Probabilistic Lipschitz Analysis of Neural Networks

Ravi Mangal[1], Kartik Sarangmath[1], Aditya V. Nori[2], and Alessandro Orso[1]

[1] Georgia Institute of Technology, Atlanta GA 30332, USA
{rmangal3,kartiksarangmath,orso}@gatech.edu
[2] Microsoft Research, Cambridge, CB1 2FB, UK
Aditya.Nori@microsoft.com

Abstract. We are interested in algorithmically proving the robustness of neural networks. Notions of robustness have been discussed in the literature; we are interested in probabilistic notions of robustness that assume it feasible to construct a statistical model of the process generating the inputs of a neural network. We find this a reasonable assumption given the rapid advances in algorithms for learning generative models of data. A neural network f is then defined to be probabilistically robust if, for a randomly generated pair of inputs, f is likely to demonstrate k-Lipschitzness, i.e., the distance between the outputs computed by f is upper-bounded by the k^{th} multiple of the distance between the pair of inputs. We name this property, *probabilistic Lipschitzness*.
We model generative models and neural networks, together, as programs in a simple, first-order, imperative, probabilistic programming language, *pcat*. Inspired by a large body of existing literature, we define a denotational semantics for this language. Then we develop a sound *local Lipschitzness* analysis for *cat*, a non-probabilistic sublanguage of *pcat*. This analysis can compute an upper bound of the "Lipschitzness" of a neural network in a bounded region of the input set. We next present a provably correct algorithm, `PROLIP`, that analyzes the behavior of a neural network in a user-specified box-shaped input region and computes - (i) lower bounds on the probabilistic mass of such a region with respect to the generative model, (ii) upper bounds on the Lipschitz constant of the neural network in this region, with the help of the local Lipschitzness analysis. Finally, we present a sketch of a proof-search algorithm that uses `PROLIP` as a primitive for finding proofs of probabilistic Lipschitzness. We implement the `PROLIP` algorithm and empirically evaluate the computational complexity of `PROLIP`.

1 Introduction

Neural networks (`NNs`) are useful for modeling a variety of computational tasks that are beyond the reach of manually written programs. We like to think of `NNs` as programs in a first-order programming language specialized to operate over vectors from high-dimensional Euclidean spaces. However, `NNs` are algorithmically learned from observational data about the task being modeled. These tasks

typically represent natural processes for which we have large amounts of data but limited mathematical understanding. For example, NNs have been successful at image recognition [40] - assigning descriptive labels to images. In this case, the underlying natural process that we want to mimic computationally is image recognition as it happens in the human brain. However, insufficient mathematical theory about this task makes it hard to develop a hand-crafted algorithm.

Given that NNs are discovered algorithmically, it is important to ensure that a learned NN actually models the computational task of interest. With the perspective of NNs as programs, this reduces to proving that the NN behaves in accordance with the formal specification of the task at hand. Unfortunately, limited mathematical understanding of the tasks implies that, in general, we are unable to even state the formal specification. In fact, it is precisely in situations where we are neither able to manually design an algorithm nor able to provide formal specifications in which NNs tend to be deployed. This inability to verify or make sense of the computation represented by a NN is one of the primary challenges to the widespread adoption of NNs, particularly for safety critical applications. In practice, NNs are tested on a limited number of manually provided tests (referred to as test data) before deploying. However, a natural question is, what formal correctness guarantees, if any, can we provide about NNs?

A hint towards a useful notion of correctness comes from an important observation about the behavior of NNs, first made by [51]. They noticed that state-of-the-art NNs that had been learned to perform the image recognition task were unstable - small changes in the inputs caused the learned NNs to produce large, unexpected, and undesirable changes in the outputs. In the context of the image recognition task, this meant that small changes to the images, imperceptible to humans, caused the NN to produce very different labels. The same phenomenon has been observed by others, and in the context of very different tasks, like natural language processing [35,2] and speech recognition [13,14,45]. This phenomenon, commonly referred to as lack of *robustness*, is widespread and undesirable. This has motivated a large body of work (see [59,43,62] for broad but non-exhaustive surveys) on algorithmically proving NNs robust. These approaches differ not only in the algorithms employed but also in the formal notions of robustness that they prove.

An majority of the existing literature has focused on local notions of robustness. Informally, a NN is *locally robust* at a specific input, x_0, if it behaves robustly in a bounded, local region of the input Euclidean space centered at x_0. There are multiple ways of formalizing this seemingly intuitive property. A common approach is to formalize this property as, $\forall x.(\|x - x_0\| \leqslant r) \rightarrow \phi((fx), (fx_0))$, where f is the NN to be proven locally robust at x_0, (fx) represents the result of applying the NN f on input x, $\phi((fx), (fx_0))$ represents a set of linear constraints imposed on (fx), and $\|\cdot\|$ represents the norm or distance metric used for measuring distances in the input and output Euclidean spaces (typically, an l_p norm is used with $p \in \{1, 2, \infty\}$). An alternate, less popular, formulation of local robustness, referred to as *local Lipschitzness* at a point, requires that $\forall x, x'.(\|x - x_0\| \leqslant r) \wedge (\|x' - x_0\| \leqslant r) \rightarrow (\|fx - fx'\| \leqslant k * \|x - x'\|)$. Local

Lipschitzness ensures that in a ball of radius r centered at x_0, changes in the input only lead to bounded changes in the output. One can derive other forms of local robustness from local Lipschitzness. (see Theorem 3.2 in [58]). We also find local Lipschitzness to be an aesthetically more pleasing and natural property of a function. But, local Lipschitzness is a relational property [12,6]/hyperproperty [18] unlike the first formulation, which is a safety property [41]. Algorithms for proving safety properties of programs have been more widely studied and there are a number of mature approaches to build upon, which may explain the prevalence of techniques for proving the former notion of local robustness. For instance, [49,28] are based on variants of polyhedral abstract interpretation [21], [10,37,38] encode the local robustness verification problem as an SMT constraint.

Local robustness (including local Lipschitzness) is a useful but limited guarantee. For inputs where the NN has not been proven to be locally robust, no guarantees can be given. Consequently, a global notion of robustness is desirable. Local Lipschitzness can be extended to a global property - a NN f is *globally Lipschitz* or k-*Lipschitz* if, $\forall x, x'.(\|fx - fx'\| \leqslant k * \|x - x'\|)$. Algorithms have been proposed in programming languages and machine learning literature for computing Lipschitz constant upper bounds. Global robustness is guaranteed if the computed upper bound is $\leqslant k$.

Given the desirability of global robustness over local robustness, the focus on local robustness in the existing literature may seem surprising. There are two orthogonal reasons that, we believe, explain this state of affairs - (i) proving global Lipschitzness, particularly with a tight upper bound on the Lipschitz constant, is more technically and computationally challenging than proving local Lipschitzness, which is itself hard to prove due its relational nature; (ii) requiring NNs to be globally Lipschitz with some low constant k can be an excessively stringent specification, unlikely to be met by most NNs in practice. NNs, unlike typical programs, are algorithmically learnt from data. Unless the learning algorithm enforces the global robustness constraint, it is unlikely for a learned NN to exhibit this "strong" property. Unfortunately, learning algorithms are ill-suited for imposing such logical constraints. These algorithms search over a set of NNs (referred to as the hypothesis class) for the NN minimizing a cost function (referred to as loss function) that measures the "goodness" of a NN for modeling the computational task at hand. These algorithms are greedy and iterative, following the gradient of the loss function. Modifying the loss function in order to impose the desired logical constraints significantly complicates the function structure and makes the gradient-based, greedy learning algorithms ineffective.[3]

Consequently, in this work, we focus on a probabilistic notion of global robustness. This formulation, adopted from [44], introduces a new mathematical object to the NN verification story, namely, a probability measure over the inputs to the NN under analysis. One assumes it feasible to construct a statistical model of the process generating the inputs of a NN. We find this a reasonable assumption given the rapid advances in algorithms for learning generative models of data [39,32]. Such a statistical model yields a distribution D over the inputs of

[3] Recent work has tried to combine loss functions with logical constraints [27].

the NN. Given distribution D and a NN f, this notion of robustness, that we refer to as *probabilistic Lipschitzness*, is formally stated as,

$$\Pr_{x,x' \sim D} \left(\left\| fx - fx' \right\| \leqslant k * \left\| x - x' \right\| \mid \left\| x - x' \right\| \leqslant r \right) \geqslant 1 - \epsilon$$

This says that if we randomly draw two samples, x and x' from the distribution D, then, under the condition that x and x' are r-close, there is a high probability ($\geqslant (1-\epsilon)$) that NN f behaves stably for these inputs. If the parameter $\epsilon = 0$ and $r = \infty$, then we recover the standard notion of k-Lipschitzness. Conditioning on the event of x and x' being r-close reflects the fact that we are primarily concerned with the behavior of the NN on pairs of inputs that are close.

To algorithmically search for proofs of probabilistic Lipschitzness, we model generative models and NNs together as programs in a simple, first-order, imperative, probabilistic programming language, *pcat*. First-order probabilistic programming languages with a `sample` construct, like *pcat*, have been well-studied.[4] Programs in *pcat* denote transformers from Euclidean spaces to probability measures over Euclidean spaces. *pcat*, inspired by the non-probabilistic language *cat* [28], is explicitly designed to model NNs, with vectors in $\mathbb{R}^n$ as the basic datatype. The suitability of *pcat* for representing generative models stems from the fact that popular classes of generative models (for instance, the generative network of generative adversarial networks [32] and the decoder network of variational autoencoders [39]) are represented by NNs. Samples from the input distribution D are obtained by drawing a sample from a standard distribution (typically a normal distribution) and running this sample through generative or decoder networks. In *pcat*, this can be represented as the program, $z \leftsquigarrow N(0,1); g$, where the first statement represents the sampling operation (referred to as sampling from the latent space, with z as the latent variable) and g is the generative or decoder NN. If the NN to be analyzed is f, then we can construct the program, $z \leftsquigarrow N(0,1); g; f$, in *pcat*, and subject it to our analysis.

Adapting a language-theoretic perspective allows us to study the problem in a principled, general manner and utilize existing program analysis and verification literature. In particular, we are interested in sound algorithms that can verify properties of probabilistic programs without needing manual intervention. Thus approaches based on interactive proofs [8,9], requiring manually-provided annotations and complex side-conditions [36,15,7] or only providing statistical guarantees [46,11] are precluded. Frameworks based on abstract interpretation [22,54] are helpful for thinking about analysis of probabilistic programs but we focus on a class of completely automated proof-search algorithms [29,47,1] that only consider probabilistic programs where all randomness introducing statements (i.e., `sample` statements) are independent of program inputs, i.e. samples are drawn from fixed, standard probability distributions, similar to our setting. These algorithms analyze the program to generate symbolic constraints (i.e., sentences in first-order logic with theories supported by SMT solvers) and then compute the probability mass or "volume", with respect to a fixed probability measure, of the set of values satisfying these constraints. These algorithms are

[4] *pcat* has no `observe` or `score` construct and cannot be used for Bayesian reasoning.

unsuitable for parametric probability measures but suffice for our problem. Both generating symbolic constraints and computing volumes can be computationally expensive (and even intractable for large programs), so a typical strategy is to break down the task into simpler sub-goals. This is usually achieved by defining the notion of "program path" and analyzing each path separately. This per path strategy is unsuitable for `NNs`, with their highly-branched program structure. We propose partitioning the program input space (i.e., the latent space in our case) into box-shaped regions, and analyzing the program behavior separately on each box. The box partitioning strategy offers two important advantages - (i) by not relying explicitly on program structure to guide partitioning strategy, we have more flexibility to balance analysis efficiency and precision; (ii) computing the volume of boxes is easier than computing the same for sets with arbitrary or even convex structure.

For the class of probabilistic programs we are interested in (with structure, $z \leftsquigarrow N(0,1); g; f$), the box-partitioning strategy implies repeatedly analyzing the program $g; f$ while restricting z to from box shaped regions. In every run, the analysis of $g; f$ involves computing a box-shaped overapproximation, x_B, of the outputs computed by g when z is restricted to some specific box z_B and computing an upper bound on the local Lipschitz constant of f in the box-shaped region x_B. We package these computations, performed in each iteration of the proof-search algorithm, in an algorithmic primitive, `PROLIP`. For example, consider the scenario where f represents a classifier, trained on the MNIST dataset, for recognizing hand-written digits, and g represents a generative `NN` modeling the distribution of the MNIST dataset. In order to prove probabilistic Lipschitzness of f with respect to the distribution D represented by the generative model $z \leftsquigarrow N(0,1); g$, we iteratively consider box-shaped regions in the latent space (i.e., in the input space of g). For each such box-shaped region σ^B in the input space of g, we first compute an overapproximation $\tilde{\sigma}^B$ of the corresponding box-shaped region in the output space of g. Since the output of g is the input of f, we next compute an upper bound on the local Lipschitz constant of f in the region $\tilde{\sigma}^B$. If the computed upper bound is less than the required bound, we add the probabilistic mass of region σ^B to an accumulator maintaining the probability of f being Lipschitz with respect to the distribution D.

For computing upper bounds on local Lipschitz constants, we draw inspiration from existing literature on Lipschitz analysis of programs [16] and `NNs` [51,19,26,42,53,57,61,52,33]. In particular, we build on the algorithms presented in [57,61]. We translate these algorithms in to our language-theoretic setting and present the local Lipschitzness analysis in the form of an abstract semantics for the *cat* language, which is a non-probabilistic sublanguage of *pcat*. In the process, we also simplify and generalize the original algorithms.

