Lecture 12: Certified Defenses I

November 7, 2019

In Lecture 9 we saw a number of settings where adversarial attacks exist against neural networks, and in Lecture 10 we saw a number of proposed empirical defenses against these attacks. With one notable exception—adverarial training—these defenses have been broken. To ensure this doesn't happen, an alternative line of work has focused on defenses with *provable robustness*, that is, one can, given a model f and new test point (X, y), the defense can (hopefully) prove that $f(X') = f(X) = y$ for all X' in the allowed perturbation set. These defenses typically split into two categories, with further sub-categories amongst them:

- **Exact certification.** Exact certification techniques try to truly answer the question of whether or not there exists an adversarial perturbation to f at (X, y). In other words, their output is either YES, in which case there does not exist an adversarial perturbation, or NO, in which case there does exist an adversarial perturbation.
- **Relaxed certification.** Relaxed certification techniques allow for an alternative output: MAYBE, which the certification algorithm is always guaranteed to output. However, if it outputs YES, then it must be true (perhaps with some caveats we'll talk about later) that there does not exist any adversarial perturbations of X. And of course, the algorithm should not output MAYBE or NO too often.

In this lecture, we'll primarily focus on exact certification, near the end we'll also discuss some of the techniques for relaxed certification. In particular, we will (mostly) assume we are given a neural network, and our only job will be to certify whether or not it is robust. However, certification techniques often need to be paired with an companion training algorithm, that encourages robustness; after all, if your algorithm is not robust, then it certainly cannot be certified as such. This will be less of a problem for exact certification techniques. This is because it seems that adversarial training is in fact truly robust (at least, to some extent), and so if you have an exact certificate, you can just certify that an adversarially trained model is robust. On the other hand, for relaxed certification techniques we will often need to design training algorithms that complement the certification procedure.

We'll primarily focus on deep, feed-forward ReLU networks in this class (and in the next couple of classes), as they are the most common form of neural network, and moreover these methods, as we'll see, are naturally well equipped to deal with the combinatorial structure of ReLU networks. Formally, our networks will defined by a sequence of functions (the layers)

$f_1, \ldots, f_\ell$ so that $f_i(x) = \sigma(W_i x + b_i)$, where W_i is a weight matrix (or tensor) the b_i are a set of biases, and σ is the entrywise ReLU operation, i.e. $\sigma(x)_i = \max(x_i, 0)$. The overall network representation will be given by $F : \mathbb{R}^d \to \mathbb{R}^m$ by

$$F(x) = W_{\ell+1} f_\ell \circ f_{\ell-1} \circ \ldots \circ f_1(x) \ ,$$

which gives produces the logits of the final representation. We will usually ignore issues such as max-pooling, batch normalization, skip connections, etc. Some of these are naturally incorporated into these frameworks; others are not.

1 Exact certification

This is perhaps the dream goal for certified defenses, and indeed, for any defense overall: provably demonstrate whether or not the network is robust at a point X. The main difficulty with exact certification is that for neural networks, the problem is provably hard:

Theorem 1.1 ([1])**.** *Exact certification of a neural network with ReLU activations at a point (X, y) and for the perturbation set $\mathcal{P}_{\infty,\varepsilon}(X)$ is* coNP*-complete.*

So that's some bad news. However, it is not the end of the story: as SAT-solvers become increasingly powerful, one could hope that by leveraging some of these efficient algorithms for NP-complete problems, one could still manage to verify real neural networks, on real datasets. Unfortunately, we are not really at this point, however, also not as far away as one might naively expect.

Exact certification techniques usually take one of two forms. Either, they are based on satisfiability modulo theories (SMT) [1], or mixed integer-linear programming (MILP) [2]. In this lecture we'll primarily focus on the latter, following presentation from [3, 4], because it provides the state of the art numbers, and requires less background in SAT solvers and proof complexity theory (that the author doesn't possess) to understand.

2 Mixed Integer Linear Programming

We first define what MILPs are. First, recall that *linear programs* (LPs) can be written in the following generic form (although the exact form won't be super important to us later on):

$$\begin{aligned} \text{maximize} \quad & c^\top x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \ . \end{aligned}$$

On the other hand, *integer programs* (IPs) are problems where, in addition to the constraints above, we add the constraint $x \in \mathbb{Z}$, that is x is integer-valued. Finally, a *mixed integer linear program* (MILP) is a program where some of the x variables are constrained to be

integers, and others are not. LPs can be solved in polynomial time by methods such as the ellipsoid method or cutting plane methods, heuristically by simplex methods, and also by other convex optimization based methods. IPs, on the other hand, are one of Karp's 21 NP-complete problems. As MILPs contain IPs as a special case, they are also NP-hard (in fact complete). However, there do exist solvers for MILPs which sometimes work in practice.

