on CIFAR-100. We obtain these variant models by following [147], i.e., by applying a well-known approach for model reduction that uses a parameter called the *width multiplier* [69]: each variant model reduces the number of channels in most tensors by a width multiplier; we use three values $\{0.5, 0.25, 0.1\}$ for the width multiplier. We train just one model on ImageNet due to the large amount of computation involved: for each model, 44 training runs (11 choices for r and 4 runs for each choice) are required for $\pi_{\mathrm{ours},r}$ and each run on ImageNet takes nearly a half day with 16 GPUs. We use ShuffleNet-v2 for ImageNet since the model shows interesting memory-accuracy tradeoffs when trained on the (smaller) CIFAR datasets.

ImageNet. Figure 6.3 presents training results of ShuffleNet-v2 on ImageNet: its left graph plots the average training trajectory for each precision assignment, and its right graph shows how

each precision assignment trades off between memory and accuracy, where memory is represented (inversely) by the low-precision ratio of the assignment (which is averaged over all epochs; see §6.4.2) and accuracy is the best test accuracy of the model during training. Each point in the right graph shows the average accuracy of four runs of training, while the shaded area shows the variation in accuracy among those four training runs.

Figure 6.3 shows three points. First, as the parameter r increases, the average accuracy drop of $\pi_{\text{ours},r}$ from π_{fp32} increases (up to 5%). In contrast, π_{unif} and $\pi_{\text{op}'}$ have a much larger average accuracy drop (more than 30%), as some training runs diverge when π_{unif} and $\pi_{\text{op}'}$ are used. Second, the tradeoff given by $\pi_{\text{ours},r}$ is better (i.e., closer to Pareto-optimal) than by π_{op}: $\pi_{\text{ours},r}$ for $r \in \{0.3, 0.4\}$ has both higher accuracy and larger low-precision ratio (i.e., memory reduction) than π_{op}. In particular, $\pi_{\text{ours},0.4}$ has 1.6× the memory reduction of π_{op}. Third, $\pi_{\text{ours},r}$ provides options that π_{op} cannot (which has an accuracy drop of >1%). If we want accuracy closer to π_{fp32}, say within 0.5%, we can use $\pi_{\text{ours},0.2}$ with 2.6% more memory than π_{op}. If we can tolerate a larger accuracy loss, say ≈ 3%, then we can use $\pi_{\text{ours},0.7}$ with 2.9× the memory reduction of π_{op}.

CIFAR-10/100. Figure 6.4 presents the memory-accuracy tradeoffs of precision assignments for the four models on CIFAR-10 and CIFAR-100, and their smaller variants (with width multiplier 0.25) on CIFAR-100. The results for other smaller variants are similar and included in Figure C.3 (see Appendix C.3.1).

The conclusions from Figure 6.3 hold for Figure 6.4: $\pi_{\text{ours},r}$ provides a range of options by varying r and exhibits a better tradeoff than π_{unif}, π_{op}, and $\pi_{\text{op}'}$ in almost all cases. We give a detailed comparison as follows. First, in half of all 12 plots, π_{unif} shows a similar tradeoff to $\pi_{\text{ours},1}$. But in the remaining half, π_{unif} has an accuracy drop much larger than all other precision assignments including $\pi_{\text{ours},r}$, since using π_{unif} often makes training diverge while using, e.g., $\pi_{\text{ours},1}$ does not do so. Second, in all but two plots, $\pi_{\text{ours},r}$ shows a strictly better tradeoff than π_{op}: $\pi_{\text{ours},r}$ has noticeably larger ($> 2\times$) memory reduction than π_{op} while maintaining similar accuracy. Even in the two plots, $\pi_{\text{ours},r}$ has a tradeoff very close to π_{op}. Note that in three plots, π_{op} has an accuracy drop of >1% while $\pi_{\text{ours},r}$ provides several options that have smaller accuracy drops and more memory savings at the same time. Third, $\pi_{\text{ours},r}$ shows a strictly better (or similar) tradeoff than $\pi_{\text{op}'}$ in all but two (or two) plots. Note that $\pi_{\text{op}'}$ has accuracy smaller than π_{op} in all but one plots. Also it has an accuracy drop of >1% in half of all plots, and sometimes makes training even diverge (in one plot here and three other plots in Figure C.3).

Additional results. To isolate the effect of our precision promotion technique on the above results (Figures 6.3, 6.4 and C.3), we compare our precision assignments and the baseline assignments while applying the precision promotion to all of them, and present the results in Appendix C.3.1 (Figures C.6–C.8). Two observations can be made in these results: for each precision assignment, (i) if training diverged without the precision promotion, applying the precision promotion prevents such divergence and produces much higher accuracy; (ii) otherwise, the accuracy and the low-precision ratio remain similar regardless of using the precision promotion. These observations lead to the

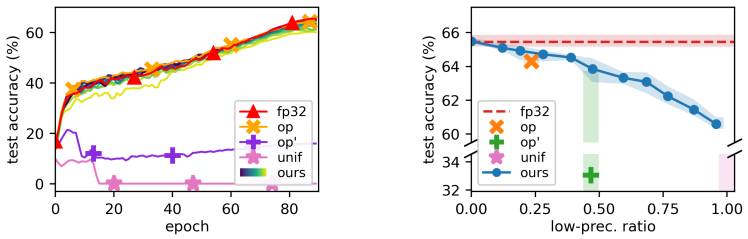


Figure 6.3: Results of training ShuffleNet-v2 on ImageNet with π_{fp32}, π_{unif} [104], π_{op} [147], $\pi_{\text{op}'}$ [78], and $\pi_{\text{ours},r}$. Left: Each line shows the average training trajectory for each precision assignment; $\pi_{\text{ours},r}$ is colored from navy to yellow (darker for smaller r). A zoomed-in version of this plot can be found in Appendix C.3.1. Right: Each point shows the memory-accuracy tradeoff of each precision assignment; a red-dashed line shows the accuracy of π_{fp32}; and shaded areas show the variation among four training runs. In the right figure, top-right points are better than bottom-left ones. Observe that there are ●s above and to the right of ✕ and ✚, respectively. ★ is missing as its y-value is too small.

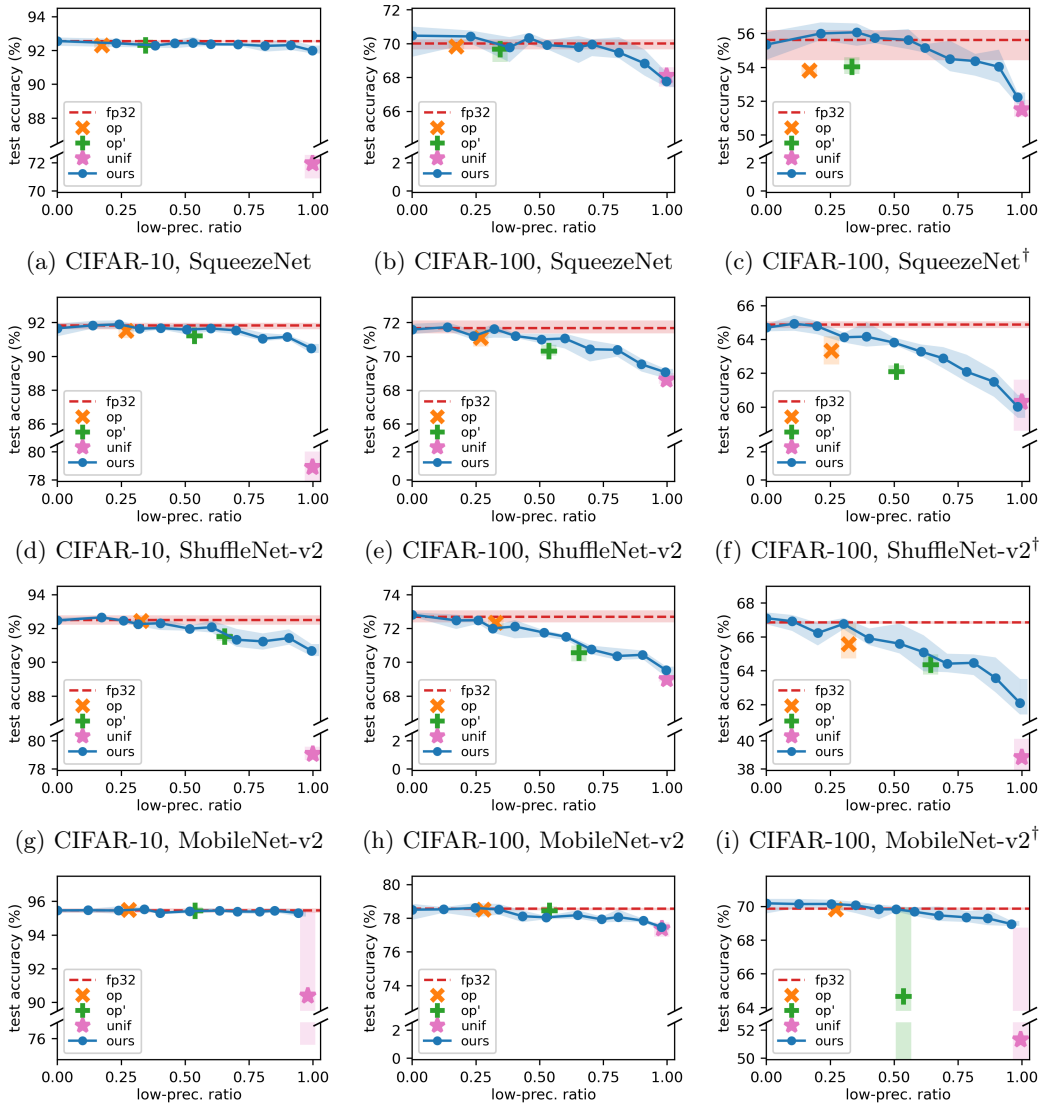
same conclusion as in the above: our assignments provide similar or better tradeoff between memory and accuracy than the baseline assignments, even when the latter are equipped with our precision promotion technique. In addition, these observations also indicate that our precision promotion technique can effectively handle divergence in training (see §6.4.4 for more results on this).

6.4.4 Ablation Study: Precision Demotion and Promotion

Precision demotion. To evaluate the decision to use precision demotion in decreasing-size order, we train the four models on CIFAR-100 with $\pi_{\text{ours},r}$, $\pi_{\text{ours[inc]},r}$ (which demotes tensor groups in increasing-size order) and $\pi_{\text{ours[rand]},r}$ (which demotes tensor groups in random order). For $\pi_{\text{ours[rand]}}$, three different random orders are used in each case. The results, presented in Figure 6.5 (and Appendix C.3.2), show that the order of precision demotion has a significant impact on the resulting memory-accuracy tradeoff, and that decreasing order provides the best results in nearly all cases. Increasing order consistently shows the worst results, suggesting our intuition (given in §6.3.1) for choosing decreasing order has some basis in reality.

Precision promotion. To understand whether precision promotion of overflowing tensors is important to our technique, we train ShuffleNet-v2 on ImageNet using $\pi_{\text{ours[no-promo]},r}$ which does not promote tensors. The results, presented in Figure 6.6(a), show that several training trajectories diverge in early epochs and fail to recover afterwards. Figure 6.6(b) plots the top-5 tensor overflow ratios for the highlighted trajectory in Figure 6.6(a). The overflow ratios first spike about when divergence occurs around epoch 11. A closer look shows that the spike in overflow ratio occurs shortly before divergence, and starts first in a few tensors and then propagates to others. These observations indicate that an excessive number of overflows in a few tensors are the cause of the training divergence.

Finally, Figure 6.6(c-d) shows that precision promotion is effective at preventing the divergence of



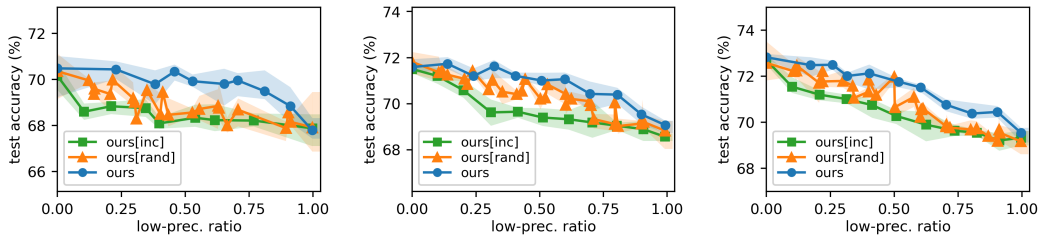
(a) CIFAR-10, SqueezeNet (b) CIFAR-100, SqueezeNet (c) CIFAR-100, SqueezeNet†

(d) CIFAR-10, ShuffleNet-v2 (e) CIFAR-100, ShuffleNet-v2 (f) CIFAR-100, ShuffleNet-v2†

(g) CIFAR-10, MobileNet-v2 (h) CIFAR-100, MobileNet-v2 (i) CIFAR-100, MobileNet-v2†

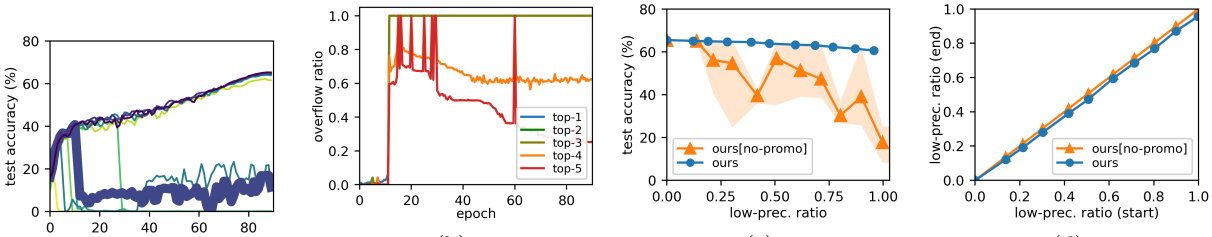
(j) CIFAR-10, ResNet-18 (k) CIFAR-100, ResNet-18 (l) CIFAR-100, ResNet-18†

Figure 6.4: Memory-accuracy tradeoffs of π_{unif} [104], π_{op} [147], $\pi_{\text{op}'}$ [78], and $\pi_{\text{ours},r}$ for four models and their smaller variants on CIFAR-10 and CIFAR-100. The variant models have width multiplier 0.25 and are marked by †. Top-right points are better than bottom-left ones. In all but three plots, there are ●s above and to the right of ✕ and ✚, respectively; even in the three plots (g,h,k), ●s have almost the same tradeoffs to ✕ and ✚. In half of all plots, ★ has much smaller y-values than other points. The training trajectories for the above plots and the results of other smaller models are in Appendix C.3.1.



(a) SqueezeNet (b) ShuffleNet-v2 (c) MobileNet-v2

Figure 6.5: Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$, $\pi_{\text{ours[inc]},r}$, and $\pi_{\text{ours[rand]},r}$ for three models on CIFAR-100. Observe that ●s are above and to the right of other points in nearly all cases. The results of ResNet-18 are in Appendix C.3.2.



(a) (b) (c) (d)

Figure 6.6: Training ShuffleNet-v2 on ImageNet with $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. (a) Training trajectories of $\pi_{\text{ours[no-promo]},r}$ for different r; colors denote r values (darker for smaller r). (b) Top-5 overflow ratios of tensors at each epoch, for the highlighted trajectory in (a); the largest ratio is blue and the fifth largest red. (c) Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. (d) Low-precision ratio when training ends vs. when training starts, for $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. The results on CIFAR-10 are in Appendix C.3.2.

training while sacrificing only a small amount of memory reduction. The figure shows ShuffleNet-v2 on ImageNet trained using our technique with and without precision promotion. Figure 6.6(c) shows that without precision promotion large accuracy drops occur due to divergence, whereas with precision promotion training converges. Figure 6.6(d) shows that the total size of tensors promoted to high precision is small for all r values. See Appendix C.3.2 for similar results for CIFAR-10.

6.4.5 Choosing the Value of r

The time and space savings of our method are most significant when a model is regularly retrained, which commonly occurs when new data is periodically incorporated into an existing model. Assuming that new data has a similar distribution to existing data, we can choose a single r (a parameter in our method) by conducting one set of experiments where we train with π_{fp32} and $\pi_{\text{ours},r}$ for different r and then choose the r value that maximizes model aggregate savings while still having an acceptable drop in accuracy.

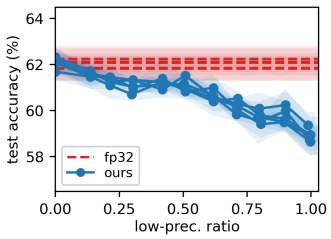


Figure 6.7: Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$ for ShuffleNet-v2 on ImageNet-200-i ($i \in [5]$).

To simulate this scenario, we create five datasets ImageNet-200-i ($i \in [5]$) as follows, so that each of them contains different but similar data: randomly select 1/5 of the classes in ImageNet (which has 1000 classes in total), and split the training data of each class evenly into five new datasets.

For each ImageNet-200-i, we train ShuffleNet-v2 with π_{fp32} and $\pi_{\text{ours},r}$ and present the results in Figure 6.7. Based on the tradeoff results of $\pi_{\text{ours},r}$, we can choose $r = 0.4$ if we desire an average of $<$ 1% accuracy drop from π_{fp32}, and we can choose $r = 0.9$ if an average $\approx 3\%$ accuracy drop is tolerable. We make two more observations: the tradeoff result of $\pi_{\text{ours},r}$ is similar across all five datasets even though each dataset is different, and for each r the variance in the accuracy of $\pi_{\text{ours},r}$ from different datasets and runs of training is similar to that of π_{fp32}. Thus we expect that on a new but similar dataset, $\pi_{\text{ours},r}$ would have an accuracy drop similar to Figure 6.7 with acceptable variance.

6.5 Related Work

Low-precision floating-point training has been extensively studied since the work of [104]. One active research direction is to select appropriate floating-point formats (or their variants) for low- and high-precision numbers in training. Various floating-point formats have been proposed, including FP16 [104], BF16 [78], FP8 [105, 155], HFP8 [147], and FP6 [26], along with some variants such as HBFP [44], S2FP8 [22], and BM [48]. Recently, the problem of automatically selecting such floating-point formats has been considered [160]. Another research direction is to develop algorithmic techniques that improve training accuracy under low precision: e.g., [14, 134, 161, 163]. This chapter is orthogonal and complementary to all these prior works: they consider various floating-point formats or training algorithms but use a *fixed* precision assignment, which is either the uniform or operator-based assignment (or their variants); this chapter explores *various* precision assignments once floating-point formats and training algorithms are fixed (e.g., based on the prior works). The tradeoff between memory and accuracy in training is also considered in [160], but the work differs from ours: they vary *floating-point formats* when a precision assignment is fixed, while we vary *precision assignments* when floating-point formats are fixed.

Low-precision fixed-point training uses fixed-point formats as a low-precision representation instead of a floating-point format. Some works use fixed-point formats for forward tensors and

floating-point formats for backward tensors: e.g., [27, 32, 74, 148, 162]. Other works use only fixed-point formats for all tensors: e.g., [5, 38, 58, 127, 135, 159, 165, 166]. Among all these works, some consider various mixed precision assignments with different bitwidth (fixed-point) formats: [135, 165]; but they are not applicable to our context (i.e., floating-point training) since they rely on some properties of *fixed-point* formats that do not hold for *floating-point* formats (e.g., all numbers in a given format are equally distributed). The approach taken in [127] is orthogonal and complementary to ours: they use only the *uniform* precision assignment, but change the underlying low-precision formats during training; we consider various *mixed* precision assignments, but fix the underlying low-precision formats during training.

Low-precision inference, often called neural network quantization (in a narrow sense), aims at reducing the latency or memory of neural network inference (instead of training) by using low-precision numbers [110]. Existing approaches typically assume a pre-trained model and try to find low-precision formats for each part of the inference computation, either by retraining the model (called quantization-aware training) or without any retraining (called post-training quantization); see, e.g., [51, 125] for surveys. Some works on inference consider various mixed precision assignments, but they are not applicable to our context: they focus on making inference more efficient and usually assume a pre-trained model; we focus on making training more efficient and aim at learning a model from scratch.

Floating-point tuning is another related topic, which considers the following problem: given a program, assign appropriate formats (among given candidates) to the program's floating-point variables such that the program's output has an error smaller than a given threshold for all given inputs, while also maximizing performance [24, 57, 103, 131, 132]. This problem is different from the problem we focus on: the former considers the *floating-point error* after a single run of a program, while we consider the *training accuracy* after a large number of runs of a program (i.e., a gradient computation) where each run affects the next run; further, the former considers *general-purpose* programs, while we consider *deep learning* programs and exploit their unique features.

6.6 Conclusion

In this chapter, we formally introduce the memory-accuracy tradeoff problem to explore better mixed precision assignments for low-precision floating-point training, and prove the NP-hardness of the problem. We then present a novel precision assignment technique, as a heuristic solution to the tradeoff problem, that proposes assignments based on a single parameter denoting a desired upper bound on the model aggregate. We also present a novel technique that handles an excessive number of overflows arising in training while using a small amount of additional memory. We demonstrate that the mixed precision assignments found by our method do explore the tradeoff between memory and training accuracy, and outperform existing precision assignment methods.

Chapter 7

Conclusion

In this dissertation we considered three classes of continuous computations used in the real world and studied the discrepancy between the theory and practice of these computations, especially the use of reals in theory and floats in practice. First, in Chapters 3 and 4, we focused on computations that implement math libraries using floats and presented automatic techniques to formally verify their correctness. Second, in Chapter 5, we focused on computations that calculate derivatives of neural networks at floating-point inputs and showed theoretical results on their correctness. Third, in Chapter 6, we focused on computations that train deep neural networks using floats and presented a systematic way to accelerate them using lower-precision floats.

Our work opens up several directions for future research. First, in Chapters 3 and 4, we did not prove the full correctness of the math libraries. For instance, Intel's `sin` and `tan` are claimed to have the 1 ulp error bound over the entire input range $[-\min \mathbb{F}, \max \mathbb{F}]$; however, we applied our automatic techniques to some subsets of the input range (e.g., $[-\pi, \pi]$) and proved an error bound of 13.33 ulps for one case (i.e., `tan` over $[\frac{17\pi}{64}, \frac{\pi}{2})$). Developing new automatic techniques to prove the full correctness is left as future work. Second, in Chapter 5, we proved our correctness results on AD under some assumptions. For example, we considered neural networks consisting of alternating analytic pre-activation functions and pointwise continuous activation functions; hence, our results might not be applicable if a network contains non-pointwise activation functions (e.g., MaxPool) or a residual connection bypassing a non-analytic activation function (e.g., ReLU). Relaxing such assumptions is interesting future work. Finally, in Chapter 6, we simulated low-precision floats and operations in software (instead of handling them natively in hardware) when performing low-precision training, as prior works have also done. Hence, the potential speedup of our method was not directly measured, though we do expect speedups to be proportional to the reduction in the model aggregate. Performing such experiments on recent/future hardware that natively supports more low-precision formats is yet another direction for future work.

Appendix A

Appendix for Chapter 4

A.1 Complete Definitions and Rules

A.1.1 Definition of Operations on Abstractions

In this subsection, assume that $\mathcal{A}_{\bar{\delta}}(x)$ denotes $a(x) + \sum_i b_i(x)\delta_i$ and δ_i ranges over $[-\Delta_i, \Delta_i]$.

$\boxed{\mathcal{A}_{\bar{\delta}}(x) \circledast \mathcal{A'}_{\bar{\delta}}(x)}$ $(* \in \{+, -, \times, /\})$

$$\begin{aligned}
\mathcal{A}_{\bar{\delta}}(x) \boxplus \mathcal{A'}_{\bar{\delta}}(x) &\triangleq \mathcal{A}_{\bar{\delta}}(x) + \mathcal{A'}_{\bar{\delta}}(x), \\
\mathcal{A}_{\bar{\delta}}(x) \boxminus \mathcal{A'}_{\bar{\delta}}(x) &\triangleq \mathcal{A}_{\bar{\delta}}(x) - \mathcal{A'}_{\bar{\delta}}(x), \\
\mathcal{A}_{\bar{\delta}}(x) \boxtimes \mathcal{A'}_{\bar{\delta}}(x) &\triangleq \mathit{linearize}(\mathcal{A}_{\bar{\delta}}(x) \times \mathcal{A'}_{\bar{\delta}}(x)), \\
\mathcal{A}_{\bar{\delta}}(x) \boxslash \mathcal{A'}_{\bar{\delta}}(x) &\triangleq \mathcal{A}_{\bar{\delta}}(x) \boxtimes \mathit{inv}(\mathcal{A'}_{\bar{\delta}}(x)).
\end{aligned}$$

$\mathit{linearize}(\cdot)$ and $\mathit{inv}(\cdot)$ are defined as:

$$\begin{aligned}
\mathit{linearize}\left(a(x) + \sum_i b_i(x)\delta_i + \sum_{i,j} b_{i,j}(x)\delta_i\delta_j\right) &\triangleq a(x) + \sum_i b_i(x)\delta_i + \sum_{i,j} b_{i,j}(x)\delta'_{i,j}, \\
\mathit{inv}(\mathcal{A}_{\bar{\delta}}(x)) &\triangleq \frac{1}{a(x)} + \frac{1}{a(x)}\delta'' \quad \text{(assumes } \Delta' < 1\text{)},
\end{aligned}$$

where $\delta'_{i,j} = \mathit{fresh}(\Delta_i\Delta_j)$ and $\delta'' = \mathit{fresh}(\frac{\Delta'}{1-\Delta'})$, and $\Delta' \in \mathbb{R}_{\geq 0}$ is computed as

$$\Delta' = \sum_i \max_{x \in X} \left|\frac{b_i(x)}{a(x)}\right| \cdot \Delta_i.$$

$\boxed{\mathcal{A}_{\bar{\delta}}(x) \circledast \delta'}$ $(* \in \{+, \times\})$

$$\mathcal{A}_{\bar{\delta}}(x) \boxplus \delta' \triangleq \mathcal{A}_{\bar{\delta}}(x) \boxplus \mathcal{A'}_{\bar{\delta}}(x) \quad \text{where } \mathcal{A'}_{\bar{\delta}}(x) = 0 + 1 \cdot \delta',$$

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes \delta' \triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A'}_{\vec{\delta}}(x) \quad \text{where } \mathcal{A'}_{\vec{\delta}}(x) = 0 + 1 \cdot \delta'.$$

$\boxed{\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta')}$

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta') \triangleq a(x) + a(x)\delta' + \sum_{i \in R} b_i(x)\delta'_i + \sum_{i \notin R} (b_i(x)\delta_i + b_i(x)\delta''_i)$$

where δ' ranges over $[-\Delta', \Delta']$, $R = \{i : \mathsf{preserve}(\delta_i) = \texttt{false}\}$, $\delta'_i = \mathit{fresh}(\Delta_i(1 + \Delta'))$ $(i \in R)$, and $\delta''_i = \mathit{fresh}(\Delta_i \Delta')$ $(i \notin R)$.

$\boxed{\mathit{compress}(\mathcal{A}_{\vec{\delta}}(x))}$

$$\mathit{compress}(\mathcal{A}_{\vec{\delta}}(x)) \triangleq a(x) + a(x)\delta' + \sum_{i \notin R \cap S} b_i(x)\delta_i \quad \text{where } \delta' = \mathit{fresh}\left(\sum_{i \in R \cap S} \gamma_i\right).$$

Here $R = \{i : \mathsf{preserve}(\delta_i) = \texttt{false}\}$, and $\gamma_i \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ and the set S are computed as:

$$\gamma_i = \max_{x \in X} \left| \frac{b_i(x)}{a(x)} \right| \cdot \Delta_i \qquad \text{and} \qquad S = \left\{ i : \frac{\gamma_i}{\varepsilon} \leq \tau \right\}$$

where $\tau \in \mathbb{R}_{\geq 0}$ is a constant.

A.1.2 Rules for Constructing Abstractions

Basic rules:

$$\frac{e \in \mathit{dom}(\mathcal{K}) \qquad \mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, \_, \_, \_)}{(\mathcal{K}, e) \rhd (\mathcal{K}, \mathcal{A}_{\vec{\delta}})} \textsc{Load}'$$

$$\frac{}{(\mathcal{K}, c) \rhd (\mathcal{K}[c \mapsto (c, \texttt{false}, \sigma(c), \mu(c))], c)} \text{ R1}' \qquad \frac{}{(\mathcal{K}, x) \rhd (\mathcal{K}[x \mapsto (x, \texttt{false}, 53, \mu(x))], x)} \text{ R2}'$$

$$\frac{\begin{array}{c} \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) \\ \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \qquad \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) \qquad \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) \qquad * \in \{+, -\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A'}_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(\varepsilon) \\ \mathcal{A'}_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta')) \\ \sigma' = \min\{\sigma, 53\}, \ \mu' = \max\{\mathrm{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A'}_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}} \text{ R14}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{\times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mu \geq 2^{-1022} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(\varepsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \circledast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mathrm{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}} \text{ R15}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{\times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mu < 2^{-1022} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(\varepsilon),\ \delta'' = \mathit{fresh}(\varepsilon') \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \circledast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta') \boxplus \delta'') \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mathrm{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}} \text{ R3}'$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ (\mathcal{K}, \mathit{bit\text{-}mask}(e_1, B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \mu = \mathcal{E}(\mathit{bit\text{-}mask}(x, B))(\mathrm{fl}^-(\mu_1)) & \mu \geq 2^{-1022} \end{array}}{(\mathcal{K}, \mathit{bit\text{-}mask}(e_1, B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(2^{-52+B}) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1+\delta')) \\ \sigma' = \min\{\sigma_1, 53-B\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_1[\mathit{bit\text{-}mask}(e_1, B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}} \text{ R4}'\text{-1}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ (\mathcal{K}, \mathit{bit\text{-}mask}(e_1, B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \mu = \mathcal{E}(\mathit{bit\text{-}mask}(x, B))(\mathrm{fl}^-(\mu_1)) & \mu < 2^{-1022} \end{array}}{(\mathcal{K}, \mathit{bit\text{-}mask}(e_1, B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(2^{-52+B}),\ \delta'' = \mathit{fresh}(2^{-1074+B}) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1+\delta') \boxplus \delta'') \\ \sigma' = \min\{\sigma_1, 53-B\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_1[\mathit{bit\text{-}mask}(e_1, B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}} \text{ R4}'\text{-2}$$

Rules for simple exact operations:

$$\frac{\begin{array}{ccc} (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & & \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, 0) & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & * \in \{+, -\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}_{1,\vec{\delta}}), \text{ where } \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}_{1,\vec{\delta}}, \texttt{true}, \sigma_1, \mu_1)]} \text{ R5}'$$

$$\frac{\begin{array}{ccc}\mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & & \\ \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(\times, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \exists n \in \mathbb{Z}. \forall x \in X.\ \mathcal{A}_{2,\vec{\delta}}(x) = 2^n \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(\times, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & n \geq -1075 - \mathrm{expnt}(\mu_1) + \sigma_1\end{array}}{(\mathcal{K}, e_1 \otimes e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathcal{A}_{1,\vec{\delta}} \times 2^n \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \otimes e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true}, \sigma', \mu')] \end{cases} \text{R6}'$$

$$\frac{\begin{array}{cc}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, c_1) & \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, c_2) & c' = c_1 \circledast c_2 \qquad * \in \{+, -, \times, /\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', c'),\ \text{where } \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (c', c' \texttt{ == } c_1 * c_2, \sigma(c'), \mu(c'))]} \text{R7}'$$

Rules for applying Sterbenz's theorem:

$$\frac{\begin{array}{ccc}\mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & & \\ \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(-, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \tfrac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \geq 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(-, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \leq 0\end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true}, \sigma', \mu')] \end{cases} \text{R8}'$$

$$\frac{\begin{array}{ccc}\mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & & \\ \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(-, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \geq 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(-, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \tfrac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \leq 0\end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true}, \sigma', \mu')] \end{cases} \text{R9}'$$

Rules for applying Dekker's theorem:

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(+, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(+, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mathsf{hasDekker}(e_1 \oplus e_2) \end{array}}{(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \mathit{fresh}(\varepsilon, \texttt{true}) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta')) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mathrm{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \oplus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}\langle\delta'\rangle, \sigma', \mu')] \end{cases}} \text{ R10}'$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\mathcal{A}_{1,\vec{\delta}}, \_, \_, \_) & \mathcal{K}_1(e_1 \oplus e_2) = (\_, \texttt{false}\langle\delta'\rangle, \_, \_) \\ (\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}_1, \_) & \mathcal{K}_1(e_2) = (\mathcal{A}_{2,\vec{\delta}}, \_, \_, \_) & \min_{x,\vec{\delta}} |\mathcal{A}_{1,\vec{\delta}}| \geq \max_{x,\vec{\delta}} |\mathcal{A}_{2,\vec{\delta}}| \end{array}}{(\mathcal{K}, e_1 \oplus e_2 \ominus e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes \delta') \\ \mathcal{K}' = \mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, 53, 2^{-1074})] \end{cases}} \text{ R11}'$$

$$\frac{(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}_1, \_) \qquad \mathcal{K}_1(e_1 \oplus e_2) = (\_, \texttt{true}, \_, \_)}{(\mathcal{K}, e_1 \oplus e_2 \ominus e_1 \ominus e_2) \rhd (\mathcal{K}', 0), \text{ where } \mathcal{K}' = \mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (0, \texttt{true}, 0, \infty)]} \text{ R12}'$$

Rule for using $\sigma(\cdot)$:

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{+, -, \times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \sigma \leq 53 \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes \mathcal{A}_{2,\vec{\delta}}) \\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true}, \sigma, \mu')] \end{cases}} \text{ R13}'$$

Appendix B

Appendix for Chapter 5

B.1 Formal Setup

In the appendix, we use the following notation. For $A \subseteq \mathbb{R}^n$, $int(A)$ and $bd(A)$ denote the interior and the boundary of A.

B.1.1 Piecewise-Analytic Functions

Definition B.1. For $A \subseteq \mathbb{R}^n$, define $pbd(A)$ as

$$pbd(A) \triangleq A \setminus int(A).$$

We call $pbd(A)$ the *proper boundary of* A. Note that $pbd(A) = bd(A) \cap A$ holds for any A.

Definition B.2. A function $f : \mathbb{R} \to \mathbb{R}$ is *piecewise-differentiable* (or *piecewise-C^1*) if there exist $n \in \mathbb{N}$, a partition $\{A_i\}_{i\in[n]}$ of $\mathbb{R}$ consisting of non-empty intervals, and differentiable (or C^1) functions $\{f_i : \mathbb{R} \to \mathbb{R}\}_{i\in[n]}$ such that $f = f_i$ on A_i for all $i \in [n]$. We call such $\{(A_i, f_i)\}_{i\in[n]}$ a *piecewise-differentiable (or piecewise-C^1) representation of* f. Moreover, for an extended derivative $g : \mathbb{R} \to \mathbb{R}$ of f, we say that the representation $\{(A_i, f_i)\}_{i\in[n]}$ *defines* g if $g = Df_i$ on A_i for all $i \in [n]$. We define a *piecewise-analytic representation of* f in a similar way.

Lemma B.3. *Let $\{A_i\}_{i\in S}$ be any partition of $\mathbb{R}^n$. Then,*

$$\bigcup_{i\in S} bd(A_i) = \bigcup_{i\in S} pbd(A_i). \tag{B.1}$$

Proof. The direction $\supseteq$ is clear, since $pbd(X) \subseteq bd(X)$ for any $X \subseteq \mathbb{R}^n$. To prove the other direction $\subseteq$, it suffices to show that for any $i \in S$ and $x \in bd(A_i)$, we have $x \in pbd(A_j)$ for some $j \in S$. Here we assume $x \notin A_i$; if not, choosing $j = i$ completes the proof. Let $j \in S$ be the index with $x \in A_j$,

where such j always exists since $\{A_i\}_{i\in S}$ is a partition of $\mathbb{R}$. Then, it suffices to show $x \in bd(A_j)$, because this and $x \in A_j$ implies $x \in pbd(A_j)$. To prove $x \in bd(A_j)$, consider any open neighborhood $U \subseteq \mathbb{R}^n$ of x. Then, there is $x' \in U \cap A_i$ (by $x \in bd(A_i)$ and $x \notin A_i$). This implies that $x' \notin U \cap A_j$ (by $A_i \cap A_j = \emptyset$ from $i \neq j$) and $x \in U \cap A_j$ (by $x \in A_j$). Hence, we have $x \in bd(A_j)$ as desired. □

Theorem B.4. *Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous, piecewise-analytic function, and $g : \mathbb{R} \to \mathbb{R}$ be an extended derivative of f. Then, the following hold.*

(i) There is a piecewise-differentiable representation $\{(A_i, f_i)\}_{i\in[n]}$ of f that defines g and satisfies the following:

$$\bigcup_{i\in[n]} bd(A_i) = \bigcup_{i\in[n]} pbd(A_i) = \mathsf{ndf}(f).$$

(ii) If g is consistent, there is a piecewise-C^1 representation $\{(A_i, f_i)\}_{i\in[n]}$ of f that defines g and satisfies the following:

$$\bigcup_{i\in[n]} bd(A_i) = \bigcup_{i\in[n]} pbd(A_i) = \mathsf{ncdf}(f), \qquad int(A_i) \neq \emptyset \text{ for all } i \in [n],$$

where $\mathsf{ncdf}(f) \subseteq \mathbb{R}$ denotes the set of real numbers at which $f : \mathbb{R} \to \mathbb{R}$ is not continuously differentiable.

Proof. We prove the two claims as follows. Note that by Lemma B.3, we do not need to prove the equality between the union of $bd(A_i)$ and that of $pbd(A_i)$ in the claims.

