CORRECTNESS VERIFICATION OF NEURAL NETWORK

Anonymous authors
Paper under double-blind review

ABSTRACT

We present the first verification that a neural network for perception tasks produces a *correct* output within a specified tolerance for *every* input of interest. We define correctness relative to a *specification* which identifies 1) a *state space* consisting of all relevant states of the world and 2) an *observation process* that produces neural network inputs from the states of the world. Tiling the state and input spaces with a finite number of tiles, obtaining ground truth bounds from the state tiles and network output bounds from the input tiles, then comparing the ground truth and network output bounds delivers an upper bound on the network output error for any input of interest. Results from two case studies highlight the ability of our technique to deliver tight error bounds for all inputs of interest and show how the error bounds vary over the state and input spaces.

1 INTRODUCTION

Neural networks are now recognized as powerful function approximators with impressive performance across a wide range of applications, especially perception tasks (e.g. vision, speech recognition). Current techniques, however, provide no *correctness guarantees* on such neural perception systems — there is currently no way to verify that a neural network provides *correct* outputs (within a specified tolerance) for *all* inputs of interest. The closest the field has come is robustness verification, which aims to verify if the network prediction is *stable* for all inputs in some neighborhood around a selected input point. But robustness verification does not verify for *all* inputs of interest — it only verifies around local regions. Besides, it does not guarantee that the output, even if stable, is actually *correct* — there is no specification that defines the correct output for any input except for the manually-labeled center point of each region.

We present the first *correctness verification* of neural networks for perception — the first verification that a neural network produces a *correct* output within a specified tolerance for *every* input of interest. Neural networks are often used to predict some property of the world given an observation such as an image or audio recording. We therefore define correctness relative to a *specification* which identifies 1) a *state space* consisting of all relevant states of the world and 2) an *observation process* that produces neural network inputs from the states of the world. Then the inputs of interest are all inputs that can be observed from the state space via the observation process. We define the set of inputs of interest as the *feasible input space*. Because the quantity of interest that the network predicts is some property of the state of the world, the state defines the ground truth output (and therefore defines the correct output for each input to the neural network).

We present *Tiler*, the algorithm for correctness verification of neural networks. Evaluating the correctness of the network on a single state is straightforward — use the observation process to obtain the possible inputs for that state, use the neural network to obtain the possible outputs, then compare the outputs to the ground truth from the state. To do correctness verification, we generalize this idea to work with *tiled* state and input spaces. We cover the state and input spaces with a finite number of *tiles*: each state tile comprises a set of states; each input tile is the image of the corresponding state tile under the observation process. The state tiles provide ground truth bounds for the corresponding input tiles. We use recently developed techniques from the robustness verification literature to obtain network output bounds for each input tile (Xiang et al., 2018; Gehr et al., 2018; Weng et al., 2018; Bastani et al., 2016; Lomuscio and Maganti, 2017; Tjeng et al., 2019). A comparison of the ground truth and output bounds delivers an error upper bound for that region of the state space. The error bounds for all the tiles jointly provide the correctness verification result.

We present two case studies. The first involves a world with a (idealized) fixed road and a camera that can vary its horizontal offset and viewing angle with respect to the centerline of the road (Section 5). The state of the world is therefore characterized by the offset δ and the viewing angle θ. A neural network takes the camera image as input and predicts the offset and the viewing angle. The state space includes the δ and θ of interest. The observation process is the camera imaging process, which maps camera positions to images. This state space and the camera imaging process provide the specification. The feasible input space is the set of camera images that can be observed from all camera positions of interest. For each image, the camera positions of all the states that can produce the image give the possible ground truths. We tile the state space using a grid on (δ, θ). Each state tile gives a bound on the ground truth of δ and θ. We then apply the observation process to project each state tile into the image space. We compute a bounding box for each input tile and apply techniques from robustness verification (Tjeng et al., 2019) to obtain neural network output bounds for each input tile. Comparing the ground truth bounds and the network output bounds gives upper bounds on network prediction error for each tile. We verify that our trained neural network provides good accuracy across the majority of the state space of interest and bound the *maximum* error the network will ever produce on any feasible input.

The second case study verifies a neural network that classifies a LiDAR measurement of a sign in an (idealized) scene into one of three shapes (Section 6). The state space includes the position of the LiDAR sensor and the shape of the sign. We tile the state space, project each tile into the input space via the LiDAR observation process, and again apply techniques from robustness verification to verify the network, including identifying regions of the input space where the network may deliver an incorrect classification.

This paper makes the following contributions:

Specification: We show how to use state spaces and observation processes to specify the global correctness of neural networks for perception (the space of all inputs of interest, and the correct output for each input). This is the first systematic approach (to our knowledge) to give global correctness specification for perception neural networks.

Verification: We present an algorithm, *Tiler*, for correctness verification. With state spaces and observation processes providing specification, this is the first algorithm (to our knowledge) for verifying that a neural network produces the *correct* output (up to a specified tolerance) for *every* input of interest. The algorithm can also compute tighter correctness bounds for focused regions of the state and input spaces.

Case Study: We apply this framework to a problem of predicting camera position from image and a problem of classifying shape of the sign from LiDAR measurement. We obtain the first correctness verification of neural networks for perception tasks.

2 Related Work

Motivated by the vulnerability of neural networks to adversarial attacks (Papernot et al., 2016; Szegedy et al., 2014), researchers have developed a range of techniques for verifying robustness — they aim to verify if the neural network prediction is stable in some neighborhood around a selected input point. Salman et al. (2019); Liu et al. (2019) provide an overview of the field. A range of approaches have been explored, including layer-by-layer reachability analysis (Xiang et al., 2017; 2018) with abstract interpretation (Gehr et al., 2018) or bounding the local Lipschitz constant (Weng et al., 2018), formulating the network as constraints and solving the resulting optimization problem (Bastani et al., 2016; Lomuscio and Maganti, 2017; Cheng et al., 2017; Tjeng et al., 2019), solving the dual problem (Dvijotham et al., 2018; Wong and Kolter, 2018; Raghunathan et al., 2018), and formulating and solving using SMT/SAT solvers (Katz et al., 2017; Ehlers, 2017; Huang et al., 2017). When the adversarial region is large, several techniques divide the domain into smaller subdomains and verify each of them (Singh et al., 2019; Bunel et al., 2017). Unlike the research presented in this paper, none of this prior research formalizes or attempts to verify that a neural network for perception computes *correct* (instead of stable) outputs within a specified tolerance for *all* inputs of interest (instead of local regions around labelled points).

Prior work on neural network testing focuses on constructing better test cases to expose problematic network behaviors. Researchers have developed approaches to build test cases that improve coverage

on possible states of the neural network, for example neuron coverage (Pei et al., 2017; Tian et al., 2018) and generalizations to multi-granular coverage (Ma et al., 2018) and MC/DC (Kelly J. et al., 2001) inspired coverage (Sun et al., 2018). Odena and Goodfellow (2018) presents coverage-guided fuzzing methods for testing neural networks using the above coverage criteria. Tian et al. (2018) generates realistic test cases by applying natural transformations (e.g. brightness change, rotation, add rain) to seed images. O'Kelly et al. (2018) uses simulation to test autonomous driving systems with deep learning based perception. Unlike this prior research, which tests the neural network on only a set of input points, the research presented in this paper verifies correctness for all inputs of interest.

3 Correctness Verification of Neural Networks

Consider the general perception problem of taking an input observation x and trying to predict some quantity of interest y. It can be a regression problem (continuous y) or a classification problem (discrete y). Some neural network model is trained for this task. We denote its function by $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the space of all possible inputs to the neural network and $\mathcal{Y}$ is the space of all possible outputs. Behind the input observation x there is some state of the world s. Denote $\mathcal{S}$ as the space of all states of the world that the network is expected to work in. For each state of the world, a set of possible inputs can be observed. We denote this *observation process* using a mapping $g : \mathcal{S} \to \mathcal{P}(\tilde{\mathcal{X}})$, where $g(s)$ is the set of inputs that can be observed from s. Here $\mathcal{P}(\cdot)$ is the power set, and $\tilde{\mathcal{X}} \subseteq \mathcal{X}$ is the *feasible input space*, the part of input space that may be observed from the *state space* $\mathcal{S}$. Concretely, $\tilde{\mathcal{X}} = \{x | \exists s \in \mathcal{S}, x \in g(s)\}$.

