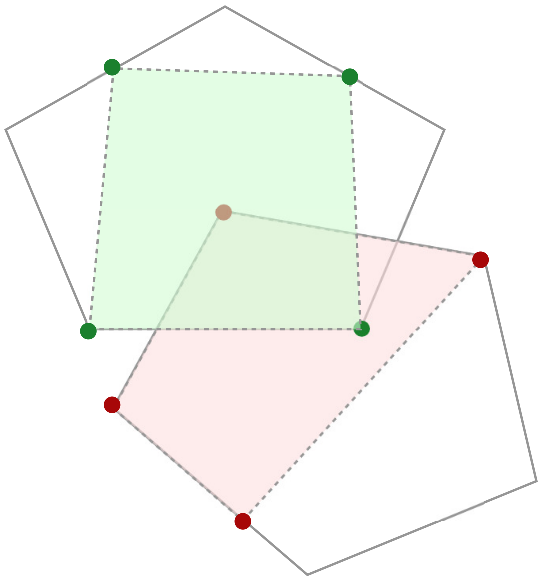
Scaling k-ReLU relaxations for Boosting Certification



Hleb Makarchuk

Master Thesis
August 2020

Supervisors:
Gagandeep Singh, Prof. Dr. Markus Püschel

ETH zürich

Abstract

With the growing adoption of neural networks for decision making in critical domains such as autonomous driving, medicine, and finance, it is critical that their classification is provably robust in an adversarial region. One popular way to construct such proof consists of symbolically propagating the adversarial region through the network. This is computationally hard due to the non-linear activation functions commonly employed in the networks. The most precise existing approach for propagating such a region that also scales to large networks involves approximating the output of the activation neurons jointly. This computation is expensive and involves solving the challenging convex hull problem. In this thesis, we develop new fast specialized methods for approximating the output of multiple ReLU neurons that allow improving speed and precision of network verification over state-of-the-art significantly.

Acknowledgment

I want to thank Gagandeep Singh and Prof. Dr. Markus Püschel for supervising this project and helping me progress in solving this challenging problem. I am grateful to my parents for always being there for me. And I would like to thank my friends in Zurich for the last wonderful two years.

Contents

List of Figures

List of Tables

1 Introduction

Neural networks are being increasingly used in safety-critical domains. However, it has been discovered that an adversary can fool the network by providing an adversarial example. An adversarial example is a specifically designed input to the network that is very similar to the original correctly classified example but causes a model to make a mistake. Therefore it is critical to provide provable guarantees that the safety-critical network is robust against adversarial attacks.

To certify that a network is robust on a particular input, we want to prove that network would provide the same output for all inputs in the adversarial region produced by perturbing the original input. Network certification is computationally challenging problem and the complexity of certification grows exponentially with the size of the network and the input.

1.1 Key challenges

To certify that the network is robust to changes in the input, we need to show that the output of the network stays the same for any input in the region. The region can be defined as a polyhedron in a multi-dimensional space. Major computational complexity is computing how the region changes when it is propagated through the network's non-linear activation functions. Computing transformation exactly is infeasible for any neural network of non-trivial size. Moreover, the optimal region description can also grow exponentially with each non-linear layer.

These computational difficulties create a need for an approximation - when propagating the input region through the activation layers, we need to trade precision for speed. Most of the prior work has been focusing on propagating the region through activation neurons separately, by computing a triangle relaxation [5]. Although fast, this approach suffers from precision loss and often fails to certify the robustness of the network.

The latest state-of-the-art method of certification propagate the region through neurons jointly [12]. This approach greatly improved precision and yielded state-of-the-art results. However, it also increases the computational complexity of certification and creates a need for a framework that would make passing region through neurons jointly fast and precise.

1.2 Contributions

The main bottleneck for certification based on multi-neuron region propagation is the computation of the region's convex relaxation. We make the following contributions:

- We introduce a novel theoretical framework that allows us to compute convex relaxations with a small loss of precision efficiently.
- We use the above framework to design a fast algorithm for computing k-ReLU relaxation that achieves a hundredfold speedup over the existing implementation.
- We provide a new methodology for computing efficient relaxations on the activation functions layer yielding analysis precision and scalability that is beyond the reach of existing methods for computing convex relaxations.
- A complete end-to-end implementation of our approach and a detailed evaluation showing the benefits of our approach. Our results show that our approach increases the percentage of certified robustness by ≥ 9 points over state-of-the-art results for 4 out of 8 benchmarks while having a similar runtime.

1.3 Structure of this Document

.

1. Chapter 2 gives the necessary background for understanding the thesis
2. Chapter 3 introduces a new theoretical framework for computing convex hull approximation
3. Chapter 4 explains a new algorithm for fast computation of k-ReLU relaxations
4. Chapter 5 introduce heuristics for fast and precise relaxation of the whole activation layer
5. Chapter 6 evaluates new algorithm for computing k-ReLU relaxations and shows the effectiveness of the method on network verification benchmarks
6. Chapter 7 discusses future work

1.4 Remarks

For simplicity, we focused on ReLU activation functions since they appear in the vast majority of benchmarks. However, most of this work can be applied for other activation functions.

Chapter 3 intentionally discusses the theoretical framework for solving convex hull problem approximately agnostically of k-ReLU relaxation, since these algorithms can be useful for other problems. In Chapter 4 we explain how these algorithms are applied to k-ReLU relaxations.

2

Background

2.1 Polyhedral computations

Convex polyhedron is a set of points that satisfy a system of linear inequalities:

$$P = P(A, b) := \{x \in \mathbb{R}^d : Ax \geq b\}$$

where $A \in \mathbb{R}^{m\times d}$ and $b \in \mathbb{R}^m$.

Polyhedra allow us to describe arbitrary linear relations between neurons. Such flexibility comes at a cost - some operations on polyhedra are very expensive and have exponential time and space complexity. In this section, we will introduce the main polyhedra computational problems that are necessary for understanding the thesis.

We will use words *polyhedron*, *polytope* and *convex polyhedron* interchangeably. Furthermore, since polyhedra arising in neural network certification are bounded, we will assume it to simplify definitions. However, the boundness is not a requirement for any of the algorithms described.

According to Minkowski-Weyl Theorem, any polyhedra has two representations - an intersection of a finite number of halfspaces and a convex hull of a finite number of vertices. These representations are called *H-representation* and *V-representation* respectively. More formally [8]:

Theorem 1 (Minkowski-Weyl's Theorem)**.** *For $P \subset \mathbb{R}^d$ the following is equivalent:*

(a) P is a polyhedron - there $\exists A \in \mathbb{R}^{m\times d}$ and $\exists b \in \mathbb{R}^m$ such that $P = \{x : Ax \geq b\}$.

(b) P is finitely generated - exist vectors v_i's in $\mathbb{R}^d$ such that $P = ConvexHull(v_1, ..., v_n)$.