Claim (i). Let $\{(\widetilde{A}_i, \widetilde{f}_i)\}_{i\in[\widetilde{n}]}$ be a piecewise-analytic representation of f that defines g and satisfies

$$(\widetilde{A}_1, \ldots, \widetilde{A}_{\widetilde{n}}) = \Big((x_0, x_1), \ldots, (x_k, x_{k+1}), \{x_1\}, \ldots, \{x_k\}\Big)$$

for some $-\infty = x_0 < x_1 < \cdots < x_k < x_{k+1} = \infty$. Such a representation always exists, because f is piecewise-analytic and g is an extended derivative of f. Note that $\mathsf{ndf}(f) \subseteq \{x_1, \ldots, x_k\}$ because f is differentiable on (x_{i-1}, x_i) for all $i \in [k+1]$ (since $\widetilde{f}_i$ is analytic and it coincides with f on $\widetilde{A}_i = (x_{i-1}, x_i)$). We then construct $\{(A_i, f_i)\}_{i\in[n]}$ from $\{(\widetilde{A}_i, f_i)\}_{i\in[\widetilde{n}]}$, by merging all adjacent intervals $\widetilde{A}_i$ (and associated functions $\widetilde{f}_i$) into a single interval (and a single function) such that the class of the singleton interval in $\{A_i\}$ are the same as $\{\{x\} \mid x \in \mathsf{ndf}(f)\}$. Then,

$$\bigcup_{i\in[n]} pbd(A_i) = \bigcup_{x\in\mathsf{ndf}(f)} \{x\} = \mathsf{ndf}(f)$$

by construction; f_i is differentiable for all $i \in [n]$; and $\{(A_i, f_i)\}_{i\in[n]}$ defines g since g is an extended derivative of f. Hence, $\{(A_i, f_i)\}_{i\in[n]}$ is a piecewise-differentiable representation of f that defines

g and satisfies the equation in the statement.

Claim (ii). By a similar argument, there is a piecewise-C^1 representation $\{(\widetilde{A}_i, \widetilde{f}_i)\}_{i\in[\widetilde{n}]}$ of f that defines g and satisfies

$$\bigcup_{i\in[\widetilde{n}]} pbd(\widetilde{A}_i) = \mathsf{ncdf}(f).$$

Note that here we need $\mathsf{ncdf}(f)$ (instead of $\mathsf{ndf}(f)$) in the above equation, to obtain a piecewise-C^1 (instead of piecewise-differentiable) representation of f. We then construct $\{(A_i, f_i)\}_{i\in[n]}$ from $\{(\widetilde{A}_i, \widetilde{f}_i)\}_{i\in[\widetilde{n}]}$, by merging each singleton interval $\widetilde{A}_i$ (and the associated function $\widetilde{f}_i$) with one of the two adjacent intervals (and its associated function) such that $\{(A_i, f_i)\}_{i\in[n]}$ defines g. Such a construction always exists, because f is continuous, g is consistent, and $\widetilde{f}_i$ is C^1 for all $i \in [\widetilde{n}]$. Then,

$$\bigcup_{i\in[n]} pbd(A_i) = \mathsf{ncdf}(f), \qquad int(A_i) \neq \emptyset \qquad \text{for all } i \in [n]$$

by construction; and f_i is C^1 for all $i \in [\widetilde{n}]$ since f is continuous. Hence, $\{(A_i, f_i)\}_{i\in[n]}$ is a piecewise-C^1 representation of f that defines g and satisfies the equation given in the statement. □

B.1.2 Neural Networks

Definition B.5. For each $(l, i) \in \mathsf{Idx}$, let

$$\{(\mathcal{I}^k_{l,i}, \sigma^k_{l,i})\}_{k\in[K_{l,i}]}$$

be a piecewise-differentiable representation of $\sigma_{l,i} : \mathbb{R} \to \mathbb{R}$ that defines $D^{\mathtt{AD}}\sigma_{l,i}$ (an extended derivative of $\sigma_{l,i}$ defined in §5.2.3), where $K_{l,i} \in \mathbb{N}$, $\mathcal{I}^k_{l,i} \subseteq \mathbb{R}$, and $\sigma^k_{l,i} : \mathbb{R} \to \mathbb{R}$. We assume that the representation satisfies the following:

$$\bigcup_{k\in[K_{l,i}]} bd(\mathcal{I}^k_{l,i}) = \bigcup_{k\in[K_{l,i}]} pbd(\mathcal{I}^k_{l,i}) = \mathsf{ndf}(\sigma_{l,i}).$$

Note that such a representation always exists by Theorem B.4.

Definition B.6. Define Γ, the set of indices denoting which piece of each activation function is used, as

$$\Gamma \triangleq \{\gamma : \mathsf{Idx} \to \mathbb{N} \mid \gamma(l, i) \in [K_{l,i}] \text{ for all } (l, i) \in \mathsf{Idx}\}.$$

Definition B.7. Let $\gamma \in \Gamma$ and $l \in [L]$. Define $\mathcal{R}^\gamma \subseteq \mathbb{R}^W$, $y^\gamma_l, z^\gamma_l : \mathbb{R}^W \to \mathbb{R}^{N_l}$, $\sigma^\gamma_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$ as:

$$\begin{aligned}
\mathcal{R}^\gamma &\triangleq \{w \in \mathbb{R}^W \mid y_{l,i}(w) \in \mathcal{I}^{\gamma(l,i)}_{l,i} \text{ for all } (l, i) \in \mathsf{Idx}\},\\
y^\gamma_l(w) &\triangleq \tau_l\big(z^\gamma_{l-1}(w), \pi_l(w)\big), \qquad z^\gamma_l(w) \triangleq \sigma^\gamma_l\big(y^\gamma_l(w)\big),\\
\sigma^\gamma_l(x) &\triangleq \big(\sigma^{\gamma(l,1)}_{l,1}(x_1), \ldots, \sigma^{\gamma(l,N_l)}_{l,N_l}(x_{N_l})\big),
\end{aligned}$$

where $\pi_l : \mathbb{R}^W \to \mathbb{R}^{W_l}$ denotes the projection function that extracts $w_l \in \mathbb{R}^{W_l}$ from $(w_1, \ldots, w_L) \in \mathbb{R}^W$, and $z_0^\gamma : \mathbb{R}^W \to \mathbb{R}^{N_0}$ is defined as $z_0^\gamma \triangleq z_0$.

Lemma B.8. *$\{\mathcal{R}^\gamma\}_{\gamma\in\Gamma}$ is a partition of $\mathbb{R}^W$.*

Proof. This follows immediately from that $\{\mathcal{I}_{l,i}^k\}_{k\in[K_{l,i}]}$ is a partition of $\mathbb{R}$ for all $(l,i) \in \mathsf{Idx}$ (since $\{(\mathcal{I}_{l,i}^k, \sigma_{l,i}^k)\}_{k\in[K_{l,i}]}$ is a representation of $\sigma_{l,i}$). □

Lemma B.9. *For all $l \in [L]$ and $\gamma \in \Gamma$, y_l and z_l are continuous, and y_l^γ and z_l^γ are differentiable.*

Proof. The continuity of y_l and z_l follows directly from that $\tau_{l'}$, $\pi_{l'}$, and $\sigma_{l',i'}$ are continuous for all $(l',i') \in \mathsf{Idx}$. Similarly, the differentiability of y_l^γ and z_l^γ follows directly from that $\tau_{l'}$, $\pi_{l'}$, and $\sigma_{l',i'}^{k'}$ are differentiable for all $(l',i') \in \mathsf{Idx}$ and $k' \in [K_{l',i'}]$. □

Lemma B.10. *Let $\gamma \in \Gamma$. Then,*

$$\mathcal{R}^\gamma = \{w \in \mathbb{R}^W \mid y_{l,i}^\gamma(w) \in \mathcal{I}_{l,i}^{\gamma(l,i)} \text{ for all } (l,i) \in \mathsf{Idx}\}.$$

Note that the RHS uses $y_{l,i}^\gamma$ instead of $y_{l,i}$.

Proof. Let $\gamma \in \Gamma$. Define $\mathcal{R}_{\le l}^\gamma, \mathcal{S}_{\le l}^\gamma \subseteq \mathbb{R}^W$ for $l \in [L]$ as

$$\mathcal{R}_{\le l}^\gamma \triangleq \{w \in \mathbb{R}^W \mid y_{l',i}(w) \in \mathcal{I}_{l',i}^{\gamma(l',i)} \text{ for all } (l',i) \in \mathsf{Idx} \text{ with } l' \le l\},$$
$$\mathcal{S}_{\le l}^\gamma \triangleq \{w \in \mathbb{R}^W \mid y_{l',i}^\gamma(w) \in \mathcal{I}_{l',i}^{\gamma(l',i)} \text{ for all } (l',i) \in \mathsf{Idx} \text{ with } l' \le l\}.$$

It suffices to show the following claim which generalizes this lemma: all $l \in [L]$,

$$y_l(w) = y_l^\gamma(w) \text{ for all } w \in \mathcal{R}_{\le l-1}^\gamma, \qquad \mathcal{R}_{\le l}^\gamma = \mathcal{S}_{\le l}^\gamma.$$

We prove this claim by induction on l.

Case $l = 1$. Since $z_0 = z_0^\gamma$, we have the first claimed equation:

$$y_1(w) = \tau_1(z_0(w), w_1) = \tau_1(z_0^\gamma(w), w_1) = y_1^\gamma(w)$$

for all $w \in \mathbb{R}^W$. From this, we have the second claimed equation:

$$\mathcal{R}_{\le 1}^\gamma = \bigcap_{i\in[N_1]} \{w \in \mathbb{R}^W \mid y_{1,i}(w) \in \mathcal{I}_{1,i}^{\gamma(1,i)}\} = \bigcap_{i\in[N_1]} \{w \in \mathbb{R}^W \mid y_{1,i}^\gamma(w) \in \mathcal{I}_{1,i}^{\gamma(1,i)}\} = \mathcal{S}_{\le 1}^\gamma.$$

Case $l > 1$. We obtain the first claimed equation as follows: for all $w \in \mathcal{R}_{\le l-1}^\gamma$,

$$y_l^\gamma(w) = \tau_l\big(\sigma_{l-1}^\gamma(y_{l-1}^\gamma(w)), \pi_l(w)\big)$$

$$= \tau_l\big(\sigma_{l-1}^{\gamma}(y_{l-1}(w)), \pi_l(w)\big)$$
$$= \tau_l\big(\sigma_{l-1}(y_{l-1}(w)), \pi_l(w)\big) = y_l(w).$$

Here the second line uses $y_{l-1}^{\gamma}(w) = y_{l-1}(w)$, which holds by induction hypothesis on $l-1$ with $w \in \mathcal{R}_{\leq l-1}^{\gamma} \subseteq \mathcal{R}_{\leq l-2}^{\gamma}$. And the third line uses $\sigma_{l-1,i}^{\gamma(l-1,i)}(y_{l-1,i}(w)) = \sigma_{l-1,i}(y_{l-1,i}(w))$ for all $i \in [N_{l-1}]$, which holds because $y_{l-1,i}(w) \in \mathcal{I}_{l-1,i}^{\gamma(l-1,i)}$ (by $w \in \mathcal{R}_{\leq l-1}^{\gamma}$) and $\{(\mathcal{I}_{l-1,i}^{k}, \sigma_{l-1,i}^{k})\}_{k\in[K_{l-1,i}]}$ is a representation of $\sigma_{l-1,i}$. Using this result, we obtain the second claimed equation as follows:

$$\begin{aligned}
\mathcal{R}_{\leq l}^{\gamma} &= \mathcal{R}_{\leq l-1}^{\gamma} \cap \bigcap_{i\in[N_l]} \{w \in \mathcal{R}_{\leq l-1}^{\gamma} \mid y_{l,i}(w) \in \mathcal{I}_{l,i}^{\gamma(l,i)}\} \\
&= \mathcal{R}_{\leq l-1}^{\gamma} \cap \bigcap_{i\in[N_l]} \{w \in \mathcal{R}_{\leq l-1}^{\gamma} \mid y_{l,i}^{\gamma}(w) \in \mathcal{I}_{l,i}^{\gamma(l,i)}\} \\
&= \mathcal{S}_{\leq l-1}^{\gamma} \cap \bigcap_{i\in[N_l]} \{w \in \mathcal{S}_{\leq l-1}^{\gamma} \mid y_{l,i}^{\gamma}(w) \in \mathcal{I}_{l,i}^{\gamma(l,i)}\} = \mathcal{S}_{\leq l}^{\gamma},
\end{aligned}$$

where the second line uses $y_{l,i}^{\gamma}(w) = y_{l,i}(w)$ for all $w \in \mathcal{R}_{\leq l-1}^{\gamma}$, which we already proved, and the third line uses $\mathcal{R}_{\leq l-1}^{\gamma} = \mathcal{S}_{\leq l-1}^{\gamma}$, which holds by induction hypothesis on $l-1$. □

Lemma B.11. *Let $\gamma \in \Gamma$. Then, for all $l \in [L]$ and $w \in \mathcal{R}^{\gamma}$,*

$$y_l^{\gamma}(w) = y_l(w), \qquad z_l^{\gamma}(w) = z_l(w).$$

Proof. Let $\gamma \in \Gamma$. The claim shown in the proof of Lemma B.10 implies the first part of the conclusion (since $\mathcal{R}_{\leq l-1}^{\gamma} \supseteq \mathcal{R}^{\gamma}$): for all $l \in [L]$ and $w \in \mathcal{R}^{\gamma}$, $y_l^{\gamma}(w) = y_l(w)$. From this, we obtain the second part of the conclusion: for all $l \in [L]$ and $w \in \mathcal{R}^{\gamma}$,

$$z_l^{\gamma}(w) = \sigma_l^{\gamma}(y_l^{\gamma}(w)) = \sigma_l^{\gamma}(y_l(w)) = \sigma_l(y_l(w)) = z_l(w),$$

where the second equality follows from the first part of the conclusion, and the third equality from $\sigma_{l,i}^{\gamma(l,i)}(y_{l,i}(w)) = \sigma_{l,i}(y_{l,i}(w))$ which holds because $y_{l,i}(w) \in \mathcal{I}_{l,i}^{\gamma(l,i)}$ (by $w \in \mathcal{R}^{\gamma}$) and $\{(\mathcal{I}_{l,i}^{k}, \sigma_{l,i}^{k})\}_{k\in[K_{l,i}]}$ is a representation of $\sigma_{l,i}$. □

B.1.3 Automatic Differentiation

As discussed in §5.1, AD operates not on mathematical functions, but on programs that represent those functions. To this end, we define a program $\mathtt{P}$ that represents a function from $\mathbb{R}^W$ to $\mathbb{R}$ as follows:

$$\mathtt{P} ::= r \mid \mathtt{w}_{l,j} \mid f(\mathtt{P}_1, \ldots, \mathtt{P}_n)$$

where $r \in \mathbb{R}$, $l \in [L]$, $j \in [W_l]$, $f \in \{\tau_{l,i}, \sigma_{l,i} \mid (l,i) \in \mathsf{Idx}\}$, and $n \in \mathbb{N}$. This definition says that a program $\mathtt{P}$ can be either a real-valued constant r, a real-valued parameter $\mathtt{w}_{l,j}$, or the application of a function $f : \mathbb{R}^n \to \mathbb{R}$ to subprograms $\mathtt{P}_1, \ldots, \mathtt{P}_n$. In Chapter 5, we focus on particular programs $\mathtt{P}_{y_{l,i}}$ and $\mathtt{P}_{z_{l,i}}$ that represent the functions $y_{l,i}(\,\cdot\,; c), z_{l,i}(\,\cdot\,; c) : \mathbb{R}^W \to \mathbb{R}$ and are defined in a canonical way as follows:

$$\begin{aligned}
\mathtt{P}_{y_{l,i}} &\triangleq \tau_{l,i}(\mathtt{P}_{z_{l-1,1}}, \ldots, \mathtt{P}_{z_{l-1,N_{l-1}}}, \mathtt{w}_{l,1}, \ldots, \mathtt{w}_{l,W_l}), \\
\mathtt{P}_{z_{l,i}} &\triangleq \sigma_{l,i}(\mathtt{P}_{y_{l,i}}),
\end{aligned}$$

where $\mathtt{P}_{z_{0,i'}} \triangleq c_{i'}$ for $i' \in [N_0]$ represents the constant function $z_{0,i'}(\,\cdot\,; c) : \mathbb{R}^W \to \mathbb{R}$.

Given a program $\mathtt{P}$, we define $[\![\mathtt{P}]\!] : \mathbb{R}^W \to \mathbb{R}$ as the function represented by $\mathtt{P}$, and $[\![\mathtt{P}]\!]^{\mathtt{AD}} : \mathbb{R}^W \to \mathbb{R}^{1\times W}$ as the function that AD essentially computes when applied to $\mathtt{P}$. These functions are defined inductively as follows [1, 10, 92]:

$$\begin{aligned}
[\![r]\!](w) &\triangleq r, \\
[\![\mathtt{w}_{l,j}]\!](w) &\triangleq w_{l,j}, \\
[\![f(\mathtt{P}_1, \ldots, \mathtt{P}_n)]\!](w) &\triangleq f\big([\![\mathtt{P}_1]\!](w), \ldots, [\![\mathtt{P}_n]\!](w)\big), \\
[\![r]\!]^{\mathtt{AD}}(w) &\triangleq , \\
[\![\mathtt{w}_{l,j}]\!]^{\mathtt{AD}}(w) &\triangleq \mathbb{1}_{l,j}, \\
[\![f(\mathtt{P}_1, \ldots, \mathtt{P}_n)]\!]^{\mathtt{AD}}(w) &\triangleq D^{\mathtt{AD}} f\big([\![\mathtt{P}_1]\!](w), \ldots, [\![\mathtt{P}_n]\!](w)\big) \cdot \big[[\![\mathtt{P}_1]\!]^{\mathtt{AD}}(w) \,/\, \cdots \,/\, [\![\mathtt{P}_n]\!]^{\mathtt{AD}}(w)\big].
\end{aligned}$$

Here $w_{l,j} \in \mathbb{R}$ is defined as $(w_{1,1}, w_{1,2} \ldots, w_{L,W_L}) \triangleq w$, , $\mathbb{1}_{l,j} \in \mathbb{R}^{1\times W}$ denote the zero matrix and the matrix whose entries are all zeros except for a single one at the $(W_1 + \cdots + W_{l-1} + j)$-th entry, $D^{\mathtt{AD}} f : \mathbb{R}^n \to \mathbb{R}^{1\times n}$ denotes a "derivative" of f used by AD, and $[M_1 \,/\, \cdots \,/\, M_n]$ denotes the matrix that stacks up matrices $M_1, \ldots, M_n$ vertically. Note that $[\![f(\mathtt{P}_1, \ldots, \mathtt{P}_n)]\!]^{\mathtt{AD}}$ captures the essence of AD: it computes derivatives based on the chain rule for differentiation.

Using the above definitions, we define $D^{\mathtt{AD}} z_L : \mathbb{R}^W \to \mathbb{R}^{N_L \times W}$ as what AD essentially computes when applied to a program that canonically represents a neural network $z_L : \mathbb{R}^W \to \mathbb{R}^{N_L}$:

$$D^{\mathtt{AD}} z_L(w) \triangleq \big[[\![\mathtt{P}_{z_{L,1}}]\!]^{\mathtt{AD}}(w) \,/\, \cdots \,/\, [\![\mathtt{P}_{z_{L,N_L}}]\!]^{\mathtt{AD}}(w)\big].$$

Note that $D^{\mathtt{AD}} z_L$ depends on the "derivative" of (pre-)activation functions (i.e., $D^{\mathtt{AD}}\sigma_{l,i}$ and $D^{\mathtt{AD}}\tau_{l,i}$) used by AD.

Lemma B.12. *For any $\gamma \in \Gamma$ and $w \in \mathcal{R}^\gamma$,*

$$D^{\mathtt{AD}} z_L(w) = D z_L^\gamma(w).$$

Proof. Let $\gamma \in \Gamma$. We prove the following claim: for all $l \in [L] \cup \{0\}$, $i \in [N_l]$, and $w \in \mathcal{R}^\gamma$,

$$Dz^\gamma_{l,i}(w) = [\![\mathtt{P}_{z_{l,i}}]\!]^{\mathtt{AD}}(w).$$

Note that this claim implies the conclusion since

$$Dz^\gamma_L(w) = [Dz^\gamma_{L,1}(w) \,/\, \cdots \,/\, Dz^\gamma_{L,N_L}(w)] = [[\![\mathtt{P}_{z_{L,1}}]\!]^{\mathtt{AD}}(w) \,/\, \cdots \,/\, [\![\mathtt{P}_{z_{L,N_L}}]\!]^{\mathtt{AD}}(w)] = D^{\mathtt{AD}} z_L(w).$$

We prove the claim by induction on l.

Case $l = 0$. Let $i \in [N_l]$ and $w \in \mathcal{R}^\gamma$. Since $\mathtt{P}_{z_{0,i}}$ is a constant program, $Dz^\gamma_{0,i}(w) = \ = [\![\mathtt{P}_{z_{0,i}}]\!]^{\mathtt{AD}}(w)$ as desired.

Case $l > 0$. Let $i \in [N_l]$ and $w \in \mathcal{R}^\gamma$. Observe that

$$\begin{aligned}
[\![\mathtt{P}_{y_{l,i}}]\!]^{\mathtt{AD}}(w) &= [\![\tau_{l,i}(\mathtt{P}_{z_{l-1,1}}, \ldots, \mathtt{P}_{z_{l-1,N_{l-1}}}, \mathtt{w}_{l,1}, \ldots, \mathtt{w}_{l,W_l})]\!]^{\mathtt{AD}}(w) \\
&= D\tau_{l,i}\big([\![\mathtt{P}_{z_{l-1,1}}]\!](w), \ldots, [\![\mathtt{P}_{z_{l-1,N_{l-1}}}]\!](w), [\![\mathtt{w}_{l,1}]\!](w), \ldots, [\![\mathtt{w}_{l,N_l}]\!](w)\big) \\
&\qquad \cdot \big[[\![\mathtt{P}_{z_{l-1,1}}]\!]^{\mathtt{AD}}(w) \,/\, \cdots \,/\, [\![\mathtt{P}_{z_{l-1,N_{l-1}}}]\!]^{\mathtt{AD}}(w) \,/\, [\![\mathtt{w}_{l,1}]\!]^{\mathtt{AD}}(w) \,/\, \cdots \,/\, [\![\mathtt{w}_{l,W_l}]\!]^{\mathtt{AD}}(w)\big] \\
&= D\tau_{l,i}\big(z_{l-1}(w), \pi_l(w)\big) \cdot \big[Dz^\gamma_{l-1,1}(w) \,/\, \cdots \,/\, Dz^\gamma_{l-1,N_{l-1}}(w) \,/\, \mathbb{1}_{l,1} \,/\, \cdots \,/\, \mathbb{1}_{l,N_l}\big] \\
&= D\tau_{l,i}\big(z_{l-1}(w), \pi_l(w)\big) \cdot \big[Dz^\gamma_{l-1}(w) \,/\, D\pi_l(w)\big] \\
&= D\tau_{l,i}\big((z_{l-1}, \pi_l)(w)\big) \cdot D(z^\gamma_{l-1}, \pi_l)(w), \qquad \text{(B.2)}
\end{aligned}$$

where $(f, g) : \mathbb{R}^n \to \mathbb{R}^{m_1+m_2}$ is defined as $(f, g)(x) \triangleq (f(x), g(x))$ for $f : \mathbb{R}^n \to \mathbb{R}^{m_1}$ and $g : \mathbb{R}^n \to \mathbb{R}^{m_2}$. Here the third line uses $[\![\mathtt{P}_{z_{l-1,i'}}]\!](w) = z_{l-1,i'}(w)$ and $[\![\mathtt{P}_{z_{l-1,i'}}]\!]^{\mathtt{AD}}(w) = Dz^\gamma_{l-1,i'}(w)$ for all $i' \in [N_{l-1}]$, where the latter holds by induction hypothesis on $l - 1$.

Using the observation above, we obtain the claim:

$$\begin{aligned}
[\![\mathtt{P}_{z_{l,i}}]\!]^{\mathtt{AD}}(w) &= [\![\sigma_{l,i}(\mathtt{P}_{y_{l,i}})]\!]^{\mathtt{AD}}(w) \\
&= D^{\mathtt{AD}}\sigma_{l,i}\big([\![\mathtt{P}_{y_{l,i}}]\!](w)\big) \cdot [\![\mathtt{P}_{y_{l,i}}]\!]^{\mathtt{AD}}(w) \\
&= D^{\mathtt{AD}}\sigma_{l,i}\big(y_{l,i}(w)\big) \cdot D\tau_{l,i}\big((z_{l-1}, \pi_l)(w)\big) \cdot D(z^\gamma_{l-1}, \pi_l)(w) \\
&= D\sigma^\gamma_{l,i}\big(y_{l,i}(w)\big) \cdot D\tau_{l,i}\big((z_{l-1}, \pi_l)(w)\big) \cdot D(z^\gamma_{l-1}, \pi_l)(w) \\
&= D\sigma^\gamma_{l,i}\big((\tau_{l,i} \circ (z^\gamma_{l-1}, \pi_l))(w)\big) \cdot D\tau_{l,i}\big((z^\gamma_{l-1}, \pi_l)(w)\big) \cdot D(z^\gamma_{l-1}, \pi_l)(w) \\
&= D(\sigma^\gamma_{l,i} \circ \tau_{l,i} \circ (z^\gamma_{l-1}, \pi_l))(w) \\
&= Dz^\gamma_{l,i}(w).
\end{aligned}$$

Here the third line uses $[\![\mathtt{P}_{y_{l,i}}]\!](w) = y_{l,i}(w)$ and Eq. (B.2), and the fourth line uses $D^{\mathtt{AD}}\sigma_{l,i}(y_{l,i}(w)) = D\sigma^{\gamma(l,i)}_{l,i}(y_{l,i}(w))$, which holds because $y_{l,i}(w) \in \mathcal{I}^{\gamma(l,i)}_{l,i}$ (by $w \in \mathcal{R}^\gamma$) and $\{(\mathcal{I}^k_{l,i}, \sigma^k_{l,i})\}_{k \in [K_{l,i}]}$ defines

$D^{\mathtt{AD}}\sigma_{l,i}$. The fifth line uses $y_{l,i}(w) = y^{\gamma}_{l,i}(w)$ and $z_{l-1}(w) = z^{\gamma}_{l-1}(w)$ (by Lemma B.11 with $w \in \mathcal{R}^{\gamma}$), and the sixth line uses the chain rule, which is applicable to $(\sigma^{\gamma}_{l,i} \circ \tau_{l,i} \circ (z^{\gamma}_{l-1}, \pi_l))$ because $\sigma^{\gamma}_{l,i}$, $\tau_{l,i}$, z^{γ}_{l-1}, and π_l are differentiable (as z^{γ}_{l-1} is differentiable by Lemma B.9). □

B.2 Upper Bounds on $\left|\mathsf{ndf}_{\Omega}(z_L) \cup \mathsf{inc}_{\Omega}(z_L)\right|$

B.2.1 Lemmas (Basic)

Lemma B.13. *For any $A, B \subseteq \mathbb{R}^n$,*

$$pbd(A \cup B) \subseteq pbd(A) \cup pbd(B), \qquad pbd(A \cap B) \subseteq pbd(A) \cup pbd(B).$$

Proof. Let $A, B \subseteq \mathbb{R}^n$. Then, $int(A \cup B) \supseteq int(A) \cup int(B)$ and $int(A \cap B) = int(A) \cap int(B)$. Using these, we obtain:

$$\begin{aligned}
pbd(A \cup B) &= (A \cup B) \setminus int(A \cup B) \\
&= (A \setminus int(A \cup B)) \cup (B \setminus int(A \cup B)) \\
&\subseteq (A \setminus int(A)) \cup (B \setminus int(B)) \\
&= pbd(A) \cup pbd(B), \\
pbd(A \cap B) &= (A \cap B) \setminus int(A \cap B) \\
&= (A \cap B) \setminus (int(A) \cap int(B)) \\
&= ((A \cap B) \setminus int(A)) \cup ((A \cap B) \setminus int(B)) \\
&\subseteq (A \setminus int(A)) \cup (B \setminus int(B)) \\
&= pbd(A) \cup pbd(B).
\end{aligned}$$

□

Lemma B.14. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function defined as $f(x) = g(x_{-n}) + c \cdot x_n$ for any $g : \mathbb{R}^n \to \mathbb{R}$ and $c \in \mathbb{R} \setminus \{0\}$, where x_{-n} denotes $(x_1, \ldots, x_{n-1})$. Then,*

$$\left|\{x \in \mathbb{M}^n \mid f(x) = 0\}\right| \leq |\mathbb{M}|^{n-1}.$$

Proof. Using the definition of f and $c \neq 0$, we obtain the conclusion:

$$\begin{aligned}
\left|\{x \in \mathbb{M}^n \mid f(x) = 0\}\right| &= \left|\{(x_{-n}, x_n) \in \mathbb{M}^{n-1} \times \mathbb{M} \mid f(x_{-n}, x_n) = 0\}\right| \\
&= \textstyle\sum_{x_{-n} \in \mathbb{M}^{n-1}} \left|\{x_n \in \mathbb{M} \mid x_n = -g(x_{-n})/c\}\right| \\
&\leq \textstyle\sum_{x_{-n} \in \mathbb{M}^{n-1}} 1 = |\mathbb{M}|^{n-1}.
\end{aligned}$$

□

B.2.2 Lemmas (Technical: Part 1)

Definition B.15. For a neural network $z_L : \mathbb{R}^W \to \mathbb{R}^{N_L}$, define the incorrect set and the non-differentiable set of z_L *over* $\mathbb{R}^W$ (not over Ω) as:

$$\mathsf{inc}_{\mathbb{R}}(z_L) \triangleq \{w \in \mathbb{R}^W \mid Dz_L(w) \neq \bot,\ D^{\mathtt{AD}} z_L(w) \neq Dz_L(w)\},$$
$$\mathsf{ndf}_{\mathbb{R}}(z_L) \triangleq \{w \in \mathbb{R}^W \mid Dz_L(w) = \bot\}.$$

Lemma B.16. *We have*

$$\mathsf{ndf}_{\mathbb{R}}(z_L) \cup \mathsf{inc}_{\mathbb{R}}(z_L) \subseteq \bigcup_{\gamma \in \Gamma} pbd(\mathcal{R}^\gamma).$$

Proof. First, observe that for all $\gamma \in \Gamma$,

$$D^{\mathtt{AD}} z_L(w) = Dz_L^\gamma(w) = Dz_L(w) \qquad \text{for all } w \in int(\mathcal{R}^\gamma),$$

where the first equality is by Lemma B.12, and the second equality is obtained by applying the following fact to $(z_L^\gamma, z_L, int(\mathcal{R}^\gamma))$: for any $f, g : \mathbb{R}^n \to \mathbb{R}^m$ and open $U \subseteq \mathbb{R}^n$, if f is differentiable on U and $f = g$ on U, then g is differentiable on U and $Df = Dg$ on U. Note that the previous fact is applicable since $int(\mathcal{R}^\gamma)$ is open, z_L^γ is differentiable (by Lemma B.9), and $z_L^\gamma = z_L$ on $int(\mathcal{R}^\gamma)$ by Lemma B.11.

From the above equation, we have

$$\bigcup_{\gamma \in \Gamma} int(\mathcal{R}^\gamma) \subseteq \mathbb{R}^W \setminus \big(\mathsf{ndf}_{\mathbb{R}}(z_L) \cup \mathsf{inc}_{\mathbb{R}}(z_L)\big).$$

From this, we obtain the conclusion:

$$\begin{aligned}\mathsf{ndf}_{\mathbb{R}}(z_L) \cup \mathsf{inc}_{\mathbb{R}}(z_L) &\subseteq \mathbb{R}^W \setminus \bigcup_{\gamma \in \Gamma} int(\mathcal{R}^\gamma) \\ &= \Big(\bigcup_{\gamma \in \Gamma} \mathcal{R}^\gamma\Big) \setminus \Big(\bigcup_{\gamma \in \Gamma} int(\mathcal{R}^\gamma)\Big) = \bigcup_{\gamma \in \Gamma} \big(\mathcal{R}^\gamma \setminus int(\mathcal{R}^\gamma)\big) = \bigcup_{\gamma \in \Gamma} pbd(\mathcal{R}^\gamma),\end{aligned}$$

where the first equality is by Lemma B.8, and the last equality is by the definition of $pbd(-)$. □

Lemma B.17. *We have*

$$\bigcup_{\gamma \in \Gamma} pbd(\mathcal{R}^\gamma) \subseteq \bigcup_{(l,i) \in \mathsf{Idx}} \bigcup_{c \in \mathsf{ndf}(\sigma_{l,i})} pbd\big(\{w \in \mathbb{R}^W \mid y_{l,i}(w) = c\}\big).$$

Proof. First, we have

$$\begin{aligned}
\bigcup_{\gamma\in\Gamma} pbd(\mathcal{R}^\gamma) &= \bigcup_{\gamma\in\Gamma} pbd\Big(\bigcap_{(l,i)\in\mathsf{Idx}} \{w\in\mathbb{R}^W \mid y_{l,i}(w)\in\mathcal{I}_{l,i}^{\gamma(l,i)}\}\Big)\\
&\subseteq \bigcup_{\gamma\in\Gamma}\bigcup_{(l,i)\in\mathsf{Idx}} pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in\mathcal{I}_{l,i}^{\gamma(l,i)}\}\Big)\\
&= \bigcup_{(l,i)\in\mathsf{Idx}}\bigcup_{\gamma\in\Gamma} pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in\mathcal{I}_{l,i}^{\gamma(l,i)}\}\Big)\\
&= \bigcup_{(l,i)\in\mathsf{Idx}}\bigcup_{k\in[K_{l,i}]} pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in\mathcal{I}_{l,i}^{k}\}\Big),
\end{aligned} \tag{B.3}$$

where the first line uses the definition of $\mathcal{R}^\gamma$, the second line uses Lemma B.13, and the last line uses that $\{\gamma(l,i) \mid \gamma\in\Gamma\} = [K_{l,i}]$ for all (l,i). Note that in the last two lines, we change the way we count the proper boundary of all subregions: from per subregion to per activation neuron.

Next, for any $(l,i)\in\mathsf{Idx}$ and $k\in[K_{l,i}]$, we have

$$\begin{aligned}
&pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in\mathcal{I}_{l,i}^{k}\}\Big)\\
&= pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in pbd(\mathcal{I}_{l,i}^{k})\}\cup\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in int(\mathcal{I}_{l,i}^{k})\}\Big)\\
&\subseteq pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in pbd(\mathcal{I}_{l,i}^{k})\}\Big)\cup pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in int(\mathcal{I}_{l,i}^{k})\}\Big)\\
&= pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in pbd(\mathcal{I}_{l,i}^{k})\}\Big),
\end{aligned} \tag{B.4}$$

where the third line is by Lemma B.13 and the last line is by the following: $pbd(A)=\emptyset$ for any open $A\subseteq\mathbb{R}^n$; and $\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in int(\mathcal{I}_{l,i}^{k})\}$ is open, because $y_{l,i}$ is continuous (by Lemma B.9) and the inverse image of an open set by a continuous function is open.

Finally, combining the above results, we obtain the conclusion:

$$\begin{aligned}
\bigcup_{\gamma\in\Gamma} pbd(\mathcal{R}^\gamma) &\subseteq \bigcup_{(l,i)\in\mathsf{Idx}}\bigcup_{k\in[K_{l,i}]} pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)\in pbd(\mathcal{I}_{l,i}^{k})\}\Big)\\
&\subseteq \bigcup_{(l,i)\in\mathsf{Idx}}\bigcup_{c\in\mathsf{ndf}(\sigma_{l,i})} pbd\Big(\{w\in\mathbb{R}^W \mid y_{l,i}(w)=c\}\Big),
\end{aligned}$$

where the first line uses Eqs. (B.3) and (B.4), and the second line uses $\bigcup_{k\in[K_{l,i}]} pbd(\mathcal{I}_{l,i}^{k}) = \mathsf{ndf}(\sigma_{l,i})$ (by Definition B.5) □

B.2.3 Theorem 5.7 (Main Lemmas)

Lemma B.18. *We have*

$$\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in\mathsf{Idx}} \bigcup_{c\in\mathsf{ndf}(\sigma_{l,i})} \{w \in \Omega \mid y_{l,i}(w) = c\}.$$

Proof. We obtain the conclusion by chaining Lemma B.16, Lemma B.17, and the following: $pbd(A) \subseteq A$ for any $A \subseteq \mathbb{R}^W$, and $\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) = \big(\mathsf{ndf}_\mathbb{R}(z_L) \cup \mathsf{inc}_\mathbb{R}(z_L)\big) \cap \Omega$. □

Lemma B.19. *Let $(l,i) \in \mathsf{Idx}$ and $c \in \mathbb{R}$. Suppose that τ_l has bias parameters. Then, for $S = \{w \in \Omega \mid y_{l,i}(w) = c\}$,*

$$|S| \leq |\mathbb{M}|^{W-1}.$$

Proof. Suppose that τ_l has bias parameters and S is given as above. Then, by the definition of having bias parameters, $W_l \geq N_l$ and there is $\tau'_{l,i} : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l - N_l} \to \mathbb{R}$ for all $i \in [N_l]$ such that

$$\tau_{l,i}(x,(u,v)) = \tau'_{l,i}(x,u) + v_i \qquad \text{for all } (u,v) \in \mathbb{R}^{W_l-N_l} \times \mathbb{R}^{N_l}.$$

From this, we have

$$y_{l,i}(w) = \tau_{l,i}(z_{l-1}(w), w_l) = \tau'_{l,i}\big(z_{l-1}(w_{1,1},\ldots,w_{l-1,W_{l-1}},0,\ldots,0),(w_{l,1},\ldots,w_{l,W_l-N_l})\big) + w_{l,W_l-N_l+i},$$

where we also use that z_{l-1} depends only on $w_1,\ldots,w_{l-1}$. Note that the function $f : \mathbb{R}^W \to \mathbb{R}$ defined by $f(w) \triangleq y_{l,i}(w) - c$ satisfies the preconditions of Lemma B.14 (after reordering the input variables of f) due to the term w_{l,W_l-N_l+i}. Using this, we obtain the desired result:

$$|S| = |\{w \in \Omega \mid f(w) = 0\}| \leq |\mathbb{M}|^{W-1},$$

where the inequality is by Lemma B.14 applied to f. □

B.2.4 Theorem 5.7 (Main Proof)

Theorem B.20. *If z_L has bias parameters, then*

$$\frac{|\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L)|}{|\Omega|} \leq \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in\mathsf{Idx}} |\mathsf{ndf}(\sigma_{l,i})|.$$

Proof. Observe that

$$\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in \mathsf{Idx}} \bigcup_{c\in A_{l,i}} B_{l,i}(c), \qquad |B_{l,i}(c)| \le |\mathbb{M}|^{W-1}, \tag{B.5}$$

where $A_{l,i} \triangleq \mathsf{ndf}(\sigma_{l,i})$ and $B_{l,i}(c) \triangleq \{w \in \Omega \mid y_{l,i}(w) = c\}$. Here the first equation is by Lemma B.18, and the second equation is by Lemma B.19 (which is applicable since τ_l has bias parameters by assumption). Combining the above observations, we obtain the conclusion:

$$\frac{|\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L)|}{|\Omega|} \le \sum_{(l,i)\in \mathsf{Idx}} \sum_{c\in A_{l,i}} \frac{|B_{l,i}(c)|}{|\Omega|} \le \sum_{(l,i)\in \mathsf{Idx}} |\mathsf{ndf}(\sigma_{l,i})| \cdot \frac{|\mathbb{M}|^{W-1}}{|\mathbb{M}|^W},$$

where the two inequalities use Eq. (B.5). □

Remark B.21. Theorem 5.7 is a direct corollary of Theorem B.20 and Theorem 5.6 (which we prove in §B.3). □

B.2.5 Lemmas (Technical: Part 2)

Lemma B.22. *Let $l \in [L]$. Suppose that $\tau_l : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}^{N_l}$ is well-structured biaffine. Then, for every $i \in [N_l]$, there is a partial map $\phi_{l,i} : [W_l] \rightharpoonup [N_{l-1}]$ and associated matrix $M \in \mathbb{R}^{N_{l-1}\times W_l}$ and constant $d \in \mathbb{R}$ such that*

$$y_{l,i}(w) = d + \sum_{j\in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(w) \cdot M_{\phi_{l,i}(j),j} \cdot w_{l,j}$$

and $M_{\phi_{l,i}(j),j} \neq 0$ for all $j \in dom(\phi_{l,i})$.

