FOOLING A COMPLETE NEURAL NETWORK VERIFIER

Anonymous authors
Paper under double-blind review

ABSTRACT

The efficient and accurate characterization of the robustness of neural networks to input perturbation is an important open problem. Many approaches exist including heuristic and exact (or complete) methods. Complete methods are expensive but their mathematical formulation guarantees that they provide exact robustness metrics. However, this guarantee is valid only if we assume that the verified network applies arbitrary-precision arithmetic and the verifier is reliable. In practice, however, both the networks and the verifiers apply limited-precision floating point arithmetic. In this paper, we show that numerical roundoff errors can be exploited to craft adversarial networks, in which the actual robustness and the robustness computed by a state-of-the-art complete verifier radically differ. We also show that such adversarial networks can be used to insert a backdoor into any network in such a way that the backdoor is completely missed by the verifier. The attack is easy to detect in its naive form but, as we show, the adversarial network can be transformed to make its detection less trivial. We offer a simple defense against our particular attack based on adding a very small perturbation to the network weights. However, our conjecture is that other numerical attacks are possible, and exact verification has to take into account all the details of the computation executed by the verified networks, which makes the problem significantly harder.

1 INTRODUCTION

In their seminal work, Szegedy et al. found that for a given neural network and input example one can always find a very small adversarial input perturbation that results in an incorrect output (Szegedy et al., 2014). This striking discovery motivated a substantial amount of research. In this area, an important research direction is *verification*, that is, the characterization of the robustness of a given network in a principled manner. A usual way of defining the verification problem involves the specification of an input domain and a property that should hold over the entire domain. For example, we might require that all the points within a certain distance from an input example share the same output label as the example itself. The verification problem is then to prove or disprove the property over the domain for a given network (Bunel et al., 2020).

There are a large number of verifiers offering different types of guarantees about their output. Complete verifiers offer the strongest guarantee: they are able to decide whether a given property holds in any given input domain. For example, the verifier of Tjeng et al. is a state-of-the-art complete verifier that we will focus on in this paper (Tjeng et al., 2019). However, it is currently standard practice to ignore the details of the computation that the network under investigation performs, such as the floating point representation or the order in which input signals are summed, and so on.

In this paper, we claim that such implicit assumptions make verifiers vulnerable to a new kind of attack where the attacker designs a network that fools the verifier, exploiting the differences between how the verifier models the computation and how the computation actually occurs in the network. Such differences include, for example, the order in which certain operations are executed causing a different roundoff error. These errors might in turn cause arbitrary differences between the verified and the actual behavior, as we will show.

Our focus on nuanced numerical details might seem purely academic, normally such issues are not at the forefront of neural network research. The reason is that the weights are robust to small perturbations due to stochastic learning algorithms (Bottou, 2010), and due to regularizers such as standard variants of weight decay and dropout (Srivastava et al., 2014). Due to this robustness, low

precision arithmetic can be applied as well (Courbariaux et al., 2015; Gupta et al., 2015). Networks do sometimes produce numerical errors (e.g., Inf or NaN values), most often due to the non-linear operations within the loss function (Odena et al., 2019) or divergence during training, but overall, numeric precision is not a key practical concern.

However, our results indicate that, when it comes to exact methods for verification, numerical issues become a central problem that can cause arbitrary errors and enable backdoors. Thus, it is important to close this vulnerability if we wish to apply verifiers to certify real-world safety-critical networks.

Our contributions are the following. In Section 3, we introduce a simple adversarial network that misleads the verifier of Tjeng et al. (2019). In Section 4, we show how to hide the large weights that are present in the simple network. In Section 5, we describe a way to utilize such adversarial networks to add a backdoor to an existing network. Finally, in Section 6 we offer a defense against the attack we presented.

2 BACKGROUND

Let us first formulate the verification problem, that is, the problem of checking whether a given property holds in a given domain. We adopt the notation used in (Tjeng et al., 2019). For a possible input x, let $\mathcal{G}(x)$ denote the set of inputs that are considered similar to x in the sense that we expect all the points in $\mathcal{G}(x)$ to get the same label as x. The set $\mathcal{G}(x)$ is normally defined as a ball around x in some metric space defined by a suitable vector norm. The input domain we need to consider is given as $\mathcal{G}(x) \cap \mathcal{X}_{valid}$ where $\mathcal{X}_{valid}$ denotes the valid input points. For example, we have $\mathcal{X}_{valid} = [0, 1]^m$ if the input is an image of m pixels with each pixel taking values from the interval $[0, 1]$.

We now have to formulate the property that we want to hold in this domain. Informally, we want all the points in the domain $\mathcal{G}(x) \cap \mathcal{X}_{valid}$ to get the same classification label as x. Let $\lambda(x)$ denote the label of x and let $f(x; \theta) : \mathbb{R}^m \to \mathbb{R}^n$ denote the neural network, parameterized by θ. This network has n outputs classifying each input x into n classes. The label of x is given by $\lambda(x) = \arg\max_i f(x; \theta)_i$. Using this notation, the property we want to hold for an input $x' \in (\mathcal{G}(x) \cap \mathcal{X}_{valid})$ is that $\lambda(x') = \lambda(x)$.