The homogeneous version of a polyhedron is called a polyhedral cone. Polyhedral cones are generated as a non-negative combination of *rays* and can also be described by the system of constraints A, with the zero b coefficient. There is a trivial transition between homogeneous cones and non-homogeneous polyhedra, and all algorithms for them are equivalent. Throughout the thesis, we will be switching between these two terms, since it is easy to illustrate how the algorithm works on polyhedra, however, most of the existing theoretical results are given in polyhedral cone formulation.

Of course, polytope does not identify its constraint or vertexes representation; there can be a multitude of representations that define the same polytope. For example, representations can differ through *redundancy* - vertex or halfspace that can be removed from the representation without changing the polytope. Determining if vertex or halfspace is redundant can be formulated as a linear program. The representation without redundancies is called *minimal* and is notably unique for every polytope. The efficient determination of the redundancies is one of the main problems in polyhedral computations. The *minimal* V-representation is known to consist of *extremal* vertexes. The vertex is called extremal if it cannot be represented as a linear combination of two other vertexes in the polyhedra.

The existence of two representations is handy - some problems are trivial to compute in one representation and extremely difficult in the other. For example - intersecting two polyhedra is trivial in their H-representation - the result is just a union of two sets of inequalities representing halfspaces. Indeed if one polyhedron is described by a set of inequalities H_1 and another by a set H_2, then x belongs to the intersection if and only if it satisfies both H_1 and H_2. On the other hand, Tiwary in [15] have shown that computing convex hull of two polyhedra in H-representation is NP-hard. The opposite holds for polyhedra in V-representation - computing convex hull of two polytopes is trivial, and the intersection is NP-hard.

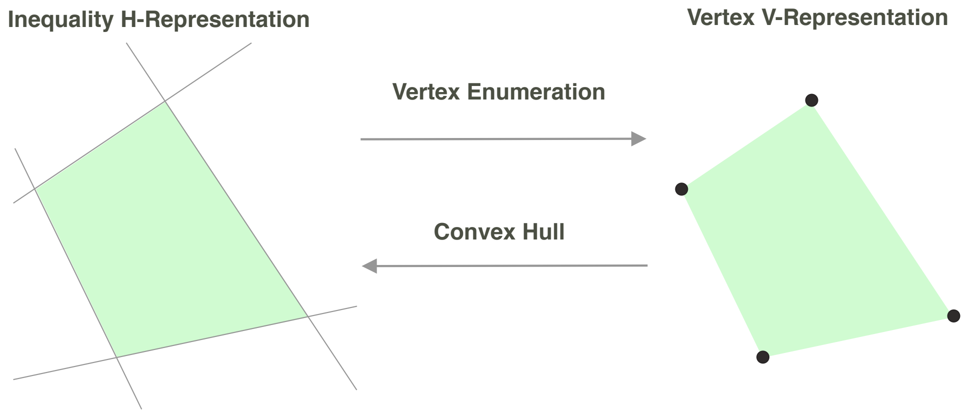


Figure 2.1: Conversion between two representations of the polytope with convex hull and vertex enumeration algorithm

Since some operations are trivial in one representation and difficult in another, it is important to be able to compute one representation from another (Fig. 2.1). Finding H-representation based on V-representation is called *convex hull* problem. The reverse computation is called *vertex enumeration* problem. Unfortunately, in the general case, both of these problems are known to be NP-hard.

Another interesting notion is polyhedra *duality*. The intuitive way of understanding duality is that constraints and vertexes are essentially the same things. From the computational standpoint, it means that many problems in polyhedra computation are connected. For example, computing the convex hull of two polyhedra in H-representation can be formulated as an intersection of dual polytopes' V-representations. The same goes for representation conversion problems - because of the duality vertex enumeration problem is equivalent to the convex hull problem. Further, we will mostly refer to the convex hull problem as vertex enumeration problem, since algorithms are often described and more intuitive to understand in this setting.

Vertex enumeration problem is one of the main problems in polyhedra computation. The problem is NP-hard, so there are no algorithms that run in time polynomial of both size of input and output. There are two main main classes of algorithms for vertex enumeration - *incremental* algorithms and *graph traversal* algorithms [3].

Graph traversal, also known as *pivoting* algorithms, are based on traversing vertexes of the polytope. The vertex can be identified as an intersection of d affinely independent halfspaces. Traversal is based on *pivoting* operation - following the edge between two vertexes by changing one of the identifying hyperplanes. Pivoting algorithms can run in polynomial time on *non-degenerate* polytopes - when every extremal vertex is incident to exactly d facets of the polytope. These algorithms also perform well when the polytope is mostly non-degenerate - the number of constraints that go through vertex is only slightly bigger than d. Unfortunately, non-degeneracy is a very strong restriction and performs pivoting algorithms plummets in degenerate cases. Then the pivoting operation has to be done an exponential number in every such vertex, and algorithms perform poorly.

The *incremental* algorithms are based on sequentially intersecting polyhedra with new halfspaces. At every step, the algorithm keeps track of both V-representation and H-representation, also known as *double description*, and computed how V-representation changes when new halfspace intersects polytope. Incremental algorithms have a caveat - choosing the intersection order with hyperplanes has a critical effect on the performance. The size of intermediate representation can grow exponentially relative to both input and output. Another critical element of incremental algorithms is *redundancy* removal. It is essential to keep the intermediate V-representation in *minimal* form, otherwise it quickly grows beyond any tractable size. Improvements to incremental algorithms are usually based on finding a better heuristic of removing redundant vertices at one step of the algorithm [8].

In practice, incremental algorithms can work well for degenerate polytopes. Here, it is important to mention *cddlib*. It is a highly efficient C-library that implements the incremental Double Description method. It implements various heuristics for making computations fast and offers the best results in our computation.

2.2 DeepPoly

DeepPoly [13] is an abstract domain designed for network verification. Since operations with polyhedra are very expensive, DeepPoly is a restricted case of polyhedra. In particular, it allows one to associate exactly one upper and lower bound relational bound with every neuron. With this restriction, the polyhedra abstract domain allows for scaling to verify large networks and supports fast operations.

A notable feature of DeepPoly is that it supports doing affine operations with neurons without loss of precision. It means that fully-connected and convolutional layers are processed without loss of precision. Our work uses DeepPoly as an underlying system for computing relational constraints between neurons that are needed to compute relaxations.

2.3 k-ReLU Relaxation

If input neurons to ReLU can take both negative and positive values, the transformation becomes non-linear and convex relaxation must be applied. The optimal convex relaxation for the case of one neuron is easy to compute - it is a triangle. Computation of the inequalities describing the triangle is trivial and follows from the basic geometric principles.

The problem arises when we want to compute the convex relaxation jointly. Note that ReLU is a simple linear transformation if lower and upper bounds are both less or greater than zero and can be applied without precision loss. Thus further, we will assume that all neurons in the group have a negative lower bound and a positive upper bound. Also, further we will denote x_1, x_2, ..., x_k as input neurons to ReLU and y_1, y_2, ..., y_k as output neurons. DeepPoly guarantees that every neuron always has a finite lower and upper bound, thus polyhedra is guaranteed to be bounded.

