Formal Methods for Machine Learning

Extension of “Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks”

Authors: Szymon Gołebiowski [6317952], Alex Lalov [5289254]
Supervisor: Anna Lukina

Q1 2025–26

1 Introduction

Formally verifying Convolutional Neural Networks (CNNs) is a difficult task because of their high-dimensional hidden states and unique architectures which make most popular verification methods obsolete either due to their time complexity or general incompatibility. Within Formal Methods for Machine Learning, this exposes a clear gap: we lack tractable, certification-grade analyses of targeted misclassification for CNNs, even on controlled benchmarks. We aim to address this gap by translating a CNN with kernels into an equivalent linear/ReLU network consumable by SMT-style verifiers. This allows us to leverage a mature solver such as Reluplex [2] without redesigning the verification backend for image classification models; we solely need to correctly interface with the Reluplex engine to define what margin-based input/output constraints must be preserved. Due to computation limitations, we focused on one-dimensional data. Concretely, we ask: For a CNN on MNIST-1D dataset [1], what are certified ϵ bounds for targeted misclassification between class pairs, and what perturbation structures emerge? Answering this question yields actionable robustness certificates per source-target class pair, insight into which classes are intrinsically confusable under bounded perturbations, and evidence about how convolutional structure interacts with solver performance.

All code and implementations referenced in this report are available on GitHub.

2 Background

Reluplex extends the simplex method to a theory of linear real arithmetic with ReLU constraints, allowing the solver to keep ReLU inputs/outputs temporarily inconsistent and repair them via specialized pivot/update rules, resorting to splitting-on-demand only when necessary. The implementation adds bound tightening, lightweight conflict analysis/backjumping, and a floating-point “tableau restoration” check for numerical stability. The paper evaluates this on the ACAS Xu collision-avoidance benchmark: an array of 45 fully connected ReLU networks (each 6 hidden layers/=300 ReLUs) derived by discretizing two state variables so each network has five inputs and five advisory outputs (COC, weak/strong left/right). Reluplex proves functional properties about advisory consistency and demonstrates local robustness (ϵ-ball queries) on selected points, outperforming general SMT/LP baselines on these networks.

ACAS Xu networks operate on low-dimensional, tabular state variables with decision outputs scored for advisories; Reluplex's properties are phrased as score inequalities over bounded input regions. In contrast, our work targets image-like 1-D signals (MNIST-1D) and targeted misclassification between class pairs under a specified norm-ball, which is a different objective and data regime. Because Reluplex natively handles fully connected ReLU nets, we translate a CNN into an equivalent linear/ReLU network so an SMT-style solver can be applied without redesigning the backend, then encode margin-based classification constraints and sweep an ϵ-grid (for tractability) to obtain certified ϵ bounds and perturbation structure across source to target pairs. Reluplex's core contribution is the ReLU-aware simplex engine validated on ACAS Xu; our contribution is adapting that machinery to convolutional image models and targeted robustness certificates via an exact CNN to MLP translation and evaluation protocol suited to classification.

3 Methodology

We aim to answer: how robust is our CNN to varying levels of perturbation magnitudes, and what is an explicit perturbation that can cause misclassification. To answer these questions we would need to define an ϵ-ball around a sample's inputs and attempt to find an adversarial input for which the maximal output of our network is no longer the ground truth class. To enable the original Reluplex [2] engine to verify image classification models we needed to: translate a trained PyTorch CNN into a MLP, and extend the authors' C++ code to correctly interface with the engine given our specific constraints.

The Reluplex verifier is implemented in C++ and accepts neural networks only in a custom file format `.nnet`. This format is designed specifically for fully-connected networks, and therefore supports only fully-connected layers with linear transformations followed by ReLU activations. Consequently, convolutional and pooling layers, as well as other typical CNN elements, are not natively supported. To enable verification of models containing such layers, a conversion pipeline was developed to transform the original architecture into an equivalent MLP that is compatible with the `.nnet` specification.

The crucial observation is that a convolutional layer is actually a special case of a fully-connected linear layer. We can think of it as if most of weights were set to 0 (kernels applied locally) and specific weights have equal values (kernels for one pair of input-output channels share weights).

