A Dual Number Abstraction for Static Analysis of Clarke Jacobians

JACOB LAUREL, University of Illinois at Urbana-Champaign, USA
REM YANG, University of Illinois at Urbana-Champaign, USA
GAGANDEEP SINGH, University of Illinois at Urbana-Champaign and VMware Research, USA
SASA MISAILOVIC, University of Illinois at Urbana-Champaign, USA

We present a novel abstraction for bounding the Clarke Jacobian of a Lipschitz continuous, but not necessarily differentiable function over a local input region. To do so, we leverage a novel abstract domain built upon dual numbers, adapted to soundly over-approximate all first derivatives needed to compute the Clarke Jacobian. We formally prove that our novel forward-mode dual interval evaluation produces a sound, interval domain-based over-approximation of the true Clarke Jacobian for a given input region.

Due to the generality of our formalism, we can compute and analyze interval Clarke Jacobians for a broader class of functions than previous works supported – specifically, arbitrary compositions of neural networks with Lipschitz, but non-differentiable perturbations. We implement our technique in a tool called DeepJ and evaluate it on multiple deep neural networks and non-differentiable input perturbations to showcase both the generality and scalability of our analysis. Concretely, we can obtain interval Clarke Jacobians to analyze Lipschitz robustness and local optimization landscapes of both fully-connected and convolutional neural networks for rotational, contrast variation, and haze perturbations, as well as their compositions.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Abstract Interpretation, Differentiable Programming, Robustness

ACM Reference Format:
Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. 2022. A Dual Number Abstraction for Static Analysis of Clarke Jacobians. *Proc. ACM Program. Lang.* 6, POPL, Article 56 (January 2022), 30 pages.https://doi.org/10.1145/3498718

1 INTRODUCTION

Recent years have seen growing adoption of machine learning (ML) models in several safety critical domains, including autonomous driving [Bojarski et al. 2016] and healthcare [Esteva et al. 2019]. The first derivatives specified via the Jacobian matrix are the backbone of many prominent learning paradigms and are used in all facets of the machine learning pipeline, from training to testing. Automatic Differentiation (AD), and more broadly, Differentiable Programming, have been developed to offer a principled, language-based method of compositionally computing derivatives. However, the needs of ML researchers have rapidly outpaced formal development on the programming languages side. For instance, ML techniques regularly must differentiate

Authors' addresses: Jacob Laurel, jlaurel2@illinois.edu, University of Illinois at Urbana-Champaign, USA; Rem Yang, remyang2@illinois.edu, University of Illinois at Urbana-Champaign, USA; Gagandeep Singh, ggnds@illinois.edu, University of Illinois at Urbana-Champaign and VMware Research, USA; Sasa Misailovic, misailo@illinois.edu, University of Illinois at Urbana-Champaign, USA.

2475-1421/2022/1-ART56
https://doi.org/10.1145/3498718

through functions that may have points of non-differentiability, such as the commonly used ReLU activation in neural networks, which is not differentiable at 0. In practice, for such a situation, existing AD frameworks just return any arbitrary number in the interval $[0, 1]$. While such an ad-hoc approach may suffice in many practical scenarios, for critical domains requiring formal certification of properties defined over the first derivatives, this lack of rigor is troubling. To resolve this limitation, programming language researchers have begun using generalizations of the Jacobian for describing AD semantics [Sherman et al. 2021], such as the *Clarke Generalized Jacobian* [Clarke 1990] for non-smooth, but Lipschitz continuous functions.

However, these developments still cannot provide the desired formal guarantees for practical verification tasks, such as obtaining formal bounds on the Lipschitz constant (which measures how rapidly a function's output changes) of the composition of a non-differentiable perturbation and a differentiable network within an input region. Lipschitz constants can be used as a metric to compare the stability and smoothness of the output of neural networks prior to deployment, as a network with a smaller constant is often preferable [Lin et al. 2019]. Formal bounds on the Lipschitz constant can also be used during training to learn classifiers that are certifiably robust to adversarial perturbations [Tsuzuku et al. 2018], robust to quantizations [Lin et al. 2019], or to improve interpretability by making network explanations themselves more robust [Alvarez-Melis and Jaakkola 2018]. Further, analyzing the Lipschitz constant has direct applications in algorithmic fairness [Dwork et al. 2012] and differential privacy [Dwork et al. 2006], where fairness and privacy are established by certifying a small Lipschitz constant. Beyond using the Jacobian for formally bounding Lipschitz constants, a Jacobian analysis can also be used to formally reason about the local geometry of ML models [Zhang et al. 2019], which has applications in explainability, e.g., why a network works well for certain inputs but not for others.

Challenges. We focus on designing a static analysis that can formally reason about not only Jacobians of functions that are differentiable (e.g., tanh), but can also handle non-differentiable behavior due to functions like ReLU and the branching that can arise in differentiable programs. Furthermore, we must also handle high-dimensional computational graphs with arbitrary arithmetic (instead of, e.g., solely neural networks that avoid non-scalar multiplication and division). Simultaneously handling *all* of these requires a theoretical formalism that is beyond the scope of the existing works [Edalat and Maleki 2017; Jordan and Dimakis 2020; Mangal et al. 2020; Zhang et al. 2019]. One of the main difficulties arises from the fact that generalized notions of Jacobians do not always obey the same rules as classical Jacobians. For example, the Clarke Jacobian cannot simply be computed by concatenating partial Clarke derivatives into a matrix [Clarke 1990; Khan and Barton 2013]. In this same light, we want our desired generality to also come with compositionality: we want to be able to combine different functions together instead of restricting ourselves to a non-compositional analysis limited to a specific, fixed type of function (e.g., a DNN) or input perturbation. More practically, we want this analysis to be scalable and fully end-to-end, specifically for real-world problems such as analyzing Lipschitz robustness with respect to multiple input perturbations. This has proven challenging, as most formalisms that aim for broad theoretical generality cannot scale beyond toy examples [Di Gianantonio and Edalat 2013], do not have implementations [Edalat and Lieutier 2004; Edalat et al. 2013; Edalat and Maleki 2017, 2018], or lack end-to-end integration with specific analyses for practical problems. Moreover, the Lipschitz robustness analyses that go beyond toy examples, such as RecurJac [Zhang et al. 2019] and ProLip [Mangal et al. 2020], are either tailored for handling fully-connected architectures (like RecurJac) and therefore cannot immediately analyze the state-of-the-art convolutional architectures, or are heavily restricted in the activation functions supported (like ProLip). These issues substantially limit the practical applicability of these tools.

This Work. To address these challenges, we propose DeepJ, a forward-mode interval abstraction built atop dual numbers (the canonical number system used for implementing forward-mode AD). A dual number $a + b\epsilon$ has two components: the *real part* a and the *dual part* b, which in applications will correspond to a function's derivative at a.

The analysis is adapted to compute an interval over-approximation of the Clarke Jacobian. Our key insight is that formalizing the static analysis on top of dual numbers as a forward-mode analysis represents a general solution suitable for reasoning about Clarke derivatives, which can simultaneously offer both an intuitive and scalable implementation. Hence, we reduce the problem of bounding a Clarke Jacobian in a local region to the problem of bounding results of dual number arithmetic and functions. Forward-mode analysis can be particularly useful for multiple practical problems, such as analyzing Lipschitz robustness with respect to individual input perturbations or their compositions. These problems have small input dimension, for which a forward-mode analysis requires fewer passes than a reverse-mode analysis.

DeepJ analyzes a first-order core (without unbounded loops or recursion) of the language proposed by Sherman et al. [2021], which we extend with conditional branch expressions and also show necessary conditions for the well-definedness of the Clarke Jacobian of these branches (Section 4). We then recursively define an interval-domain abstraction of the Clarke Jacobian for sets of points and prove this over-approximation sound (Section 5). Next, we show how to equivalently compute this interval Clarke Jacobian in a forward pass using a novel, interval-domain abstraction of dual numbers (Section 6). Finally, we demonstrate how DeepJ leverages the interval over-approximation of the Clarke Jacobian for multiple practical uses in a fully end-to-end and scalable manner, namely analyzing Lipschitz robustness and local optimization geometry of large neural networks in the face of non-smooth input perturbations (Section 8). DeepJ's implementation also optionally offers floating-point soundness [Miné 2004], i.e., its result can capture all possible outputs under different rounding modes and under different orders of computations of floating-point operations. This guarantee is not possible with any other existing method.

The novelty in our work lies in the fact that we are the first to formalize a static analysis that is simultaneously (a) defined for the more general Clarke Jacobian, thus supporting both differentiable and non-differentiable, but still Lipschitz functions, (b) extends to all arithmetic operations and is defined for branching beyond just min and max, (c) is fully compositional by leveraging an interval abstraction of forward-mode dual numbers, and (d) is integrated in a fully end-to-end manner for practical tasks that no prior work could tackle.

Results. We implement DeepJ and evaluate it on two tasks:

- *Lipschitz Robustness:* Certifies bounds on the local Lipschitz constant of a given input region.
- *Local Optimization Landscape:* Uses the Clarke Jacobian to analyze a function's local geometry in a specified input region to determine the absence of stationary points.

We apply DeepJ to neural networks with both fully-connected and convolutional layers that are trained on the CIFAR10 [Krizhevsky et al. 2009] and MNIST [LeCun et al. 1998] datasets. For each of our analyses, we compose the neural networks with three perturbation functions: *haze* (which models images as foggy), *contrast variation* (which accentuates the differences between bright and dark pixels), and *rotation* (which rotates the image by a specified angle θ with bilinear interpolation). Analyzing the Jacobian of a network with respect to these perturbations is out of reach of the existing techniques.

For each perturbation, DeepJ is able to leverage its fully localized analysis, which computes Clarke Jacobians solely for the specified input region, to obtain local Lipschitz constants that can be up to several orders of magnitude smaller than a baseline analysis based on Gouk et al. [2021] (which multiplies a network's global Lipschitz constant by the perturbation's local Lipschitz

constant instead of being fully localized). DeepJ can also extend the analysis to compositions of multiple perturbations. Furthermore, the localized Jacobian analysis can certify the absence of stationary points in a network's optimization landscape. DeepJ's parallel CPU-based implementation is efficient: it can precisely analyze 100 CIFAR10 images on our largest convolutional network containing > 62K neurons within a median time of 15 seconds each for haze and contrast variations, and under 1.4 minutes for rotation. It can also compute precise results for 10 CIFAR10 images on the same network within a median time of 9 seconds for contrast variation composed with haze, and under 51 seconds for contrast variation or haze composed with rotation. Our largest network has the same architecture as the one commonly handled by state-of-the-art CPU-based robustness verifiers [Singh et al. 2018, 2019; Urban and Miné 2021]. The differences in the computed constants between the DeepJ versions with and without sound floating-point rounding are negligible, with 2.8-4.1x execution time overhead of the sound version.

Contributions. The paper makes the following main contributions:

(1) A new dual number-based interval abstraction for analyzing Clarke Jacobians. Our domain soundly over-approximates the Clarke Generalized Jacobian of locally Lipschitz and piecewise differentiable functions, allowing it to soundly handle functions like max, ReLU, and limited branches. Furthermore, as our abstraction is defined for sets of points, we can analyze local properties of locally Lipschitz functions beyond the scope of prior work.
(2) A novel Clarke Jacobian analysis and Lipschitz certification of neural networks composed with non-differentiable perturbations.
(3) A scalable, optionally floating-point sound, implementation of our method which supports both convolutional and fully-connected neural networks.
(4) An extensive evaluation against multiple perturbation types on several deep neural networks, showing that DeepJ can (a) achieve orders of magnitude tighter bounds on local Lipschitz constants compared to a baseline analysis and (b) certify the absence of stationary points within a given input region.

DeepJ is available at https://github.com/uiuc-arc/DeepJ. Our implementation exploits CPU-level parallelism. We believe that DeepJ can be easily parallelized over GPUs to boost the scalability to even larger architectures, such as those considered by GPU-based verifiers [Müller et al. 2021]. The GPU extension of DeepJ may help train networks to be robust to semantic non-differentiable perturbations like rotation and contrast variation, which is beyond the reach of existing robust training methods [Balunović et al. 2019; Mirman et al. 2018, 2019; Zhang et al. 2021, 2020].

2 OVERVIEW

In this section, we start with a small illustrative example that showcases a real-world use of our abstraction for a scenario not handled by any prior work.

Running Example: Contrast Variation Perturbation. We consider the simple fully-connected network shown in Fig. 1. For simplicity, the network takes two inputs in the input layer, and we assume the network has been fully trained. The network contains a single hidden layer and all activation functions in both the hidden and output layers are tanh (with no biases). Most importantly, we compose the network with a perturbation function modeling contrast variation. We are interested in knowing precisely how sensitive the network's outputs are to inputs that are perturbed by contrast variation, which often arises when the image passed to a network was obtained with a fixed aperture lens [Paterson et al. 2021]. Hence, instead of passing input image pixels x_1 and x_2 directly into the input layer, we pass their *perturbed* values, x_1' and x_2'. The contrast variation

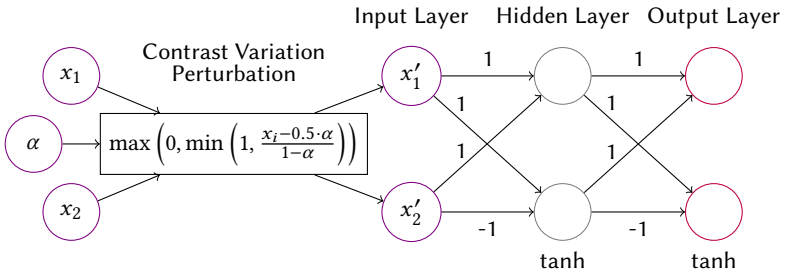

Fig. 1. Composition of a Neural Network with the Contrast Variation Perturbation

perturbation for a pixel x_i is given in Paterson et al. [2021] as

$$x_i' = \max\left(0, \min\left(1, \frac{x_i - 0.5 \cdot \alpha}{1 - \alpha}\right)\right) \tag{1}$$

where α specifies the amount of contrast variation. We apply this perturbation to every input pixel (there are only two in this example). We can thus think of the perturbation as a function of the perturbation parameter α.

Jacobian Analysis. We will measure the sensitivity of the network with respect to the perturbation by computing the Jacobian of the composition of the network with the perturbation. This allows us to analyze how sensitive and robust the network's outputs are with respect to a change in α in some local region. For this example, the local region we are interested in is when $\alpha \in [0, 0.1]$. This allows us to analyze what happens when the amount of perturbation ranges from none up to a modest amount. Because of the combination of both non-differentiable (max and min) and differentiable (tanh) functions, as well as the division in the perturbation function, computing a Jacobian for the composition of the network and the perturbation is beyond the capabilities of existing frameworks.