Putting it together, the verification problem can be expressed as deciding the feasibility of the constraint

$$x' \in (\mathcal{G}(x) \cap \mathcal{X}_{valid}) \wedge (\lambda(x') \neq \lambda(x)) \tag{1}$$

with x' as our variable. If this constraint is feasible then there is an x' that violates the property. If it is infeasible then (provided $\mathcal{G}(x) \cap \mathcal{X}_{valid}$ is not empty) there is no such x'.

2.1 APPROACHES TO VERIFICATION

There are many approaches to tackle this problem. We can, for example, search for a suitable x' in the given domain using some heuristic optimization methods (Goodfellow et al., 2015; Moosavi-Dezfooli et al., 2016; Kurakin et al., 2017; Carlini & Wagner, 2017). If the search succeeds, we can decide that equation 1 is feasible. Otherwise we cannot decide.

Other methods attempt to find a proof for the infeasibility of equation 1, however, they do not guarantee such a proof. Examples include (Wong & Kolter, 2018; Weng et al., 2018; Gehr et al., 2018; Raghunathan et al., 2018). If a proof is found, we can decide that equation 1 is infeasible. Otherwise we cannot decide. Such methods are sometimes called *incomplete* (Tjeng et al., 2019; Bunel et al., 2020).

The strongest guarantee is given by methods that are able to decide the feasibility of equation 1. Examples for such methods include (Katz et al., 2017; Cheng et al., 2017; Dutta et al., 2018; Tjeng et al., 2019). These methods are sometimes called *complete* (Tjeng et al., 2019; Bunel et al., 2020).

Note that some approaches work with a model of the neural network (e.g., linear programming formulations, as we will see) and other approaches use the network itself (e.g., heuristic optimization methods). All the verifiers that work with a model of the network are potentially vulnerable. This is because a model of the network might be inaccurate in that it might not take into account the limited numerical precision of the network, which in fact renders an actual network non-linear, or it might

ignore other details such as the exact order of certain computations. Such inaccuracies of a model, however minor, can in turn be used to attack the verifier, as we will show.

2.2 MIPVERIFY

In this paper, we are focusing on a state-of-the-art complete verifier: MIPVerify (Tjeng et al., 2019). It is based on a mixed integer linear programming (MILP) formulation. As long as the domain $\mathcal{G}(x) \cap \mathcal{X}_{valid}$ is the union of a set of polyhedra, and the neural network $f(x, \theta)$ is a piecewise linear function of x with parameters θ, the problem of checking the feasibility of the constraint in equation 1 can be formulated as a MILP instance.

$\mathcal{G}(x)$ is normally defined as a ball in a suitable norm with x as the center. In ℓ_∞ or ℓ_1 norms $\mathcal{G}(x)$ is thus a cube. Also, $\mathcal{X}_{valid}$ is normally a box or a set of boxes, so the domain is indeed the union of a set of polyhedra. The neural network is piecewise linear as long as the nonlinearities used are ReLUs (note that the last softmax normalization layer adds no extra information and can thus be ignored). For the details of the MILP formalization, please refer to (Tjeng et al., 2019).

Importantly, MIPVerify applies a presolve step that greatly increases its efficiency. In this step, the authors attempt to tighten the bounds on the variables of the problem, including on the inputs to each ReLU computation. If in this step it turns out that the input of a ReLU gate is always non-positive, the output can be fixed as a constant zero, and if the input is always non-negative then the ReLU gate can be removed from the model as it will have no effect.

The presolve step applies three approaches in a progressive manner. First, a fast but inaccurate interval arithmetic approach is used. The resulting bounds are further improved by solving a relaxed LP problem on every variable. Finally, the full MILP problem is solved for the variables but with early stopping.

2.3 FLOATING POINT REPRESENTATION

Floating point real number representations are successful and efficient tools for most real life applications (Muller et al., 2010). This arithmetic is available on most modern computers via sophisticated hardware implementations. A floating point number is represented as $s \cdot b^e$, where s is the signed *significand*, b is the *base* and e is the *exponent*. There are numerous standards to implement the exact details of this idea that differ mainly in the number of bits the significand and the exponent use. The formula to compute the represented real number has several possible variations as well.

Here, we will use the double precision (binary64) arithmetic defined by the IEEE 754-1985 standard (IEEE, 1985). There, $b = 2$ and we have a sign bit, an 11 bit exponent, and a 53 bit significand (with 52 bits stored explicitly). The so called machine epsilon (the maximum relative rounding error) is $2^{-53} \approx 1.11e{-}16$. This means that, for example, the computation $10^{20} + 2020 - 10^{20}$ will result in zero in this representation, if executed in the specified order. In our attack, we will exploit roundoff errors of this type. Note, that in the order of $10^{20} - 10^{20} + 2020$ we obtain the correct result of 2020.

3 A SIMPLE ADVERSARIAL NETWORK

In this section we present our attack in its simplest possible form. We describe an adversarial network that results in incorrect output when given to MIPVerify. The main idea is to exploit the fact that the addition of numbers of different magnitude can become imprecise in floating point arithmetic due to roundoff errors, as described previously in Section 2.