The quantity of interest y is some attribute of the state of the world. We denote the ground truth of y using a function $\lambda : \mathcal{S} \to \mathcal{Y}$. This specifies the ground truth for each input, which we denote as a mapping $\hat{f} : \tilde{\mathcal{X}} \to \mathcal{P}(\mathcal{Y})$. $\hat{f}(x)$ is the set of possible ground truth values of y for a given x:

$$\hat{f}(x) = \{y | \exists s \in \mathcal{S}, y = \lambda(s), x \in g(s)\}. \tag{1}$$

The feasible input space $\tilde{\mathcal{X}}$ and the ground truth mapping $\hat{f}$ together form a *specification*. In general, we cannot compute and represent $\tilde{\mathcal{X}}$ and $\hat{f}$ directly — indeed, the purpose of the neural network is to compute an approximation to this ground truth $\hat{f}$ which is otherwise not available given only the input x. $\tilde{\mathcal{X}}$ and $\hat{f}$ are instead determined implicitly by $\mathcal{S}$, g, and λ.

The error of the neural network is then characterized by the difference between f and $\hat{f}$. Concretely, the maximum possible error at a given input $x \in \tilde{\mathcal{X}}$ is:

$$e(x) = \max_{y \in \hat{f}(x)} d(f(x), y), \tag{2}$$

where $d(\cdot, \cdot)$ is some measurement on the size of the error between two values of the quantity of interest. For regression, we consider the absolute value of the difference $d(y_1, y_2) = |y_1 - y_2|$.[1] For classification, we consider a binary error measurement $d(y_1, y_2) = \mathbb{1}_{y_1 \neq y_2}$ (indicator function), i.e. the error is 0 if the prediction is correct, 1 if the prediction is incorrect.

The goal of correctness verification is to compute upper bounds on network prediction errors with respect to the specification. We formulate the problem of correctness verification formally here:

Problem formulation of correctness verification: Given a trained neural network f and a specification $(\tilde{\mathcal{X}}, \hat{f})$ determined implicitly by $\mathcal{S}$, g, and λ, compute upper bounds on error $e(x)$ for any feasible input $x \in \tilde{\mathcal{X}}$.

[1] For clarity, we formulate the problem with a one-dimensional quantity of interest. Extending to multidimensional output (multiple quantities of interest) is straightforward: we treat the prediction of each output dimension as using a separate neural network, all of which are the same except the final output layer.

4 *Tiler*

We next present *Tiler*, an algorithm for correctness verification of neural networks. We present here the algorithm for regression settings, with sufficient conditions for the resulting error bounds to be sound. The algorithm for classification settings is similar (see Appendix B).

Step 1: Divide the state space $\mathcal{S}$ into *state tiles* $\{\mathcal{S}_i\}$ such that $\cup_i \mathcal{S}_i = \mathcal{S}$.

The image of each $\mathcal{S}_i$ under g gives an *input tile* (a tile in the input space): $\mathcal{X}_i = \{x | x \in g(s), s \in \mathcal{S}_i\}$. The resulting tiles $\{\mathcal{X}_i\}$ satisfy the following condition:

Condition 4.1. $\tilde{\mathcal{X}} \subseteq \cup_i \mathcal{X}_i$.

Step 2: For each $\mathcal{S}_i$, compute the ground truth bound as an interval $[l_i, u_i]$, such that $\forall s \in \mathcal{S}_i, l_i \leq \lambda(s) \leq u_i$.

The bounds computed this way satisfy the following condition, which (intuitively) states that the possible ground truth values for an input point must be covered jointly by the ground truth bounds of all the input tiles that contain this point:

Condition 4.2(a). For any $x \in \tilde{\mathcal{X}}, \forall y \in \hat{f}(x), \exists \mathcal{X}_i$ such that $x \in \mathcal{X}_i$ and $l_i \leq y \leq u_i$.

Previous research has produced a variety of methods that bound the neural network output over given input region. Examples include layer-by-layer reachability analysis (Xiang et al., 2018; Gehr et al., 2018; Weng et al., 2018) and formulating constrained optimization problems (Bastani et al., 2016; Lomuscio and Maganti, 2017; Tjeng et al., 2019). Each method typically works for certain classes of networks (e.g. piece-wise linear networks) and certain classes of input regions (e.g. polytopes). For each input tile $\mathcal{X}_i$, we therefore introduce a bounding box $\mathcal{B}_i$ that 1) includes $\mathcal{X}_i$ and 2) is supported by the solving method:

Step 3: Using $\mathcal{S}_i$ and g, compute a bounding box $\mathcal{B}_i$ for each tile $\mathcal{X}_i = \{x | x \in g(s), s \in \mathcal{S}_i\}$.

The bounding boxes $\mathcal{B}_i$'s must satisfy the following condition:

Condition 4.3. $\forall i, \mathcal{X}_i \subseteq \mathcal{B}_i$.

Step 4: Given f and bounding boxes $\{\mathcal{B}_i\}$, use an appropriate solver to solve for the network output ranges $\{[l'_i, u'_i]\}$.

The neural network has a single output entry for each quantity of interest. Denote the value of the output entry as $o(x)$, $f(x) = o(x)$. The network output bounds (l'_i, u'_i) returned by the solver must satisfy the following condition:

Condition 4.4(a) $\forall x \in \mathcal{B}_i, l'_i \leq o(x) \leq u'_i$.

Step 5: For each tile, use the ground truth bound (l_i, u_i) and network output bound (l'_i, u'_i) to compute the error bound e_i:

$$e_i = \max(u'_i - l_i, u_i - l'_i). \tag{3}$$

e_i gives the upper bound on prediction error when the state of the world s is in $\mathcal{S}_i$. This is because (l_i, u_i) covers the ground truth values in $\mathcal{S}_i$, and (l'_i, u'_i) covers the possible network outputs for all inputs that can be generated from $\mathcal{S}_i$. From these error bounds $\{e_i\}$, we compute a global error bound:

$$e_{\text{global}} = \max_i e_i. \tag{4}$$

We can also compute a local error bound for any feasible input $x \in \tilde{\mathcal{X}}$:

$$e_{\text{local}}(x) = \max_{\{i | x \in \mathcal{B}_i\}} e_i. \tag{5}$$

Note that $\max_{\{i|x\in\mathcal{X}_i\}} e_i$ provides a tighter local error bound. But since it is generally much easier to check containment of x in $\mathcal{B}_i$'s than in $\mathcal{X}_i$'s, we adopt the current formulation.

Theorem 1 (Soundness of local error bound for regression)**.** *Given that Condition 4.1, 4.2(a), 4.3, and 4.4(a) are satisfied, then* $\forall x \in \tilde{\mathcal{X}}$, $e(x) \leq e_{local}(x)$, *where* $e(x)$ *is defined in Equation 2 and* $e_{local}(x)$ *is computed from Equation 3 and 5.*

Theorem 2 (Soundness of global error bound for regression)**.** *Given that Condition 4.1, 4.2(a), 4.3, and 4.4(a) are satisfied, then* $\forall x \in \tilde{\mathcal{X}}$, $e(x) \leq e_{global}$, *where* $e(x)$ *is defined in Equation 2 and* e_{global} *is computed from Equation 3 and 4.*

Algorithm 1 formally presents the *Tiler* algorithm (for regression). The implementations of DIVIDESTATESPACE, GETGROUNDTRUTHBOUND, and GETBOUNDINGBOX are problem dependent. The choice of SOLVER needs to be compatible with $\mathcal{B}_i$ and f. Conditions 4.1 to 4.4 specify the sufficient conditions for the returned results from these four methods such that the guarantees obtained are sound.