Proof. Let $l \in [L]$, $\tau_l : \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l} \to \mathbb{R}^{N_l}$ be a well-structured biaffine function, and $i \in [N_l]$. Then, there is a matrix $M \in \mathbb{R}^{N_{l-1}\times W_l}$ and a constant $d \in \mathbb{R}$ such that $\tau_{l,i}(x,u) = x^\mathsf{T} M u + d$ for all (x,u) and each column of M has at most one non-zero entry. Define a partial map $\phi_{l,i} : [W_l] \rightharpoonup [N_{l-1}]$ as:

$$\phi_{l,i}(j) \triangleq \begin{cases} i' & \text{if } M_{i',j} \neq 0 \text{ for some } i' \in [N_{l-1}] \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here $\phi_{l,i}$ is well-defined because $M_{-,j}$ contains at most one non-zero entry for all $j \in [W_l]$. We claim that $\phi_{l,i}$, M, and d satisfy the conditions in this lemma. First, by the definition of $\phi_{l,i}$, $M_{\phi_{l,i}(j),j} \neq 0$ for all $j \in dom(\phi_{l,i})$. Also, we have the desired equation as follows:

$$\begin{aligned} y_{l,i}(w) &= \tau_{l,i}(z_{l-1}(w), w_l) \\ &= d + (z_{l-1}(w)^\mathsf{T} M) \cdot w_l \end{aligned}$$

$$
\begin{aligned}
&= d + (v_1, \ldots, v_{W_{l-1}})^{\mathsf{T}} \cdot w_l \\
&= d + \textstyle\sum_{j \in [W_{l-1}]} v_j \cdot w_{l,j} \\
&= d + \textstyle\sum_{j \in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(w) \cdot M_{\phi_{l,i}(j),j} \cdot w_{l,j},
\end{aligned}
$$

where $v_j \in \mathbb{R}$ is defined as $v_j \triangleq z_{l-1,\phi_{l,i}(j)}(w) \cdot M_{\phi_{l,i}(j),j}$ if $j \in dom(\phi_{l,i})$, and $v_j \triangleq 0$ otherwise. Here the second line uses the definition of M and d, and the third and last lines use the definition of v_j. This concludes the proof. □

Lemma B.23. *For every $(l,i) \in \mathsf{Idx}$ and $c \in \mathbb{R}$, let $A_{l,i} \subseteq \mathbb{R}$ be any set and $B_{l,i}(c) \subseteq \mathbb{R}^W$ be the set $\{w \in \mathbb{R}^W \mid y_{l,i}(w) = c\}$. Suppose that for every $l \in [L]$, one of the following holds:*

(a) τ_l has bias parameters, or

(b) τ_l is well-structured biaffine.

In the case of (b), let $\phi_{l,i}$ be the partial map described in Lemma B.22 for all $i \in [N_l]$. Then,

$$
\bigcup_{(l,i) \in \mathsf{Idx}} \bigcup_{c \in A_{l,i}} pbd(B_{l,i}(c)) \subseteq \bigcup_{(l,i) \in \mathsf{Idx}} \bigcup_{c' \in A'_{l,i}} B'_{l,i}(c'),
$$

where $A'_{l,i} \subseteq \mathbb{R}$ and $B'_{l,i}(c') \subseteq \mathbb{R}^W$ are defined as

$$
A'_{l,i} \triangleq \begin{cases} A_{l,i} & \text{if } \tau_{l+1} \text{ satisfies the condition (a) or } l = L \\ A_{l,i} \cup \mathsf{bdz}(\sigma_{l,i}) & \text{if } \tau_{l+1} \text{ satisfies the condition (b),} \end{cases}
$$

$$
B'_{l,i}(c') \triangleq \begin{cases} B_{l,i}(c') & \text{if } \tau_l \text{ satisfies the condition (a)} \\ B_{l,i}(c') \cap \bigcup_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) \neq 0\} & \text{if } \tau_l \text{ satisfies the condition (b).} \end{cases}
$$

Proof. We claim that the following holds: for all $l \in [L]$, $i \in [N_l]$, and $c \in A'_{l,i}$,

$$
pbd(B_{l,i}(c)) \subseteq \bigcup_{(l',i') \in \mathsf{Idx}} \bigcup_{c' \in A'_{l',i'}} B'_{l',i'}(c'). \tag{B.6}
$$

This claim implies the conclusion because $A_{l,i} \subseteq A'_{l,i}$ for all $(l,i) \in \mathsf{Idx}$ (by the definition of $A'_{l,i}$). We prove the claim by induction on l.

Case $l = 1$. Let $i \in [N_l]$ and $c \in A'_{l,i}$. We prove Eq. (B.6) by case analysis on τ_l.

Subcase 1: τ_l satisfies the condition (a). In this subcase, Eq. (B.6) holds since

$$
pbd(B_{l,i}(c)) \subseteq B_{l,i}(c) = B'_{l,i}(c), \qquad c \in A'_{l,i},
$$

where the equality uses the definition of $B'_{l,i}$.

Subcase 2: τ_l satisfies the condition (b). In this subcase, we have

$$\begin{aligned} pbd(B_{l,i}(c)) &= pbd\Big(\Big(B_{l,i}(c) \cap \bigcup_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) \neq 0\}\Big) \\ &\qquad \cup \Big(B_{l,i}(c) \cap \bigcap_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) = 0\}\Big)\Big) \\ &\subseteq pbd\Big(B_{l,i}(c) \cap \bigcup_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) \neq 0\}\Big) \\ &\qquad \cup pbd\Big(B_{l,i}(c) \cap \bigcap_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) = 0\}\Big), \end{aligned}$$

where the inclusion uses Lemma B.13. To prove Eq. (B.6), it suffices to show that the two terms in the last two lines are contained in the RHS of Eq. (B.6). The first term does so because

$$pbd\Big(B_{l,i}(c) \cap \bigcup_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) \neq 0\}\Big) = pbd(B'_{l,i}(c)) \subseteq B'_{l,i}(c), \qquad c \in A'_{l,i},$$

where the equality is by the definition of $B'_{l,i}$ and that τ_l does not have bias parameters. The second term is also contained in the RHS of Eq. (B.6) as follows. Let $S \triangleq \bigcap_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) = 0\}$, and $M \in \mathbb{R}^{N_{l-1} \times W_l}$ and $d \in \mathbb{R}$ be a matrix and a constant associated with $\phi_{l,i}$ that are described in Lemma B.22. Then,

$$B_{l,i}(c) \cap S = \begin{cases} S & \text{if } c = d \\ \emptyset & \text{if } c \neq d, \end{cases}$$

because $w \in S$ implies $y_{l,i}(w) = d + \sum_{j \in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(w) \cdot M_{\phi_{l,i}(j),j} \cdot w_{l,j} = d$ by Lemma B.22 (which is applicable since τ_l is well-structured biaffine by assumption). From this, we have

$$pbd\Big(B_{l,i}(c) \cap \bigcap_{j \in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) = 0\}\Big) = pbd(B_{l,i}(c) \cap S) \subseteq pbd(S) \cup pbd(\emptyset).$$

Hence, it suffices to show that $pbd(S)$ is contained in the RHS of Eq. (B.6) (since $pbd(\emptyset) = \emptyset$). Using $l = 1$, we obtain this:

$$pbd(S) \subseteq pbd(\mathbb{R}^W) \cup pbd(\emptyset) = \emptyset,$$

where the inclusion follows from $S \in \{\mathbb{R}^W, \emptyset\}$ which holds because $z_{l-1,\phi_{l,i}(j)}$ is a constant function for all $j \in [N_{l-1}]$ (by $l = 1$ and the assumption on z_0).

Case $l > 1$. Let $i \in [N_l]$ and $c \in A'_{l,i}$. We prove Eq. (B.6) in the exact same way as we did for the case $l = 1$. Note that the above proof for the previous case ($l = 1$) applies directly to the current

case ($l > 1$), except for the following subclaim: if τ_l does not have bias parameters, then $pbd(S)$ is contained in the RHS of Eq. (B.6). This subclaim holds also for $l > 1$, as follows:

$$\begin{aligned}
pbd(S) &= pbd\Big(\bigcap_{j\in dom(\phi_{l,i})} \{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) = 0\}\Big) \\
&\subseteq \bigcup_{j\in dom(\phi_{l,i})} pbd\Big(\{w \in \mathbb{R}^W \mid z_{l-1,\phi_{l,i}(j)}(w) = 0\}\Big) \\
&= \bigcup_{j\in dom(\phi_{l,i})} pbd\Big(\{w \in \mathbb{R}^W \mid y_{l-1,\phi_{l,i}(j)}(w) \in pbd\big(\sigma^{-1}_{l-1,\phi_{l,i}(j)}(0)\big)\} \\
&\qquad\qquad \cup \{w \in \mathbb{R}^W \mid y_{l-1,\phi_{l,i}(j)}(w) \in int\big(\sigma^{-1}_{l-1,\phi_{l,i}(j)}(0)\big)\}\Big) \\
&\subseteq \bigcup_{j\in dom(\phi_{l,i})} pbd\Big(\{w \in \mathbb{R}^W \mid y_{l-1,\phi_{l,i}(j)}(w) \in pbd\big(\sigma^{-1}_{l-1,\phi_{l,i}(j)}(0)\big)\}\Big) \\
&\qquad\qquad \cup pbd\Big(\{w \in \mathbb{R}^W \mid y_{l-1,\phi_{l,i}(j)}(w) \in int\big(\sigma^{-1}_{l-1,\phi_{l,i}(j)}(0)\big)\}\Big) \\
&= \bigcup_{j\in dom(\phi_{l,i})} pbd\Big(\{w \in \mathbb{R}^W \mid y_{l-1,\phi_{l,i}(j)}(w) \in pbd\big(\sigma^{-1}_{l-1,\phi_{l,i}(j)}(0)\big)\}\Big) \\
&= \bigcup_{j\in dom(\phi_{l,i})} \bigcup_{b\in \mathsf{bdz}(\sigma_{l-1,\phi_{l,i}(j)})} pbd(B_{l-1,\phi_{l,i}(j)}(b)), \\
pbd(B_{l-1,\phi_{l,i}(j)}(b)) &\subseteq \bigcup_{(l',i')\in \mathsf{Idx}} \bigcup_{c'\in A'_{l',i'}} B'_{l',i'}(c') \qquad \text{for all } j \in dom(\phi_{l,i}) \text{ and } b \in \mathsf{bdz}(\sigma_{l-1,\phi_{l,i}(j)}).
\end{aligned}$$

Here the first and second inclusions use Lemma B.13, and the second last equality uses that $y_{l-1,\phi_{l,i}(j)}$ is continuous (by Lemma B.9). The last equality uses $pbd(\sigma^{-1}_{l-1,\phi_{l,i}(j)}(0)) = \mathsf{bdz}(\sigma_{l-1,\phi_{l,i}(j)})$ (which holds since $\sigma_{l-1,\phi_{l,i}(j)}$ is continuous and the preimage of a closed set by a continuous map is closed), and the definition of $B_{l-1,\phi_{l,i}(j)}$. The last inclusion is by the induction hypothesis applied to $(l-1, j, b)$ for $j \in dom(\phi_{l,i})$ and $b \in \mathsf{bdz}(\sigma_{l-1,\phi_{l,i}(j)})$, together with $dom(\phi_{l,i}) \subseteq [N_{l-1}]$ and $\mathsf{bdz}(\sigma_{l-1,\phi_{l,i}(j)}) \subseteq A'_{l-1,\phi_{l,i}(j)}$ (which holds by the definition of $A'_{l-1,\phi_{l,i}(j)}$ with $l-1 \neq L$ and that τ_l does not have bias parameters). Hence, Eq. (B.6) holds for $l > 1$, and this concludes the proof. □

B.2.6 Theorem 5.12 (Main Lemmas)

Lemma B.24. *For every $l \in [L]$, suppose that τ_l satisfies either the condition (a) or (b) in Lemma B.23. Then,*

$$\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in \mathsf{Idx}} \bigcup_{c\in A_{l,i}} B_{l,i}(c),$$

where $A_{l,i} \subseteq \mathbb{R}$ *and* $B_{l,i}(c) \subseteq \Omega$ *are defined as*

$$A_{l,i} \triangleq \begin{cases} \mathsf{ndf}(\sigma_{l,i}) & \text{if } \tau_{l+1} \text{ satisfies the condition (a) or } l = L \\ \mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i}) & \text{if } \tau_{l+1} \text{ satisfies the condition (b),} \end{cases}$$

$$B_{l,i}(c) \triangleq \begin{cases} \{w \in \Omega \mid y_{l,i}(w) = c\} & \text{if } \tau_l \text{ satisfies the condition (a)} \\ \{w \in \Omega \mid y_{l,i}(w) = c \wedge \bigvee_{j \in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(w) \neq 0\} & \text{if } \tau_l \text{ satisfies the condition (b).} \end{cases}$$

Proof. We obtain the conclusion by chaining Lemma B.16, Lemma B.17, Lemma B.23 (which is applicable by assumption), and $\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) = \big(\mathsf{ndf}_{\mathbb{R}}(z_L) \cup \mathsf{inc}_{\mathbb{R}}(z_L)\big) \cap \Omega$. □

Lemma B.25. *Let* $(l,i) \in \mathsf{Idx}$ *and* $c \in \mathbb{R}$. *Suppose that* τ_l *is well-structured biaffine. Consider* $S = \{w \in \Omega \mid y_{l,i}(w) = c \wedge \bigvee_{j \in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(w) \neq 0\}$, *where* $\phi_{l,i}$ *denotes the partial map described in Lemma B.22. Then,*

$$|S| \leq |\mathbb{M}|^{W-1}.$$

Proof. Suppose that τ_l is well-structured biaffine, and S is given as above. We make three observations. First,

$$\begin{aligned} S &= \big\{(u,v) \in \mathbb{M}^{W'} \times \mathbb{M}^{W-W'} \;\big|\; \big(\exists j \in dom(\phi_{l,i}).\, z_{l-1,\phi_{l,i}(j)}(u,0,\ldots,0) \neq 0\big) \wedge y_{l,i}(u,v) = c\big\} \\ &= \bigcup_{u \in U} \bigcup_{v \in \mathbb{M}^{W-W'}} \{(u,v) \mid y_{l,i}(u,v) = c\}, \end{aligned} \tag{B.7}$$

where the first line uses $W' \triangleq W_1 + \cdots + W_{l-1}$ and that z_{l-1} depends only on $w_1, \ldots, w_{l-1}$, and the second line uses $U \triangleq \{u \in \mathbb{M}^{W'} \mid \exists j \in dom(\phi_{l,i}).\, z_{l-1,\phi_{l,i}(j)}(u,0,\ldots,0) \neq 0\}$. Second, by Lemma B.22 (which is applicable since τ_l is well-structured biaffine by assumption), there are $M \in \mathbb{R}^{N_{l-1} \times W_l}$ and $d \in \mathbb{R}$ such that $M_{\phi_{l,i}(j),j} \neq 0$ for all $j \in \phi_{l,i}$, and

$$\begin{aligned} y_{l,i}(u,v) &= d + \sum_{j \in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(u,v) \cdot M_{\phi_{l,i}(j),j} \cdot v_j \\ &= d + \sum_{j \in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(u,0,\ldots,0) \cdot M_{\phi_{l,i}(j),j} \cdot v_j \end{aligned} \tag{B.8}$$

for all $(u,v) \in \mathbb{R}^{W'} \times \mathbb{R}^{W-W'}$, where the second equality uses that z_{l-1} depends only on u. Third, for any $u \in U$, the function $f_u : \mathbb{R}^{W-W'} \to \mathbb{R}$ defined by $f_u(v) \triangleq y_{l,i}(u,v) - c$ satisfies the preconditions of Lemma B.14 (after reordering the input variables of f_u) due to the following: $z_{l-1,\phi_{l,i}(j)}(u,0,\ldots,0) \neq 0$ for some $j \in dom(\phi_{l,i})$ since $u \in U$; and the coefficient of v_j in $f_u(v)$ is $z_{l-1,\phi_{l,i}(j)}(u,0,\ldots,0) \cdot M_{\phi_{l,i}(j),j} \neq 0$ by Eq. (B.8) and $M_{\phi_{l,i}(j),j} \neq 0$.

By combining the above observations, we obtain the conclusion:

$$\begin{aligned}
|S| &= \Big| \bigcup_{u \in U} \bigcup_{v \in \mathbb{M}^{W-W'}} \{(u,v) \mid y_{l,i}(u,v) = c\} \Big| \\
&= \sum_{u \in U} \Big| \bigcup_{v \in \mathbb{M}^{W-W'}} \{(u,v) \mid y_{l,i}(u,v) = c\} \Big| \\
&= \sum_{u \in U} \big| \{ v \in \mathbb{M}^{W-W'} \mid f_u(v) = 0 \} \big| \\
&\leq |\mathbb{M}|^{W'} \cdot |\mathbb{M}|^{W-W'-1} = |\mathbb{M}|^{W-1},
\end{aligned}$$

where the first line uses Eq. (B.7), the third line uses the definition of f_u, and the last line uses Lemma B.14 applied to f_u. □

B.2.7 Theorem 5.12 (Main Proof)

Theorem 5.12. *If τ_l either has bias parameters or is well-structured biaffine for all $l \in [L]$, then*

$$\frac{|\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L)|}{|\Omega|} \leq \frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \Big| \mathsf{ndf}(\sigma_{l,i}) \cup \big(\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1} \big) \Big|,$$

where $S_l \subseteq \mathbb{R}$ is defined by

$$S_l \triangleq \begin{cases} \emptyset & \text{if } l > L \text{ or } \tau_l \text{ has bias parameters} \\ \mathbb{R} & \text{otherwise.} \end{cases}$$

Proof. Observe that

$$\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i) \in \mathsf{Idx}} \bigcup_{c \in A_{l,i}} B_{l,i}(c), \qquad |B_{l,i}(c)| \leq |\mathbb{M}|^{W-1}, \tag{B.9}$$

where $A_{l,i} \subseteq \mathbb{R}$ and $B_{l,i}(c) \subseteq \Omega$ are defined as in Lemma B.24. Here the first equation is by Lemma B.24 and the second equation is by Lemmas B.19 and B.25, where these lemmas are applicable by the definition of $B_{l,i}(c)$ and because τ_l either has bias parameters or is well-structured biaffine (both by assumption). Observe further that

$$A_{l,i} = \mathsf{ndf}(\sigma_{l,i}) \cup (\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1}) \tag{B.10}$$

by the definition of $A_{l,i}$ and S_l, where S_l is defined in the statement of this theorem. Combining

the above observations, we obtain the conclusion:

$$\frac{|\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L)|}{|\Omega|} \le \sum_{(l,i)\in\mathsf{Idx}} \sum_{c\in A_{l,i}} \frac{|B_{l,i}(c)|}{|\Omega|} \le \sum_{(l,i)\in\mathsf{Idx}} \left|\mathsf{ndf}(\sigma_{l,i}) \cap (\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1})\right| \cdot \frac{|\mathbb{M}|^{W-1}}{|\mathbb{M}|^W},$$

where the first inequality is by Eq. (B.9) and the second inequality is by Eqs. (B.9) and (B.10). □

B.3 Upper Bounds on $|\mathsf{inc}_\Omega(z_L)|$

In the rest of the appendix, we use the following notation. For a vector $v \in \mathbb{R}^n$, $v_{a:b}$ denotes the vector $(v_a, \ldots, v_b)$. For a matrix $M \in \mathbb{R}^{n\times m}$, $M_{a:b,\,c:d}$ denotes the matrix $(M_{i,j})_{a\le i\le b,\, c\le j\le d}$; $M_{a:b,\,c}$ denotes the vector $(M_{a,c}, \ldots, M_{b,c})$; and $M_{*,\,c:d}$ denotes $M_{1:n,\,c:d}$ (and similarly for $M_{a:b,\,*}$ and $M_{*,\,c}$).

B.3.1 Lemmas (Basic)

Lemma B.26. *Let $n \in \mathbb{N}$. For each $j \in [n]$, let $f_j : \mathbb{R} \to \mathbb{R}$ and $\mathcal{A}_j$ be a finite cover of $\mathbb{R}$ (i.e., $\bigcup_{A\in\mathcal{A}_j} A = \mathbb{R}$ and $|\mathcal{A}_j| < \infty$). Consider $x \in \mathbb{R}$. Then, there is $\{x_i\}_{i\in\mathbb{N}} \subseteq (x, \infty)$ such that $\lim_{i\to\infty} x_i = x$ and for all $j \in [n]$,*

$$\{f_j(x_i) \mid i \in \mathbb{N}\} \subseteq A \qquad \text{for some } A \in \mathcal{A}_j.$$

Further, there is $\{x'_i\}_{i\in\mathbb{N}} \subseteq (-\infty, x)$ that satisfies the same conditions stated above.

Proof. Consider f_j, $\mathcal{A}_j$, and x stated above ($j \in [n]$). Let $x_i \triangleq x + 1/i$ for $i \in \mathbb{N}$. Then,

$$\{x_i\}_{i\in\mathbb{N}} \subseteq (x, \infty), \qquad \lim_{i\to\infty} x_i = x. \tag{B.11}$$

For each $(i, j) \in \mathbb{N} \times [n]$, let $A_{i,j} \in \mathcal{A}_j$ be the set satisfying $f_j(x_i) \in A_{i,j}$, and $A_i \triangleq (A_{i,1}, \ldots, A_{i,n}) \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$, where $A_{i,j}$ always exists since $\mathcal{A}_j$ is a cover of $\mathbb{R}$. Observe that since $|\mathcal{A}_1 \times \cdots \times \mathcal{A}_n| < \infty$ (by $|\mathcal{A}_j| < \infty$ and $n < \infty$) and $|\mathbb{N}| = \infty$, there must exist $\{k_i\}_{i\in\mathbb{N}} \subseteq \mathbb{N}$ such that

$$k_1 < k_2 < \cdots, \qquad A_{k_1} = A_{k_2} = \cdots. \tag{B.12}$$

We claim that $\{x_{k_i}\}_{i\in\mathbb{N}}$ satisfies the desired conditions. First, by Eq. (B.11) and $\lim_{i\to\infty} k_i = \infty$ (due to Eq. (B.12)), $\{x_{k_i}\}_{i\in\mathbb{N}} \subseteq (x, \infty)$ and $\lim_{i\to\infty} x_{k_i} = x$. Second, by Eq. (B.12), $\{f_j(x_{k_i}) \mid i \in \mathbb{N}\} \subseteq A_{k_1,j}$ for all $j \in [n]$. Hence, the claim holds and this concludes the proof. □

Lemma B.27. *Let $f, g : \mathbb{R} \to \mathbb{R}$ and $x \in \mathbb{R}$. Suppose that f and g are differentiable at x, and there is $\{x_i\}_{i\in\mathbb{N}} \subseteq \mathbb{R} \setminus \{x\}$ such that $\lim_{i\to\infty} x_i = x$ and $f(x_i) = g(x_i)$ for all $i \in \mathbb{N}$. Then,*

$$Df(x) = Dg(x).$$

Proof. Consider $f, g : \mathbb{R} \to \mathbb{R}$, $x \in \mathbb{R}$, and $\{x_i\}_{i\in\mathbb{N}} \subseteq \mathbb{R} \setminus \{x\}$ stated above. Then,

$$f(x) = \lim_{i\to\infty} f(x_i) = \lim_{i\to\infty} g(x_i) = g(x),$$

where the first and third equalities are by that f and g are continuous at x (as they are differentiable at x) and $x_i \to x$, and the second equality by that $f(x_i) = g(x_i)$ for all $i \in \mathbb{N}$. Using this, we obtain

$$Df(x) = \lim_{i\to\infty} \frac{f(x_i) - f(x)}{x_i - x} = \lim_{i\to\infty} \frac{g(x_i) - g(x)}{x_i - x} = Dg(x),$$

where the first and third equalities are by that f and g are differentiable at x, $x_i \to x$, and $x_i \neq x$ for all $i \in \mathbb{N}$, and the second equality by that $f(x_i) = g(x_i)$ for all $i \in \mathbb{N}$. This completes the proof. □

B.3.2 Lemmas (Technical: Part 1)

Definition B.28. Let $\gamma \in \Gamma$. Define $\mathcal{R}^\gamma_{\mathsf{cl}} \subseteq \mathbb{R}^W$ as

$$\mathcal{R}^\gamma_{\mathsf{cl}} \triangleq \bigcap_{(l,i)\in \mathsf{Idx}} \{w \in \mathbb{R}^W \mid y_{l,i}(w) \in cl(\mathcal{I}^{\gamma(l,i)}_{l,i})\}.$$

Note that when defining $\mathcal{R}^\gamma$ in Definition B.7, we used $\mathcal{I}^{\gamma(l,i)}_{l,i}$ instead of $cl(\mathcal{I}^{\gamma(l,i)}_{l,i})$.

Definition B.29. For $\gamma \in \Gamma$ and $l \in [L]$, define

$$\begin{aligned}
\widetilde{\tau}_l &: \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l + W_{l+1} + \cdots + W_L} \to \mathbb{R}^{N_l} \times \mathbb{R}^{W_{l+1} + \cdots + W_L}, \\
\widetilde{\sigma}_l, \widetilde{\sigma}^\gamma_l &: \mathbb{R}^{N_l} \times \mathbb{R}^{W_{l+1} + \cdots + W_L} \to \mathbb{R}^{N_l} \times \mathbb{R}^{W_{l+1} + \cdots + W_L}, \\
\widetilde{z}_l, \widetilde{z}^\gamma_l &: \mathbb{R}^{N_{l-1}} \times \mathbb{R}^{W_l + W_{l+1} + \cdots + W_L} \to \mathbb{R}^{N_L}
\end{aligned}$$

as follows:

$$\begin{aligned}
&\widetilde{\tau}_l(x,u) \triangleq \big(\tau_l(x, u_1, \ldots, u_{W_l}), u_{W_l+1}, \ldots, u_{W_l + W_{l+1} + \cdots + W_L}\big), \\
&\widetilde{\sigma}_l(x,u) \triangleq (\sigma_l(x), u), && \widetilde{\sigma}^\gamma_l(x,u) \triangleq (\sigma^\gamma_l(x), u), \\
&\widetilde{z}_l(x,u) \triangleq (\widetilde{z}_{l+1} \circ \widetilde{\sigma}_l \circ \widetilde{\tau}_l)(x,u) && \widetilde{z}^\gamma_l(x,u) \triangleq (\widetilde{z}^\gamma_{l+1} \circ \widetilde{\sigma}^\gamma_l \circ \widetilde{\tau}_l)(x,u)
\end{aligned}$$

where $\widetilde{z}_{L+1}, \widetilde{z}^\gamma_{L+1} : \mathbb{R}^{N_L} \to \mathbb{R}^{N_L}$ are defined as the identity function.

Lemma B.30. *For all $l \in [L]$ and $\gamma \in \Gamma$, $\widetilde{z}_l$ is continuous and $\widetilde{z}^\gamma_l$ is differentiable.*

Proof. Since the proof is similar to that of Lemma B.9, we omit it. □

Lemma B.31. *Let $\gamma \in \Gamma$, $w = (w_1, \ldots, w_L) \in \mathcal{R}^\gamma_{\mathsf{cl}}$, $l \in [L]$, and $x = (z_{l-1}(w), w_l, \ldots, w_L)$. Then,*

$$\widetilde{\tau}_l(x) = \big(y_l(w), w_{l+1}, \ldots, w_L\big), \qquad (\widetilde{\sigma}^\gamma_l \circ \widetilde{\tau}_l)(x) = \big(z_l(w), w_{l+1}, \ldots, w_L\big).$$

Proof. By the definition of $\widetilde{\tau}_l$ and $\widetilde{\sigma}_l^\gamma$, we get the conclusion:

$$\begin{aligned}\widetilde{\tau}_l(x) &= \big(\tau_l(z_{l-1}(w), w_l), w_{l+1}, \ldots, w_L\big) = \big(y_l(w), w_{l+1}, \ldots, w_L\big),\\ (\widetilde{\sigma}_l^\gamma \circ \widetilde{\tau}_l)(x) &= \big(\sigma_l^\gamma(y_l(w)), w_{l+1}, \ldots, w_L\big) = \big(z_l(w), w_{l+1}, \ldots, w_L\big),\end{aligned}$$

where the last equality is by the observation that $\sigma_{l,i}^{\gamma(l,i)}(y_{l,i}(w)) = \sigma_{l,i}(y_{l,i}(w))$ for all $i \in [N_l]$. Here the observation holds because $\sigma_{l,i}^{\gamma(l,i)}$ and $\sigma_{l,i}$ coincide on $cl(\mathcal{I}_{l,i}^{\gamma(l,i)})$ (as they coincide on $\mathcal{I}_{l,i}^{\gamma(l,i)}$ and are both continuous) and $y_{l,i}(w) \in cl(\mathcal{I}_{l,i}^{\gamma(l,i)})$ (by $w \in \mathcal{R}_{\mathsf{cl}}^\gamma$). □

Lemma B.32. *Let $\gamma \in \Gamma$ and $l \in [L]$. Then, for all $w = (w_1, \ldots, w_L) \in \mathbb{R}^W$,*

$$\widetilde{z}_l\big(z_{l-1}(w), w_l, \ldots, w_L\big) = z_L(w), \qquad \widetilde{z}_l^\gamma\big(z_{l-1}^\gamma(w), w_l, \ldots, w_L\big) = z_L^\gamma(w).$$

Proof. Let $\gamma \in \Gamma$. The proof is by induction on $l \in [L]$ (starting from $l = L+1$).

Case $l = L+1$. Since $\widetilde{z}_{L+1}$ and $\widetilde{z}_{L+1}^\gamma$ are identity functions, the desired equations clearly hold.

Case $l < L+1$. We obtain the first desired equation as follows:

$$\begin{aligned}\widetilde{z}_l\big(z_{l-1}(w), w_l, \ldots, w_L\big) &= (\widetilde{z}_{l+1} \circ \widetilde{\sigma}_l \circ \widetilde{\tau}_l)\big(z_{l-1}(w), w_l, \ldots, w_L\big)\\ &= (\widetilde{z}_{l+1} \circ \widetilde{\sigma}_l)\big(\tau_l(z_{l-1}(w), w_l), w_{l+1}, \ldots, w_L\big)\\ &= (\widetilde{z}_{l+1} \circ \widetilde{\sigma}_l)\big(y_l(w), w_{l+1}, \ldots, w_L\big)\\ &= \widetilde{z}_{l+1}\big(\sigma_l(y_l(w)), w_{l+1}, \ldots, w_L\big)\\ &= \widetilde{z}_{l+1}\big(z_l(w), w_{l+1}, \ldots, w_L\big)\\ &= z_L(w),\end{aligned}$$

where all but last lines use the definition of $\widetilde{z}_l$, $\widetilde{\tau}_l$, $\widetilde{\sigma}_l$, y_l, and z_l, and the last line uses induction hypothesis on $l+1$. We can obtain the second desired equation similarly, by using induction hypothesis on $l+1$ and the definition of $\widetilde{z}_l^\gamma$, $\widetilde{\tau}_l$, $\widetilde{\sigma}_l^\gamma$, y_l^γ, and z_l^γ. □

Lemma B.33. *Let $\gamma \in \Gamma$ and $l \in [L]$. Then, for all $w = (w_1, \ldots, w_L) \in \mathcal{R}^\gamma$,*

$$\widetilde{z}_l\big(z_{l-1}(w), w_l, \ldots, w_L\big) = \widetilde{z}_l^\gamma\big(z_{l-1}(w), w_l, \ldots, w_L\big).$$

Proof. By Lemma B.32, we have the conclusion as follows:

$$\widetilde{z}_l(z_{l-1}(w), w_l, \ldots, w_L) = z_L(w) = z_L^\gamma(w) = \widetilde{z}_l^\gamma(z_{l-1}(w), w_l, \ldots, w_L),$$

where the second equality is by Lemma B.11 with $w \in \mathcal{R}^\gamma$. □

B.3.3 Lemmas (Technical: Part 2)

Definition B.34. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ and $i \in [n]$. Define

$$D_i f : \mathbb{R}^n \to \mathbb{R}^m \cup \{\bot\}$$

be the partial derivative of f with respect to its i-th argument, where $\bot$ denotes non-differentiability. Hence, for any $x \in \mathbb{R}^n$ and $i \in [n]$, $Df(x) \neq \bot$ implies $D_i f(x) = (Df(x))_{1:m,\,i}$.

Lemma B.35. *Let $l \in [L]$, $w = (w_1, \ldots, w_L) \in \mathbb{R}^W$, and $j \in [W]$ with $j > W_{<l}$, where $W_{<l} \triangleq W_1 + \cdots + W_{l-1}$. Suppose that $\widetilde{z}_l$ is differentiable with respect to its $(N_{l-1} + (j - W_{<l}))$-th argument at $(z_{l-1}(w), w_l, \ldots, w_L)$, i.e., $D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l(z_{l-1}(w), w_l, \ldots, w_L) \neq \bot$. Then, there are $\gamma \in \Gamma$ and $\{t_n\}_{n\in\mathbb{N}} \subseteq (v_j, \infty)$ satisfying the following conditions:*

- $w \in \mathcal{R}^\gamma_{\mathsf{cl}}$,
- $D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l\big(z_{l-1}(w), w_l, \ldots, w_L\big) = D_{N_{l-1}+(j-W_{<l})}\widetilde{z}^\gamma_l\big(z_{l-1}(w), w_l, \ldots, w_L\big)$,
- $\lim_{n\to\infty} t_n = v_j$, *and*
- $(v_1, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W) \in \mathcal{R}^\gamma \quad$ *for all $n \in \mathbb{N}$,*

where $(v_1, \ldots, v_W) \triangleq w$ denotes the scalar values of w (recall that $w_l \in \mathbb{R}^{W_l}$ is not scalar by definition). Further, there are $\gamma' \in \Gamma$ and $\{t'_n\}_{i\in\mathbb{N}} \subseteq (-\infty, v_j)$ that satisfy the same conditions stated above.

Proof. Consider $l \in [L]$, $w \in \mathbb{R}^W$, and $j \in [W]$ stated above. We show the existence of γ and $\{t_n\}_{n\in\mathbb{N}}$, and will omit the proof of the existence of γ' and $\{t'_n\}_{n\in\mathbb{N}}$ since the proof is almost identical.

First, we show that there is $\{t_n\}_{n\in\mathbb{N}} \subseteq (v_j, \infty)$ such that $\lim_{n\to\infty} t_n = v_j$ and

$$\big\{(v_1, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W) \;\big|\; n \in \mathbb{N}\big\} \subseteq \mathcal{R}^\gamma \qquad \text{for some } \gamma \in \Gamma. \tag{B.13}$$

By the definition of $\mathcal{R}^\gamma$, Eq. (B.13) is equivalent to the following: for all $(l, i) \in \mathsf{Idx}$,

$$\big\{f_{l,i}(t_n) \;\big|\; n \in \mathbb{N}\big\} \subseteq \mathcal{I}^k_{l,i} \qquad \text{for some } k \in [K_{l,i}],$$

where $f_{l,i} : \mathbb{R} \to \mathbb{R}$ is defined as $f_{l,i}(t) \triangleq y_{l,i}(v_1, \ldots, v_{j-1}, t, v_{j+1}, \ldots, v_W)$. Note that Lemma B.26 is applicable to $(f_{l,i}, \{\mathcal{I}^k_{l,i}\}_{k\in[K_{l,i}]}, v_j)$, since $\{\mathcal{I}^k_{l,i}\}_{k\in[K_{l,i}]}$ is a finite cover of $\mathbb{R}$ for all (l, i). Hence, by the lemma, there is $\{t_n\}_{n\in\mathbb{N}} \subseteq (v_j, \infty)$ such that $\lim_{n\in\infty} t_n = v_j$ and Eq. (B.13) holds with some $\gamma \in \Gamma$.