2.1 Encoding Robust Certifiability as a MILP

We now turn our attention to encoding robust certifiability as MILP.

Encoding ℓ_p constraints First, notice that ℓ_∞ perturbations can easily be captured as an LP by adding an additional variable, as

$$\|x - x'\|_\infty = \min \varepsilon \qquad \text{s.t.} \qquad \varepsilon \geq x_i - x'_i \text{ and } \varepsilon \geq x'_i - x_i \qquad \text{for all } i \ .$$

Similarly ℓ_1 perturbations in $\mathbb{R}^d$ can be captured as an LP by adding d auxiliary variables. Notably ℓ_p constraints for $p \notin \{1, \infty\}$ cannot be captured as a MILP. However, ℓ_2 constraints can be captured by mixed integer quadratic programming.

Encoding ReLU constraints The ℓ_∞ constraints are not so bad; after all, ℓ_p norms for $p \geq 1$ are convex and so it's not surprising that encoding ℓ_∞ doesn't need actual integer constraints. The main difficulty will arise in attempting to encode the ReLU constraints; after all, this is where the non-linearity comes it. Suppose that we have a univariate variable x, and we want to encode that $y = \sigma(x)$ where σ is the ReLU function. Notice that the ReLU can in fact be expressed as a convex minimization problem, as after all:

$$\sigma(x) = \min y \qquad \text{s.t. } y \geq x \text{ and } y \geq 0 \ .$$

However, we want to encode the value of y as a variable within an overarching MILP, that is, we want y to be defined by a set of inequalities and equalities on x. This will allow us to use y as a variable in the MILP, however, this becomes a bit more complicated. One naive approach, which doesn't quite work, is to introduce an additional variable $a \in \{0, 1\}$, and enforce:

$$a = \mathbf{1}[x \geq 0] \text{ and } y = ax \ .$$

The reason why this naive approach doesn't work is twofold: first, the constraint on a is not naturally enforceable as an MILP, and secondly, the constraint on y has degree 2. To get around these issues, observe that if we have an lower and upper bounds on x, that is, we know that $x \in [\ell, u]$, then we can in fact encode y by a set of linear and integer constraints:

Lemma 2.1. *Suppose that $x \in [\ell, u]$. Then we have that*

$$y = \sigma(x) \leftrightarrow (y \leq x - \ell(1 - a)) \wedge (y \geq x) \wedge (y \leq u \cdot a) \wedge (y \geq 0) \wedge (a \in \{0, 1\}) \ , \qquad (1)$$

and moreover $a = \mathbf{1}[x \geq 0]$ (where take the convention that a can be either 0 or 1 if $x = 0$).

Proof. The proof proceeds by case analysis. Suppose that $a = 0$. Then the last two constraints in (1) are tight, so $y = 0$. The second constraint in (1) implies that $y \geq x$, which implies that $x \leq 0$. Moreover, the first constraint is loose, as $x \geq \ell$ by assumption. Hence this system is satisfiable, and $y = \sigma(x)$ and $a = \mathbf{1}[x \geq 0]$ in this case. On the other hand, suppose that $a = 1$. Then the first two constraints imply that $y = x$. The third constraint is loose, and the last constraint implies that $x \geq 0$, and again the Lemma holds. □

Notice that crucially this construction requires an upper and lower bound on how large x can be; as we shall see, this will be a recurring theme as well in the convex relaxation based approaches.

We are almost done: by applying these relationships entrywise for every non-linearity of the neural network (and using linear relationships to encode the linear transformations), we can encode all the transformations up until the last layer. To complete the description of the MILP instance, we need to be able to take the maximum of the logits at the last layer. This can be done via a similar process to the Lemma above, and we leave this to the homework.

2.2 ReLU stability

Intuitively, the difficulty with solving the MILP will come from the binary variables; in our case, the a variables. Evaluating these equations and searching over possible choices of a will be the bottleneck for most applications of the MILP solver. Naively, we will introduce one binary variable for every non-linearity. However, one can do some optimizations to improve this. If we can reduce this number from say 10000 to 100, we should expect a huge improvement in runtime.