Algorithm 1 Tiler (for regression)

Input: $\mathcal{S}, g, \lambda, f$
Output: $e_{\text{global}}, \{e_i\}, \{\mathcal{B}_i\}$
1: **procedure** TILER($\mathcal{S}, g, \lambda, f$)
2: $\{\mathcal{S}_i\} \leftarrow$ DIVIDESTATESPACE($\mathcal{S}$) ▷ Step 1
3: **for** each $\mathcal{S}_i$ **do**
4: $(l_i, u_i) \leftarrow$ GETGROUNDTRUTHBOUND($\mathcal{S}_i, \lambda$) ▷ Step 2
5: $\mathcal{B}_i \leftarrow$ GETBOUNDINGBOX($\mathcal{S}_i, g$) ▷ Step 3
6: $(l_i', u_i') \leftarrow$ SOLVER($f, \mathcal{B}_i$) ▷ Step 4
7: $e_i \leftarrow \max(u_i' - l_i, u_i - l_i')$ ▷ Step 5
8: **end for**
9: $e_{\text{global}} \leftarrow \max(\{e_i\})$ ▷ Step 5
10: **return** $e_{\text{global}}, \{e_i\}, \{\mathcal{B}_i\}$ ▷ $\{e_i\}, \{\mathcal{B}_i\}$ can be used later to compute $e_{\text{local}}(x)$
11: **end procedure**

The complexity of this algorithm is determined by the number of tiles, which scales with the dimension of the state space $\mathcal{S}$. Because the computations for each tile are independent, our *Tiler* implementation executes these computations in parallel.

4.1 DEALING WITH NOISY OBSERVATION

Our formulation also applies to the case of noisy observations. Notice that the observation process g maps from a state to a set of possible inputs, so noise can be incorporated here. The above version of *Tiler* produces *hard* guarantees, i.e. the error bounds computed are valid for all cases. This works for observations with bounded noise. For cases where noise is unbounded (e.g. Gaussian noise), *Tiler* can be adjusted to provide probabilistic guarantees: we compute bounding boxes $\mathcal{B}_i$ such that $P(x \in \mathcal{B}_i | x \sim g(s), s \in \mathcal{S}_i) > 1 - \epsilon$ for some small ϵ. Here we also need the probability measure associated with the observation process — $g(s)$ now gives the probability distribution of input x given state s. This will give an error bound that holds with probability at least $1 - \epsilon$ for any state in this tile. We demonstrate how this is achieved in practice in the second case study (Section 6).

4.2 DETECTING ILLEGAL INPUTS

Tiler provides a way to verify the correctness of the neural network over the whole feasible input space $\tilde{\mathcal{X}}$. To make the system complete, we need a method to detect whether a new observed input is within $\tilde{\mathcal{X}}$ (the network is designed to work for it, and we have guaranteed correctness) or not (the network is not designed for it, so we don't have guarantees). In general, checking containment directly with $\tilde{\mathcal{X}}$ is hard, since there is no explicit representation of it. Instead, we use the bounding boxes $\{\mathcal{B}_i\}$ returned by *Tiler* as a proxy for $\tilde{\mathcal{X}}$: for a new input x^*, we check if x^* is contained in any of the $\mathcal{B}_i$'s. Since the network output ranges computed in Step 4 covers the inputs in each $\mathcal{B}_i$, and the error bounds incorporate the network output ranges, we know that the network output will not have unexpected drastic changes in $\mathcal{B}_i$'s. This makes $\mathcal{B}_i$'s a good proxy for the space of legal inputs.

Searching through all the $\mathcal{B}_i$'s can introduce a large overhead. We propose a way to speed up the search by utilizing the network prediction and the verified error bounds. Given the network prediction $y^* = f(x^*)$ and the global error bound e_{global}, we can prune the search space by discarding tiles that do not overlap with $[y^* - e_{\text{global}}, y^* + e_{\text{global}}]$ in the ground truth attribute. The idea is that we only need to search the local region in the state space that has ground truth attribute close to the prediction,

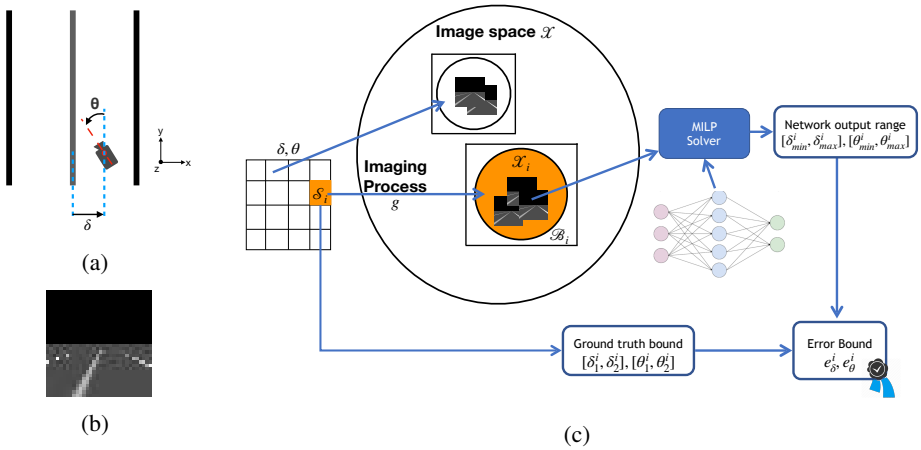


Figure 1: (a) Schematic of the scene. (b) Example image taken by the camera. (c) Schematics of *Tiler* applied on the road scene case.

since we have verified the bound for the maximum prediction error. We demonstrate this detection method and the prediction-guided search in the first case study (Section 5).

5 Case Study 1: Position Measurement from Road Scene

Problem set-up: Consider a world containing a road with a centerline, two side lines, and a camera taking images of the road. The camera is positioned at a fixed height above the road, but can vary its horizontal offset and viewing angle with respect to the centerline of the road. Figure 1a presents a schematic of the scene. The state of the world s is characterized by the offset δ and angle θ of the camera position. We therefore label the states as $s_{\delta,\theta}$. We consider the camera position between the range $\delta \in [-40, 40]$ (length unit of the scene, the road width from the centerline to the side lines is 50 units) and $\theta \in [-60°, 60°]$, so the state space $\mathcal{S} = \{s_{\delta,\theta} | \delta \in [-40, 40], \theta \in [-60°, 60°]\}$.

The input x to the neural network is the image taken by the camera. The observation process g is the camera imaging process. For each pixel, we shoot a ray from the center of that pixel through the camera focal point and compute the intersection of the ray with objects in the scene. The intensity of that intersection point is taken as the intensity of the pixel. The resulting x's are 32×32 gray scale images with intensities in $[0, 255]$ (see Appendix C.1 and C.2 for detailed descriptions of the scene and the camera imaging process). Figure 1b presents an example image. The feasible input space $\tilde{\mathcal{X}}$ is the set of all images that can be taken with $\delta \in [-40, 40]$ and $\theta \in [-60°, 60°]$.

The quantity of interest y is the camera position (δ, θ). The ground truth function λ is simply $\lambda(s_{\delta,\theta}) = (\delta, \theta)$. For the neural network, we use the same ConvNet architecture as $\mathbf{CNN_A}$ in Tjeng et al. (2019) and the *small* network in Wong et al. (2018). It has 2 convolutional layers (size 4×4, stride 2) with 16 and 32 filters respectively, followed by a fully connected layer with 100 units. All the activation functions are ReLUs. The output layer is a linear layer with 2 output nodes, corresponding to the predictions of δ and θ. The network is trained on 130k images and validated on 1000 images generated from our imaging process. The camera positions of the training and validation images are sampled uniformly from the range $\delta \in [-50, 50]$ and $\theta \in [-70°, 70°]$. The network is trained with an l_1-loss function, using *Adam* (Kingma and Ba, 2014) (see Appendix E for more training details).

For error analysis, we treat the predictions of δ and θ separately. The goal is to find upper bounds on the prediction errors $e_\delta(x)$ and $e_\theta(x)$ for any feasible input $x \in \tilde{\mathcal{X}}$.

Tiler Figure 1c presents a schematic of how we apply *Tiler* to this problem. Tiles are constructed by dividing $\mathcal{S}$ on (δ, θ) into a grid of equal-sized rectangles with length a and width b. Each cell in the grid is then $\mathcal{S}_i = \{s_{\delta,\theta} | \delta \in [\delta^i_1, \delta^i_2], \theta \in [\theta^i_1, \theta^i_2]\}$, with $\delta^i_2 - \delta^i_1 = a$ and $\theta^i_2 - \theta^i_1 = b$. Each tile $\mathcal{X}_i$ is the set of images that can be observed from $\mathcal{S}_i$. The ground truth bounds can be naturally obtained from $\mathcal{S}_i$: for δ, $l_i = \delta^i_1$ and $u_i = \delta^i_2$; for θ, $l_i = \theta^i_1$ and $u_i = \theta^i_2$.