We will then describe the main idea behind computing joint relaxation originally presented by Singh et al. in [12]. We start with $A(x_1, ..., x_k)$ inequality representation of polyhedron obtained from DeepPoly. Then we split the input in 2^k smaller polyhedra such that $\forall i \in 1, ..., k$ we have either $x_i \leq 0$ or $x_i \geq 0$. Doing such splitting in inequality representation is trivial, each of the smaller polyhedra can be described by adding to the original set of constraints new k inequalities of the form $x_i \leq 0$ or $x_i \geq 0$.

The idea behind this split is to establish a trivial relation between y_i and x_i for each of the smaller polyhedra. Indeed, if $x_i \leq 0$, then trivially $y_i = 0$ else if $x_i \geq 0$ then $y_i = x_i$. Thus if we compute the vertex representation of polyhedra, we can trivially compute the vertex representation in space $(x_1, ..., x_k, y_1, ..., y_k)$.

The convex hull operation in vertex representation is trivial - we have to take the union of extreme vertices. The main computational complexity represents converting back from vertex representation into inequality representation, which is done with a convex hull algorithm. The computation is NP-hard and creates a major bottleneck for computing k-ReLU relaxation despite significant research efforts in designing an efficient algorithm.

The upside is that for neural network certification, one does not have to compute precise convex hull relaxation, but can compute over-approximation instead. It opens up opportunities to design an approximate convex relaxation algorithm and trade precision for speed. Moreover, the inputs we get in the neural network setting have a specific structure that can be used to advantage.

3

Partial Double Description

Convex hull, or *vertex enumeration*, in the general case, is an NP-hard problem. Thus, the runtime of any algorithm that solves the problem precisely grows very quickly with the problem's size. Finding the exact solution becomes infeasible, and thus, we need to find a *sound* approximation. In this chapter, we propose a theoretical framework for computing approximation of convex hull and allows to gain significant speedup at a small cost of precision, which will show in the chapter on evaluation.

The sound approximation of the solution to the convex hull problem consists of finding a set of constraints that bound the input set of vertices. Note that this definition is very flexible, and the polytope bounded by constraints only has to contain the polytope spanned by the vertex set. Of course, for the solution to be useful, we want the convex relaxation to enclose the set of vertices tightly, but no restrictions on precision are posed. Note that in homogeneous view of the problem for polyhedral cones, if constraints are given by matrix A and rays are given by matrix R, then approximation would be sound if scalar product between any ray and any constraint is non-negative.

We will call the dual version of this problem *partial vertex enumeration* (Fig. 3.1), and it can be formulated as finding a set of vertices inside the polytope induced by input constraints such that it tightly props the hyperplanes. It might seem counter-intuitive that the dual version of computing convex hull *over-approximation* is enumerating vertices that are *inside* of the polytope induced by constraints. However, this is the case because in dual view of the problem scalar product remains non-negative, and thus vertices have to be inside the polytope.

We will further discuss the problem in its partial vertex enumeration formulation to be consistent with other works on the double description method [16][8]. This chapter aims to set theoretical grounding and explain the algorithm of fast computation of convex relaxation of two polytopes. In the next chapter, we will show how this algorithm can be used to accelerate the computation of k-Relu relaxation.

Note that due to the equivalence of homogeneous polyhedral cone and non-homogeneous polytope, we will use these notions interchangeably. Respectively we will use notions or rays and vertices interchangeably. We will call polyhedral cone A-induced, or A-cone if A is a set of constraints bounding the cone. And we will call the cone R-spanned if every polytope ray is a non-negative combination of rays in R.

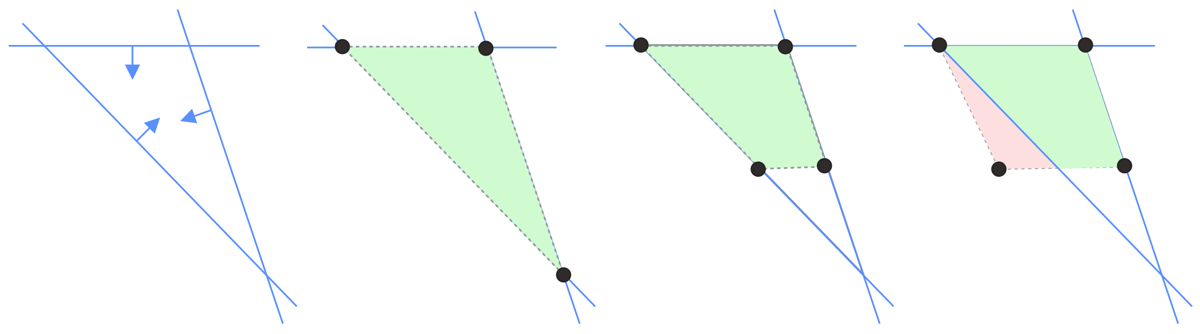
The constraint $a \in A$ is called active for ray r, if $a \cdot r = 0$, i.e. the ray is incident to the constraint. The set of all active constraints for ray r we will call active set of r and denote as the matrix of these constraints as A_r. Note that the rank of A_r has a significant theoretical importance - ray r is extremal if and only if $rank(A_r) = d - 1$ [8]. We will call $rank(A_r)$ as rank of r in A.

3.1 Relaxation of double description

Double description, or incremental, methods can perform well for vertex enumeration problems. However, the problem is NP-hard, and the algorithm's runtime can grow very quickly due to the potentially large size of the intermediate representation. We propose a concept of *partial double description*, or *PDD*, for polyhedral cones that allow for a significant speedup for some common operations at a small precision cost. In particular, it allows us to define a fast algorithm for finding approximate V-representation of the intersection of two cones.

Definition 3.1.1. Pair (A, R) where $A \in \mathbb{R}^{m \times d}$ and $R \in \mathbb{R}^{n \times d}$ is called a partial double description pair if for any row $r \in R$ and any row $a \in A$, holds $a \cdot r \geq 0$.

Note that the difference with the double description is that inclusion has to hold only one way. I.e., the R-spanned cone is contained entirely in the A-induced cone by the system $Ax \geq 0$. This is equivalent to the requirement of *soundness*—this way, for example, Fig. 3.1c is a sound PDD because although the vertex is inside polyhedra, its scalar product with all constraints is positive. Fig. 3.1d is, on the other hand, unsound, because one of the vertices gives negative scalar product with the constraint.



(a) Input Constraints (b) Vertex Enumeration (c) Partial Vertex Enumeration (d) Unsound Vertex Enumeration

Figure 3.1: Variations of Vertex Enumeration

Although definition doesn't pose any restrictions except for the soundness, for relaxation to be useful, we are aiming to keep polyhedral R-spanned cone as close as possible to A-induced polyhedra. This way, a double description is the best possible partial double description since R-spanned polyhedral cone is optimal.

