Appendix to: Abstract Interpretation of ReLU Neural Networks with Optimizable Polynomial Relaxations

Philipp Kern[1][0000−0002−7618−7401] and Carsten Sinz[1,2][0000−0001−9718−1802]

[1] Institute for Theoretical Computer Science, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
philipp.kern@kit.edu, carsten.sinz@kit.edu
[2] Karlsruhe University of Applied Sciences, Moltkestraße 30, 76133 Karlsruhe, Germany
carsten.sinz@h-ka.de

Abstract. Neural networks have shown to be highly successful in a wide range of applications. However, due to their black box behavior, their applicability can be restricted in safety-critical environments, and additional verification techniques are required. Many state-of-the-art verification approaches use abstract interpretation based on linear overapproximation of the activation functions. Linearly approximating non-linear activation functions clearly incurs loss of precision. One way to overcome this limitation is the utilization of polynomial approximations. A second way shown to improve the obtained bounds is to optimize the slope of the linear relaxations. Combining these insights, we propose a method to enable similar parameter optimization for polynomial relaxations. Given arbitrary values for a polynomial's monomial coefficients, we can obtain valid polynomial overapproximations by appropriate upward or downward shifts. Since any value of monomial coefficients can be used to obtain valid overapproximations in that way, we use gradient-based methods to optimize the choice of the monomial coefficients. Our evaluation on verifying robustness against adversarial patches on the MNIST and CIFAR10 benchmarks shows that we can verify more instances and achieve tighter bounds than state of the art bound propagation methods.

Keywords: Neural Network Verification · Abstract Interpretation · Polynomial Overapproximation.

A Representation of Polynomials

The efficiency of SIP with polynomial bounds is highly dependent on an efficient implementation of addition of polynomials, multiplication of a polynomial by constant matrices, as well as elementwise application of univariate polynomials and fast, but precise calculation of concrete lower and upper bounds. Therefore, we utilize the polynomial representation developed for sparse polynomial

zonotopes [2]. In this setting, the tuple

$$p(\mathbf{x}) = \langle G, E, \mathbf{id} \rangle \ , \tag{1}$$

where $G \in \mathbb{R}^{n \times m}, E \in \mathbb{N}^{d \times m}$ and $\mathbf{id} \in \mathbb{N}^d$, represents a multidimensional polynomial $p : \mathbb{R}^d \to \mathbb{R}^n$

$$p(\mathbf{x}) = \sum_{i=1}^{m} \left(\prod_{k=1}^{d} x_k^{E_{ki}} \right) G_i \tag{2}$$

with m monomial terms. In the remainder of this paper, we sometimes just write *polynomial* instead of *multidimensional polynomial* where it is clear from the context. The monomial coefficients for the i-th monomial are stored in the i-th column G_i of the generator matrix and the corresponding powers of the d variables are stored in the i-th column E_i of the exponent matrix. An identifier for the k-th variable is stored in id_k. For example, the tuple

$$\left\langle \begin{pmatrix} 5 & 4 & 3 & 0 \\ 2 & 3 & 0 & -7 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 2 & 0 \\ 0 & 1 & 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\rangle \tag{3}$$

is a representation of the polynomial

$$p(x_1, x_2) = \begin{pmatrix} 5 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 3 \end{pmatrix} x_1 x_2 + \begin{pmatrix} 3 \\ 0 \end{pmatrix} x_1^2 + \begin{pmatrix} 0 \\ -7 \end{pmatrix} x_2^2 \ . \tag{4}$$

Given polynomials $p : \mathbb{R}^d \to \mathbb{R}^n, p(\mathbf{x}) = \langle G_p, E_p, \mathbf{id} \rangle$ and $q : \mathbb{R}^d \to \mathbb{R}^n, q(\mathbf{x}) = \langle G_q, E_q, \mathbf{id} \rangle$ depending on the same input variables $\mathbf{x}$ and a matrix $A \in \mathbb{R}^{t \times n}$, a representation of the *addition* of two polynomials is given as

$$p(\mathbf{x}) + q(\mathbf{x}) = \langle \hat{G}, \hat{E}, \mathbf{id} \rangle \quad \hat{G} = \begin{bmatrix} G_p \ G_q \end{bmatrix}, \ \hat{E} = \begin{bmatrix} E_p \ E_q \end{bmatrix} \ . \tag{5}$$