We next encapsulate each tile $\mathcal{X}_i$ with an l_∞-norm ball $\mathcal{B}_i$ by computing, for each pixel, the range of possible values it can take within the tile. As the camera position varies in a cell $\mathcal{S}_i$, the intersection point between the ray from the pixel and the scene sweeps over a region in the scene. The range of intensity values in that region determines the range of values for that pixel. We compute this region for each pixel, then find the pixel value range (see Appendix C.3). The resulting $\mathcal{B}_i$ is an l_∞-norm ball in the image space covering $\mathcal{X}_i$, represented by 32×32 pixel-wise ranges.

To solve the range of outputs of the ConvNet for inputs in the l_∞-norm ball, we adopt the approach from (Tjeng et al., 2019). They formulate the robustness verification problem as mixed integer linear program (MILP). Presolving on ReLU stability and progressive bound tightening are used to improve efficiency. We adopt the same formulation but change the MILP objectives. For each l_∞-norm ball, we solve 4 optimization problems: maximizing and minimizing the output entry for δ, and another two for θ. Denote the objectives solved as δ^i_{min}, δ^i_{max}, θ^i_{min}, θ^i_{max}.

We then use Equation 3 to compute the error bounds for each tile: $e^i_\delta = \max(\delta^i_{max} - \delta^i_1, \delta^i_2 - \delta^i_{min})$, $e^i_\theta = \max(\theta^i_{max} - \theta^i_1, \theta^i_2 - \theta^i_{min})$. e^i_δ and e^i_θ give upper bounds on prediction error when the state of the world s is in cell $\mathcal{S}_i$. These error bounds can later be used to compute global and local error bounds as in Equation 4 and 5.

Experimental results We run *Tiler* with a cell size of 0.1 (the side length of each cell in the (δ, θ) grid is $a = b = 0.1$). The step that takes the majority of time is the optimization solver. With parallelism, the optimization step takes about 15 hours running on 40 CPUs@3.00 GHz, solving 960000×4 MILP problems.

Global error bound: We compute global error bounds by taking the maximum of e^i_δ and e^i_θ over all tiles. The global error bound for δ is 12.66, which is 15.8% of the measurement range (80 length units for δ); for θ is 7.13° (5.94% of the 120° measurement range). We therefore successfully verify the correctness of the network with these tolerances for all feasible inputs.

Error bound landscape: We present the visualizations of the error bound landscape by plotting the error bounds of each tile as heatmaps over the (δ, θ) space. Figures 2a and 2d present the resulting heatmaps for e^i_δ and e^i_θ, respectively. To further inspect the distribution of the error bounds, we compute the percentage of the state space $\mathcal{S}$ (measured on the (δ, θ) grid) that has error bounds below some threshold value. The percentage varying with threshold value can be viewed as a cumulative distribution. Figures 2c and 2f present the cumulative distributions of the error bounds. It can be seen that most of the state space can be guaranteed with much lower error bounds, with only a small percentage of the regions having larger guarantees. This is especially the case for the offset measurement: 99% of the state space is guaranteed to have error less than 2.65 (3.3% of the measurement range), while the global error bound is 12.66 (15.8%).

A key question is how well the error bounds reflect the actual maximum error made by the neural network. To study the tightness of the error bounds, we compute empirical estimates of the maximum errors for each $\mathcal{S}_i$, denoted as $\bar{e}^i_\delta$ and $\bar{e}^i_\theta$. We sample multiple (δ, θ) within each cell $\mathcal{S}_i$, generate input images for each (δ, θ), then take the maximum over the errors of these points as the empirical estimate of the maximum error for $\mathcal{S}_i$. The sample points are drawn on a sub-grid within each cell, with sampling spacing 0.05. This estimate is a lower bound on the maximum error for $\mathcal{S}_i$, providing a reference for evaluating the tightness of the error upper bounds we get from *Tiler*.

We take the maximum of $\bar{e}^i_\delta$'s and $\bar{e}^i_\theta$'s to get a lower bound estimate of the global maximum error. The lower bound estimate of the global maximum error for δ is 9.12 (11.4% of the measurement range); for θ is 4.08° (3.4% of the measurement range). We can see that the error bounds that *Tiler* delivers are close to the lower bound estimates derived from the observed errors that the network exhibits for specific inputs.

Having visualized the heatmaps for the bounds e^i_δ and e^i_θ, we subtract the error estimates $\bar{e}^i_\delta$ and $\bar{e}^i_\theta$ and plot the heatmaps for the resulting gaps in Figures 2b and 2e. We can see that most of the regions that have large error bounds are due to the fact that the network itself has large errors there. By computing the cumulative distributions of these gaps between bounds and estimates, we found that for angle measurement, 99% of the state space has error gap below 1.9° (1.6% of measurement range); and for offset measurement, 99% of the state space has error gap below 1.41 length units (1.8%). The gaps indicate the maximum possible improvements on the error bounds.

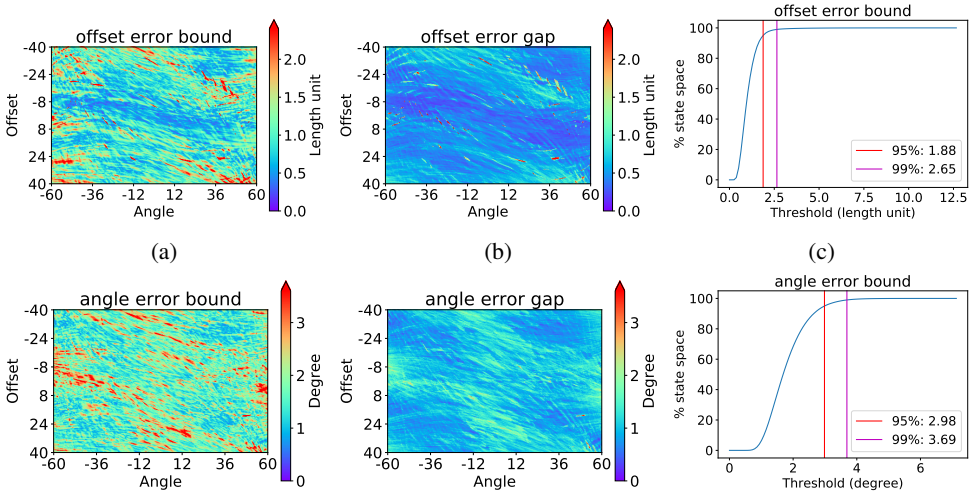


Figure 2: (a,d) Heatmaps of the upper bounds on the maximum error of each tile over the offset-angle space. (b,e) Corresponding heatmaps after subtracting empirical estimates of the actual maximum error. (c,f) Percentage of the state space with error upper bounds below some threshold value (cumulative distribution).

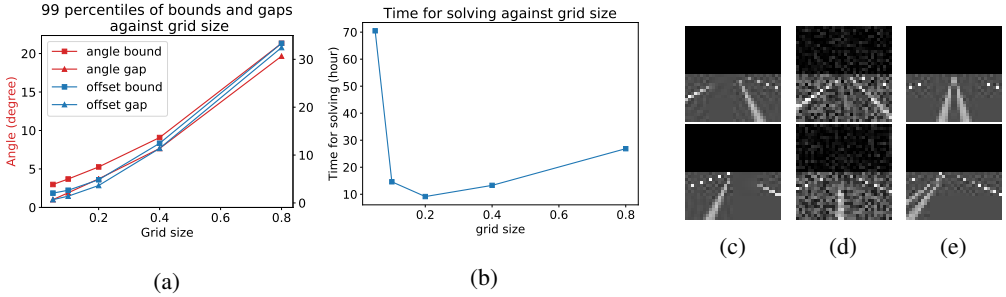
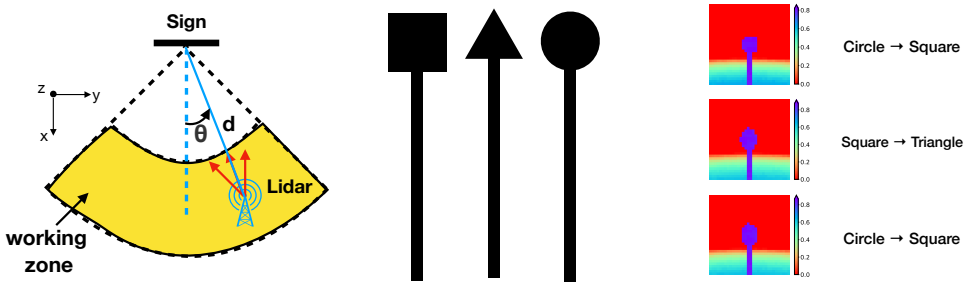


Figure 3: (a) 99 percentiles of the error upper bounds and the gaps between upper and lower bounds against cell size. (b) Total solving time against cell size. (c) Example legal inputs. (d) Example corrupted inputs. (e) Example inputs from new scene.