To summarize, our primary contributions in this work are - (i) we present a provably sound algorithmic primitive `PROLIP` and a sketch of a proof-search algorithm for probabilistic Lipschitzness of `NNs`, (ii) we develop a simplified and generalized version of the local Lipschitzness analysis in [57], capable of computing an upper bound on the local Lipschitz constant of box-shaped input regions

$$
\begin{array}{rl}
\text{(variables)} & x, y \in V \\
\text{(naturals)} & m, n \in \mathbb{N} \\
\text{(weights)} & w \in \bigcup_{m,n \in \mathbb{N}} \mathbb{R}^{m \times n} \\
\text{(biases)} & \beta \in \bigcup_{n \in \mathbb{N}} \mathbb{R}^{n}
\end{array}
$$

$$
\begin{array}{rcl}
s & ::= & \textbf{skip} \mid y \leftarrow w \cdot x + \beta \mid y \leftsquigarrow N(0,1) \mid s; s \mid \textbf{if } b \textbf{ then } s \textbf{ else } s \\
s^{-} & ::= & \textbf{skip} \mid y \leftarrow w \cdot x + \beta \mid s^{-}; s^{-} \mid \textbf{if } b \textbf{ then } s^{-} \textbf{ else } s^{-} \\
b & ::= & \boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n) \mid \boldsymbol{\pi}(x,n) \geqslant 0 \mid \boldsymbol{\pi}(x,n) < 0 \mid b \wedge b \mid \neg b \\
e & ::= & \boldsymbol{\pi}(x,n) \mid w \cdot x + \beta
\end{array}
$$

Fig. 1: *pcat* syntax

for any program in the *cat* language, (iii) we develop a strategy for computing proofs of probabilistic programs that limits probabilistic reasoning to volume computation of regularly shaped sets with respect to standard distributions, (iv) we implement the `PROLIP` algorithm, and evaluate its computational complexity.

2 Language Definition

2.1 Language Syntax

pcat (probabilistic conditional affine transformations) is a first-order, imperative probabilistic programming language, inspired by the *cat* language [28]. *pcat* describes always terminating computations on data with a base type of vectors over the field of reals (i.e., of type $\bigcup_{n \in \mathbb{N}} \mathbb{R}^n$). *pcat* is not meant to be a practical language for programming, but serves as a simple, analyzable, toy language that captures the essence of programs structured like `NNs`. We emphasize that *pcat* does not capture the learning component of `NNs`. We think of *pcat* programs as objects learnt by a learning algorithm (commonly stochastic gradient descent with symbolic gradient computation). We want to analyze these learned programs and prove that they satisfy the probabilistic Lipschitzness property.

pcat can express a variety of popular `NN` architectures and generative models. For instance, *pcat* can express ReLU, convolution, maxpool, batchnorm, transposed convolution, and other structures that form the building blocks of popular `NN` architectures. We describe the encodings of these structures in Appendix F. The probabilistic nature of *pcat* further allows us to express a variety of generative models, including different generative adversarial networks (GANs) [32] and variational autoencoders (VAEs) [39].

pcat syntax is defined in Figure 1. *pcat* variable names are drawn from a set V and refer to vector of reals. Constant matrices and vectors appear frequently in *pcat* programs, playing the role of learned weights and biases of `NNs`, and are typically represented by w and β, respectively. Programs in *pcat* are composed of basic statements for performing linear transformations of vectors ($y \leftarrow w \cdot x + \beta$) and sampling vectors from normal distributions ($y \leftsquigarrow N(0,1)$). Sampling from parametric distributions is not allowed. Programs can be composed sequentially ($s; s$) or conditionally (**if** b **then** s **else** s). *pcat* does not have a loop construct,

$$
\begin{aligned}
\Sigma &\triangleq V \to \textstyle\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \\
[\![e]\!] &: \Sigma \to \textstyle\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \\
[\![\boldsymbol{\pi}(x,n)]\!](\sigma) &= \sigma(x)_n \\
[\![w \cdot x + \beta]\!](\sigma) &= w \cdot \sigma(x) + \beta
\end{aligned}
$$

$$
\begin{aligned}
[\![b]\!] &: \Sigma \to \{\mathbf{tt}, \mathbf{ff}\} \\
[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!](\sigma) &= \textbf{if } ([\![\boldsymbol{\pi}(x,m)]\!](\sigma) \geqslant [\![\boldsymbol{\pi}(y,n)]\!](\sigma)) \textbf{ then tt else ff} \\
[\![\boldsymbol{\pi}(x,m) \geqslant 0]\!](\sigma) &= \textbf{if } ([\![\boldsymbol{\pi}(x,m) \geqslant 0]\!](\sigma)) \textbf{ then tt else ff} \\
[\![\boldsymbol{\pi}(x,m) < 0]\!](\sigma) &= \textbf{if } ([\![\boldsymbol{\pi}(x,m)]\!](\sigma) < 0) \textbf{ then ttelse ff} \\
[\![b_1 \wedge b_2]\!](\sigma) &= [\![b_1]\!](\sigma) \wedge [\![b_2]\!](\sigma) \\
[\![\neg b]\!](\sigma) &= \textbf{if } ([\![b]\!] = \mathbf{tt}) \textbf{ then ff else tt}
\end{aligned}
$$

$$
\begin{aligned}
[\![s]\!] &: \Sigma \to P(\Sigma) \\
[\![\textbf{skip}]\!](\sigma) &= \delta_\sigma \\
[\![y \leftarrow w \cdot x + \beta]\!](\sigma) &= \delta_{\sigma[y \mapsto [\![w \cdot x + \beta]\!](\sigma)]} \\
[\![y \leftsquigarrow N(0,1)]\!](\sigma) &= \mathbb{E}_{a\sim N(0,1)}[\lambda\nu.\delta_{\sigma[y\mapsto\nu]}] \\
[\![s_1; s_2]\!](\sigma) &= \mathbb{E}_{\tilde{\sigma}\sim[\![s_1]\!](\sigma)}[[\![s_2]\!]] \\
[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma) &= \textbf{if } ([\![b]\!](\sigma)) \textbf{ then } [\![s_1]\!](\sigma) \textbf{ else } [\![s_2]\!](\sigma)
\end{aligned}
$$

$$
\begin{aligned}
\widehat{[\![s]\!]} &: P(\Sigma) \to P(\Sigma) \\
\widehat{[\![s]\!]}(\mu) &= \mathbb{E}_{\sigma\sim\mu}[[\![s]\!]]
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{[\![s^-]\!]} &: \Sigma \to \Sigma \\
\widetilde{[\![\textbf{skip}]\!]}(\sigma) &= \sigma \\
\widetilde{[\![y \leftarrow w \cdot x + \beta]\!]}(\sigma) &= \sigma[y \mapsto [\![w \cdot x + \beta]\!](\sigma)] \\
\widetilde{[\![s_1; s_2]\!]}(\sigma) &= \widetilde{[\![s_2]\!]}(\widetilde{[\![s_1]\!]}(\sigma)) \\
\widetilde{[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]}(\sigma) &= \textbf{if } ([\![b]\!](\sigma)) \textbf{ then } \widetilde{[\![s_1]\!]}(\sigma) \textbf{ else } \widetilde{[\![s_2]\!]}(\sigma)
\end{aligned}
$$

Fig. 2: *pcat* denotational semantics

acceptable as many `NN` architectures do not contain loops. *pcat* provides a projection operator $\boldsymbol{\pi}(x,n)$ that reads the n^{th} element of the vector referred by x. For *pcat* programs to be well-formed, all the matrix and vector dimensions need to fit together. Static analyses [50,31] can ensure correct dimensions. In the rest of the paper, we assume that the programs are well-formed.

2.2 Language Semantics

We define the denotational semantics of *pcat* in Figure 2, closely following those presented in [8]. We present definitions required to understand these semantics.

Definition 1. *A $\sigma-$algebra on a set X is a set Σ of subsets of X such that it contains X, is closed under complements and countable unions. A set with a $\sigma-$algebra is a measurable space and the subsets in Σ are measurable.*

A measure on a measurable space (X, Σ) is a function $\mu : \Sigma \to [0, \infty]$ such that $\mu(\varnothing) = 0$ and $\mu(\bigcup_{i\in\mathbb{N}} B_i) = \sum_{i\in\mathbb{N}} \mu(B_i)$ such that B_i is a countable family

of disjoint measurable sets. A probability measure or probability distribution is a measure μ with $\mu(X) = 1$.

Given set X, we use $P(X)$ to denote the set of all probability measures over X. A *Dirac distribution* centered on x, written δ_x, maps x to 1 and all other elements of the underlying set to 0. Note that when giving semantics to probabilistic programming languages, it is typical to consider sub-distributions (measures such that $\mu(X) \leqslant 1$ for a measurable space (X, Σ)), as all programs in *pcat* terminate, we do not describe the semantics in terms of sub-distributions. Next, following [8], we give a monadic structure to probability distributions.

Definition 2. *Let $\mu \in P(A)$ and $f : A \to P(B)$. Then, $\mathbb{E}_{a\sim\mu}[f] \in P(B)$ is defined as, $\mathbb{E}_{a\sim\mu}[f] \triangleq \lambda\nu. \int_A f(a)(\nu)\, d\mu(a)$*

Note that in the rest of the paper, we write expressions of the form $\int_A f(a)\, d\mu(a)$ as $\int_{a\in A} \mu(a) \cdot f(a)$ for notational convenience. The metalanguage used in Figure 2 and the rest of the paper is standard first-order logic with ZFC set theory, but we borrow notation from a variety of sources including languages like C and ML as well as standard set-theoretic notation. As needed, we provide notational clarification.

We define the semantics of *pcat* with respect to the set Σ of states. A state σ is a map from variables V to vectors of reals of any finite dimension. The choice of real vectors as the basic type of values is motivated by the goal of *pcat* to model `MN` computations. The set $P(\Sigma)$ is the set of probability measures over Σ. A *pcat* statement transforms a distribution over Σ to a new distribution over the same set. $[\![e]\!]$ and $[\![b]\!]$ denote the semantics of expressions and conditional checks, respectively. Expressions map states to vectors of reals while conditional checks map states to boolean values.

The semantics of statements are defined in two steps. We first define the standard semantics $[\![s]\!]$ where statements map incoming states to probability distributions. Next, the lifted semantics, $\widehat{[\![s]\!]}$, transform a probability distribution over the states, say μ, to a new probability distribution. The lifted semantics ($\widehat{[\![s]\!]}$) are obtained from the standard semantics ($[\![s]\!]$) using the monadic construction of Definition 2. Finally, we also defined a lowered semantics ($\widetilde{[\![s^-]\!]}$) for the *cat* sublanguage of *pcat*. As per these lowered semantics, statements are maps from states to states. Moreover, the lowered semantics of *cat* programs is tightly related to their standard semantics, as described by the following lemma.

Lemma 3. *(Equivalence of semantics)*
$\forall p \in s^-, \sigma \in \Sigma.\ [\![p]\!](\sigma) = \delta_{\widetilde{[\![p]\!]}(\sigma)}$

Proof. Appendix A ■

The lemma states that one can obtain the standard probabilistic semantics for a program p in *cat*, given an initial state σ, by a Dirac delta distribution centered at $\widetilde{[\![p]\!]}(\sigma)$. Using this lemma, one can prove the following useful corollary.

Corollary 4. $\forall p \in s^-, \sigma \in \Sigma, \mu \in P(\Sigma).\ \widehat{[\![p]\!]}(\mu)(\widetilde{[\![p]\!]}(\sigma)) \geqslant \mu(\sigma)$

Proof. Appendix B ■

3 Lipschitz Analysis

A function f is locally Lipschitz in a bounded set S if, $\forall x, x' \in S. \|fx - fx'\| \leqslant k \cdot \|x - x'\|$, where $\|\cdot\|$ can be any l_p norm. Quickly computing tight upper bounds on the local Lipschitzness constant (k) is an important requirement of our proof-search algorithm for probabilistic Lipschitzness of *pcat* programs. However, as mentioned previously, local Lipschitzness is a relational property (hyperproperty) and computing upper bounds on k can get expensive.

The problem can be made tractable by exploiting a known relationship between Lipschitz constants and directional directives of a function. Let f be a function of type $\mathbb{R}^m \to \mathbb{R}^n$, and let $S \subset \mathbb{R}^m$ be a convex bounded set. From [58] we know that the local Lipschitz constant of f in the region S can be upper bounded by the maximum value of the norm of the directional directives of f in S, where the directional directive, informally, is the derivative of f in the direction of some vector v. Since f is a vector-valued function (i.e., mapping vectors to vectors), the derivative (including directional derivative) of f appears as a matrix of the form, $\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \cdots & & \cdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$, referred to as the Jacobian matrix of f (with x and y referring to the input and output of f). Moreover, to compute the norm of $\mathbf{J}$, i.e. $\|\mathbf{J}\|$, we use the operator norm, $\|\mathbf{J}\| = \inf\{c \geqslant 0 \mid \|\mathbf{J}v\| \leqslant c\|v\| \text{ for all } v \in \mathbb{R}^m\}$. Intuitively, thinking of a matrix M as a linear operator mapping between two vector spaces, the operator norm of M measures the maximum amount by which a vector gets "stretched" when mapped using M.

For piecewise linear functions with a finite number of "pieces"(i.e., the type of functions that can be computed by *cat*), using lemma 3.3 from [58], we can compute an upper bound on the Lipschitz constant by computing the operator norm of the Jacobian of each linear piece, and picking the maximum value. Since each piece of the function is linear, computing the Jacobian for a piece is straightforward. But the number of pieces in piecewise linear functions represented by `NNs` (or *cat* programs) can be exponential in the number of layers in the `NN`, even in a bounded region S. Instead of computing the Jacobian for each piece, we instead define a static analysis inspired by the Fast-Lip algorithm presented in [57] that computes lower and upper bounds of each element (i.e., each partial derivative) appearing in the Jacobian. Since our analysis is sound, such an interval includes all the possible values of the partial derivative in a given convex region S. We describe this Jacobian analysis in the rest of the section.