One trivial but important observation is that if both u and ℓ are positive or negative, the problem becomes trivial, and indeed we can remove the a: if they are both positive then we can just set $y = x$. We call this case *stably active*. If they are both negative, we call this *stably inactive*, and we can set $y = 0$. In either case, we have removed the need for the integer variable here, which is a big win from a runtime perspective. The remaining case, when $\ell < 0 < u$, is the tricky case, and we call this the *unstable* case.

The fraction of ReLU activations which are stable will depend on what ℓ, u we can obtain. In practice, we cannot typically get tight bounds on ℓ, u (except at the first layer), and so we will need to do some relaxation to compute the best bounds on ℓ, u that we can achieve. Moreover, these bounds will typically depend on how good of a bound on ℓ, u we could obtain the previous layers. Thus it's really important to try to get the tightest bounds on ℓ, u that we can. The most basic way of doing this is called *interval arithmetic*, which computes these bounds in a layerwise way. For any $i = 1, \ldots, d$, let ℓ_i and u_i be vectors so that ℓ_{ij} is a lower bound for the jth activation, and similarly assume u_{ij} is an upper bound for it First, at level 0, if the perturbation set is an ℓ_1 or ℓ_∞ ball, the u_0 and ℓ_0 upper and lower bounds are trivial to compute. Then, at level i, we can compute valid bounds on these quantities as follows. Let our weight matrix be W_i. Let $W^+ = \max(W, 0)$ and $W^- = \min(W, 0)$. Then, if

we define ℓ_{i+1} and u_{i+1} by

$$u_{i+1} = W^+ u_i + W^- \ell_i + b_i \ell_{i+1} = W^+ \ell_i + W^- u_i + b_i \ , \tag{2}$$

then it can be verified that u_{i+1} and ℓ_{i+1} still are valid upper and lower bounds on how large the variables at level $i+1$ can be. Note that these bounds are not tight; for the correctness of the MILP all we need is that they are valid bounds. But the looseness of these bounds can cause us to mark certain ReLUs as unstable when in actuality, if we used the tightest possible bounds, they are stable. But the main looseness from this comes from the fact that when we have stably inactive or stably active activations at degree i, we can actually get more information about what the values at the next iteration can be; for instance, if it's stably inactive we can simply remove that variable from contributing to either of these products, since we know that in actuality the variable at this level must be zero. Building on this idea, the paper [4] introduces a technique called *improved interval arithmetic*, which allows them to get somewhat tighter bounds.

However, it could be that even with the tightest bounds, most ReLUs are simply not stable: that there exist perturbations with bounded ℓ_∞ so that when we track the changes all the way to the current non-linearity, the value of the variable could be either negative or positive. In this case, not even the tightest computation of the bounds will help. To try to avoid this case, or at least mitigate its effects as much as possible, [4] also proposes to combine exact certification via this method with a *training* routine to obtain a neural network that has few unstable ReLUs. Specifically, if one adds the upper and lower bounds as variables into the training optimization, then one could hope to regularize for ReLU stability. For instance, one could add the regularization term $R(u_{ij}, \ell_{ij}) = 1 - \operatorname{sgn}(u_{ij}) \cdot \ell_{ij}$ to the optimization objective: then the optimization is penalized for having unstable ReLUs. However, this objective is clearly highly non-differentiable, and so instead they propose using the objective

$$R(u_{ij}, \ell_{ij}) = -\tanh\left(1 + \operatorname{sgn}(u_{ij}) \cdot \ell_{ij}\right) \ ,$$

which has a similar behavior (small if their signs agree, and large otherwise), and which is differentiable. They then add this to the training procedure, and by doing so, decrease the number of unstable ReLUs that the learned classifier typically has. As a result, they are able to certify much faster, by a factor of $10\times$ [4]. This of course says nothing about the quality of the solutions, however, by combining this with adversarial training, it appears that one gets fairly decent numbers.

Other optimizations There are some other optimizations that have been tried. For instance, if the weight matrix W is sparse, then MILP solvers tend to run faster, as there are fewer constraints propagating through the system. Thus, one can also train to encourage sparsity of the weight matrix, and this also tends to help runtime as well.

Limitations Despite these optimizations, MILP is still severely limited by the runtime, and can only run on relatively small models for e.g. MNIST and CIFAR, which cannot

obtain state of the art robust accuracy. Moreover, larger problems such as ImageNet seem completely out of the current realm of possibility.

References

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

[2] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[3] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[4] Kai Xiao, Vincent Tjeng, Nur Muhammad Shafiullah, and Aleksander Madry. Training for faster adversarial robustness verification via inducing re {LU} stability. In *International Conference on Learning Representations*, number 2019, 2019.