Contributing factors to the gap: The first factor is that we use interval arithmetic to compute the error bound in *Tiler*: *Tiler* takes the maximum distance between the range of possible ground truths and the range of possible network outputs as the bound. The second factor is the extra space included in the box $\mathcal{B}_i$ that is not in the tile $\mathcal{X}_i$. This results in a larger range on network output being used for calculating error bound, which in turn makes the error bound itself larger.

Effect of tile size: Both of the factors described above are affected by the tile size. We run *Tiler* with a sequence of cell sizes (0.05, 0.1, 0.2, 0.4, 0.8) for the (δ, θ) grid. Figure 3a shows how the 99 percentiles of the error upper bounds and the gap between error bounds and estimates vary with cell size. As tile size gets finer, *Tiler* provides better error bounds, and the tightness of bounds improves.

These results show that we can get better error bounds with finer tile sizes. But this improvement might be at the cost of time: reducing tile sizes also increases the total number of tiles and the number of optimization problems to solve. Figure 3b shows how the total solving time varies with cell size. For cell sizes smaller than 0.2, the trend can be explained by the above argument. For cell sizes larger than 0.2, total solving time increases with cell size instead. The reason is each optimization problem becomes harder to solve as the tile becomes large. Specifically, the approach we adopt (Tjeng et al., 2019) relies on the presolving on ReLU stability to improve speed. The number of unstable ReLUs will increase drastically as the cell size becomes large, which makes the solving slower.



(a) (b) (c)

Figure 4: (a) Schematic for case study 2. (b) Shapes of the sign. (c) Example inputs in the bounding boxes that cause misclassification. The inputs are visualized using their values after preprocessing of the neural network. The left hand side is the ground truth label, and the right hand side is the misclassified label.

Detecting illegal inputs: We implement the input detector by checking if the new input x^* is contained in any of the bounding boxes $\mathcal{B}_i$. We test the detector with 3 types of inputs: 1) legal inputs, generated from the state space through the imaging process; 2) corrupted inputs, obtained by applying i.i.d uniformly distributed per-pixel perturbation to legal inputs; 3) inputs from a new scene, where the road is wider and there is a double centerline. Figure 3c to 3e show some example images for each type. We randomly generated 500 images for each type. Our detector is able to flag all inputs from type 1 as legal, and all inputs from type 2 and 3 as illegal. On average, naive search (over all $\mathcal{B}_i$) takes 1.04s per input, while prediction-guided search takes 0.04s per input. So the prediction-guided search gives a 26$\times$ speedup without any compromise in functionality.

6 Case Study 2: Shape Classification from LiDAR Sensing

Problem set-up The world in this case contains a planar sign standing on the ground. There are 3 types of signs with different shapes: square, triangle, and circle (Figure 4b). A LiDAR sensor takes measurement of the scene, which is used as input to a neural network to classify the shape of the sign. The sensor can vary its distance d and angle θ with respect to the sign, but its height is fixed, and it is always facing towards the sign. Figure 4a shows the schematic of the set-up. Assume the working zone for the LiDAR sensor is with position $d \in [30, 60]$ and $\theta \in [-45°, 45°]$. Then the state space $\mathcal{S}$ has 3 dimensions: two continuous (d and θ), and one discrete (sign shape c). The LiDAR sensor emits an array of 32$\times$32 laser beams in fixed directions. The measurement from each beam is the distance to the first object hit in that direction. We consider a LiDAR measurement model where the maximum distance that can be measured is `MAX_RANGE`=300, and the measurement has a Gaussian noise with zero mean and a standard deviation of 0.1% of `MAX_RANGE`. This gives the observation process g. Appendix D.1 and D.2 provides more details on the scene and LiDAR measurement model.

We use a CNN with 2 convolutional layers (size 4$\times$4) with 16 filters, followed by a fully connected layer with 100 units. The distance measurements are preprocessed before feeding into the network: first dividing them by `MAX_RANGE` to scale to [0,1], then using 1 minus the scaled distances as inputs. This helps the network training. We train the network using 50k points from each class, and validating using 500 points from each class. The training settings are the same as the previous case study (Appendix E).

Tiler The state tiles are constructed in each of the three shape subspaces. We divide the θ dimension uniformly into 90 intervals and the d dimension uniformly in the inverse scale into 60 intervals to obtain a grid with 5400 cells per shape. To compute the bounding box $\mathcal{B}_i$ for a given tile $\mathcal{S}_i$, we first find a lower bound and an upper bound on the distance of object for each beam as the sensor position varies within that tile $\mathcal{S}_i$ (Appendix D.3). We extend this lower and upper bound by 5σ, where σ is the standard deviation of the Gaussian measurement noise. This way we have $P(x \in \mathcal{B}_i | x \sim g(s), s \in \mathcal{S}_i) >= (P(|a| <= 5\sigma | a \sim \mathcal{N}(0, \sigma^2)))^N > 0.999$, where $N = 32 \times 32$ is the input dimension. The factor 5 can be changed, depending on the required probabilistic guarantee.

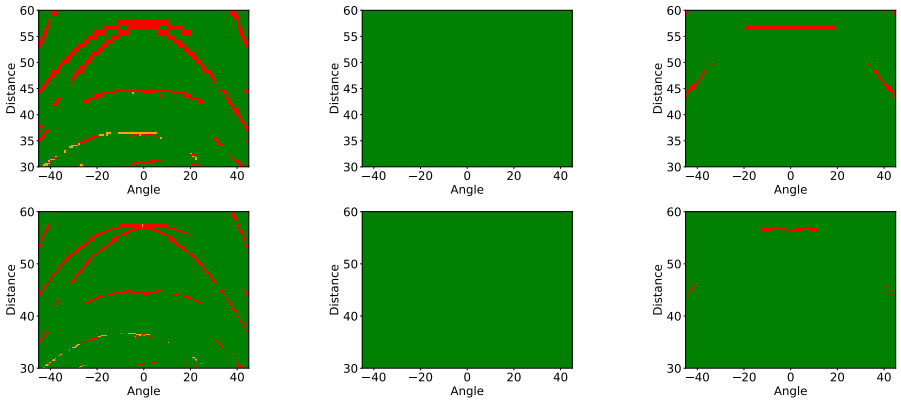


Figure 5: Verification results over the state space for the LiDAR case study. The top row corresponds to tiling with a grid of 90×60 cells, the bottom row corresponds to a grid of 180×120 cells. Green cells indicate that the tile is verified ($e_i = 0$); red cells indicate that we cannot verify that tile ($e_i = 1$); orange cells indicate that the solver exceeds time limit while solving for that tile.

Same as the previous case study, we adopt the MILP method (Tjeng et al., 2019) to solve the network output, which is used to decide whether the tile is verified to be correct or not.

Experimental results We plot the verification results as heatmaps over the state space in Figure 5 (top row). We are able to verify the correctness of the network over the majority of the state space. In particular, we verify that the network is always correct when the shape of the sign is triangle.

Besides the tiling described above, we also run a finer tiling with half the cell sizes in both d and θ. Figure 5 (bottom row) shows the verification results. By reducing the tile sizes, we can verify more regions in the state space.

For the tiles that we are unable to verify correctness (red squares in the heatmaps), there are inputs within those bounding boxes on which the network will predict a different class. We inspect several of such tiles to see the inputs that cause problems. Figure 4c shows a few examples. In some of the cases (top two in figure), the 'misclassified' inputs actually do not look like coming from the ground truth class. This is because the extra space included in $\mathcal{B}_i$ is too large, so that it includes inputs that are reasonably different from feasible inputs. Such cases will be reduced as the tile size becomes smaller, since the extra spaces in $\mathcal{B}_i$'s will be shrunk. We have indeed observed such phenomenon, as we can verify more regions when the tile size becomes smaller. In some other cases (bottom example), however, the misclassified inputs are perceptually similar to the ground truth class. Yet the network predicts a different class. This reveals that the network is potentially not very robust on inputs around these points. In this sense, our framework provides a way to systematically find regions of the input space of interests where the network is potentially vulnerable.