Duplicate monomials – i.e. duplicate columns in $\hat{E} \in \mathbb{R}^{d \times m'}$ – can be summarized by only keeping one of these columns along with the sum of the corresponding columns in the new generator matrix $\hat{G}$. This requires scanning the m' columns of the new exponent matrix and comparing the d exponents in each column, which can be done in expected time of $O(m'd)$ using a hash map. Overall, the operation has an expected time complexity of $O(m'(n+d))$. If both polynomials share the same exponent matrix (i.e. $E_p = E_q$), there are no redundant monomials and the expression reduces to

$$p(\mathbf{x}) + q(\mathbf{x}) = \langle G_p + G_q, E, \mathbf{id} \rangle \ , \tag{6}$$

thus saving the cost of summarizing duplicate monomials.

The *linear map* also admits an efficient representation by multiplication with the generator matrix G_p:

$$A \ p(\mathbf{x}) = \langle A \ G_p, E_p, \mathbf{id} \rangle \tag{7}$$

For ease of notation, we now let $p(\mathbf{x}) = \langle G, E, \mathbf{id}\rangle$ and additionally consider a polynomial $f(\mathbf{y}) : \mathbb{R}^n \to \mathbb{R}^n$, where all $f_i(x) = \sum_{j=0}^{k} a_{ij}x^j$ are *univariate* polynomials, each of which can be described by a vector $(a_{i0}, a_{i1}, \ldots, a_{ik})$ of monomial coefficients. The *component-wise polynomial map* is defined as the polynomial $r(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^n$ where $r_i(\mathbf{x}) = f_i(p_i(\mathbf{x}))$, $\forall i \in [n]$. We can then write $f_i(p_i(\mathbf{x})) = \sum_{j=0}^{k} a_{ij} p_i(\mathbf{x})^j$. Since addition of polynomials was already introduced above, we only need to consider the operation of raising a polynomial to a given power j. To this end, we extend the result by [3] for $j = 2$ to arbitrary powers $j \in \mathbb{N}_+$:

$$p(\mathbf{x})^j = \left\langle \hat{G}, \hat{E}, \mathbf{id} \right\rangle \ , \tag{8}$$

with

$$\begin{aligned} \hat{G} &= \left[\hat{G}_1\ \hat{G}_2 \ldots \hat{G}_{\binom{j+m-1}{m-1}}\right] & \hat{E} &= \left[\hat{E}_1\ \hat{E}_2 \ldots \hat{E}_{\binom{j+m-1}{m-1}}\right] \\ \hat{G}_i &= \tbinom{j}{\alpha_1, \alpha_2, \ldots, \alpha_m} \prod_{l=1}^{m} G_l^{\alpha_l} & \hat{E}_i &= \sum_{l=1}^{m} \alpha_l E_l \ , \end{aligned}$$

where $\binom{j}{\alpha_1, \alpha_2, \ldots, \alpha_m}$ is the multinomial coefficient and the index $1 \le i \le \binom{j+m-1}{m-1}$ enumerates the possible combinations of $\alpha_1 + \alpha_2 + \cdots + \alpha_m = j$ such that $\boldsymbol{\alpha} \ge 0$. Note that this number is in $\Theta(m^j)$ for fixed $j \in \mathbb{N}$.