Abstract Domain. To perform the analysis, we need to compute bounds on the Jacobian of the composition of the network and the perturbation. However, this is complicated by the fact that (a) we cannot settle for the derivatives at a single point (as standard automatic differentiation gives) and instead need a bound on the derivatives for an entire input region, (b) max and min are Lipschitz, but not differentiable, thus we need a more general notion of differentiation that works for such functions, and (c) the analysis cannot be restricted to only neural networks, since it needs to be able to compositionally handle arbitrary combinations of functions. As mentioned, prior work cannot address these challenges, thus our solution necessitates a novel abstract domain. The full formalism is described in Sections 5 and 6.

Our abstract domain associates to each variable an interval bounding the variable itself and an interval bounding that variable's Clarke derivative via a *dual interval* of the form $[a, b] + [c, d]\epsilon$, where $a, b, c, d \in \mathbb{R} \cup \{\pm\infty\}$, $a \leq b$ and $c \leq d$. We will call $[a, b]$ the real part and $[c, d]$ the dual part. Dual intervals are an adapted interval-domain abstraction of the canonical dual numbers of forward-mode automatic differentiation. The key benefit of this approach is that we can leverage an existing numerical system to track derivatives instead of relying on non-extensible, ad-hoc approaches as in Edalat et al. [2013]; Mangal et al. [2020]; Zhang et al. [2019]. However, a naive adaptation is not sufficient, as one still has to contend with non-differentiable functions such as max. Furthermore, all primitive operations must be reinterpreted for dual interval arithmetic.

Abstract Interpretation of the Perturbation and Network. We now step through the abstract interpretation of the composition of the neural network and the contrast variation perturbation, which

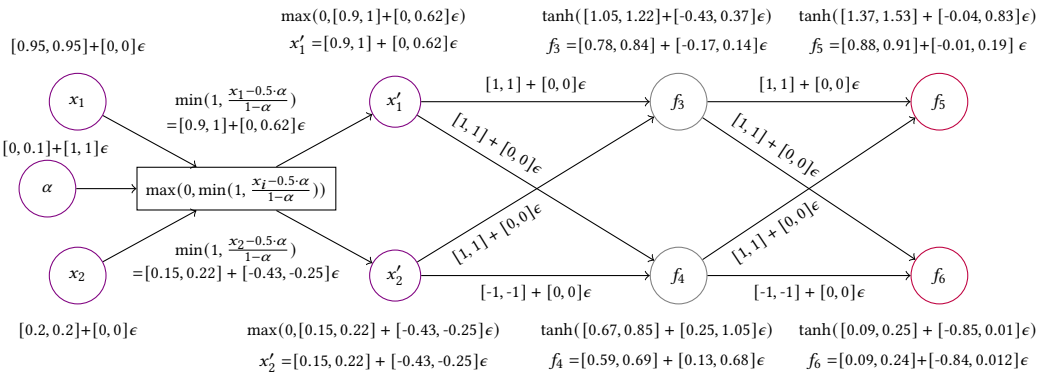

Fig. 2. The Dual Interval Abstraction of the Neural Network and Perturbation Function from Fig. 1

is detailed in Fig. 2. As every term in the abstract domain must be a dual interval, to perform the analysis, we first must lift the constant edge weights w_i to dual intervals of the form $[w_i, w_i]+[0,0]\epsilon$, as shown in Fig. 2. Upon lifting all constants and edge weights to the abstract domain, we start from the inputs x_1, x_2 and the perturbation parameter α. As our goal is to compute the sensitivity solely with respect to α, we set its dual part to $[1,1]\epsilon$. This means that we treat x_1 and x_2 as constant; since we do not need to compute derivatives with respect to x_1 and x_2, the analysis sets their dual part to $[0,0]\epsilon$. Furthermore, as x_1 and x_2 are pixel values of some fixed image, their real parts are just the degenerate interval of their pixel intensities: $[0.95, 0.95]$ and $[0.2, 0.2]$, respectively.

We begin by propagating the dual interval inputs through the contrast variation perturbation function in Eq. 1. This requires that all arithmetic operations $(+, -, \cdot, /)$ and function primitives (min and max) be redefined for dual intervals. Section 6 presents a full formalization of dual intervals.

For each pixel x_i, we first compute $\frac{x_i - 0.5\cdot\alpha}{1-\alpha}$ inside the min function. Though not shown in the function, the constants 0.5 and 1 are actually interpreted as the dual intervals $[0.5, 0.5] + [0,0]\epsilon$ and $[1,1]+[0,0]\epsilon$, respectively. The numerator $x_1 - 0.5\cdot\alpha$ abstractly evaluates to $[0.9, 0.95]+[\text{-}0.5, \text{-}0.5]\epsilon$ and likewise $x_2 - 0.5\cdot\alpha$ evaluates to $[0.15, 0.2] + [\text{-}0.5, \text{-}0.5]\epsilon$. These follow from the rules of dual interval arithmetic: scaling by a constant scales a term's real and dual part, and dual interval addition between terms adds their respective real and dual components. Therefore, we have implicitly encoded the notion of linearity of the derivative.

To compute the entire quotient for each variable, we next perform dual interval division (described in Section 6.1). The terms $\frac{x_1 - 0.5\cdot\alpha}{1-\alpha}$ and $\frac{x_2 - 0.5\cdot\alpha}{1-\alpha}$ ultimately evaluate to $[0.9, 1.05]+[0.4, 0.62]\epsilon$ and $[0.15, 0.22] + [\text{-}0.43, \text{-}0.25]\epsilon$, respectively. Dual interval division implicitly encodes the quotient rule of differentiation.

Non-Differentiable Functions. Upon computing $\frac{x_1 - 0.5\cdot\alpha}{1-\alpha}$ and $\frac{x_2 - 0.5\cdot\alpha}{1-\alpha}$, we now must take the min of each with $[1,1] + [0,0]\epsilon$. This is highly challenging as min is not differentiable. To resolve this, we must use a more general notion of differentiation, specifically the Clarke Jacobian [Clarke 1990], which has also recently emerged in the programming languages literature [Di Gianantonio and Edalat 2013; Sherman et al. 2021]. The Clarke Jacobian can compute a generalized derivative for non-differentiable, but Lipschitz functions like min, max, ReLU, abs, and even functions defined by conditional branching statements (provided one proves such functions are continuous and piecewise differentiable). The Clarke Jacobian does so by returning a *convex set of points* as the "derivative" instead of just a single point. To perform the Clarke differentiation through the min function, one takes the standard derivative of whichever of the min function's two arguments

attains the minimum. The caveat is that if both arguments attain the minimum, one must take the *convex hull* of both arguments' derivatives. However, this becomes even more difficult for the interval domain, as due to the inherent uncertainty, when two intervals overlap, it is possible that either could attain the minimum. Thus, if the real parts of two dual intervals overlap, our analysis must take the convex hull of their respective dual parts.

Recall that for x_1 we must take the min of $[1,1] + [0,0]\epsilon$ and $\frac{x_1 - 0.5\cdot\alpha}{1-\alpha} = [0.9, 1.05] + [0.4, 0.62]\epsilon$. In this case, the real parts overlap, so we must take the convex hull (denoted by **co**) of their respective dual parts: $[0,0]$ and $[0.4, 0.62]$, as shown in Eq. 2.

$$\begin{aligned} \min\left(1, \frac{x_1 - 0.5\cdot\alpha}{1-\alpha}\right) &= \min([1,1] + [0,0]\epsilon, [0.9, 1.05] + [0.4, 0.62]\epsilon) \\ &= \min([1,1], [0.9, 1.05]) + \mathbf{co}([0,0], [0.4, 0.62])\epsilon \\ &= [0.9, 1] + [0, 0.62]\epsilon \end{aligned} \tag{2}$$

For x_2, when computing the min of $[1,1] + [0,0]\epsilon$ and $\frac{x_2 - 0.5\cdot\alpha}{1-\alpha} = [0.15, 0.22] + [-0.43, -0.25]\epsilon$, the real parts do not intersect, thus the result is exactly just $[0.15, 0.22] + [-0.43, -0.25]\epsilon$.

Finally, as the contrast variation perturbation is also composed with the max function, we must repeat the same procedure, albeit with max instead of min. For x_1 and x_2, we compute:

$$x_1' = \max([0,0] + [0,0]\epsilon, [0.9, 1] + [0, 0.62]\epsilon) = [0.9, 1] + [0, 0.62]\epsilon$$

$$x_2' = \max([0,0] + [0,0]\epsilon, [0.15, 0.22] + [-0.43, -0.25]\epsilon) = [0.15, 0.22] + [-0.43, -0.25]\epsilon$$

Propagation through the Network. Upon computing the perturbed inputs x_1' and x_2', we propagate their abstracted values through the network itself. For simplicity in presentation, we give each network node a fresh variable name (f_3-f_6). To compute the value of f_3, we first multiply x_1' and x_2' by the corresponding edge weights using dual interval multiplication and sum incoming terms, resulting in the dual interval $[1.05, 1.22] + [-0.43, 0.37]\epsilon$. Then, we apply the dual interval lifting of tanh to the argument $[1.05, 1.22] + [-0.43, 0.37]\epsilon$. The dual interval lifting of tanh applies the interval lifting of tanh (where $\tanh([a,b]) = [\tanh(a), \tanh(b)]$) to the real part of its argument, then applies the interval lifting of the closed-form derivative $(1 - \tanh^2)$ to the dual part.

$$\begin{aligned} f_3 = \tanh([1.05, 1.22] + [-0.43, 0.37]\epsilon) &= \tanh([1.05, 1.22]) + \Big((1 - \tanh^2([1.05, 1.22])) \cdot [-0.43, 0.37]\Big)\epsilon \\ &= [0.78, 0.84] + [-0.17, 0.14]\epsilon \end{aligned}$$

The dual part of the result is also multiplied by the dual part of the input ($[-0.43, 0.37]$), implicitly encoding the chain rule. In this example, f_3 evaluates to $[0.78, 0.84] + [-0.17, 0.14]\epsilon$.

Likewise for f_4, the input to the tanh function is the sum of x_1' and x_2' scaled by the edge weights, which is just $[0.67, 0.85] + [0.25, 1.05]\epsilon$. Hence, f_4 evaluates to $[0.59, 0.69] + [0.13, 0.68]\epsilon$. As our analysis is fully compositional, we easily repeat this procedure for the subsequent (final) layer. Multiplying f_3 and f_4 by their respective edge weights, then passing these values to the tanh activations, allows us to determine that $f_5 = [0.88, 0.91] + [-0.01, 0.19]\ \epsilon$ and $f_6 = [0.09, 0.24] + [-0.84, 0.012]\epsilon$.

Interval Clarke Jacobian. Upon computing the outputs f_5 and f_6, we take their dual parts as the Interval Clarke Jacobian. This is because we show in Section 6.3 that abstractly evaluating functions with dual intervals is equivalent to computing the Interval Clarke Jacobian, which in turn soundly over-approximates the true Clarke Jacobian. Since we are modeling the composition of the network and the perturbation as a function of only α (while holding all other inputs fixed), the Interval Clarke Jacobian is a 2×1 interval matrix, as the network has 2 outputs. In this case, it is $[[-0.01, 0.19], [-0.84, 0.012]]^T$.

Practical Applications. Upon computing this over-approximation of the Clarke Jacobian, we can use it for several practical applications. For instance, we can compute the local Lipschitz constant in the region $\alpha \in [0, 0.1]$ by taking the maximum norm of the Interval Clarke Jacobian, which intuitively gives us a point summary of the network's sensitivity to perturbations by α in this local region. For this example, the Lipschitz constant (with respect to the ℓ_∞-norm) evaluates to 0.84.

We can also use the over-approximation of the Clarke Jacobian to analyze the local landscape of the composition of the network and the perturbation for stationary points. If a point is a local extremum, then the Clarke Jacobian at that point contains $\mathbf{0}$. Therefore, if any entry of the Interval Clarke Jacobian does not contain 0, it certifies that no point in the input range is a local extremum. In this example, as both entries contain 0, the analysis determines that the input region $\alpha \in [0, 0.1]$ could still contain a local extremum.

3 PRELIMINARIES

We define all the mathematical preliminaries needed to describe automatic differentiation, as well as the Clarke Jacobian. We start with the definition of the standard Jacobian.

Definition 3.1. The Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ differentiable at a point $\mathbf{x_0} \in \mathbb{R}^m$ is

$$\mathbf{J}(f, \mathbf{x_0}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} |_{x=\mathbf{x_0}} & \cdots & \frac{\partial f_1}{\partial x_m} |_{x=\mathbf{x_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} |_{x=\mathbf{x_0}} & \cdots & \frac{\partial f_n}{\partial x_m} |_{x=\mathbf{x_0}} \end{bmatrix}$$

When $m = n = 1$, the Jacobian is merely the classical derivative: $\mathbf{J}(f, \mathbf{x_0}) = \frac{df}{dx}|_{x=\mathbf{x_0}}$.

Dual Numbers. The question then arises, how do we automatically compute this Jacobian? The most popular method is via *Automatic Differentiation*, or AD. AD has two modes: reverse and forward. The former recursively evaluates derivatives of sub-expressions, and can be thought of as a generalization of *backpropagation*. We focus on the latter, as forward-mode automatic differentiation is much easier to implement and excels when the function's input dimension is small (as will be in our use cases). To implement forward-mode automatic differentiation, one may overload all primitive arithmetic operations to work on *dual numbers*, which we now describe.

Definition 3.2. Dual numbers are numbers of the form $a + b\epsilon$, where $a, b \in \mathbb{R}$ and ϵ is a symbolic variable (akin to i for imaginary numbers). We denote the set of all dual numbers as $\mathbb{D}$. Dual number arithmetic is given by the following rules:

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \cdot (c + d\epsilon) = (ac) + (ad + bc)\epsilon$$

$$(a + b\epsilon)/(c + d\epsilon) = \left(\frac{a}{c}\right) + \left(\frac{bc - ad}{c^2}\right)\epsilon$$

The above arithmetic rules for dual numbers implicitly encode linearity, the product rule, and the quotient rule in the computation of their dual part. To access the real part of a dual number, we write $fst(a + b\epsilon) = a$, and likewise the dual part is accessed by $snd(a + b\epsilon) = b$. For any differentiable function $f : \mathbb{R} \to \mathbb{R}$, we use the standard lifting of f to dual numbers $f : \mathbb{D} \to \mathbb{D}$ given as

$$f(a + b\epsilon) = f(a) + (f'(a) \cdot b)\epsilon$$

Therefore, the dual part of a dual number corresponds to the value of the function's derivative evaluated at the real part, a. Further, multiplying the derivative $f'(a)$ by the existing dual part b implicitly encodes the chain rule of calculus.