We converted convolutional layers (`torch.nn.Conv1d` module) to fully-connected linear layers (`torch.nn.Linear` module) using the procedure showcased in Algorithm 1. It must be noted that multiple input channels are preserved, but rather than being treated as separate tensors in the second dimension, their values are concatenated sequentially in the flattened vector. The output of each layers is only a two-dimensional tensor of shape $[\,\text{batch\_size}, \text{channels} \cdot \text{input\_size}\,]$.

The process begins by calculating the expected spatial output length, L_{out}, derived from standard convolution arithmetic using the input length, kernel size, stride, and padding. Next, standard dense layer parameters are initialized with zeros: a weight matrix W and a bias vector b. These are sized to operate on completely flattened input and output tensors; specifically,

W has dimensions $(C_{out} \cdot L_{out}) \times (C_{in} \cdot L_{in})$. The algorithm then systematically maps the shared weights of the convolution kernel into explicit, redundant positions within this large matrix. For every output unit (uniquely identified by its channel *co* and spatial position *pos*), the algorithm first assigns the appropriate channel bias to the corresponding index in the flattened vector b. It then determines which input units contribute to this output by iterating through all input channels and kernel elements. The equivalent spatial input index is calculated as $in\_pos = pos \cdot S + k - P$. If this index falls within the valid data range (i.e., it is not just zero-padding), the specific kernel weight is copied into W at the precise row and column coordinates linking that input neuron to the current output neuron. The resulting fully-connected layer, despite being highly sparse, exactly reproduces the original sliding window computation when applied to a flattened input vector.

Layers that are already fully compatible with the `.nnet` format require no conversion. Fully-connected linear layers and ReLU activations can be used directly in the `.nnet` format. Similarly, flatten operations do not require any explicit transformation because, throughout the conversion process, data is already represented as a single flat vector with channels concatenated sequentially.

```
Data: A 1D convolutional layer with $C_{in}$ input channels, $C_{out}$ output channels, kernel
      size $K$, stride $S$, padding $P$, and input length $L_{in}$
Result: An equivalent fully-connected layer representing the same computation

$L_{out} \leftarrow \frac{L_{in}+2P-K}{S} + 1$;
Initialize weight matrix $W$ of size $(C_{out} \cdot L_{out}) \times (C_{in} \cdot L_{in})$ with zeros;
Initialize bias vector $b$ of size $(C_{out} \cdot L_{out})$ with zeros;
for each output channel co from $0$ to $C_{out} - 1$ do
    for each output position pos from $0$ to $L_{out} - 1$ do
        $b[co \cdot L_{out} + pos] \leftarrow$ bias of channel $co$;
        for each input channel ci from $0$ to $C_{in} - 1$ do
            for each kernel element k from $0$ to $K - 1$ do
                $in\_pos \leftarrow pos \cdot S + k - P$;
                if in_pos is within input range then
                    $W[co \cdot L_{out} + pos, ci \cdot L_{in} + in\_pos] \leftarrow$ corresponding kernel weight;
                end
            end
        end
    end
end
Construct a fully-connected layer with weights $W$ and bias $b$;
```

Algorithm 1: Conversion of a one-dimensional 1D convolutional layer to an equivalent fully connected layer.

Average pooling layers are also converted into linear layers because they perform a fixed average over a sliding window. For a kernel of size k, the pooled output at each position is simply the sum of k input elements multiplied by $1/k$. This corresponds exactly to a linear transformation with weights set uniformly to $1/k$ for the positions covered by the kernel, and zeros elsewhere. Biases are set to zero. This procedure is presented in Algorithm 2.