Our attack crucially depends on the order in which the inputs and the bias are added within a unit. We assume that the bias is always the last to be added, when computing the forward pass. Note that the creator of the network who submits it for verification can control the execution of the network in actual applications, since the verification is only about the structure but not the minute details of execution such as the order of addition in each unit. Nevertheless, we also conjecture that any fixed order of addition, or indeed any fixed algorithm for determining the order could similarly be exploited in an attack.

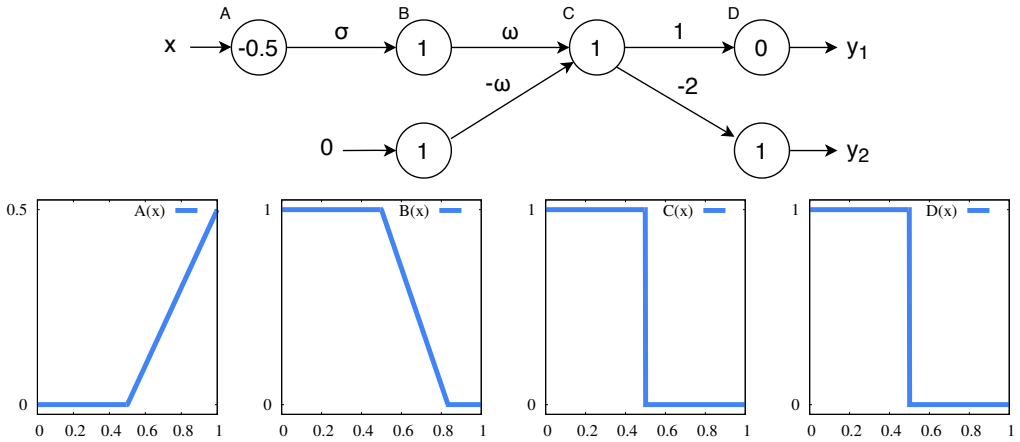

Figure 1: A naive adversarial network. The neurons (represented by circles) are ReLUs, the numbers in the circles represent the additive bias, the numbers over the connection represent the weight. The valid input range is assumed to be $x \in [0, 1]$. Parameter σ defines the steepness of the transition region in $B(x)$, while ω is the "large weight" parameter. The output of four different neurons is shown as a function of x, assuming $\sigma = -3$ and $\omega = 10^{17}$.

In approaches such as MIPVerify, which rely on state-of-the-art commercial solvers like Gurobi (Gurobi, 2020), the mapping of the actual computation—such as the order of addition—to computations performed by the solver is non-trivial and hard to control, as it is defined by the many (typically proprietary, hence black box) heuristics and techniques applied while solving the MILP problem.

The simplest form of our adversarial network is shown in Figure 1. This network performs a binary classification over its input $x \in [0, 1]$. By construction, we know that $y_1 \in [0, 1]$. Since MIPVerify expects multi-class models and thus two outputs, we add another technical output y_2, such that the two classes are defined by $y_1 < y_2$ and $y_1 \geq y_2$, respectively. Also, we include a neuron with a constant input of zero.

The key element of the network is neuron C (Figure 1). The idea is that the maximal value of $C(x)$ is given by $\omega - \omega + 1$. The computation of this value might lead to a roundoff error if ω is too large *and* if 1 is not the last addend. For example, when using 64 bit floating point representation, if $\omega > 2^{53}$ (recall that 2^{-53} is the machine epsilon) then a roundoff error is possible. In case of a roundoff error $\omega + 1 - \omega$ is computed to be zero, that is, $C(x) = 0, x \in [0, 1]$. This means that we get the incorrect output $y_1(x) = 0, x \in [0, 1]$. In other words, the entire input domain appears to belong to the $y_2 > y_1$ class. The roundoff error thus masks the fact that there are input points that in reality belong to the other class; this property will be used later on to add a backdoor to an existing network.

The role of σ is more subtle. It defines the steepness of the transition region of $B(x)$. We should set σ so that the range of $B(x)$ is the interval $[0, 1]$. This means that we need to have $\sigma < -2$.

We repeat here, that if the constant 1 is added as the last addend then the roundoff error can be avoided; however, when solving the related MILP, this is not guaranteed. Indeed, we evaluated MIPVerifiy experimentally using two commercial solvers: Gurobi (Gurobi, 2020), and CPLEX (CPLEX, 2020), and the open source GLPK (GLPK, 2020). During these evaluations, we experimented with different values of σ and ω to see whether our adversarial networks fool the MIPVerify approach. We randomly generated 500 values for σ from the interval $[-15, -2]$ and for all the sampled σ values we tested ω values $2^{54}, 2^{55}, \ldots, 2^{70}$. For each parameter setting we tested whether the input point $x = 0.75$ has an adversarial example within a radius of 1. Recall, that the valid input range is $x \in [0, 1]$ so in fact we evaluated the problem over the entire valid input range.

Clearly, the correct answer is yes, that is, the constraint in equation 1 is feasible. Yet, we found that all three solvers found the problem infeasible for all the parameter combinations we tried. That is, our simple adversarial network rather reliably fools the MIPVerify model.