3.1 Lipschitz Continuity

We subsequently show how to extend the previous concepts to non-differentiable but locally Lipschitz functions. Hence, we first define the local Lipschitz property.

Definition 3.3. A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is locally Lipschitz on $X \subseteq \mathbb{R}^m$ if there exists a positive constant $K^{\alpha,\beta} \in \mathbb{R}_{>0}$ such that for any $x_1, x_2 \in X$ we have

$$\|f(x_1) - f(x_2)\|_\beta \leq K^{\alpha,\beta}\|x_1 - x_2\|_\alpha$$

where $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ are arbitrary p-norms over $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. Furthermore, if for a given point $x_0 \in \mathbb{R}^m$, there exists a positive real $\delta > 0$ such that f is locally Lipschitz within a ball of radius δ centered at x_0, we say f *is Lipschitz near* x_0.

Lipschitz Constant. The constant $K^{\alpha,\beta}$ is called the (local) Lipschitz constant, which provides a formal bound on how much a function's output can change (measured by the $\|\cdot\|_\beta$ norm) given a change in input (measured by the $\|\cdot\|_\alpha$ norm). The constant can easily be obtained once one has the Jacobian $\mathbf{J}$. For a Lipschitz function $f : \mathbb{R}^m \to \mathbb{R}^n$ that is differentiable, one can compute the local Lipschitz constant on a region X by taking the maximum dual norm of the Jacobian over X:

$$K^{\alpha,\beta} = \sup_{x \in X} \|\mathbf{J}(f, x)\|_{\alpha,\beta}$$

where the dual norm of any matrix $M \in \mathbb{R}^{n \times m}$ is given as $\|M\|_{\alpha,\beta} = \sup_{\|v\|_\alpha \leq 1} \|Mv\|_\beta$. For common values of α and β, there is a closed-form expression for $\|M\|_{\alpha,\beta}$. For instance, $\|M\|_{1,1} = \max_{1 \leq j \leq m}(\sum_{i=1}^{n} |M_{i,j}|)$, thus we simply take the norm of the Jacobian (over all points in X) that has the maximum absolute column sum.

3.2 Clarke Generalized Jacobian

However, the following question arises: what if we need to compute the derivative at a point where it is not defined, such as $\mathrm{ReLU}(x)$ at $x = 0$? Can one extend Jacobians (and methods to compute them) to non-differentiable functions? We follow recent work by Sherman et al. [2021] and employ the notion of the Clarke Generalized Jacobian [Clarke 1990] for this extension. Intuitively, this generalizes the notion of a Jacobian to non-differentiable, but still Lipschitz-continuous functions.

Convexity. The Clarke Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ evaluates to a *convex* set of $n \times m$ matrices, hence we define the following operators. Let $\mathbf{Co}(\mathbb{R}^{n \times m})$ denote all convex sets of $n \times m$ real matrices. Further, let $\mathbf{co} : \mathcal{P}(\mathbb{R}^{n \times m}) \to \mathbf{Co}(\mathbb{R}^{n \times m})$ be the convex hull operator, which given a set of matrices, takes their convex hull. We can now define the Clarke Generalized Jacobian.

Definition 3.4. (Clarke Thm. 2.5.1 [Clarke 1990]) Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a function that is locally Lipschitz at $\mathbf{x_0} \in \mathbb{R}^m$, where the set of non-differentiable points of f has Lebesgue measure 0 (we denote that set as $S \subset \mathbb{R}^m$). The Clarke Generalized Jacobian of f at the point $\mathbf{x_0}$ is denoted with the following signature $\partial_c : (\mathbb{R}^m \to \mathbb{R}^n) \times \mathbb{R}^m \to \mathbf{Co}(\mathbb{R}^{n \times m})$, and is given as:

$$\partial_c(f, \mathbf{x_0}) = \mathbf{co}\{\lim_{j \to \infty} \mathbf{J}(f, \mathbf{x}_j) : \lim_{j \to \infty} \mathbf{x}_j = \mathbf{x_0} \textit{ and } \mathbf{x}_j \notin S \text{ for all } j \in \mathbb{N}\} \tag{3}$$

We first detail this definition when $\mathbf{x}_0$ is a point of non-differentiability of f. Intuitively, since we cannot compute the actual Jacobian $\mathbf{J}$ at a point of non-differentiability, we instead take any *sequence of points* $(\mathbf{x}_0)$ that converges to that point of non-differentiability, such that the Jacobian is defined for all points in that sequence; we then compute what the Jacobians evaluated at those points in the sequence converge to. The Clarke Jacobian is then just the convex hull over all such sequences' respective limiting Jacobians. Thus, we obtain a convex set of matrices. When the function *is* differentiable at the point $\mathbf{x}_0$, this limit reduces to exactly the standard Jacobian at

$$
\begin{array}{rcl}
f & ::= & f_1 \times f_2 \\
& | & f_1 + f_2 \\
& | & f_1 \cdot f_2 \\
& | & 1/f_1 \\
& | & C^1 \circ f_1 \\
& | & f_0 > c \; ? \; f_1 : f_2 \\
& | & x_i \\
& | & c \in \mathbb{R} \\
\\
C^1 & ::= & e^x \mid \sin(x) \mid \cos(x) \\
& | & \sigma(x) \mid \tanh(x) \mid \log(x) \\
& | & \text{sqrt}(x) \mid \text{softplus}(x) \; \ldots
\end{array}
$$

$$\frac{}{\Gamma \vdash c : \mathbb{R}^m \to \mathbb{R}} \qquad \frac{}{\Gamma \vdash x_i : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}^n \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}^p}{\Gamma \vdash f_1 \times f_2 : \mathbb{R}^m \to \mathbb{R}^{n+p}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R} \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash f_1 + f_2 : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R} \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash f_1 \cdot f_2 : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash 1/f_1 : \mathbb{R}^m \to \mathbb{R}} \qquad \frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash C^1 \circ f_1 : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_0 : \mathbb{R}^m \to \mathbb{R} \quad \Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}^n \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}^n}{\Gamma \vdash f_0 > c \; ? \; f_1 : f_2 : \mathbb{R}^m \to \mathbb{R}^n}$$

Fig. 3. Locally Lipschitz Function Syntax and Typing Rules

$\mathbf{x}_0$ (and the convex hull becomes superfluous), hence $\partial_c(f, \mathbf{x_0})$ will be a singleton set containing exactly $\mathbf{J}(f, \mathbf{x_0})$.

Example 3.5. Let $f(x) = \text{ReLU}(x)$. Then $\partial_c(f, 0) = \mathbf{co}\{0, 1\} = [0, 1]$.

4 LANGUAGE SYNTAX AND SEMANTICS

We describe our differentiable programming language, which is based upon λ_S [Sherman et al. 2021] but with additional branching primitives. Our language is first-order and purely functional (no side-effects), yet expressive enough to encode neural networks and locally Lipschitz perturbations.

4.1 Syntax

Figure 3 presents the syntax of the language. DeepJ syntactically supports standard arithmetic operations, differentiable function primitives, and limited branching for encoding Lipschitz but non-differentiable functions like min and max. One can encode neural networks in DeepJ; however, instead of encoding them as a series of edge-weight matrix multiplications, our syntax constructs them inductively via compositions: $\circ$, Cartesian products: $\times$, and branching (e.g., for ReLU networks).

Arithmetic Operations. We support all of the key arithmetic primitives: addition, multiplication, and division. While syntactically speaking, these are only defined for functions of a single output variable, one can easily encode multi-variable versions (e.g., vector addition) by also making use of the Cartesian product, $\times$, where $(f_1 \times f_2)(x) = (f_1(x), f_2(x))$. For example, $(f_1 \times f_2) + (f_3 \times f_4) = (f_1 + f_3) \times (f_2 + f_4)$.

Differentiable Functions. Syntactically, we support as primitives all the standard primitive differentiable functions (e.g., e^x, $\sin(x)$, $\tanh(x)$, etc.). Hence, we denote these as C^1 because each function is C^1-smooth, meaning the function is continuous and differentiable everywhere on its domain, and the first derivative is also continuous everywhere on its domain.

Non-differentiable, Lipschitz Functions. We support a branching primitive, $f_0 > 0 \; ? \; f_1 : f_2$, to implement Lipschitz, but not necessarily differentiable functions such as abs, ReLU, min, and max

$$\partial_c : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbb{R}^m\right) \to \mathbf{Co}\left(\mathbb{R}^{n\times m}\right)$$

$$\begin{aligned}
\partial_c(f_1 \times f_2, \mathbf{x_0}) &\subseteq \begin{bmatrix}\partial_c(f_1, \mathbf{x_0})\\ \partial_c(f_2, \mathbf{x_0})\end{bmatrix} \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz near } \mathbf{x_0}, \text{ else } \top \\
\partial_c(f_1 + f_2, \mathbf{x_0}) &\subseteq \partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0}) \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz near } \mathbf{x_0}, \text{ else } \top \\
\partial_c(f_1 \cdot f_2, \mathbf{x_0}) &\subseteq f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) +_c f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz near } \mathbf{x_0}, \text{ else } \top \\
\partial_c(1/f_1, \mathbf{x_0}) &\subseteq -\partial_c(f_1, \mathbf{x_0}) /_c f_1(\mathbf{x_0})^2 \\
&\quad \text{if } f_1 \text{ Lipschitz near } \mathbf{x_0} \text{ and } f_1(\mathbf{x_0}) \neq 0, \text{ else } \top \\
\partial_c(C^1 \circ f_1, \mathbf{x_0}) &= \mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot_c \partial_c(f_1, \mathbf{x_0}) \\
&\quad \text{if } f_1 \text{ Lipschitz near } \mathbf{x_0} \text{ and } C^1 \text{ differentiable at } f_1(\mathbf{x_0}), \text{ else } \top \\
\partial_c(f_0 > c \;?\; f_1 : f_2, \mathbf{x_0}) &\subseteq \begin{cases}\partial_c(f_1, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) > c \\ \partial_c(f_2, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) < c \\ \mathbf{co}\left(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0})\right) & \text{otherwise}\end{cases} \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz near } \mathbf{x_0} \text{ and} \\
&\quad f_1, f_2 \text{ agree on } \{x : f_0(x) = c\}, \text{ else } \top \\
\partial_c(x_i, \mathbf{x_0}) &= \{\mathbf{e}_i\} \\
&\quad \text{provided } i \in \{1, ..., m\} \\
\partial_c(c, \mathbf{x_0}) &= \{\mathbf{0}\}
\end{aligned}$$

Fig. 4. Clarke Jacobian Rules

(and by extension max-pooling). For example, abs can be implemented as $x > 0 \;?\; x : -x$. However, one could easily define a discontinuous function such as $x > 0 \;?\; 1 : 0$ using our syntax. Thus, for the computed Jacobian (and by extension Lipschitz constant) to be semantically meaningful and valid, we restrict the type of branching we support. We will later show that checking for these restrictions is undecidable in Section 4.2, and thus the responsibility of ensuring that the restrictions are satisfied rests upon the programmer or an (incomplete) program analysis.

Lastly, even though we restrict the type of branches we support, more complex branches with arbitrary Boolean predicates can be systematically desugared into simpler ones. For example, the branch $\left(c_1 < x \wedge x < c_2\right) \;?\; f_1 : f_2$ can be desugared into $c_1 < x \;?\; \left(x < c_2 \;?\; f_1 : f_2\right) : f_2$. Disjunctions and negations of Booleans can be encoded similarly.

Type System. As our language only employs real-valued functions, the typing rules for a function are simple and based on standard real-valued arithmetic. For instance, when adding two functions, their dimensions must agree. Likewise, when dividing by a function f_1 (as in $1/f_1$), the output dimension of the function f_1 must be 1. The full typing rules can be seen in Fig. 3, where Γ corresponds to the typing context that maps all arithmetic expressions (including intermediate ones) to their types.

4.2 Standard Interpretation

As our language is an augmented differentiable programming language, the semantic interpretation of a function f is its derivative, which in our case corresponds to its Clarke Generalized Jacobian $\partial_c(f, \cdot)$. Our language is inspired by Sherman et al. [2021], hence we follow their convention of lifting the Clarke Jacobian to become a *total* function by defining the result to be $\top$ whenever ∂_c would be undefined, such as trying to evaluate the Clarke Jacobian of $\log(x)$ at $x = 0$. Because of this, $\top$ corresponds to the entire space of all real $n \times m$ matrices ($\mathbb{R}^{n\times m}$).

The semantic rules for recursively defining the Clarke Generalized Jacobian of each language primitive are shown in Figure 4. Unlike the regular Jacobian, these rules use $\subseteq$ instead of equality. This means that the *exact* Clarke Jacobian is not computable; however, our end goal is to compute a sound over-approximation.

Convex Arithmetic Operations. We denote $+_c : \mathbf{Co}(\mathbb{R}^{n\times m}) \times \mathbf{Co}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ to be the addition of two convex sets (Minkowksi addition), where $A +_c B = \{a + b : a \in A, b \in B\}$. Likewise, we denote $\cdot_c : \mathbb{R} \times \mathbf{Co}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ where $v \cdot_c A = \{v \cdot a : a \in A\}$ and $/_c \colon \mathbf{Co}(\mathbb{R}^{n\times m}) \times \mathbb{R}_{\neq 0} \to \mathbf{Co}(\mathbb{R}^{n\times m})$ where $A/_c\, v = \{\frac{1}{v} \cdot a : a \in A\}$ to be convex scalar multiplication and division, respectively. Lastly, as mentioned, we let $\mathbf{co} : \mathcal{P}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ be the convex hull operator that takes the convex hull of a set of matrices. We now detail the rules of Fig. 4.

Variables and Constants. The Clarke Jacobian of a single variable x_i is $\{1\}$, but if we compute it with respect to $\mathbf{x_0} \in \mathbb{R}^m$, the Clarke Jacobian will be m-dimensional, hence denoted as $\{\mathbf{e}_i\}$, where the i^{th} entry of $\mathbf{e}_i$ is 1 and all other $m - 1$ entries are 0. Equivalently, $f \triangleq x_i$ can be thought of as a projection function going from $\mathbb{R}^m \to \mathbb{R}$ that takes the i^{th} component of $\mathbf{x_0}$. Similarly, the Clarke Jacobian of any constant is just the vector of m zeroes, denoted as $\{\mathbf{0}\}$.