3.1 Instrumented *cat* Semantics

We define an instrumented denotational semantics for *cat* (the non-probabilistic sublanguage of *pcat*) in Figure 3 that computes Jacobians for a particular program path, in addition to the standard meaning of the program (as defined in Figure 2). The semantics are notated by $\widetilde{[\![\cdot]\!]}_D$ (notice the subscript D). Program states, Σ^D, are pairs of maps such that the first element of each pair

$$
\begin{aligned}
\Sigma^D &\triangleq \Sigma \times (V \to ((\textstyle\bigcup_{m,n\in\mathbb{N}} (\mathbb{R})^{m\times n}) \times V) \\
\widetilde{[\![e]\!]}_D &: \Sigma^D \to \textstyle\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \times (V \to ((\bigcup_{m,n\in\mathbb{N}} (\mathbb{R})^{m\times n}) \\
\widetilde{[\![w \cdot x + \beta]\!]}_D(\sigma^D) &= \textbf{let } l = \textbf{dim}(w)_1 \textbf{ in} \\
&\quad \textbf{let } m = \textbf{dim}(w)_2 \textbf{ in} \\
&\quad \textbf{let } n = \textbf{dim}(\sigma_2^D(x)_1)_2 \textbf{ in} \\
&\quad \textbf{let } a = [\![w \cdot x + \beta]\!](\sigma_1^D) \textbf{ in} \\
&\quad \textbf{let } b = \\
&\quad \left[\sum_{i=1}^{m} w_{j,i} \cdot (((\sigma_2^D(x))_1)_{i,k}) \,\middle|\, j \in \{1,..,l\},\ k \in \{1,...,n\} \right] \textbf{ in} \\
&\quad (a, b)
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{[\![b]\!]}_D &: \Sigma^D \to \{\textbf{tt}, \textbf{ff}\} \\
\widetilde{[\![b]\!]}_D(\sigma^D) &= [\![b]\!](\sigma_1^D)
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{[\![s^-]\!]}_D &: \Sigma^D \to \Sigma^D \\
\widetilde{[\![\textbf{skip}]\!]}_D(\sigma^D) &= \sigma^D \\
\widetilde{[\![y \leftarrow w \cdot x + \beta]\!]}_D(\sigma^D) &= (\sigma_1^D[y \mapsto (\widetilde{[\![w \cdot x + \beta]\!]}_D(\sigma^D))_1], \sigma_2^D[y \mapsto ((\widetilde{[\![w \cdot x + \beta]\!]}_D(\sigma^D))_2, \sigma_2^D(x)_2)]) \\
\widetilde{[\![s_1; s_2]\!]}_D(\sigma^D) &= \widetilde{[\![s_2]\!]}_D(\widetilde{[\![s_1]\!]}_D(\sigma^D)) \\
\widetilde{[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]}_D(\sigma^D) &= \textbf{if } (\widetilde{[\![b]\!]}_D(\sigma^D) = \textbf{tt}) \textbf{ then } \widetilde{[\![s_1]\!]}_D(\sigma^D) \textbf{ else } \widetilde{[\![s_2]\!]}_D(\sigma^D)
\end{aligned}
$$

Fig. 3: *cat* denotational semantics instrumented with Jacobians

belongs to the previously defined set Σ of states, while the second element of each pair is a map that records the Jacobians. The second map is of type $V \to ((\bigcup_{m,n\in\mathbb{N}} (\mathbb{R})^{m\times n}) \times V)$, mapping each variable in V to a pair of values, namely, a Jacobian which is matrix of reals, and a variable in V. A *cat* program can map multiple input vectors to multiple output vectors, so one can compute a Jacobian of the *cat* program for each output vector with respect to each input vector. This explains the type of the second map in Σ^D - for each variable, the map records the corresponding Jacobian of the *cat* program computed with respect to the input variable that forms the second element of the pair.

Before explaining the semantics in Figure 3, we clarify the notation used in the figure. We use subscript indices, starting from 1, to refer to elements in a pair or a tuple. For instance, we can read $((\sigma_2^D(x))_1)_{i,k}$ in the definition of $\widetilde{[\![w \cdot x + \beta]\!]}_D$ as follows - σ_2^D refers to the second map of the σ^D pair, $\sigma_2^D(x)_1$ extracts the first element (i.e., the Jacobian matrix) of the pair mapped to variable x, and then finally, we extract the element at location (i, k) in the Jacobian matrix. Also, we use let expressions in a manner similar to ML, and list comprehensions similar to Haskell (though we extend the notation to handle matrices). **dim** is polymorphic and returns the dimensions of vectors and matrices.

The only interesting semantic definitions are the ones associated with the expression $w \cdot x + \beta$ and the statement $y \leftarrow w \cdot x + \beta$. The value associated with any variable in a *cat* program is always of the form, $w_n(w_{n-1}(...(w_2(w_1 \cdot x + \beta_1) + \beta_2)...) + \beta_{n-1}) + \beta_n = w_n \cdot w_{n-1} \cdot ... \cdot w_2 \cdot w_1 \cdot x + w_n \cdot w_{n-1} \cdot ... \cdot w_2 \cdot \beta_1 + w_n \cdot w_{n-1} \cdot ... \cdot w_3 \cdot \beta_2 + ... + \beta_n$. The derivative (the Jacobian) of this term with respect to x is $w_n \cdot w_{n-1} \cdot ... \cdot w_2 \cdot w_1$. Thus, calculating the Jacobian of a *cat* program for a particular output variable with respect to a particular input variable only

$$
\begin{array}{rl}
\Sigma^B & \triangleq V \to \bigcup_{n\in\mathbb{N}}(\mathbb{R}\times\mathbb{R})^n \\
\Sigma^L & \triangleq \Sigma^B \times (V \to ((\bigcup_{m,n\in\mathbb{N}}(\mathbb{R}\times\mathbb{R})^{m\times n}) \times (V \cup \{\bot,\top\}))\\
[\![e]\!]_{_L} & : \Sigma^L \to ((\bigcup_{n\in\mathbb{N}}(\mathbb{R}\times\mathbb{R})^n) \times (\bigcup_{m,n\in\mathbb{N}}(\mathbb{R}\times\mathbb{R})^{m\times n}))\\
[\![w\cdot x+\beta]\!]_{_L}(\sigma^L) = & \textbf{let } l = \textbf{dim}(w)_1 \textbf{ in}\\
& \textbf{let } m = \textbf{dim}(w)_2 \textbf{ in}\\
& \textbf{let } n = \textbf{dim}(\sigma_2^L(x)_1)_2 \textbf{ in}\\
& \textbf{let } a = [\![w\cdot x+\beta]\!]_{_B}(\sigma_1^L) \textbf{ in}\\
& \textbf{let } b = \\
& \left[\begin{array}{l|l}
\begin{array}{l}((\sum\limits_{i=1\wedge w_{j,i}\geqslant 0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_1 + \\
\sum\limits_{i=1\wedge w_{j,i}<0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_2) \\
(\sum\limits_{i=1\wedge w_{j,i}\geqslant 0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_2 + \\
\sum\limits_{i=1\wedge w_{j,i}<0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_1)\end{array} & j\in\{1,..,l\},\ k\in\{1,...,n\}
\end{array}\right] \textbf{ in}\\
& (a,b)
\end{array}
$$

$$
\begin{array}{rl}
\bigsqcup_{_L} & : \Sigma^L\times\Sigma^L\to\Sigma^L\\
\sigma^L \bigsqcup_{_L} \tilde{\sigma}^L = & ((\sigma_1^L \bigsqcup_{_B} \sigma_2^L),\\
& (\lambda v.\ \textbf{let } (m,n) = \textbf{dim}(\sigma_2^L(v)) \textbf{ in}\\
& \quad \textbf{if } (\sigma_2^L(v)_2 = \tilde{\sigma}_2^L(v)_2) \textbf{ then}\\
& \quad ([(\textbf{min}\{(\sigma_2^L(v)_1)_{i,j},(\tilde{\sigma}_2^L(v)_1)_{i,j}\}, \textbf{max}\{(\sigma_2^L(v)_1)_{i,j},(\tilde{\sigma}_2^L(v)_1)_{i,j}\}) \mid\\
& \quad\ i\in\{1,...,m\},\ j\in\{1,...,n\}], \sigma_2^L(v)_2)\\
& \quad \textbf{else } ([(-\infty,\infty) \mid i\in\{1,...,m\},\ j\in\{1,...,n\}],\top)
\end{array}
$$

$$
\begin{array}{rl}
[\![b]\!]_{_L} & : \Sigma^L\to\{\textbf{tt},\textbf{ff},\top\}\\
[\![b]\!]_{_L}(\sigma^L) = & [\![b]\!]_{_B}(\sigma_1^L)
\end{array}
$$

$$
\begin{array}{rl}
[\![s^-]\!]_{_L} & : \Sigma^L\to\Sigma^L\\
[\![\textbf{skip}]\!]_{_L}(\sigma^L) = & \sigma^L\\
[\![\textbf{assert } b]\!]_{_L}(\sigma^L) = & ((([\![\textbf{assert } b]\!]_{_B}(\sigma_1^L)),\sigma_2^L)\\
[\![y\leftarrow w\cdot x+\beta]\!]_{_L}(\sigma^L) = & (\sigma_1^L[y\mapsto([\![w\cdot x+\beta]\!]_{_L}(\sigma^L))_1],\sigma_2^L[y\mapsto(([\![w\cdot x+\beta]\!]_{_L}(\sigma^L))_2,\sigma_2^L(x)_2)])\\
[\![s_1;s_2]\!]_{_L}(\sigma^L) = & [\![s_2]\!]_{_L}([\![s_1]\!]_{_L}(\sigma^L))\\
[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]_{_L}(\sigma^L) = & \textbf{if } \quad ([\![b]\!]_{_L}(\sigma^L)=\textbf{tt}) \textbf{ then } [\![s_1]\!]_{_L}(\sigma^L)\\
& \textbf{else if } ([\![b]\!]_{_L}(\sigma^L)=\textbf{ff}) \textbf{ then } [\![s_2]\!]_{_L}(\sigma^L)\\
& \textbf{else } \quad [\![s_1]\!]_{_L}([\![\textbf{assert } b]\!]_{_L}(\sigma^L)) \bigsqcup_{_L} [\![s_2]\!]_{_L}([\![\textbf{assert } \neg b]\!]_{_L}(\sigma^L))
\end{array}
$$

Fig. 4: *cat* abstract semantics for Jacobian analysis

requires multiplying the relevant weight matrices together and the bias terms can be ignored. This is exactly how we define the semantics of $w\cdot x+\beta$.

3.2 Jacobian Analysis

The abstract version of the instrumented denotational semantics of *cat* is defined in Figure 4. The semantics are notated by $[\![\cdot]\!]_{_L}$ (notice the subscript L). The analysis computes box-shaped overapproximations of all the possible outcomes of a *cat* program when executed on inputs from a box-shaped bounded set. This is similar to standard interval analysis except that *cat* operates on data of base type of real vectors. The analysis maintains bounds on real vectors by computing

intervals for every element of a vector. In addition, this analysis also computes an overapproximation of all the possible Jacobian matrices. Note that the Jacobian matrices computed by the instrumented semantics of *cat* only depend on the path through the program, i.e. the entries in the computed Jacobian are control-dependent on the program inputs but not data-dependent. Consequently, for precision, it is essential that our analysis exhibit some notion of path-sensitivity. We achieve this by evaluating the branch conditions using the computed intervals and abstractly interpreting both the branches of an **if then else** statement only if the branch direction cannot be resolved.

An abstract program state, $\sigma^L \in \Sigma^L$, is a pair of maps. The first map in an abstract state maps variables in V to abstract vectors representing a box-shaped set of vectors. Each element of an abstract vector is pair of reals representing a lower bound and an upper bound on the possible values (first element of the pair is the lower bound and second element is the upper bound). The second map in an abstract state maps variables in V to pairs of abstract Jacobian matrices and elements in V extended with a top and a bottom element. Like abstract vectors, each element of an abstract Jacobian matrix is a pair of reals representing lower and upper bounds of the corresponding partial derivative.

The definition of the abstract semantics is straightforward but we describe the abstract semantics for affine expressions and for conditional statements. First, we discuss affine expressions. As a quick reminder of the notation, a term of the form $(((\sigma_2^L(x))_1)_{i,k})_1$ represents the lower bound of the element at location (i, k) in the abstract Jacobian associated with variable x. Now, recall that the instrumented semantics computes Jacobians simply by multiplying the weight matrices. In the abstract semantics, we multiply abstract Jacobians such that the bounds on each abstract element in the output abstract Jacobian reflect the minimum and maximum possible values that the element could take given the input abstract Jacobians. The abstract vectors for the first map are computed using the abstract box semantics (notated by $[\![\cdot]\!]_B$), defined in Appendix G.For conditional statements, as mentioned previously, we first evaluate the branch condition using the abstract state. If this evaluation returns $\top$, meaning that the analysis was unable to discern the branch to be taken, we abstractly interpret both the branches and then join the computed abstract states. Note that before abstractly interpreting both branches, we update the abstract state to reflect that the branch condition should hold before executing s_1 and should not hold before executing s_2. However, the **assert** b statement is not a part of the *cat* language, and only used for defining the abstract semantics. The join operation $(\bigcup_L)$ is as expected, except for one detail that we want to highlight - in case the Jacobians along different branches are computed with respect to different input variables we make the most conservative choice when joining the abstract Jacobians, bounding each element with $(-\infty, \infty)$ as well as recording $\top$ for the input variable.

Next, we define the concretization function (γ_L) for the abstract program states that maps elements in Σ^L to sets of elements in Σ^D and then state the soundness theorem for our analysis.

Definition 5. *(Concretization function for Jacobian analysis)*
$\gamma_L(\sigma^L) = \{\sigma^D \mid (\bigwedge_{v \in V} . \sigma_1^L(v)_1 \leqslant \sigma_1^D(v) \leqslant \sigma_1^L(v)_2) \wedge (\bigwedge_{v \in V} . (\sigma_2^L(v)_1)_1 \leqslant \sigma_2^D(v)_1 \leqslant (\sigma_2^L(v)_1)_2) \wedge \sigma_2^D(v)_2 \in \gamma_V(\sigma_2^L(v)_2)\}$ *where* $\gamma_V(v) = v$ *and* $\gamma_V(\top) = V$

Theorem 6. *(Soundness of Jacobian analysis)*
$\forall p \in s^-, \sigma^L \in \Sigma^L . \{\widetilde{[\![p]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![p]\!]_L(\sigma^L))$

Proof. Appendix C ■

We next define the notion of operator norm of an abstract Jacobian. This definition is useful for stating Corollary 8. Given an abstract Jacobian, we construct a matrix J such that every element of J is the maximum of the absolute values of the corresponding lower and upper bound in the abstract Jacobian.

Definition 7. *(Operator norm of abstract Jacobian)*
If $J = \sigma_2^L(v)_1$ *for some* σ^L *and* v*, and* $(m, n) = \mathbf{dim}(J)$ *then* $\|J\|_L$ *is defined as,*
$\|J\|_L = \|[\mathbf{max}\{|(J_{k,l})_1|, |(J_{k,l})_2|\} \mid k \in \{1, ..., m\}, l \in \{1, ..., n\}]\|$

Corollary 8 shows that the operator norm of the abstract Jacobian computed by the analysis for some variable v is an upper bound of the operator norms of all the Jacobians possible for v when a program p is executed on the set of inputs represented by $\gamma_L(\sigma^L)$, for any program p and any abstract state σ^L.