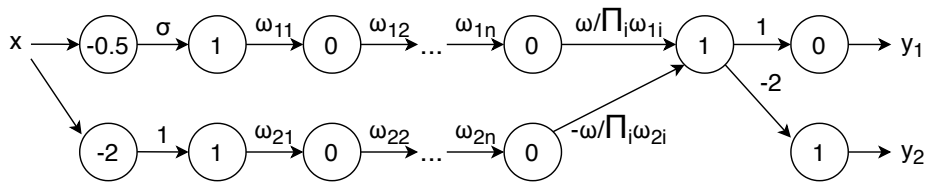

Figure 2: An adversarial network without extremely large weights. The network is equivalent to that in Figure 1, only parameter ω is distributed over n layers, and the neuron with constant zero input is made less obvious with the help of an additional neuron with constant zero output over the valid input domain.

Table 1: Percentage of successful attacks when $\omega = 2^{54}$.

$\sigma \in$	$[-15, -10]$	$[-10, -6]$	$[-6, -4]$	$[-4, -3]$	$[-3, -2]$
$n = 20$	97.4	97.5	98.6	92.3	96.0
$n = 50$	97.4	97.3	95.9	97.3	98.7

4 Obfuscating the Network

The naive network in Figure 1 works as an attack, however it is painfully obvious to spot, since it has very large weights, which is highly unusual, and it also has a neuron with constant zero input, which is also suspicious. Here, we argue that the network can be made to look more "normal" relatively easily. Obviously, this problem might also lead to an arms race between obfuscators and detectors, and it is not our goal to offer an undetectable obfuscation. We simply wish to argue that detection can be made harder if an attacker puts in some effort to hide the attack. Here, we focus on making the weight distribution more realistic.

Figure 2 illustrates the construction. We wire the neuron that requires a constant zero input to a neuron that has a constant zero output over the input domain, due to the application of the rectifier nonlinearity. We then add n extra layers and spread the weights ω and $-\omega$ over these layers effectively creating two large products $\prod_i \omega_{1i}$ and $\prod_i \omega_{2i}$. The last weight on both the top and bottom paths is set so that the entire product is ω and $-\omega$, respectively. Each weight ω_{ji} is drawn from the normal distribution $\omega_{ji} \sim \mathcal{N}(\omega^{\frac{1}{n}}, (\omega^{\frac{1}{n}}/4)^2)$ independently, but negative values are redrawn until a positive value is found.

We evaluated these networks as well, using Gurobi, GLPK and CPLEX. The values for parameter n (the number of layers) we tested were $n = 20$ and $n = 50$. In both cases, we experimented with the same values of σ and ω as in the case of the simple network, and followed the same evaluation methodology. We generated the network weights ω_{ij} based on the algorithm above, independently for all the pairs of σ and ω parameters. The algorithm was successful in generating a good network (that is, when the top and bottom paths multiply to the same weights in absolute value) for the first try in at least 70% of the cases, so it is not particularly difficult to generate a suitable network: we can simply try a different random seed until the algorithm is successful.

We found that all three solvers were fooled, that is, they all found the problem infeasible for all the parameter combinations we tried, when $\omega > 2^{54}$. For the value of $\omega = 2^{54}$ the verifier found the problem feasible for a small percentage of the networks, meaning that in these cases we failed to fool the verifier. Table 1 contains the percentage of the successful adversarial networks in different ranges of σ. It is interesting to note that, for a given problem, all the three solvers we tested always output the same result, that is, all three of them agree whether the problem is feasible or infeasible. This means that the success of the attack is robust to the solver used, and depends mainly on the problem definition.

All in all, we conclude that distributing the weights over many links does not prevent us from reliably fooling the MIPVerify model. The values of all the weights ω_{ij} are positive. One could also add negative weights if the desired mean weight is zero. Such links could point to "garbage" neurons

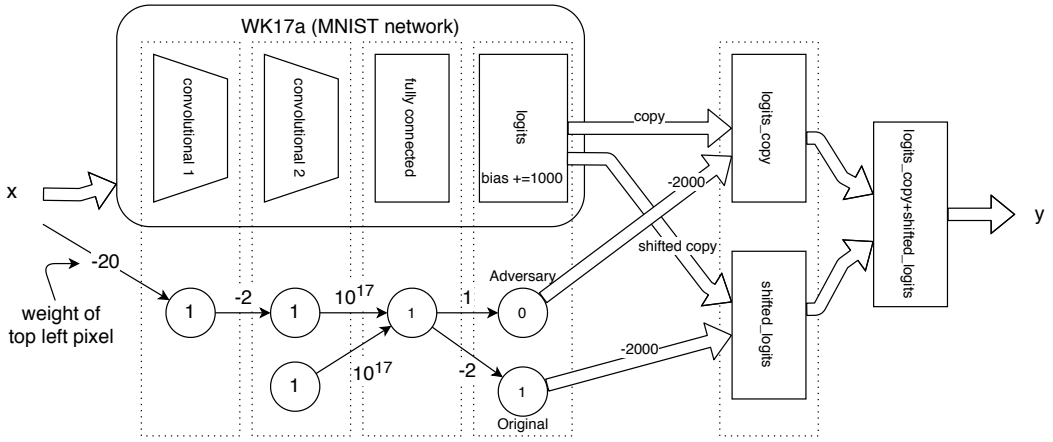

Figure 3: Diagram of the MNIST network extended with a backdoor that is activated by the top left pixel of the input image x. If this pixel is larger than 0.05 then the backdoor is activated and shifts the predicted labels. The output y is the predicted distribution of the MNIST labels. Thick empty arrows signify multiple connections. See text for a detailed explanation.

that have no effect on the output by design. Besides, when using this network as a backdoor to some relatively larger legitimate network (see Section 5), these weights are mixed with realistic weights.