Cartesian Product. The Clarke Jacobian of the Cartesian product of two functions (defined over the same input) is a *subset* of the matrix concatenation of each function's respective Clarke Jacobians. This follows directly from proposition 2.6.2 (e) of Clarke [1990].

Addition. The Clarke Jacobian does not obey exact linearity; however, the Clarke Jacobian of the sum of two functions is *contained* in the Minkowski sum of each function's respective Clarke Jacobian. This rule follows directly from Proposition 2.3.2 of Clarke [1990].

Multiplication and Division. The Clarke Jacobian follows both a product and quotient rule, but as with the other rules, the relationship is of containment instead of strict equality. These follow directly from Propositions 2.3.13 and 2.3.14 of Clarke [1990].

Composition. To ensure our language is fully compositional, we can exploit the fact that the Clarke Jacobian follows a chain rule, when the outermost function is C^1-smooth. This result follows directly from Theorems 2.3.9 and 2.6.6 of Clarke [1990].

C^1 Functions. For the C^1 primitive functions, the Clarke Jacobian reduces to the standard Jacobian $\mathbf{J}(C^1, \cdot)$. Furthermore, one can catch erroneous behavior by leveraging knowledge about the known domain of each C^1 primitive function. For instance, if one tries to evaluate $\partial_c(\text{sqrt}(x_i), -1)$, this would require evaluating $\mathbf{J}(\text{sqrt}, -1)$. However, since -1 lies outside the domain of sqrt, this result is undefined; thus, the expression will evaluate to $\top$, and errors will be caught at this step. One can then propagate $\top$ up the remainder of a function's expression tree, since all other rules in Fig. 4 first check if any sub-expression evaluates to $\top$ (in which case they will also evaluate to $\top$).

Branching. Branching is absent in Clarke's original formulation. Furthermore, λ_S [Sherman et al. 2021] and Di Gianantonio and Edalat [2013] also do not support a branching primitive. This is

because a branch can introduce the possibility of encoding a discontinuous function, such as $f(x) = x > 0\ ?\ 1 : 0$. As a branch can be thought of as "splitting" a function into two separate pieces, we need to ensure that the entire function is still locally Lipschitz continuous (on the region of interest containing $\mathbf{x_0}$) for it to have a well-defined Clarke Jacobian at $\mathbf{x_0}$. Thus, to formally establish the necessary conditions for the well-definedness of the Clarke Jacobian of a branching function, we use results from the theory of piecewise differentiable functions [Khan and Barton 2013; Scholtes 2012]. We first state a useful lemma.

Lemma 4.1. *Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a function expressible using the syntax of Figure 3 that does not contain branches. For any $\mathbf{x_0}$ where $\mathbf{J}(f, \mathbf{x_0})$ is defined, we have that $\partial_c(f, \mathbf{x_0}) = \{\mathbf{J}(f, \mathbf{x_0})\}$.*

Proof. (Sketch) Since a function that does not have branches is a constant or only uses addition, composition with a C^1 function, multiplication, division, or the Cartesian product, one can directly compute $\mathbf{J}(f, \mathbf{x_0})$ using linearity, chain rule, product or quotient rule, or the direct computation of the derivative (for C^1 functions and constants), provided $\mathbf{J}(f, \mathbf{x_0})$ is well-defined (e.g., there is no division by 0). Furthermore, by Proposition 2.2.4 of Clarke [Clarke 1990], if a function has a well-defined Jacobian at a point, the Clarke Jacobian reduces to that value. □

We now formally define piecewise differentiability.

Definition 4.2. (Piecewise Differentiability [Scholtes 2012]) A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is piecewise differentiable on an open set $\mathcal{X} \subseteq \mathbb{R}^m$ if f is continuous on $\mathcal{X}$ and for every $x \in \mathcal{X}$ there exists an open neighborhood $O \subseteq \mathcal{X}$ and a finite number of differentiable functions, denoted $\{g_1, .., g_k\}$, such that for any $x_o \in O$, $f(x_o) \in \{g_1(x_o), ..., g_k(x_o)\}$. We will refer to the set of differentiable functions $\{g_1, ..., g_k\}$ as the selection set.

Any function f that is differentiable on a set $\mathcal{X} \subseteq \mathbb{R}^m$ (meaning $\mathbf{J}(f, \cdot)$ exists) is trivially piecewise differentiable on $\mathcal{X}$, albeit with a single piece. A piecewise differentiable function has a well-defined Clarke Jacobian that can be given in terms of the convex hull of the standard Jacobian of constituent pieces, provided an *active set* is known a priori [Scholtes 2012]. If f has the selection set of differentiable functions $\{g_1, ..., g_k\}$, then (by Proposition 4.3.1 of Scholtes [2012]):

$$\partial_c(f, \mathbf{x_0}) = \mathbf{co}\{\mathbf{J}(g_i, \mathbf{x_0}) \mid i \in A(f, \mathbf{x_0})\} \tag{4}$$

where $A(f, \mathbf{x_0}) \subseteq \{1, .., k\}$ is the *active set*, which denotes at $\mathbf{x_0}$ which of the k selection functions satisfy $g_i(\mathbf{x_0}) = f(\mathbf{x_0})$. To adapt this to our setting, we start with the simplest branching function $f \triangleq f_0 > c\ ?\ f_1 : f_2$ where f_1, f_2 are branch-free (meaning they are compositions, sums, or products of C^1 functions). Since f_1 and f_2 do not contain branches, one can compute $\mathbf{J}(f_i, \mathbf{x_0})$ directly in accordance with Lemma 4.1. Hence, f_1 and f_2 *are* the selection set, as they are differentiable (by assumption) and the value of $f_0 > c\ ?\ f_1 : f_2$ for some x_0 will necessarily be in $\{f_1(x_0), f_2(x_0)\}$. Therefore, when writing a branch, the selection set is known *by construction*. So all that remains for f to be piecewise differentiable (and have a well-defined ∂_c) is to ensure that it is continuous, which is true provided $f_1(x) = f_2(x)$ for $\{x : f_0(x) = c\}$.

We do not need to restrict to cases where f_1 and f_2 are branch-free. We can nest branches arbitrarily deeply, provided they agree on the decision boundary $\{x : f_0(x) = c\}$, as all this does is increase the number of selection functions (or pieces) by finitely many. This is because we do not allow infinite recursion or while loops that would permit expressing countably infinite possible branches. Intuitively, we recursively "unroll" the nested branches into all possible innermost functions (which themselves will no longer contain branches). This ultimately yields a finite set of branch-free functions f_i for which we can compute $\mathbf{J}(f_i, \cdot)$ directly (as in Lemma 4.1). Using these notions, we can now formally describe necessary conditions for the Clarke Jacobian of the branching primitive to be well-defined.

Theorem 4.3. *(Well-Definedness of the Clarke Jacobian of a Branch) The function given by the branch $f_0 > c\ ?\ f_1 : f_2$ is piecewise differentiable on an open set $\mathcal{X} \subseteq \mathbb{R}^m$ if f_1 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \geq c\}$, f_2 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \leq c\}$, and $f_1(x_0) = f_2(x_0)$ for all $x_0 \in \{x \in \mathcal{X} : f_0(x) = c\}$.*

Proof. (Sketch) Since f_1 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \geq c\}$, it has some selection set $\{g_1, ..g_k\}$, hence $f_0 > c\ ?\ f_1 : f_2$ is also piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \geq c\}$, with the same selection set on that region. Likewise, since f_2 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \leq c\}$, it has some selection set $\{h_1, ..., h_l\}$ thus $f_0 > c\ ?\ f_1 : f_2$ is also piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \leq c\}$, with the same selection set. Furthermore, since $\{x \in \mathcal{X} : f_0(x) = c\} \subseteq \{x \in \mathcal{X} : f_0(x) \geq c\}$ and $\{x \in \mathcal{X} : f_0(x) = c\} \subseteq \{x \in \mathcal{X} : f_0(x) \leq c\}$, the selection set for $\{x \in \mathcal{X} : f_0(x) = c\}$ is $\{g_1, ..g_k, h_1, ..., h_l\}$. Lastly, since $x_0 \in \{x \in \mathcal{X} : f_0(x) = c\}$ implies $f_1(x_0) = f_2(x_0)$, then $f_0 > c\ ?\ f_1 : f_2$ is continuous on its entire domain. □

We now present the rule for computing the Clarke Jacobian for a branching function, using the notions of piecewise differentiability. Formally, for $f_0 : \mathbb{R}^m \to \mathbb{R}$, $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}^n$, and $\mathbf{x_0} \in \mathbb{R}^m$:

$$\partial_c(f_0 > c\ ?\ f_1 : f_2, \mathbf{x_0}) \subseteq \begin{cases} \partial_c(f_1, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) > c \\ \partial_c(f_2, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) < c \\ \mathbf{co}\Big(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0})\Big) & \text{otherwise} \end{cases} \tag{5}$$

When $f_0(\mathbf{x_0}) > c$, the active set is $A(f_0, \mathbf{x_0}) = \{f_1\}$; similarly, when $f_0(\mathbf{x_0}) < c$, the active set is $A(f_0, \mathbf{x_0}) = \{f_2\}$. Along the decision boundary, both f_1 and f_2 are in the active set, hence the Clarke Jacobian is the convex hull, **co**, of both of their respective Clarke Jacobians. In the rule shown in Eq. 5, we evaluate $\partial_c(f_1, \mathbf{x_0})$ and $\partial_c(f_2, \mathbf{x_0})$ instead of $\mathbf{J}(f_1, \mathbf{x_0})$ and $\mathbf{J}(f_2, \mathbf{x_0})$ as in Eq. 4. This recursive definition allows us to capture the notion of unrolling a nested branch, as once f_1 and f_2 are themselves branch free, $\partial_c(f_i, \mathbf{x_0})$ and $\mathbf{J}(f_i, \mathbf{x_0})$ become equivalent (by Lemma 4.1); thus, this equation would coincide with Eq. 4.

Example 4.4. We can encode $\text{ReLU}(x) \triangleq x > 0\ ?\ x : 0$, and hence $\partial_c(x > 0\ ?\ x : 0, 0) = \mathbf{co}(\partial_c(x, 0), \partial_c(0, 0)) = \mathbf{co}(\{1, 0\}) = [0, 1]$.

Despite the elegant theory of Scholtes [2012] allowing us to characterize the conditions and well-definedness of the Clarke Jacobian of a piecewise differentiable function (or branch in our language), the problem of *statically checking* that these conditions are satisfied is undecidable. This is because even the smaller problem of ensuring that a branch is continuous along the decision boundary is known to be undecidable [Chaudhuri et al. 2010; Di Gianantonio and Edalat 2013; Griewank 2013]. As those works do not target the exact framework we focus on, we offer a short self-contained result:

Lemma 4.5. *(Undecidability of Ensuring Piecewise Differentiability) Let $f_1 : \mathbb{R}^m \to \mathbb{R}^n$ and $f_2 : \mathbb{R}^m \to \mathbb{R}^n$ be arbitrary functions in the language, and let $\mathcal{X} \subseteq \mathbb{R}^m$ be an arbitrary set on which f_1 and f_2 are both defined. Checking if $f_1(x) = f_2(x)$ for all $x \in \mathcal{X}$ where $f_0(x) = c$ is undecidable.*

Proof. (Sketch) Reduction using Richardson's Theorem [Richardson 1969]. □

Existing works can heuristically check for this condition using SMT solvers [Chaudhuri et al. 2010], or restrict the language to only pre-specified operators (e.g., max, min, ReLU and abs) [Beck and Fischer 1994; Di Gianantonio and Edalat 2013; Sherman et al. 2021]. Despite the difficulties introduced by a branching primitive, it will prove immensely useful in making the static analysis more precise, particularly for functions such as bilinear interpolation.

5 INTERVAL CLARKE JACOBIAN

Having now defined the syntax and the standard Clarke Jacobian, we now formalize a computable, sound over-approximation: the Interval Clarke Jacobian, ∂^{Int}. We will then show how to scalably implement our abstraction with an equivalent formulation that leverages a sound abstraction of forward-mode automatic differentiation.

5.1 Interval Domain

To soundly approximate the Clarke Jacobian, we use the interval domain as it is fully computable and can soundly abstract the convex set of matrices corresponding to the original Clarke Generalized Jacobian, since interval matrices are necessarily convex sets.

Preliminaries. Denote the set of real-valued intervals of the form $[a, b]$ where $a, b \in \mathbb{R} \cup \{\pm\infty\}$ and $a \leq b$ as $\mathbb{IR}$. The set of $n \times m$ matrices of intervals is denoted as $\mathbb{IR}^{n\times m}$. We will use the notation $\widehat{\mathbf{x_0}}$ instead of $\mathbf{x_0}$ to distinguish matrices and vectors whose entries are intervals instead of scalars. Similarly, to denote the evaluation of a function f where all of its operations are lifted to interval arithmetic, we will write $f(\widehat{\mathbf{x_0}})$. To denote the lower and upper bounds of an interval $\widehat{\mathbf{x_0}} = [a, b]$, we will write $lb(\widehat{\mathbf{x_0}})$ and $ub(\widehat{\mathbf{x_0}})$, respectively. We denote $+_{\mathbb{IR}}$, $\cdot_{\mathbb{IR}}$, and $/_{\mathbb{IR}}$ to be the integer arithmetic versions of addition, multiplication, and division, respectively. Likewise, we denote $\sqcup : \mathbb{IR}\times\mathbb{IR} \to \mathbb{IR}$ to be the standard interval join, which returns the smallest interval enclosing both arguments. We may also apply $\sqcup$ element-wise to matrices in $\mathbb{IR}^{n\times m}$. Lastly, we denote $\top = \mathbb{IR}^{n\times m}$, thus $\top$ is the entire space of $n \times m$ interval matrices. We now define the Interval Clarke Jacobian ∂^{Int}.

Definition 5.1. The Interval Clarke Jacobian $\partial^{Int} : \left((\mathbb{R}^m \to \mathbb{R}^n) \times \mathbb{IR}^m\right) \to \mathbb{IR}^{n\times m}$ for a function $f : \mathbb{R}^m \to \mathbb{R}^n$ and interval vector $\widehat{\mathbf{x}} \in \mathbb{IR}^m$ is denoted $\partial^{Int}(f, \widehat{\mathbf{x_0}})$ and is given by the rules of Fig. 5.

The interpretation of the Jacobian of a function f is now an interval matrix that over-approximates the convex set of matrices corresponding to the original Clarke Jacobian. This will be necessary to be able to define the local neighborhood of points for which we want to abstractly analyze or compute a Lipschitz constant. We now detail each abstract transformer.