The validity of this result can be seen from expansion of $p(\mathbf{x})^j$ using the multinomial theorem:

$$p_i(\mathbf{x})^j = \left(\sum_{l=1}^{m} \left(\prod_{k=1}^{d} x_k^{E_{kl}} \right) G_{il} \right)^j \tag{9}$$

$$= \sum_{\substack{\alpha_1+\alpha_2+\cdots+\alpha_m=j \\ \boldsymbol{\alpha}\ge 0}} \binom{j}{\alpha_1, \alpha_2, \ldots, \alpha_m} \prod_{l=1}^{m} \left(\left(\prod_{k=1}^{d} x_k^{E_{kl}} \right)^{\alpha_l} G_{il}^{\alpha_l} \right) \tag{10}$$

$$= \sum_{\substack{\alpha_1+\alpha_2+\cdots+\alpha_m=j \\ \boldsymbol{\alpha}\ge 0}} \binom{j}{\alpha_1, \alpha_2, \ldots, \alpha_m} \prod_{l=1}^{m} \left(x_k^{\left(\sum_{l=1}^{m} \alpha_l E_{kl}\right)} \right) \prod_{l=1}^{m} G_{il}^{\alpha_l} \ . \tag{11}$$

Straightforward implementation of this formula would require $\binom{j+m-1}{m-1}$ additions of the m columns of $E \in \mathbb{N}^{d\times m}$ and the same amount of multiplications of the m columns of $G \in \mathbb{R}^{n\times m}$. However, for $j \le m$ at most j elements of $\boldsymbol{\alpha}$ can be non-zero at the same time. Therefore, the operation requires $O(\binom{j+m-1}{m-1} j(d+n))$ computations. For fixed $j \in \mathbb{N}$, this is equivalent to $O(m^j(d+n))$.

Propagation of a polynomial $p(\mathbf{x})$ through an elementwise polynomial map $r(\mathbf{x})$ of degree j requires the propagation through the linear part and the polynomial parts of $r(\mathbf{x})$ and then summarizing duplicated monomials. Computation of each $p(\mathbf{x})^k$ for $k = 2, \ldots, j$ uses $O(m^k(d+n))$ operations and produces $O(m^k)$ monomials. Propagation through the linear part requires $O(mn)$ operations and produces $O(m)$ monomials. During this process $O(m^j)$ monomials are created, whose summarization requires $O(m^j(n+d))$ operations in expectation. The expected total number of operations required is therefore in $O(m^j(n+d))$.

Concrete lower and upper bounds $\underline{p}$ and $\overline{p}$ of a polynomial $p(\mathbf{x}) = \langle G, E, \mathbf{id} \rangle$ can be efficiently computed via interval arithmetic. To further increase the efficiency and also the tightness of the bounds calculation, we first normalize the input variables to $\mathbf{x} \in [-1, 1]^d$ [1]. and then obtain:

$$\underline{p} = G^+ \underline{E} + G^- \overline{E} \qquad \overline{p} = G^+ \overline{E} + G^- \underline{E} \tag{12}$$

with

$$\underline{E} = \begin{pmatrix} \underline{e_1} \\ \vdots \\ \underline{e_m} \end{pmatrix} \quad \underline{e_i} = \begin{cases} 1, & E_{ki} = 0, \ \forall k \text{ (constant monomial)} \\ 0, & E_{ki} \mod 2 = 0, \ \forall k \\ -1, & \text{otherwise} \end{cases} \quad \overline{E} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} .$$

Note that normalization only has to be done once before propagating the input set through the network.

B Proof for Time Complexity of Polynomial Backsubstitution

For convenience, we restate Proposition 1:

Proposition 1.