The basic data structure for describing consists of two matrices with constraints and rays, respectively. It can also be beneficial to store the incidence information between constraints and rays instead of computing it when it is needed (Alg. 1).

Algorithm 1 Data structure for PDD

```
1: struct PDD
2:     $A$: Matrix $\mathbb{R}^{m \times d}$ where rows represent constraints of the PDD
3:     $R$: Matrix $\mathbb{R}^{n \times d}$ where rows represent rays of the PDD
4:     $Incidence$: Sets of incident constraints for every ray.
5: end struct
```

3.2 Removing redundancies from representation

A step of the double description method consists of computing how generator representation R changes after adding the next constraint. In a primitive version of the method, a massive amount of redundant vertices gets created that should somehow get eliminated. Otherwise, the size of representation explodes. Removing redundancies efficiently is a key component of any incremental method. Efficient double description methods offer different approaches to dealing with this problem. For example, Verge in [16] proposes criteria to detect redundant vertices and then use the incidence relation between two representations to remove the rest. Fukuda in [8] avoids creating redundant constraints altogether but does it at the cost of computing and storing information about adjacency between vertices.

In essence, all efficient incremental approaches are based on exploiting the relation between two descriptions to remove redundancies efficiently. In the case of partial double description, we apply a similar principle to reduce the representation's size.

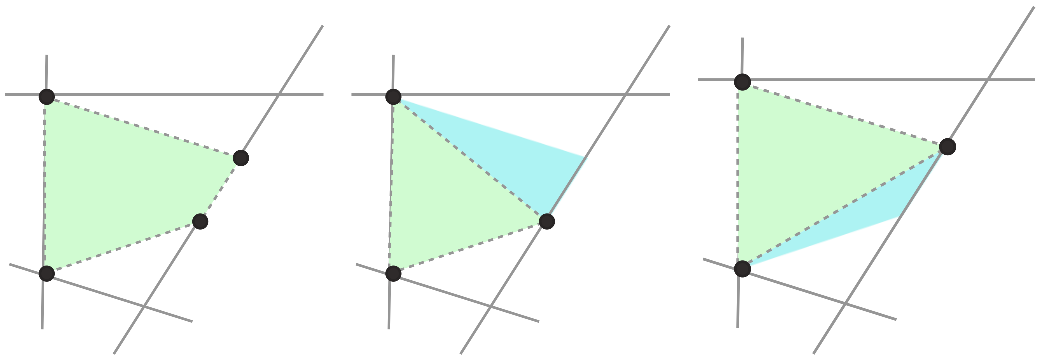
Definition 3.2.1. Let (A, R) be a PDD. (A, R) is called A-irredundant if for any ray $r \in R$, there is no other ray $r' \in R$, s.t. the active set of constraints for ray r is a subset of active set for any other ray $r' \in R$.

Note that A-irredundancy is equivalent to irredundancy in a case (A, R) is a double description. For removing redundancies we take the common idea from the double description methods [16][6] and leave out rays that have maximal active set, i.e. such that for any ray $r \in R$, there is no other ray $r' \in R$ such that $A_r \subset A'_r$ (Alg. 2).

Although the procedure is the same, there are a few key differences when working with PDDs. In the case of double description, the procedure generates the set of extremal vertices of A-cone. For PDDs, computing A-irredundancy can lead to the loss of precision. This way on Fig. 3.2 it can be seen that the A-irredundant vertex set covers less area than the original vertex

Algorithm 2 Algorithm for making PDD A-irredundant

```
 1: procedure ComputeIrredundant(PDD)
 2:     Sort indexes of rays by cardinality of their incidence set
 3:     Initialize list of A-irredundant rays as an empty list
 4:     for Iterate through sorted rays do
 5:         if Current ray's active set is not a subset of any other ray's active set in list of A-irredundant rays then
 6:             Add the ray to the list of A-irredundant rays
 7:         end if
 8:     end for
 9:     return A-irredundant PDD
10: end procedure
```



(a) Initial PDD that has A-redundancies (b) First option of making PDD A-irredundant (c) Second option of making PDD A-irredundant

Figure 3.2: Computation of A-irredundancy. Green area denotes polytope spanned by vertices, blue area denotes precision loss due to A-irredundancy computation.

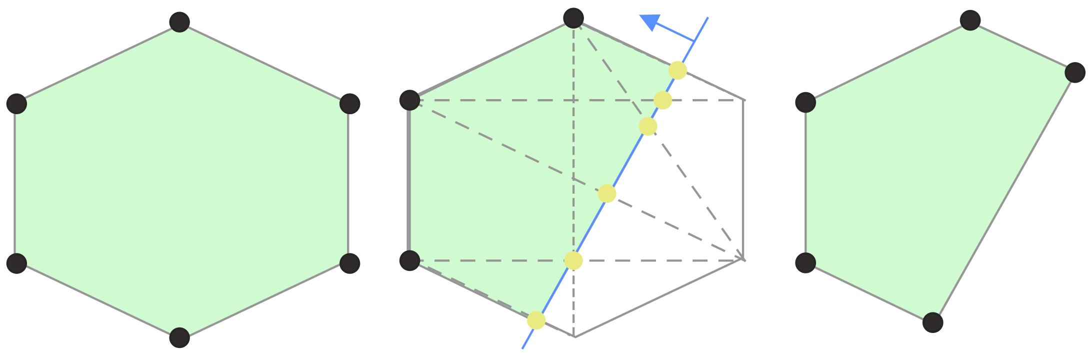
set. This way, A-irredundancy doesn't provide the same theoretical guarantees as in double description algorithm. Note that the computation is sound since vertices remain to be bounded by constraints. The algorithm also preserves a useful theoretical property that if ray $r \in R$ is extremal in A-cone, them the procedure will preserve it. This is due to the fact that extremal rays have maximal active sets among all possible rays in cone [16].

Another notable distinction with irredundancy computation for double description is that the result depends on the ordering in which we filter vertices. Indeed, in Fig. 3.2, two possible options are shown for the final set of vertices. It contrasts with computation of redundancy for double description that can give only one result fully consisting of extremal vertices.

There are other possible approaches to redundancy elimination that do not suffer from the precision loss. It is known that detecting a single redundancy is equivalent to solving LP and can be formulated as LP [7]. There is also ongoing work to create heuristics to solve this problem more efficiently [10]. However, the key distinction of double description methods is that relation between two representations allows to implement this procedure very efficiently. Hence, it offers a good speed-precision trade-off for PDDs.