Constants and Variables. The Interval Clarke Jacobian of a constant function is nearly identical to the respective Clarke Jacobian (in that ∂^{Int} of a constant is 0); it is lifted to become the constant interval $[0, 0]$ of the same dimension as $\widehat{\mathbf{x_0}}$, which we denote as $\widehat{\mathbf{0}} \in \mathbb{IR}^m$. Likewise, the Interval Clarke Jacobian of a single variable x_i is also the constant vector where the i^{th} component is 1 and all other $m - 1$ components are 0, again lifted to become a constant interval, which we denote as $\widehat{\mathbf{e}}_i \in \mathbb{IR}^m$.

Cartesian Product. The Interval Clarke Jacobian of a Cartesian product of functions is just the concatenation of their respective Interval Clarke Jacobians.

Addition. The Interval Clarke Jacobian obeys linearity exactly.

Multiplication and Division. The Interval Clarke Jacobian satisfies a lifted interval arithmetic version of the product and quotient rules.

Composition. For compositions with C^1-smooth functions, the Interval Clarke Jacobian satisfies a chain rule as well, where $\mathbf{J}^{Int}$ is the classical Jacobian of the C^1-smooth function, just interpreted using interval arithmetic instead of ordinary arithmetic.

Example 5.2. $\mathbf{J}^{Int}(\sin, [x_0^\ell, x_0^u]) = \cos([x_0^\ell, x_0^u])$ and $\mathbf{J}^{Int}(\tanh, [x_0^\ell, x_0^u]) = [1, 1] -_{\mathbb{IR}} [\tanh(x_0^\ell), \tanh(x_0^u)]^2$.

Just as with ∂_c, primitive C^1 functions are where we can check if the Jacobian is undefined (in which case ∂^{Int} evaluates to $\top$), which we likewise propagate up the function's expression tree.

$$\partial^{Int} : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbb{IR}^m\right) \to \mathbb{IR}^{n \times m}$$

$$\begin{aligned}
\partial^{Int}(f_1 \times f_2, \widehat{\mathbf{x_0}}) &= \begin{bmatrix} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\ \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) \end{bmatrix} \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}}) &= \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
\partial^{Int}(f_1 \cdot f_2, \widehat{\mathbf{x_0}}) &= f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) +_{\mathbb{IR}} f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
\partial^{Int}(1/f_1, \widehat{\mathbf{x_0}}) &= -\partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \,/_{\mathbb{IR}} f_1(\widehat{\mathbf{x_0}})^2 \\
&\quad \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and } 0 \notin f_1(\widehat{\mathbf{x_0}}), \text{ else } \top \\
\partial^{Int}(C^1 \circ f_1, \widehat{\mathbf{x_0}}) &= \mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}})) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\
&\quad \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and } C^1 \text{ differentiable on } f_1(\widehat{\mathbf{x_0}}), \text{ else } \top \\
\partial^{Int}(f_0 > c \,?\, f_1 : f_2, \widehat{\mathbf{x_0}}) &= \begin{cases} \partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}})) & \text{if } lb(f_0(\widehat{\mathbf{x_0}})) > c \\ \partial^{Int}(f_2, [\![f_0 < c]\!](\widehat{\mathbf{x_0}})) & \text{if } ub(f_0(\widehat{\mathbf{x_0}})) < c \\ \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})) & \text{otherwise} \end{cases} \\
&\quad \text{if } f_1 \,\&\, f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and} \\
&\quad f_1(x) = f_2(x) \text{ for } x \in \{x' : f_0(x') = c \wedge x' \in \widehat{\mathbf{x_0}}\}, \text{ else } \top \\
\partial^{Int}(x_i, \widehat{\mathbf{x_0}}) &= \widehat{\mathbf{e}}_i \\
\partial^{Int}(c, \widehat{\mathbf{x_0}}) &= \widehat{\mathbf{0}}
\end{aligned}$$

Fig. 5. Interval Clarke Jacobian Rules

Branching. If the lower bound of $f_0(\widehat{\mathbf{x_0}})$ is larger than c, we definitively know that f_1 is the only possible function of the active set. Similarly, if the upper bound of $f_0(\widehat{\mathbf{x_0}})$ is smaller than c, we definitively know that f_2 is the only possible function of the active set. If $c \in f_0(\widehat{\mathbf{x_0}})$, then it is possible that both f_1 and f_2 are in the active set, thus we have to consider both possibilities.

In either case, we can also refine our information about $\widehat{\mathbf{x_0}}$. For example, when evaluating ∂^{Int} of the function $f(x) \triangleq x > 0 \,?\, f_1 : f_2$ on the interval $\widehat{\mathbf{x_0}} = [-1, 1]$, conditional upon entering the true branch, we should only evaluate $\partial^{Int}(f_1, [0, 1])$. Likewise, conditional upon entering the false branch, we should only evaluate $\partial^{Int}(f_2, [-1, 0])$. We denote the refinement of $\widehat{\mathbf{x_0}}$ for a Boolean guard of the form $f_0 > c$ as $[\![f_0 > c]\!](\widehat{\mathbf{x_0}})$. However, the general problem of determining how to optimally refine the input $\widehat{\mathbf{x_0}}$, conditioned on the information of the branch $f_0 > c$, is undecidable. Therefore, we follow the approach of Miné [Miné 2017] and refine $\widehat{\mathbf{x_0}}$ when the function f_0 in the Boolean guard has a simple form: a single variable x_i, and then use the fallback transformer (the identity function) for all other cases (which can then be simplified to get the rule from Fig. 5):

$$[\![f_0 \; op \; c]\!](\widehat{\mathbf{x_0}}) = \begin{cases} \widehat{\mathbf{x_0}} \cap \left([-\infty, \infty] \times \ldots \times [-\infty, c] \times \ldots \times [-\infty, \infty]\right) & \text{if } f_0 = x_i \;\wedge\; op \in \{<, \leq\} \\ \widehat{\mathbf{x_0}} \cap \left([-\infty, \infty] \times \ldots \times [c, \infty] \times \ldots \times [-\infty, \infty]\right) & \text{if } f_0 = x_i \;\wedge\; op \in \{>, \geq\} \\ \widehat{\mathbf{x_0}} & \text{otherwise} \end{cases}$$

5.2 Soundness of the Abstraction

We now state the soundness of ∂^{Int}. We first define the concretization γ.

Definition 5.3. Define the concretization $\gamma : \mathbb{IR}^{n\times m} \to \mathbf{Co}\left(\mathbb{R}^{n\times m}\right)$ for an interval matrix $S \in \mathbb{IR}^{n\times m}$ as follows:

$$\gamma(S) = \{s \in \mathbb{R}^{n\times m} \mid s \in S\}$$

This is because an interval matrix is already by definition a convex set of matrices. The soundness theorem is now given as follows.

Theorem 5.4. *(Soundness) Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be any function constructible according to Fig. 3 and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbb{IR}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. Then*

$$\partial_c(f, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f, \widehat{\mathbf{x_0}}))$$

Proof. See Appendix A.2 (Laurel et al. [2022]). □

6 DUAL INTERVAL DOMAIN

While we could compute ∂^{Int} recursively in a reverse-mode pass, a key contribution of our work is to provide an equivalent, forward-mode version of the static analysis based on dual numbers.

6.1 Dual Interval Arithmetic

Definition 6.1. The set of dual intervals, denoted as $\mathbb{ID}$, are tuples of the form $[a, b] + [c, d]\epsilon$, where $a, b, c, d \in \mathbb{R} \cup \{\pm\infty\}$, $a \leq b$ and $c \leq d$. Intuitively, a dual interval represents a *set* of dual numbers where the real part is within $[a, b]$ and the dual part is within $[c, d]$. To access the real part $[a, b]$ of a dual interval, we will write $fst([a, b] + [c, d]\epsilon)$; to access the coefficients of the dual part $[c, d]$, we will write $snd([a, b] + [c, d]\epsilon)$. We will denote the set of m-dimensional vectors of dual intervals as $\mathbb{ID}^m$ and the set of $n \times m$ dimensional matrices as $\mathbb{ID}^{n\times m}$. Lastly, we denote $\top = \mathbb{ID}^{n\times m}$ (the entire space of dual interval matrices).

We can lift the ordinary arithmetic operators to dual intervals. We define dual interval addition, $+_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as follows:

$$\left([a, b] + [c, d]\epsilon\right) +_{\mathbb{ID}} \left([e, f] + [g, h]\epsilon\right) = \left([a, b] +_{\mathbb{IR}} [e, f]\right) + \left([c, d] +_{\mathbb{IR}} [g, h]\right)\epsilon$$

We define dual interval multiplication, $\cdot_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as

$$\left([a, b] + [c, d]\epsilon\right) \cdot_{\mathbb{ID}} \left([e, f] + [g, h]\epsilon\right) = \left([a, b] \cdot_{\mathbb{IR}} [e, f]\right) + \left([a, b] \cdot_{\mathbb{IR}} [g, h] +_{\mathbb{IR}} [c, d] \cdot_{\mathbb{IR}} [e, f]\right)\epsilon$$

And dual interval division, $/_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as

$$\left([a, b] + [c, d]\epsilon\right) /_{\mathbb{ID}} \left([e, f] + [g, h]\epsilon\right) = \left([a, b] /_{\mathbb{IR}} [e, f]\right) + \left(\left([c, d] \cdot_{\mathbb{IR}} [e, f] -_{\mathbb{IR}} [a, b] \cdot_{\mathbb{IR}} [g, h]\right) /_{\mathbb{IR}} [e, f]^2\right)\epsilon$$

It will also be useful to define a join for dual intervals, $\sqcup_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as

$$\left([a, b] + [c, d]\epsilon\right) \sqcup_{\mathbb{ID}} \left([e, f] + [g, h]\epsilon\right) = \left([a, b] \sqcup [e, f]\right) + \left([c, d] \sqcup [g, h]\right)\epsilon$$

6.2 Forward-Mode Abstract Evaluation with Dual Intervals

We now describe how to perform a forward-mode abstract evaluation where all operations are lifted to operate on dual intervals using the abstract interpreter $Eval_{\mathbb{ID}}$.

Definition 6.2. The abstract interpreter $Eval_{\mathbb{ID}} : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbb{ID}^m\right) \to \mathbb{ID}^n$ takes a real-valued function $f : \mathbb{R}^m \to \mathbb{R}^n$ and evaluates it abstractly by lifting the interpretation of all operations to dual interval arithmetic, as shown in Fig. 6.

Because our language is functional, there is no "state." However, the second argument to $Eval_{\mathbb{ID}}$ is the input, which serves the same purpose. We now detail the rules shown in Fig. 6.

$$Eval_{\mathbb{ID}} : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbb{ID}^m\right) \to \mathbb{ID}^n$$

$$
\begin{array}{rcl}
Eval_{\mathbb{ID}}(f_1 \times f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \times Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
& & \text{if } f_1 \ \& \ f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbb{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) +_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
& & \text{if } f_1 \ \& \ f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbb{ID}}(f_1 \cdot f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \cdot_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
& & \text{if } f_1 \ \& \ f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & \left([1,1] + [0,0]\epsilon\right) /_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
& & \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbb{ID}}(C^1 \circ f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & Eval_{\mathbb{ID}}(C^1, Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon)) \\
& & \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and } C^1 \text{ differentiable on } f_1(\widehat{\mathbf{x_0}}), \text{ else } \top \\
Eval_{\mathbb{ID}}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & \begin{cases} Eval_{\mathbb{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & lb(fst(Eval_{\mathbb{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon))) > c \\ Eval_{\mathbb{ID}}(f_2, [\![f_0 < c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & ub(fst(Eval_{\mathbb{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon))) < c \\ Eval_{\mathbb{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & \text{otherwise} \\ \quad \sqcup_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & \end{cases} \\
Eval_{\mathbb{ID}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & \widehat{\mathbf{x_0}}[i] + \widehat{\mathbf{y_0}}[i]\epsilon \\
Eval_{\mathbb{ID}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) & = & [c,c] + [0,0]\epsilon \\
Eval_{\mathbb{ID}}(C^1, [x_0^\ell, x_0^u] + [y_0^\ell, y_0^u]\epsilon) & = & C^1([x_0^\ell, x_0^u]) + \left(\mathbf{J}^{Int}(C^1, [x_0^\ell, x_0^u]) \cdot_{\mathbb{IR}} [y_0^\ell, y_0^u]\right)\epsilon \\
& & \text{if } C^1 \text{ differentiable on } [x_0^\ell, x_0^u], \text{ else } \top
\end{array}
$$

Fig. 6. Dual Interval Forward Mode Abstract Evaluation

Constants and Variables. The abstract evaluation of a constant is that constant, albeit abstracted to a (degenerate) dual interval. Likewise, the evaluation of a single variable x_i is the i^{th} element of the input dual interval (denoted by the $[i]$ accessor).

Cartesian Product. To abstractly evaluate the Cartesian product of two functions f_1 and f_2 on a given input $\widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon$, we abstractly evaluate each one, then take the Cartesian product of the results.

Addition. To abstractly evaluate the sum of two functions, we abstractly evaluate each on the given input $\widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon$, then take the dual interval sum $+_{\mathbb{ID}}$ of the respective results.

Multiplication and Division. As with addition, to abstractly evaluate the product of two functions or their quotient, we first abstractly evaluate the individual functions on the input, then use the dual interval forms of multiplication $\cdot_{\mathbb{ID}}$ and division $/_{\mathbb{ID}}$.

Composition. The abstract evaluation of the composition of any function f_1 with a C^1 function for an input dual interval $\widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon$, is the composition of the successive abstract evaluations: f_1 is abstractly evaluated on the input, then the C^1 function is abstractly evaluated on that result.

Branching. When evaluating a branch abstractly, *we only use the real part* of the dual interval, denoted $fst(Eval_{\mathbb{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon))$, to select which branch to take (or whether to abstractly evaluate both). While this may seem strange, one will note that in Eq. 5, the Clarke Jacobian is only computed for the piece that is ultimately chosen – f_1 or f_2 (possibly both) – and not the threshold function f_0. In fact, f_0 is only used to select which branch to take, hence its derivative information (which corresponds to its dual part) is unnecessary. If the lower bound $lb(fst(Eval_{\mathbb{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon)))$ is larger than c, we definitively know to take only the true branch. Conversely, if the upper bound of the real part $ub(fst(Eval_{\mathbb{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon)))$ is less than c, we only take the false branch. Otherwise, we abstractly evaluate both branches then take their join $\sqcup_{\mathbb{ID}}$. In all cases, we can refine the information of the real part $\widehat{\mathbf{x_0}}$ but not the dual part $\widehat{\mathbf{y_0}}$, as there is no way of knowing which sub-regions of $\widehat{\mathbf{x_0}}$ correspond to which sub-regions in $\widehat{\mathbf{y_0}}$, since the interval domain is non-relational.