Next, we show that $w \in \mathcal{R}^\gamma_{\mathsf{cl}}$. By the definition of $\mathcal{R}^\gamma_{\mathsf{cl}}$, this is equivalent to $y_{l,i}(w) \in cl(\mathcal{I}^{\gamma(l,i)}_{l,i})$ for all $(l, i) \in \mathsf{Idx}$. To show this, let $(l, i) \in \mathsf{Idx}$. By Eq. (B.13) and the definition of $\mathcal{R}^\gamma$, we have

$$\big\{y_{l,i}(v_1, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W) \;\big|\; n \in \mathbb{N}\big\} \subseteq \mathcal{I}^{\gamma(l,i)}_{l,i}. \tag{B.14}$$

Using this, we obtain

$$y_{l,i}(w) = \lim_{n\to\infty} y_{l,i}(v_1, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W) \in cl(\mathcal{I}_{l,i}^{\gamma(l,i)}),$$

where the equality is from the continuity of $y_{l,i}$ (by Lemma B.9) and $\lim_{n\to\infty} t_n = v_j$ (by the above), and the inclusion is by Eq. (B.14). Hence, we have $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$ as desired.

Lastly, we show that $D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l(z_{l-1}(w), w_l, \ldots, w_L) = D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l^{\gamma}(z_{l-1}(w), w_l, \ldots, w_L)$. To do so, define $g, g^{\gamma} : \mathbb{R} \to \mathbb{R}^{N_L}$ as:

$$g(t) \triangleq \widetilde{z}_l\big(z_{l-1}(w), v_{W_{<l}+1}, \ldots, v_{j-1}, t, v_{j+1}, \ldots, v_W\big),$$
$$g^{\gamma}(t) \triangleq \widetilde{z}_l^{\gamma}\big(z_{l-1}(w), v_{W_{<l}+1}, \ldots, v_{j-1}, t, v_{j+1}, \ldots, v_W\big).$$

Using them, we obtain the desired equation as follows:

$$D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l\big(z_{l-1}(w), w_l, \ldots, w_L\big) = Dg(v_j) = Dg^{\gamma}(v_j) = D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l^{\gamma}\big(z_{l-1}(w), w_l, \ldots, w_L\big),$$

where the first and third equalities are by the definition of partial derivatives, and the second equality comes from Lemma B.27 applied to $(g, g^{\gamma}, v_j, \{t_n\}_{n\in\mathbb{N}})$. Here Lemma B.27 is applicable due to the following: g is differentiable at v_j (as $Dg(v_j) = D_{N_{l-1}+(j-W_{<l})}\widetilde{z}_l(z_{l-1}(w), w_l, \ldots, w_L) \neq \bot$ by assumption); g^{γ} is differentiable (as $\widetilde{z}_l^{\gamma}$ is differentiable by Lemma B.30); $\lim_{n\to\infty} t_n = v_j$ with $t_n \neq v_j$ (by the above); and $g(t_n) = g^{\gamma}(t_n)$ for all $n \in \mathbb{N}$ because

$$\begin{aligned}
g(t_n) &= \widetilde{z}_l\big(z_{l-1}(w), v_{W_{<l}+1}, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W\big) \\
&= \widetilde{z}_l\big(z_{l-1}(v_1, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W), v_{W_{<l}+1}, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W\big) \\
&= \widetilde{z}_l^{\gamma}\big(z_{l-1}(v_1, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W), v_{W_{<l}+1}, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W\big) \\
&= \widetilde{z}_l^{\gamma}\big(z_{l-1}(w), v_{W_{<l}+1}, \ldots, v_{j-1}, t_n, v_{j+1}, \ldots, v_W\big) = g^{\gamma}(t_n),
\end{aligned}$$

where the second and fourth lines use that z_{l-1} depends only on its first $W_{<l}$ arguments and $j > W_{<l}$, and the third line is by Lemma B.33 and Eq. (B.13). This completes the proof. □

Lemma B.36. *Let $w = (w_1, \ldots, w_L) \in \mathbb{R}^W$ and $(l, i) \in \mathsf{Idx}$. Suppose the following hold: z_L is differentiable at w; τ_l has bias parameters; $\sigma_{l,i}$ is not differentiable at $y_{l,i}(w)$; and for all $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_2}$, $D_i\widetilde{z}_{l+1}^{\gamma_1}(z_l(w), w_{l+1}, \ldots, w_L) = D_i\widetilde{z}_{l+1}^{\gamma_2}(z_l(w), w_{l+1}, \ldots, w_L)$. Then, for all $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$,*

$$D_i\widetilde{z}_{l+1}^{\gamma}\big(z_l(w), w_{l+1}, \ldots, w_L\big) = (0, \ldots, 0).$$

Proof. Consider $w \in \mathbb{R}^W$ and $(l,i) \in \mathsf{Idx}$ satisfying the conditions in the lemma. First, we show that

$$D_{N_{l-1}+(W_l-N_l+i)}\widetilde{z}_l^\gamma(z_{l-1}(w), w_l, \dots, w_L) = D_i\widetilde{z}_{l+1}^\gamma(z_l(w), w_{l+1}, \dots, w_L) \cdot D\sigma_{l,i}^{\gamma(l,i)}(y_{l,i}(w)),$$

for any $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^\gamma$. To this end, we derive two derivatives:

$$\big(D\widetilde{\tau}_l(x)\big)_{*,\, N_{l-1}+(W_l-N_l+i)} \quad \text{and} \quad \big(D\widetilde{\sigma}_l^\gamma(x')\big)_{*,\, i}.$$

Since τ_l has bias parameters (by assumption), and by the definitions of $\widetilde{\tau}_l$ and $\widetilde{\sigma}_l^\gamma$, we have the following: for all $\gamma \in \Gamma$ and $i' \in [N_l + W_{l+1} + \cdots + W_L]$, there is $\tau'_{l,i'} : \mathbb{R}^{N_{l-1}+(W_l-N_l)} \to \mathbb{R}$ such that

$$\widetilde{\tau}_{l,i'}(x) = \begin{cases} \tau'_{l,i'}(x_1, \dots, x_{N_{l-1}+(W_l-N_l)}) + x_{N_{l-1}+(W_l-N_l+i')} & \text{if } i' \le N_l \\ x_{N_{l-1}+(W_l-N_l+i')} & \text{if } i' > N_l, \end{cases}$$

$$\widetilde{\sigma}_{l,i'}^\gamma(x') = \begin{cases} \sigma_{l,i'}^{\gamma(l,i')}(x'_{i'}) & \text{if } i' \le N_l \\ x'_{i'} & \text{if } i' > N_l, \end{cases}$$

for all $x \in \mathbb{R}^{N_{l-1}+W_l+\cdots+W_L}$ and $x' \in \mathbb{R}^{N_l+W_{l+1}+\cdots+W_L}$. From this and $i \in [N_l]$, we obtain two derivatives:

$$\big(D\widetilde{\tau}_l(x)\big)_{*,\, N_{l-1}+(W_l-N_l+i)} = e_i, \qquad \big(D\widetilde{\sigma}_l^\gamma(x')\big)_{*,\, i} = e_i \cdot D\sigma_{l,i}^{\gamma(l,i)}(x'_i), \tag{B.15}$$

where $e_i \in \mathbb{R}^{N_l+W_{l+1}+\cdots+W_L}$ denotes the standard unit vector with 1 at the i-th coordinate, $D\sigma_{l,i}^{\gamma(l,i)}(x'_i)$ is considered as a scalar value, and both equalities are by $i \in [N_l]$. Using this, we obtain the following equation for $x \triangleq (z_{l-1}(w), w_l, \dots, w_L)$ and for any $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^\gamma$:

$$\begin{aligned} D_{N_{l-1}+(W_l-N_l+i)}\widetilde{z}_l^\gamma(x) &= \Big(D\widetilde{z}_l^\gamma(x)\Big)_{*,\, N_{l-1}+(W_l-N_l+i)} \\ &= \Big(D\widetilde{z}_{l+1}^\gamma((\widetilde{\sigma}_l^\gamma \circ \widetilde{\tau}_l)(x)) \cdot D\widetilde{\sigma}_l^\gamma(\widetilde{\tau}_l(x)) \cdot D\widetilde{\tau}_l(x)\Big)_{*,\, N_{l-1}+(W_l-N_l+i)} \\ &= D\widetilde{z}_{l+1}^\gamma((\widetilde{\sigma}_l^\gamma \circ \widetilde{\tau}_l)(x)) \cdot D\widetilde{\sigma}_l^\gamma(\widetilde{\tau}_l(x)) \cdot \Big(D\widetilde{\tau}_l(x)\Big)_{*,\, N_{l-1}+(W_l-N_l+i)} \\ &= D\widetilde{z}_{l+1}^\gamma((\widetilde{\sigma}_l^\gamma \circ \widetilde{\tau}_l)(x)) \cdot \Big(D\widetilde{\sigma}_l^\gamma(\widetilde{\tau}_l(x))\Big)_{*,\, i} \\ &= \Big(D\widetilde{z}_{l+1}^\gamma((\widetilde{\sigma}_l^\gamma \circ \widetilde{\tau}_l)(x))\Big)_{*,\, i} \cdot D\sigma_{l,i}^{\gamma(l,i)}(\widetilde{\tau}_{l,i}(x)) \\ &= D_i\widetilde{z}_{l+1}^\gamma(z_l(w), w_{l+1}, \dots, w_L) \cdot D\sigma_{l,i}^{\gamma(l,i)}(y_{l,i}(w)), \end{aligned} \tag{B.16}$$

where the first two lines use $\widetilde{z}_l^\gamma = \widetilde{z}_{l+1}^\gamma \circ \widetilde{\sigma}_l^\gamma \circ \widetilde{\tau}_l$ and that $\widetilde{z}_l^\gamma$, $\widetilde{z}_{l+1}^\gamma$, $\widetilde{\sigma}_l^\gamma$, and $\widetilde{\tau}_l$ are differentiable (by Lemma B.30), the fourth and fifth lines use Eq. (B.15), and the last line uses Lemma B.31 with $w \in \mathcal{R}_{\mathsf{cl}}^\gamma$.

Next, we derive a sufficient condition for the conclusion by using Eq. (B.16) and applying

Lemma B.35 to (l, w, j) with $j \triangleq W_{<l} + (W_l - N_l + i)$, where $W_{<l} \triangleq W_1 + \cdots + W_{l-1}$. Note that the lemma is applicable here due to the following: $W_{<l} < j \leq W$, because $W_l \geq N_l$ (as τ_l has bias parameters by assumption) and $1 \leq i \leq N_l$ (as $(l, i) \in \mathsf{Idx}$); and $\widetilde{z}_l$ is differentiable with respect to its $(N_{l-1} + (W_l - N_l + i))$-th argument at $(z_{l-1}(w), w_l \ldots, w_L)$, because

$$D_{N_{l-1}+(W_l-N_l+i)}\widetilde{z}_l(z_{l-1}(w), w_l, \ldots, w_L) = D_{W_{<l}+(W_l-N_l+i)}z_L(w) \neq \bot$$

where the equality follows from that $z_L(w') = \widetilde{z}_l(z_{l-1}(w'_1, \ldots, w'_{l-1}, 0, \ldots, 0), w'_l, \ldots, w'_L)$ for all $w' \in \mathbb{R}^W$ by Lemma B.32, and the inequality from that z_L is differentiable at w (by assumption). Let $(v_1, \ldots, v_W) \triangleq w$ be the scalar values of w. By applying Lemma B.35 to (l, w, j), it holds that there are $\gamma_+, \gamma_- \in \Gamma$, $\{t_n^+\}_{n\in\mathbb{N}} \subseteq (v_j, \infty)$, and $\{t_n^-\}_{n\in\mathbb{N}} \subseteq (-\infty, v_j)$ such that $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_+} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_-}$ and

$$D_{N_{l-1}+(W_l-N_l+i)}\widetilde{z}_l^{\gamma_+}(x) = D_{N_{l-1}+(W_l-N_l+i)}\widetilde{z}_l^{\gamma_-}(x),$$

$$\{(v_1, \ldots, v_{j-1}, t_n^+, v_{j+1}, \ldots, v_W)\}_{n\in\mathbb{N}} \subseteq \mathcal{R}^{\gamma_+}, \qquad \lim_{n\to\infty} t_n^+ = v_j, \tag{B.17}$$

$$\{(v_1, \ldots, v_{j-1}, t_n^-, v_{j+1}, \ldots, v_W)\}_{n\in\mathbb{N}} \subseteq \mathcal{R}^{\gamma_-}, \qquad \lim_{n\to\infty} t_n^- = v_j, \tag{B.18}$$

where $x \triangleq (z_{l-1}(w), w_l, \ldots, w_L)$. By the first line and Eq. (B.16) with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_+} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_-}$, we have

$$D_i\widetilde{z}_{l+1}^{\gamma_+}(x') \cdot D\sigma_{l,i}^{\gamma_+(l,i)}(y_{l,i}(w)) = D_i\widetilde{z}_{l+1}^{\gamma_-}(x') \cdot D\sigma_{l,i}^{\gamma_-(l,i)}(y_{l,i}(w)),$$

where $x' \triangleq (z_l(w), w_{l+1}, \ldots, w_L)$. From this, and since $D_i\widetilde{z}_{l+1}^{\gamma}(x')$ is the same for all $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$ (by assumption), we immediately obtain the conclusion (i.e., $D_i\widetilde{z}_{l+1}^{\gamma}(x') = (0, \ldots, 0)$ for all $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$) if the following holds:

$$D\sigma_{l,i}^{\gamma_+(l,i)}(y_{l,i}(w)) \neq D\sigma_{l,i}^{\gamma_-(l,i)}(y_{l,i}(w)). \tag{B.19}$$

Hence, to prove the conclusion, it suffices to show Eq. (B.19).

Finally, we prove Eq. (B.19) in two steps. We first show that there are $\delta^+, \delta^- > 0$ such that

$$\big(y_{l,i}(w), y_{l,i}(w) + \delta^+\big) \subseteq \mathcal{I}_{l,i}^{\gamma_+(l,i)}, \qquad \big(y_{l,i}(w) - \delta^-, y_{l,i}(w)\big) \subseteq \mathcal{I}_{l,i}^{\gamma_+(l,i)}. \tag{B.20}$$

Fix $j \triangleq W_{<l} + (W_l - N_l + i)$ and $(v_1, \ldots, v_W) \triangleq w$ as above. Observe that we have

$$y_{l,i}(v_1, \ldots, v_{j-1}, t_n^+, v_{j+1}, \ldots, v_W) \in \mathcal{I}_{l,i}^{\gamma_+(l,i)} \text{ for all } n \in \mathbb{N},$$

$$y_{l,i}(v_1, \ldots, v_{j-1}, t_n^+, v_{j+1}, \ldots, v_W) > y_{l,i}(v_1, \ldots, v_W) = y_{l,i}(w) \text{ for all } n \in \mathbb{N},$$

$$\lim_{n\to\infty} y_{l,i}(v_1, \ldots, v_{j-1}, t_n^+, v_{j+1}, \ldots, v_W) = y_{l,i}(v_1, \ldots, v_W) = y_{l,i}(w),$$

where the first line uses Eq. (B.17), the third line uses Eq. (B.17) and that $y_{l,i}$ is continuous (by

Lemma B.9), and the second line uses the following and that $t_n^+ > v_j$ for all $n \in \mathbb{N}$: for all $t \in \mathbb{R}$,

$$y_{l,i}(v_1, \dots, v_{j-1}, t, v_{j+1}, \dots, v_W) = \tau'_{l,i}(z_{l-1}(w), v_{W_{<l}+1}, \dots, v_{W_{<l}+(W_l - N_l)}) + t,$$

which holds since z_{l-1} depends only on its first $W_{<l}$ arguments, τ_l has bias parameters, and $j = W_{<l} + (W_l - N_l + i) > W_{<l}$. By these results, and since $\mathcal{I}_{l,i}^{\gamma_+(l,i)}$ is an interval, there is $\delta^+ > 0$ satisfying Eq. (B.20); similarly, there is $\delta^- > 0$ satisfying Eq. (B.20), due to Eq. (B.18) and $t_n^- < v_j$ for all n.

We next show that Eq. (B.19) indeed holds. By Eq. (B.20) and $\sigma_{l,i} = \sigma_{l,i}^k$ on $\mathcal{I}_{l,i}^k$ for all k, we have

$$\sigma_{l,i} = \sigma_{l,i}^{\gamma_+(l,i)} \text{ on } \big[y_{l,i}(w), y_{l,i}(w) + \delta^+\big), \qquad \sigma_{l,i} = \sigma_{l,i}^{\gamma_-(l,i)} \text{ on } \big(y_{l,i}(w) - \delta^-, y_{l,i}(w)\big],$$

where the inclusion of $y_{l,i}(w)$ is by that $\sigma_{l,i}$ and $\sigma_{l,i}^k$ are continuous for all k. From this, we have

$$\begin{aligned} D\sigma_{l,i}^{\gamma_+(l,i)}\big(y_{l,i}(w)\big) &= \lim_{h \to 0^+} \frac{1}{h}\Big(\sigma_{l,i}(y_{l,i}(w) + h) - \sigma_{l,i}(y_{l,i}(w))\Big), \\ D\sigma_{l,i}^{\gamma_-(l,i)}\big(y_{l,i}(w)\big) &= \lim_{h \to 0^-} \frac{1}{h}\Big(\sigma_{l,i}(y_{l,i}(w) + h) - \sigma_{l,i}(y_{l,i}(w))\Big). \end{aligned}$$

Suppose here that Eq. (B.19) does not hold, i.e., $D\sigma_{l,i}^{\gamma_+(l,i)}(y_{l,i}(w)) = D\sigma_{l,i}^{\gamma_-(l,i)}(y_{l,i}(w))$. Then,

$$\lim_{h \to 0} \frac{1}{h}\Big(\sigma_{l,i}(y_{l,i}(w) + h) - \sigma_{l,i}(y_{l,i}(w))\Big) = D\sigma_{l,i}^{\gamma_+(l,i)}\big(y_{l,i}(w)\big) \neq \bot,$$

where the inequality is by that $\sigma_{l,i}^{\gamma_+(l,i)}$ is differentiable. This implies $D\sigma_{l,i}\big(y_{l,i}(w)\big) \neq \bot$, which contradicts to that $\sigma_{l,i}$ is non-differentiable at $y_{l,i}(w)$ (by assumption). Hence, Eq. (B.19) should hold. □

B.3.4 Theorem 5.6 (Main Lemmas)

Lemma B.37. *Let $w \in \mathbb{R}^W$ and $j \in [W]$. Suppose that z_L is differentiable with respect to its j-th argument at w (i.e., $D_j z_L(w) \neq \bot$). Then, there is $\gamma \in \Gamma$ such that $w \in \mathcal{R}_{\mathsf{cl}}^\gamma$ and*

$$D_j z_L(w) = D_j z_L^\gamma(w).$$

Proof. Consider $w \in \mathbb{R}^W$ and $j \in [W]$ stated above. First, by Lemma B.32, and since $z_0 = z_0^\gamma$ is a constant function, we have $z_L(w') = \widetilde{z}_1(z_0(0, \dots, 0), w')$ and $z_L^\gamma(w') = \widetilde{z}_1^\gamma(z_0(0, \dots, 0), w')$ for all $w' \in \mathbb{R}^W$ and $\gamma \in \Gamma$. From this, we have

$$\begin{aligned} D_j z_L(w) &= D_{N_0+j}\widetilde{z}_1\big(z_0(0, \dots, 0), w\big) = D_{N_0+j}\widetilde{z}_1\big(z_0(w), w\big), \\ D_j z_L^\gamma(w) &= D_{N_0+j}\widetilde{z}_1^\gamma\big(z_0(0, \dots, 0), w\big) = D_{N_0+j}\widetilde{z}_1^\gamma\big(z_0(w), w\big) \quad \text{for all } \gamma \in \Gamma, \end{aligned}$$

where the second and fourth equalities follow from that z_0 is a constant function. Second, by

Lemma B.35 applied to $(l = 1, w, j)$, there is $\gamma \in \Gamma$ such that

$$w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}, \qquad D_{N_0+j}\widetilde{z}_1\big(z_0(w), w\big) = D_{N_0+j}\widetilde{z}_1^{\gamma}\big(z_0(w), w\big).$$

Here Lemma B.35 is applicable, because $D_{N_0+j}\widetilde{z}_1(z_0(w), w) = D_j z_L(w) \neq \bot$ (by the above and by assumption). From these results, there is $\gamma \in \Gamma$ such that $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$ and $D_j z_L(w) = D_j z_L^{\gamma}(w)$. □

Lemma B.38. *Let $w \in \mathbb{R}^W$. Suppose that the following hold:*

- *z_L is differentiable at w.*
- *For all $l \in [L]$, if τ_l does not have bias parameters, then $\sigma_{l,i}$ is differentiable at $y_{l,i}(w)$ for all $i \in [N_l]$.*

Then, for all $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_2}$,

$$Dz_L^{\gamma_1}(w) = Dz_L^{\gamma_2}(w).$$

Proof. Let $w \in \mathbb{R}^W$. Consider the following claim: for all $l \in [L+1]$ and $\gamma_1, \gamma_2 \in \Gamma$, if $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_2}$, then

$$D\widetilde{z}_l^{\gamma_1}\big(z_{l-1}(w), w_l, \ldots, w_L\big) = D\widetilde{z}_l^{\gamma_2}\big(z_{l-1}(w), w_l, \ldots, w_L\big).$$

Note that the claim implies the conclusion: for any $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_2}$,

$$Dz_L^{\gamma_1}(w) = \Big(D\widetilde{z}_1^{\gamma_1}(z_0(0, \ldots, 0), w)\Big)_{*,\, N_0+1:N_0+W} = \Big(D\widetilde{z}_1^{\gamma_2}(z_0(0, \ldots, 0), w)\Big)_{*,\, N_0+1:N_0+W} = Dz_L^{\gamma_2}(w),$$

where the first and third equalities follow from that $z_L^{\gamma}(w') = \widetilde{z}_1^{\gamma}(z_0(0, \ldots, 0), w')$ for all $\gamma \in \Gamma$ and $w' \in \mathbb{R}^W$ (by Lemma B.32 and since $z_0^{\gamma} = z_0$ is a constant function), and $\widetilde{z}_1^{\gamma}$ is differentiable for all $\gamma \in \Gamma$ (by Lemma B.30); and the second equality is by the claim for $l = 1$ and that z_0 is a constant function. We prove the claim by induction on l (starting from $L + 1$).

Case $l = L + 1$. The claim clearly holds, since $\widetilde{z}_{L+1}^{\gamma}$ is the identity function for all $\gamma \in \Gamma$.

Case $l < L + 1$. To show the claim, we first analyze the derivatives mentioned in the claim. Let $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$, and consider any $x \in \mathbb{R}^{N_{l-1}+W_l+\cdots+W_L}$ and $x' \in \mathbb{R}^{N_l+W_{l+1}+\cdots+W_L}$. Recall the definition of $\widetilde{z}_l^{\gamma}$ and $\widetilde{\sigma}_l^{\gamma}$: for all $i \in [N_l + W_{l+1} + \cdots + W_L]$,

$$\widetilde{z}_l^{\gamma}(x) = (\widetilde{z}_{l+1}^{\gamma} \circ \widetilde{\sigma}_l^{\gamma} \circ \widetilde{\tau}_l)(x), \qquad \widetilde{\sigma}_{l,i}^{\gamma}(x') = \begin{cases} \sigma_{l,i}^{\gamma(l,i)}(x'_i) & \text{if } i \leq N_l \\ x'_i & \text{if } i > N_l. \end{cases}$$

Since every function in the RHS of the above equation is differentiable (by Lemma B.30), the following hold for all $i \in [N_l + W_{l+1} + \cdots + W_L]$:

$$D\widetilde{z}_l^{\gamma}(x) = D\widetilde{z}_{l+1}^{\gamma}\big((\widetilde{\sigma}_l^{\gamma} \circ \widetilde{\tau}_l)(x)\big) \cdot D\widetilde{\sigma}_l^{\gamma}\big(\widetilde{\tau}_l(x)\big) \cdot D\widetilde{\tau}_l(x), \tag{B.21}$$

$$\big(D\widetilde{\sigma}_l^{\gamma}(x')\big)_{*,\,i} = \begin{cases} e_i \cdot D\sigma_{l,i}^{\gamma(l,i)}(x_i') & \text{if } i \le N_l \\ e_i & \text{if } i > N_l, \end{cases}$$

where the first line uses the chain rule, $e_i \in \mathbb{R}^{N_l + W_{l+1} + \cdots + W_L}$ denotes the standard unit vector with 1 at the i-th coordinate, and $D\sigma_{l,i}^{\gamma(l,i)}(x_i')$ is considered as a scalar value. By the second line, the following holds for all $i \in [N_l + W_{l+1} + \cdots + W_L]$:

$$\Big(D\widetilde{z}_{l+1}^{\gamma}((\widetilde{\sigma}_l^{\gamma} \circ \widetilde{\tau}_l)(x)) \cdot D\widetilde{\sigma}_l^{\gamma}(\widetilde{\tau}_l(x))\Big)_{*,\,i} = D_i\widetilde{z}_{l+1}^{\gamma}\big((\widetilde{\sigma}_l^{\gamma} \circ \widetilde{\tau}_l)(x)\big) \cdot \begin{cases} D\sigma_{l,i}^{\gamma(l,i)}\big(\widetilde{\tau}_{l,i}(x)\big) & \text{if } i \le N_l \\ 1 & \text{if } i > N_l. \end{cases} \tag{B.22}$$

We can further simplify the two term in the RHS when $x = (z_{l-1}(w), w_l, \ldots, w_L)$, as follows:

$$D\widetilde{z}_{l+1}^{\gamma}\big((\widetilde{\sigma}_l^{\gamma} \circ \widetilde{\tau}_l)(x)\big) = D\widetilde{z}_{l+1}^{\gamma}(x'), \quad D\sigma_{l,i}^{\gamma(l,i)}\big(\widetilde{\tau}_{l,i}(x)\big) = D\sigma_{l,i}^{\gamma(l,i)}\big(y_{l,i}(w)\big) \text{ for all } i \in [N_l], \tag{B.23}$$

where $x' \triangleq (z_l(w), w_{l+1}, \ldots, w_L)$ and both equalities are by Lemma B.31 with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$.

We now prove the claim. Let $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_2}$, and fix $x \triangleq (z_{l-1}(w), w_l, \ldots, w_L)$ and $x' \triangleq (z_l(w), w_{l+1}, \ldots, w_L)$. By induction hypothesis on $l+1$, we obtain

$$D\widetilde{z}_{l+1}^{\gamma_1}(x') = D\widetilde{z}_{l+1}^{\gamma_2}(x'). \tag{B.24}$$

Since we want to show $D\widetilde{z}_l^{\gamma_1}(x) = D\widetilde{z}_l^{\gamma_2}(x)$, it suffices to show the following due to Eqs. (B.21)–(B.24): for all $i \in [N_l]$,

$$D_i\widetilde{z}_{l+1}^{\gamma_1}(x') \cdot D\sigma_{l,i}^{\gamma_1(l,i)}\big(y_{l,i}(w)\big) = D_i\widetilde{z}_{l+1}^{\gamma_1}(x') \cdot D\sigma_{l,i}^{\gamma_2(l,i)}\big(y_{l,i}(w)\big). \tag{B.25}$$

Let $i \in [N_l]$. We prove Eq. (B.25) by case analysis on i.

Subcase 1: $\sigma_{l,i}$ is non-differentiable at $y_{l,i}(w)$. To show Eq. (B.25), it suffices to show that

$$D_i\widetilde{z}_{l+1}^{\gamma_1}(x') = (0, \ldots, 0).$$

We obtain this equation by applying Lemma B.36 to $(w, (l,i), \gamma_1)$. Note that the lemma is applicable here because: z_L is differentiable at w (by assumption); $\sigma_{l,i}$ is non-differentiable at $y_{l,i}(w)$ and so τ_l has bias parameters (by assumption); $D_i\widetilde{z}_{l+1}^{\gamma}(x')$ is independent of γ for all $\gamma \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$ (by induction hypothesis on $l+1$); and $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1}$.

Subcase 2: $\sigma_{l,i}$ is differentiable at $y_{l,i}(w)$. To show Eq. (B.25), it suffices to show that for all $j \in [2]$,

$$D\sigma_{l,i}^{\gamma_j(l,i)}\big(y_{l,i}(w)\big) = D\sigma_{l,i}\big(y_{l,i}(w)\big). \tag{B.26}$$

Let $j \in [2]$. If $y_{l,i}(w) \in \mathcal{I}_{l,i}^{\gamma_j(l,i)}$, then we obtain Eq. (B.26) as follows:

$$D\sigma_{l,i}^{\gamma_j(l,i)}\big(y_{l,i}(w)\big) = D^{\mathtt{AD}}\sigma_{l,i}\big(y_{l,i}(w)\big) = D\sigma_{l,i}\big(y_{l,i}(w)\big),$$

where the first equality holds because $y_{l,i}(w) \in \mathcal{I}_{l,i}^{\gamma_j(l,i)}$ and $\{(\sigma_{l,i}^k, \mathcal{I}_{l,i}^k)\}_{k\in[K_{l,i}]}$ defines $D^{\mathtt{AD}}\sigma_{l,i}$; and the second equality holds because $\sigma_{l,i}$ is differentiable at $y_{l,i}(w)$ and $D^{\mathtt{AD}}\sigma_{l,i}$ is an extended derivative of $\sigma_{l,i}$. If $y_{l,i}(w) \notin \mathcal{I}_{l,i}^{\gamma_j(l,i)}$, then we obtain Eq. (B.26) directly from Lemma B.27 applied to $(\sigma_{l,i}^{\gamma_j(l,i)}, \sigma_{l,i}, y_{l,i}(w))$. Note that the lemma is applicable here because: $\sigma_{l,i}^{\gamma_j(l,i)}$ and $\sigma_{l,i}$ are differentiable at $y_{l,i}(w)$; they coincide on $\mathcal{I}_{l,i}^{\gamma_j(l,i)}$; $y_{l,i}(w) \in cl(\mathcal{I}_{l,i}^{\gamma(l,i)})$ (by $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma}$); and $int(\mathcal{I}_{l,i}^{\gamma(l,i)}) \neq \emptyset$ (by $y_{l,i}(w) \notin \mathcal{I}_{l,i}^{\gamma(l,i)}$). Therefore, Eq. (B.25) holds and this completes the proof. □

B.3.5 Theorem 5.6 (Main Proof)

Theorem 5.6. *If z_L has bias parameters, then for all $w \in \mathbb{R}^W$ at which z_L is differentiable,*

$$D^{\mathtt{AD}}z_L(w) = Dz_L(w).$$

This implies that $|\mathsf{inc}_\Omega(z_L)| = 0$.

Proof. Let $w \in \mathbb{R}^W$ such that z_L is differentiable at w (i.e., $Dz_L(w) \neq \bot$). By Lemma B.8, there is (unique) $\gamma \in \Gamma$ such that $w \in \mathcal{R}^\gamma$. Using the γ, we obtain the conclusion:

$$\begin{aligned}
Dz_L(w) &= \big[\, D_1 z_L(w) \;\big|\; \cdots \;\big|\; D_W z_L(w) \,\big] \\
&= \big[\, D_1 z_L^{\gamma_1}(w) \;\big|\; \cdots \;\big|\; D_W z_L^{\gamma_W}(w) \,\big] \quad \text{for some } \gamma_j \in \Gamma \text{ with } w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_j} \;\; (j \in [W]) \\
&= \big[\, D_1 z_L^{\gamma}(w) \;\big|\; \cdots \;\big|\; D_W z_L^{\gamma}(w) \,\big] \\
&= Dz_L^{\gamma}(w) = D^{\mathtt{AD}}z_L(w).
\end{aligned}$$

Here the second line uses Lemma B.37 with that z_L is differentiable at w. The third line uses Lemma B.38 with the following: z_L is differentiable at w; τ_l has bias parameters for all $l \in [L]$ (by assumption); and $w \in \mathcal{R}^\gamma \subseteq \mathcal{R}_{\mathsf{cl}}^{\gamma}$ and $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_j}$ for all $j \in [W]$ (by the second line). The last line uses Lemma B.12 with $w \in \mathcal{R}^\gamma$. □

B.3.6 Lemmas (Technical: Part 3)

Lemma B.39. *Let z_L be a neural network, $w \in \mathbb{R}^W$, and $A \subseteq \mathsf{Idx}$. Suppose that $w \notin pbd(\{u \in \mathbb{R}^W \,|\, y_{l,i}(u) = c\})$ for all $(l,i) \in A$ and $c \in \mathsf{ndf}(\sigma_{l,i})$. Then, there is a neural network z'_L (which*

consists of τ'_l, $\sigma'_{l,i}$, y'_l, z'_l, and $D^{\mathtt{AD}}\sigma'_{l,i}$) satisfying the following conditions:

① *$Dz'_L(w) = Dz_L(w)$, $D^{\mathtt{AD}}z'_L(w) = D^{\mathtt{AD}}z_L(w)$, and $\tau'_l = \tau_l$ for all $l \in [L]$.*

② *$y'_{l,i}(w) \notin \mathsf{ndf}(\sigma'_{l,i})$ for all $(l,i) \in A$.*

Proof. Consider the setup given above. Define a function f from neural networks to $\mathbb{N}$ as:

$$f(z'_L) \triangleq \left|\left\{ (l,i,c) \;\middle|\; (l,i) \in A,\, c \in \mathsf{ndf}(\sigma'_{l,i}),\, w \in int\big(\{u \in \mathbb{R}^W \,|\, y'_{l,i}(u) = c\}\big) \right\}\right|.$$

Note that $f(z_L) \in \mathbb{N}$ (i.e., $f(z_L) < \infty$), because $\sigma_{l,i}$ is continuous, piecewise-analytic and so $|\mathsf{ndf}(\sigma_{l,i})| < \infty$ for all $(l,i) \in \mathsf{Idx}$ (by Theorem B.4). The proof proceeds by induction on $f(z_L)$.

Case $f(z_L) = 0$**.** We claim that z_L satisfies ①-②. Clearly, it satisfies ①. Further, it also satisfies ②: by the assumption and $f(z_L) = 0$, we have that for all $(l,i) \in A$ and $c \in \mathsf{ndf}(\sigma_{l,i})$,

$$w \notin pbd\big(\{u \in \mathbb{R}^W \mid y_{l,i}(u) = c\}\big) \cup int\big(\{u \in \mathbb{R}^W \mid y_{l,i}(u) = c\}\big) = \{u \in \mathbb{R}^W \mid y_{l,i}(u) = c\},$$

which implies that $y_{l,i}(w) \neq c$. Hence, $y_{l,i}(w) \notin \mathsf{ndf}(\sigma_{l,i})$ for all $(l,i) \in A$, as desired.

Case $f(z_L) > 0$**.** Since $f(z_L) > 0$, there are $(l,i) \in A$ and $c \in \mathsf{ndf}(\sigma_{l,i})$ such that $w \in int(\{u \in \mathbb{R}^W | \, y_{l,i}(u) = c\})$. This implies that there is an open $U \subseteq \mathbb{R}^W$ such that $w \in U$ and $y_{l,i}(u) = c$ for all $u \in U$. Let z'_L be the exactly same neural network as z_L except that it uses different $\sigma'_{l,i}$ and $D^{\mathtt{AD}}\sigma'_{l,i}$:

$$\sigma'_{l,i}(x) \triangleq D^{\mathtt{AD}}\sigma_{l,i}(c) \cdot (x - c) + \sigma_{l,i}(c), \qquad D^{\mathtt{AD}}\sigma'_{l,i}(x) \triangleq D^{\mathtt{AD}}\sigma_{l,i}(c).$$

Note that $\sigma'_{l,i}$ and $D^{\mathtt{AD}}\sigma'_{l,i}$ satisfy the assumptions in §5.2.2–§5.2.3: the former is continuous and piecewise-analytic (since it is differentiable), and the latter is an extended derivative of the former (since the former is differentiable and $D^{\mathtt{AD}}\sigma'_{l,i} = D\sigma'_{l,i}$). Moreover, z'_L satisfies ① because $y_{l,i}(u) = c$ for all $u \in U$, and because $\sigma'_{l,i}(c) = \sigma_{l,i}(c)$ and $D^{\mathtt{AD}}\sigma'_{l,i}(c) = D^{\mathtt{AD}}\sigma_{l,i}(c)$. Further, we have that $w \notin pbd(\{u \in \mathbb{R}^W | \, y'_{l',i'}(u) = c'\})$ for all $(l',i') \in A$ and all $c' \in \mathsf{ndf}(\sigma'_{l',i'})$, and that

$$f(z'_L) = f(z_L) - 1,$$

where both results follow from $\mathsf{ndf}(\sigma'_{l,i}) = \emptyset$, $\mathsf{ndf}(\sigma'_{l',i'}) = \mathsf{ndf}(\sigma_{l',i'})$ for all $(l',i') \neq (l,i)$, and $y'_{l',i'} = y_{l',i'}$ on U for all $(l',i') \in \mathsf{Idx}$. Hence, we can apply induction to z'_L, and by induction hypothesis, there is a neural network z''_L such that (z''_L, z'_L) (instead of (z'_L, z_L)) satisfies ①-②. From this, and since (z'_L, z_L) satisfies ① (by the above), we conclude (z''_L, z_L) satisfies ①-②, as desired. □

Lemma B.40. *We have*

$$\mathsf{inc}_{\mathbb{R}}(z_L) \subseteq \bigcup_{(l,i)\in \mathsf{Idx}} \bigcup_{c\in \mathsf{ndf}(\sigma_{l,i})\cap S_l} pbd\big(\{w \in \mathbb{R}^W \mid y_{l,i}(w) = c\}\big),$$

where $S_l \subseteq \mathbb{R}$ is defined by $S_l \triangleq \emptyset$ if τ_l has bias parameters, and $S_l \triangleq \mathbb{R}$ otherwise.

Proof. Let $U \subseteq \mathbb{R}^W$ be the RHS of the above equation:

$$U \triangleq \bigcup_{l\in[L]:S_l=\mathbb{R}} \bigcup_{i\in[N_l]} \bigcup_{c\in \mathsf{ndf}(\sigma_{l,i})} pbd\big(\{w \in \mathbb{R}^W \mid y_{l,i}(w) = c\}\big).$$

Then, it suffices to show that for any $w \in \mathbb{R}^W$, $w \notin U$ implies $w \notin \mathsf{inc}_{\mathbb{R}}(z_L)$. Consider any $w \in \mathbb{R}^W$ with $w \notin U$. We want to show $w \notin \mathsf{inc}_{\mathbb{R}}(z_L)$. If z_L is not differentiable at w, then $w \notin \mathsf{inc}_{\mathbb{R}}(z_L)$ clearly holds by the definition of $\mathsf{inc}_{\mathbb{R}}(-)$. Hence, assume that z_L is differentiable at w. By the definition of $\mathsf{inc}_{\mathbb{R}}(-)$, it suffices to show the following:

$$D^{\mathsf{AD}} z_L(w) = Dz_L(w). \tag{B.27}$$

We prove this in two steps.