3.3 Adding constraints to PDD

The double description method operates by adding constraints iteratively one-by-one to the double description of the polytope. There are multiple variations of this procedure. However, in essence, rays get split into three sets R_-, R_0, and R_+ based on whether they are outside, on the border, or inside the halfspace induced by new constraint. Then every pair of rays $(r_-, r_+) \in (R_-, R_+)$ creates a new ray $r*$ based on intersection with the new hyperplane. The final set of vertices in the procedure is $R = R_0 \cup R_+ \cup R_*$. Notably, a huge number of redundant rays can be generated. Thus redundancy elimination is essential.



(a) Initial PDD

(b) Computation of new vertexes created by adding new constraint to PDD

(c) PDD after removing redundant vertices with A-irredundancy procedure

Figure 3.3: Procedure of changing the PDD when adding one constraint

Similar to DD, the PDD also supports adding constraints based on the same principle. Note that in Fig. 3.3 adding a new constraint to the PDD creates many redundant vertices colored in yellow. A-redundancy elimination procedure filters out these vertices, thus significantly reducing the size of the representation.

Note, however, that adding one constraint to PDD is the same procedure as for DD and is not novel. The advantage of PDD is that operations don't need to be precise, and thus some operations can be done significantly faster. Here we propose an algorithm for adding constraints in *batch* rather than one at a time. The main problem of adding constraints iteratively is that it potentially can create a big number of intermediate rays that will be eliminated at the later stage of the algorithm.

Adding constraints in batch can skip large intermediate representations at a cost of precision. The idea is similar to the iterative procedure - we split the input set of rays into ones satisfying all new constraints R_+ and the ones that have at least one violation R_-. Then we construct new rays from pairs $(r_-, r_+) \in (R_-, R_+)$.

The algorithm's efficiency comes from the fact that despite potentially many constraints between r_- and r_+, only one new vertex will be generated. To find this vertex, a simple ray-shooting procedure is computed where we iterate through the list of constraints that r_- violates and choose the one that is closest to the vertex r_+. Note that in Fig. 3.4a the vertices marked with red the ones quickly rejected by the ray-shooting procedure. This way we efficiently split vertices that don't belong to the final polytope, thus contributing to the algorithm's efficiency.

Algorithm 3 Algorithm for adding one constraint to PDD

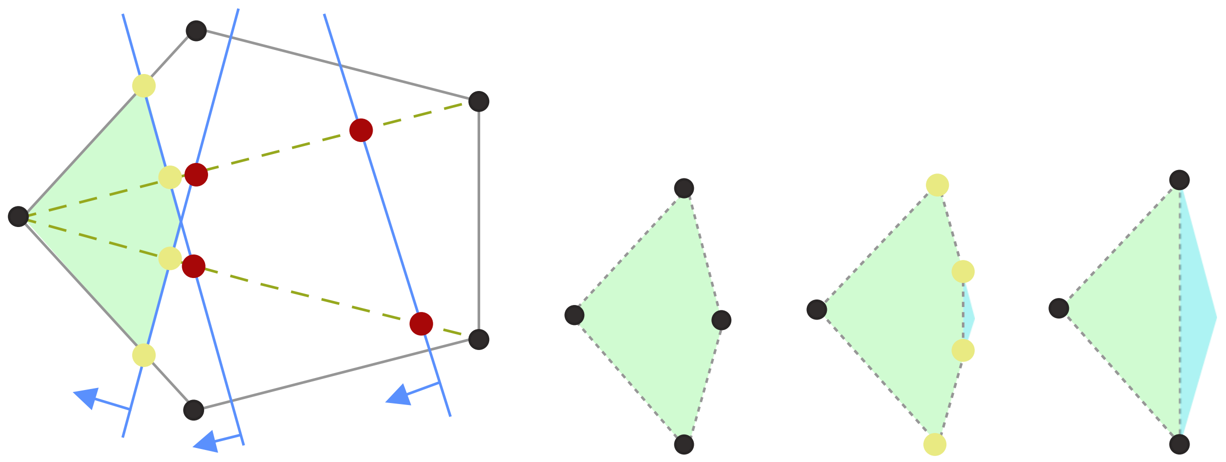
```
1: procedure AddOneConstraint(PDD, new constraint $a$)
2:     Based on sign scalar product between rays of PDD and new constraint split rays of PDD
       into three sets $R_-$, $R_0$ and $R_+$
3:     Let $R^*$ be an empty list of new vertices
4:     for $r_- \leftarrow R_-$ do
5:         for $r_+ \leftarrow R_+$ do
6:             Compute new ray $r^* = (a \cdot r_+)r_- - (a \cdot r_-) \cdot r_+$
7:             Append new ray $r^*$ to the list of ray $R^*$
8:         end for
9:     end for
10:    Add new constraint to the list of constraints of PDD
11:    Set new set of rays of PDD as $R_+$, $R_0$ and $R^*$
12:    Make PDD A-irredundant with Alg. 2
13:    return PDD
14: end procedure
```

It is important to note that the pair $(r_-, r_-^*) \in (R_-, R_-)$ also has a small chance of creating up to two new vertices through the ray-shooting procedure. It is possible when the ray, connecting r_- and r_-^* has a region that does not violate any of the constraints. In practice, in our computations of k-ReLU relaxations, it was rarely the case. Thus we disabled it as optimization. However, for proving some interesting theoretical properties of the algorithm, we will be assuming that ray shooting also happens between r_- and r_-^*

Algorithm 4 Algorithm for adding batch of constraints to PDD

```
1: procedure AddBatchConstraints(PDD, New constraints)
2:     Let $P$ be a scalar product between PDDs rays and new constraints
3:     Based on $P$ split rays into violating at least one constraint $R_-$ and the rest $R_+$
4:     Let $R^*$ be an empty list of new rays
5:     for $r_- \leftarrow R_-$ do
6:         for $r_+ \leftarrow R_+$ do
7:             Compute new ray $r^*$ with a ray shooting procedure from $r_+$ to $r_-$ based on $P$
8:             Append new ray $r^*$ to the list of rays $R^*$
9:         end for
10:    end for
11:    Add new constraints to the list of constraints for PDD
12:    Set new set of rays of PDD as $R_+$ and $R^*$
13:    Make PDD A-irredundant with Alg. 2
14:    return PDD
15: end procedure
```

Although the algorithm doesn't have any guarantees in the general case, we can prove an interesting theoretical property of the algorithm if the input is the double description. Note that in our proof, we assume that we also do ray-shooting between violating vertices R_-, although it was not practically useful on our particular problem.



(a) The vertices are discovered with ray-shooting procedure

(b) Vertices that would be computed by precise algorithm

(c) Vertices discovered by ray-shooting procedure

(d) Vertices after computing A-irredundant set of vertices

Figure 3.4: Procedure of changing the PDD when adding batch of constraints. Blue area denotes precision loss compared to the optimal solution.