Corollary 8. *(Upper bound of Jacobian operator norm)*
$\forall p \in s^-, \sigma^L \in \Sigma^L, v \in V.$
$\mathbf{max}\{\left\| ((\widetilde{[\![p]\!]}_D(\sigma^D))_2)(v)_1 \right\| \mid \sigma^D \in \gamma_L(\sigma^L)\} \leqslant \|((([\![p]\!]_L(\sigma^L))_2(v))_1\|_L$

Proof. Appendix D ■

3.3 Box Analysis

The box analysis abstracts the lowered *cat* semantics instead of the instrumented semantics. Given a box-shaped set of input states, it computes box-shaped over-approximations of the program output in a manner similar to the Jacobian analysis. In fact, the box analysis only differs from the Jacobian analysis in not computing abstract Jacobians. We define a separate box analysis to avoid computing abstract Jacobians when not needed. The concretization function (γ_B) for the box analysis and the soundness theorem are stated below. However, we do not provide a separate proof of soundness for the box analysis since such a proof is straightforward given the soundness proof for the Jacobian analysis. Details of the box analysis are available in Appendix G.

Definition 9. *(Concretization function for box analysis)*
$\gamma_B(\sigma^B) = \{\sigma \mid \bigwedge_{v \in V} . \sigma^B(v)_1 \leqslant \sigma(v) \leqslant \sigma^B(v)_2\}$

Theorem 10. *(Soundness of box analysis)*
$\forall p \in s^-, \sigma^B \in \Sigma^B . \{\widetilde{[\![p]\!]}(\sigma) | \sigma \in \gamma_B(\sigma^B)\} \subseteq \gamma_B([\![p]\!]_B(\sigma^B))$

4 Algorithms

We now describe our proof-search algorithms for probabilistic Lipschitzness of NNs. The PROLIP algorithm (Section 4.1) is an algorithmic primitive that can be used by a proof-search algorithm for probabilistic Lipschitzness. We provide the sketch of such an algorithm using PROLIP in Section 4.2.

4.1 PROLIP Algorithmic Primitive

The PROLIP algorithm expects a *pcat* program p of the form $z \leftsquigarrow N(0,1); g; f$ as input, where g and f are *cat* programs. $z \leftsquigarrow N(0,1); g$ represents the generative model and f represents the NN under analysis. Other inputs expected by PROLIP are a box-shaped region z_B in z and the input variable as well as the output variable of f (**in** and **out** respectively). Typically, NNs consume a single input and produce a single output. The outputs produced by PROLIP are (i) k_U, an upper bound on the local Lipschitzness constant of f in a box-shaped region of **in** (say $\mathbf{in}_B$) that overapproximates the set of **in** values in the image of z_B under g, (ii) d, the maximum distance between **in** values in $\mathbf{in}_B$, (iii) vol, the probabilistic volume of the region $z_B \times z_B$ with respect to the distribution $N(0,1) \times N(0,1)$.

Algorithm 1: PROLIP algorithmic primitive

Input:
p: *pcat* program.
z_B: Box in z.
in: Input variable of f.
out: Output variable of f.
Output:
k_U: Lipschitz constant.
d: Max **in** distance.
vol: Mass of $z_B \times z_B$.

1 $\sigma^B := \lambda v.(-\infty, \infty)$;
2 $\tilde{\sigma}^B := [\![g]\!]_B(\sigma^B[z \mapsto z_B])$;
3 $\sigma^L := (\tilde{\sigma}^B, \lambda v.(I, v))$;
4 $\tilde{\sigma}^L := [\![f]\!]_L(\sigma^L)$;
5 **if** $(\tilde{\sigma}_2^L(\mathbf{out})_2 = \mathbf{in})$ **then**
6 $\quad J := \tilde{\sigma}_2^L(\mathbf{out})_1$;
7 $\quad k_U := \|J\|_L$;
8 **else**
9 $\quad k_U := \infty$;
10 $d :=$ DIAG_LEN$(\tilde{\sigma}^B(\mathbf{in}))$;
11 $vol :=$ VOL$(N \times N, z_B \times z_B)$;
12 **return** (k_U, d, vol);

PROLIP starts by constructing an initial abstract program state (σ^B) suitable for the box analysis (line 1). σ^B maps every variable in V to abstract vectors with elements in the interval $(-\infty, \infty)$. We assume that for the variables accessed in p, the length of the abstract vectors is known, and for the remaining variables we just assume vectors of length one in this initial state. Next, the initial entry in σ^B for z is replaced by z_B, and this updated abstract state is used to perform box analysis of g, producing $\tilde{\sigma}^B$ as the result (line 2). Next, $\tilde{\sigma}^B$ is used to create the initial abstract state σ^L for the Jacobian analysis of f (line 3). Initially, every variable is mapped to an identity matrix as the Jacobian and itself as the variable with respect to which the Jacobian is computed. The initial Jacobian is a square matrix with side length same as that of the abstract vector associated with the variable being mapped. Next, we use σ^L to perform Jacobian analysis of f producing $\tilde{\sigma}^L$ as the result (line 4). If the abstract Jacobian mapped to **out** in $\tilde{\sigma}^L$ is computed with respect to **in** (line 5), we proceed down

the true branch else we assume that nothing is known about the required Jacobian and set k_U to ∞ (line 9). In the true branch, we first extract the abstract Jacobian and store it in J (line 6). Next, we compute the operator norm of the abstract Jacobian J using Definition 7, giving us the required upper bound on the Lipschitz constant (line 7). We then compute the maximum distance between **in** values in the box described by $\tilde{\sigma}^B(\mathbf{in})$ using the procedure `DIAG_LEN` that just computes the length of the diagonal of the hyperrectangle represented by $\tilde{\sigma}^B(\mathbf{in})$ (line 10). We also compute the probabilistic mass of region $z_B \times z_B$ with respect to the distribution $N(0,1) \times N(0,1)$ (line 11). This is an easy computation since we can form an analytical expression and just plug in the boundaries of z_B. Finally, we return the tuple (k_U, d, vol) (line 12). This `PROLIP` algorithm is correct as stated by the following theorem.

Theorem 11. *(Soundness of* `PROLIP`*)*
Let $p = z \leftsquigarrow N(0,1); g; f$ *where* $g, f \in s^-$, $(k_U, d, vol) = \texttt{PROLIP}(p, z_B)$, $z \notin \mathbf{outv}(g), z \notin \mathbf{outv}(f), x \in \mathbf{inv}(f)$, *and* $y \in \mathbf{outv}(f)$ *then,* $\forall \sigma_0 \in \Sigma$.

$$\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} ((\|\sigma(y) - \sigma'(y)\| \leqslant k_U \cdot \|\sigma(x) - \sigma'(x)\|) \wedge (\sigma(z), \sigma'(z) \in \gamma(z_B))) \geqslant vol$$

Proof. Appendix E ■

This theorem is applicable for any program p in the required form, such that g and f are *cat* programs, variable z is not written to by g and f ($\mathbf{outv}(\cdot)$ gives the set of variables that a program writes to, $\mathbf{inv}(\cdot)$ gives the set of live variables at the start of a program). It states that the result (k_U, d, vol) of invoking `PROLIP` on p with box z_B is safe, i.e., with probability at least vol, any pair of program states (σ, σ'), randomly sampled from the distribution denoted by $[\![p]\!](\sigma_0)$, where σ_0 is any initial state, satisfies the Lipschitzness property (with constant k_U) and has z variables mapped to vectors in the box z_B.

4.2 Sketch of Proof-Search Algorithm

We give a sketch of a proof-search algorithm that uses the `PROLIP` algorithm as a primitive. The inputs to such an algorithm are a *pcat* program p in the appropriate form, the constants r, ϵ, and k that appear in the formulation of probabilistic Lipschitzness, and a resource bound `gas` that limits the number of times `PROLIP` is invoked. This algorithm either finds a proof or runs out of `gas`. Before describing the algorithm, we recall the property we are trying to prove, stated as follows,

$$\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} (\|\sigma(y) - \sigma'(y)\| \leqslant k * \|\sigma(x) - \sigma'(x)\| \mid \|\sigma(x) - \sigma'(x)\| \leqslant r) \geqslant 1 - \epsilon$$

The conditional nature of this probabilistic property complicates the design of the proof-search algorithm, and we use the fact that $Pr(A \mid B) = Pr(A \wedge B)/Pr(B)$ for computing conditional probabilities. Accordingly, the algorithm maintains three different probability counters, namely, pr_l, pr_r, and pr_f, which are all initialized to zero as the first step (line 1).

Algorithm 2: Checking Probabilistic Robustness.

Input:
p: *pcat* program.
r: Input closeness bound.
ϵ: Probabilistic bound.
k: Lipschitz constant.
gas: Iteration bound.
Output: $\{\mathbf{tt}, ?\}$

```
1  pr_l := 0; pr_r := 0; pr_f := 0;
2  α := INIT_AGENT(dim(z), r, ϵ, k);
3  while (pr_l < (1 − ϵ)) ∧ (gas ≠ 0)
   do
4  |  gas := gas − 1;
5  |  z_B := CHOOSE(α);
6  |  (k_U, d, vol) :=
   |   PROLIP(p, z_B, x, y);
7  |  UPDATE_AGENT(α, k_U, d, vol);
8  |  if d ⩽ r then
9  |  |  pr_r := pr_r + vol;
10 |  |  if k_U ⩽ k then
11 |  |  |  pr_l := pr_l + vol;
12 |  |  |  pr_f := pr_f/pr_r;
13 end while
14 if gas = 0 then
15 |  return ? ;
16 else
17 |  return tt ;
```

pr_l records the probability that a randomly sampled pair of program states (σ, σ') satisfies the Lipschitzness and closeness property (i.e., $(\|\sigma(y) - \sigma'(y)\| \leqslant k * \|\sigma(x) - \sigma'(x)\|) \wedge (\|\sigma(x) - \sigma'(x)\| \leqslant r)$). pr_r records the probability that a randomly sampled pair of program states satisfies the closeness property (i.e., $\|\sigma(x) - \sigma'(x)\| \leqslant r$). pr_f tracks the conditional probability which is equal to pr_l/pr_r. After initializing the probability counters, the algorithm initializes an "agent" (line 2), which we think of as black-box capable of deciding which box-shaped regions in z should be explored. Ideally, we want to pick a box such that - (i) it has a high probability mass, (ii) it satisfies, both, Lipschitzness and closeness. Of course, we do not know a priori if Lipschitzness and closeness will hold for a particular box in z, the crux of the challenge in designing a proof-search algorithm. Here, we leave the algorithm driving the agent's decisions unspecified (and hence, refer to the proof-search algorithm as a sketch). After initializing the agent, the algorithm enters a loop (lines 3 - 13) that continues till we have no gas left or we have found a proof. Notice that if $(pr_l \geqslant (1 - \epsilon))$, the probabilistic Lipschitzness property is certainly true, but this is an overly strong condition that maybe false even when probabilistic Lipschitzness holds. For instance, if ϵ was 0.1 and the ground-truth value of pr_r for the program p was 0.2, then pr_l could never be $\geqslant 0.9$, even if probabilistic Lipschitzness holds. However, continuing with our algorithm description, after decrementing gas (line 4), the algorithm queries the agent for a box in z (line 5), and runs PROLIP with this box, assuming x as the input variable of f and y as the output (line 6). Next, the agent is updated with the result of calling PROLIP, allowing the agent to update it's internal state (line 7). Next, we check if for the currently considered box (z_B), the maximum distance between the inputs to f is less than r (line 8), and if so, we update the closeness probability counter pr_r (line 9). We also check if the upper bound of the local Lipschitzness constant returned by PROLIP is less than k (line 10), and if so, update pr_l (line 11) and

pr_f (line 12). Finally, if we have exhausted the `gas`, we were unable to prove the property, otherwise we have a proof of probabilistic Lipschitzness.

4.3 Discussion

Informally, we can think of the Jacobian analysis as computing two different kinds of "information" about a neural network: (i) an overapproximation of the outputs, given a set of inputs σ^B, using the box analysis; (ii) an upper bound on the local Lipschitz constant of the neural network for inputs in σ^B. The results of the box analysis are used to overapproximate the set of "program paths" in the neural network exercised by inputs in σ^B, safely allowing the Jacobian computation to be restricted to this set of paths. Consequently, it is possible to replace the use of box domain in (i) with other abstract domains like zonotopes [30] or DeepPoly [49] for greater precision in overapproximating the set of paths. In contrast, one needs to be very careful with the abstract domain used for the analysis of the generative model g in Algorithm 1, since the choice of the abstract domain has a dramatic effect on the complexity of the volume computation algorithm `VOL` invoked by the `PROLIP` algorithm. While Gaussian volume computation of boxes is easy, it is hard for general convex bodies [4,25,23] unless one uses randomized algorithms for volume computation [24,20]. Finally, note that the design of a suitable agent for iteratively selecting the input regions to analyze in Algorithm 2 remains an open problem.

5 Empirical Evaluation

We aim to empirically evaluate the computational complexity of `PROLIP`. We ask the following questions: **(RQ1)** Given a program, is the run time of `PROLIP` affected by the size and location of the box in z? **(RQ2)** What is the run time of `PROLIP` on popular generative models and `NNs`?

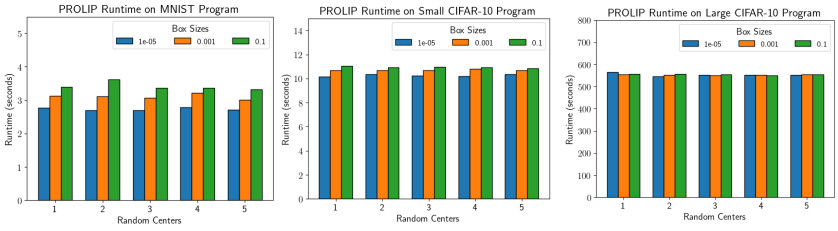
5.1 Experimental Setup

We implement `PROLIP` in Python, using Pytorch, Numpy, and SciPy for the core functionalities, and Numba for program optimization and parallelization. We run `PROLIP` on three *pcat* programs corresponding to two datasets: the MNIST dataset and the CIFAR-10 dataset. Each program has a generator network g and a classifier network f. The g networks in each program consist of five convolution transpose layers, four batch norm layers, four ReLU layers, and a tanh layer. The full generator architectures and parameter weights can be seen in [48]. The f network for the MNIST program consists of three fully connected layers and two ReLU layers. For the CIFAR-10 dataset, we create two different *pcat* programs: one with a large classifier architecture and one with a small classifier architecture. The f network for the large CIFAR-10 program consists of seven convolution layers, seven batch norm layers, seven ReLU layers, four maxpool layers, and one fully connected layer. The f network for the small CIFAR-10 program consists

of two convolution layers, two maxpool layers, two ReLU layers, and three fully connected layers. The full classifier architectures and parameter weights for the MNIST and large CIFAR-10 program can be seen in [17].