5 Creating Backdoors

We demonstrate that the adversarial network we described can be used to extend a non-trivial network with a backdoor, so that the extended network passes verification exactly as the original network, however, in practice it will have a backdoor (invisible to verification) that can be used to trigger arbitrary behavior.

For our evaluation, we selected an MNIST classifier described in Wong & Kolter (2018) and used in (Tjeng et al., 2019) to evaluate MIPverify. The network has two convolutional layers followed by a fully connected layer, with ReLU activations. The network was trained to be robust to attacks of radius 0.1 in the ℓ_∞ norm by the method of Wong and Kolter.

The backdoor construction is illustrated in Figure 3. The basic idea is that we insert the adversarial network as a switch that is triggered by a pattern in the input. Since the adversarial network is invisible to verification, the backdoor will pass the verification undetected. Here, the backdoor is activated whenever the top left pixel is larger than 0.05. Note that in the MNIST dataset the top left pixel is always 0. Thus, when running verification on any input with an ℓ_∞ radius of 0.1, the network will in fact be guaranteed to be vulnerable to at least one adversarial perturbation vector, namely the one that activates the backdoor.

Adding our adversarial network to a convolutional network requires the creation of an additional filter for each neuron in the corresponding layer. The output of every such neuron can then be recovered from the computed new feature map and can be used in the next layers. The neuron with the two large input weights is already in the fully connected layer, so it can simply be added to that layer. This neuron is wired to just two pixels in the two feature maps in the previous convolutional layer that correspond to the outputs of its two input neurons.

We empirically determined that the absolute value of the signal in each unit is always less than 1000 in the logits layer of the MNIST network before the application of the ReLU activation. The logits layer of the MNIST network is modified by increasing the bias in each unit by 1000. This constant is large enough to ensure that all the values become positive. This further means that the ReLU function will not alter these values. We then add the two output neurons of the adversarial network to the logits layer.

The next layer is a new one, in which we copy the original logits layer, as well as the same layer but shifted by one index. The adversarial network is then used to suppress either the original copy or the

Table 2: Test accuracy of the WK17a MNIST network with perturbed parameters.

ϵ	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-9}	0
original	0.9788	0.9811	0.9810	0.9811	0.9811	0.9811	0.9811	0.9811
with backdoor	0.1118	0.3744	0.9725	0.98105	0.98105	0.9811	0.9811	0.9811

shifted copy. If the backdoor is activated then the original copy will be suppressed, otherwise the shifted one. Suppression is implemented by subtracting the value 2000 from all the inputs so that they become negative. This way, the output will be zero due to the ReLU activation.

In the final layer, the original and shifted halves of the previous layer are summed. Since one of these halves have been suppressed, the last layer will be equal to either the original layer or the shifted layer. We then apply softmax normalization to obtain the predicted label distribution y. This distribution will equal the one predicted by the original MNIST network if the backdoor is not active, otherwise it will be shifted, so that if the original prediction was i then the shifted prediction is $i + 1 \mod 10$. Note that the backdoor could trigger an arbitrary behavior, shifting is used here as an ad hoc example.

We verified this extended network with MIPVerify using Gurobi as our solver over the test set of the MNIST dataset, using a radius of 0.1 in ℓ_∞ norm. The verification result was identical to that reported in (Tjeng et al., 2019), namely 4.38% adversarial error, as if no backdoor had been present. The correct verification result should have been 100% adversarial error because, by design, the backdoor mechanism is fully functional in the actual network, and the backdoor pattern is always well within the radius of 0.1 from any example.

6 A DEFENSE

A naive idea for a defense could be to use a precision that is higher than that of the network while solving the optimization problem. That might indeed work in this case, however, it would open another similar attack, namely the network could count on the fact that it will commit a certain roundoff error while the verifier will not. Using combinations of different precisions is also an option but here, instead of attempting to predict the outcome of such an arms race, we assume that both the network and the optimizer use the same double precision internal representation.

The defense we propose here consists of adding a very small perturbation to the parameters of the network. More precisely, we replace every parameter w with $\hat{w} \sim w + \mathcal{U}(-|w|\epsilon, |w|\epsilon)$, where ϵ is the relative scale parameter of the uniform noise term. The key insight is that some of the parameters of our adversarial network are rather sensitive to noise whereas non-adversarial networks are naturally robust to a very small perturbation of their parameters.

Indeed, we tested the sensitivity of the MNIST network we studied in Section 5. We perturbed the parameters of both the standalone MNIST network and the one extended with a backdoor (see Figure 3), using various values of ϵ and we measured the test accuracy of the networks. The results are shown in Table 2 (the results are averages of 10 independent perturbed networks). Although the network with a backdoor is somewhat less robust, for a small noise such as $\epsilon = 10^{-9}$ the behavior of the network under normal circumstances remains unaltered. Note that the test examples do not activate the backdoor.