7 Discussion

The techniques presented in this paper work with specifications provided by the combination of a state space of the world and an observation process that converts states into neural network inputs. Results from the case studies highlight how well the approach works for a state space characterized by several attributes and a camera imaging or LiDAR measurement observation process. We anticipate that the technique will also work well for other problems that have a low dimensional state space (but potentially a high dimensional input space). For higher dimensional state spaces, the framework makes it possible to systematically target specific regions of the input space to verify. Potential applications include targeted verification, directed testing, and the identification of illegal inputs for which the network is not expected to work on.

REFERENCES

Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. 2016. Measuring Neural Net Robustness with Constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., USA, 2621–2629. http://dl.acm.org/citation.cfm?id=3157382.3157391

Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. 2017. A Unified View of Piecewise Linear Neural Network Verification. arXiv:cs.AI/1711.00455

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum Resilience of Artificial Neural Networks. In *ATVA*.

Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. 2018. A dual approach to scalable verification of deep networks. In *Proceedings of the Thirty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-18)*. AUAI Press, Corvallis, Oregon, 162–171.

Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. *CoRR* abs/1705.01320 (2017). arXiv:1705.01320 http://arxiv.org/abs/1705.01320

T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 3–18. https://doi.org/10.1109/SP.2018.00058

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–29.

Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification*. Springer International Publishing, 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). http://arxiv.org/abs/1412.6980

Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. 2019. Algorithms for Verifying Deep Neural Networks. *CoRR* abs/1903.06758 (2019). arXiv:1903.06758 http://arxiv.org/abs/1903.06758

Alessio Lomuscio and Lalit Maganti. 2017. An approach to reachability analysis for feed-forward ReLU neural networks. *CoRR* abs/1706.07351 (2017). arXiv:1706.07351 http://arxiv.org/abs/1706.07351

Lei Ma, Felix Juefei-Xu, Jiyuan Sun, Chunyang Chen, Ting Su, Fuyuan Zhang, Minhui Xue, Bo Li, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Comprehensive and Multi-Granularity Testing Criteria for Gauging the Robustness of Deep Learning Systems. *CoRR* abs/1803.07519 (2018). arXiv:1803.07519 http://arxiv.org/abs/1803.07519

Augustus Odena and Ian J. Goodfellow. 2018. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. *CoRR* abs/1807.10875 (2018).

Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, John Duchi, and Russ Tedrake. 2018. Scalable End-to-End Autonomous Vehicle Testing via Rare-event Simulation. arXiv:cs.LG/1811.00145

Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), 372–387.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Certified Defenses against Adversarial Examples. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bys4ob-Rb

Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. 2019. A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks. arXiv:cs.LG/1902.08722

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An Abstract Domain for Certifying Neural Networks. *Proc. ACM Program. Lang.* 3, POPL, Article 41 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290354

Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *CoRR* abs/1803.04792 (2018). arXiv:1803.04792 http://arxiv.org/abs/1803.04792

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. *CoRR* abs/1312.6199 (2014).

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HyGIdiRqtm

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

Eric Wong and Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 5286–5295. http://proceedings.mlr.press/v80/wong18a.html

Eric Wong, Frank Schmidt, Jan Hendrik Metzen, and J. Zico Kolter. 2018. Scaling provable adversarial defenses. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 8400–8409. http://papers.nips.cc/paper/8060-scaling-provable-adversarial-defenses.pdf

Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. 2017. Reachable Set Computation and Safety Verification for Neural Networks with ReLU Activations. *CoRR* abs/1712.08163 (2017). arXiv:1712.08163 http://arxiv.org/abs/1712.08163

Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. 2018. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* (March 2018). https://doi.org/10.1109/TNNLS.2018.2808470

APPENDICES

A PROOFS FOR THEOREM 1 AND 2

Theorem 1 (Local error bound for regression)**.** *Given that Condition 4.1, 4.2(a), 4.3, and 4.4(a) are satisfied, then* $\forall x \in \tilde{\mathcal{X}}$*,* $e(x) \leq e_{local}(x)$*, where* $e(x)$ *is defined in Equation 2 and* $e_{local}(x)$ *is computed from Equation 3 and 5.*

Proof. For any $x \in \tilde{\mathcal{X}}$, we have

$$\begin{aligned} e(x) &= \max_{y \in \hat{f}(x)} d(f(x), y) \\ &= \max_{y \in \hat{f}(x)} |f(x) - y| \\ &= \max_{y \in \hat{f}(x)} \{\max(f(x) - y, y - f(x))\} \end{aligned}$$

Condition 4.1 and 4.2(a) guarantees that for any $x \in \tilde{\mathcal{X}}$ and $y \in \hat{f}(x)$, we can find a tile $\mathcal{X}_i$ such that $x \in \mathcal{X}_i$ and $l_i \leq y \leq u_i$. Let $t(y, x)$ be a function that gives such a tile $\mathcal{X}_{t(y,x)}$ for a given x and $y \in \hat{f}(x)$. Then

$$e(x) \leq \max_{y \in \hat{f}(x)} \{\max(f(x) - l_{t(y,x)}, u_{t(y,x)} - f(x))\}.$$

Since $x \in \mathcal{X}_{t(y,x)}$ and $\mathcal{X}_{t(y,x)} \subseteq \mathcal{B}_{t(y,x)}$ (Condition 4.3), $x \in \mathcal{B}_{t(y,x)}$. By Condition 4.4(a),

$$l'_{t(y,x)} \leq f(x) \leq u'_{t(y,x)}, \forall y \in \hat{f}(x).$$

This gives

$$\begin{aligned} e(x) &\leq \max_{y \in \hat{f}(x)} \{\max(u'_{t(y,x)} - l_{t(y,x)}, u_{t(y,x)} - l'_{t(y,x)})\} \\ &= \max_{y \in \hat{f}(x)} e_{t(y,x)} \end{aligned}$$

Since $x \in \mathcal{B}_{t(y,x)}$ for all $y \in \hat{f}(x)$, we have $\{t(y, x) | y \in \hat{f}(x)\} \subseteq \{i | x \in \mathcal{B}_i\}$, which gives

$$\begin{aligned} e(x) &\leq \max_{\{t(y,x) | y \in \hat{f}(x)\}} e_{t(y,x)} \\ &\leq \max_{\{i | x \in \mathcal{B}_i\}} e_i \\ &= e_{\text{local}}(x) \end{aligned}$$

□

Theorem 2 (Global error bound for regression)**.** *Given that Condition 4.1, 4.2(a), 4.3, and 4.4(a) are satisfied, then* $\forall x \in \tilde{\mathcal{X}}$*,* $e(x) \leq e_{global}$*, where* $e(x)$ *is defined in Equation 2 and* e_{global} *is computed from Equation 3 and 4.*

Proof. By Theorem 1, we have $\forall x \in \tilde{\mathcal{X}}$,

$$\begin{aligned} e(x) &\leq \max_{\{i | x \in \mathcal{B}_i\}} e_i \\ &\leq \max_i e_i \\ &= e_{\text{global}} \end{aligned}$$

□

B CLASSIFICATION

We present here the algorithm of *Tiler* for classification settings.

Step 1 (tiling the space) is the same as regression.

Step 2: For each $\mathcal{S}_i$, compute the ground truth bound as a set $\mathcal{C}_i \subseteq \mathcal{Y}$, such that $\forall s \in \mathcal{S}_i, \lambda(s) \in \mathcal{C}_i$.

The bounds computed this way satisfy the following condition:

Condition 4.2(b). For any $x \in \tilde{\mathcal{X}}, \forall y \in \hat{f}(x), \exists \mathcal{X}_i$ such that $x \in \mathcal{X}_i$ and $y \in \mathcal{C}_i$.

The idea behind *Condition 4.2(b)* is the same as that of *Condition 4.2(a)*, but formulated for discrete y. For tiles with $\mathcal{C}_i$ containing more than 1 class, we cannot verify correctness since there is more than 1 possible ground truth class for that tile. Therefore, we should try to make the state tiles containing only 1 ground truth class each when tiling the space in step 1. For tiles with $|\mathcal{C}_i| = 1$, we proceed to the following steps.

Step 3 (compute bounding box for each input tile) is the same as regression.