6.3 Equivalence

We now prove that $Eval_{\mathbb{ID}}$ can be used to compute exactly the same result as ∂^{Int}.

Theorem 6.3. *(Equivalence of ∂^{Int} and $Eval_{\mathbb{ID}}$) Let $f : \mathbb{R}^m \to \mathbb{R}^n$ and let $\widehat{\mathbf{x_0}} \in \mathbb{IR}^m$. Then*

$$\partial^{Int}(f, \widehat{\mathbf{x_0}}) = snd\Big(Eval_{\mathbb{ID}}(f, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Proof. We show the sketch for select cases; the full proof is in Appendix A.4.

Base Cases. Constants and single variables are straightforward.

$$snd\Big(Eval_{\mathbb{ID}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) = [0,0] \times \ldots \times [0,0] = \partial^{Int}(c, \widehat{\mathbf{x_0}})$$

$$snd\Big(Eval_{\mathbb{ID}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) = [0,0] \times \ldots \times [1,1] \times \ldots \times [0,0] = \widehat{\mathbf{e}}_i = \partial^{Int}(x_i, \widehat{\mathbf{x_0}})$$

Arithmetic Operations. Equivalence for arithmetic primitives is straightforward, though it requires the inductive hypothesis and using the definitions of $+_{\mathbb{ID}}, \cdot_{\mathbb{ID}}, /_{\mathbb{ID}}$. We now detail addition:

$$\begin{aligned}
\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}}) &= \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) && \text{(Def.)}\\
&= snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) && \text{(Ind. Hyp.)}\\
&\quad +_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) &&\\
&= snd\Big(Eval_{\mathbb{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) && \text{(Def. of } +_{\mathbb{ID}}\text{, } \times \text{ distributes with } snd\text{)}
\end{aligned}$$

The case for the Cartesian product proceeds analogously. Composition also uses the inductive hypothesis and relies upon the fact that $snd\big(Eval_{\mathbb{ID}}(C^1, [x_0^l, x_0^u] + [1,1]\epsilon)^T\big) = \mathbf{J}^{Int}(C^1, [x_0^l, x_0^u])$.

Branching. As stated in Lemma 4.1, up to this point, without branching the language is just a standard differentiable programming language, thus the result is not surprising. However, with branching, establishing this equivalence is more challenging. We split the proof into three cases: (1) $lb(f(\widehat{\mathbf{x_0}})) > c$, (2) $ub(f(\widehat{\mathbf{x_0}})) < c$, and (3) $c \in f(\widehat{\mathbf{x_0}})$. Starting with $lb(f_0(\widehat{\mathbf{x_0}})) > c$:

$$\partial^{Int}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))$$

And for any $i \in \{1, .., m\}$, $lb(f_0(\widehat{\mathbf{x_0}})) > c$ iff $lb(fst(Eval_{\mathbb{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}_i}\epsilon))) > c$, hence:

$$Eval_{\mathbb{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}_i}\epsilon) = Eval_{\mathbb{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}_i}\epsilon)$$

By induction and the definitions of $Eval_{\mathbb{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}_i}\epsilon)$ and $\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))$:

$$\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}})) = snd\Big(Eval_{\mathbb{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}_1}\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}_m}\epsilon)^T\Big) \quad \text{(Ind.)}$$

$$\partial^{Int}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}}) = snd\Big(Eval_{\mathbb{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}_1}\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}_m}\epsilon)^T\Big) \quad \text{(Def.)}$$

The case where $ub(f_0(\widehat{\mathbf{x_0}})) < c$ proceeds exactly the same by symmetry. When we assume $c \in f_0(\widehat{\mathbf{x_0}})$, we must take the join. Starting from the desired right-hand side:

$$
\begin{aligned}
&snd\Big(Eval_{\mathbb{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) && \text{(Desired RHS)}\\
&= snd\Big(Eval_{\mathbb{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \sqcup_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots && \text{(Assump.)}\\
&\quad \ldots \times Eval_{\mathbb{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T \sqcup_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\\
&= snd\Big(Eval_{\mathbb{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\\
&\quad \sqcup snd\Big(Eval_{\mathbb{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times \ldots \times Eval_{\mathbb{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big) && \text{(Def. of } \sqcup_{\mathbb{ID}}\text{, } \times \text{ distributes with } snd\text{)}\\
&= \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})) = \partial^{Int}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}}) && \text{(Def. of } \partial^{Int} \text{ and Ind. Hyp.)}
\end{aligned}
$$

□

6.4 Complexity

The key insight of Theorem 6.3 is that, as with standard forward-mode AD, the dual interval abstraction requires as many forward passes as there are inputs. Therefore, for $f : \mathbb{R}^m \to \mathbb{R}^n$, we require m independent passes of $Eval_{\mathbb{ID}}$ in order to compute the full interval over-approximation of the Clarke Jacobian. However, they can be computed in parallel (as we do). Hence, as with standard forward-mode AD, our method is better suited for functions where the input dimension is small. Additionally, the real part for each pass will be the same, and need only be computed once.

Furthermore, for the basic arithmetic functions, dual interval arithmetic requires more primitive operations than the same operation defined over the reals. For example, multiplication of two dual intervals as shown in Section 6.1 requires 12 primitive multiplications to perform all the interval arithmetic multiplications for both the real and dual parts, as well as 2 primitive additions. However, for all arithmetic operations $\{+, -, \cdot, /\}$, the amount of additional primitive operations required for the dual interval version (vs. the real-valued version) is still just a constant factor more.

While it has been established that for standard AD, the amount of primitive operations of a single pass of forward-mode evaluation with dual numbers is at most a constant amount (typically $2 - 3$x) more than the number of operations to evaluate the function itself [Griewank and Walther 2008], this is not the case for our analysis. A key complexity issue arises due to our support of branching. As the abstract evaluator $Eval_{\mathbb{ID}}$ could potentially evaluate both branches of a conditional, we may have *exponentially* many more evaluations compared to evaluating the function with ordinary real numbers. Thus, if $OPS(f)$ denotes the number of primitive arithmetic operations to evaluate f (not its Jacobian) over real numbers, then $O(2^{OPS(f)})$ is the upper bound on the number of primitive arithmetic operations required to evaluate a single pass of the dual interval lifted version of f. For the evaluation, the ReLU function, as well as the contrast variation and rotation perturbations employ branching. Yet, despite the theoretically worst-case exponential operation complexity, in practice, DeepJ's implementation of $Eval_{\mathbb{ID}}$ was still quite fast. Lastly, each term will always have only two intervals associated to it (the real and dual parts), hence unlike affine interval arithmetic [De Figueiredo and Stolfi 2004] or tools like Rosa [Darulova and Kuncak 2017], the number of intervals we must track per variable does *not* grow as a function of the expression size.

7 METHODOLOGY

We next describe the experimental setup, including the perturbation functions and networks used in our experiments. We also detail the training procedures and accuracies of these networks. We ran our experiments on a 2.20 GHz 14 core Intel Xeon Gold 5120 CPU with 256 GB of main memory.

Implementation. We implemented our analysis in a tool called DeepJ, written in C++. DeepJ employs operator and function overloading to allow dual interval inputs to be propagated through arbitrary perturbation functions and neural networks. Even though our formalization is from a purely functional view, one can easily use an ANF conversion [Flanagan et al. 1993] to produce imperative code, since our programs do not have recursions or while loops. All code, neural networks, and results are available at https://github.com/uiuc-arc/DeepJ.

Perturbation Functions. We consider three previously studied image perturbations affecting pixel intensity and image geometry, as well as compositions of these perturbations. We assume that the images' pixel values are in the range $[0, 1]$.

(1) **Haze** [Paterson et al. 2021]: The intensity x_i of the i^{th} pixel in the original image is perturbed to $(1 - \alpha)x_i + \alpha$, where $\alpha = [0, \alpha_{max}] + [1, 1]\epsilon$; the parameter $\alpha_{max} \in [0, 1]$ represents the degree of haze.
(2) **Contrast** [Paterson et al. 2021]: The intensity x_i of the i^{th} pixel in the original image is perturbed to $\max(0, \min(1, \frac{x_i - 0.5 \cdot \alpha}{1-\alpha}))$, where $\alpha = [0, \alpha_{max}] + [1, 1]\epsilon$; the parameter $\alpha_{max} \in [0, 1]$ represents the degree of contrast.
(3) **Rotation** [Balunović et al. 2019]: We analyze image rotations with bilinear interpolation within a range of angles θ, where $\theta = [-\theta_{max}, \theta_{max}] + [1, 1]\epsilon$; the parameter $\theta_{max} \in \mathbb{R}_{\geq 0}$ represents the rotation angle in radians.
(4) **Composition:** We look at three ways of composing the above functions – haze followed by rotation, contrast followed by rotation, and contrast followed by haze.

The contrast, rotation, and composite perturbations described above are non-differentiable but Lipschitz continuous, and therefore cannot be handled by prior work [Edalat and Maleki 2017; Jordan and Dimakis 2020; Mangal et al. 2020; Zhang et al. 2019]. In our experiments, we consider $\alpha_{max} \leq 0.63$ for the haze and contrast perturbations, and $\theta_{max} \leq 0.32$ radians (which corresponds to approximately $\pm 18°$) for rotation.

Network Architectures. We trained three ReLU networks each for the CIFAR10 and MNIST datasets. The first network (FFNN) is a 7-layer fully-connected architecture from RecurJac [Zhang et al. 2019]. The other networks are convolutional networks from Mirman et al. [2018] – ConvMed features two convolutional and two fully-connected layers, while ConvBig features four convolutional and three fully-connected layers. Details on these networks are in Appendix A.5. Our largest network is the CIFAR ConvBig network containing > 62 K neurons. These network sizes are comparable to those of other state-of-the-art verification techniques [Singh et al. 2019]. Furthermore, for the local optimization landscape experiment, we trained 7 fully-connected networks on the MNIST dataset, varying the total number of layers from 3 to 9. Each hidden layer contains 30 neurons, and a ReLU activation is applied after every layer (including the final layer).

Data Transformation. For all MNIST networks, we transformed the training set so that for each image, we padded it by 4 and took a random 28x28 crop of the resulting 32x32 image. For the CIFAR10 networks, we introduced random cropping with padding 4 *and* randomly flipping each image horizontally with probability 0.5. Afterwards, we normalized the images using $\mu = 0.1307$, $\sigma = 0.3081$ for MNIST and $\mu = (0.4914, 0.4822, 0.4465)$, $\sigma = (0.2023, 0.1994, 0.2010)$ for CIFAR10.

Training Hyperparameters. For the 7 networks trained for the local optimization experiment, we used the Adam optimizer [Kingma and Ba 2014] with a learning rate of 10^{-4} and L2-regularization with $\lambda = 0.001$; we trained with a batch size of 500 for 60 epochs, using 6,000 images from the training set for validation. For all other networks, we used the Adam optimizer with a learning rate of 10^{-3} on MNIST and 10^{-4} on CIFAR10. We trained the MNIST networks with a batch size of 500 for

30 epochs, using 6,000 images from the training set for validation; we trained the CIFAR10 networks with a batch size of 64 for 60 epochs, using 5,000 images from the training set for validation. In all cases, we then picked the model that attained the highest validation accuracy out of all epochs.

Network Accuracies. The fully-connected networks trained for the local optimization landscape experiment all attain test accuracy of at least 92%. The classification accuracies for the networks in the Lipschitz experiments are shown in Table 1.

Table 1. Classification accuracy on test set for our networks.

	CIFAR10	MNIST
FFNN	56.71	98.34
ConvMed	67.26	98.95
ConvBig	79.58	99.50

8 EXPERIMENTAL EVALUATION

We evaluate the effectiveness of our approach on two tasks: (i) Lipschitz robustness certification and (ii) optimization landscape analysis. Both tasks are defined across a variety of datasets, perturbation functions, and neural network architectures, demonstrating our language's flexibility and scalability.

8.1 Lipschitz Robustness

We compute an upper bound on the Lipschitz constant of functions of the form $f \circ n \circ p$, where p denotes one or more perturbation functions, n is an input normalization function (which simply rescales images by a constant), and f is a neural network. The correctness of bounding the Lipschitz constant via the Interval Clarke Jacobian is proved in Appendix A.3. We use the ℓ_∞-norm for the calculation of all Lipschitz constants. For both individual and composite perturbations, we consider five versions of our tool: DeepJ with the given range for the input perturbation parameter(s) and four versions of DeepJ where the input intervals are subdivided uniformly, denoted DeepJ kx, where k represents how many subintervals are used *per input parameter*. When using splitting, we compute the upper bound on the Lipschitz constant separately for each split and take the maximum constant across the splits. Each pass of our analysis per image and split is completely independent of one another, and is thus parallelized in our implementation. Since no existing work can handle functions of the form $f \circ n \circ p$ considered in our evaluation, we employ a baseline combining global and local Lipschitz analysis. The baseline computes $L_{loc}(p) \cdot L_{global}(n) \cdot L_{global}(f)$, where L_{loc} and L_{global} are the local and global Lipschitz constants of their corresponding functions. As p is not globally Lipschitz, we obtain $L_{loc}(p)$ with DeepJ. We compute $L_{global}(n)$ as the reciprocal of the standard deviation (MNIST) or the reciprocal of the smallest standard deviation of the three RGB channels (CIFAR10) used for normalizing the input. We calculate $L_{global}(f)$ by multiplying the norm of each layer's weights using the tool from Gouk et al. [2021].