Lemma 2. *Let ray r has rank $d-2$ in A-cone and is extremal in $A \cup A^*$-cone. Then there exists constraint $a* \in A^*$, s.t. r is extremal in $A \cup a^*$.*

Proof. By definition if rank of r in A-cone is $d-2$, then the maximal linearly independent group of active constraints from A_r has size $d-2$. We will denote this group G. Since r is extremal in $A \cup A^*$ we similarly have the linearly independent group G^* of size $d-1$ [16]. Since the rank of G^* is greater than the rank of G, there is at least one row a^* in G^* that is linearly independent of G. However, a^* can't be the row from A since otherwise maximal linearly independent r's zero sets in A would have size $d-1$, which contradicts the rank condition. Thus a^* has to be from A^*. □

Proposition 3. *Let $P(A, R)$ be a double description of a polyhedral cone in d-dimensional space and A^* be a set of new constraints. Then the result of the algorithm 4 would be a partial double description $P(A \cup A^*, R')$ where R' is guaranteed to enumerate **all** of the following extremal rays of $A \cup A^*$ cone:*

1. *Rays that are extremal in A-induced cone*
2. *Rays that have rank $d-2$ in A-induced cone and extremal in $A \cup A^*$-induced cone*

Proof. We can formally divide the rays of partial vertex enumeration R' produced by the algorithm 4 in two non-intersecting groups:

- Rays R_+ of R, i.e. rays in the input R that did not violate any of the inequalities A^*
- New rays R_* created with a ray-shooting procedure

Because the R is a vertex enumeration of the cone P, R has to contain all extremal rays of P. It implies that all extremal rays of A that do not violate any of the new inequalities, will go into R_+. Since rays remain extremal when new constraints are added, they can't be eliminated by the redundancy elimination algorithm (Alg. 2) and thus will remain. This concludes the proof of the first part of the statement.

For the second part we will use the following fact. Let's take any extremal ray r from $A \cup A^*$-cone that has rank $d-2$ in A-cone. It was shown that r has to intersect one of the inequalities a^* of A^*. Also from [8] we know that if r has rank $d-2$ in A it can be represented as a positive linear combination of two extremal rays of A. Since r is a positive combination and intersects inequality hyperplane, at least one of these extremal rays $\notin A \cup A^*$-cone. Thus the pair will be processed in the algorithm 4. The r will be detected by ray-tracing between two vertices which concludes the proof.

Note that since these enumerated rays are extremal in $A \cup A^*$ cone, they cannot be eliminated with A-redundancy elimination procedure (Alg. 2). □

Based on proposition 3, we can build an understanding of batch intersection in terms of the double description method. It efficiently does one step of the double description method from the initial representation (A, R) for every constraint from A^* separately. The efficiency comes from the ray shooting procedure - the vertices that do not belong to the final PDD get efficiently rejected. It comes at the cost of precision - extremal rays of $A \cup A^*$ cone that are created at the intersection of two or more constraints from A^* are unlikely to be discovered by the algorithm.

Note that adding constraints in a batch essentially **inverts** the process compared to the double description method. In the double description method, the constraints are added iteratively, and for every constraint, we iterate through pairs of vertices. In the method of adding constraints in a batch, we iterate through pairs of vertices and for every pair very efficiently iterate through new constraints with ray-shooting.

3.4 Computing intersection and convex relaxation of two PDDs

Enumerating the intersection of two polyhedral cones is NP-hard problem [15] even if both A and R are available. The precise way to do this computation would be to start from double description representation $P_1(A_1, R_1)$ of one of the cones and then iteratively adding hyperplanes from A_2. However, due to the potential explosion of the intermediate representation size, the runtime of this approach can be very close to solving the enumeration problem from scratch. Using the approximate partial double description together with the batch intersection can bring a significant speedup. The main drawback of such an approach is the potential loss of too much precision. Additionally, we are not exploiting the second given partial double description $P_2(A_2, R_2)$.

Here we present a fast algorithm for computing the intersection of two polyhedral cones in the partial double description representation 5. The algorithm exploits the connection between A and R in both PDDs and has similar runtime as batch intersection while potentially having much higher precision. Note that due to polytope duality, the algorithm is equivalent to computing the convex hull of two polytopes, which we will use in the next chapter.

Algorithm 5 Algorithm for computing intersection of two PDDs

```
1: procedure ComputeIntersection($PDD_1$,$PDD_2$)
2:     Let $PDD_1$ = AddBatchConstraints($PDD_1$, constraints of $PDD_2$)
3:     Let $PDD_2$ = AddBatchConstraints($PDD_2$, constraints of $PDD_1$)
4:     Let $A$ be a union of constraints of $PDD_1$ and $PDD_2$
5:     Let $R$ be a union of rays of $PDD_1$ and $PDD_2$
6:     Compute irredundant PDD from $(A, R)$ with Alg. 2
7:     return PDD
8: end procedure
```

Although the algorithm can't provide theoretical guarantees when the input is PDD, we can formulate the following theoretical result for DD that would help to understand why the algorithm is effective.

Proposition 4. *Let $P_1(A_1, R_1)$ and $P_2(A_2, R_2)$ be double description pairs of two polyhedral cones. Then the following holds for the partial double description $P(A, R)$ computed with algorithm 5. The algorithm is guaranteed to enumerate* ***all*** *of the the following extremal rays of $A_1 \cup A_2$-cone:*

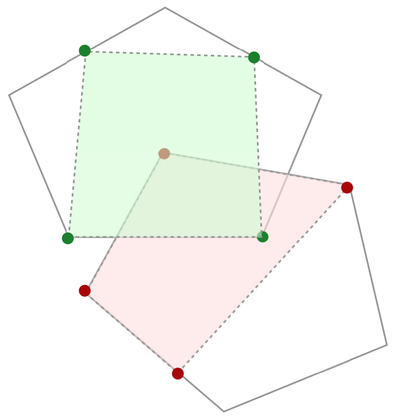
1. *Rays that are extremal in A_1-cone.*
2. *Rays that have rank $d - 2$ in A_1-cone and extremal in $A_1 \cup A_2$-cone.*
3. *Rays that are extremal in A_2-cone.*
4. *Rays that have rank $d - 2$ in A_2-cone and extremal in $A_1 \cup A_2$-cone.*

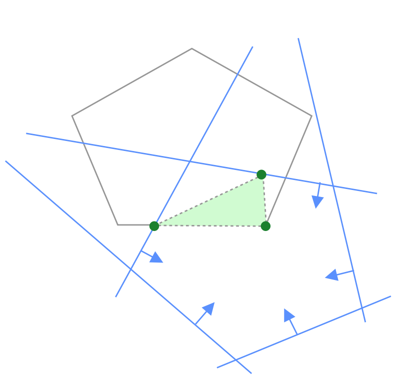
Proof. By proposition 3 batch intersection of P_1 with A_2 guarantees to produce rays (1) and (2). Similarly, batch intersection of P_2 with A_1 will produce rays (3) and (4). Since all of these rays are extremal in $A_1 \cup A_2$, the redundancy removal algorithm 2 will not eliminate them. $\square$

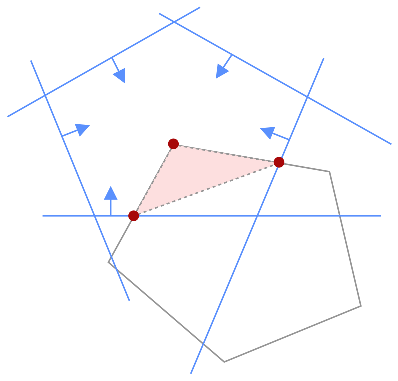
This result allows us to formulate the precision guarantees for the algorithm is smaller dimensions.