Step 1. Since z_L at w does not satisfy the assumption of Lemma B.38 (which we will apply to show Eq. (B.27)), we construct another neural network z'_L that is identical to z_L nearby w while satisfying the assumption. To do so, we apply Lemma B.39 to (z_L, w, A) with $A \triangleq \{(l,i) \in \mathsf{Idx} \mid S_l = \mathbb{R}\}$. The lemma is applicable here, since $w \notin pbd(\{v \in \mathbb{R}^W \mid y_{l,i}(v) = c\})$ for all $(l,i) \in A$ and $c \in \mathsf{ndf}(\sigma_{l,i})$ (by $w \notin U$). Hence, by Lemma B.39, we get a neural network z'_L (which consists of τ'_l, $\sigma'_{l,i}$, y'_l, z'_l, and $D^{\mathsf{AD}}\sigma'_{l,i}$) satisfying the following conditions:

① $Dz'_L(w) = Dz_L(w)$, $D^{\mathsf{AD}} z'_L(w) = D^{\mathsf{AD}} z_L(w)$, and $\tau'_l = \tau_l$ for all $l \in [L]$.

② $y'_{l,i}(w) \notin \mathsf{ndf}(\sigma'_{l,i})$ for all $(l,i) \in A$.

Step 2. We now prove Eq. (B.27) based on z'_L. Let Γ', $\mathcal{R}'$, and $\mathcal{R}'_{\mathsf{cl}}$ be the counterparts of Γ, $\mathcal{R}$, and $\mathcal{R}_{\mathsf{cl}}$ for z'_L. Then, by Lemma B.8, there is $\gamma' \in \Gamma'$ such that $w \in \mathcal{R}'^{\gamma'}$. Using z'_L and γ', we obtain Eq. (B.27):

$$\begin{aligned}
Dz_L(w) &= Dz'_L(w) \\
&= \big[\, D_1 z'_L(w) \;\big|\; \cdots \;\big|\; D_W z'_L(w) \,\big] \\
&= \big[\, D_1 z'^{\gamma'_1}_L(w) \;\big|\; \cdots \;\big|\; D_W z'^{\gamma'_W}_L(w) \,\big] \quad \text{for some } \gamma'_j \in \Gamma' \text{ with } w \in \mathcal{R}'^{\gamma'_j}_{\mathsf{cl}} \ (j \in [W]) \\
&= \big[\, D_1 z'^{\gamma'}_L(w) \;\big|\; \cdots \;\big|\; D_W z'^{\gamma'}_L(w) \,\big]
\end{aligned}$$

$$= Dz'_L{}^{\gamma'}(w) = D^{\mathtt{AD}} z'_L(w) = D^{\mathtt{AD}} z_L(w).$$

Here the first and last lines use ① and Lemma B.12 with $w \in \mathcal{R}'^{\gamma'}$. The third line uses Lemma B.37 with that z'_L is differentiable at w (by ①). The fourth line uses Lemma B.38 with the following: z'_L is differentiable at w (by ①); for all $(l,i) \in \mathsf{Idx}$, if τ'_l does not have bias parameters, then $y'_{l,i}(w) \notin \mathsf{ndf}(\sigma'_{l,i})$, i.e., $\sigma'_{l,i}$ is differentiable at $y'_{l,i}(w)$ (by ① and ②); and $w \in \mathcal{R}'^{\gamma'} \subseteq \mathcal{R}'_{\mathsf{cl}}{}^{\gamma'}$ and $w \in \mathcal{R}'_{\mathsf{cl}}{}^{\gamma'_j}$ for all $j \in [W]$ (by the third line). □

B.3.7 Theorem 5.14 (Main Lemma)

Lemma B.41. *Suppose that for every $l \in [L]$, one of the following holds:*

(a) τ_l has bias parameters, or

(b) τ_l is well-structured biaffine.

In the case of (b), let $\phi_{l,i}$ be the partial map described in Lemma B.22 for all $i \in [N_l]$. Then,

$$\mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in\mathsf{Idx}} \bigcup_{c\in A_{l,i}} B_{l,i}(c),$$

where $A_{l,i} \subseteq \mathbb{R}$ and $B_{l,i}(c) \subseteq \Omega$ are defined as

$$A_{l,i} \triangleq \begin{cases} (\mathsf{ndf}(\sigma_{l,i}) \cap S_l) & \text{if } \tau_{l+1} \text{ satisfies the condition (a) or } l = L \\ (\mathsf{ndf}(\sigma_{l,i}) \cap S_l) \cup \mathsf{bdz}(\sigma_{l,i}) & \text{if } \tau_{l+1} \text{ satisfies the condition (b),} \end{cases}$$

$$B_{l,i}(c) \triangleq \begin{cases} \{w \in \Omega \mid y_{l,i}(w) = c\} & \text{if } \tau_l \text{ satisfies the condition (a)} \\ \{w \in \Omega \mid y_{l,i}(w) = c \wedge \bigvee_{j\in dom(\phi_{l,i})} z_{l-1,\phi_{l,i}(j)}(w) \neq 0\} & \text{if } \tau_l \text{ satisfies the condition (b),} \end{cases}$$

and $S_l \subseteq \mathbb{R}$ is defined as $S_l \triangleq \emptyset$ if τ_l has bias parameters, and $S_l \triangleq \mathbb{R}$ otherwise.

Proof. We obtain the conclusion by chaining Lemma B.40, Lemma B.23 (which is applicable by the assumption on τ_l), and $\mathsf{ndf}_\Omega(z_L) \cup \mathsf{inc}_\Omega(z_L) = \big(\mathsf{ndf}_\mathbb{R}(z_L) \cup \mathsf{inc}_\mathbb{R}(z_L)\big) \cap \Omega$. □

B.3.8 Theorem 5.14 (Main Proof)

Theorem 5.14. *If τ_l either has bias parameters or is well-structured biaffine for all $l \in [L]$, then*

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \leq \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in\mathsf{Idx}} \Big|\big(\mathsf{ndf}(\sigma_{l,i}) \cap S_l\big) \cup \big(\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1}\big)\Big|,$$

where $S_l \subseteq \mathbb{R}$ is defined by

$$S_l \triangleq \begin{cases} \emptyset & \text{if } l > L \text{ or } \tau_l \text{ has bias parameters} \\ \mathbb{R} & \text{otherwise.} \end{cases}$$

Proof. Observe that

$$\mathsf{inc}_\Omega(z_L) \subseteq \bigcup_{(l,i)\in \mathsf{Idx}} \bigcup_{c \in A_{l,i}} B_{l,i}(c), \qquad |B_{l,i}(c)| \le |\mathbb{M}|^{W-1}, \tag{B.28}$$

where $S_l \subseteq \mathbb{R}$, $A_{l,i} \subseteq \mathbb{R}$ and $B_{l,i}(c) \subseteq \Omega$ for $l \in [L]$ are defined as in Lemma B.41. Here the first equation is by Lemma B.41 and the second equation is by Lemmas B.19 and B.25, where these lemmas are applicable by the definition of $B_{l,i}(c)$ and because τ_l either has bias parameters or is well-structured biaffine (by assumption). Observe further that

$$A_{l,i} = (\mathsf{ndf}(\sigma_{l,i}) \cap S_l) \cup (\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1}), \tag{B.29}$$

by the definition of $A_{l,i}$ and S_l, where we use $S_{L+1} \triangleq \emptyset$. Combining the above observations, we obtain the conclusion:

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \le \sum_{(l,i)\in\mathsf{Idx}} \sum_{c\in A_{l,i}} \frac{|B_{l,i}(c)|}{|\Omega|} \le \sum_{(l,i)\in\mathsf{Idx}} \left|(\mathsf{ndf}(\sigma_{l,i}) \cap S_l) \cup (\mathsf{bdz}(\sigma_{l,i}) \cap S_{l+1})\right| \cdot \frac{|\mathbb{M}|^{W-1}}{|\mathbb{M}|^W},$$

where the first inequality uses Eq. (B.28) and the second inequality uses Eqs. (B.28) and (B.29). □

B.4 Lower Bounds on $|\mathsf{ndf}_\Omega(z_L)|$ and $|\mathsf{inc}_\Omega(z_L)|$

B.4.1 Theorem 5.8 (Main Proof)

Theorem 5.8. *For any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ with $1 \le |\mathbb{M}| < \infty$, $n \ge 2$, and $\alpha \le |\mathbb{M}|/(n-1)$, there is a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that satisfies*

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \ge \frac{1}{2} \cdot \frac{1}{|\mathbb{M}|} \sum_{(l,i)\in\mathsf{Idx}} |\mathsf{ndf}(\sigma_{l,i})|$$

and the following: z_L has bias parameters, it has $n+1$ neurons, and $|\mathsf{ndf}(\sigma_{1,i})| = \alpha$ for all $i \in [N_1]$.

Proof. Consider any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ that satisfy the assumption. We claim that there is a neural network z_L that has $L = 2$ layers, $N = n+1$ neurons, and $W = n+1$ parameters, and satisfies the given inequality.

We first define a few components to be used in the network. Let $\{x_1, \ldots, x_\alpha\} \subseteq \mathbb{M}$ be distinct

machine-representable numbers, and $h : \mathbb{R} \to \mathbb{R}$ be a continuous, piecewise-analytic function such that $\mathsf{ndf}(h) = \{x_1, \ldots, x_\alpha\}$. Note that such x_j always exists since $|\mathbb{M}| \geq \alpha$ (by assumption). Using h, define a function $f : \mathbb{R}^W \to \mathbb{R}$ as

$$f(w) = w_{n+1} + \sum_{i \in [n]} h(w_i).$$

We assume here (and in the rest of the proof) that $w \in \mathbb{R}^W$ is represented as $w = (w_1, \ldots, w_W)$ for $w_i \in \mathbb{R}$ (instead of $w = (w_{1,1}, w_{1,2}, \ldots, w_{L,W_L})$ with $w_{l,j} \in \mathbb{R}$ as we assumed so far).

Given these, we construct a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that is essentially the same as f, as follows

$$\begin{aligned}
&z_0(w) = 0 \in \mathbb{R}, \\
&y_1(w) = (w_1, \ldots, w_n) \in \mathbb{R}^n, && z_1(w) = (h(w_1), \ldots, h(w_n)) \in \mathbb{R}^n, \\
&y_2(w) = f(x) \in \mathbb{R}, && z_2(w) = f(x) \in \mathbb{R}.
\end{aligned}$$

Then, z_L has 2 layers, $n+1$ neurons, and $n+1$ parameters, and $|\mathsf{ndf}(\sigma_{1,i})| = |\mathsf{ndf}(h)| = \alpha$ for all i. Also, we can easily make all τ_l have bias parameters (e.g., by using $\tau_1(x, w_1, \ldots, w_n) = (x+w_1, \ldots, x+w_n)$). What remains is to prove that z_L satisfies the inequality in the conclusion. To do so, observe that

$$\begin{aligned}
\mathsf{ndf}_\Omega(z_L) &\supseteq \{w \in \Omega \mid w_i \in \mathsf{ndf}(h) \text{ for some } i \in [n]\} \\
&= \Omega \setminus \{w \in \Omega \mid w_i \notin \mathsf{ndf}(h) \text{ for all } i \in [n]\},
\end{aligned}$$

which follows from the definition of f and $\mathsf{ndf}(h) \subseteq \mathbb{M}$. From this, we have

$$\begin{aligned}
\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} &\geq \frac{1}{|\mathbb{M}|^{n+1}}\Big(|\mathbb{M}|^{n+1} - |\mathbb{M}| \cdot (|\mathbb{M}| - \alpha)^n\Big) = 1 - \Big(1 - \frac{\alpha}{|\mathbb{M}|}\Big)^n \\
&\geq 1 - \Big(1 - n\frac{\alpha}{|\mathbb{M}|} + \frac{1}{2}n(n-1)\Big(\frac{\alpha}{|\mathbb{M}|}\Big)^2\Big) = \frac{n\alpha}{|\mathbb{M}|}\Big(1 - \frac{n-1}{2}\frac{\alpha}{|\mathbb{M}|}\Big) \\
&\geq \frac{1}{2} \cdot \frac{n\alpha}{|\mathbb{M}|},
\end{aligned}$$

where the first inequality uses $\mathsf{ndf}(h) \subseteq \mathbb{M}$ and $|\mathsf{ndf}(h)| = \alpha$, the second inequality follows from $(1-x)^n \leq 1 - nx + \frac{1}{2}n(n-1)x^2$ (for any $x \leq 1$ and $n \in \mathbb{N}$) and $\alpha \leq |\mathbb{M}|$, and the third inequality is by the assumption that $\alpha \leq |\mathbb{M}|/(n-1)$. By combining this result and

$$\frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} |\mathsf{ndf}(\sigma_{l,i})| = \frac{n\alpha}{|\mathbb{M}|},$$

we obtain the desired inequality. □

B.4.2 Theorem 5.13 (Main Proof)

Theorem 5.13. *For any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ with $1 \leq |\mathbb{M}| < \infty$, $n \geq 4$, and $\alpha \leq |\mathbb{M}|/(n-1)$, there is a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that satisfies*

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{9} \cdot \frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \left| \mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i}) \right|$$

and the following: (i) τ_l is well-structured biaffine without bias parameters for all $l < L$, and has bias parameters for $l = L$; (ii) z_L has $n+1$ neurons; and (iii) $|\mathsf{ndf}(\sigma_{1,i})| = \alpha$, $|\mathsf{bdz}(\sigma_{1,i})| = 0$ for all i. We get the same result for (i), (ii'), and (iii'): (ii') z_L has $2n+1$ neurons; and (iii') $|\mathsf{ndf}(\sigma_{1,i})| = 0$, $|\mathsf{bdz}(\sigma_{1,i})| = \alpha$ for all i.

Proof. We prove the two cases (one for (i), (ii), (iii), and the other for (i), (ii'), (iii')) as follows. Consider any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ that satisfy the assumption. Let $\{x_1, \ldots, x_\alpha\} \subseteq \mathbb{M}$ be distinct machine-representable numbers; such x_j always exists since $|\mathbb{M}| \geq \alpha$ (by assumption). In the rest of the proof, we assume that $w \in \mathbb{R}^W$ is represented as $w = (w_1, \ldots, w_W)$ for $w_i \in \mathbb{R}$, as in the proof of Theorem 5.8 (see §B.4.1).

First case. Let $W = n+1$ and $h : \mathbb{R} \to \mathbb{R}$ be a continuous, piecewise-analytic function such that $\mathsf{ndf}(h) = \{x_1, \ldots, x_\alpha\}$ and $h(x) > 0$ for all $x \in \mathbb{R}$. Using this h, define a function $f : \mathbb{R}^W \to \mathbb{R}$ as

$$f(w) = w_{n+1} + \sum_{i \in [n]} h(w_i).$$

We now construct a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that is essentially the same as f, as follows:

$$\begin{aligned} &z_0(w) = 1 \in \mathbb{R}, \\ &y_1(w) = (w_1, \ldots, w_n) \in \mathbb{R}^n, && z_1(w) = (h(w_1), \ldots, h(w_n)) \in \mathbb{R}^n, \\ &y_2(w) = f(x) \in \mathbb{R}, && z_2(w) = f(x) \in \mathbb{R}. \end{aligned}$$

Then, z_L has $L = 2$ layers, $N = n+1$ neurons, and $W = n+1$ parameters, and $|\mathsf{ndf}(\sigma_{1,i})| = |\mathsf{ndf}(h)| = \alpha$ and $|\mathsf{bdz}(\sigma_{1,i})| = |\mathsf{bdz}(h)| = 0$ for all i. Also, we can easily make τ_l be well-structured biaffine without bias parameters for all $l < L$, and make τ_L have bias parameters (e.g., by using $\tau_1(x, w_1, \ldots, w_n) = (x \cdot w_1, \ldots, x \cdot w_n)$). This shows that (i), (ii), and (iii) are satisfied.

What remains is to prove that z_L satisfies the inequality in the conclusion. To do so, observe that

$$\mathsf{ndf}_\Omega(z_L) \supseteq \{w \in \Omega \mid w_i \in \mathsf{ndf}(h) \text{ for some } i \in [n]\},$$

which follows from the definition of f and $\mathsf{ndf}(h) \subseteq \mathbb{M}$. From this, we have

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{2} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

as shown in the proof of Theorem 5.8 (see §B.4.1). Here we used $\mathsf{ndf}(h) \subseteq \mathbb{M}$ and $|\mathsf{ndf}(h)| = \alpha$, as well as $\alpha \leq |\mathbb{M}|/(n-1)$ and $n \geq 2$ (by assumption). Further, observe that

$$\frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \big|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\big| = \frac{n\alpha + 1}{|\mathbb{M}|} = \Big(1 + \frac{1}{n\alpha}\Big) \cdot \frac{n\alpha}{|\mathbb{M}|} \leq \frac{3}{2} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

where the inequality uses $n \geq 2$ and $\alpha \geq 1$ (by assumption). From these results, we obtain the desired inequality.

Second case. Let $W = n + 2$ and $h : \mathbb{R} \to \mathbb{R}$ be an analytic function such that $h(x_j) = 0$ and $Dh(x_j) = 1$ for all $j \in [\alpha]$, and $|\mathsf{bdz}(h)| = \alpha$. We remark that such a function h always exists due to Hermite interpolation [21]. Using this h, define a function $f : \mathbb{R}^W \to \mathbb{R}$ as

$$f(w) = w_{n+2} + \sum_{i \in [n]} \mathrm{ReLU}(h(w_i) \cdot w_{n+1}).$$

We now construct a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that is essentially the same as f, as follows:

$$\begin{aligned}
&z_0(w) = 1, \\
&y_1(w) = (w_1, \dots, w_n), && z_1(w) = (h(w_1), \dots, h(w_n)), \\
&y_2(w) = (h(w_1) \cdot w_{n+1}, \dots, h(w_n) \cdot w_{n+1}), && z_2(w) = (\mathrm{ReLU}(h(w_1) \cdot w_{n+1}), \dots, \mathrm{ReLU}(h(w_n) \cdot w_{n+1})), \\
&y_3(w) = f(x), && z_3(w) = f(x).
\end{aligned}$$

Then, z_L has $L = 3$ layers, $N = 2n + 1$ neurons, and $W = n + 2$ parameters, and $|\mathsf{ndf}(\sigma_{1,i})| = |\mathsf{ndf}(h)| = 0$ and $|\mathsf{bdz}(\sigma_{1,i})| = |\mathsf{bdz}(h)| = \alpha$ for all i. Also, we can easily make τ_l be well-structured biaffine without bias parameters for all $l < L$, and make τ_L have bias parameters, as discussed above. This shows that (i), (ii'), and (iii') are satisfied.

What remains is to prove that z_L satisfies the inequality in the conclusion. To do so, observe that

$$\begin{aligned}
\mathsf{ndf}_\Omega(z_L) &\supseteq \{w \in \Omega \mid w_{n+1} \neq 0 \text{ and } w_i \in \mathsf{bdz}(h) \text{ for some } i \in [n]\} \\
&= \Omega \setminus \big(\{w \in \Omega \mid w_{n+1} = 0\} \cup \{w \in \Omega \mid w_i \notin \mathsf{bdz}(h) \text{ for all } i \in [n]\}\big),
\end{aligned}$$

which follows from the definition of f and $\mathsf{bdz}(h) \subseteq \mathbb{M}$. From this, we have

$$\frac{|\mathsf{ndf}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{|\mathbb{M}|^{n+2}} \Big(|\mathbb{M}|^{n+2} - |\mathbb{M}|^{n+1} - |\mathbb{M}|^2 \cdot (|\mathbb{M}| - \alpha)^n\Big)$$

$$\geq \frac{1}{2} \cdot \frac{n\alpha}{|\mathbb{M}|} - \frac{1}{|\mathbb{M}|} = \left(\frac{1}{2} - \frac{1}{n\alpha}\right) \cdot \frac{n\alpha}{|\mathbb{M}|}$$
$$\geq \frac{1}{4} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

where the second inequality follows from an argument in the proof of Theorem 5.8 (see §B.4.1), and the third inequality uses $n \geq 4$ and $\alpha \geq 1$ (by assumption). Note that when proving the second inequality, we used $\mathsf{bdz}(h) \subseteq \mathbb{M}$ and $|\mathsf{bdz}(h)| = \alpha$, as well as $\alpha \leq |\mathbb{M}|/(n-1)$ and $n \geq 2$ (by assumption). Further, observe that

$$\frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \left|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\right| = \frac{n\alpha + n + 1}{|\mathbb{M}|} = \left(1 + \frac{1}{\alpha} + \frac{1}{n\alpha}\right) \cdot \frac{n\alpha}{|\mathbb{M}|} \leq \frac{9}{4} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

where the inequality uses $n \geq 4$ and $\alpha \geq 1$ (by assumption). From these results, we obtain the desired inequality. □

B.4.3 Theorem 5.15 (Main Proof)

Theorem 5.15. *For any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ with $1 \leq |\mathbb{M}| < \infty$, $n \geq 4$, and $\alpha \leq |\mathbb{M}|/(n-1)$, there is a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that satisfies*

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{13} \cdot \frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \left|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\right|$$

and the following: (i) τ_l is well-structured biaffine without bias parameters for all $l < L$, and has bias parameters for $l = L$; (ii) z_L has $2n+1$ neurons; and (iii) $|\mathsf{ndf}(\sigma_{1,i})| = \alpha$, $|\mathsf{bdz}(\sigma_{1,i})| = 0$ for all i. We get the same result for (i), (ii'), and (iii'): (ii') z_L has $3n+1$ neurons; and (iii') $|\mathsf{ndf}(\sigma_{1,i})| = 0$, $|\mathsf{bdz}(\sigma_{1,i})| = \alpha$ for all i.

Proof. We prove the two cases (one for (i), (ii), (iii), and the other for (i), (ii'), (iii')) as follows. Consider any $\mathbb{M} \subseteq \mathbb{R}$ and $n, \alpha \in \mathbb{N}$ that satisfy the assumption. Let $\{x_1, \ldots, x_\alpha\} \subseteq \mathbb{M}$ be distinct machine-representable numbers; such x_j always exists since $|\mathbb{M}| \geq \alpha$ (by assumption). In the rest of the proof, we assume that $w \in \mathbb{R}^W$ is represented as $w = (w_1, \ldots, w_W)$ for $w_i \in \mathbb{R}$, as in the proof of Theorem 5.8 (see §B.4.1).

First case. Let $W = n + 1$. Without loss of generality, assume that α is even and $0 < x_1 < \cdots < x_{\alpha/2}$; other cases can be handled in a similar way. Consider a continuous, piecewise-analytic function $h : \mathbb{R} \to \mathbb{R}$ that satisfies the following conditions: for all $j \in [\alpha/2]$, $h(x_j) = 1$ if j is odd, and $h(x_j) = 2$ if j is even; $\mathsf{ndf}(h) \cap (0, \infty) = \{x_1, \ldots, x_{\alpha/2}\}$; h is piecewise linear, constant on $[x_{\alpha/2}, \infty)$, and even (i.e., $h(x) = h(-x)$ for all $x \in \mathbb{R}$). For this h, consider a (consistent) extended derivative $D^{\mathsf{AD}}h : \mathbb{R} \to \mathbb{R}$ that takes the slope of the right piece of the function at non-differentiable points: e.g., $D^{\mathsf{AD}}h(x_2) = (h(x_3) - h(x_2))/(x_3 - x_2)$ and $D^{\mathsf{AD}}h(-x_2) = (h(-x_1) - h(-x_2))/(-x_1 + x_2)$. Using this

h, define a function $f : \mathbb{R}^W \to \mathbb{R}$ as

$$f(w) = w_{n+1} + \sum_{i \in [n]} h(w_i) - h(-w_i).$$

Then, by using a similar approach taken in the proof of Theorem 5.13 (see §B.4.2), we can construct a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that is essentially the same as f and satisfies the following: z_L has $L = 2$ layers, $N = 2n + 1$ neurons, and $W = n + 1$ parameters (where $2n$ neurons are at layer 1 and 1 neuron is at layer 2); τ_l is well-structured biaffine without bias parameters for all $l < L$, and has bias parameters for $l = L$; and $|\mathsf{ndf}(\sigma_{1,i})| = |\mathsf{ndf}(h)| = \alpha$ and $|\mathsf{bdz}(\sigma_{1,i})| = |\mathsf{bdz}(h)| = 0$ for all i. This shows that (i), (ii), and (iii) are satisfied.

What remains is to prove that z_L satisfies the inequality in the conclusion. To do so, observe that

$$\mathsf{inc}_\Omega(z_L) \supseteq \{w \in \Omega \mid w_i \in \{x_1, \dots, x_{\alpha/2}\} \text{ for some } i \in [n]\},$$

which follows from the definition of f and $\{x_1, \dots, x_{\alpha/2}\} \subseteq \mathbb{M}$. From this, we have

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{4} \cdot \frac{n\alpha}{|\mathbb{M}|}$$

by a similar argument to that in the proof of Theorem 5.8 (see §B.4.1). Here we used $\{x_1, \dots, x_{\alpha/2}\} \subseteq \mathbb{M}$ as well as $\alpha \leq |\mathbb{M}|/(n-1)$ and $n \geq 2$ (by assumption). Further, observe that

$$\frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \left|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\right| = \frac{2n\alpha + 1}{|\mathbb{M}|} = \left(2 + \frac{1}{n\alpha}\right) \cdot \frac{n\alpha}{|\mathbb{M}|} \leq \frac{9}{4} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

where the inequality uses $n \geq 4$ and $\alpha \geq 1$ (by assumption). From these results, we obtain the desired inequality.

Second case. Let $W = n + 2$ and $h : \mathbb{R} \to \mathbb{R}$ be an analytic function such that $h(x_j) = 0$ and $Dh(x_j) = 1$ for all $j \in [\alpha]$, and $|\mathsf{bdz}(h)| = \alpha$. Using this h, define a function $f : \mathbb{R}^W \to \mathbb{R}$ as

$$f(w) = w_{n+2} + \sum_{i \in [n]} \mathrm{ReLU}(h(w_i) \cdot w_{n+1}) - \mathrm{ReLU}(-h(w_i) \cdot w_{n+1}),$$

and let $D^{\mathsf{AD}}\mathrm{ReLU} = \mathbf{1}_{(0,\infty)}$. By using an approach similar to the above, we can construct a neural network $z_L : \mathbb{R}^W \to \mathbb{R}$ that is essentially the same as f and satisfies the following: z_L has $L = 3$ layers, $N = 3n + 1$ neurons, and $W = n + 2$ parameters (where n neurons are at layer 1, $2n$ neurons at layer 2, and 1 neuron at layer 3); τ_l is well-structured biaffine without bias parameters for all $l < L$, and has bias parameters for $l = L$; and $|\mathsf{ndf}(\sigma_{1,i})| = |\mathsf{ndf}(h)| = 0$ and $|\mathsf{bdz}(\sigma_{1,i})| = |\mathsf{bdz}(h)| = \alpha$ for all i. This shows that (i), (ii'), and (iii') are satisfied.

What remains is to prove that z_L satisfies the inequality in the conclusion. To do so, observe that

$$\mathsf{inc}_\Omega(z_L) \supseteq \{w \in \Omega \mid w_{n+1} \neq 0 \text{ and } w_i \in \mathsf{bdz}(h) \text{ for some } i \in [n]\},$$

which follows from the definition of f and $\mathsf{bdz}(h) \subseteq \mathbb{M}$. From this, we have

$$\frac{|\mathsf{inc}_\Omega(z_L)|}{|\Omega|} \geq \frac{1}{4} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

as shown in the proof of Theorem 5.13 (see §B.4.2). Here we used $\mathsf{bdz}(h) \subseteq \mathbb{M}$ and $|\mathsf{bdz}(h)| = \alpha$, as well as $1 \leq \alpha \leq |\mathbb{M}|/(n-1)$ and $n \geq 4$ (by assumption). Further, observe that

$$\frac{1}{|\mathbb{M}|} \sum_{(l,i) \in \mathsf{Idx}} \left|\mathsf{ndf}(\sigma_{l,i}) \cup \mathsf{bdz}(\sigma_{l,i})\right| = \frac{n\alpha + 2n + 1}{|\mathbb{M}|} = \left(1 + \frac{2}{\alpha} + \frac{1}{n\alpha}\right) \cdot \frac{n\alpha}{|\mathbb{M}|} \leq \frac{13}{4} \cdot \frac{n\alpha}{|\mathbb{M}|},$$

where the inequality uses $n \geq 4$ and $\alpha \geq 1$ (by assumption). From these results, we obtain the desired inequality. □

B.5 Computation of Standard Derivatives

B.5.1 Lemmas (Basic)

Lemma B.42. *Let $f, f_1, \ldots, f_n : \mathbb{R}^d \to \mathbb{R}^{d'}$ ($n \in \mathbb{N}$), $x \in \mathbb{R}^d$, and $U \subseteq \mathbb{R}^d$ be an open neighborhood of x. Suppose that for all $y \in U$, $f(y) = f_i(y)$ for some $i \in [n]$. Also, assume that $f(x) = f_i(x)$ for all $i \in [n]$, and $Df_i(x) = Df_j(x) \neq \bot$ for all $i, j \in [n]$. Then,*

$$Df(x) = Df_i(x) \neq \bot \qquad \text{for all } i \in [n].$$

Proof. Consider the setup of the statement. By the assumption, it suffices to show that $Df(x) = Df_1(x)$, which is equivalent to the following: for all $\varepsilon > 0$, there exists $\delta > 0$ such that for all $h \in \mathbb{R}^d$,

$$0 < \|h\| < \delta \quad \Longrightarrow \quad \frac{\|f(x+h) - f(x) - Df_1(x) \cdot h\|}{\|h\|} < \varepsilon,$$

where $\|\cdot\|$ denotes the ℓ_2-norm. To show this, consider any $\varepsilon > 0$. Since $Df_i(x) \neq \bot$ (by assumption), there is $\delta_i > 0$ for each $i \in [n]$ such that for all $h \in \mathbb{R}^d$,

$$0 < \|h\| < \delta_i \quad \Longrightarrow \quad \frac{\|f_i(x+h) - f_i(x) - Df_i(x) \cdot h\|}{\|h\|} = \frac{\|f_i(x+h) - f_1(x) - Df_1(x) \cdot h\|}{\|h\|} < \varepsilon,$$

where the equality is by assumption. Choose $0 < \delta < \min\{\delta_i \mid i \in [n]\}$ such that $\{x+h \mid \|h\| < \delta\} \subseteq U$, which is possible because U is an open neighborhood of x. Then, for all $h \in \mathbb{R}^d$, $0 < \|h\| < \delta$ implies

that

$$\frac{\|f(x+h)-f(x)-Df_1(x)\cdot h\|}{\|h\|} = \frac{\|f_j(x+h)-f_1(x)-Df_1(x)\cdot h\|}{\|h\|} < \varepsilon$$

for some $j \in [n]$, where the equality is by assumption and $x+h \in U$ and the inequality is by $\delta < \delta_j$. This proves $Df(x) = Df_1(x)$ as desired. □

B.5.2 Lemmas (Technical: Part 1)

In this subsection, we formally define the partial derivative $\partial^{\mathtt{AD}} z_L / \partial z_{l,i} \in \mathbb{R}^{N_L}$ of z_L with respect to $z_{l,i}$ that *reverse-mode* automatic differentiation computes (as a byproduct of computing $D^{\mathtt{AD}} z_L$). To do so, we fix $l' \in [L]$ and $w' \in \mathbb{R}^W$, and define $\partial^{\mathtt{AD}} z_L / \partial z_{l',i} \in \mathbb{R}^{N_L}$ at w' ($i \in [N_{l'}]$) in a similar way we defined $D^{\mathtt{AD}} z_L$ in §B.1.3.

We first define a program $\mathtt{Q}$ (different from $\mathtt{P}$ in §B.1.3) that represents a function from $\mathbb{R}^{N_{l'}}$ to $\mathbb{R}$ as follows:

$$\mathtt{Q} ::= r \mid \mathtt{x}_i \mid f(\mathtt{Q}_1, \ldots, \mathtt{Q}_n)$$

where $r \in \mathbb{R}$, $i \in [N_{l'}]$, $f \in \{\tau_{l,i}, \sigma_{l,i} \mid (l,i) \in \mathsf{Idx}, l > l'\}$, and $n \in \mathbb{N}$. This definition says that a program $\mathtt{Q}$ can be either a real-valued constant r, a real-valued variable $\mathtt{x}_i$ denoting the neuron $z_{l',i}$, or the application of a function $f : \mathbb{R}^n \to \mathbb{R}$ to subprograms $\mathtt{Q}_1, \ldots, \mathtt{Q}_n$. We focus on particular programs $\mathtt{Q}_{y_{l,i}}$ and $\mathtt{Q}_{z_{l,i}}$ ($l > l'$) that represent the neurons $y_{l,i}$ and $z_{l,i}$ but as functions of the neurons $z_{l',1}, \ldots, z_{l',N_{l'}}$ (instead of functions of parameters $w_{1,1}, w_{1,2}, \ldots, w_{L,W_L}$). These programs are defined in a canonical way as follows:

$$\begin{aligned}
\mathtt{Q}_{y_{l,i}} &\triangleq \tau_{l,i}(\mathtt{Q}_{z_{l-1,1}}, \ldots, \mathtt{Q}_{z_{l-1,N_{l-1}}}, w'_{l,1}, \ldots, w'_{l,W_l}), \\
\mathtt{Q}_{z_{l,i}} &\triangleq \sigma_{l,i}(\mathtt{Q}_{y_{l,i}}),
\end{aligned}$$

where $\mathtt{Q}_{z_{l',i}} \triangleq \mathtt{x}_i$ for $i \in [N_{l'}]$ represents the projection function from $\mathbb{R}^{N_{l'}}$ to $\mathbb{R}$. Note that $w'_{l,j}$ in the above equation is not a variable but a constant, while $\mathtt{x}_i$ in the definition of $\mathtt{Q}_{z_{l',i}}$ is a variable.

Given a program $\mathtt{Q}$, we define the function $[\![\mathtt{Q}]\!] : \mathbb{R}^{N_{l'}} \to \mathbb{R}$ that $\mathtt{Q}$ represents, and the function $[\![\mathtt{Q}]\!]^{\mathtt{AD}} : \mathbb{R}^{N_{l'}} \to \mathbb{R}^{1 \times N_{l'}}$ that reverse-mode automatic differentiation computes for $\mathtt{Q}$ (as a byproduct of computing other derivatives):

$$\begin{aligned}
[\![r]\!](x) &\triangleq r, \\
[\![\mathtt{x}_i]\!](x) &\triangleq x_i, \\
[\![f(\mathtt{Q}_1, \ldots, \mathtt{Q}_n)]\!](x) &\triangleq f\big([\![\mathtt{Q}_1]\!](x), \ldots, [\![\mathtt{Q}_n]\!](x)\big), \\
[\![r]\!]^{\mathtt{AD}}(x) &\triangleq ,
\end{aligned}$$

$$[\![\mathtt{x}_i]\!]^{\mathtt{AD}}(x) \triangleq \mathbb{1}_i,$$
$$[\![f(\mathtt{Q}_1,\ldots,\mathtt{Q}_n)]\!]^{\mathtt{AD}}(x) \triangleq D^{\mathtt{AD}}f\big([\![\mathtt{Q}_1]\!](x),\ldots,[\![\mathtt{Q}_n]\!](x)\big)\cdot\big[[\![\mathtt{Q}_1]\!]^{\mathtt{AD}}(x)\,/\cdots/\,[\![\mathtt{Q}_n]\!]^{\mathtt{AD}}(x)\big].$$

Here $(x_1,\ldots,x_{N_{l'}}) \triangleq x$ denote the scalar values of x, the notation $,\mathbb{1}_i \in \mathbb{R}^{1\times W}$ denote the zero matrix and the matrix whose entries are all zeros except for a single one at the i-th entry, $D^{\mathtt{AD}}f : \mathbb{R}^n \to \mathbb{R}^{1\times n}$ denotes a "derivative" of f used by automatic differentiation, and $[M_1\,/\cdots/\,M_n]$ denotes the matrix that stacks up matrices $M_1,\ldots,M_n$ vertically. Note that the definitions of $[\![\mathtt{Q}]\!]$ and $[\![\mathtt{Q}]\!]^{\mathtt{AD}}$ are almost the same as that of $[\![\mathtt{P}]\!]$ and $[\![\mathtt{P}]\!]^{\mathtt{AD}}$ in §B.1.3.

Using the above definitions, $\partial^{\mathtt{AD}}z_L/\partial z_{l',i}$ at w' for $i \in [N_{l'}]$ (i.e., the partial derivative of z_L with respect to $z_{l',i}$ at w' that reverse-mode automatic differentiation computes) can be defined as follows:

$$\frac{\partial^{\mathtt{AD}}z_L}{\partial z_{l',i}} \text{ at } w' \triangleq \Big[[\![\mathtt{Q}_{z_{L,1}}]\!]^{\mathtt{AD}}(z_{l'}(w'))\,/\cdots/\,[\![\mathtt{Q}_{z_{L,N_L}}]\!]^{\mathtt{AD}}(z_{l'}(w'))\Big]_{1:N_L,\,i} \in \mathbb{R}^{N_L}.$$

Lemma B.43 (shown below) shows that $\partial^{\mathtt{AD}}z_L/\partial z_{l,i}$ can be expressed in terms of $\widetilde{z}^{\gamma}_{l+1}$ (defined in §B.3), as $D^{\mathtt{AD}}z_L$ can be expressed in terms of z^{γ}_L (Lemma B.12). We will rely on this lemma in the rest of this section, when working with $\partial^{\mathtt{AD}}z_L/\partial z_{l,i}$.