Results for Individual Perturbations. Figure 7 shows the Lipschitz constant results for single perturbations on our larger convolutional architectures for both MNIST and CIFAR10. The results for the remaining networks are in Appendix A.5. The x-axis for the haze and contrast perturbations shows the value of α_{max} used for defining the input range $\alpha = [0, \alpha_{max}] + [1, 1]\epsilon$, while for rotation, the x-axis shows θ_{max} used for defining the input range $\theta = [-\theta_{max}, \theta_{max}] + [1, 1]\epsilon$. The y-axis shows the upper bound on the Lipschitz constant computed with different methods. Both axes use logarithmic scales. Each data point is the average over 100 correctly classified images, selected by taking the first 10 correctly classified test-set images from each output category. The same set of images are used per dataset for each experiment. We use $\alpha_{max} \in \{10^{-k/4} \cdot 2 \mid k \in [2, 18]\}$ for haze and contrast and $\theta_{max} \in \{10^{-k/4} \mid k \in [2, 18]\}$ for rotation. For the rotation perturbation, the

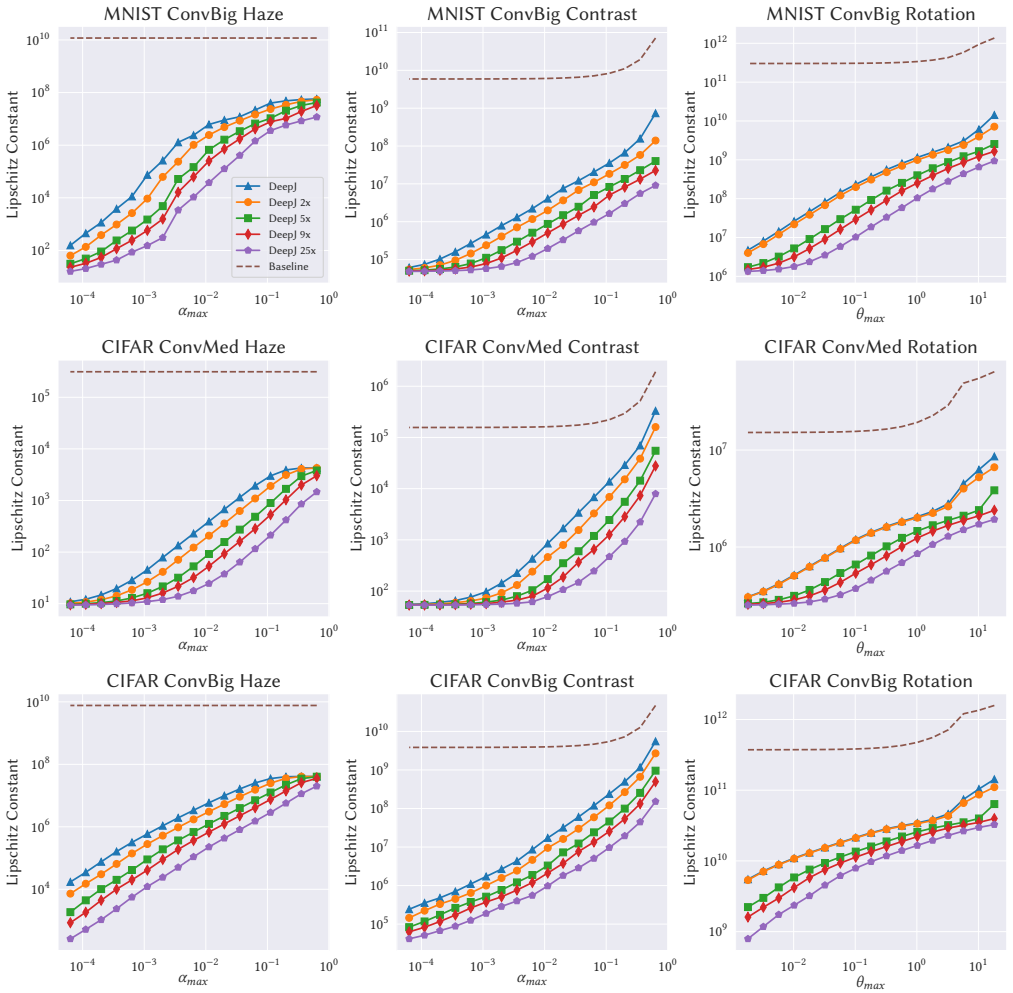

Fig. 7. Upper bounds on the Lipschitz constants with respect to individual perturbations.

input to our function is in radians and covers the same range of interval sizes as the other two perturbations, but we convert the units to degrees for clearer presentation.

In all cases, DeepJ is more precise than the baseline, often obtaining Lipschitz bounds orders of magnitude smaller than the baseline. The computed bounds become larger as the perturbation size is increased. It can also be seen that increased splitting leads to more precise results, with DeepJ 25x achieving much lower bounds than vanilla DeepJ (which does not do splitting).

Table 2 shows statistics on the runtime of the different methods for the same networks as in Figure 7. We consider all values of the parameters on the x-axis shown in Fig. 7. The runtimes for the remaining networks can be found in Appendix A.5. We report the minimum, median, and maximum runtimes across the 100 images for each function. DeepJ 25x takes the longest to run due to more splits, while the baseline usually runs the fastest, except for rotation on large angles. In some cases, splitting may be faster than the baseline or vanilla DeepJ, since without splitting, intervals can become too over-approximate; evaluating conditionals on over-approximate intervals

Table 2. Runtimes in seconds to compute Interval Clarke Jacobians for individual perturbations over 100 images. For each network, the six rows represent the times for DeepJ vanilla, 2x, 5x, 9x, 25x, and Baseline.

	Haze			Contrast			Rotation		
	Min	Med	Max	Min	Med	Max	Min	Med	Max
MNIST ConvBig	10.4	10.9	11.7	10.3	10.5	10.9	24.5	24.8	130.5
	20.6	21.1	21.9	20.4	20.7	21.4	48.9	49.2	129.8
	51.1	51.8	57.4	50.7	51.1	53.6	79.4	80.1	127.4
	91.5	101.5	391.8	91.4	93.1	359.9	134.4	141.9	311.5
	254.0	258.8	397.2	252.7	282.0	489.5	352.3	379.7	689.5
	0.05	0.06	0.07	0.04	0.06	0.08	13.6	14.0	114.9
CIFAR ConvMed	0.9	0.9	1.0	0.9	0.9	0.9	69.9	70.0	715.6
	1.7	1.7	1.8	1.7	1.7	1.8	139.9	140.1	614.2
	4.1	4.2	4.3	4.1	4.2	4.3	142.5	143.4	550.2
	7.3	7.4	7.6	7.3	7.4	7.5	215.7	241.7	630.2
	20.1	21.8	88.7	20.3	21.6	191.7	503.1	531.6	865.5
	0.05	0.05	0.07	0.06	0.06	0.07	65.4	66.2	685.6
CIFAR ConvBig	14.3	14.7	14.9	14.4	14.7	15.0	83.3	83.7	734.7
	28.5	29.1	29.5	28.6	29.2	29.6	165.6	167.0	635.4
	70.8	71.9	73.3	70.1	72.0	72.9	208.3	211.3	475.4
	127.3	130.3	142.9	127.5	131.2	150.5	335.1	339.3	768.1
	350.7	360.9	757.5	351.2	357.6	510.9	840.5	880.6	1029.7
	0.05	0.06	0.08	0.07	0.07	0.08	65.5	66.2	685.6

often requires evaluating both branches, which may lead to the exponential blowup phenomenon discussed in Section 6.4. For rotation with bilinear interpolation, we empirically observed this beyond ± 0.1 radians.

The computation of $L_{loc}(p)$ via our method contributes the most to the runtime of the baseline. For all versions of DeepJ, the rotation perturbation has the highest runtime. For a given perturbation type, the runtime increases with the size of the network. On the most expensive rotation perturbation with the CIFAR ConvBig network, the median time for DeepJ to finish is under 1.4 min (per 100 images). Finally, the precision of our analysis can be improved by increasing the number of splits (as seen in Figure 7) at the cost of additional runtime (as seen in Table 2).

Results for Composite Perturbations. Figure 8 shows the upper bounds on the Lipschitz constant computed for compositions of perturbations on the same networks as in Figure 7. The results for the remaining networks can be found in Appendix A.5. Haze-Rotation denotes a haze perturbation composed with a rotation, in that order; the terminology is similar for the other composite perturbations. We use the same interval width when perturbing each parameter independently; the x-axis shows this width. For compositions that involve rotation, if the interval size on the x-axis is denoted s, we utilize the real interval $[-s/2, s/2]$ for rotation and the interval $[0, s]$ for the other perturbation. The y-axis shows the upper bounds on the Lipschitz constants computed by each method. Again, both axes use logarithmic scales. Each data point is the average over 10 correctly classified images, taking the first correctly classified test-set image from each output category. We use $s \in \{10^{-k/4} \cdot 2 \mid k \in \{4, 7, 10, 13, 16, 19\}\}$ for all experiments.

As with individual perturbations, DeepJ is better than the baseline in all cases, obtaining upper bounds on the Lipschitz constants orders of magnitude smaller than the baseline, as seen in Figure 8. DeepJ 9x with the largest number of splits is the most precise. The Lipschitz constants for both the Haze-Rotation and Contrast-Rotation perturbations are nearly identical, since the entries in the Clarke Jacobian corresponding to the rotation variable have much higher magnitudes. Thus, these entries dominate the ℓ_∞-norm, overshadowing the effect of the other perturbation.

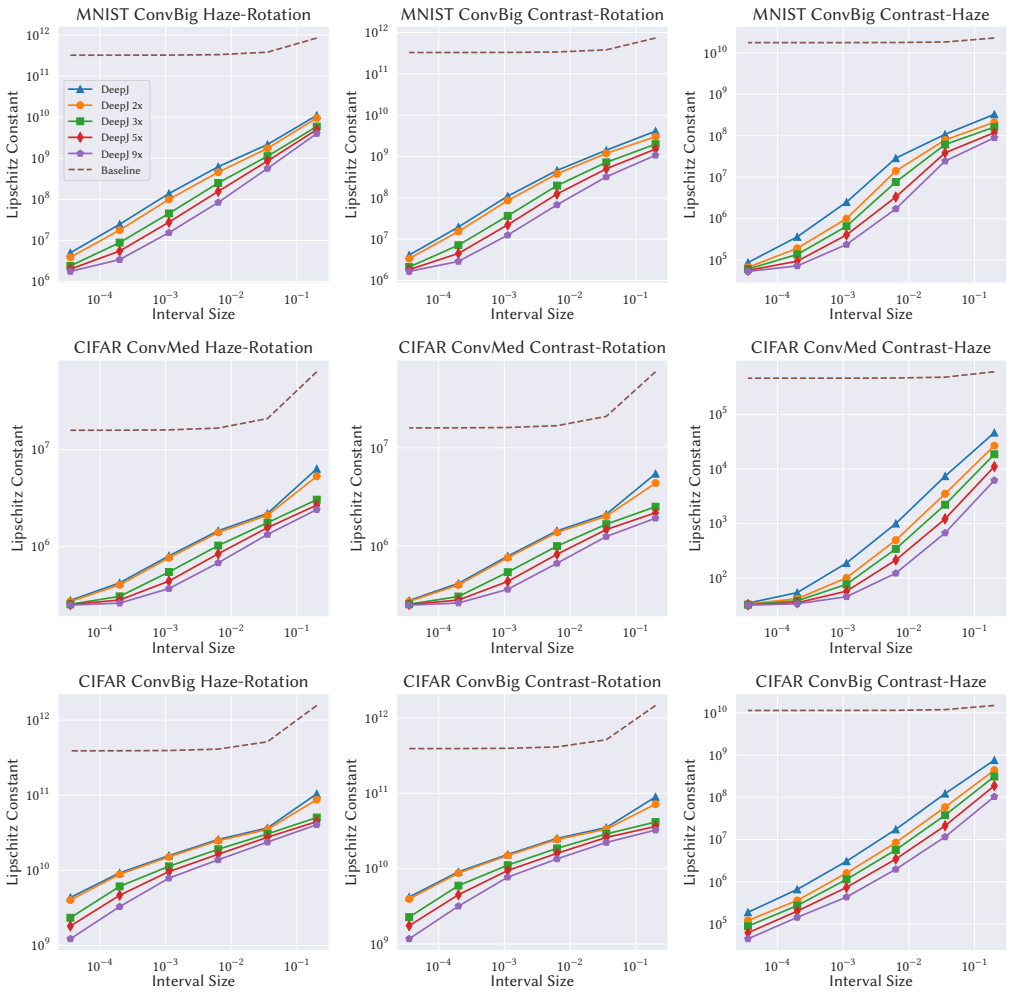

Fig. 8. Upper bounds on the Lipschitz constants with respect to composite perturbations.

Table 3 shows the runtime statistics for the different methods in the same way as Table 2. The results for the remaining networks can be found in Appendix A.5. We observe similar trends in the relative runtimes of the different methods as with individual perturbations. For all versions of DeepJ and for the baseline, the compositions that take the longest time to analyze are those containing rotations. On the most expensive composite perturbation (Haze-Rotation) with the CIFAR ConvBig network, the median time for DeepJ to finish is under 51 seconds (per 10 images). As with individual perturbations, the precision of DeepJ can be improved by considering more splits at the cost of additional runtime.

Our implementation, though implemented via floating-point, assumes real arithmetic. To ensure this assumption does not lead to substantially different results, we also implemented a floating-point sound version using the techniques in [Miné 2004]. We ran floating-point sound experiments for vanilla DeepJ. Table 4 shows that the difference in the computed Lipschitz constants is negligible ($< 10^{-10}$) in all cases. However, ensuring floating-point soundness adds up to 4x runtime overhead.

Table 3. Runtimes in seconds to compute Interval Clarke Jacobians for composite perturbations over 10 images. For each network, the six rows represent the times for DeepJ vanilla, 2x, 3x, 5x, 9x, and Baseline.

	Haze-Rotation			Contrast-Rotation			Contrast-Haze		
	Min	Med	Max	Min	Med	Max	Min	Med	Max
MNIST ConvBig	14.8	14.9	24.4	14.7	14.9	20.9	6.3	6.4	6.7
	58.7	59.6	72.2	58.8	59.1	70.9	24.8	25.2	26.0
	76.0	76.6	78.0	75.7	76.3	77.6	33.8	34.2	35.0
	102.5	103.4	107.6	102.3	102.5	105.7	52.3	52.8	55.4
	247.7	251.3	263.6	246.4	248.8	257.3	158.6	160.0	174.0
	8.2	8.2	13.9	8.2	8.2	13.8	0.06	0.07	0.08
CIFAR ConvMed	42.2	42.2	80.7	42.1	42.3	82.0	0.6	0.8	1.8
	167.2	167.6	239.2	167.9	173.0	251.3	2.0	2.1	2.2
	209.1	209.4	216.1	209.3	210.1	214.9	2.9	3.1	3.5
	247.4	247.9	264.1	247.4	247.6	265.3	4.3	4.4	4.8
	527.7	530.6	559.6	526.1	530.1	561.7	12.7	13.3	13.5
	39.4	39.4	79.1	39.4	39.4	76.1	0.05	0.05	0.07
CIFAR ConvBig	49.9	50.4	89.1	50.0	50.1	89.7	8.6	8.8	16.8
	198.9	199.8	270.9	198.8	199.3	269.7	34.2	34.7	35.2
	251.1	252.5	257.9	251.1	251.9	258.5	46.0	46.7	52.4
	313.9	315.0	328.3	314.2	315.9	327.8	72.7	73.7	74.7
	695.3	701.8	728.3	693.6	696.5	725.9	220.6	224.0	227.1
	39.4	39.4	79.1	39.4	39.4	76.1	0.06	0.06	0.08

Table 4. Error and overhead of floating-point sound computations for DeepJ. For each network, the two rows represent maximum relative error and maximum relative time overhead, respectively.