The next step is to solve the network output range. Suppose the quantity of interest has K possible classes. Then the output layer of the neural network is typically a softmax layer with K output nodes. Denote the k-th output score before softmax as $o_k(x)$. We use the solver to solve the minimum difference between the output score for the ground truth class and each of the other classes:

Step 4: Given f, $\mathcal{B}_i$, and the ground truth class c_i (the only element in $\mathcal{C}_i$), use appropriate solver to solve lower bounds on the difference between the output score for class c_i and each of the other classes: $l_i'^{(k)}$ for $k \in \{1, \ldots, K\} \setminus \{c_i\}$. The bounds need to satisfy:

Condition 4.4(b) $\forall x \in \mathcal{B}_i, \forall k \in \{1, \ldots, K\} \setminus \{c_i\}, l_i'^{(k)} \leq o_{c_i}(x) - o_k(x)$.

Step 5: For each tile, compute an error bound e_i, with $e_i = 0$ meaning the network is guaranteed to be correct for this state tile, and $e_i = 1$ meaning no guarantee:

$$e_i = 0 \text{ if and only if } \forall k \in \{1, \ldots, K\} \setminus \{c_i\}, l_i'^{(k)} \geq 0 \tag{6}$$

Otherwise, $e_i = 1$. We can then compute the global and local error bounds using Equation 4 and 5, same as in the regression case.

Theorem 3 (Local error bound for classification)**.** *Given that Condition 4.1, 4.2(b), 4.3, and 4.4(b) are satisfied, then* $\forall x \in \tilde{\mathcal{X}}$, $e(x) \leq e_{local}(x)$, *where* $e(x)$ *is defined in Equation 2 and* $e_{local}(x)$ *is computed from Equation 6 and 5. Equivalently, when* $e_{local}(x) = 0$, *the network prediction is guaranteed to be correct at* x.

Proof. We aim to prove that $\forall x \in \tilde{\mathcal{X}}$, if $e_{\text{local}}(x) = 0$, then $f(x) = \hat{f}(x)$. First, according to the definition of $e_{\text{local}}(x)$, $e_{\text{local}}(x) = 0$ implies $e_i = 0$ for all $i \in \{i | x \in \mathcal{B}_i\}$. Arbitrarily pick a $p \in \{i | x \in \mathcal{B}_i\}$, we have $e_p = 0$. Then $\mathcal{C}_p$ only contains one element with class index c_p. Condition 4.4(b) gives:

$$\forall k \in \{1, \ldots, K\} \setminus \{c_p\}, o_{c_p}(x) - o_k(x) \geq l_p'^{(k)}$$

By Equation 6, $\forall k \in \{1, \ldots, K\} \setminus \{c_p\}, l_p'^{(k)} \geq 0$. Therefore we have:

$$o_{c_p}(x) \geq o_k(x)$$

for all $k \neq c_p$. This gives $f(x) = c_p$. For any $q \in \{i | x \in \mathcal{B}_i, i \neq p\}$, we can similarly derive $f(x) = c_q$. Therefore, we must have $c_q = c_p$ for all $q \in \{i | x \in \mathcal{B}_i, i \neq p\}$, otherwise there will be a contradiction. In another word, we have

$$\bigcup_{\{i | x \in \mathcal{B}_i\}} \mathcal{C}_i = \{c_p\}$$

Now, according to Condition 4.2(b), $\forall y \in \hat{f}(x), \exists \mathcal{X}_i$ such that $x \in \mathcal{X}_i$ and $y \in \mathcal{C}_i$. Rewriting this condition, we get

$$\forall y \in \hat{f}(x), y \in \bigcup_{\{i | x \in \mathcal{X}_i\}} \mathcal{C}_i$$

which also means $\hat{f}(x) \subseteq \bigcup_{\{i|x\in\mathcal{X}_i\}} \mathcal{C}_i$. But $\{i|x \in \mathcal{X}_i\} \subseteq \{i|x \in \mathcal{B}_i\}$ (Condition 4.3), which gives

$$\hat{f}(x) \subseteq \bigcup_{\{i|x\in\mathcal{X}_i\}} \mathcal{C}_i \subseteq \bigcup_{\{i|x\in\mathcal{B}_i\}} \mathcal{C}_i = \{c_p\}.$$

Since $\hat{f}(x)$ is not empty, we have $\hat{f}(x) = \{c_p\} = f(x)$. □

Theorem 4 (Global error bound for classification)**.** *Given that Condition 4.1, 4.2(b), 4.3, and 4.4(b) are satisfied, then if $e_{global} = 0$, the network prediction is guaranteed to be correct for all $x \in \tilde{\mathcal{X}}$. e_{global} is computed from Equation 6 and 4.*

Proof. We aim to prove that if $e_{\text{global}} = 0$, then $\forall x \in \tilde{\mathcal{X}}$, $f(x) = \hat{f}(x)$. According to Equation 4, $e_{\text{global}} = 0$ means $e_i = 0$ for all i. Then $\forall x \in \tilde{\mathcal{X}}$, $e_{\text{local}}(x) = 0$, which by Theorem 3 indicates that $f(x) = \hat{f}(x)$. □

Algorithm 2 formally presents the *Tiler* algorithm for classification.

Algorithm 2 Tiler (for classification)

Input: $\mathcal{S}, g, \lambda, f$
Output: $e_{\text{global}}, \{e_i\}, \{\mathcal{B}_i\}$

procedure TILER($\mathcal{S}, g, \lambda, f$)
  $\{\mathcal{S}_i\} \leftarrow$ DIVIDESTATESPACE($\mathcal{S}$) ▷ Step 1
  for each $\mathcal{S}_i$ **do**
    $c_i \leftarrow$ GETGROUNDTRUTHBOUND($\mathcal{S}_i, \lambda$) ▷ Step 2
    $\mathcal{B}_i \leftarrow$ GETBOUNDINGBOX($\mathcal{S}_i, g$) ▷ Step 3
    $\{l_i'^{(k)}\}_{k\neq c_i} \leftarrow$ SOLVER($f, \mathcal{B}_i, c_i$) ▷ Step 4
    $e_i \leftarrow$ GETERRORBOUND($\{l_i'^{(k)}\}_{k\neq c_i}$) ▷ Step 5, Equation 6
  end for
  $e_{\text{global}} \leftarrow \max(\{e_i\})$ ▷ Step 5
  return $e_{\text{global}}, \{e_i\}, \{\mathcal{B}_i\}$ ▷ $\{e_i\}, \{\mathcal{B}_i\}$ can be used later to compute $e_{\text{local}}(x)$
end procedure

C Position Measurement from Road Scene

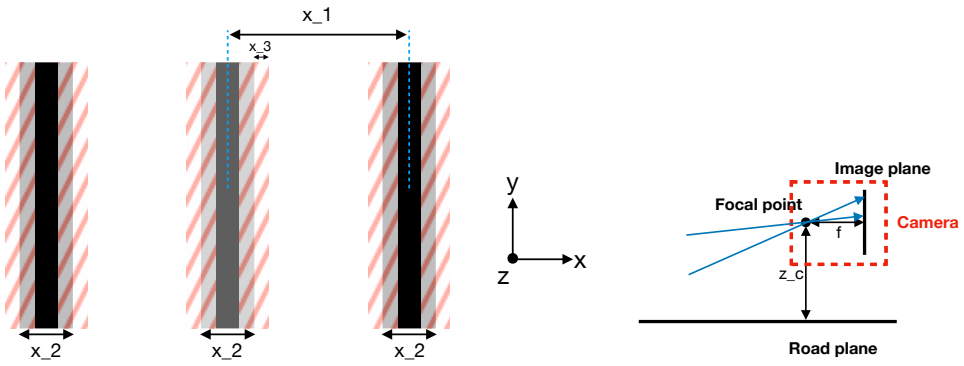
C.1 Scene

For clarity, we describe the scene in a Cartesian coordinate system. Treating 1 length unit as roughly 5 centimeters will give a realistic scale. The scene contains a road in the xy-plane, extending along the y-axis. A schematic view of the road down along the z-axis is shown in Figure 6a. The road contains a centerline and two side lines, each with width $\mathrm{x}_2 = 4.0$. The width of the road (per lane) is $\mathrm{x}_1 = 50.0$, measuring from the center of the centerline to the center of each side line. Each point in the scene is associated with an intensity value in $[0.0, 1.0]$ grayscale. The intensity of the side lines is $i_1 = 1.0$, centerline $i_2 = 0.7$, road $i_3 = 0.3$, and sky $i_4 = 0.0$. The intensity adopts a ramp change at each boundary between the lines and the road. The half width of the ramp is $\mathrm{x}_3 = 1.0$.