Lemma B.43. *Let $\gamma \in \Gamma$. Then, for all $l \in [L]$ and $w \in \mathcal{R}^{\gamma}$,*

$$\frac{\partial^{\mathtt{AD}}z_L}{\partial z_{l,i}} \text{ at } w = D_i\widetilde{z}^{\gamma}_{l+1}\big(z_l(w), w_{l+1},\ldots,w_L\big).$$

Proof. The proof is similar to Lemma B.12, except that it uses Lemma B.33 instead of Lemma B.11; thus, we omit it. □

B.5.3 Lemmas (Technical: Part 2)

Lemma B.44. *Let $w \in \mathbb{R}^W$. Suppose that for all $(l,i) \in \mathsf{Idx}$, $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ implies that*

$$D_i\widetilde{z}^{\gamma}_{l+1}(z_l(w), w_{l+1},\ldots,w_L) = (0,\ldots,0)$$

for the $\gamma \in \Gamma$ with $w \in \mathcal{R}^{\gamma}$. Then, for all $l \in [L+1]$ and $\gamma_1,\gamma_2 \in \Gamma$ with $w \in \mathcal{R}^{\gamma_1}_{\mathsf{cl}} \cap \mathcal{R}^{\gamma_2}_{\mathsf{cl}}$,

$$D\widetilde{z}^{\gamma_1}_{l}(z_{l-1}(w), w_l,\ldots,w_L) = D\widetilde{z}^{\gamma_2}_{l}(z_{l-1}(w), w_l,\ldots,w_L).$$

Proof. The proof is similar to that of Lemma B.38, except that this lemma assumes that certain partial derivatives are all zero while Lemma B.38 derives this assumption (in addition to proving the conclusion of this lemma). Let $w \in \mathbb{R}^W$ that satisfies the assumption of this lemma. The proof proceeds by induction on l (starting from $l = L+1$).

Case $l = L+1$. In this case, $\widetilde{z}^{\gamma}_{L+1}$ is the identity function for all $\gamma \in \Gamma$. Hence, the conclusion

clearly holds.

Case $l < L+1$. For simple notation, let $x \triangleq (z_{l-1}(w), w_l, \dots, w_L)$ and $x' \triangleq (z_l(w), w_{l+1}, \dots, w_L)$. Observe that the following hold for any $\gamma \in \Gamma$ with $w \in \mathcal{R}^\gamma_{\mathsf{cl}}$, due to Eqs. (B.21)–(B.23) in the proof of Lemma B.38:

$$D\widetilde{z}^\gamma_l(x) = D\widetilde{z}^\gamma_{l+1}\big((\widetilde{\sigma}^\gamma_l \circ \widetilde{\tau}_l)(x)\big) \cdot D\widetilde{\sigma}^\gamma_l\big(\widetilde{\tau}_l(x)\big) \cdot D\widetilde{\tau}_l(x),$$

$$\Big(D\widetilde{z}^\gamma_{l+1}((\widetilde{\sigma}^\gamma_l \circ \widetilde{\tau}_l)(x)) \cdot D\widetilde{\sigma}^\gamma_l(\widetilde{\tau}_l(x))\Big)_{*,\,i} = D_i\widetilde{z}^\gamma_{l+1}(x') \cdot \begin{cases} D\sigma^{\gamma(l,i)}_{l,i}\big(y_{l,i}(w)\big) & \text{if } i \le N_l \\ 1 & \text{if } i > N_l. \end{cases}$$

Using this observation, we prove the conclusion for l. Let $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}^{\gamma_1}_{\mathsf{cl}} \cap \mathcal{R}^{\gamma_2}_{\mathsf{cl}}$. We want to show $D\widetilde{z}^{\gamma_1}_l(x) = D\widetilde{z}^{\gamma_2}_l(x)$. By induction hypothesis on $l+1$, we obtain $D\widetilde{z}^{\gamma_1}_{l+1}(x') = D\widetilde{z}^{\gamma_2}_{l+1}(x')$. From this and the above equation, it suffices to show the following claim for all $i \in [N_l]$:

$$D_i\widetilde{z}^{\gamma_1}_{l+1}(x') \cdot D\sigma^{\gamma_1(l,i)}_{l,i}\big(y_{l,i}(w)\big) = D_i\widetilde{z}^{\gamma_1}_{l+1}(x') \cdot D\sigma^{\gamma_2(l,i)}_{l,i}\big(y_{l,i}(w)\big).$$

Let $i \in [N_l]$. We prove this claim by case analysis on i.

Subcase 1: $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$. Observe that for the $\gamma \in \Gamma$ with $w \in \mathcal{R}^\gamma$, we have

$$D_i\widetilde{z}^{\gamma_1}_{l+1}(x') = D_i\widetilde{z}^{\gamma}_{l+1}(x') = (0, \dots, 0),$$

where the first equality is by induction hypothesis on $l+1$ with $w \in \mathcal{R}^\gamma \subseteq \mathcal{R}^\gamma_{\mathsf{cl}}$, and the second equality by assumption with $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$. This directly implies the claim.

Subcase 2: $y_{l,i}(w) \notin \mathsf{ndf}(\sigma_{l,i})$. To show the claim, it suffices to show that for all $j \in [2]$,

$$D\sigma^{\gamma_j(l,i)}_{l,i}\big(y_{l,i}(w)\big) = D\sigma_{l,i}\big(y_{l,i}(w)\big).$$

This is exactly the same as Eq. (B.26) in the proof of Lemma B.38, and we can prove this in the exact same way as before. Therefore, the claim holds and this completes the proof. □

Lemma B.45. *Let $w \in \mathbb{R}^W$. Suppose that for all $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}^{\gamma_1}_{\mathsf{cl}} \cap \mathcal{R}^{\gamma_2}_{\mathsf{cl}}$,*

$$Dz^{\gamma_1}_L(w) = Dz^{\gamma_2}_L(w).$$

Then, z_L is differentiable at w.

Proof. Consider the setup of this lemma. To apply Lemma B.42, we show the following claims for $\Gamma_w \triangleq \{\gamma \in \Gamma \mid w \in \mathcal{R}^\gamma_{\mathsf{cl}}\}$:

(i) For some open neighborhood $U \subseteq \mathbb{R}^W$ of w, if $w' \in U$, then $z_L(w') = z^\gamma_L(w')$ for some $\gamma \in \Gamma_w$.

(ii) $z_L(w) = z^\gamma_L(w)$ for all $\gamma \in \Gamma_w$.

(iii) $Dz_L^{\gamma_1}(w) = Dz_L^{\gamma_2}(w) \neq \bot$ for all $\gamma_1, \gamma_2 \in \Gamma_w$.

If these claims hold, then Lemma B.42 implies that $Dz_L(w) \neq \bot$ (i.e., z_L is differentiable at w). So what remains is to show these claims. First, (iii) follows from the assumption of this lemma and that z_L^γ is analytic for all $\gamma \in \Gamma$. Second, (ii) follows from Lemma B.31. Finally, (i) holds as follows. Consider any $\gamma \in \Gamma \setminus \Gamma_w$. Then, by $w \notin \mathcal{R}_{\mathsf{cl}}^\gamma$ and the definition of $\mathcal{R}_{\mathsf{cl}}^\gamma$, there is $(l,i) \in \mathsf{Idx}$ such that $y_{l,i}(w) \in A$ and $A \cap \mathcal{I}_{l,i}^{\gamma(l,i)} = \emptyset$ for some open $A \subseteq \mathbb{R}$. Since $y_{l,i}$ is continuous and $\mathcal{R}^\gamma \subseteq y_{l,i}^{-1}(\mathcal{I}_{l,i}^{\gamma(l,i)})$, the set $U^\gamma \triangleq y_{l,i}^{-1}(A)$ is an open neighborhood of w such that $U^\gamma \cap \mathcal{R}^\gamma = \emptyset$. We now define

$$U \triangleq \bigcap_{\gamma \in \Gamma \setminus \Gamma_w} U^\gamma.$$

Then, because Γ is finite, U is an open neighborhood of w such that $U \cap \bigcup_{\gamma \in \Gamma \setminus \Gamma_w} \mathcal{R}^\gamma = \emptyset$. Using this, we obtain (i) as follows: for any $w' \in U$, we have $w' \notin \bigcup_{\gamma \in \Gamma \setminus \Gamma_w} \mathcal{R}^\gamma$ and so $w' \in \mathcal{R}^\gamma$ for some $\gamma \in \Gamma_w$ (by Lemma B.8); this implies that $z_L(w') = z_L^\gamma(w')$ (by Lemma B.11). This completes the proof. □

B.5.4 Theorems 5.9 and 5.16 (Main Lemmas)

Lemma B.46. *Let $w \in \mathbb{R}^W$. Suppose that the following holds:*

- *For all $(l,i) \in \mathsf{Idx}$, $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ implies that $D_i \widetilde{z}_{l+1}^\gamma(z_l(w), w_{l+1}, \ldots, w_L) = \vec{0}$ for the $\gamma \in \Gamma$ with $w \in \mathcal{R}^\gamma$.*

Then, we have the following:

- *$w \notin \mathsf{ndf}_{\mathbb{R}}(z_L)$ (i.e., z_L is differentiable at w).*

Proof. Consider the setup of this lemma. For all $\gamma_1, \gamma_2 \in \Gamma$ with $w \in \mathcal{R}_{\mathsf{cl}}^{\gamma_1} \cap \mathcal{R}_{\mathsf{cl}}^{\gamma_2}$,

$$Dz_L^{\gamma_1}(w) = D\widetilde{z}_1^{\gamma_1}(z_0(w), w_1, \ldots, w_L) = D\widetilde{z}_1^{\gamma_2}(z_0(w), w_1, \ldots, w_L) = Dz_L^{\gamma_2}(w),$$

where the first and last equalities are by Lemma B.32, and the second equality is by Lemma B.44. Then, by applying Lemma B.45, we obtain that z_L is differentiable at w, as desired. □

Lemma B.47. *Let $w \in \mathbb{R}^W$. Suppose that the following hold:*

- *$w \notin \mathsf{ndf}_{\mathbb{R}}(z_L)$ (i.e., z_L is differentiable at w).*
- *For all $(l,i) \in \mathsf{Idx}$, $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ implies that τ_l has bias parameters.*

Then, we have the following:

- *$w \notin \mathsf{ndf}_{\mathbb{R}}(z_L) \cup \mathsf{inc}_{\mathbb{R}}(z_L)$ (i.e., $D^{\mathtt{AD}} z_L(w) = Dz_L(w) \neq \bot$).*
- *For all $(l,i) \in \mathsf{Idx}$, $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ implies that $D_i \widetilde{z}_{l+1}^\gamma(z_l(w), w_{l+1}, \ldots, w_L) = \vec{0}$ for the $\gamma \in \Gamma$ with $w \in \mathcal{R}^\gamma$.*

Proof. Consider the setup in the statement. By exactly following the proof of Theorem 5.6 (given in §B.3.5) under this setup, we obtain the conclusion of Theorem 5.6: $D^{\mathsf{AD}} z_L(w) = Dz_L(w)$, which implies the first conclusion of this lemma. Moreover, the second conclusion was already shown in the proof of Lemma B.38 (which has the same assumption as this lemma), especially in Subcase 1 of Case $l < L + 1$ in the proof. This completes the proof. □

B.5.5 Theorems 5.9 and 5.16 (Main Proofs)

Theorem 5.9. *If z_L has bias parameters, then the following are equivalent for all $w \in \mathbb{R}^W$.*

- *z_L is non-differentiable at w.*
- *$y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ and $\partial^{\mathsf{AD}} z_L / \partial z_{l,i} \neq \vec{0}$ at w for some $(l,i) \in \mathsf{Idx}$.*

Proof. Let $w \in \mathbb{R}^W$. Suppose that z_L has bias parameters. Then, by Lemmas B.46 and B.47, the following are equivalent:

(i) $w \notin \mathsf{ndf}_{\mathbb{R}}(z_L)$ (i.e., z_L is differentiable at w).

(ii) For all $(l,i) \in \mathsf{Idx}$, $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ implies that $D_i \widetilde{z}^{\gamma}_{l+1}(z_l(w), w_{l+1}, \ldots, w_L) = \vec{0}$ for the $\gamma \in \Gamma$ with $w \in \mathcal{R}^{\gamma}$.

By taking the negation of (i)-(ii) and applying Lemma B.43 to (ii), we obtain the conclusion. □

Theorem 5.16. *Let $w \in \mathbb{R}^W$. If $y_{l,i}(w) \notin \mathsf{ndf}(\sigma_{l,i})$ for all $(l,i) \in \mathsf{Idx}$ such that τ_l does not have bias parameters or $\partial^{\mathsf{AD}} z_L / \partial z_{l,i} \neq \vec{0}$ at w, then*

$$D^{\mathsf{AD}} z_L(w) = Dz_L(w) \neq \bot.$$

Proof. Let $w \in \mathbb{R}^W$. Suppose that it satisfies the given assumption, which is equivalent to the following by Lemma B.43:

> For all $(l,i) \in \mathsf{Idx}$, $y_{l,i}(w) \in \mathsf{ndf}(\sigma_{l,i})$ implies that
>
> (i) τ_l has bias parameters, and
>
> (ii) $D_i \widetilde{z}^{\gamma}_{l+1}(z_l(w), w_{l+1}, \ldots, w_L) = \vec{0}$ for the $\gamma \in \Gamma$ with $w \in \mathcal{R}^{\gamma}$.

First, by Lemma B.46 with (ii), we have

> (iii) $w \notin \mathsf{ndf}_{\mathbb{R}}(z_L)$ (i.e., z_L is differentiable at w).

Next, by Lemma B.47 with (i) and (iii), we have the conclusion:

$$w \notin \mathsf{ndf}_{\mathbb{R}}(z_L) \cup \mathsf{inc}_{\mathbb{R}}(z_L) \text{ (i.e., } D^{\mathsf{AD}} z_L = Dz_L(w) \neq \bot).$$

□

B.6 Computation of Clarke Subderivatives

B.6.1 Lemmas (Basic)

Definition B.48. Let $A \subseteq \mathbb{R}^n$ and $x \in \mathbb{R}^n$ (where A does not need to contain x). For $B \subseteq \mathbb{R}^n$, we say that A *has B-directions around x* if for all $b \in B$, there is $\delta > 0$ such that $\{x + tb \mid t \in (0, \delta)\} \subseteq A$. We say that A *has sufficient directions around x* if A has B-directions around x for some $B \subseteq \mathbb{R}^n$ with $span(B) = \mathbb{R}^n$, where $span(B) \triangleq \{\sum_{i=1}^{k} t_i b_i \mid k \in \mathbb{N}, t_i \in \mathbb{R}, b_i \in B\}$ denotes the span of B.

Lemma B.49. *Let $A \subseteq \mathbb{R}^n$ and $x \in \mathbb{R}^n$.*

1. *If $x \in int(A)$, then A has $\mathbb{R}^n$-directions around x.*
2. *Let $\alpha \in \{\pm 1\}$, $\varepsilon \in \mathbb{R}_{>0} \cup \{\infty\}$, and $f : \mathbb{R}^n \to \mathbb{R}$. If f is differentiable at x and*

$$A = \{y \in \mathbb{R}^n \mid \alpha \cdot (f(y) - f(x)) \in (0, \varepsilon)\},$$

   then A has B-directions around x for

$$B \triangleq \{y \in \mathbb{R}^n \mid \alpha \cdot (Df(x) \cdot y) \in (0, \infty)\}.$$

3. *Let $A', B \subseteq \mathbb{R}^n$. If A has B-directions around x and $A \subseteq A'$, then A' has B-directions around x.*
4. *Let $A', B, B' \subseteq \mathbb{R}^n$. If A has B-directions around x and A' has B'-directions around x, then $(A \cap A')$ has $(B \cap B')$-directions around x.*
5. *If A has B-directions around x for some nonempty, open $B \subseteq \mathbb{R}^n$, then A has sufficient directions around x.*

Proof. The proofs of (1), (3), and (4) are straightforward, so we omit them.

Proof of (2). Consider the setup stated above. Assume that f is differentiable at x, and let $b \in B$. We want to show there is $\delta > 0$ such that $\{x + tb \mid t \in (0, \delta)\} \subseteq A$. We show this when $\alpha = 1$; we omit the case when $\alpha = -1$, as the proof is similar. Observe that since f is differentiable at x, there is $\delta' > 0$ such that for all $h \in \mathbb{R}^n$,

$$0 < \|h\| < \delta' \implies \frac{|f(x+h) - f(x) - Df(x) \cdot h|}{\|h\|} < \frac{Df(x) \cdot b}{2\|b\|}, \tag{B.30}$$

where $\|\cdot\|$ denotes the ℓ_2-norm. Here we used $Df(x) \cdot b > 0$ and $\|b\| > 0$, which hold by $b \in B$ and the definition of B.

We claim that $\{x + tb \mid t \in (0, \delta)\} \subseteq A$ holds for the following choice of δ:

$$\delta \triangleq \min\left\{\frac{\delta'}{\|b\|}, \frac{2\varepsilon}{3(Df(x) \cdot b)}\right\} > 0.$$

To show this, consider any $t \in (0, \delta)$. It suffices to show $x + tb \in A$. Observe that for $h = tb$, we have $0 < \|h\| = \|tb\| < \delta\|b\| \le (\delta'/\|b\|) \cdot \|b\| = \delta'$. Hence, Eq. (B.30) implies that

$$\begin{aligned} \big|f(x+tb) - f(x) - Df(x)\cdot(tb)\big| &< \|tb\| \cdot \tfrac{1}{2\|b\|}(Df(x)\cdot b) = \tfrac{1}{2}(Df(x)\cdot b)t, \\ 0 < \tfrac{1}{2}(Df(x)\cdot b)t &< f(x+tb) - f(x) < \tfrac{3}{2}(Df(x)\cdot b)t < \varepsilon, \end{aligned}$$

where the second line uses $Df(x)\cdot b > 0$ and $t < \delta \le \frac{2}{3}\varepsilon/(Df(x)\cdot b)$. From this, and by the definition of A, we have $x + tb \in A$ as desired.

Proof of (5). This follows from the fact that the span of any nonempty, open set in $\mathbb{R}^n$ is $\mathbb{R}^n$. □

Lemma B.50. *Let $f, g : \mathbb{R}^n \to \mathbb{R}^m$, $A \subseteq \mathbb{R}^n$, and $x \in \mathbb{R}^n$. Suppose that $f = g$ on $A \cup \{x\}$, A has sufficient directions around x, and f and g are differentiable at x. Then,*

$$Df(x) = Dg(x).$$

Proof. Consider the setup stated above. Since A has sufficient directions around x, there is $B \subseteq \mathbb{R}^n$ such that A has B-directions around x and $span(B) = \mathbb{R}^n$. We claim that $Df(x)\cdot b = Dg(x)\cdot b$ for all $b \in B$. Note that this claim implies the conclusion: by the claim and $span(B) = \mathbb{R}^n$, we have $Df(x)\cdot v = Dg(x)\cdot v$ for all $v \in \mathbb{R}^n$, and so $Df(x) = Dg(x)$.

We now prove the above claim. Let $b \in B$. Note that it suffices to show:

$$\|(Df(x) - Dg(x))\cdot b\| < \|b\|\varepsilon \qquad \text{for all } \varepsilon > 0,$$

since this implies $(Df(x) - Dg(x))\cdot b = 0$, where $\|\cdot\|$ denotes the ℓ_2-norm. Let $\varepsilon > 0$. Since f and g are differentiable at x, there is $\delta > 0$ such that for any $t \in (0, \delta)$,

$$\frac{\|f(x+tb) - f(x) - Df(x)\cdot(tb)\|}{\|tb\|} < \frac{\varepsilon}{2}, \qquad \frac{\|g(x+tb) - g(x) - Dg(x)\cdot(tb)\|}{\|tb\|} < \frac{\varepsilon}{2}. \tag{B.31}$$

Also, since $b \in B$, there is $\delta' > 0$ such that $\{x + tb \mid t \in (0, \delta')\} \subseteq A$. Fix $t \triangleq \min\{\delta, \delta'\}/2 > 0$. Then, we obtain the desired equation based on this t:

$$\begin{aligned} \big\|\big(Df(x) - Dg(x)\big)\cdot b\big\| &= \frac{1}{t}\big\|\big(Df(x) - Dg(x)\big)\cdot(tb)\big\| \\ &= \frac{1}{t}\big\|\big(f(x+tb) - f(x) - Df(x)\cdot(tb)\big) - \big(f(x+tb) - f(x) - Dg(x)\cdot(tb)\big)\big\| \\ &= \frac{1}{t}\big\|\big(f(x+tb) - f(x) - Df(x)\cdot(tb)\big) - \big(g(x+tb) - g(x) - Dg(x)\cdot(tb)\big)\big\| \\ &= \frac{1}{t}\Big(\big\|f(x+tb) - f(x) - Df(x)\cdot(tb)\big\| + \big\|g(x+tb) - g(x) - Dg(x)\cdot(tb)\big\|\Big) \\ &= \frac{1}{t}\cdot\Big(\frac{\varepsilon}{2} + \frac{\varepsilon}{2}\Big)\|tb\| = \|b\|\varepsilon, \end{aligned}$$

where the third line uses that $f = g$ on $A \cup \{x\}$ (by assumption) and $x + tb \in A$ (by $t < \delta'$), and the last line uses Eq. (B.31) (by $t < \delta$). □

Lemma B.51. *Let $n \in \mathbb{N}$, $\{d_i \in \mathbb{N}\}_{i\in[n]}$ such that $d_1 < \cdots < d_n$, and $\{f_i : \mathbb{R}^{d_i - 1} \to \mathbb{R}\}_{i\in[n]}$. Then, for any $c \in \mathbb{R}^n$, there is $u \in \mathbb{R}^{d_n}$ such that*

$$f_i(u_1, \ldots, u_{d_i - 1}) + u_{d_i} = c_i \qquad \text{for all } i \in [n].$$

Proof. The proof proceeds by induction on n.

Case $n = 1$. For any $c \in \mathbb{R}$, $u \triangleq (0, \ldots, 0, c - f_1(0, \ldots, 0)) \in \mathbb{R}^{d_1}$ satisfies the desired equation.

Case $n > 1$. Let $c \in \mathbb{R}^n$. By induction hypothesis on $n - 1$, there is $v \in \mathbb{R}^{d_{n-1}}$ such that $f_i(v_1, \ldots, v_{d_i - 1}) + v_{d_i} = c_i$ for all $i \in [n - 1]$. Define $u \triangleq (v, 0, \ldots, 0, c_n - f_n(v, 0, \ldots, 0)) \in \mathbb{R}^{d_n}$. Then, u satisfies the desired equations, since $(u_1, \ldots, u_{d_{n-1}}) = v$ by $d_{n-1} < d_n$. □

B.6.2 Lemmas (Technical)

In the following subsections, we consider a piecewise-C^1 (not piecewise-differentiable) representation of each $\sigma_{l,i}$, using the same notation in the previous sections. Formally, we make the following definitions.

Definition B.52. For each $(l, i) \in \mathsf{Idx}$, let

$$\{(\mathcal{I}^k_{l,i}, \sigma^k_{l,i})\}_{k\in[K_{l,i}]}$$

be a piecewise-C^1 representation of $\sigma_{l,i} : \mathbb{R} \to \mathbb{R}$ that defines $D^{\mathtt{AD}}\sigma_{l,i}$, where $K_{l,i} \in \mathbb{N}$, $\mathcal{I}^k_{l,i} \subseteq \mathbb{R}$, and $\sigma^k_{l,i} : \mathbb{R} \to \mathbb{R}$. We assume that the representation satisfies:

$$\bigcup_{k\in[K_{l,i}]} bd(\mathcal{I}^k_{l,i}) = \bigcup_{k\in[K_{l,i}]} pbd(\mathcal{I}^k_{l,i}) = \mathsf{ncdf}(\sigma_{l,i}),$$

where $\mathsf{ncdf}(f) \subseteq \mathbb{R}$ denotes the set of real numbers at which $f : \mathbb{R} \to \mathbb{R}$ is not continuously differentiable. If $D^{\mathtt{AD}}\sigma_{l,i}$ is consistent, we further assume that the representation satisfies the following:

$$int(\mathcal{I}^k_{l,i}) \neq \emptyset \text{ for all } k \in [K_{l,i}].$$

Note that such a representation always exists by Theorem B.4. Based on these new representations $\{(\mathcal{I}^k_{l,i}, \sigma^k_{l,i})\}_{k\in[K_{l,i}]}$, we define Γ, $\mathcal{R}^\gamma$, y^γ_l, z^γ_l, and σ^γ_l for $\gamma \in \Gamma$ and $l \in [L]$, as we defined them in §B.1.2; we omit their definitions here. □

Since we consider a piecewise-C^1 (not piecewise-differentiable) representation of $\sigma_{l,i}$, we have Lemma B.53 (shown below) that is stronger than Lemma B.9. Moreover, Lemmas B.8 and B.10–B.12

continue to hold under the new representations; the proofs are exactly the same as before, so we omit them.

Lemma B.53. *For all $l \in [L]$ and $\gamma \in \Gamma$, y_l and z_l are continuous, and y_l^γ and z_l^γ are C^1.*

Proof. The continuity of y_l and z_l follows directly from that $\tau_{l'}$, $\pi_{l'}$, and $\sigma_{l',i'}$ are continuous for all $(l', i') \in \mathsf{Idx}$. Similarly, the continuous differentiability of y_l^γ and z_l^γ follows directly from that $\tau_{l'}$, $\pi_{l'}$, and $\sigma_{l',i'}^{k'}$ are C^1 for all $(l', i') \in \mathsf{Idx}$ and $k' \in [K_{l',i'}]$. □

B.6.3 Theorems 5.10 and 5.17 (Main Lemmas)

Lemma B.54. *Let $\gamma \in \Gamma$ and $w \in \mathcal{R}^\gamma$. Suppose that for all $l \in [L]$, if τ_l does not have bias parameters, then $y_{l,i}(w) \notin pbd(\mathcal{I}_{l,i}^{\gamma(l,i)})$ for all $i \in [N_l]$. Also, assume that $D^{\mathsf{AD}}\sigma_{l,i}$ is consistent for all $(l, i) \in \mathsf{Idx}$. Then,*

$$int(\mathcal{R}^\gamma) \text{ has sufficient directions around } w.$$

Proof. First, observe that

$$\begin{aligned}
int(\mathcal{R}^\gamma) &= int\Big(\bigcap_{(l,i)\in\mathsf{Idx}} \big\{w' \in \mathbb{R}^W \mid y_{l,i}(w') \in \mathcal{I}_{l,i}^{\gamma(l,i)}\big\}\Big) \\
&= int\Big(\bigcap_{(l,i)\in\mathsf{Idx}} \big\{w' \in \mathbb{R}^W \mid y_{l,i}^\gamma(w') \in \mathcal{I}_{l,i}^{\gamma(l,i)}\big\}\Big) \\
&= \bigcap_{(l,i)\in\mathsf{Idx}} int\Big(\big\{w' \in \mathbb{R}^W \mid y_{l,i}^\gamma(w') \in \mathcal{I}_{l,i}^{\gamma(l,i)}\big\}\Big) \\
&\supseteq \bigcap_{(l,i)\in\mathsf{Idx}} A_{l,i} \qquad \text{for } A_{l,i} \triangleq \big\{w' \in \mathbb{R}^W \mid y_{l,i}^\gamma(w') \in int(\mathcal{I}_{l,i}^{\gamma(l,i)})\big\},
\end{aligned}$$

where the second line uses Lemma B.10, the third line uses that $int(U \cap V) = int(U) \cap int(V)$ for any $U, V \subseteq \mathbb{R}^n$, and the fourth line uses that $int(f^{-1}(U)) \supseteq f^{-1}(int(U))$ for any $U \subseteq \mathbb{R}^m$ and continuous $f : \mathbb{R}^n \to \mathbb{R}^m$. Note that $A_{l,i}$ is open, since $int(\mathcal{I}_{l,i}^{\gamma(l,i)})$ is open and $y_{l,i}^\gamma$ is continuous (by Lemma B.53).

Next, we show that it suffices to find some $B_{l,i} \subseteq \mathbb{R}^W$ for every $(l, i) \in \mathsf{Idx}$ such that

(i) $A_{l,i}$ has $B_{l,i}$-directions around w, and

(ii) $\bigcap_{(l,i)\in\mathsf{Idx}} B_{l,i}$ is nonempty and open.

Suppose that there are such $B_{l,i}$'s. By applying Lemma B.49-(4) to (i), we have

$$\textstyle\bigcap_{(l,i)\in\mathsf{Idx}} A_{l,i} \text{ has } \bigcap_{(l,i)\in\mathsf{Idx}} B_{l,i}\text{-directions around } w.$$

By applying Lemma B.49-(3) to the above and $\bigcap_{(l,i)\in\mathsf{Idx}} A_{l,i} \subseteq int(\mathcal{R}^\gamma)$, we have

$$int(\mathcal{R}^\gamma) \text{ has } \textstyle\bigcap_{(l,i)\in\mathsf{Idx}} B_{l,i}\text{-directions around } w.$$

By applying Lemma B.49-(5) to the above and (ii), we obtain the desired conclusion:

$$int(\mathcal{R}^\gamma) \text{ has sufficient directions around } w.$$

What remains is to show that there is $B_{l,i}$ satisfying (i) and (ii). We claim that the $B_{l,i}$ defined below satisfies (i) and (ii):

$$B_{l,i} = \begin{cases} \mathbb{R}^W & \text{if } w \in A_{l,i} \\ \{v \in \mathbb{R}^W \mid \alpha_{l,i} \cdot (Dy^\gamma_{l,i}(w) \cdot v) \in (0, \infty)\} & \text{if } w \notin A_{l,i}, \end{cases}$$

where $\alpha_{l,i} \in \{\pm 1\}$ is defined as

$$\alpha_{l,i} = \begin{cases} 1 & \text{if } w \notin A_{l,i} \text{ and } y^\gamma_{l,i}(w) = \inf \mathcal{I}^{\gamma(l,i)}_{l,i} \\ -1 & \text{if } w \notin A_{l,i} \text{ and } y^\gamma_{l,i}(w) = \sup \mathcal{I}^{\gamma(l,i)}_{l,i}. \end{cases}$$

Before proving (i) and (ii), we point out that $B_{l,i}$ is well-defined. In particular, $Dy^\gamma_{l,i}(w)$ exists since $y^\gamma_{l,i}$ is differentiable (by Lemma B.53); and $\alpha_{l,i}$ is well-defined (i.e., the cases in the definition of $\alpha_{l,i}$ covers all possible cases) since $w \notin A_{l,i}$ implies

$$y^\gamma_{l,i}(w) \in pbd(\mathcal{I}^{\gamma(l,i)}_{l,i}) = \{\inf \mathcal{I}^{\gamma(l,i)}_{l,i}, \sup \mathcal{I}^{\gamma(l,i)}_{l,i}\}. \tag{B.32}$$

Here the equality comes from that $\mathcal{I}^{\gamma(l,i)}_{l,i}$ is an interval in $\mathbb{R}$, and the inclusion comes from:

$$y^\gamma_{l,i}(w) = y_{l,i}(w), \qquad y^\gamma_{l,i}(w) \notin int(\mathcal{I}^{\gamma(l,i)}_{l,i}), \qquad y_{l,i}(w) \in \mathcal{I}^{\gamma(l,i)}_{l,i}, \tag{B.33}$$

where the first equation is by Lemma B.11 and $w \in \mathcal{R}^\gamma$, the second equation by $w \notin A_{l,i}$, and the third equation by $w \in \mathcal{R}^\gamma$.

We now prove that the $B_{l,i}$ defined above satisfies (i) and (ii).

Proof of (i). Consider $(l,i) \in \mathsf{Idx}$. If $w \in A_{l,i}$, then $A_{l,i}$ has $\mathbb{R}^W$-directions around w by Lemma B.49-(1), since $w \in int(A_{l,i}) = A_{l,i}$ (as $A_{l,i}$ is open); hence, (i) holds for this case. For the other case, suppose that $w \notin A_{l,i}$. Let $\varepsilon_{l,i} \in \mathbb{R} \cup \{\infty\}$ be the length of the interval $\mathcal{I}^{\gamma(l,i)}_{l,i}$. Then,

$$\varepsilon_{l,i} > 0, \qquad A_{l,i} = \{v \in \mathbb{R}^W \mid \alpha_{l,i} \cdot (y^\gamma_{l,i}(v) - y^\gamma_{l,i}(w)) \in (0, \varepsilon_{l,i})\}.$$

Here the former holds, since we have $int(\mathcal{I}^{\gamma(l,i)}_{l,i}) \neq \emptyset$ (by Definition B.52) and that $D^{\mathtt{AD}}\sigma_{l,i}$ is consistent (by assumption). The latter holds, since $int(\mathcal{I}^{\gamma(l,i)}_{l,i})$ is either $(y^\gamma_{l,i}(w), y^\gamma_{l,i}(w) + \varepsilon_{l,i})$ or $(y^\gamma_{l,i}(w) - \varepsilon, y^\gamma_{l,i}(w))$ by $w \notin A_{l,i}$ and Eq. (B.32). By these two observations, and since $y^\gamma_{l,i}$ is differentiable, Lemma B.49-(2) is applicable to $(A_{l,i}, B_{l,i}, w)$ and directly implies (i).

Proof of (ii). First, $\bigcap_{(l,i)\in \mathsf{Idx}} B_{l,i}$ is open as desired, since every $B_{l,i}$ is open and Idx is finite. Second, we show that $\bigcap_{(l,i)\in \mathsf{Idx}} B_{l,i}$ is nonempty. Let $\mathsf{Idx}' \triangleq \{(l,i) \in \mathsf{Idx} \mid w \notin A_{l,i}\}$. By the definition of $B_{l,i}$, what we want to show is that for some $v' \in \mathbb{R}^W$,

$$\alpha_{l,i} \cdot (Dy^{\gamma}_{l,i}(w) \cdot v') > 0 \qquad \text{for all } (l,i) \in \mathsf{Idx}'.$$

Since $\alpha_{l,i} \neq 0$ for all $(l,i) \in \mathsf{Idx}'$, it suffices to show that for some $v' \in \mathbb{R}^W$,

$$Dy^{\gamma}_{l,i}(w) \cdot v' = \alpha_{l,i} \qquad \text{for all } (l,i) \in \mathsf{Idx}'. \tag{B.34}$$

To prove this, we analyze the above equation as follows. Consider any $(l,i) \in \mathsf{Idx}'$. Then, we have $w \notin A_{l,i}$, which implies $y_{l,i}(w) \in pbd(\mathcal{I}^{\gamma(l,i)}_{l,i})$ by Eqs. (B.32) and (B.33). From this, τ_l has bias parameters (by assumption). So, for all $v = (v_1, \ldots, v_W) \in \mathbb{R}^W$,

$$\begin{aligned} y^{\gamma}_{l,i}(v) &= \tau_{l,i}\big(z^{\gamma}_{l-1}(v), \pi_l(v)\big) \\ &= \tau_{l,i}\big(z^{\gamma}_{l-1}(v_1, \ldots, v_{W'}, 0, \ldots, 0), (v_{W'+1}, \ldots, v_{W'+W_l})\big) \\ &= \tau'_{l,i}\big(z^{\gamma}_{l-1}(v_1, \ldots, v_{W'}, 0, \ldots, 0), (v_{W'+1}, \ldots, v_{W'+(W_l-N_l)})\big) + v_{W'+(W_l-N_l+i)}, \end{aligned} \tag{B.35}$$

where the second line uses $W' \triangleq W_1 + \cdots + W_{l-1}$ and the fact that z^{γ}_{l-1} depends only on the parameters of $\tau_1, \ldots, \tau_{l-1}$, and the third line uses that τ_l has bias parameters. Let $\psi_{l,i} \triangleq W' + (W_l - N_l + i)$. Since the first term in Eq. (B.35) does not depend on $v_{\psi_{l,i}}, \ldots, v_W$, the following holds for all $j \geq \psi_{l,i}$:

$$\big(Dy^{\gamma}_{l,i}(w)\big)_j = \begin{cases} 1 & \text{if } j = \psi_{l,i} \\ 0 & \text{if } j > \psi_{l,i}. \end{cases}$$

From this, the following holds for all $v \in \mathbb{R}^W$:

$$Dy^{\gamma}_{l,i}(w) \cdot v = \sum_{j\in[W]} \big(Dy^{\gamma}_{l,i}(w)\big)_j \cdot v_j = f_{l,i}(v_1, \ldots, v_{\psi_{l,i}-1}) + v_{\psi_{l,i}},$$

where $f_{l,i} : \mathbb{R}^{\psi_{l,i}-1} \to \mathbb{R}$ is defined as $f_{l,i}(u) \triangleq \sum_{j\in[\psi_{l,i}-1]} \big(Dy^{\gamma}_{l,i}(w)\big)_j \cdot u_j$. Hence, what we planned to show (i.e., Eq. (B.34) holds for some $v' \in \mathbb{R}^W$) is equivalent to the following: for some $v' \in \mathbb{R}^W$,

$$f_{l,i}(v'_1, \ldots, v'_{\psi_{l,i}-1}) + v'_{\psi_{l,i}} = \alpha_{l,i} \qquad \text{for all } (l,i) \in \mathsf{Idx}'. \tag{B.36}$$

Since $\psi_{l,i} \neq \psi_{l',i'}$ for any $(l,i) \neq (l',i')$, Lemma B.51 implies that there is $v' \in \mathbb{R}^W$ satisfying Eq. (B.36). This proves (ii), and concludes the proof. □

Lemma B.55. *Let $\gamma \in \Gamma$ and $w \in \mathcal{R}^{\gamma}$. Suppose that $int(\mathcal{R}^{\gamma})$ has sufficient directions around w.*

Then,

$$D^{\mathtt{AD}} z_L(w) = \begin{cases} Dz_L(w) & \text{if } Dz_L(w) \neq \bot \\ \lim_{n\to\infty} Dz_L(w'_n) \text{ for some } w'_n \to w & \text{if } Dz_L(w) = \bot. \end{cases}$$

Proof. Let $\gamma \in \Gamma$ and $w \in \mathcal{R}^\gamma$ such that $int(\mathcal{R}^\gamma)$ has sufficient directions around w. By Lemmas B.11 and B.12,

$$z_L(w') = z_L^\gamma(w') \quad \wedge \quad D^{\mathtt{AD}} z_L(w') = Dz_L^\gamma(w') \qquad \text{for all } w' \in \mathcal{R}^\gamma. \tag{B.37}$$

We prove the conclusion for each of the two cases: $Dz_L(w) \neq \bot$ and $Dz_L(w) = \bot$.