We also verified a perturbed network without the backdoor with MIPVerify, with $e = 10^{-9}$, and the results are identical to that of the unperturbed network for every single test example, yielding 4.38% adversarial error. Thus, in this network, even adversarial robustness is robust to small weight perturbation not only test accuracy.

Although the network with the backdoor appears to be robust to noise, what we are after is being able to detect the backdoor using MIPVerify and that becomes possible due to the noise. This is because the small perturbation of the parameters will alter the small adversarial network's computations in such a way that they will no longer depend on the order of addition, hence roundoff errors will no longer play a role. This is true both for the simple and the obfuscated versions in Figures 1 and 2, respectively.

To understand this, consider that in order for the attack to work, the two large input weights of neuron C (Figure 1) have to be of identical magnitude with opposite signs. Due to the small noise, with high probability, these two large weights will have a difference that is several orders of magnitude larger than the bias of neuron C. Thus, over the range $x < 0.5$, the sum of the weighted inputs will either be a large negative value or a large positive value. This means that the order in which the bias is added has no significance, a roundoff error will not alter the output of neuron C. The same reasoning applies to the obfuscated network as well. A detailed derivation of this is included in Section B in the Appendix.

It also follows from the reasoning above that the small perturbation of the parameters can have two possible effects on the adversarial networks. On the one hand, if neuron C has a negative output for the entire input range $x \in [0, 1]$ then its output will be constant zero. If such a network is used as the switch for the backdoor then the backdoor will stay off permanently, independently of the input pattern. On the other hand, if neuron C has a large positive output for the range $x < 0.5$ then the behavior of the network remains intact, however, the verifier will detect this behavior. These two different outcomes of the perturbation have an approximately equal probability, since the outcome largely depends only on the sign of the difference of the perturbed input weights of neuron C (or the perturbed versions of the two products $\prod_i \hat{\omega}_{1i}$ and $\prod_i \hat{\omega}_{2i}$ in the case of the obfuscated network).

We verified the simple as well as the obfuscated networks, with $\epsilon = 10^{-3}, 10^{-6}$ and 10^{-9}, using the same methodology as in the previous sections, with the same hyperparameters σ, ω and n, as before. We tested only those networks where the original behavior remained intact (which happens in approximately 50% of the cases, as we argued before). With Gurobi, the verification revealed the true behavior in 100% of the cases for all our settings, thus the defense indeed worked.

Finally, we verified the MNIST network with the backdoor added, perturbed with $\epsilon = 10^{-9}$, using Gurobi. We selected a perturbation in which the backdoor switch remained functional. This time, the result of the verification successfully revealed the backdoor for 98.3% of the test examples. Since such a perturbation has a probability of only about 50%, it might be necessary to repeat the verification with independently sampled perturbations which allows one to increase this probability to a desired level. Alternatively, the approval might be assigned to the perturbed network, as opposed to the original network. This way, if the perturbation turns the backdoor off permanently (and thus the verification does not find problems) it is still an acceptable outcome.

7 Conclusions

We proposed an attack against a complete verifier, MIPVerify (Tjeng et al., 2019). The idea was that we exploited a floating point roundoff error that was made by all the optimizers we tested. The attack allowed us to modify any given network by adding a backdoor that enables triggering arbitrary behavior using a specified pattern in the input. This backdoor was completely missed by the verification.

Although we did offer a defense for the particular attack we presented, we believe that our work still implies that for a reliable verification, a verifier must take into account all the details of the implementation of the network. This includes the details of the representation of the numeric parameters as well as the order of the operations. Otherwise, potentially exploitable differences in the actual computation and the model are guaranteed to exist. This way, though, the verification would be valid only for a specific implementation. The implementation of a network can also be non-deterministic. For example, a parallel hierarchical implementation of addition can result in an exponential number of different actual executions of the same addition, depending on the specifics of the hardware the network is running on. In this case, the verifier must make sure that its output is valid for every possible groupings and orderings of each operation performed during a forward pass of the network.

The attack we proposed is rather straightforward, just like the defense. However, without the defense, the attack can completely alter the behavior of any network undetected. This means that it is important to keep the numerical vulnerability of verification methods in mind, and further research is needed to find solutions that explicitly prevent such numeric attacks.

REFERENCES

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta (eds.), *Proceedings of COMPSTAT'2010*, pp. 177–186, Heidelberg, 2010. Physica-Verlag HD. doi: 10.1007/978-3-7908-2604-3_16.

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H.S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(42):1–39, 2020.

Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pp. 39–57, 2017. doi: 10.1109/SP.2017.49.

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In Deepak D'Souza and K. Narayan Kumar (eds.), *Automated Technology for Verification and Analysis*, pp. 251–268, Cham, 2017. Springer International Publishing. doi: 10.1007/978-3-319-68167-2_18.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR*, 2015. Workshop Track.

CPLEX. Cplex optimizer, 2020. URL `https://www.ibm.com/analytics/cplex-optimizer`.

Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In Aaron Dutle, César A. Muñoz, and Anthony Narkawicz (eds.), *NASA Formal Methods – 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pp. 121–138. Springer, 2018. doi: 10.1007/978-3-319-77935-5_9.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pp. 3–18. IEEE Computer Society, 2018. doi: 10.1109/SP.2018.00058.