Similar to the convolution conversion, it begins by determining the output spatial dimension, L_{out}, based on the input length, kernel size, stride, and padding. A weight matrix W and bias vector b are initialized with zeros to accommodate the flattened input and output tensors. Because standard average pooling operates depthwise (independently on each channel), the weight matrix dimensions are $(C_{in} \cdot L_{out}) \times (C_{in} \cdot L_{in})$, ensuring no cross-channel interaction. The bias vector remains entirely zero, as average pooling is a purely linear operation without additive components. The algorithm iterates through every channel and spatial output position to define the connections for each neuron in the new dense layer. For a given output unit, it identifies the corresponding receptive field in the input using the formula $in\_pos = out\_pos \cdot S + k - P$. If an input position is valid (i.e., not part of the virtual padding), the connection weight in W is set to $1/K$. This fixed weight value ensures that when the fully-connected layer performs its dot product operation, it effectively sums the inputs in the window and divides by the kernel size, exactly replicating the average pooling computation.

```
Data: A 1D average pooling layer with $C_{in}$ input channels, kernel size $K$, stride $S$,
      padding $P$, and input length $L_{in}$
Result: An equivalent fully-connected layer representing the same computation

$L_{out} \leftarrow \frac{L_{in}+2P-K}{S} + 1$;
Initialize weight matrix $W$ of size $(C_{in} \cdot L_{out}) \times (C_{in} \cdot L_{in})$ with zeros;
Initialize bias vector $b$ of size $(C_{in} \cdot L_{out})$ with zeros;
for each input channel $c$ from $0$ to $C_{in} - 1$ do
    for each output position $out\_pos$ from $0$ to $L_{out} - 1$ do
        for each kernel element $k$ from $0$ to $K - 1$ do
            $in\_pos \leftarrow out\_pos \cdot S + k - P$;
            if $in\_pos$ is within input range then
                $W[c \cdot L_{out} + out\_pos,\ c \cdot L_{in} + in\_pos] \leftarrow 1/K$;
            end
        end
    end
end
Construct a fully-connected layer with weights $W$ and biases $b \leftarrow 0$;
```

Algorithm 2: Conversion of a one-dimensional average pooling layer to an equivalent fully connected layer.

Overall, these steps allow convolutional, average pooling, ReLU, and flattening operations to be accurately mapped to MLP-compatible layers. The resulting architecture preserves the

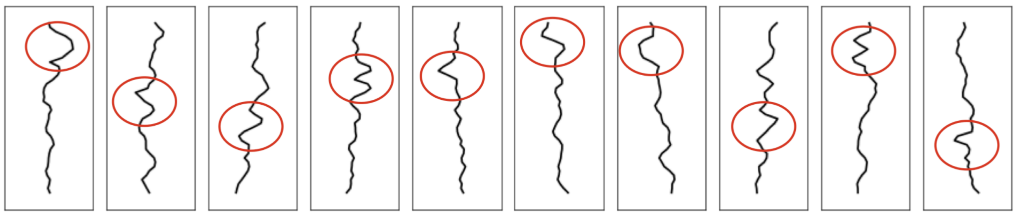


Figure 1: One-dimensional samples from the MNIST-1D dataset. The red circles highlight the patterns that determine the assignment to one of the ten classes.

Dataset	Logistic regression	MLP	CNN	GRU
MNIST	94 ± 0.5	> 99	> 99	> 99
MNIST-1D	32 ± 1	68 ± 2	94 ± 2	91 ± 2
MNIST-1D (shuffled)	32 ± 1	68 ± 2	56 ± 2	57 ± 2

Figure 2: Comparison of model accuracies on MNIST and MNIST-1D datasets.

numerical behaviour of the original network while satisfying the input constraints imposed by Reluplex. We validated the correctness of the implemented conversions with unit tests and ensured that converted models return exactly the same results on all samples from the test split.

4 Dataset

We run the evaluation on a a MNIST-1D dataset [1]. It is a one-dimensional, synthetic, fully parametric alternative of MNIST, where each sample is generated from predefined waveform templates rather than collected from real handwriting. Digits are represented as 1-dimensional signals (typically of length 40) constructed by sampling parameterized functions with controlled variations such as shifts, scaling, and noise, ensuring the entire data distribution is explicitly defined. Example samples of all 10 classes are presented in Figure 1.

Despite being lower-dimensional, MNIST-1D is actually more difficult to classify and requires models with more non-linearity than original MNIST digits. The comparison of model accuracies on both datasets is shown in Figure 2. MNIST-1D was created especially for the purpose of conducting experiments with computationally demanding pattern recognition methods, which is exactly our case. Although the dataset is one-dimensional and thus one-dimensional convolutions will be used, all results can be easily generalized to a typical, two-dimensional case.

MNIST-1D allows to generate a dataset of arbitrary complexity. Our goal was to conduct experiments on a dataset that can be classified by the model reasonably well but still poses some challenge. We set the dataset generation parameters (Table 1) so that the model would obtain the accuracy circa 80-90%. Example samples of all classes are presented in Figure **??**.

Parameter	**Value**
`num_samples`	`5000`
`train_split`	`0.8`
`template_len`	`12`
`padding`	`[36, 60]`
`scale_coeff`	`0.4`
`max_translation`	`48`
`corr_noise_scale`	`0.25`
`iid_noise_scale`	`0.02`
`shear_scale`	`0.75`

Table 1: MNIST-1D parameters used in experiments.