Case 1: $Dz_L(w) \neq \bot$ (i.e., z_L is differentiable at w). We want to show

$$D^{\mathtt{AD}} z_L(w) = Dz_L(w).$$

This holds as follows:

$$D^{\mathtt{AD}} z_L(w) = Dz_L^\gamma(w) = Dz_L(w),$$

where the first equality is by Eq. (B.37), and the second equality follows directly from Lemma B.50 applied to $(z_L^\gamma, z_L, \mathcal{R}^\gamma, w)$. Here Lemma B.50 is applicable since its preconditions are satisfied: z_L^γ is differentiable at w (by Lemma B.53); z_L is differentiable at w (by assumption); $z_L^\gamma = z_L$ on $int(\mathcal{R}^\gamma) \cup \{w\}$ (by Eq. (B.37)); and $int(\mathcal{R}^\gamma)$ has sufficient directions around w (by assumption).

Case 2: $Dz_L(w) = \bot$ (i.e., z_L is not differentiable at w). We want to show:

$$D^{\mathtt{AD}} z_L(w) = \lim_{n\to\infty} Dz_L(w'_n) \qquad \text{for some } w'_n \to w. \tag{B.38}$$

Since $int(\mathcal{R}^\gamma)$ has sufficient directions around w (by assumption), there is $\{w'_n \in int(\mathcal{R}^\gamma)\}_{n\in\mathbb{N}}$ such that $w'_n \to w$. We show that these w'_n satisfy Eq. (B.38) as follows:

$$D^{\mathtt{AD}} z_L(w) = Dz_L^\gamma(w) = \lim_{n\to\infty} Dz_L^\gamma(w'_n) = \lim_{n\to\infty} Dz_L(w'_n),$$

where the first equality is by Eq. (B.37), the second equality uses that Dz_L^γ is continuous (by Lemma B.53), and the third equality uses that $Dz_L^\gamma(w'_n) = Dz_L(w'_n)$ for all n (since $w'_n \in int(\mathcal{R}^\gamma)$ and $z_L^\gamma = z_L$ on $\mathcal{R}^\gamma$ by Eq. (B.37)). This concludes the proof. □

B.6.4 Theorems 5.10 and 5.17 (Main Proofs)

Theorem 5.10. *If z_L has bias parameters and $D^{\mathtt{AD}}\sigma_{l,i}$ is consistent for all $(l,i) \in \mathsf{Idx}$, then for all*

$w \in \mathbb{R}^W$,

$$D^{\mathsf{AD}} z_L(w) = \begin{cases} Dz_L(w) & \text{if } Dz_L(w) \neq \bot \\ \lim_{n\to\infty} Dz_L(w'_n) \text{ for some } w'_n \to w & \text{if } Dz_L(w) = \bot. \end{cases}$$

This implies that $D^{\mathsf{AD}} z_L$ is a Clarke subderivative of z_L.

Proof. This theorem is a special case of Theorem 5.17; we omit the proof. □

Theorem 5.17. *Let $w \in \mathbb{R}^W$ and assume that $D^{\mathsf{AD}}\sigma_{l,i}$ is consistent for all $(l,i) \in \mathsf{Idx}$. If $y_{l,i}(w) \notin \mathsf{ncdf}(\sigma_{l,i})$ for all $(l,i) \in \mathsf{Idx}$ such that τ_l does not have bias parameters, then*

$$D^{\mathsf{AD}} z_L(w) = \begin{cases} Dz_L(w) & \text{if } Dz_L(w) \neq \bot \\ \lim_{n\to\infty} Dz_L(w'_n) & \\ \quad \text{for some } w'_n \to w & \text{if } Dz_L(w) = \bot \end{cases}$$

and so $D^{\mathsf{AD}} z_L(w)$ is a Clarke subderivative of z_L at w.

Proof. Let $w \in \mathbb{R}^W$ that satisfies the assumption in the statement. By Lemma B.8, there is $\gamma \in \Gamma$ such that $w \in \mathcal{R}^\gamma$. Note that Lemma B.54 is applicable to (γ, w) because: $D^{\mathsf{AD}}\sigma_{l,i}$ is consistent for all $(l,i) \in \mathsf{Idx}$ (by assumption); and for all $l \in [L]$, if τ_l does not have bias parameters, then $y_{l,i}(w) \notin \mathsf{ncdf}(\sigma_{l,i})$ and so $y_{l,i}(w) \notin pbd(\mathcal{I}_{l,i}^{\gamma(l,i)})$ for all $i \in [N_l]$, where the former follows from the assumption and the latter from $pbd(\mathcal{I}_{l,i}^{\gamma(l,i)}) \subseteq \mathsf{ncdf}(\sigma_{l,i})$ (by Definition B.52). Hence, Lemma B.54 implies that $int(\mathcal{R}^\gamma)$ has sufficient directions around w, which subsequently implies the conclusion by Lemma B.55. □

Appendix C

Appendix for Chapter 6

C.1 Problem: Deferred Proof

Theorem 6.2. *Problem 6.1 is NP-hard.*

Proof. We prove the NP-hardness of Problem 6.1 (the memory-accuracy tradeoff problem) by reducing the knapsack problem (which is NP-hard) to the tradeoff problem. More precisely, we prove that the knapsack problem can be solved in polynomial time if we assume an oracle for the tradeoff problem.

Recall the knapsack problem: given n items with weights $w_i \in \mathbb{N}$ and profits $p_i \in \mathbb{N}$ ($i \in [n]$), and given a threshold $W \in \mathbb{N}$, decide which items to choose such that the total profit of the chosen items is maximized while their total weight does not exceed W. That is, find $\alpha \in \{0,1\}^n$ that maximizes $\sum_{i\in[n]} \alpha_i p_i$ subject to $\sum_{i\in[n]} \alpha_i w_i \le W$. This problem is well-known to be NP-hard [79].

Given an instance of the knapsack problem (w, p, W), we construct an instance of the tradeoff problem as follows.

- **Notations.** The following construct uses a constant $k \in \mathbb{N}$ and floating-point formats $\mathsf{fp}_{\mathsf{hi}}, \mathsf{fp}_{\mathsf{lo}} \in \mathsf{FP}$ (one for high precision and the other for low precision). Below we will specify the conditions they should satisfy, and show that some k, $\mathsf{fp}_{\mathsf{hi}}$, and $\mathsf{fp}_{\mathsf{lo}}$ indeed satisfy the conditions. We write $\mathrm{rnd}_{\mathsf{hi}}(\cdot)$ and $\mathrm{rnd}_{\mathsf{lo}}(\cdot)$ as shorthand for $\mathrm{rnd}_{\mathsf{fp}_{\mathsf{hi}}}(\cdot)$ and $\mathrm{rnd}_{\mathsf{fp}_{\mathsf{lo}}}(\cdot)$.

- **Training setups.** We consider a very simple setting for training: the gradient descent algorithm with a learning rate $\eta = 2^{-l}$ ($l \in \mathbb{N}$) is applied for just one epoch; all parameters are initialized to 0 and their master copies are represented in $\mathsf{fp}_{\mathsf{hi}}$; and the negative loss of a model on training data (i.e., $-L(f_\theta(x), y)$ using notations to be described below) is used as the accuracy of the model. Here $l \in \mathbb{N}$ can be any natural number.

- **Model and loss networks.** A model network $\mathcal{M}$ and a loss network $\mathcal{L}$ are given as Figure C.1, where $\mathcal{M}$ has n parameter tensors $\theta_i \in \mathbb{R}^{w_i}$ of size w_i ($i \in [n]$). For an input-output pair

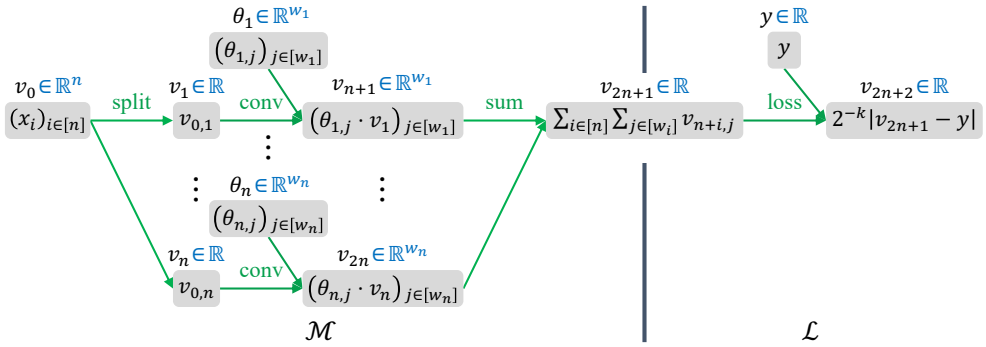


Figure C.1: The model network $\mathcal{M}$ and the loss network $\mathcal{L}$ used in the proof of Theorem 6.2.

$(x, y) \in \mathbb{R}^n \times \mathbb{R}$, $\mathcal{M}$ and $\mathcal{L}$ compute a predicted output $f_\theta(x) \in \mathbb{R}$ and a loss $L(f_\theta(x), y) \in \mathbb{R}$ as follows (assuming that no rounding functions are applied):

$$f_\theta(x) \triangleq \sum_{i\in[n]} \sum_{j\in[w_i]} \theta_{i,j} x_i, \qquad L(f_\theta(x), y) \triangleq 2^{-k}|f_\theta(x) - y|.$$

Roughly speaking, $\mathcal{M}$ is (a variant of) a linear classifier and $\mathcal{L}$ is a ℓ_1-loss (scaled by 2^{-k}).

- **Training data.** Training data consists of a single input-output pair $(x, y) \in \mathbb{R}^n \times \mathbb{R}$ that satisfies the following:

$$x_i = \mathrm{rnd}_{\mathsf{lo}}(\sqrt{p_i/w_i}), \qquad y < -2^{-(k+l)} \sum_{i\in[n]} w_i x_i^2$$

for all $i \in [n]$. Here y can take any value as long as it satisfies the above inequality. Note that x_i can be different from $\sqrt{p_i/w_i}$ since the latter value may not be representable in $\mathsf{fp}_{\mathsf{lo}}$.

- **Precision-candidate assignment.** A precision-candidate assignment $\mathcal{C} : \mathsf{TS} \times \{\mathsf{hi}, \mathsf{lo}\} \to \mathsf{FP}$ is given as:

$$\mathcal{C}(t, \mathsf{hi}) \triangleq \mathsf{fp}_{\mathsf{hi}}, \quad \mathcal{C}(t, \mathsf{lo}) \triangleq \mathsf{fp}_{\mathsf{lo}} \quad \text{for all } t \in \mathsf{TS}.$$

That is, for all tensors, $\mathsf{fp}_{\mathsf{hi}}$ is used as a high-precision format and $\mathsf{fp}_{\mathsf{hi}}$ as a low-precision format. Here $\mathsf{fp}_{\mathsf{hi}}$ and $\mathsf{fp}_{\mathsf{lo}}$ should satisfy the following:

$$e_{\mathsf{hi}} \geq e_{\mathsf{lo}}, \quad m_{\mathsf{hi}} \geq m_{\mathsf{lo}}, \tag{C.1}$$

$$|\mathrm{rnd}_{\mathsf{lo}}(s) - s| < |s| \cdot err \quad \text{for all } s \in S_1, \tag{C.2}$$

$$\mathrm{rnd}_{\mathsf{lo}}(s) = 0 \quad \text{for all } s \in S_2, \tag{C.3}$$

$$\mathrm{rnd}_{\mathsf{hi}}(s) = s \quad \text{for all } s \in S_2 \cup S_3. \tag{C.4}$$

Here e_{hi} and m_{hi} (and e_{lo} and m_{lo}) denote the number of exponent bits and mantissa bits of $\mathsf{fp}_{\mathsf{hi}}$

(and $\mathsf{fp}_{\mathsf{lo}}$), and err and S_j are defined as: $err \triangleq 1/(6n \cdot \max_{i\in[n]} p_i)$, $S_1 \triangleq \{\sqrt{p_i/w_i} \mid i \in [n]\}$, $S_2 \triangleq \{2^{-k}\} \cup \{2^{-k} x_i \mid i \in [n]\}$, and $S_3 \triangleq \{2^{-(k+l)} x_i \mid i \in [n]\}$. Eq. (C.2) says that the relative error of representing each $s \in S_1$ in $\mathsf{fp}_{\mathsf{lo}}$ should be less than err. Eq. (C.3) says that each $s \in S_2$ should underflow to 0 when represented in $\mathsf{fp}_{\mathsf{lo}}$. Eq. (C.4) says that each $s \in S_2 \cup S_3$ should be representable in $\mathsf{fp}_{\mathsf{hi}}$.

- **Low-precision ratio.** A lower bound $r \in [0, 1]$ on the low-precision ratio is given as:

$$r \triangleq \max\left\{0, 1 - \frac{2W+1}{\mathrm{size}(\mathsf{TS})}\right\} \in [0, 1].$$

  So r decreases linearly as W increases.

We make three points on the above construction.

- First, each part of the knapsack problem (w, p, W) is used in the following parts of the construction: w_i is used mainly in the size of the parameter tensor θ_i; p_i in the input x_i; and W in the lower bound r.

- Second, there exist $k \in \mathbb{N}$ and $\mathsf{fp}_{\mathsf{hi}}, \mathsf{fp}_{\mathsf{lo}} \in \mathsf{FP}$ that satisfy Eqs. (C.1)–(C.4). This can be shown as follows: first, by taking sufficiently many exponent and mantissa bits for $\mathsf{fp}_{\mathsf{lo}}$, we can make Eq. (C.2) satisfied; next, by taking a sufficiently large k, we can make Eq. (C.3) satisfied; finally, by taking sufficiently many exponent and mantissa bits for $\mathsf{fp}_{\mathsf{hi}}$, we can make Eq. (C.1) and Eq. (C.4) satisfied (since x_i is representable in $\mathsf{fp}_{\mathsf{lo}}$ and $2^{-(k+l)}$ is a power of two).

- Third, some well-known models (e.g., ShuffleNet-v2) have a similar structure to $\mathcal{M}$ in that they apply the following operations as a subroutine: split a tensor into multiple tensors, apply some operators to each split tensor, and combine the resulting tensors into a single tensor.

We now prove that the knapsack problem (w, p, W) can be solved in polynomial time, if an oracle to the above tradeoff problem is given. Suppose that $\pi \in \Pi(\mathcal{C})$ is an optimal solution to the above tradeoff problem (given by the oracle). Define an item selection $\alpha \in \{0, 1\}^n$ for the knapsack problem as:

$$\alpha_i \triangleq \begin{cases} 1 & \text{if } \pi(d\theta_i) = \pi(dv_{n+i}) = \pi(dv_{2n+1}) = \mathsf{fp}_{\mathsf{hi}} \\ 0 & \text{otherwise} \end{cases}$$

for each $i \in [n]$. Note that α can be constructed from π in linear time. Thus, it suffices to show that α is an optimal solution to the knapsack problem (w, p, W), which is equivalent to the following two claims:

- Claim 1: We have $\sum_{i\in[n]} \alpha_i w_i \le W$.

- Claim 2: For any $\alpha' \in \{0, 1\}^n$ with $\sum_{i\in[n]} \alpha'_i w_i \le W$, we have $\sum_{i\in[n]} \alpha'_i p_i \le \sum_{i\in[n]} \alpha_i p_i$.

We now prove each claim as follows.

Proof of Claim 1. If $\alpha = (0, \dots, 0)$, then the claim clearly holds. Suppose that $\alpha \neq (0, \dots, 0)$. Then,

$$1 - \frac{1 + 2\sum_{i\in[n]} \alpha_i w_i}{\mathrm{size}(\mathsf{TS})} \geq \mathrm{ratio}_{\mathsf{lo}}(\pi) \geq r \geq 1 - \frac{1 + 2W}{\mathrm{size}(\mathsf{TS})}.$$

Here the first inequality uses $\alpha \neq (0, \dots, 0)$ and the definition of α and $\mathcal{M}$; the second inequality uses the fact that π is a valid solution to the above tradeoff problem; and the third inequality uses the definition of r. Hence, the claim holds.

Proof of Claim 2. Suppose that the claim does not hold. Then, there exists $\alpha' \in \{0, 1\}^n$ such that

$$\sum_{i\in[n]} \alpha'_i w_i \leq W, \qquad \sum_{i\in[n]} \alpha'_i p_i > \sum_{i\in[n]} \alpha_i p_i.$$

Define a precision assignment $\pi' \in \Pi(\mathcal{C})$ as:

$$\begin{aligned} \pi'(dv_{2n+1}) &\triangleq \mathsf{fp}_{\mathsf{hi}}, \\ \pi'(d\theta_i) \triangleq \pi'(dv_{n+i}) &\triangleq \mathsf{fp}_{\mathsf{hi}} \quad \text{for all } i \in [n] \text{ with } \alpha'_i = 1, \\ \pi'(t) &\triangleq \mathsf{fp}_{\mathsf{lo}} \quad \text{for all other } t \in \mathsf{TS}. \end{aligned}$$

Then, we have $\mathrm{ratio}_{\mathsf{lo}}(\pi') \geq r$ by $\sum_{i\in[n]} \alpha'_i w_i \leq W$ and the definition of π', $\mathcal{M}$, and r. Hence, it suffices to show $\mathrm{acc}(\pi) < \mathrm{acc}(\pi')$, because this would contradict to the fact that π is an optimal solution.

To show $\mathrm{acc}(\pi) < \mathrm{acc}(\pi')$, we prove the following two lemmas: the first lemma gives a closed form of $\mathrm{acc}(\pi)$ and $\mathrm{acc}(\pi')$, and the second lemma shows that $\sum_{i\in[n]} \beta_i w_i x_i^2$ is close to $\sum_{i\in[n]} \beta_i p_i$ (where the former summation appears in $\mathrm{acc}(\pi)$ and $\mathrm{acc}(\pi')$).

Lemma C.1. *The following hold:*

$$\mathrm{acc}(\pi) = 2^{-k} y + 2^{-(2k+l)} \sum_{i\in[n]} \alpha_i w_i x_i^2, \qquad \mathrm{acc}(\pi') = 2^{-k} y + 2^{-(2k+l)} \sum_{i\in[n]} \alpha'_i w_i x_i^2.$$

Proof. We prove the equation for $\mathrm{acc}(\pi)$ only, since the equation for $\mathrm{acc}(\pi')$ can be proved similarly.

First, we show that for all $i \in [n]$ and $j \in [w_i]$,

$$\hat{d\theta}_{i,j} = \alpha_i \cdot 2^{-k} x_i. \tag{C.5}$$

Pick any $i \in [n]$ and $j \in [w_i]$. Note that by the definition of $\mathcal{M}$, we have

$$\begin{aligned} \hat{d\theta}_{i,j} &= \mathrm{rnd}_{\pi(d\theta_i)}\Big(\mathrm{rnd}_{\pi(dv_{n+i})}(\mathrm{rnd}_{\pi(dv_{2n+1})}(2^{-k})) \cdot \mathrm{rnd}_{v_i}(\mathrm{rnd}_{v_0}(x_i))\Big) \\ &= \mathrm{rnd}_{\pi(d\theta_i)}\Big(\mathrm{rnd}_{\pi(dv_{n+i})}(\mathrm{rnd}_{\pi(dv_{2n+1})}(2^{-k})) \cdot x_i\Big), \end{aligned}$$

where the second equality uses Eq. (C.1) and that x_i is representable in $\mathsf{fp}_{\mathsf{lo}}$. We prove Eq. (C.5) by case analysis on α_i. Suppose $\alpha_i = 1$. Then, by the definition of α_i, $\pi(d\theta_i) = \pi(dv_{n+i}) = \pi(dv_{2n+1}) = \mathsf{fp}_{\mathsf{hi}}$. From this, we get the desired equation:

$$\hat{d\theta}_{i,j} = \mathrm{rnd}_{\mathsf{hi}}\Big(\mathrm{rnd}_{\mathsf{hi}}(\mathrm{rnd}_{\mathsf{hi}}(2^{-k})) \cdot x_i\Big) = \mathrm{rnd}_{\mathsf{hi}}(2^{-k} \cdot x_i) = 2^{-k} x_i,$$

where the last two equalities use Eq. (C.4). Suppose now $\alpha_i = 0$. Then, by the definition of α_i, at least one of $\pi(d\theta_i)$, $\pi(dv_{n+i})$, and $\pi(dv_{2n+1})$ is $\mathsf{fp}_{\mathsf{lo}}$. If $\pi(dv_{n+i}) = \mathsf{fp}_{\mathsf{lo}}$ or $\pi(dv_{2n+1}) = \mathsf{fp}_{\mathsf{lo}}$, we get the desired equation:

$$\hat{d\theta}_{i,j} = \mathrm{rnd}_{\pi(d\theta_i)}\Big(\mathrm{rnd}_{\mathsf{lo}}(2^{-k}) \cdot x_i\Big) = \mathrm{rnd}_{\pi(d\theta_i)}(0 \cdot x_i) = 0,$$

where the first equality uses Eq. (C.1) and Eq. (C.4), and the second equality uses Eq. (C.3). The remaining case is when $\pi(dv_{n+i}) = \pi(dv_{2n+1}) = \mathsf{fp}_{\mathsf{hi}}$ and $\pi(d\theta_i) = \mathsf{fp}_{\mathsf{lo}}$. We get the desired equation in this case as well:

$$\hat{d\theta}_{i,j} = \mathrm{rnd}_{\mathsf{lo}}\Big(\mathrm{rnd}_{\mathsf{hi}}(\mathrm{rnd}_{\mathsf{hi}}(2^{-k})) \cdot x_i\Big) = \mathrm{rnd}_{\mathsf{lo}}(2^{-k} \cdot x_i) = 0,$$

where the second equality uses Eq. (C.4), and the last equality uses Eq. (C.3). Hence, we have proved Eq. (C.5).

Next, let θ_i be the i-th parameter tensor before training starts, and θ'_i be the corresponding tensor after training ends ($i \in [n]$). Then, by the definition of the tradeoff problem constructed above, we have $\theta_{i,j} = 0$ and

$$\theta'_{i,j} = \theta_{i,j} - \mathrm{rnd}_{\mathsf{hi}}(2^{-l} \cdot \hat{d\theta}_{i,j}) = 0 - \mathrm{rnd}_{\mathsf{hi}}(2^{-l} \cdot (\alpha_i \cdot 2^{-k} x_i)) = \alpha_i \cdot (-2^{-(k+l)} x_i),$$

where the second equality uses Eq. (C.5) and the third equality uses Eq. (C.4). Using this equation, we finally obtain the conclusion of this lemma:

$$\begin{aligned}
\mathrm{acc}(\pi) &= -L(f_{\theta'}(x), y) \\
&= -2^{-k}\Big|y - \sum_{i\in[n]} \sum_{j\in[w_i]} \theta'_{i,j} x_i\Big| \\
&= -2^{-k}\Big|y - \sum_{i\in[n]} \sum_{j\in[w_i]} \alpha_i \cdot (-2^{-(k+l)} x_i) \cdot x_i\Big| \\
&= -2^{-k}\Big|y + \sum_{i\in[n]} \alpha_i \cdot 2^{-(k+l)} w_i x_i^2\Big| \\
&= 2^{-k}\Big(y + \sum_{i\in[n]} \alpha_i \cdot 2^{-(k+l)} w_i x_i^2\Big)
\end{aligned}$$

$$= 2^{-k}y + 2^{-(2k+l)} \sum_{i\in[n]} \alpha_i w_i x_i^2,$$

where the first two equalities use the definition of accuracy, and the second last equality uses the definition of y. This concludes the proof of the lemma. ■

Lemma C.2. *For any $\beta \in \{0,1\}^n$,*

$$\Big| \sum_{i\in[n]} \beta_i w_i x_i^2 - \sum_{i\in[n]} \beta_i p_i \Big| < \frac{1}{2}.$$

Proof. We first show that for any $i \in [n]$,

$$|w_i x_i^2 - p_i| < \frac{1}{2n}.$$

Pick any $i \in [n]$. By Eq. (C.2) and the definition of x_i, we have

$$\Big| x_i - \sqrt{\frac{p_i}{w_i}} \Big| < \sqrt{\frac{p_i}{w_i}} \cdot \frac{1}{6n \cdot \max_{j\in[n]} p_j} \le \sqrt{\frac{p_i}{w_i}} \cdot \frac{1}{6np_i}.$$

From this, we have

$$\sqrt{\frac{p_i}{w_i}}\Big(1 - \frac{1}{6np_i}\Big) < x_i < \sqrt{\frac{p_i}{w_i}}\Big(1 + \frac{1}{6np_i}\Big), \qquad \frac{p_i}{w_i}\Big(1 - \frac{1}{6np_i}\Big)^2 < x_i^2 < \frac{p_i}{w_i}\Big(1 + \frac{1}{6np_i}\Big)^2.$$

From this, we obtain the desired result:

$$|w_i x_i^2 - p_i| < p_i\Big(\Big(1 + \frac{1}{6np_i}\Big)^2 - 1\Big) = p_i\Big(\frac{1}{3np_i} + \frac{1}{(6np_i)^2}\Big) < p_i\Big(\frac{1}{3np_i} + \frac{1}{6np_i}\Big) = p_i \cdot \frac{1}{2np_i} = \frac{1}{2n},$$

where the second inequality uses $6np_i > 1$ (as $n, p_i \in \mathbb{N}$).

Using this result, we can show the conclusion as follows:

$$\Big| \sum_{i\in[n]} \beta_i w_i x_i^2 - \sum_{i\in[n]} \beta_i p_i \Big| = \Big| \sum_{i\in[n]} \beta_i (w_i x_i^2 - p_i) \Big| \le \sum_{i\in[n]} |\beta_i| \cdot |w_i x_i^2 - p_i| < \sum_{i\in[n]} \frac{1}{2n} = \frac{1}{2},$$

where the last inequality uses $|\beta_i| \le 1$. This completes the proof of the lemma. ■

Using the two lemmas, we now prove $\mathrm{acc}(\pi) < \mathrm{acc}(\pi')$. By Lemma C.2 and $\sum_{i\in[n]} \alpha_i p_i < \sum_{i\in[n]} \alpha'_i p_i$, we have

$$\sum_{i\in[n]} \alpha_i w_i x_i^2 < \sum_{i\in[n]} \alpha_i p_i + \frac{1}{2} \le \sum_{i\in[n]} \alpha'_i p_i - \frac{1}{2} < \sum_{i\in[n]} \alpha'_i w_i x_i^2,$$

where the second inequality comes from $\alpha_i, \alpha'_i \in \{0,1\}$ and $p_i \in \mathbb{N}$. From this, and by Lemma C.1,

we obtain $\mathrm{acc}(\pi) < \mathrm{acc}(\pi')$ as desired. This concludes the proof of Claim 2, thereby finishing the proof of the theorem. □

Remark C.3. In the proof of Theorem 6.2, we proved the NP-hardness of Problem 6.1 by making use of only a few limited aspects of the problem. For instance, we used the fact that some values representable in a high-precision format round to *zero* in a low-precision format; on the other hand, many other values representable in a high-precision format round to *non-zero* values in a low-precision format, and this indeed occurs in practical training (even more frequently than underflows). Also, we used a simple setting for training in which a gradient descent algorithm is applied for *one epoch*, training data consist of *one input-output pair*, and test data is *the same as* training data; on the other hand, in practical training, a gradient descent algorithm is applied for *many epochs*, training data consists of *many input-output pairs*, and test data is *different from* training data.

Problem 6.1 is general enough so that it embraces all the aforementioned aspects of floating-points and training, including those that are not considered in the proof of Theorem 6.2. Since those aspects are likely to make the problem even more difficult, we conjecture that the problem would be more intractable than being NP-hard. □

C.2 Experiments: Deferred Details

The datasets we use have the following licenses:

- CIFAR-10 and CIFAR-100: These datasets are under the MIT license.
- ImageNet: This dataset can be used "only for non-commercial research and educational purposes." For more details, see its webpage [145].

The implementations of models we use have the following licenses:

- SqueezeNet for CIFAR-10 and CIFAR-100: We adapt an implementation of the model in a public GitHub repository [119], whose license information is not available.
- ShuffleNet-v2, MobileNet-v2, and ResNet-18 for CIFAR-10 and CIFAR-100: We adapt an implementation of these models in a public GitHub repository [86], which is under the MIT license.
- ShuffleNet-v2 for ImageNet and ImageNet-200-i: We adapt an implementation of the model in the torchvision library [122], which is under the BSD 3-Clause license.

The details of how we train models are as follows:

- Four models on CIFAR-10 and CIFAR-100: We train the four models with a standard setup [86]. In particular, we run the (non-Nesterov) SGD optimizer for 200 epochs with minibatch size of 128 (over 1 GPU), learning rate of 0.1, momentum of 0.9, weight decay of 5×10^{-4}, and the cosine

annealing scheduler for learning rate. For dynamic loss scaling, we use initial scale of 2^{16}, growth factor of 2, back-off factor of 0.5, and growth interval of 1 epoch, as suggested in PyTorch [121].

- ShuffleNet-v2 on ImageNet: We train the model with the default setup given in PyTorch's GitHub repository [123], except that we use larger minibatch size and learning rate as in [55, 78, 85, 124] to reduce the wall-clock time of training. In particular, we run the (non-Nesterov) SGD optimizer for 90 epochs with minibatch size of 1024 (over 16 GPUs), learning rate of 0.4, momentum of 0.9, weight decay of 10^{-4}, and the cosine annealing scheduler for learning rate. For dynamic loss scale, we use initial scale of 2^{16}, growth factor of 2, back-off factor of 0.5, and growth interval of 0.5 epoch, as suggested in PyTorch [121].

- ShuffleNet-v2 on ImageNet-200-i: We train the model with the same settings for ImageNet except that we use the default values for minibatch size and learning rate given in [123], i.e., minibatch size of 256 (over 4 GPUs) and learning rate of 0.1.

C.3 Experiments: Deferred Results

C.3.1 Comparison with Existing Precision Assignments

Figure C.2 presents a zoomed-in version of Figure 6.3 (left).

Figure C.3 presents results omitted in Figure 6.4: training results of smaller variant models (which have width multiplier 0.5 or 0.1) on CIFAR-100 with π_{fp32}, π_{unif}, π_{op}, $\pi_{\text{op}'}$, and $\pi_{\text{ours},r}$. The figure shows similar results to Figure 6.4: the results for the variant models with width multiplier 0.5 (and 0.1) are similar to those for the original models (and the variant models with width multiplier 0.25).

Figures C.4 and C.5 show the average training trajectories for the configurations presented in Figures 6.4 and C.3.

Figures C.6 and C.7 present the same results as Figures 6.3 and 6.4 except the following: in the former, π_{op} and $\pi_{\text{op}'}$ are equipped with our precision promotion technique, whereas in the latter they do not so. Figures C.6 and C.7 do not include π_{unif} because this assignment with the precision promotion is identical to $\pi_{\text{ours},1}$.

C.3.2 Ablation Study: Precision Demotion and Promotion

Figure C.9 presents results omitted in Figure 6.5: training results of ResNet-18 on CIFAR-100 with $\pi_{\text{ours},r}$, $\pi_{\text{ours[inc]},r}$, and $\pi_{\text{ours[rand]},r}$. The figure shows similar results to Figure 6.5 except that it shows smaller differences in memory-accuracy tradeoff between the three precision assignments.

Figure C.10 presents results omitted in Figure 6.6: training results of four models on CIFAR-10 with $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. The figure shows similar results to Figure 6.6 except that the training of ResNet-18 on CIFAR-10 does not diverge even with $\pi_{\text{ours[no-promo]},r}$ for all r values.

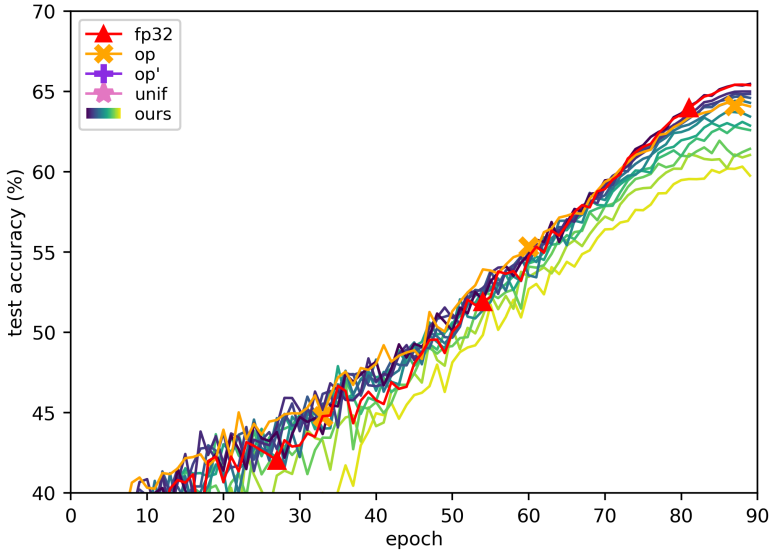
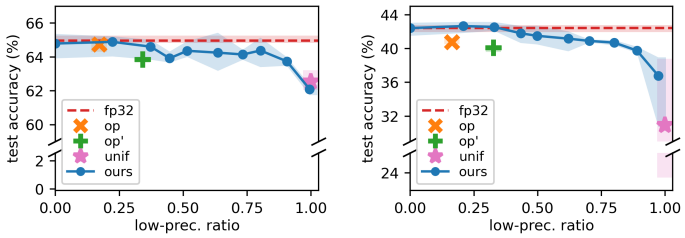
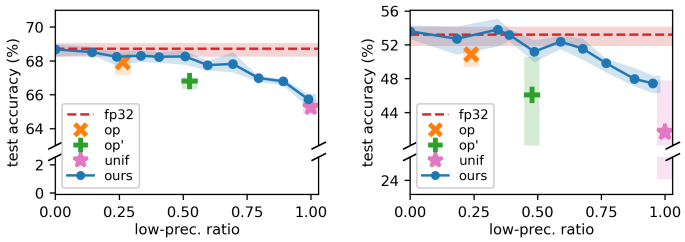


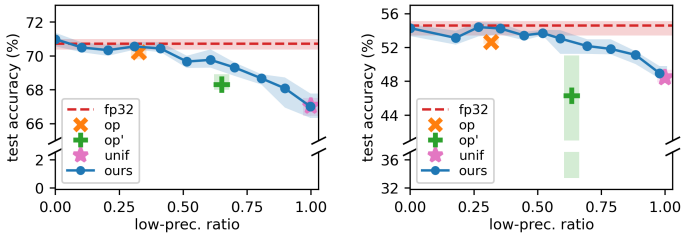
Figure C.2: A zoomed-in version of Figure 6.3 (left). Results of training ShuffleNet-v2 on ImageNet with π_{fp32}, π_{unif} [104], π_{op} [147], $\pi_{\text{op}'}$ [78], and $\pi_{\text{ours},r}$. Each line shows the average training trajectory for each precision assignment; $\pi_{\text{ours},r}$ is colored from navy to yellow (darker for smaller r).



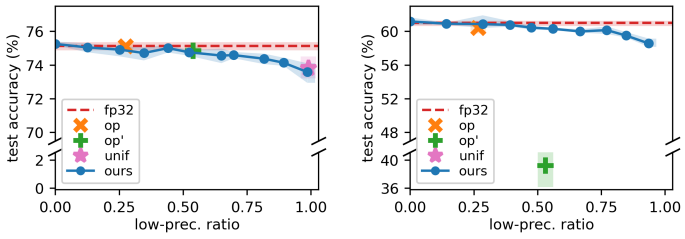
(a) CIFAR-100, SqueezeNet‡ (b) CIFAR-100, SqueezeNet¶



(c) CIFAR-100, ShuffleNet-v2‡ (d) CIFAR-100, ShuffleNet-v2¶

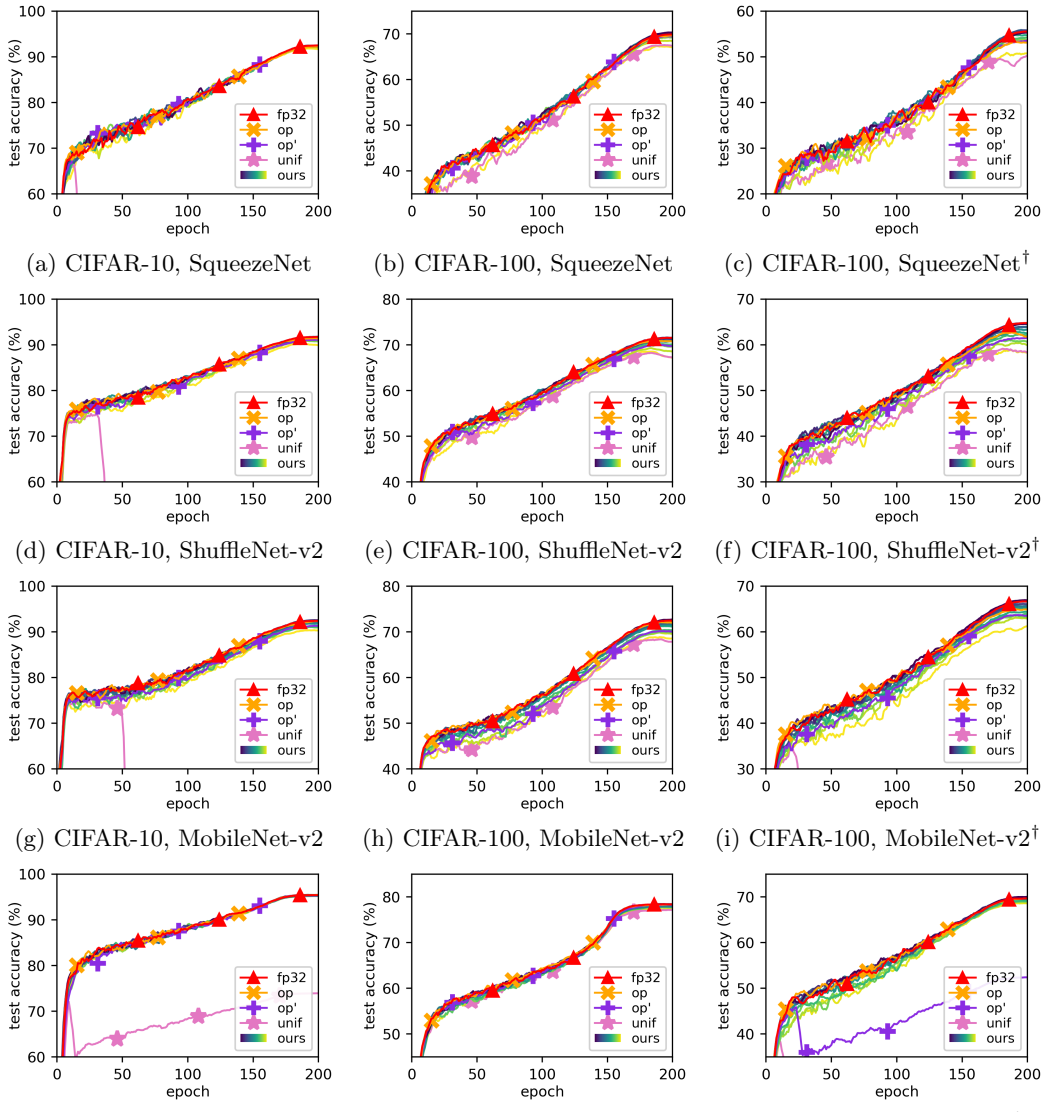


(e) CIFAR-100, MobileNet-v2‡ (f) CIFAR-100, MobileNet-v2¶



(g) CIFAR-100, ResNet-18‡ (h) CIFAR-100, ResNet-18¶

Figure C.3: Results continued from Figure 6.4. Memory-accuracy tradeoffs of π_{unif} [104], π_{op} [147], $\pi_{\text{op}'}$ [78], and $\pi_{\text{ours},r}$ for smaller variants of four models on CIFAR-100. The variant models have width multiplier 0.5 (marked by ‡) or 0.1 (marked by ¶). Top-right points are better than bottom-left ones. In all but one plots, there are ●s above and to the right of ✕ and +, respectively; even in the one plot (g), ●s have almost the same tradeoffs to ✕ and +. In three of all plots, ★ has much smaller y-values than other points; ★ is missing in (h) as its y-value is too small.