The schematic of the camera is shown in Figure 6b. The camera's height above the road is fixed at $\mathrm{z}_c = 20.0$. The focal length $f = 1.0$. The image plane is divided into 32×32 pixels, with pixel side length $d = 0.16$.

C.2 Camera Imaging Process

The camera imaging process we use can be viewed as an one-step ray tracing: the intensity value of each pixel is determined by shooting a ray from the center of that pixel through the focal point, and take the intensity of the intersection point between the ray and the scene. In this example scene, the intersection points for the top half of the pixels are in the sky (intensity 0.0). The intersection points for the lower half of the pixels are on the xy-plane. The position of the intersection point (in the



(a) (b)

Figure 6: (a) Schematic of the top view of the road. (b) Schematic of the camera.

Table 1: Scene parameters

Parameter	Description	Value	Parameter	Description	Value
x_1	Road width	50.0	i_1	Intensity (side line)	1.0
x_2	Line width	4.0	i_2	Intensity (centerline)	0.7
x_3	Ramp half width	1.0	i_3	Intensity (road)	0.3
z_c	Camera height	20.0	i_4	Intensity (sky)	0.0
f	Focal length	1.0	n	Pixel number	32
d	Pixel side length	0.16			

world coordinate) can be computed using a transformation from the pixel coordinate in homogeneous coordinate systems: [2]

$$\tilde{X}_W^{(i,j)} = T_{FW} \cdot P_p \cdot R_{CF} \cdot P_{PC} \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \tag{7}$$

P_{PC} is the transformation from pixel coordinate to camera coordinate. Camera coordinate has the origin at the focal point, and axes aligned with the orientation of the camera. We define the focal coordinate to have its origin also at the focal point, but with axes aligned with the world coordinate (the coordinate system used in Appendix C.1). R_{CF} is the rotation matrix that transforms from camera coordinate to focal coordinate. P_p represents the projection to the road plane through the focal point, in the focal coordinate. Finally, T_{FW} is the translation matrix that transforms from the focal coordinate to the world coordinate. The transformation matrices are given below:

$$T_{FW} = \begin{bmatrix} 1 & 0 & 0 & \delta \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathrm{z}_c \\ 0 & 0 & 0 & 1 \end{bmatrix}, P_p = \begin{bmatrix} -\mathrm{z}_c & 0 & 0 & 0 \\ 0 & -\mathrm{z}_c & 0 & 0 \\ 0 & 0 & -\mathrm{z}_c & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$R_{CF} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, P_{PC} = \begin{bmatrix} 0 & d & \frac{d}{2} - \frac{nd}{2} \\ 0 & 0 & f \\ -d & 0 & -(\frac{d}{2} - \frac{nd}{2}) \\ 0 & 0 & 1 \end{bmatrix}$$

[2]Note that for this imaging process, flipping the image on the image plane to the correct orientation (originally upside-down) is equivalent to taking the image on a virtual image plane that is in front of the focal point by the focal length, by shooting rays from the focal point through the (virtual) pixel centers. We compute the intersection points from the pixel coordinates on the virtual image plane.

The variables are defined as in Table 1, with δ and θ being the offset and angle of the camera.

After the intensity values of the pixels are determined, they are scaled and quantized to the range $[0, 255]$, which are used as the final image taken.

C.3 Method to Compute Pixel-wise Range for Each Tile

In the road scene example, we need to encapsulate each tile in the input space with a l_∞-norm ball for the MILP solver to solve. This requires a method to compute a range for each pixel that covers all values this pixel can take for images in this tile. This section presents the method used in this paper.

A tile in this example corresponds to images taken with camera position in a local range $\delta \in [\delta_1, \delta_2]$, $\theta \in [\theta_1, \theta_2]$. For pixels in the upper half of the image, their values will always be the intensity of the sky. For each pixel in the lower half of the image, if we trace the intersection point between the projection ray from this pixel and the road plane, it will sweep over a closed region as the camera position varies in the δ-θ cell. The range of possible values for that pixel is then determined by the range of intensities in that region. In this example, there is an efficient way of computing the range of intensities in the region of sweep. Since the intensities on the road plane only varies with x, it suffices to find the span on x for the region. The extrema on x can only be achieved at: 1) the four corners of the δ-θ cell; 2) the points on the two edges $\delta = \delta_1$ and $\delta = \delta_2$ where θ gives the ray of that pixel perpendicular to the y axis (if that θ is contained in $[\theta_1, \theta_2]$). Therefore, by computing the location of these critical points, we can obtain the range of x. We can then obtain the range of intensities covered in the region of sweep, which will give the range of pixel values.

D Sign Classification using LiDAR Sensor

D.1 Scene

The sign is a planer object standing on the ground. It consists of a holding stick and a sign head on top of the stick. The stick is of height 40.0 and width 2.0 (all in terms of the length unit of the scene). The sign head has three possible shapes: square (with side length 10.0), equilateral triangle (with side length 10.0), and circle (with diameter 10.0). The center of the sign head coincides with the middle point of the top of the stick.

The LiDAR sensor can vary its position within the working zone (see Figure 4a). Its height is fixed at 40.0 above the ground. The center direction of the LiDAR is always pointing parallel to the ground plane towards the centerline of the sign.

D.2 LiDAR Measurement Model

The LiDAR sensor emits an array of 32×32 laser beams and measures the distance to the first object along each beam. The directions of the beams are arranged as follows. At distance $f = 4.0$ away from the center of the sensor, there is a (imaginary) 32×32 grid with cell size 0.1. There is a beam shooting from the center of the sensor through the center of each cell in the grid.

The LiDAR model we consider has a maximum measurement range of `MAX_RANGE`=300.0. If the distance of the object is larger than `MAX_RANGE`, the reflected signal is too weak to be detected. Therefore, all the distances larger than `MAX_RANGE` will be measured as `MAX_RANGE`. The distance measurement contains a Guassian noise $n \sim \mathcal{N}(0, \sigma^2)$, where $\sigma = 0.001 \times$ `MAX_RANGE`. So the measured distance for a beam is given by

$$d = d_0 + n, n \sim \mathcal{N}(0, \sigma^2)$$

where d_0 is the actual distance and d is the measured distance.

D.3 Method to compute distance bounds for each beam

To compute the bounding boxes in *Tiler*, we need to compute a lower bound and an upper bound on the measured distance for each beam as the sensor position varies within a tile. We first determine whether the intersection point p between the beam and the scene is 1) always on the sign, 2) always on the background (ground/sky), or 3) covers both cases, when the sensor position varies within the

tile. Denote the d range of the tile as $[d_1, d_2]$ and the θ range as $[\theta_1, \theta_2]$. We find the intersection point p at a list of critical positions: (d_1, θ_1), (d_1, θ_2), (d_2, θ_1), (d_2, θ_2). If at some angle $\theta^* \in [\theta_1, \theta_2]$, the beam direction has no component along the $\mathtt{y}$-axis, then we add (d_1, θ^*) to the list of critical positions. The cases for p (on sign/background) covered in the critical points determine the cases covered in the whole tile.

In most situations, the distances of p at the list of critical positions (we refer it as the list of critical distances) contain the maximum and minimum distances of p in the tile. There is one exception: at $d = d_1$, as θ varies in $[\theta_1, \theta_2]$, if the intersection point shifts from sign to background (or vice versa), then the minimum distance of p can occur at (d_1, θ') where θ' is not equal to θ_1 or θ_2. To handle this case, if at the previous step we find that p do occur on the sign plane in the tile, then we add the distance of the intersection point between the beam and the *sign plane* at position (d_1, θ_1) (or (d_1, θ_2), whichever gives a smaller distance) to the list of critical distances. Notice that this intersection point is not necessarily on the *sign*. In this way, the min and max among the critical distances are guaranteed to bound the distance range for the beam as the sensor position varies within the tile. After this, the bounds can be extended according to the noise scale to get the bounding boxes.

E Additional Training Details

This section presents the additional details of the training of the neural networks in the case studies. We use Adam optimizer with learning rate 0.01. We use early stopping based on the loss on the validation set: we terminate training if the validation performance does not improve in 5 consecutive epochs. We take the model from the epoch that has the lowest loss on the validation set.