In our experiments, each generative model has a latent space dimension of 100, meaning that the model samples a vector of length 100 from a multi-dimensional normal distribution, which is then used by the generator network. We create five random vectors of length 100 by randomly sampling each element of the vectors from a normal distribution. For each vector, we create three different sized square boxes by adding and subtracting a constant from each element in the vector. This forms an upper and lower bound for the randomly-centered box. The constants we chose to form these boxes are 0.00001, 0.001, and 0.1. In total, 15 different data points are collected for each program. We ran these experiments on a Linux machine with 32 vCPU's, 204 GB of RAM, and no GPU.

5.2 Results



(a) (b) (c)

Fig. 5: PROLIP run times

RQ1. As seen in Figures 5a and 5b, there is a positive correlation between box size and run time of PROLIP on the MNIST and small CIFAR-10 programs. This is likely because as the z input box size increases, more branches in the program stay unresolved, forcing the analysis to reason about more of the program. However, z box size does not seem to impact PROLIP run time on the large CIFAR-10 program (Figure 5c) as the time spent in analyzing convolution layers completely dominates any effect on run time of the increase in z box size.

RQ2. There is a significant increase in the run time of PROLIP for the large CIFAR-10 program compared to the MNIST and small CIFAR-10 programs, and this is due to the architectures of their classifiers. When calculating the abstract Jacobian matrix for an affine assignment statement ($y \leftarrow w \cdot x + \beta$), we multiply the weight matrix with the incoming abstract Jacobian matrix. The dimensions of a weight matrix for a fully connected layer is $N_{in} \times N_{out}$ where N_{in} is the number of input neurons and N_{out} is the number of output neurons. The dimensions of a weight matrix for a convolution layer are $C_{out} \cdot H_{out} \cdot W_{out} \times C_{in} \cdot$

$H_{in} \cdot W_{in}$ where C_{in}, H_{in}, and W_{in} are the input's channel, height, and width dimensions and C_{out}, H_{out}, and W_{out} are the output's channel, height, and width dimensions. For our MNIST and small CIFAR-10 classifiers, the largest weight matrices formed had dimensions of 784×256 and 4704×3072 respectively. In comparison, the largest weight matrix calculated in the large CIFAR-10 classifier had a dimension of 131072×131072. Propagating the Jacobian matrix for the large CIFAR-10 program requires first creating a weight matrix of that size, which is memory intensive, and second, multiplying the matrix with the incoming abstract Jacobian matrix, which is computationally expensive. The increase in run time of the PROLIP algorithm can be attributed to the massive size blow-up in the weight matrices computed for convolution layers.

Other Results. Table 1 shows the upper bounds on local Lipschitz constant computed by the PROLIP algorithm for every combination of box size and *pcat* program considered in our experiments. The computed upper bounds are comparable to those computed by the Fast-Lip algorithm from [57] as well as other state-of-the-art approaches for computing Lipschitz constants of neural networks. A phenomenon observed in our experiments is the convergence of local Lipschitz constants to an upper bound, as the z box size increases. This occurs because beyond a certain z box size, for every box in z, the output bounds of g represent the entire input space for f. Therefore any increase in the z box size, past the tipping point, results in computing an upper bound on the global Lipschitz constant of f.

Box Size	**MNIST Lip Constant**	**Large CIFAR Lip Constant**	**Small CIFAR Lip Constant**
1e-05	1.683e1	5.885e14	3.252e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.072e1	5.331e14	1.814e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.460e1	6.740e14	2.719e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.754e1	6.571e14	2.868e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.312e1	5.647e14	2.884e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5

Table 1: Local Lipschitz constants discovered by PROLIP

The run time of the PROLIP algorithm can be improved by utilizing a GPU for matrix multiplication. The multiplication of massive matrices computed in the Jacobian propagation of convolution layers or large fully connected layers accounts for a significant portion of the run time of PROLIP, and the run time can benefit from GPU-based parallelization of matrix multiplication. Another factor that slows down our current implementation of PROLIP algorithm is the creation of the weight matrix for a convolution layer. These weight matrices are quite sparse, and constructing sparse matrices that hold '0' values implicitly can be much faster than explicitly constructing the entire matrix in memory, which is what our current implementation does.

6 Related Work

Our work draws from different bodies of literature, particularly literature on *verification of* NNs, *Lipschitz analysis of programs and* NNs, and *semantics and verification of probabilistic programs*. These connections and influences have been described in detail in Section 1. Here, we focus on describing connections with existing work on proving probabilistic/statistical properties of NNs.

[44] is the source of the probabilistic Lipschitzness property that we consider. They propose a proof-search algorithm that (i) constructs a product program [5], (ii) uses an abstract interpreter with a powerset polyhedral domain to compute input pre-conditions that guarantee the satisfaction of the Lipschitzness property, (iii) computes approximate volumes of these input regions via sampling. They do not implement this algorithm. If one encodes the Lipschitzness property as disjunction of polyhedra, the number of disjuncts is exponential in the number of dimensions of the output vector. There is a further blow-up in the number of disjuncts as we propagate the abstract state backwards.

Other works on probabilistic properties of NNs [55,56] focus on local robustness. Given an input x_0, and an input distribution, they compute the probability that a random sample x' drawn from a ball centered at x_0 causes non-robust behavior of the NN at x' compared with x_0. [55] computes these probabilities via sampling while [56] constructs analytical expressions for computing upper and lower bounds of such probabilities. Finally, [3] presents a model-counting based approach for proving quantitative properties of NNs. They translate the NN as well as the property of interest into SAT constraints, and then invoke an approximate model-counting algorithm to estimate the number of satisfying solutions. We believe that their framework may be general enough to encode our problem but the scalability of such an approach remains to be explored. We also note that the guarantees produced by [3] are statistical, so one is unable to claim with certainty if probabilistic Lipschitzness is satisfied or violated.

7 Conclusion

We study the problem of algorithmically proving probabilistic Lipschitzness of NNs with respect to generative models representing input distributions. We em-

ploy a language-theoretic lens, thinking of the generative model and `NN`, together, as programs of the form $z \leftsquigarrow N(0,1); g; f$ in a first-order, imperative, probabilistic programming language *pcat*. We develop a sound local Lipschitzness analysis for *cat*, a non-probabilistic sublanguage of *pcat* that performs a Jacobian analysis under the hood. We then present `PROLIP`, a provably correct algorithmic primitive that takes in a box-shaped region in the latent space of the generative model as an input, and returns a lower bound on the volume of this region as well as an upper bound on a local Lipschitz constant of f. Finally, we sketch a proof-search algorithm that uses `PROLIP` and avoids expensive volume computation operations in the process of proving theorems about probabilistic programs. Empirical evaluation of the computational complexity of `PROLIP` suggests its feasibility as an algorithmic primitive, although convolution-style operations can be expensive and warrant further investigation.

References

1. Albarghouthi, A., D'Antoni, L., Drews, S., Nori, A.V.: FairSquare: Probabilistic verification of program fairness. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 80:1–80:30 (Oct 2017)
2. Alzantot, M., Sharma, Y., Elgohary, A., Ho, B.J., Srivastava, M., Chang, K.W.: Generating Natural Language Adversarial Examples. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 2890–2896. Association for Computational Linguistics, Brussels, Belgium (Oct 2018)
3. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative Verification of Neural Networks and Its Security Applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1249–1264. CCS '19, Association for Computing Machinery, London, United Kingdom (Nov 2019)
4. Bárány, I., Füredi, Z.: Computing the volume is difficult. Discrete & Computational Geometry **2**(4), 319–326 (Dec 1987)
5. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004. pp. 100–114 (Jun 2004)
6. Barthe, G., Crespo, J.M., Kunz, C.: Relational Verification Using Product Programs. In: Butler, M., Schulte, W. (eds.) FM 2011: Formal Methods. pp. 200–214. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011)
7. Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing Probabilistic Invariants via Doob's Decomposition. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 43–61. Lecture Notes in Computer Science, Springer International Publishing, Cham (2016)
8. Barthe, G., Espitau, T., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: An Assertion-Based Program Logic for Probabilistic Programs. In: Ahmed, A. (ed.) European Symposium on Programming Languages and Systems. pp. 117–144. Lecture Notes in Computer Science, Springer International Publishing, Cham (2018)
9. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.Y.: Proving expected sensitivity of probabilistic programs. Proceedings of the ACM on Programming Languages **2**(POPL), 57:1–57:29 (Dec 2017)

10. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring Neural Net Robustness with Constraints. In: Lee, D.D., Sugiyama, M., Luxburg, U.V., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29, pp. 2613–2621. Curran Associates, Inc. (2016), http://papers.nips.cc/paper/6339-measuring-neural-net-robustness-with-constraints.pdf
11. Bastani, O., Zhang, X., Solar-Lezama, A.: Probabilistic verification of fairness properties via concentration. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 118:1–118:27 (Oct 2019)
12. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 14–25. POPL '04, Association for Computing Machinery, Venice, Italy (Jan 2004)
13. Carlini, N., Mishra, P., Vaidya, T., Zhang, Y., Sherr, M., Shields, C., Wagner, D., Zhou, W.: Hidden Voice Commands. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 513–530 (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/carlini
14. Carlini, N., Wagner, D.: Audio Adversarial Examples: Targeted Attacks on Speech-to-Text. In: 2018 IEEE Security and Privacy Workshops (SPW). pp. 1–7 (May 2018)
15. Chakarov, A., Sankaranarayanan, S.: Probabilistic Program Analysis with Martingales. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 511–526. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013)
16. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving programs robust. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 102–112. ESEC/FSE '11, Association for Computing Machinery, Szeged, Hungary (Sep 2011)
17. Chen, A.: Aaron-xichen/pytorch-playground (May 2020), https://github.com/aaron-xichen/pytorch-playground
18. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium. pp. 51–65 (Jun 2008)
19. Combettes, P.L., Pesquet, J.C.: Lipschitz Certificates for Neural Network Structures Driven by Averaged Activation Operators. arXiv:1903.01014 [math] (Jul 2019), http://arxiv.org/abs/1903.01014
20. Cousins, B., Vempala, S.: Gaussian Cooling and $O^{\hat{*}}(n\hat{3})$ Algorithms for Volume and Gaussian Volume. SIAM Journal on Computing **47**(3), 1237–1273 (Jan 2018)
21. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 84–96. POPL '78, Association for Computing Machinery, Tucson, Arizona (Jan 1978)
22. Cousot, P., Monerau, M.: Probabilistic Abstract Interpretation. In: Seidl, H. (ed.) European Symposium on Programming Languages and Systems. pp. 169–193. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012)
23. Dyer, M.E., Frieze, A.M.: On the Complexity of Computing the Volume of a Polyhedron. SIAM Journal on Computing **17**(5), 967–974 (Oct 1988)
24. Dyer, M., Frieze, A., Kannan, R.: A random polynomial-time algorithm for approximating the volume of convex bodies. Journal of the ACM **38**(1), 1–17 (Jan 1991)
25. Elekes, G.: A geometric inequality and the complexity of computing volume. Discrete & Computational Geometry **1**(4), 289–292 (Dec 1986)

26. Fazlyab, M., Robey, A., Hassani, H., Morari, M., Pappas, G.: Efficient and Accurate Estimation of Lipschitz Constants for Deep Neural Networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., d\textquotesingle Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 11427–11438. Curran Associates, Inc. (2019), http://papers.nips.cc/paper/9319-efficient-and-accurate-estimation-of-lipschitz-constants-for-deep-neural-networks.pdf
27. Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., Vechev, M.: DL2: Training and Querying Neural Networks with Logic. In: International Conference on Machine Learning. pp. 1931–1941 (May 2019), http://proceedings.mlr.press/v97/fischer19a.html
28. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 3–18 (May 2018)
29. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 166–176. ISSTA 2012, Association for Computing Machinery, Minneapolis, MN, USA (Jul 2012)
30. Ghorbal, K., Goubault, E., Putot, S.: The Zonotope Abstract Domain Taylor1+. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 627–633. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2009)
31. Gibbons, J.: APLicative Programming with Naperian Functors. In: Yang, H. (ed.) European Symposium on Programming Languages and Systems. pp. 556–583. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2017)
32. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative Adversarial Nets. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 27, pp. 2672–2680. Curran Associates, Inc. (2014), http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf
33. Gouk, H., Frank, E., Pfahringer, B., Cree, M.: Regularisation of Neural Networks by Enforcing Lipschitz Continuity. arXiv:1804.04368 [cs, stat] (Sep 2018), http://arxiv.org/abs/1804.04368
34. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37. pp. 448–456. ICML'15, JMLR.org, Lille, France (Jul 2015)
35. Jia, R., Liang, P.: Adversarial Examples for Evaluating Reading Comprehension Systems. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing. pp. 2021–2031. Association for Computational Linguistics, Copenhagen, Denmark (Sep 2017)
36. Katoen, J.P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-Invariant Generation for Probabilistic Programs:. In: Cousot, R., Martel, M. (eds.) Static Analysis. pp. 390–406. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2010)
37. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 97–117. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017)

38. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.: The Marabou Framework for Verification and Analysis of Deep Neural Networks. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 443–452. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019)
39. Kingma, D.P., Welling, M.: Auto-Encoding Variational Bayes. arXiv:1312.6114 [cs, stat] (May 2014), http://arxiv.org/abs/1312.6114
40. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates, Inc. (2012), http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
41. Lamport, L.: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering **3**(2), 125–143 (Mar 1977)
42. Latorre, F., Rolland, P., Cevher, V.: Lipschitz constant estimation of Neural Networks via sparse polynomial optimization. arXiv:2004.08688 [cs, stat] (Apr 2020), http://arxiv.org/abs/2004.08688
43. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for Verifying Deep Neural Networks. arXiv:1903.06758 [cs, stat] (Mar 2019), http://arxiv.org/abs/1903.06758
44. Mangal, R., Nori, A.V., Orso, A.: Robustness of neural networks: A probabilistic and practical approach. In: Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results. pp. 93–96. ICSE-NIER '19, IEEE Press, Montreal, Quebec, Canada (May 2019)
45. Qin, Y., Carlini, N., Cottrell, G., Goodfellow, I., Raffel, C.: Imperceptible, Robust, and Targeted Adversarial Examples for Automatic Speech Recognition. In: International Conference on Machine Learning. pp. 5231–5240 (May 2019), http://proceedings.mlr.press/v97/qin19a.html
46. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 112–122. PLDI '14, Association for Computing Machinery, Edinburgh, United Kingdom (Jun 2014)
47. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 447–458. PLDI '13, Association for Computing Machinery, Seattle, Washington, USA (Jun 2013)
48. Singh, C.: Csinva/gan-vae-pretrained-pytorch (May 2020), https://github.com/csinva/gan-vae-pretrained-pytorch
49. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 41:1–41:30 (Jan 2019)
50. Slepak, J., Shivers, O., Manolios, P.: An Array-Oriented Language with Static Rank Polymorphism. In: Shao, Z. (ed.) European Symposium on Programming Languages and Systems. pp. 27–46. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2014)
51. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. In: International Conference on Learning Representations (2014), http://arxiv.org/abs/1312.6199

52. Tsuzuku, Y., Sato, I., Sugiyama, M.: Lipschitz-margin training: Scalable certification of perturbation invariance for deep neural networks. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. pp. 6542–6551. NIPS'18, Curran Associates Inc., Montréal, Canada (Dec 2018)
53. Virmaux, A., Scaman, K.: Lipschitz regularity of deep neural networks: Analysis and efficient estimation. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 3835–3844. Curran Associates, Inc. (2018), http://papers.nips.cc/paper/7640-lipschitz-regularity-of-deep-neural-networks-analysis-and-efficient-estimation.pdf
54. Wang, D., Hoffmann, J., Reps, T.: PMAF: An algebraic framework for static analysis of probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 513–528. PLDI 2018, Association for Computing Machinery, Philadelphia, PA, USA (Jun 2018)
55. Webb, S., Rainforth, T., Teh, Y.W., Kumar, M.P.: A Statistical Approach to Assessing Neural Network Robustness. In: International Conference on Learning Representations (Sep 2018), https://openreview.net/forum?id=S1xcx3C5FX
56. Weng, L., Chen, P.Y., Nguyen, L., Squillante, M., Boopathy, A., Oseledets, I., Daniel, L.: PROVEN: Verifying Robustness of Neural Networks with a Probabilistic Approach. In: International Conference on Machine Learning. pp. 6727–6736 (May 2019), http://proceedings.mlr.press/v97/weng19a.html
57. Weng, L., Zhang, H., Chen, H., Song, Z., Hsieh, C.J., Daniel, L., Boning, D., Dhillon, I.: Towards Fast Computation of Certified Robustness for ReLU Networks. In: International Conference on Machine Learning. pp. 5276–5285 (Jul 2018), http://proceedings.mlr.press/v80/weng18a.html
58. Weng*, T.W., Zhang*, H., Chen, P.Y., Yi, J., Su, D., Gao, Y., Hsieh, C.J., Daniel, L.: Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach. In: International Conference on Learning Representations (Feb 2018), https://openreview.net/forum?id=BkUHlMZ0b
59. Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial Examples: Attacks and Defenses for Deep Learning. arXiv:1712.07107 [cs, stat] (Jul 2018), http://arxiv.org/abs/1712.07107
60. Zeiler, M.D., Krishnan, D., Taylor, G.W., Fergus, R.: Deconvolutional networks. In: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. pp. 2528–2535 (Jun 2010)
61. Zhang, H., Zhang, P., Hsieh, C.J.: RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications. Proceedings of the AAAI Conference on Artificial Intelligence **33**(01), 5757–5764 (Jul 2019)
62. Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine Learning Testing: Survey, Landscapes and Horizons. IEEE Transactions on Software Engineering pp. 1–1 (2020)

A Proof of Lemma 3

Lemma 3. *(Equivalence of semantics)*
$\forall p \in s^-, \sigma \in \Sigma.\ [\![p]\!](\sigma) = \delta_{\widetilde{[\![p]\!](\sigma)}}$

Proof. We prove this by induction on the structure of statements in s^-.
We first consider the base cases:

(i) $\underline{\textbf{skip}}$
By definition, for any state σ,
$[\![\textbf{skip}]\!](\sigma) = \delta_\sigma = \delta_{\widetilde{[\![\textbf{skip}]\!](\sigma)}}$

(ii) $\underline{y \leftarrow w \cdot x + \beta}$
Again, by definition, for any state σ,
$[\![y \leftarrow w \cdot x + \beta]\!](\sigma) = \delta_{\sigma[y \mapsto [\![w \cdot x + \beta]\!](\sigma)]} = \delta_{\widetilde{[\![y \leftarrow w \cdot x + \beta]\!](\sigma)}}$

Next, we consider the inductive cases:

(iii) $\underline{s_1^-; s_2^-}$

$$\begin{aligned}
[\![s_1^-; s_2^-]\!](\sigma) &= \mathbb{E}_{\tilde{\sigma} \sim [\![s_1^-]\!](\sigma)}[[\![s_2^-]\!]] \\
&= \lambda\nu. \textstyle\int_{\tilde{\sigma} \in \Sigma} [\![s_1^-]\!](\sigma)(\tilde{\sigma}) \cdot [\![s_2^-]\!](\tilde{\sigma})(\nu) \\
&= \lambda\nu. \textstyle\int_{\tilde{\sigma} \in \Sigma} \delta_{\widetilde{[\![s_1^-]\!](\sigma)}}(\tilde{\sigma}) \cdot \delta_{\widetilde{[\![s_2^-]\!](\tilde{\sigma})}}(\nu) \quad \text{(using inductive hypothesis)} \\
&= \lambda\nu. \delta_{\widetilde{[\![s_2^-]\!](\widetilde{[\![s_1^-]\!](\sigma)})}}(\nu) \\
&= \delta_{\widetilde{[\![s_2^-]\!](\widetilde{[\![s_1^-]\!](\sigma)})}} \\
&= \delta_{\widetilde{[\![s_1^-; s_2^-]\!](\sigma)}}
\end{aligned}$$

(iv) $\underline{\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-}$

$$\begin{aligned}
[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!](\sigma) &= \textbf{if } ([\![b]\!](\sigma)) \textbf{ then } [\![s_1^-]\!](\sigma) \textbf{ else } [\![s_2^-]\!](\sigma) \\
&= \textbf{if } ([\![b]\!](\sigma)) \textbf{ then } \delta_{\widetilde{[\![s_1^-]\!](\sigma)}} \textbf{ else } \delta_{\widetilde{[\![s_2^-]\!](\sigma)}} \\
&\qquad \text{(using inductive hypothesis)} \\
&= \delta_{\textbf{if } ([\![b]\!](\sigma)) \textbf{ then } \widetilde{[\![s_1^-]\!](\sigma)} \textbf{ else } \widetilde{[\![s_2^-]\!](\sigma)}} \\
&= \delta_{\widetilde{[\![\textbf{if } ([\![b]\!](\sigma)) \textbf{ then } s_1^- \textbf{ else } s_2^-]\!](\sigma)}}
\end{aligned}$$

■

B Proof of Corollary 4

Corollary 4. $\forall p \in s^-, \sigma \in \Sigma, \mu \in P(\Sigma).\ \widehat{[\![p]\!]}(\mu)(\widetilde{[\![p]\!]}(\sigma)) \geqslant \mu(\sigma)$

Proof. By definition,

$$\begin{aligned}
\widehat{[\![p]\!]}(\mu) &= \mathbb{E}_{\sigma \sim \mu}[[\![p]\!]] \\
&= \lambda\nu. \textstyle\int_{\sigma \in \Sigma} \mu(\sigma) \cdot [\![p]\!](\sigma)(\nu) \\
&= \lambda\nu. \textstyle\int_{\sigma \in \Sigma} \mu(\sigma) \cdot \delta_{\widetilde{[\![p]\!](\sigma)}}(\nu) \quad \text{(using previous lemma)}
\end{aligned}$$

Now suppose, $\nu = \widetilde{[\![p]\!]}(\tilde{\sigma})$. Then, continuing from above,

$$\begin{aligned}\widehat{[\![p]\!]}(\mu)(\widetilde{[\![p]\!]}(\tilde{\sigma})) &= \textstyle\int_{\sigma\in\Sigma} \mu(\sigma)\cdot\delta_{\widetilde{[\![p]\!]}(\sigma)}(\widetilde{[\![p]\!]}(\tilde{\sigma})) \\ &\geqslant \mu(\tilde{\sigma})\end{aligned}$$

■

C Proof of Theorem 6

We first prove a lemma needed for the proof.

Lemma 12. *(Soundness of abstract conditional checks)*
$\forall c \in b, \sigma^L \in \Sigma^L.\ \{\widetilde{[\![c]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![c]\!]_L(\sigma^L))$ *where*
$\gamma_C(\mathbf{tt}) = \{\mathbf{tt}\}$, $\gamma_C(\mathbf{ff}) = \{\mathbf{ff}\}$, $\gamma_C(\top) = \{\mathbf{tt}, \mathbf{ff}\}$

Proof. We prove this by induction on the structure of the boolean expressions in b.
We first consider the base cases:

(i) $\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)$
By definition, $[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_L(\sigma^L) = [\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_B(\sigma_1^L)$

Consider the case where, $[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_B(\sigma_1^L) = \mathbf{tt}$, then, by the semantics described in Figure 6, we know that,

$$\sigma_1^L(x)_m)_1 \geqslant (\sigma_1^L(y)_n)_2) \tag{1}$$

By the definition of γ_L (Definition 5), we also know that,

$$\forall \sigma^D \in \gamma_L.\ (\sigma_1^L(x)_1 \leqslant \sigma_1^D(x) \leqslant \sigma_1^L(x)_2) \wedge (\sigma_1^L(y)_1 \leqslant \sigma_1^D(y) \leqslant \sigma_1^L(y)_2) \tag{2}$$

where the comparisons are performed pointwise for every element in the vector.
From 1 and 2, we can conclude that,

$$\forall \sigma^D \in \gamma_L(\sigma^L).\ \sigma_1^D(y)_n \leqslant (\sigma_1^L(y)_n)_2 \leqslant (\sigma_1^L(x)_m)_1 \leqslant \sigma_1^D(x)_m \tag{3}$$

Now,

$$\begin{aligned}[\![\boldsymbol{\pi}(x,m) \widetilde{\geqslant} \boldsymbol{\pi}(y,n)]\!]_D(\sigma^D) = [\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!](\sigma_1^D) = \\ \textbf{if } \sigma_1^D(x)_m \geqslant \sigma_1^D(y)_n \textbf{ then tt else ff}\end{aligned} \tag{4}$$

From 3 and 4, we can conclude that,
$\forall \sigma^D \in \gamma_L(\sigma^L).\ [\![\boldsymbol{\pi}(x,m) \widetilde{\geqslant} \boldsymbol{\pi}(y,n)]\!]_D(\sigma^D) = \mathbf{tt}$, or in other words,
$\{[\![\boldsymbol{\pi}(x,m) \widetilde{\geqslant} \boldsymbol{\pi}(y,n)]\!]_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_L(\sigma^L))$
when the analysis returns $\mathbf{tt}$.
We can similarly prove the case when the analysis returns $\mathbf{ff}$. In case, the analysis returns $\top$, the required subset containment is trivially true since $\gamma_C(\top) = \{\mathbf{tt}, \mathbf{ff}\}$.

(ii) $\underline{\boldsymbol{\pi}(x, m) \geqslant 0}$
The proof is very similar to the first case, and we skip the details.

(iii) $\underline{\boldsymbol{\pi}(x, m) < 0}$
The proof is very similar to the first case, and we skip the details.

We next consider the inductive cases:

(iv) $\underline{b_1 \wedge b_2}$
By the inductive hypothesis, we know that,
$\{\widetilde{[\![b_1]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![b_1]\!]_L(\sigma^L))$
$\{\widetilde{[\![b_2]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![b_2]\!]_L(\sigma^L))$
If $[\![b_1]\!]_L(\sigma^L) = \top \vee [\![b_2]\!]_L(\sigma^L) = \top$, then, as per the semantics in Figure 6, $[\![b_1 \wedge b_2]\!]_L(\sigma^L) = \top$, and the desired property trivially holds.
However, if $[\![b_1]\!]_L(\sigma^L) \neq \top \wedge [\![b_2]\!]_L(\sigma^L) \neq \top$, then using the inductive hypotheses, we know that for all $\sigma^D \in \gamma_L(\sigma^L)$, $\widetilde{[\![b_1]\!]}_D(\sigma^D)$ evaluates to the same boolean value as $[\![b_1]\!]_L(\sigma^L)$. We can make the same deduction for b_2. So, evaluating $\widetilde{[\![b_1 \wedge b_2]\!]}_D$ also yields the same boolean value for all $\sigma^D \in \gamma_L(\sigma^L)$, and this value is equal to $[\![b_1 \wedge b_2]\!]_L(\sigma^L)$.

(v) $\underline{\neg b}$
By the inductive hypothesis, we know that,
$\{\widetilde{[\![b]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![b]\!]_L(\sigma^L))$
If $[\![b]\!]_L(\sigma^L) = \mathbf{tt}$, then $\forall \sigma^D \in \gamma_L(\sigma^L).\ \widetilde{[\![b]\!]}_D(\sigma^D) = \mathbf{tt}$.
So, $\forall \sigma^D \in \gamma_L(\sigma^L).\ \widetilde{[\![\neg b]\!]}_D(\sigma^D) = \mathbf{ff}$, and we can conclude that,
$\{\widetilde{[\![\neg b]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![\neg b]\!]_L(\sigma^L)) = \{\mathbf{ff}\}$.
We can similarly argue about the case when $[\![b]\!]_L(\sigma^L) = \mathbf{ff}$, and as stated previously, the case with, $[\![b]\!]_L(\sigma^L) = \top$ trivially holds.