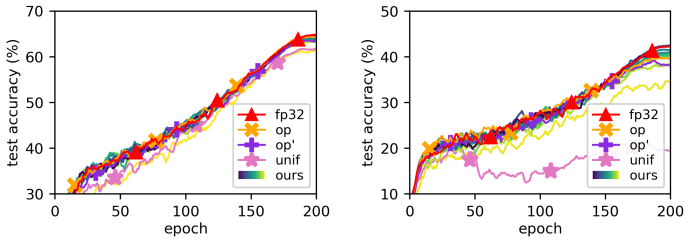
(a) CIFAR-10, SqueezeNet (b) CIFAR-100, SqueezeNet (c) CIFAR-100, SqueezeNet†

(d) CIFAR-10, ShuffleNet-v2 (e) CIFAR-100, ShuffleNet-v2 (f) CIFAR-100, ShuffleNet-v2†

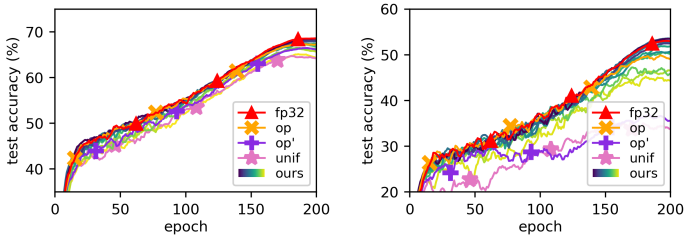
(g) CIFAR-10, MobileNet-v2 (h) CIFAR-100, MobileNet-v2 (i) CIFAR-100, MobileNet-v2†

(j) CIFAR-10, ResNet-18 (k) CIFAR-100, ResNet-18 (l) CIFAR-100, ResNet-18†

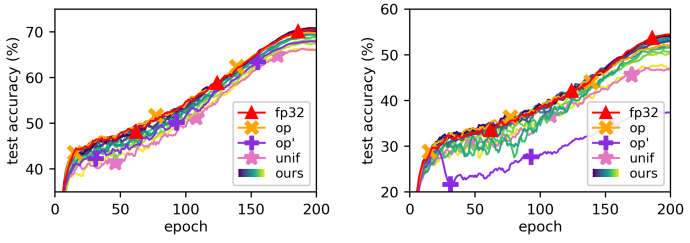
Figure C.4: Training trajectories for the configurations shown in Figure 6.4. Each line shows the average training trajectory for each precision assignment. $\pi_{\text{ours},r}$ is colored from navy to yellow (darker for smaller r).



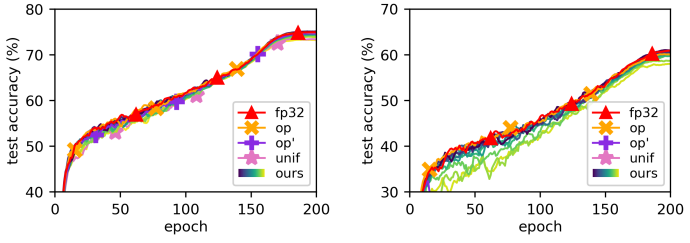
(a) CIFAR-100, SqueezeNet‡ (b) CIFAR-100, SqueezeNet¶



(c) CIFAR-100, ShuffleNet-v2‡ (d) CIFAR-100, ShuffleNet-v2¶



(e) CIFAR-100, MobileNet-v2‡ (f) CIFAR-100, MobileNet-v2¶



(g) CIFAR-100, ResNet-18‡ (h) CIFAR-100, ResNet-18¶

Figure C.5: Training trajectories for the configurations shown in Figure C.3. Each line shows the average training trajectory for each precision assignment. $\pi_{\text{ours},r}$ is colored from navy to yellow (darker for smaller r).

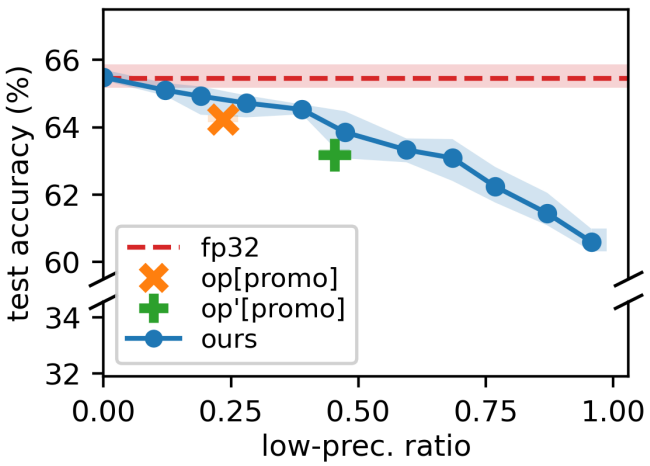
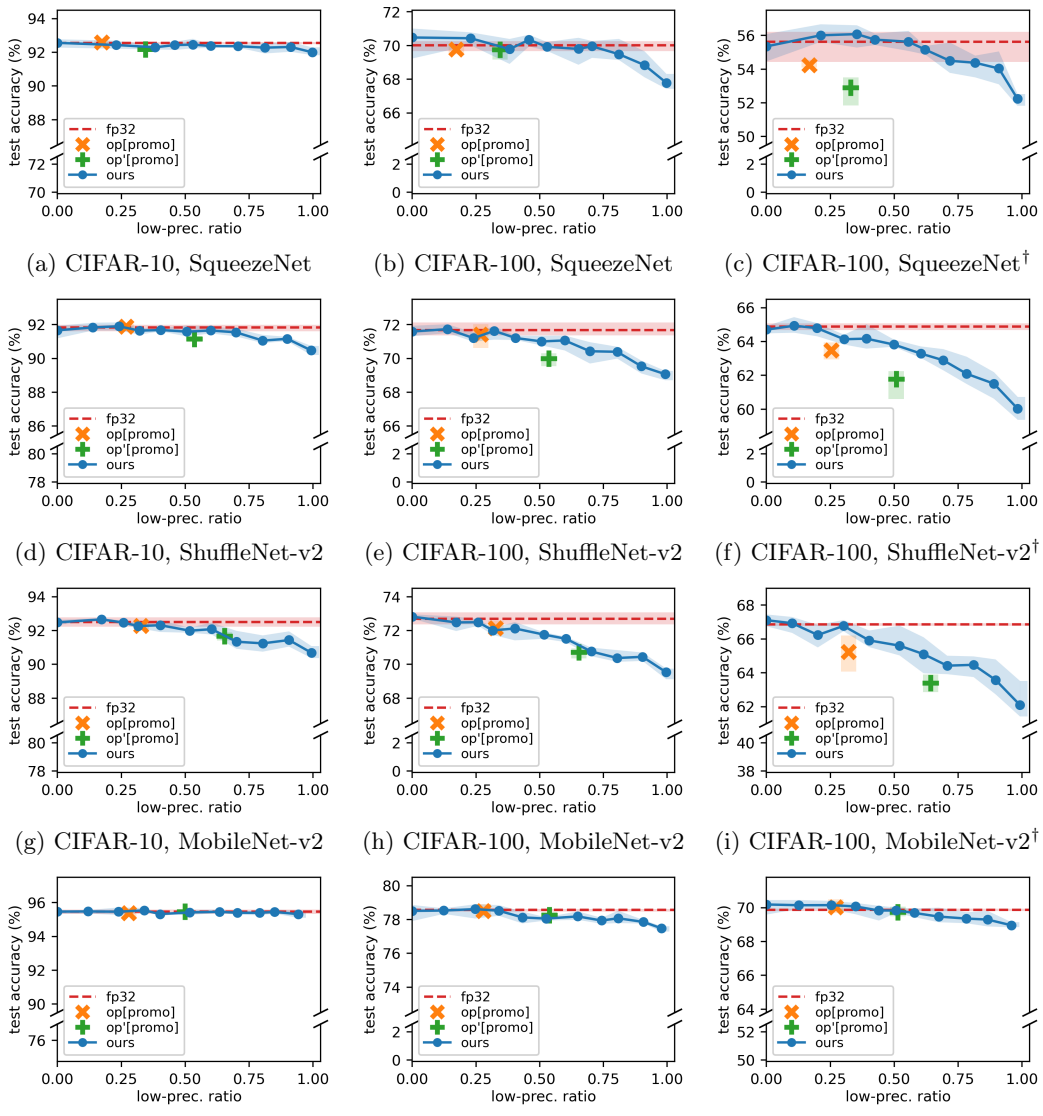


Figure C.6: Results corresponding to Figure 6.3. The only difference from Figure 6.3 is that π_{op} and $\pi_{\text{op}'}$ here are equipped with our precision promotion technique, whereas π_{op} and $\pi_{\text{op}'}$ in the previous figure do not so.



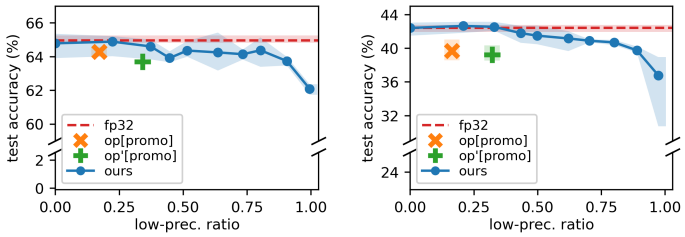
(a) CIFAR-10, SqueezeNet (b) CIFAR-100, SqueezeNet (c) CIFAR-100, SqueezeNet†

(d) CIFAR-10, ShuffleNet-v2 (e) CIFAR-100, ShuffleNet-v2 (f) CIFAR-100, ShuffleNet-v2†

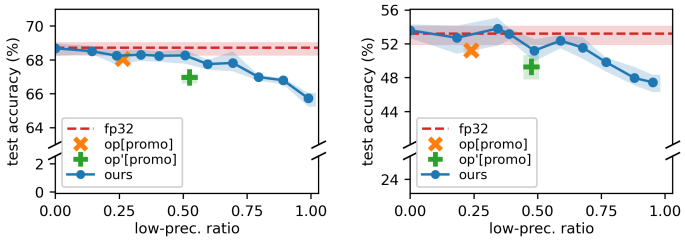
(g) CIFAR-10, MobileNet-v2 (h) CIFAR-100, MobileNet-v2 (i) CIFAR-100, MobileNet-v2†

(j) CIFAR-10, ResNet-18 (k) CIFAR-100, ResNet-18 (l) CIFAR-100, ResNet-18†

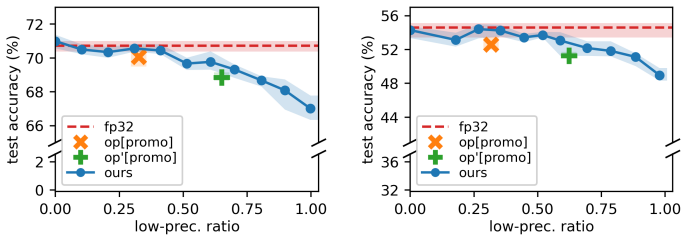
Figure C.7: Results corresponding to Figure 6.4. The only difference from Figure 6.4 is that π_{op} and $\pi_{\text{op}'}$ here are equipped with our precision promotion technique, whereas π_{op} and $\pi_{\text{op}'}$ in the previous figure do not so.



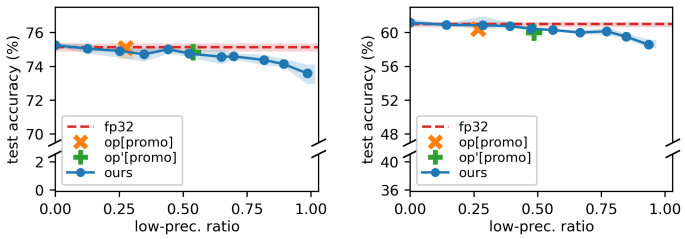
(a) CIFAR-100, SqueezeNet‡ (b) CIFAR-100, SqueezeNet¶



(c) CIFAR-100, ShuffleNet-v2‡ (d) CIFAR-100, ShuffleNet-v2¶

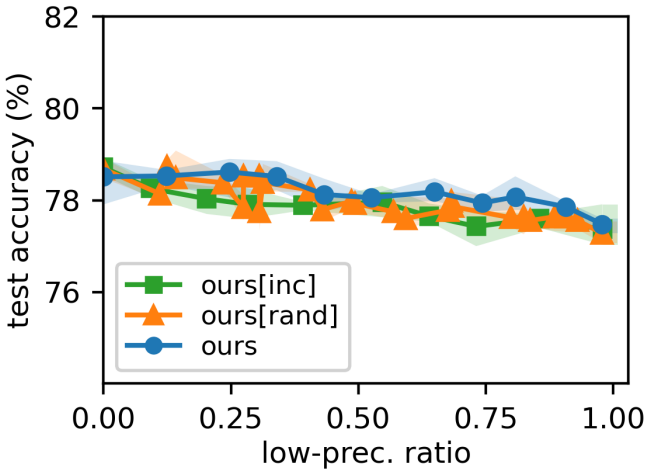


(e) CIFAR-100, MobileNet-v2‡ (f) CIFAR-100, MobileNet-v2¶



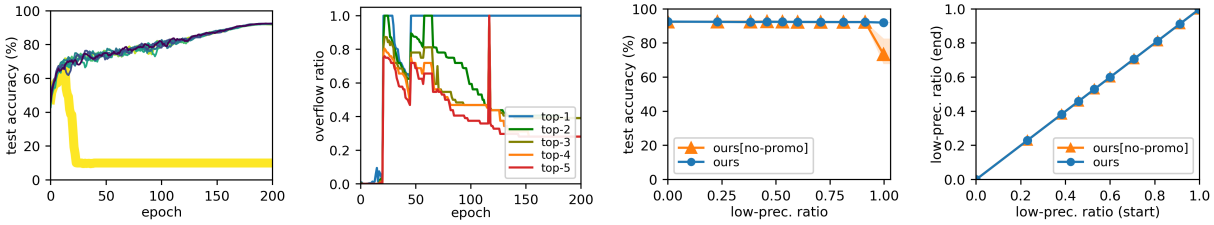
(g) CIFAR-100, ResNet-18‡ (h) CIFAR-100, ResNet-18¶

Figure C.8: Results corresponding to Figure C.3. The only difference from Figure C.3 is that π_{op} and $\pi_{\text{op}'}$ here are equipped with our precision promotion technique, whereas π_{op} and $\pi_{\text{op}'}$ in the previous figure do not so.

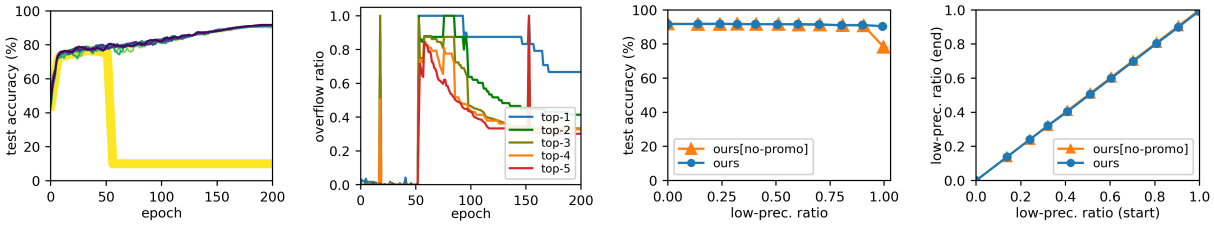


(a) ResNet-18

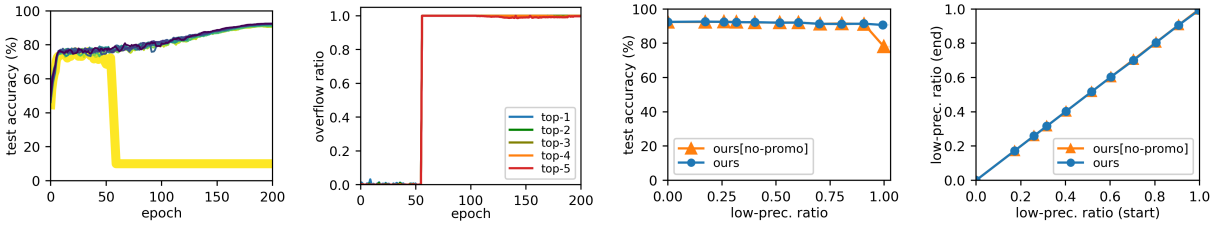
Figure C.9: Results continued from Figure 6.5. Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$, $\pi_{\text{ours[inc]},r}$, and $\pi_{\text{ours[rand]},r}$ for ResNet-18 on CIFAR-100. Observe that ●s are above and to the right of other points in nearly all cases.



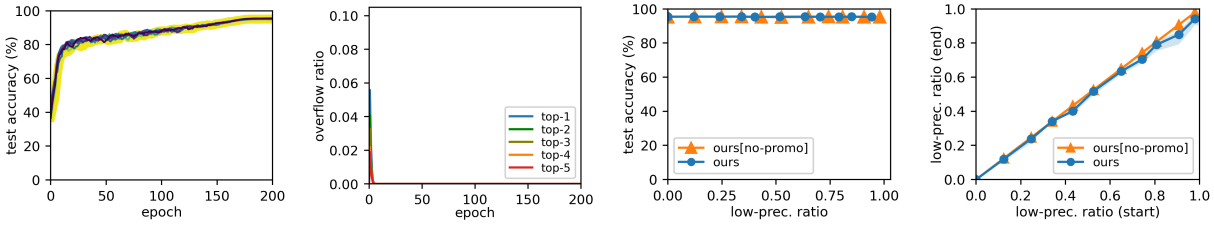
(a) CIFAR-10, SqueezeNet



(b) CIFAR-10, ShuffleNet-v2



(c) CIFAR-10, MobileNet-v2



(d) CIFAR-10, ResNet-18

Figure C.10: Results continued from Figure 6.6. Training four models on CIFAR-10 with $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. Column 1: Training trajectories of $\pi_{\text{ours[no-promo]},r}$ for different r; colors denote r values (darker for smaller r). Column 2: Top-5 overflow ratios of tensors at each epoch, for the highlighted trajectory in (a); the largest ratio is blue and the fifth largest red. Column 3: Memory-accuracy tradeoffs of $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$. Column 4: Low-precision ratio when training ends vs. when training starts, for $\pi_{\text{ours},r}$ and $\pi_{\text{ours[no-promo]},r}$.

Bibliography

[1] M. Abadi and G. D. Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):38:1–38:28, 2020.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[4] M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy. NVIDIA Hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022. Accessed on July 2023.

[5] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. Scalable methods for 8-bit training of neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 5151–5159, 2018.

[6] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 549–560, 2013.

[7] G. Barthe, R. Crubillé, U. D. Lago, and F. Gavazzo. On the versatility of open logical relations: Continuity, automatic differentiation, and a containment theorem. In *European Symposium on Programming (ESOP)*, pages 56–83, 2020.

[8] P. I. Barton, K. A. Khan, P. Stechlinski, and H. A. J. Watson. Computationally relevant generalized derivatives: Theory, evaluation and applications. *Optimization Methods and Software*, 33(4-6):1030–1072, 2018.

[9] A. G. Baydin, B. A. Pearlmutter, and J. M. Siskind. Diffsharp: An AD library for .NET languages. In *International Conference on Algorithmic Differentiation (AD)*, 2016. URL https://arxiv.org/abs/1611.03423.

[10] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18:153:1–153:43, 2017.

[11] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 453–462, 2012.

[12] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU math compiler in Python. In *Python in Science Conference (SciPy)*, pages 18–24, 2010.

[13] D. Bertoin, J. Bolte, S. Gerchinovitz, and E. Pauwels. Numerical influence of ReLU'(0) on backpropagation. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 468–479, 2021.

[14] J. Björck, X. Chen, C. De Sa, C. P. Gomes, and K. Weinberger. Low-precision reinforcement learning: Running soft actor-critic in half precision. In *International Conference on Machine Learning (ICML)*, pages 980–991, 2021.

[15] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[16] S. Boldo, C.-P. Jeannerod, G. Melquiond, and J.-M. Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023.

[17] J. Bolte and E. Pauwels. Conservative set valued fields, automatic differentiation, stochastic gradient method and deep learning. *Mathematical Programming*, 188:19–51, 2020.

[18] J. Bolte and E. Pauwels. A mathematical model for automatic differentiation in machine learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 10809–10819, 2020.

[19] J. Bolte, R. Boustany, E. Pauwels, and B. Pesquet-Popescu. On the complexity of nonsmooth automatic differentiation. In *International Conference on Learning Representations (ICLR)*, 2023.

[20] A. Brunel, D. Mazza, and M. Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *Proceedings of the ACM on Programming Languages*, 4(POPL):64:1–64:27, 2020.

[21] R. L. Burden, J. D. Faires, and A. M. Burden. *Numerical analysis*. Cengage learning, 10th edition, 2015.

[22] L. Cambier, A. Bhiwandiwalla, T. Gong, O. H. Elibol, M. Nekuii, and H. Tang. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[23] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, page 015001, 2009.

[24] W. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamaric. Rigorous floating-point mixed-precision tuning. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 300–315, 2017.

[25] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52, 2014.

[26] B. Chmiel, L. Ben-Uri, M. Shkolnik, E. Hoffer, R. Banner, and D. Soudry. Neural gradients are near-lognormal: Improved quantized and sparse training. In *International Conference on Learning Representations (ICLR)*, 2021.

[27] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan. PACT: Parameterized clipping activation for quantized neural networks. *arXiv:1805.06085*, 2018.

[28] F. H. Clarke. Generalized gradients and applications. *Transactions of the American Mathematical Society*, 205:247–262, 1975.

[29] F. H. Clarke. *Optimization and nonsmooth analysis*. Classics in Applied Mathematics: Volume 5. SIAM, 1990.

[30] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic crosschecking of floating-point and simd code. In *European Conference on Computer Systems (EuroSys)*, pages 315–328, 2011.

[31] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like environment for machine learning. In *NIPS BigLearn Workshop*, 2011.

[32] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 3123–3131, 2015.

[33] Y. Cui and J.-S. Pang. *Modern nonconvex nondifferentiable optimization*. MOS-SIAM Series on Optimization. SIAM, 2021.

[34] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, and J.-M. Muller. CR-Libm, a library of correctly rounded elementary functions in double-precision. Available at https://ens-lyon.hal.science/ensl-01529804/document, 2005.

[35] E. Darulova and V. Kuncak. Sound compilation of reals. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 235–248, 2014.

[36] E. Darulova and V. Kuncak. Towards a compiler for reals. *ACM Transactions on Programming Languages and Systems*, 39(2):8:1–8:28, 2017.

[37] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha. Scalable yet rigorous floating-point error analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 51, 2020.

[38] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke, P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. Pirogov. Mixed precision training of convolutional neural networks using integer operations. In *International Conference on Learning Representations (ICLR)*, 2018.

[39] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):2:1–2:20, 2010.

[40] D. Davis, D. Drusvyatskiy, S. M. Kakade, and J. D. Lee. Stochastic subgradient method converges on tame functions. *Foundations of Computational Mathematics*, 20(1):119–154, 2020.

[41] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[42] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[43] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 53–69, 2009.

[44] M. Drumond, T. LIN, M. Jaggi, and B. Falsafi. Training DNNs with hybrid block floating point. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 451–461, 2018.

[45] S. Duplichan. Intel overstates FPU accuracy. http://notabs.org/fpuaccuracy/, 2013. Accessed on July 2023.

[46] A. Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39(1):54–67, 1997.

[47] C. Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):70:1–70:29, 2018.

[48] S. Fox, S. Rasoulinezhad, J. Faraone, david boland, and P. Leong. A block minifloat representation for training deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2021.

[49] R. Frostig, M. Johnson, and C. Leary. Compiling machine learning programs via high-level tracing. In *SysML Conference*, 2018.

[50] Z. Fu and Z. Su. Achieving high coverage for floating-point code via unconstrained programming. In *ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 306–319, 2017.

[51] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision: Improving the Efficiency of Artificial Intelligence*, pages 291–326. CRC Press, 2022. URL https://arxiv.org/abs/2103.13630.

[52] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[53] E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. In *International Workshop on Numerical and Symbolic Abstract Domains (NSAD)*, 2005.

[54] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 3–20, 2007.

[55] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv:1706.02677*, 2017.

[56] A. Griewank and A. Walther. *Evaluating derivatives: Principles and techniques of algorithmic differentiation*. SIAM, 2nd edition, 2008.

[57] H. Guo and C. Rubio-González. Exploiting community structure for floating-point precision tuning. In *International Symposium on Software Testing and Analysis (ISSTA)*, page 333–343, 2018.

[58] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning (ICML)*, pages 1737–1746, 2015.

[59] J. L. Gustafson and I. T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *The Journal of Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.

[60] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 131–140, 2012.

[61] J. Harrison. Floating-point verification in HOL Light: The exponential function. *Formal Methods in System Design*, 16(3):271–305, 2000.

[62] J. Harrison. Formal verification of floating point trigonometric functions. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 217–233, 2000.

[63] J. Harrison, T. Kubaska, S. Story, and P. T. P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 4:234–251, 1999.

[64] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. G. Thacher, and C. Witzgall. *Computer approximations*. John Wiley & Sons, 1968.

[65] L. Hascoët and V. Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43, 2013.

[66] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, 1996.

[67] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[68] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2nd edition, 2002.

[69] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[70] M. Huot, S. Staton, and M. Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 319–338, 2020.

[71] M. Huot, A. K. Lew, V. K. Mansinghka, and S. Staton. ωPAP spaces: Reasoning denotationally about higher-order, recursive probabilistic and differentiable programs. In *ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2023.

[72] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1mb model size. *arXiv:1602.07360*, 2016.

[73] IEEE Computer Society. IEEE standard for floating-point arithmetic (IEEE Std 754-2019), 2019.

[74] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.

[75] A. Jacot, C. Hongler, and F. Gabriel. Neural tangent kernel: Convergence and generalization in neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 8580–8589, 2018.

[76] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *International Conference on Multimedia (MM)*, pages 675–678, 2014.

[77] S. M. Kakade and J. D. Lee. Provably correct automatic sub-differentiation for qualified programs. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 7125–7135, 2018.

[78] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey. A study of BFLOAT16 for deep learning training. *arXiv:1905.12322*, 2019.

[79] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[80] K. A. Khan and P. I. Barton. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software*, 30(6):1185–1212, 2015.

[81] P. Kidger and T. Lyons. Universal approximation with deep narrow networks. In *Conference on Learning Theory (COLT)*, pages 2306–2327, 2020.

[82] D. E. Knuth. *The art of computer programming, Volume II: Seminumerical algorithms*. Addison-Wesley, 3rd edition, 1998.

[83] F. Krawiec, S. P. Jones, N. Krishnaswami, T. Ellis, R. A. Eisenberg, and A. W. Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages*, 6(POPL):48:1–48:30, 2022.

[84] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. URL `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`. Accessed on July 2023.

[85] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997*, 2014.

[86] kuangliu. `https://github.com/kuangliu/pytorch-cifar`, 2021.

[87] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy: Search-based floating point constraint solving for symbolic execution. In *International Conference on Testing Software and Systems (ICTSS)*, pages 142–157, 2010.

[88] T. Laurent and J. von Brecht. The multilinear structure of ReLU networks. In *International Conference on Machine Learning (ICML)*, pages 2914–2922, 2018.

[89] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[90] W. Lee, R. Sharma, and A. Aiken. Verifying bit-manipulations of floating-point. In *ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 70–84, 2016.

[91] W. Lee, R. Sharma, and A. Aiken. On automatically proving the correctness of math.h implementations. *Proceedings of the ACM on Programming Languages*, 2(POPL):47:1–47:32, 2018.

[92] W. Lee, H. Yu, X. Rival, and H. Yang. On correctness of automatic differentiation for non-differentiable functions. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 6719–6730, 2020.

[93] W. Lee, S. Park, and A. Aiken. On the correctness of automatic differentiation for neural networks with machine-representable parameters. In *International Conference on Machine Learning (ICML)*, pages 19094–19140, 2023.

[94] W. Lee, R. Sharma, and A. Aiken. Training with mixed-precision floating-point assignments. *Transactions on Machine Learning Research*, 2023.

[95] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl. Make it real: Effective floating-point reasoning via exact arithmetic. In *Design, Automation, and Test in Europe (DATE)*, pages 1–4, 2014.

[96] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 6232–6240, 2017.

[97] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *European Conference on Computer Vision (ECCV)*, pages 122–138, 2018.

[98] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in Numpy. In *ICML AutoML Workshop*, 2015.

[99] V. Magron, G. A. Constantinides, and A. F. Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Transactions on Mathematical Software*, 43(4):34:1–34:31, 2017.

[100] P. Markstein. *IA-64 and elementary functions: Speed and precision.* Prentice Hall, 2000.

[101] D. Mazza and M. Pagani. Automatic differentiation in PCF. *Proceedings of the ACM on Programming Languages*, 5(POPL):28:1–28:27, 2021.

[102] G. Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012.

[103] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger. ADAPT: Algorithmic differentiation applied to floating-point precision tuning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 48:1–48:13, 2018.

[104] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. In *International Conference on Learning Representations (ICLR)*, 2018.

[105] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. F. Oberman, M. Shoeybi, M. Y. Siu, and H. Wu. FP8 formats for deep learning. *arXiv:2209.05433*, 2022.

[106] A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Workshops on Automated Theory eXploration and on Invariant Generation (ATx/WInG)*, pages 55–70, 2012. URL https://hal.science/hal-00748094.

[107] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *International Static Analysis Symposium (SAS)*, pages 316–333, 2011.

[108] J.-M. Muller. *Elementary functions: Algorithms and implementation.* Springer, 3rd edition, 2016.

[109] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefevre, G. Melquiond, N. Revol, and S. Torres. *Handbook of floating-point arithmetic*. Springer, 2nd edition, 2018.

[110] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort. A white paper on neural network quantization. *arXiv:2106.08295*, 2021.

[111] G. C. Necula and S. Gulwani. Randomized algorithms for program analysis and verification. In *International Conference on Computer Aided Verification (CAV)*, page 1, 2005.

[112] A. Nötzli and F. Brown. Lifejacket: Verifying precise floating-point optimizations in LLVM. In *International Workshop on the State Of the Art in Program Analysis (SOAP)*, pages 24–29, 2016.

[113] Nvidia. Documentation of `apex.amp`. https://nvidia.github.io/apex/amp.html, 2019. Accessed on July 2023.

[114] M. L. Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

[115] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 1–11, 2015.

[116] S. Park, C. Yun, J. Lee, and J. Shin. Minimum width for universal approximation. In *International Conference on Learning Representations (ICLR)*, 2021.

[117] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[118] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019.

[119] G. Pathak. https://github.com/gsp-27/pytorch_Squeezenet, 2020.

[120] B. A. Pearlmutter and J. M. Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems*, 30(2):7:1–7:36, 2008.

[121] PyTorch. Documentation of `torch.amp`. https://pytorch.org/docs/stable/amp.html, 2022. Accessed on July 2023.

[122] PyTorch. https://github.com/pytorch/vision/tree/main/torchvision/models, 2022.

[123] PyTorch. https://github.com/pytorch/vision/tree/main/references/classification, 2022.

[124] PyTorch. https://github.com/pytorch/vision/tree/main/references/classification#resnext, 2022.

[125] H. Qin, Y. Ding, W. Fan, C. Leff, M. Bahri, and E. Shaw. Awesome model quantization. https://github.com/htqin/awesome-model-quantization, 2022.

[126] A. Radul, A. Paszke, R. Frostig, M. J. Johnson, and D. Maclaurin. You only linearize once: Tangents transpose to gradients. *Proceedings of the ACM on Programming Languages*, 7 (POPL):43:1–43:29, 2023.

[127] A. Rajagopal, D. A. Vink, S. I. Venieris, and C.-S. Bouganis. Multi-precision policy enforced training (MuPPET): A precision-switching strategy for quantised fixed-point training of CNNs. In *International Conference on Machine Learning (ICML)*, pages 7943–7952, 2020.

[128] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin. A unified Coq framework for verifying C programs with floating-point computations. In *International Conference on Certified Programs and Proofs (CPP)*, pages 15–26, 2016.

[129] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892*, 2016.

[130] R. T. Rockafellar and R. J.-B. Wets. *Variational analysis*. A Series of Comprehensive Studies in Mathematics: Volume 317. Springer Science & Business Media, 1998.

[131] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 27:1–27:12, 2013.

[132] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point precision tuning using blame analysis. In *International Conference on Software Engineering (ICSE)*, page 1074–1085, 2016.

[133] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[134] C. D. Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré. High-accuracy low-precision training. *arXiv:1803.03383*, 2018.

[135] C. Sakr and N. Shanbhag. Per-tensor fixed-point quantization of the back-propagation algorithm. In *International Conference on Learning Representations (ICLR)*, 2019.

[136] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[137] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs using tunable precision. In *ACM Symposium on Programming Language Design and Implementation (PLDI)*, pages 53–64, 2014.

[138] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[139] S. Scholtes. *Introduction to piecewise differentiable equations*. SpringerBriefs in Optimization. Springer Science & Business Media, 2012.

[140] F. Seide and A. Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, page 2135, 2016.

[141] E. Slusanschi and V. Dumitrel. ADiJaC – automatic differentiation of Java classfiles. *ACM Transactions on Mathematical Software*, 43(2):9:1–9:33, 2016.

[142] T. Smeding and M. Vákár. Efficient dual-numbers reverse AD via well-known program transformations. *Proceedings of the ACM on Programming Languages*, 7(POPL):54:1–54:28, 2023.

[143] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In *International Symposium on Formal Methods (FM)*, pages 532–550, 2015.

[144] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1):2:1–2:39, 2019.

[145] Stanford Vision Lab. `https://image-net.org/download.php`, 2020. Accessed on July 2023.

[146] P. H. Sterbenz. *Floating-point computation*. Prentice Hall, Englewood Cliffs, NJ, 1973.

[147] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan. Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 4901–4910, 2019.

[148] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 1796–1807, 2020.

[149] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Y. Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 2002–2011, 2019.

[150] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[151] M. Vákár. Reverse AD at higher types: Pure, principled and denotationally correct. In *European Symposium on Programming (ESOP)*, pages 607–634, 2021.

[152] B. van Merrienboer, D. Moldovan, and A. B. Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 6259–6268, 2018.

[153] J. von Neumann and H. H. Goldstine. Numerical inverting of matrices of high order: I. *Bulletin of the American Mathematical Society*, 53(11):1021–1099, 1947.

[154] A. Walther and A. Griewank. Getting started with ADOL-C. In *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman & Hall/CRC Computational Science, 2012.

[155] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 7686–7695, 2018.

[156] K. Weihrauch. *Computable analysis: An introduction*. Springer, 2000.

[157] J. H. Wilkinson. *Rounding errors in algebraic processes*. Prentice Hall, 1963.

[158] J. H. Wilkinson. *The algebraic eigenvalue problem*. Oxford University Press, 1988.

[159] S. Wu, G. Li, F. Chen, and L. Shi. Training and inference with integers in deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[160] C. Yang, Z. Wu, J. Chee, C. D. Sa, and M. Udell. How low can we go: Trading memory for error in low-precision training. In *International Conference on Learning Representations (ICLR)*, 2022.

[161] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. De Sa. SWALP: Stochastic weight averaging in low precision training. In *International Conference on Machine Learning (ICML)*, pages 7015–7024, 2019.

[162] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X.-s. Hua. Quantization networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7308–7316, 2019.

[163] P. Zamirai, J. Zhang, C. R. Aberger, and C. D. Sa. Revisiting BFloat16 training. *arXiv:2010.06192*, 2020.

[164] T. Zhang, Z. Lin, G. Yang, and C. D. Sa. QPyTorch: A low-precision arithmetic simulation framework. *arXiv:1910.04540*, 2019.

[165] X. Zhang, S. Liu, R. Zhang, C. Liu, D. Huang, S. Zhou, J. Guo, Q. Guo, Z. Du, T. Zhi, and Y. Chen. Fixed-point back-propagation training. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2327–2335, 2020.

[166] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv:1606.06160*, 2016.