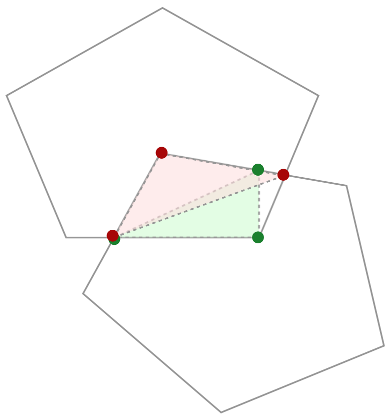
Proposition 5. *Let $P_1(A_1, R_1)$ and $P_2(A_2, R_2)$ be double description pairs of two polyhedral cones with dimension $d \leq 4$. Then partial double description $P(A, R)$ computed with algorithm 5 is in fact double description. Moreover, R is irredundant, i.e. consists only of extremal vertices.*

Proof. For briefness sake we will show result for $d = 4$, with proof being similar for smaller dimensions. Let $R*$ be a set of extremal rays of the $A_1 \cup A_2$-polytope. Let R^* be a set of extremal rays of this polytope. Since rays $r \in R*$ are extremal we have $rank(r, A_1 \cup A_2) = 3$ - there is at least one set of 3 linearly independent rows in $A_1 \cup A_2$. From the pegionhole principle at least one holds for these 3 rows.


(a) Two initial PDDs


(b) Batch intersection of PDD_1 with constraints of PDD_2


(c) Batch intersection of PDD_2 with constraints of PDD_1


(d) Intersection of PDDs computed by taking union of vertices and constraints

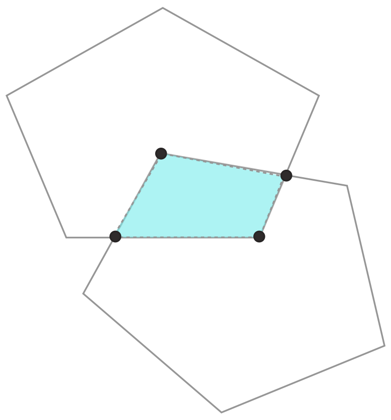

(e) Intersection of PDDs after removing redundancies

Figure 3.5: Procedure of intersecting two PDDs

- all are from A_1, then $rank(r, A_1) = 3$
- all are from A_2, then $rank(r, A_2) = 3$
- two are from A_1 and one is from A_2, then $rank(r, A_1) \geq 2$ and $rank(r, A_1 \cup A_2) = 3$.
- two are from A_2 and one if from A_1, then $rank(r, A_2) \geq 2$ and $rank(r, A_1 \cup A_2) = 3$.

Since by proposition 4 all these cases are enumerated by the algorithm 5 - we will enumerate all extremal vertices of $A_1 \cup A_2$. The A-redundancy elimination algorithm 2 in that case works as a regular redundancy elimination algorithm and thus resulting R will be irredundant. $\square$

Although the intersection algorithm essentially uses batch intersection algorithm 4 two times, it provides a much stronger theoretical guarantee. Even if we start of *double description* of $d = 3$ polyhedral cone, batch intersection can lead to the loss of precision. In contrast, the intersection algorithm 5 guarantees precise operations for $d = 4$ polyhedral cones (or 3D polytopes) given in *double description*.

And even though when the input is PDD, it is impossible to give any guarantees for the algorithm, it can have good results in practice. An interesting case is shown in Fig. 3.5. Both PDDs in the input is far from the double description - the enumerated vertices describe only a small part of the polytope. After the computation of batch intersection, vertices in each of PDD cover only part of the polytope. However, in the final step, when we take the vertices' union, it becomes the optimal vertex enumeration.

Note that the precision of the algorithm starts to drop when we go to higher dimensions. Following the logic in this proof we can see that for cones of dimension $d = 5$ the algorithm doesn't guarantee enumerate all of the extremal rays. As the dimension grows, the number of extremal rays that are not enumerated will continue to grow by implication of an increasing number of combinatorial options of how necessary rank of $d-1$ can be achieved from two sets A_1 and A_2.

Algorithm 6 Algorithm for computing convex relaxation of two PDDs

```
1: procedure ComputeRelaxation($PDD_1$,$PDD_2$)
2:     Let $PDD_1^*$ be the dual of $PDD_1$ and $PDD_2^*$ the dual of $PDD_2$
3:     Let $PDD^* = ComputeIntersection(PDD_1^*, PDD_2^*)$
4:     return the dual of $PDD^*$
5: end procedure
```

The algorithm for computing convex relaxations of two PDDs is easy to formulate in terms of intersection based on the dual view of the problem (Alg. 6). This algorithm is useful for computing convex relaxation of k-ReLU, as will be shown in the next chapter.

4

Fast k-ReLU relaxation

Computing k-ReLU relaxations can constitute a significant part of the runtime during network verification. Accelerating this computation would improve the speed of verification. Additionally, it would lead to improved precision since more computationally expensive strategies could be tried. In this chapter, we propose algorithms for fast computation of k-ReLU approximation that give more than a hundredfold speedup over the exact methods as a small cost of precision.

4.1 k-ReLU relaxation problem

Formally, the problem of computing k-ReLU relaxation can be formulated as given the input constraints A on variables $(x_1, ..., x_k)$ and relation $y_i = ReLU(x_i)$, $\forall i = 1..k$, compute set of constraints A^* on variables $(x_1, ..., x_k, y_1, ..., y_k)$, s.t. for $\forall(x_1, ..., x_k)$ that satisfy A, $(x_1, ..., x_k, ReLU(y_1), ..., ReLU(y_k))$ satisfies A^*.

From the algorithmic perspective, the computation of k-ReLU optimal relaxation can be formally divided into two parts. In the first part, all extreme vertices of the polytope are computed. In the second part of the computation, the convex hull problem is solved to compute constraints that bound these vertices.

In order to simplify the description of the algorithm, we will introduce the following definition:

Definition 4.1.1. Let A be a set of constraints defining a polytope P in space $(x_1, ..., x_n)$. Then we will call *quadrant* a maximal subset of the polytope where $\forall x \in P$ and $\forall i \in 1, .., n$ it holds that $x_i \geq 0$ or $x_i \leq 0$.