■

Theorem 6. *(Soundness of Jacobian analysis)*
$\forall p \in s^-, \sigma^L \in \Sigma^L.\ \{\widetilde{[\![p]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![p]\!]_L(\sigma^L))$

Proof. We prove this by induction on the structure of statements in s^-.
We first consider the base cases:

(i) $\underline{\mathbf{skip}}$
By definition, for any state σ^L,

$$[\![\mathbf{skip}]\!]_L(\sigma^L) = \sigma^L \tag{5}$$

$$\{\widetilde{[\![\mathbf{skip}]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} = \{\sigma^D \mid \sigma^D \in \gamma_L(\sigma^L)\} = \gamma_L(\sigma^L) \tag{6}$$

From Equations 5 and 6,

$$\{\widetilde{[\![\mathbf{skip}]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\mathbf{skip}]\!]_L(\sigma^L)) \tag{7}$$

(ii) $\underline{y \leftarrow w \cdot x + \beta}$
We first observe that when multiplying an interval (l, u) with a constant c, if $c \geqslant 0$, then the result is simply given by the interval $(c \cdot l, c \cdot u)$. But if $c < 0$, then the result is in the interval $(c \cdot u, c \cdot l)$, i.e., the use of the lower bounds and upper bounds gets flipped. Similarly, when computing the dot product of an abstract vector v with a constant vector w, for each multiplication operation $v_i \cdot w_i$, we use the same reasoning as above. Then, the lower bound and upper bound of the dot product result are given by,
$(\sum_{i=1 \wedge w_i \geqslant 0}^{n} w_i \cdot (v_i)_1 + \sum_{i=1 \wedge w_i < 0}^{n} w_i \cdot (v_i)_2, \sum_{i=1 \wedge w_i \geqslant 0}^{n} w_i \cdot (v_i)_2 + \sum_{i=1 \wedge w_i < 0}^{n} w_i \cdot (v_i)_1)$
where $(v_i)_1$ represents the lower bound of the i^{th} element of v and $(v_i)_2$ represents the lower bound of the i^{th} element of v, and we assume $\mathbf{dim}(w) = \mathbf{dim}(v) = n$.
We do not provide the rest of the formal proof for this case since it just involves using the definitions.

Next, we consider the inductive cases:

(iii) $\underline{s_1^-; s_2^-}$
From the inductive hypothesis, we know,

$$L_1 = \{\widetilde{[\![s_1^-]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma^L)) \tag{8}$$

$$L_2 = \{\widetilde{[\![s_2^-]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L([\![s_1^-]\!]_L(\sigma^L))\} \subseteq \gamma_L([\![s_2^-]\!]_L([\![s_1^-]\!]_L(\sigma^L))) \tag{9}$$

From Equations 8 and 9, we conclude,

$$\{\widetilde{[\![s_2^-]\!]}_D(\sigma^D) \mid \sigma^D \in L_1\} \subseteq L_2 \subseteq \gamma_L([\![s_2^-]\!]_L([\![s_1^-]\!]_L(\sigma^L))) \tag{10}$$

Rewriting, we get,

$$\{\widetilde{[\![s_2^-]\!]}_D(\widetilde{[\![s_1^-]\!]}_D(\sigma^D)) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_2^-]\!]_L([\![s_1^-]\!]_L(\sigma^L))) \tag{11}$$

and this can be simplified further as,

$$\{\widetilde{[\![s_1^-; s_2^-]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_1^-; s_2^-]\!]_L(\sigma^L)) \tag{12}$$

(iv) $\underline{\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-}$
From the inductive hypothesis, we know,

$$\{\widetilde{[\![s_1^-]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma^L)) \tag{13}$$

$$\{\widetilde{[\![s_2^-]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_2^-]\!]_L(\sigma^L)) \tag{14}$$

The conditional check can result in three different outcomes while performing the analysis - **tt**, **ff**, or $\top$. From Lemma 12, we know that the abstract boolean checks are sound. We analyze each of the three cases separately.

(a) **tt**
Since we only consider the true case, we can write,

$$[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L) = [\![s_1^-]\!]_L(\sigma^L) \tag{15}$$

Also, from Lemma 12,

$$\{\widetilde{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_D}(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} = \{\widetilde{[\![s_1^-]\!]_D}(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \tag{16}$$

From 13, 15, and 16,

$$\{\widetilde{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_D}(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L)) \tag{17}$$

(b) **ff**
Similar to the **tt** case, for the **ff** case, we can show,

$$\{\widetilde{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_D}(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L)) \tag{18}$$

(c) $\top$
We first prove the following about the join ($\bigsqcup_L$) operation,

$$\gamma_L(\sigma^L) \cup \gamma_L(\tilde{\sigma}^L) \subseteq \gamma_L(\sigma^L \sqcup_L \tilde{\sigma}^L) \tag{19}$$

By definition of γ_L,

$$\begin{aligned}\gamma_L(\sigma^L) = \{\sigma^D \mid (\bigwedge_{v \in V} .\sigma_1^L(v)_1 \leqslant \sigma_1^D(v) \leqslant \sigma_1^L(v)_2) \wedge \\ (\bigwedge_{v \in V} .(\sigma_2^L(v)_1)_1 \leqslant \sigma_2^D(v)_1 \leqslant (\sigma_2^L(v)_1)_2) \wedge \\ \sigma_2^D(v)_2 \in \gamma_V(\sigma_2^L(v)_2)\}\end{aligned} \tag{20}$$

$\gamma_L(\tilde{\sigma}^L)$ can be defined similarly.
The join operation combines corresponding intervals in the abstract states by taking the smaller of the two lower bounds and larger of the two upper bounds. We do not prove the following formally, but from the definition of γ_L and $\bigsqcup_L$, one can see that the intended property holds.
Next, we consider the **assert** statements that appear in the abstract denotational semantics for the $\top$ case.
Let us call, $\sigma_1^L = [\![\textbf{assert } b]\!]_L(\sigma^L)$ and $\sigma_2^L = [\![\textbf{assert } \neg b]\!]_L(\sigma^L)$.
From inductive hypothesis (13 and 14) we know,

$$L_1 = \{\widetilde{[\![s_1^-]\!]_D}(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma_1^L)\} \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma_1^L)) \tag{21}$$

$$L_2 = \{\widetilde{[\![s_2^-]\!]_D}(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma_2^L)\} \subseteq \gamma_L([\![s_2^-]\!]_L(\sigma_2^L)) \tag{22}$$

From 19,21, and 22,

$$L_1 \cup L_2 \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma_1^L)) \cup \gamma_L([\![s_2^-]\!]_L(\sigma_2^L)) \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma_1^L) \sqcup [\![s_2^-]\!]_L(\sigma_2^L)) \tag{23}$$

Then, if we can show that,

$$\{\sigma^D \mid \sigma^D \in \gamma_L(\sigma^L) \wedge [\![b]\!](\sigma^D) = \mathbf{tt}\} \subseteq \gamma_L(\sigma_1^L) \tag{24}$$

$$\{\sigma^D \mid \sigma^D \in \gamma_L(\sigma^L) \wedge [\![b]\!](\sigma^D) = \mathbf{ff}\} \subseteq \gamma_L(\sigma_2^L) \tag{25}$$

then, from 21, 22,23,24,25, and the semantics of **if** b **then** s_1^- **else** s_2^-, we can say,

$$\{\widetilde{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L)) \tag{26}$$

Now, we need to show that 24 and 25 are true. The **assert** statements either behave as identity or produce a modified abstract state (see Figure 6). When **assert** behaves as identity, 24 and 25 are obviously true. We skip the proof of the case when **assert** produces a modified abstract state.

■

D Proof of Corollary 8

Corollary 8. *(Upper bound of Jacobian operator norm)*
$\forall p \in s^-, \sigma^L \in \Sigma^L, v \in V.$
$\mathbf{max}\{\left\|((\widetilde{[\![p]\!]}_D(\sigma^D))_2)(v)_1\right\| \mid \sigma^D \in \gamma_L(\sigma^L)\} \leqslant \|((([\![p]\!]_L(\sigma^L))_2(v))_1\|_L$

Proof. From Theorem 6, we know that for any $p \in s^-, \sigma^L \in \Sigma^L$,

$$\{\widetilde{[\![p]\!]}_D(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![p]\!]_L(\sigma^L)) \tag{1}$$

Let us define, $D_V = \{((\widetilde{[\![p]\!]}_D(\sigma^D))_2(v))_1 \mid \sigma^D \in \gamma_L(\sigma^L)\}$. This is the set of all Jacobian matrices associated with the variable v after executing p on the set of input states, $\gamma_L(\sigma^L)$. Note that the set D_V does not distinguish the Jacobians on the basis of the input that we are differentiating with respect to.
Let $D_V^L = \{(\tilde{\sigma}_2^D(v))_1 \mid \tilde{\sigma}^D \in \gamma_L([\![p]\!]_L(\sigma^L))\}$, and $J = (([\![p]\!]_L(\sigma^L))_2(v))_1$.
Using Definition 5 of γ_L, we can show,

$$\forall d \in D_V^L.\ J_1 \leqslant d \leqslant J_2 \tag{2}$$

where $\leqslant$ is defined pointwise on the matrices, and $J_1(J_2)$ refers to the matrix of lower(upper) bounds.
Then, from 1 and definitions of D_V and D_V^L, we can deduce that,

$$D_V \subseteq D_V^L \tag{3}$$

From 2 and 3,

$$\forall d \in D_V.\ J_1 \leqslant d \leqslant J_2 \tag{4}$$

Let $J' = [\mathbf{max}\{|(J_{k,l})_1|, |(J_{k,l})_2|\} \mid k \in \{1, ..., m\}, l \in \{1, ..., n\}]$. Then,

$$\forall d \in D_V.\ |d| \leqslant J' \tag{5}$$

where $|\cdot|$ applies pointwise on matrices d.
Using definition of operator norm, one can show that,

$$M_1 \leqslant M_2 \implies \|M_1\| \leqslant \|M_2\| \tag{6}$$

where M_1 and M_2 are matrices with $\leqslant$ applied pointwise.
Finally, from 5 and 6, we conclude,

$$\forall d \in D_V.\ \|d\| \leqslant \left\|J'\right\| = \|J\| \tag{7}$$

Unrolling the definitions,

$$\mathbf{max}\{\left\|((\widetilde{[\![p]\!]}_D(\sigma^D))_2)(v)_1\right\| \mid \sigma^D \in \gamma(\sigma^L)\} \leqslant \|((([\![p]\!]_L(\sigma^L))_2(v))_1\|_L \tag{8}$$

■

E Proof of Theorem 11

Theorem 11. *(Soundness of* PROLIP*)*
Let $p = z \leftsquigarrow N(0,1); g; f$ *where* $g, f \in s^-$, $(k_U, d, vol) = \texttt{PROLIP}(p, z_B)$, $z \notin \mathbf{outv}(g), z \notin \mathbf{outv}(f), x \in \mathbf{inv}(f)$, *and* $y \in \mathbf{outv}(f)$ *then,* $\forall \sigma_0 \in \Sigma$.

$$\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} ((\|\sigma(y) - \sigma'(y)\| \leqslant k_U \cdot \|\sigma(x) - \sigma'(x)\|) \wedge (\sigma(z), \sigma'(z) \in \gamma(z_B))) \geqslant vol$$

Proof. We prove this theorem in two parts.

First, let us define set Σ_P as, $\Sigma_P = \{\sigma \mid \sigma \in \gamma_B([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1)\}$
In words, Σ_P is the concretization of the abstract box produced by abstractly "interpreting" $g; f$ on the input box z_B. Assuming that z is not written to by g or f, it is easy to see from the definitions of the abstract semantics in Figures 6 and 4 that, $([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1(z) = z_B$, i.e., the final abstract value of z is the same as the initial value z_B. Moreover, from Theorem 8, we know that the operator norm of the abstract Jacobian matrix, $\|J\|_L$ upper bounds the operator norm of every Jacobian of f for variable y with respect to x (since $x \in \mathbf{inv}(f), y \in \mathbf{outv}(f)$) for every input in $\gamma_B([\![g]\!]_B(\sigma^B[z \mapsto z_B]))$, which itself is an upper bound on the local Lipschitz constant in the same region.
In other words, we can say that,
$\forall \sigma, \sigma' \in \Sigma_P.\ \sigma(z), \sigma'(z) \in \gamma(z_B) \wedge \|\sigma(y) - \sigma'(y)\| \leqslant k_U \cdot \|\sigma(x) - \sigma'(x)\|$.

To complete the proof, we need to show that, $\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} (\sigma, \sigma' \in \Sigma_P) \geqslant vol$. We show this in the second part of this proof.
Using the semantic definition of *pcat* (Figure 2), we know that,
$[\![p]\!](\sigma_0) = \widehat{[\![f]\!]}(\widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0)))$

We first analyze $[\![z \leftsquigarrow N(0,1)]\!](\sigma_0)$. Again using the semantic definition of *pcat*, we write,

$$\begin{aligned}[\![z \leftsquigarrow N(0,1)]\!](\sigma_0) &= \mathbb{E}_{z\sim N(0,1)}[\lambda\nu.\delta_{\sigma_0[z\mapsto\nu]}] \\ &= \lambda\nu'.\textstyle\int_a N(a)\cdot\delta_{\sigma_0[z\mapsto a]}(\nu') \\ &= \lambda\nu'.1_{\nu'=\sigma_0[z\mapsto a]}\cdot N(a)\end{aligned} \tag{1}$$

We are interested in the volume of the set Σ_z, defined as, $\Sigma_z = \{\sigma \mid \sigma(z) \in z_B\}$. Using the expression for $[\![z \leftsquigarrow N(0,1)]\!](\sigma_0)$ from above, we can now compute the required probability as follows,

$$\begin{aligned}\Pr_{\sigma\sim[\![z\leftsquigarrow N(0,1)]\!](\sigma_0)}(\sigma\in\Sigma_z) &= \textstyle\int_{\sigma\in\Sigma}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma)\cdot 1_{\sigma\in\Sigma_z} \\ &= \textstyle\int_{\sigma\in\Sigma}(1_{\sigma=\sigma_0[z\mapsto a]}\cdot N(a))\cdot 1_{\sigma\in\Sigma_z} \\ &= \textstyle\int_{\sigma\in\Sigma_z}(1_{\sigma=\sigma_0[z\mapsto a]}\cdot N(a)) \\ &= \textstyle\int_{a\in z_B} N(a) \quad \text{(by uniqueness of } \sigma_0[z\mapsto a]) \\ &= vol'\end{aligned} \tag{2}$$