Proof. In order to analyze the time complexity of Algorithm 2, we step through the operations one by one. To normalize the inputs to the NN to the range of $[-1, 1]$, we substitute $x_i = 1/2(\overline{x}_i - \underline{x}_i)\tilde{x}_i + 1/2(\underline{x}_i + \overline{x}_i)$. Merging this substitution with the weights and biases of the first layer, we get

$$\tilde{W}_1 \tilde{\mathbf{x}} + \tilde{\mathbf{b}}_1 = \frac{1}{2} W_1 \left((\overline{\mathbf{x}} - \underline{\mathbf{x}}) \odot \tilde{\mathbf{x}} \right) + \frac{1}{2} W_1 (\underline{\mathbf{x}} + \overline{\mathbf{x}}) + b \ . \tag{13}$$

Obtaining $\tilde{W}_1 \in \mathbb{R}^{n \times m}$ and $\tilde{b}_1 \in \mathbb{R}^n$ has time complexity $O(nm)$ since it requires multiplying each column of W_1 by the respective scaling factor $(\overline{x}_i - \underline{x}_i)$ as well as a matrix-vector product and vector addition for $\tilde{b}$.

Concrete upper bounds on $\overline{\hat{\mathbf{n}}}_1 \geq \tilde{W}_1 \tilde{\mathbf{x}} + \tilde{\mathbf{b}}_1$ can be obtained by setting $\overline{\hat{\mathbf{n}}}_1 = \tilde{W}_1^+ \mathbf{1} + \tilde{W}_1^- (-\mathbf{1}) + \tilde{\mathbf{b}}_1$ (similarly for the lower bound) and thus requires again $O(nm)$ steps due to the matrix-vector multiplications.

Finding polynomial overapproximations requires computing the exact maximum or minimum deviation of the given polynomial to the ReLU function for each of the n neurons in the first layer. We make no further assumptions on the maximization procedure and thus simply denote the runtime by $O(n\tau_d)$ with τ_d representing the time taken by an optimization procedure to calculate the maximum of a degree d polynomial.

Propagation of an n-dimensional polynomial with m monomial terms through an elementwise polynomial map of degree d requires $O(m^d(m+n)) = O(m^{d+1}n)$ operations in expectation and produces an n-dimensional polynomial with $m' \in \Theta(m^d)$ monomials according to Appendix A. During propagation, we jointly

summarize duplicate monomials in $\mathbf{lb}(\mathbf{x})$ and $\mathbf{ub}(\mathbf{x})$ to ensure that both have the same exponent matrix.

Note that all of the above operations only need to be computed once during execution of Algorithm 2 and that their time complexity is dominated by the $O(m^{d+1}n)$ operations for the propagation through the polynomial relaxations.

Each iteration of the loop in line 7 is responsible for calculating concrete bounds on layer i of the NN. To this end, Algorithm 1 is used for linear backsubstitution from layer i down to the second layer, which requires $O((i-1)n^3)$ operations.

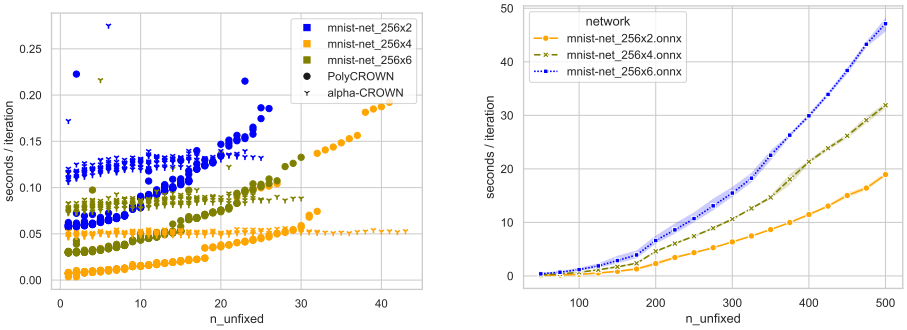
Another important factor is the substitution of the polynomial relaxations for the first layer in lines 9 and 10. During this step, we need to multiply $\Gamma^+, \Gamma^-, \Lambda^+, \Lambda^- \in \mathbb{R}^{n\times n}$ with the coefficient matrices $G_l, G_u \in \mathbb{R}^{n\times m'}$ of $\mathbf{lb}(\mathbf{x})$ and $\mathbf{ub}(\mathbf{x})$, requiring $O(n^2m^d)$ operations. Since $\mathbf{lb}(\mathbf{x})$ and $\mathbf{ub}(\mathbf{x})$ have the same exponent matrix, Equation (6) is applicable and addition is not a factor in the asymptotic cost. Overall, a single iteration of the for-loop requires $O((i-1)n^3 + n^2m^d) = O((i-1)n^3 + n^2m^d)$ operations.