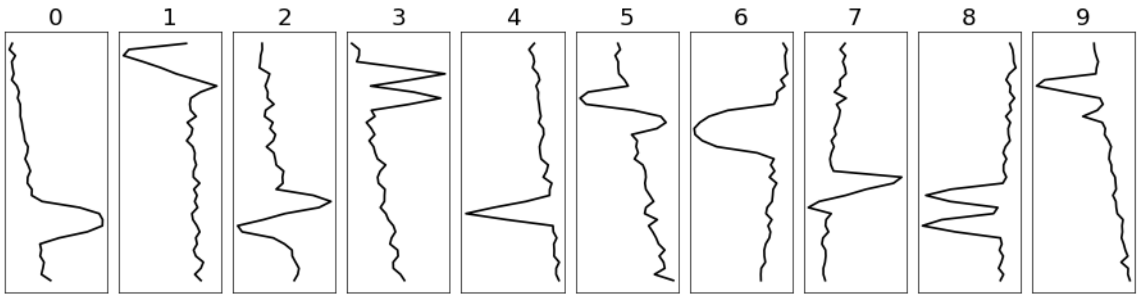


Figure 3: Perfect 1-Dimensional Digits from Dataset

5 Experiments

5.1 Setup

First, we extracted the model's most confident predictions for each digit within both the train and test datasets. Perturbing these *perfect* digits to cause a misclassification is theoretically more difficult and should display Reluplex's ability to uncover explicit adversarial inputs even in the best scenario. In Figure 3, we can see the digits for which the model's output confidence, after applying softmax, was at its highest. We can observe that these digits feature considerably less noise than most samples in the dataset, alongside being relatively legible.

Secondly, we created a set of ϵ values which would best display the difference between different digits' chances of misclassification, and applied each ϵ to the lower and upper boundaries of the slack variables defined as inputs to the Reluplex engine. The set consists of: $\{1.00, 0.75, 0.50, 0.25, 0.10\}$. We settled on that range as the engine was able to find a solution (adversarial input) for almost all inputs within the ± 1.0 bound. One such example can be seen in Figure 5a. We opted for a coarse general ϵ grid instead of performing a binary-search as provably UNSAT results for some ϵ values took over 10 minutes per verification. A further study could look at the potential technical improvement of using hardware acceleration (GPU) to parallelize the process of finding a tighter ϵ by running multiple runs with varying ϵ values.

Lastly, we encoded the conditions which would dictate whether or not an adversarial input exists within those bounds, by initializing static cells through one of Reluplex's object methods. The experiment focuses on misclassification from a ground truth class towards a specific other class i; rather than the general case of misclassification. We can encode this by guaranteeing that for any given runner-up i (the incorrect class we want to classify the digit as) the difference between its output neuron y_i and all other classes y_j is larger than or equal to 0, or more realistically 1e-6.

$$\forall i \neq j : \quad y_i - y_j \geq 10^{-6}$$

5.2 Evaluation

We ran the experiment with a one minute timeout on a trained CNN and, we were able to obtain the minimum values of ϵ for which misclassification occurs, with a mode uncertainty of 0.25 given that those are our steps. We used a compact 1-D convolutional neural network for sequences of length 40 with a single input channel. The first convolutional block applies a 1-D convolution with 10 filters (kernel size = 5, stride = 5, no padding), followed by ReLU, reducing the temporal dimension from 40 to 8 (feature map size: 10×8). The second block applies a 1-D convolution with 20 filters (kernel size = 8, stride = 8, no padding) and ReLU, reducing the temporal dimension from 8 to 1 (feature map size: 20×1). The output is flattened (dimension = 20) and passed to a fully connected layer with 10 outputs (logits). Unlike in the Reluplex ACAS Xu network, the maximal value among these logits determines the predicted class the 1-D image belongs to.

Figure 4 depicts a matrix which shows to what minimum degree one would need to perturb any of the digits before misclassification to the other digit occurs. Rows capture source fragility while columns reveal attractor targets. Consistent low-ϵ columns act as confusability sinks as they are easy to misclassify given a small fixed perturbation. Patterns where a row has one distinctly low ϵ for a given target and are dark red imply a dominant perturbation direction; rows with several moderate lows indicate greater diffuse vulnerability as many targets can be found.