GLPK. Gnu linear programming kit, 2020. URL `https://www.gnu.org/software/glpk/`.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR*, 2015. Workshop Track.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1737–1746, Lille, France, 07–09 Jul 2015.

Gurobi. Gurobi optimizer, 2020. URL `http://www.gurobi.com/`.

IEEE. 754-1985 - IEEE standard for binary floating-point arithmetic, 1985.

Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak (eds.), *Computer Aided Verification*, pp. 97–117, Cham, 2017. Springer International Publishing. doi: 10.1007/978-3-319-63387-9_5.

Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. In *5th International Conference on Learning Representations, ICLR*, 2017.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *The IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 2574–2582, June 2016.

Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. doi: 10.1007/978-0-8176-4705-6.

Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 4901–4911, Long Beach, California, USA, 09–15 Jun 2019.

Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *International Conference on Learning Representations*, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1): 1929–1958, January 2014.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR*, 2014.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR*, 2019.

Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for ReLU networks. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5276–5285, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018.

Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5286–5295, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018.

A SPECIFICATION OF OUR EXPERIMENTAL ENVIRONMENT

Since our work depends on the internals of commercial solvers, for reproducibility, we give the full specification of the environment that we used:

- CPU: Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz
- Operating System: Ubuntu 18.04.4 LTS
- GLIBC 2.27
- Julia version 1.5.0
- Gurobi: Gurobi Optimizer version 9.0.2 build v9.0.2rc0 (linux64)
- Gurobi julia package: Gurobi v0.8.1
- CPLEX: IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 12.10.0.0
- CPLEX julia package: CPLEX v0.6.6
- GLPK v4.64
- GLPK julia package: GLPK v0.13.0, GLPKMathProgInterface v0.5.0
- MIPVerify julia package: MIPVerify v0.2.3
- JuMP julia package: JuMP v0.18.6, ConditionalJuMP v0.1.0
- MathProgBase julia package: MathProgBase v0.7.8

The implementation of the experiments is not yet shared due to the double blind review.

B ANALYSIS OF THE DEFENSE PERTURBATION

Let us consider the simple network in Figure 1. The defense consists of adding a small perturbation to the parameters of the network with uniform distribution. More precisely, we replace every parameter w with $\hat{w} \sim w + \mathcal{U}(-|w|\epsilon, |w|\epsilon)$, where ϵ is the relative scale parameter of the uniform noise term.

We will assume that $\epsilon < 1$, which means that adding noise to a weight will never change the sign of the weight. In practice, ϵ is intended to be very small, for example, for double precision we used $\epsilon = 10^{-9}$. From the construction, we also know that $x \in [0, 1]$ and $\sigma < -2$. For simplification, we will assume here that $\omega > 0$ although that is not strictly necessary. Note that, in practice, we set $\omega > 2^{54}$ for attacking double precision floating point arithmetic, because smaller values do not guarantee adversariality.

For a given input x, the output of every neuron is now a random variable depending on the random perturbation. From the definition of neuron A, however, we know that for every $x \leq (1-\epsilon)/2$ we have $A(x) = 0$. Formally, we have $Pr(A(x) = 0 | x \leq (1-\epsilon)/2) = 1$. For this reason, the distribution of the output of every neuron is independent of x, if $x \leq (1-\epsilon)/2$, because the output of each neuron depends on x only through neuron A. This means that it suffices to study $x = 0$ to describe the distribution of the output in this interval.

From now on, any variable u_i will denote a random variable with the distribution $u_i \sim \mathcal{U}(-\epsilon, +\epsilon)$. We also assume that each variable u_i is drawn independently. We have seen that $A(0) = 0$. From this, it follows that $B(0) \in [1-\epsilon, 1+\epsilon]$ because $B(0) = 1 + u_0$. Let us now examine the input function of C, f_C, that we can derive from Figure 1. We have $f_C(0) = \omega(1+u_1)B(0) - \omega(1+u_2)(1+u3) + (1+u_4) = \omega(1+u_1)(1+u_0) - \omega(1+u_2)(1+u_3) + (1+u_4)$. From this, we can compute lower and upper bounds that give us the range of $f_C(0)$:

$$f_C(0) \in [-4\omega\epsilon + 1 - \epsilon, 4\omega\epsilon + 1 + \epsilon]. \tag{2}$$

Within this interval, the distribution of $f_C(0)$ is symmetrical to the center of the interval, that is, 1. This further means that we have $Pr(f_C(0) < 1) = 1/2$. However, the probability mass of $f_C(0)$ is very small in the interval $[0, 1]$ for typical parameter settings, because $\omega\epsilon$ is several orders larger than 1. For example, when $\epsilon = 10^{-9}$ and $\omega = 2^{54}$, we have $\omega\epsilon = 2^{54} \cdot 10^{-9} \approx 1.8e7$. So, we have $Pr(f_C(0) < 0) \approx 1/2$.

If $f_C(0) < 0$ then it is easy to see that $f_C(x) < 0$, $x \in [0, 1]$. This is because $f_C(0)$ is an upper bound of $f_C(x)$, which follows from the fact that our only input x has an effect only through a linear chain of neurons all of which will thus have a monotonous output. However, if $f_C(x) < 0$ then $C(x) = 0$ due to the ReLU activation, which means that $y_1(x) = 0$, thus $y_1(x) < y_2(x)$ for all $x \in [0, 1]$.