This shows that starting from any $\sigma_0 \in \Sigma$, after executing the first statement of p, the probability that the value stored at z lies in the box z_B is vol'.
Next, we analyze $[\![z \leftsquigarrow N(0,1)]\!](\sigma_0)$. In particular, we are interested in the volume of the set, $\widetilde{[\![g]\!]}(\Sigma_z)$ (which is notational abuse for the set $\{\widetilde{[\![g]\!]}(\sigma) \mid \sigma \in \Sigma_z\}$). We can lower bound this volume as follows,

$$\begin{aligned}\Pr_{\sigma\sim\widehat{[\![g]\!]}([\![z\leftsquigarrow N(0,1)]\!](\sigma_0))}(\sigma\in\widetilde{[\![g]\!]}(\Sigma_z)) &= \textstyle\int_{\sigma\in\Sigma}(\widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma)\cdot 1_{\sigma\in\widetilde{[\![g]\!]}(\Sigma_z)} \\ &= \textstyle\int_{\sigma\in\widetilde{[\![g]\!]}(\Sigma_z)}\widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma) \\ &\geqslant \textstyle\int_{\sigma\in\Sigma_z}[\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma) \text{ (from Corollary 4)} \\ &= vol' \quad \text{(from 2)}\end{aligned} \tag{3}$$

We can similarly show that,

$$\Pr_{\sigma\sim\widehat{[\![f]\!]}(\widehat{[\![g]\!]}([\![z\leftsquigarrow N(0,1)]\!](\sigma_0)))}(\sigma\in\widetilde{[\![f]\!]}(\widetilde{[\![g]\!]}(\Sigma_z))) \geqslant vol' \tag{4}$$

Now, $\sigma^B[z\mapsto z_B]$ defined on line 2 of Algorithm 1 is such that $\gamma(\sigma^B[z\mapsto z_B]) = \Sigma_z$. From Theorem 10, we can conclude that,

$$\widetilde{[\![g]\!]}(\Sigma_z) \subseteq \gamma([\![g]\!]_B(\sigma^B[z\mapsto z_B])) \tag{5}$$

Similarly, from Theorem 6, we can conclude that,

$$\widetilde{[\![f]\!]}(\widetilde{[\![g]\!]}(\Sigma_z)) \subseteq \gamma([\![f]\!]_L([\![g]\!]_B(\sigma^B[z\mapsto z_B]))_1) \tag{6}$$

From 4 and 6, we conclude that,

$$\Pr_{\sigma\sim[\![p]\!](\sigma_0)}(\sigma\in\gamma([\![f]\!]_L([\![g]\!]_B(\sigma^B[z\mapsto z_B])))_1) \geqslant vol' \tag{7}$$

Consequently,

$$\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} (\sigma, \sigma' \in \gamma([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1) \geqslant vol' \times vol' = vol \quad (8)$$

since each act of sampling is independent. ■

F Translating Neural Networks Into *pcat*

`NNs` are often described as a sequential composition of "layers", with each layer describing the computation to be performed on an incoming vector. Many commonly used layers can be expressed in the *pcat* language. For instance, [28] describes the translation of maxpool, convolution, ReLU, and fully connected layers into the *cat* language. Here, we describe the translation of two other common layers, namely, the batchnorm layer [34] and the transposed convolution layer (also referred to as the deconvolution layer) [60].

Batchnorm layer. A batchnorm layer typically typically expects an input $x \in \mathbb{R}^{C \times H \times W}$ which we flatten, using a row-major form in to $x' \in \mathbb{R}^{C \cdot H \cdot W}$ where, historically, C denotes the number of channels in the input, H denotes the height, and W denotes the width. For instance, given an RGB image of dimensions 28 $\times$ 28 pixels, $H = 28$, $W = 28$, and $C = 3$.

A batchnorm layer is associated with vectors m and v such that $\mathbf{dim}(m) = \mathbf{dim}(v) = C$ where $\mathbf{dim}(\cdot)$ returns the dimension of a vector. m and v represent the running-mean and running-variance of the values in each channel observed during the training time of the `NN`. A batchnorm layer is also associated with a scaling vector s^1 and a shift vector s^2, both also of dimension c. For a particular element $x_{i,j,k}$ in the input, the corresponding output element is $s^1_i \cdot (\frac{x_{i,j,k} - m_i}{\sqrt{v_i + \epsilon}}) + s^2_i$ where ϵ is a constant that is added for numerical stability (commonly set to $1e^{-5}$). Note that the batchnorm operation produces an output of the same dimensions as the input. We can represent the batchnorm operation by the statement, $y \leftarrow w \cdot x' + \beta$, where x' is the flattened input, w is a weight matrix of dimension $C \cdot H \cdot W \times C \cdot H \cdot W$ and β is a bias vector of dimension $C \cdot H \cdot W$, such that,

$w = I \cdot [\frac{s^1_{\lfloor i/H \cdot W \rfloor}}{\sqrt{v_{\lfloor i/H \cdot W \rfloor} + \epsilon}} \mid i \in \{1, ..., C \cdot H \cdot W\}]$

$\beta = [-\frac{s^1_{\lfloor i/H \cdot W \rfloor} \cdot m_{\lfloor i/H \cdot W \rfloor}}{\sqrt{v_{\lfloor i/H \cdot W \rfloor} + \epsilon}} + s^2_{\lfloor i/H \cdot W \rfloor} \mid i \in \{1, ..., C \cdot H \cdot W\}]$

where I is the identity matrix with dimension $(C \cdot H \cdot W, C \cdot H \cdot W)$, $\lfloor \cdot \rfloor$ is the floor operation that rounds down to an integer, and $[\ |\ ]$ is the list builder/comprehension notation.

Transposed convolution layer. A convolution layer applies a kernel or a filter on the input vector and typically, compresses this vector so that the output vector is of a smaller dimension. A deconvolution or transposed convolution layer does the opposite - it applies the kernel in a manner that produces a larger output vector. A transposed convolution layer expects an input $x \in \mathbb{R}^{C_{in} \times H_{in} \times W_{in}}$ and applies a kernel $k \in \mathbb{R}^{C_{out} \times C_{in} \times K_h \times K_w}$ using a stride S. For simplicity of

presentation, we assume that $K_h = K_w = K$ and $W_{in} = H_{in}$. In *pcat*, the transposed convolution layer can be expressed by the statement, $y \leftarrow w \cdot x'$, where x' is the flattened version of input x, w is a weight matrix that we derive from the parameters associated with the transposed convolution layer, and the bias vector, β, is a zero vector in this case. To compute the dimensions of the weight matrix, we first calculate the height (H_{out}) and width (W_{out}) of each channel in the output using formulae, $H_{out} = H_{in} \cdot S + K$, and $W_{out} = W_{in} \cdot S + K$. Since we assume $W_{in} = H_{in}$, we have $W_{out} = H_{out}$ here. Then, the dimension of w is $C_{out} \cdot H_{out} \cdot W_{out} \times C_{in} \cdot H_{in} \cdot W_{in}$, and the definition of w is as follows,

$$w = \left[\begin{array}{l} \textbf{let } x = \lceil i/C_{out} \rceil \quad \textbf{in} \\ \textbf{let } y = \lceil j/C_{in} \rceil \quad \textbf{in} \\ \textbf{let } h = 1 + \lfloor ((i \bmod C_{out}) - (\lfloor ((j \bmod C_{in}) - 1)/H_{in} \rfloor \cdot \\ \qquad H_{out} \cdot S + 1 + (((j \bmod C_{in}) - 1) \bmod H_{in}) \cdot S))/H_{out} \rfloor \quad \textbf{in} \\ \textbf{let } w = 1 + ((i \bmod C_{out}) - (\lfloor ((j \bmod C_{in}) - 1)/H_{in} \rfloor \cdot \\ \qquad H_{out} \cdot S + 1 + (((j \bmod C_{in}) - 1) \bmod H_{in}) \cdot S)) \bmod H_{out} \ \textbf{in} \\ \textbf{if } h, w \in \{1...K\} \textbf{ then } k_{x,y,h,w} \textbf{ else } 0 \end{array} \middle| \begin{array}{l} \\ \\ \\ \\ \\ i \in I, \\ j \in J \end{array} \right]$$

where $I = \{1, ..., C_{out} \cdot H_{out} \cdot W_{out}\}$ and $J = \{1, ..., C_{in} \cdot H_{in} \cdot W_{in}\}$

G Details of Box Analysis

$$
\begin{aligned}
\Sigma^B &\triangleq V \to \textstyle\bigcup_{n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^n \\
[\![e]\!]_B &: \Sigma^B \to \textstyle\bigcup_{n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^n \\
[\![\boldsymbol{\pi}(x,n)]\!]_B(\sigma^B) &= \sigma^B(x)_n \\
[\![w\cdot x+\beta]\!]_B(\sigma^B) &= \textbf{let } m = \textbf{dim}(w)_1 \textbf{ in} \\
&\quad \textbf{let } n = \textbf{dim}(\sigma^B(x)) \textbf{ in} \\
&\quad [((\sum_{j=1\wedge w_{i,j}\geqslant 0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_1 + \\
&\quad \sum_{j=1\wedge w_{i,j}<0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_2 + \beta_i), \\
&\quad (\sum_{j=1\wedge w_{i,j}\geqslant 0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_2 + \\
&\quad \sum_{j=1\wedge w_{i,j}<0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_1 + \beta_i)) \mid i\in\{1,...,m\}]
\end{aligned}
$$

$$
\begin{aligned}
[\![b]\!]_B &: \Sigma^B \to \{\textbf{tt},\textbf{ff},\top\} \\
[\![\boldsymbol{\pi}(x,m)\geqslant\boldsymbol{\pi}(y,n)]\!]_B(\sigma^B) &= \textbf{if } ((\sigma^B(x)_m)_1 \geqslant (\sigma^B(y)_n)_2) \textbf{ then tt} \\
&\quad \textbf{else if } ((\sigma^B(x)_m)_2 < (\sigma^B(y)_n)_1) \textbf{ then ff} \\
&\quad \textbf{else } \top \\
[\![\boldsymbol{\pi}(x,m)\geqslant 0]\!]_B(\sigma^B) &= \textbf{if } ((\sigma^B(x)_m)_1 \geqslant 0) \textbf{ then tt} \\
&\quad \textbf{else if } ((\sigma^B(x)_m)_2 < 0) \textbf{ then ff} \\
&\quad \textbf{else } \top \\
[\![\boldsymbol{\pi}(x,m)<0]\!]_B(\sigma^B) &= \textbf{if } ((\sigma^B(x)_m)_2 < 0) \textbf{ then tt} \\
&\quad \textbf{else if } ((\sigma^B(x)_m)_1 \geqslant 0) \textbf{ then ff} \\
&\quad \textbf{else } \top \\
[\![b_1\wedge b_2]\!]_B(\sigma^B) &= \textbf{if } ([\![b_1]\!]_B(\sigma^B)=\top \vee [\![b_2]\!]_B(\sigma^B)=\top) \textbf{ then } \top \\
&\quad \textbf{else } [\![b_1]\!]_B(\sigma^B) \wedge [\![b_2]\!]_B(\sigma^B) \\
[\![\neg b]\!]_B(\sigma^B) &= \textbf{if } ([\![b]\!]_B(\sigma^B)=\textbf{tt}) \textbf{ then ff} \\
&\quad \textbf{else if } ([\![b]\!]_B(\sigma^B)=\textbf{ff}) \textbf{ then tt} \\
&\quad \textbf{else } \top
\end{aligned}
$$

$$
\begin{aligned}
\textstyle\bigsqcup_B &: \Sigma^B\times\Sigma^B\to\Sigma^B \\
\sigma^B \textstyle\bigsqcup_B \tilde{\sigma}^B &= \lambda v.\ [(\textbf{min}\{(\sigma^B(v)_i)_1, (\tilde{\sigma}^B(v)_i)_1\}, \textbf{max}\{(\sigma^B(v)_i)_2, (\tilde{\sigma}^B(v)_i)_2\}) \mid \\
&\qquad i\in\{1,...,\textbf{dim}(\sigma^B(v))\}]
\end{aligned}
$$

$$
\begin{aligned}
[\![s^-]\!]_B &: \Sigma^B\to\Sigma^B \\
[\![\textbf{skip}]\!]_B(\sigma^B) &= \sigma^B \\
[\![\textbf{assert } \boldsymbol{\pi}(x,m)\geqslant 0]\!]_B(\sigma^B) &= \sigma^B[x_m \mapsto (0, \textbf{max}\{(\sigma^B(x)_m)_2, 0\})] \\
[\![\textbf{assert } \boldsymbol{\pi}(x,m)< 0]\!]_B(\sigma^B) &= \sigma^B[x_m \mapsto (\textbf{min}\{(\sigma^B(x)_m)_1, 0\}, 0)] \\
[\![\textbf{assert } \neg(\boldsymbol{\pi}(x,m)\geqslant 0)]\!]_B(\sigma^B) &= [\![\textbf{assert } \boldsymbol{\pi}(x,m)< 0]\!]_B(\sigma^B) \\
[\![\textbf{assert } \neg(\boldsymbol{\pi}(x,m)< 0)]\!]_B(\sigma^B) &= [\![\textbf{assert } \boldsymbol{\pi}(x,m)\geqslant 0]\!]_B(\sigma^B) \\
[\![\textbf{assert } \hat{b}]\!]_B(\sigma^B) &= \sigma^B \text{ (where } \hat{b} \text{ refers to all other boolean expressions)} \\
[\![y\leftarrow w\cdot x+\beta]\!]_B(\sigma^B) &= \sigma^B[y\mapsto [\![w\cdot x+\beta]\!]_B(\sigma^B)] \\
[\![s_1;s_2]\!]_B(\sigma^B) &= [\![s_2]\!]_B([\![s_1]\!]_B(\sigma^B)) \\
[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]_B(\sigma^B) &= \textbf{if } ([\![b]\!]_B(\sigma^B)=\textbf{tt}) \textbf{ then } [\![s_1]\!]_B(\sigma^B) \\
&\quad \textbf{else if } ([\![b]\!]_B(\sigma^B)=\textbf{ff}) \textbf{ then } [\![s_2]\!]_B(\sigma^B) \\
&\quad \textbf{else } [\![s_1]\!]_B([\![\textbf{assert } b]\!]_B(\sigma^B)) \textstyle\bigsqcup_B [\![s_2]\!]_B([\![\textbf{assert } \neg b]\!]_B(\sigma^B))
\end{aligned}
$$

Fig. 6: *cat* abstract semantics for box analysis