Interestingly, the matrix is asymmetric along the diagonal therefore the difficulty in misclassifying any i class to any j class does not correlate to the difficulty in misclassifying any j class to any i class. This asymmetry implies that the model is not able to differentiate between all classes equally therefore some classes are easier to classify and others misclassify. Moreover, the asymmetric ϵ magnitudes across digit pairs suggest that the learned weights and biases preferentially shape decision boundaries to favor certain classes over others.

Both digits 4 and 5 seem to be the easiest to perturb as they have the most amount of runner-up digits which can be found within $\epsilon > 0.1$ of their inputs; whereas both digits 6 and 8 are distinct enough that they are the hardest to misclassify. The latter pair is particularly interesting since they are both most easily perturbed to become each other. This is justified by the fact that in the earlier Figure 3, there is a visible similarity between those two digits as both are generally left-leaning 'arches' with the only difference being that digit 8 has a centre point along the middle of its 'arch'.

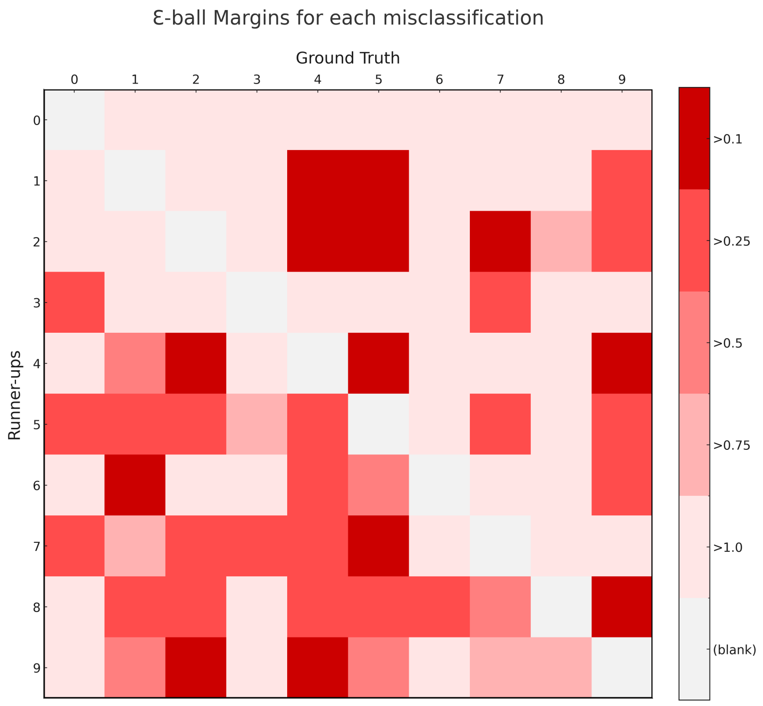


Figure 4: ϵ-ball margins for each misclassification

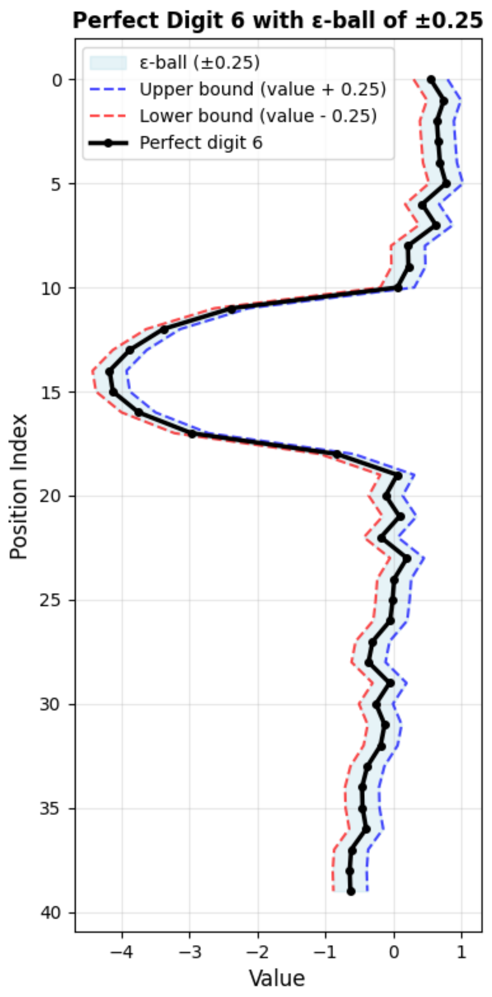
Looking into a specific perturbation, such as how the perfect digit 6 needs to be noised to get misclassified as a digit 8 we revealed an interesting insight. It seems the perturbation is mostly noise in the varying parts of the vector input rather than concrete, legible changes to the digit itself. This implies that the network might not have learned the important features of a digit 6 but rather an offset for which that class is most likely. This is further supported by the fact that the perturbation doesn't actually change the aforementioned 'arch' that visibly makes this one-dimensional vector a digit 6.