	Haze	Contrast	Rotation	Haze-Rotation	Contrast-Rotation	Contrast-Haze
MNIST ConvBig	2.24e-11	9.34e-13	1.23e-12	1.24e-12	1.25e-12	9.77e-13
	3.67x	3.78x	3.51x	3.59x	3.58x	4.08x
CIFAR ConvMed	8.08e-12	1.79e-12	8.93e-13	9.07e-13	8.99e-13	5.62e-12
	3.58x	3.68x	2.83x	3.28x	3.25x	3.45x
CIFAR ConvBig	2.84e-11	2.81e-12	1.30e-12	1.36e-12	1.36e-12	6.85e-12
	3.45x	3.43x	2.94x	3.31x	3.27x	3.66x

8.2 Local Optimization Landscape Analysis

Obtaining an Interval Clarke Jacobian allows us to study the local geometry of $f \circ n \circ p$. We focus on finding the largest input regions where no stationary point exists. To prove that a given region does not contain a stationary point, we check if there exists an interval entry $[l_{i,j}, u_{i,j}]$ in the Interval Clarke Jacobian such that $l_{i,j} > 0$ or $u_{i,j} < 0$. If this holds, then we can guarantee that the Clarke Jacobian matrix will not become zero, and therefore no stationary point exists within the given input region (which follows from Theorem 2.3.2 of [Clarke 1990]). Hence, similar to RecurJac [Zhang et al. 2019], we study the relationship between network depth and the maximal interval size for which the absence of stationary points can still be guaranteed. (Zhang et al. [2019] cannot handle functions of the form $f \circ n \circ p$ considered in our work.)

We define the set of input intervals to analyze as $\{[0, 10^{-k/4} \cdot 2] \mid k \in [2, 21]\}$ for haze and contrast and $\{[-10^{-k/4}, 10^{-k/4}] \mid k \in [2, 21]\}$ for rotation. We consider 100 test images from MNIST that are correctly classified, using the first 10 correctly classified test images from each category. For each combination of perturbation type and network, we compute the largest input region where a stationary point does not exist. For each image, we uniformly split every input interval into 25 subintervals, computing the Interval Clarke Jacobians separately for each subinterval. Next, we identify the subintervals where the entry in the Interval Clarke Jacobian corresponding to

Table 5. Largest interval sizes that guarantee no stationary point exists for CIFAR10 (left) and MNIST (right) networks, averaged over 100 images.

	Haze	Contrast	Rotation
FFNN	6.2e-5	7.5e-5	3.6e-7
ConvMed	4.2e-3	4.0e-3	4.4e-5
ConvBig	4.4e-6	4.7e-6	2.3e-8

	Haze	Contrast	Rotation
FFNN	9.7e-4	2.0e-3	6.7e-5
ConvMed	7.8e-3	1.8e-2	1.3e-3
ConvBig	1.3e-4	7.6e-4	1.9e-5

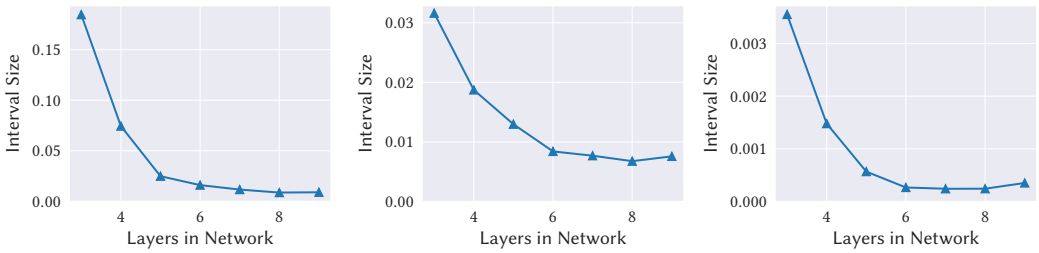

Fig. 9. Largest interval size that guarantees no stationary point exists for Haze (left), Contrast (center), and Rotation (right), averaged over 100 images.

the correctly classified class is either strictly positive or negative. We sum the widths of all such subintervals, then compute the maximum of these sums across all candidate widths.

Figure 9 shows that as the depth of the fully-connected network increases, the maximal interval size that guarantees a non-zero Jacobian decreases for all perturbations. Further, we use the same procedure to obtain maximal interval sizes for the networks used in Section 8.1, as shown in Table 5. We observe that rotation requires noticeably smaller interval sizes as it is a more complex perturbation involving interpolation, whereas the other two are simpler pixel-wise operations. Furthermore, we can certify the absence of stationary points for larger regions on MNIST compared to CIFAR10, due to the former's smaller input dimension.

9 RELATED WORK

While there is considerable work on the static analysis of input-output properties of ML models specified via constraints on the network inputs and outputs [Balunović et al. 2019; Ehlers 2017; Huang et al. 2017; Katz et al. 2017, 2019; Singh et al. 2018, 2019; Sotoudeh and Thakur 2020; Urban and Miné 2021], such as for robustness and safety, much less work has been done on formally analyzing the Jacobian matrix.

From the programming languages literature, prior works [Di Gianantonio and Edalat 2013; Edalat et al. 2013; Sherman et al. 2021] have examined automatic differentiation through the Clarke Jacobian [Clarke 1990]. However, most define their semantics for computing the Clarke Jacobian of a single point [Di Gianantonio and Edalat 2013; Khan and Barton 2013], instead of an abstraction of points. Along similar lines, Di Gianantonio and Edalat [2013] use the L-derivative (equivalent to the Clarke derivative), but they also only define the semantics for scalar input points. Moreover, they restrict functions' domains to [-1,1] and require all Lipschitz constants be less than 1, thus their analysis cannot be used for local Lipschitz certification.

Additionally, the existing works that can formally analyze Jacobians for sets of input points are insufficient for our tasks. Edalat and Lieutier [2004] is restricted to functions of a single variable with input domain on $[0, 1]$. Follow up works [Edalat et al. 2013; Edalat and Maleki 2017, 2018] all suffer other restrictions, particularly requiring the output dimension be one and only supporting limited arithmetic operations (e.g., no division), which render them unable to analyze both neural networks

and our perturbations. Furthermore, these techniques provide only a theoretical discussion, with no implementation of their approaches. Lastly, λ_S [Sherman et al. 2021] presents semantics for the Clarke Jacobian for concrete input points, extended partially to intervals (described in their appendix) as compared to our approach, which presents a sound abstract interpretation of the Clarke Jacobian that we compute with a set of fully compositional dual interval domain transformers, whose behavior is specified exactly for all language primitives.

In addition, while the practical problem of local Lipschitz certification of neural networks has been studied [Jordan and Dimakis 2020; Scaman and Virmaux 2018; Weng et al. 2018a,b, 2020; Zhang et al. 2019], to the best of our knowledge, none of these works can bound local Lipschitz constants for composite, non-smooth perturbations. Both Mangal et al. [2020] and Jordan and Dimakis [2020] use proof techniques that require the classifier function analyzed be piecewise linear, hence they cannot support arbitrary activations. Additionally, RecurJac [Zhang et al. 2019] cannot support arbitrary arithmetic primitives like non-scalar multiplication or division. Hence, none of these works can reason about non-differentiable input perturbations, such as those generated by rotation [Balunović et al. 2019] or contrast variation [Paterson et al. 2021].

Our work also bears similarity with Rosa [Darulova and Kuncak 2017], as they also bound Jacobians to compute Lipschitz constants. However, unlike us, they do not support bounding Clarke Jacobians. Further, while their abstract domain tracks "ϵ-terms," the semantics of these terms do not correspond to first derivatives, but rather numerical round-off error effects.

10 CONCLUSION

We developed a novel abstraction for bounding the Clarke Jacobian of a Lipschitz, but not necessarily differentiable function for local input regions. Our domain, based upon dual numbers, soundly over-approximates all first derivatives needed to compute the Clarke Jacobian. We implemented our analysis in tool named DeepJ and showed that it can efficiently compute Lipschitz bounds and analyze the local geometry of multiple deep neural networks with respect to multiple non-differentiable input perturbations. Our work is the first to address the problem of local Lipschitz certification of non-smooth perturbations, such as haze, contrast variation, rotation with bilinear interpolation, and their compositions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and Ben Sherman for helpful discussions during the early stages of this work. This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, USDA NIFA Grant No. NIFA-2024827, a gift from Facebook, and a Sloan Graduate Fellowship.

REFERENCES

David Alvarez-Melis and Tommi S Jaakkola. 2018. Towards robust interpretability with self-explaining neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*.

Mislav Balunović, Maximilian Baader, Gagandeep Singh, Timon Gehr, and Martin Vechev. 2019. Certifying geometric robustness of neural networks. *Advances in Neural Information Processing Systems 32* (2019).

Thomas Beck and Herbert Fischer. 1994. The if-problem in automatic differentiation. *J. Comput. Appl. Math.* (1994).

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. arXiv:1604.07316 [cs.CV]

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

Frank Clarke. 1990. 2. Generalized Gradients. In *Optimization and Nonsmooth Analysis*. Society for Industrial and Applied Mathematics, 24–109.

Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* (2017).

Luiz Henrique De Figueiredo and Jorge Stolfi. 2004. Affine arithmetic: concepts and applications. *Numerical Algorithms* (2004).

Pietro Di Gianantonio and Abbas Edalat. 2013. A language for differentiable functions. In *International Conference on Foundations of Software Science and Computational Structures*.

Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*. 214–226.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

Abbas Edalat and André Lieutier. 2004. Domain theory and differential calculus (functions of one variable). *Mathematical Structures in Computer Science* (2004).

Abbas Edalat, André Lieutier, and Dirk Pattinson. 2013. A computational model for multi-variable differential calculus. *Information and Computation* (2013).

Abbas Edalat and Mehrdad Maleki. 2017. Differentiation in logical form. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*.

Abbas Edalat and Mehrdad Maleki. 2018. Differential calculus with imprecise input and its logical framework. In *International Conference on Foundations of Software Science and Computation Structures*.

Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 269–286.

Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, and Jeff Dean. 2019. A guide to deep learning in healthcare. *Nature Medicine* 25 (01 2019).

Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 237–247.

Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael J Cree. 2021. Regularisation of neural networks by enforcing lipschitz continuity. *Machine Learning* 110 (2021). https://github.com/henrygouk/keras-lipschitz-networks

Andreas Griewank. 2013. On stable piecewise linearization and generalized algorithmic differentiation. *Optimization Methods and Software* (2013).

Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 3–29.

Matt Jordan and Alexandros G Dimakis. 2020. Exactly computing the local lipschitz constant of relu networks. *arXiv preprint arXiv:2003.01219* (2020).

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 443–452.

Kamil A Khan and Paul I Barton. 2013. Evaluating an element of the Clarke generalized Jacobian of a composite piecewise differentiable function. *ACM Transactions on Mathematical Software (TOMS)* 39, 4 (2013).

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. 2022. Appendix to DeepJ. https://jsl1994.github.io/papers/POPL2022_appendix.pdf

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

Ji Lin, Chuang Gan, and Song Han. 2019. Defensive Quantization: When Efficiency Meets Robustness. In *International Conference on Learning Representations*.

Ravi Mangal, Kartik Sarangmath, Aditya V Nori, and Alessandro Orso. 2020. Probabilistic Lipschitz Analysis of Neural Networks. In *International Static Analysis Symposium*. Springer, 274–309.

Antoine Miné. 2004. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*. Springer, 3–17.

Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372.

Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*.

Matthew Mirman, Gagandeep Singh, and Martin T. Vechev. 2019. A Provable Defense for Deep Residual Networks. *CoRR* abs/1903.12519 (2019).

Ramon Moore, R. Baker Kearfott, and Michael Cloud. [n.d.]. *Introduction to Interval Analysis*. Chapter 7. Interval Matrices.

Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2021. Scaling Polyhedral Neural Network Verification on GPUs. In *Proc. Machine Learning and Systems (MLSys)*, Vol. 3. 733–746.

Colin Paterson, Haoze Wu, John Grese, Radu Calinescu, Corina S. Pasareanu, and Clark Barrett. 2021. DeepCert: Verification of Contextually Relevant Robustness for Neural Network Image Classifiers. arXiv:arXiv:2103.01629

Daniel Richardson. 1969. Some undecidable problems involving elementary functions of a real variable. *The Journal of Symbolic Logic* (1969).

Kevin Scaman and Aladin Virmaux. 2018. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 3839–3848.

Stefan Scholtes. 2012. *Piecewise Differentiable Functions*.

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. Lambda S: Computable Semantics for Differentiable Programming with Higher-Order Functions and Datatypes. POPL (2021).

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T Vechev. 2018. Fast and Effective Robustness Certification. *NeurIPS* 1, 4 (2018), 6.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Matthew Sotoudeh and Aditya V Thakur. 2020. Abstract Neural Networks. In *International Static Analysis Symposium*.

Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. 2018. Lipschitz-margin training: scalable certification of perturbation invariance for deep neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*.

Caterina Urban and Antoine Miné. 2021. A Review of Formal Methods applied to Machine Learning. *arXiv preprint arXiv:2104.02466* (2021).

Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S. Dhillon, and Luca Daniel. 2018a. Towards Fast Computation of Certified Robustness for ReLU Networks. In *International Conference on Machine Learning (ICML)*.

Tsui-Wei Weng, Huan Zhang, Pin-Yu Chen, Jinfeng Yi, Dong Su, Yupeng Gao, Cho-Jui Hsieh, and Luca Daniel. 2018b. Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach. In *International Conference on Learning Representations*.

Tsui-Wei Weng, Pu Zhao, Sijia Liu, Pin-Yu Chen, Xue Lin, and Luca Daniel. 2020. Towards certificated model robustness against weight perturbations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 6356–6363.

Bohang Zhang, Tianle Cai, Zhou Lu, Di He, and Liwei Wang. 2021. Towards Certifying L-infinity Robustness using Neural Networks with L-inf-dist Neurons. In *Proc. International Conference on Machine Learning (ICML)*.

Huan Zhang, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane Boning, and Cho-Jui Hsieh. 2020. Towards Stable and Efficient Training of Verifiably Robust Neural Networks. In *Proc. International Conference on Learning Representations (ICLR)*.

Huan Zhang, Pengchuan Zhang, and Cho-Jui Hsieh. 2019. RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications. In *The 33rd AAAI Conference on Artificial Intelligence, (AAAI)*.