Using the backsubstitution approach to calculate bounds for every layer up until the output layer L involves computing the relaxation for the first hidden layer once and $L-1$ loop iterations. Adding the time complexity of the previous steps to the overall complexity of the loop, the total expected cost is in $O(m^{d+1}n + n\tau_d + L^2n^3 + Ln^2m^d)$.

C Experimental Results for Runtime

In Proposition 1, we showed that the number of unfixed input dimensions n_{un} has a large impact on the time complexity of Algorithm 2. To investigate the effect on runtime in practice, we first analyze the runtimes for a single forward pass of PolyCROWN on the MNIST benchmark set for the properties with small to intermediate n_{un} described in Section 4 and compare to the runtime of a single forward pass of α-CROWN, before looking at the behavior for larger n_{un}.

For comparison of the runtime of α-CROWN and PolyCROWN, we we restrict our analysis to properties, where α-CROWN was able to verify the property with the initial (unoptimized) relaxation – in these cases, the overall runtime matches the time for one iteration. This is necessary, since we did not log the number of iterations taken by α-CROWN, but only its overall runtime for a verification task, Figure 1a shows that the time required by a single forward pass of PolyCROWN already grows polynomially for small to intermediate n_{un}. Interestingly, a single forward pass of α-CROWN takes almost constant time regardless of n_{un} – meaning that the algorithm does not capitalize upon potential runtime savings for small input dimensions. Therefore, a single iteration of PolyCROWN is indeed faster than a single iteration of α-CROWN for up to 19-30 unfixed input dimensions depending on the network used. For larger n_{un}, however, PolyCROWN's runtime increases, while α-CROWN's time still stays virtually constant.



(a) Small to intermediate number of unfixed input dimensions.

(b) Large number of unfixed input dimensions.

Fig. 1. Runtime on NNs trained on MNIST for a single forward pass for α-CROWN and PolyCROWN (left) and just PolyCROWN (right) for properties with increasing number of unfixed input dimensions.

To specifically examine PolyCROWN's runtime for larger n_{un}, we conducted as separate experiment. We selected the first 5 images from the MNIST benchmark set described in Section 4 and increased the number of unfixed input dimensions n_{un} from 50 to 500 with a step-size of 25. We then executed PolyCROWN with 20 iterations of gradient-optimization for each combination of image, n_{un} and MNIST NN with $2, 4$ and 6 hidden layers of 256 neurons each. While the average runtimes per iteration for $n_{un} = 50$ are quite fast ranging from 0.07 seconds (2-layer NN) to 0.39 seconds (6-layers), Figure 1b shows a significant increase to between 18.93 and 47.11 seconds depending on the NN, when 500 unfixed input dimensions are used.

References

1. Althoff, M., Grebenyuk, D., Kochdumper, N.: Implementation of taylor models in cora 2018. In: Frehse, G. (ed.) ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems. EPiC Series in Computing, vol. 54, pp. 145–173. EasyChair (2018). https://doi.org/10.29007/zzc7, https://easychair.org/publications/paper/9Tz3
2. Kochdumper, N., Althoff, M.: Sparse polynomial zonotopes: A novel set representation for reachability analysis. IEEE Trans. Autom. Control. **66**(9), 4043–4058 (2021). https://doi.org/10.1109/TAC.2020.3024348, https://doi.org/10.1109/TAC.2020.3024348
3. Kochdumper, N., Schilling, C., Althoff, M., Bak, S.: Open- and closed-loop neural network verification using polynomial zonotopes. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13903, pp. 16–36. Springer (2023). https://doi.org/10.1007/978-3-031-33170-1_2, https://doi.org/10.1007/978-3-031-33170-1_2