It is important to note that these insights would be incredibly difficult to verify for 2-dimensional images, but in this case because of the choice of dataset (1-D MNIST) and tool (Reluplex) we are able to uncover adversarial results through formal formulation proving the extent to which our model is not robust both qualitatively and quantitatively.

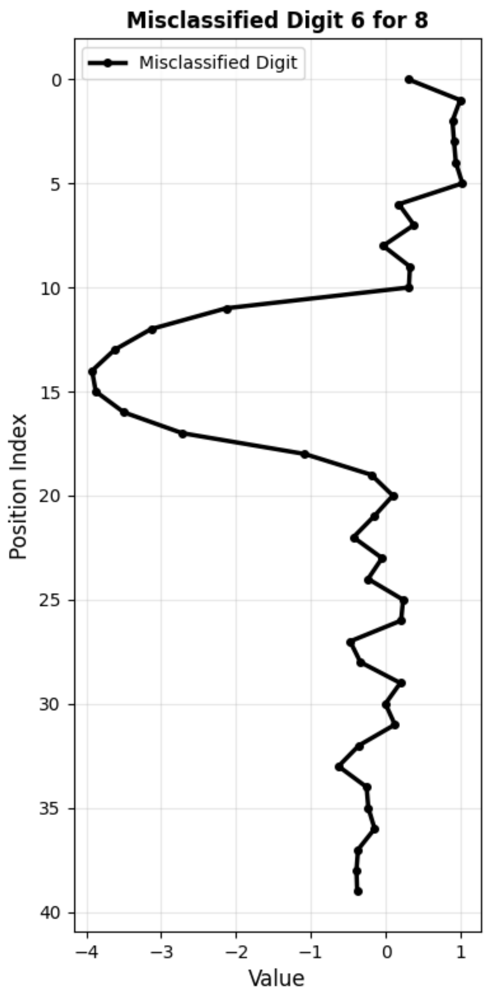
6 Conclusion

We demonstrated that a standard SMT-based verifier (Reluplex) can be applied to convolutional models by translating convolutions into an equivalent linear/ReLU network, preserving the network's function and predictions end-to-end. On MNIST-1D, this enabled certified ϵ-bounds for targeted misclassification across all class pairs. The resulting matrix reveals directional asymmetries ($i \to j \neq j \to i$), identifies fragile source classes (e.g., 4, 5) and attractor/robust targets (e.g., 6, 8), and provides concrete, per-pair certificates rather than heuristic robustness estimates. This illustrates how convolutional architectures interact with SMT solvers when cast into a fully connected form.

In terms of limitations, this report did not consider any other search strategies to narrow down ϵ values, alternative 2-D image datasets or CNNs with no piecewise-linear activation



(a) Perfect Digit 6 with ϵ-ball



(b) Misclassified Perfect Digit 6 as 8

Figure 5: Comparison of the original and misclassified versions of Digit 6

functions. As previously mentioned, creating a search algorithm would be simple from an implementation perspective but would require an excessive amount of time to execute given that Reluplex takes over 5 minutes to find certain perturbations. Extending the code to work with 2-D convolutions would greatly increase the space and time complexity of the problem and would require augmenting the Reluplex engine to handle parallel computations; which is not feasible given that the SMT Solver and all algorithms relating to pivoting and splitting ReLUs rely on the sequential execution of tableau commands. Lastly, if the CNN includes activation functions which are not piecewise-linear, one would not be able to encode them easily within the equations in the aforementioned tableau and a linear solver would simply not be able to resolve them.

All of the code and implementations referenced in this report are available on GitHub.

References

[1] S. Greydanus and D. Kobak. Scaling down deep learning with mnist-1d, 2024.

[2] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks, 2017.