Now, let us consider the case when $f_C(0) > 0$ and thus $C(0) = f_C(0)$. In this case, we have $y_1(0) < y_2(0)$ if and only if $C(0)(1 + u_5) < -2C(0)(1 + u_6) + 1 + u_7$. Here, we know that $C(0)(1-\epsilon) < C(0)(1+u_5)$ and $-2C(0)(1+u_6)+1+u_7 < -2C(0)(1-\epsilon)+1+\epsilon$. From this, it can be derived that if $C(0) > 1$ then we must have $\epsilon > 1/2$ to have $y_1(0) < y_2(0)$ with a probability larger than zero. Since typical values of epsilon will be much smaller than $1/2$, we can conclude that if $C(0) > 1$ then $y_1(0) > y_2(0)$. Previously, we have shown that $Pr(C(0) > 1) = 1/2$. In the interval $C(0) \in [0, 1]$ the output depends on the value of epsilon as well as on the actual values of u_5, u_6 and u_7. However, as mentioned before, the probability of a perturbation that results in $C(0)$ ending up in this interval is negligible.

Let us now consider the case when $x > (1 - \epsilon)/2$. Looking at Figure 1, we can see that the transition interval where $C(x)$ decreases from 1 to 0 is very short. We will show that, in fact, it is shorter than the machine epsilon, so practically $C(0)$ is a step function. When adding noise, this transition interval becomes somewhat longer (that is, when $C(0) > 0$, because otherwise there is no transition at all) and will be in the order of ϵ at most. To see this, we will need an upper bound on f_C and derive the point where it reaches zero. We know that

$$f_C(x) \leq \omega(1+\epsilon)B(x) - \omega(1-\epsilon)(1-\epsilon) + (1+\epsilon), \tag{3}$$

$$B(x) \leq \sigma(1-\epsilon)A(x) + (1+\epsilon), \text{ and} \tag{4}$$

$$A(x) \geq x - \frac{1}{2}(1+\epsilon). \tag{5}$$

Note that we need a lower bound on $A(x)$ because $\sigma < 0$. Now, we need to substitute the bounds on $A(x)$ and $B(x)$ to the bound on $f_C(x)$ and find x, for which this bound is zero. This gives

$$x = \frac{1}{2}(1+\epsilon) + \frac{1}{\sigma}\left(\frac{1-\epsilon}{1+\epsilon} - \frac{1+\epsilon}{1-\epsilon} - \frac{1}{\omega(1-\epsilon)}\right) \leq \frac{1}{2} + \frac{2\epsilon}{1-\epsilon^2} + \frac{1}{2\omega(1-\epsilon)}, \tag{6}$$

where we used that $\sigma \leq -2$. This bound on x is very close to $1/2$. In fact, without perturbation (that is, with $\epsilon = 0$), the offset is just $1/2\omega$ which is less than the machine epsilon, for our settings of ω. Since ϵ^2 and $1/2\omega$ are negligibly small, we can approximate the bound as $1/2 + 2\epsilon$. Since $f_C(x)$ is monotone decreasing, this means that $f_C(x) < 0$ for $x \in [1/2 + 2\epsilon, 1]$. This further implies that $C(x) = 0$ for $x \in [1/2 + 2\epsilon, 1]$, and thus $y_1(x) < y_2(x)$ over this interval.

To sum up, we proved that if $x \in [0, 1/2 - \epsilon/2]$ then with at least 50% probability we have $y_1(x) \geq y_2(x)$ and with almost 50% probability we have $y_1(x) < y_2(x)$, and if $x \in [1/2 + 2\epsilon, 1]$ then we always have $y_1(x) < y_2(x)$. When $y_1(x) \geq y_2(x)$, the value of $C(x)$ is large enough with overwhelming probability for reasonable parameter settings (e.g., $\epsilon = 10^{-9}$, $\omega = 2^{54}$) so that roundoff errors will not occur. The interval $x \in [1/2 - \epsilon/2, 1/2 + 2\epsilon]$ was not discussed; here, the outcome depends on the actual noise values and the other parameters, however, this is an extremely short interval of length 2.5ϵ.

As a final note, the network in Figure 2 can be treated in a very similar fashion, the only difference is that the noise that is effectively added to ω and $-\omega$ will follow a different distribution. Focusing on ω ($-\omega$ is very similar), the noisy product has the form $\prod_i(\omega_{1i} + u_{1i})$. The effective *absolute* noise added to ω will be more similar to a normal distribution as it is mainly defined by the sum of the first order noise terms: $\sum_{i=1}^{n} u_{ji} \prod_{k \neq i} \omega_{jk}$. Thus, the expectation is zero and the variance grows with $\epsilon\omega^{\frac{n-1}{n}}\sqrt{n}$. The effective *relative* noise is thus increased by a factor of $\sqrt{n}$, approximately. So, $Pr(0 < C(x) < 1)$ is still very small, and the range of $f_C(x)$ is larger, so our arguments about the simple case transfer to the obfuscated case as well. The upper bound on the length of the transition interval will be somewhat larger due to the larger variance but it will still be very small.