In order to find extreme vertices, one can note that ReLU transformation consists of two linear branches. The input polytope induced by A can be split in 2^k quadrants, and in every quadrant, a neuron has an exclusively positive or negative value. Then for every such quadrant, extreme vertices are found by solving the vertex enumeration problem. Note that in every quadrant, ReLU is a linear transformation - thus, it is trivial to project these vertices on space $(x_1, ..., x_k, y_1, ..., y_k)$. The union of all these vertices will enumerate the vertices of the final polytope. The last part of the computation is solving the convex hull problem in order to find the constraint representation of the polytope.

In this chapter we will present novel fast algorithm for computing k-ReLU relaxations. From the computational perspective, at least 90% of the runtime is spent on computing the final convex hull problem. We address it with the algorithm based on approximate convex relaxation methods described in Chapter 3 (Alg. 6) and decomposition principle described in the next section. It allows to reduce part of the runtime spent on computing constraints significantly. We also propose algorithms for fast computation of the first part of the problem - finding extreme vertices of quadrants. I

Also we will introduce an improved version of the exact algorithm based on CDD that computes *sound* relaxation using a mixed exact and floating-point arithmetic that achieves significant speedup over existing CDD-based implementation.

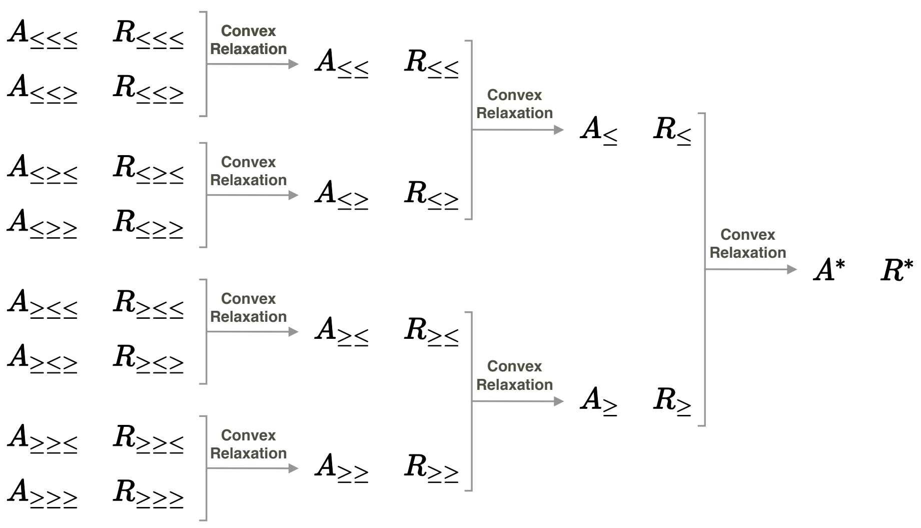
4.2 Decomposition for computing convex relaxation

The majority of the runtime is spent on computing constraints from the union of all vertices. The specific setting of the problem allows us to efficiently leverage the algorithm for computing convex relaxation of two polytopes (Alg. 6) through *decomposing* the problem. As an input to the algorithm, we assume a double description (A_q, R_q) for every quadrant is already computed.

The algorithm's main idea is to leverage a fast algorithm for computing convex hull of two polytopes given in partial double description. Notice that all quadrants can be split into disjoint groups of two, such that quadrants in every group differ only by sign in one particular orthant e.g., x_n. Then convex relaxation can be computed for each of these groups. The process is repeated with newly created groups for all other orthants. Figure 4.1 schematically shows the process for $k = 3$.

This algorithm is computationally efficient for two reasons. Contrary to the standard algorithm, we do not need to project in space $(x_1, ..., x_n, y_1, ..., y_n)$. Indeed projecting in space $(x_1, ..., x_n, y_n)$ is sufficient since $y_1, ..., y_{n-1}$ are already strictly defined as $y_i = 0$ or $y_i = x_i$. Thus, the algorithm benefits from solving a problem in space in a lower dimension. And instead of finding precise convex hull, convex relaxation can be computed *approximately* with Alg. 6. As we have shown in the previous chapter, the speedup comes from having both A and R representations for both polytopes.

To give an example, we will formally go through the computation for a case $k = 2$. Let A be input set of constraints, the algorithm assumes that it gets double descriptions of quadrants (A_q, R_q) as an input:



$A_{\leq\leq\leq}$ $R_{\leq\leq\leq}$
$A_{\leq\leq\geq}$ $R_{\leq\leq\geq}$
$A_{\leq\geq\leq}$ $R_{\leq\geq\leq}$
$A_{\leq\geq\geq}$ $R_{\leq\geq\geq}$
$A_{\geq\leq\leq}$ $R_{\geq\leq\leq}$
$A_{\geq\leq\geq}$ $R_{\geq\leq\geq}$
$A_{\geq\geq\leq}$ $R_{\geq\geq\leq}$
$A_{\geq\geq\geq}$ $R_{\geq\geq\geq}$
Convex Relaxation
$A_{\leq\leq}$ $R_{\leq\leq}$
$A_{\leq\geq}$ $R_{\leq\geq}$
$A_{\geq\leq}$ $R_{\geq\leq}$
$A_{\geq\geq}$ $R_{\geq\geq}$
$A_{\leq}$ $R_{\leq}$
$A_{\geq}$ $R_{\geq}$
A^* R^*

Figure 4.1: Scheme of computing convex hull for 3-ReLU through decomposition. The subscript shows the sign of the first orthants in the PDD. E.g. $A_{\leq\geq}$ means that $x_1 \leq 0$, $x_2 \geq 0$ and no restrictions are posed on x_3.

$$A_{\leq\leq} = \begin{bmatrix} A \\ x_1 \leq 0 \\ x_2 \leq 0 \end{bmatrix} \quad A_{\leq\geq} = \begin{bmatrix} A \\ x_1 \leq 0 \\ x_2 \geq 0 \end{bmatrix} \quad A_{\geq\leq} = \begin{bmatrix} A \\ x_1 \geq 0 \\ x_2 \leq 0 \end{bmatrix} \quad A_{\geq\geq} = \begin{bmatrix} A \\ x_1 \geq 0 \\ x_2 \geq 0 \end{bmatrix}$$

$$R_{\leq\leq} = \begin{bmatrix} | & | \\ a_2 & a_2 \\ | & | \end{bmatrix} \quad R_{\leq\geq} = \begin{bmatrix} | & | \\ b_1 & b_2 \\ | & | \end{bmatrix} \quad R_{\geq\leq} = \begin{bmatrix} | & | \\ c_1 & c_2 \\ | & | \end{bmatrix} \quad R_{\geq\geq} = \begin{bmatrix} | & | \\ d_1 & d_2 \\ | & | \end{bmatrix}$$

Note that $(R_{\leq\leq}, A_{\leq\leq})$, $(R_{\leq\geq}, A_{\leq\geq})$, $(R_{\geq\leq}, A_{\geq\leq})$ and $(R_{\geq\geq}, A_{\geq\geq})$ are double description. Also note that computing a convex hull of quadrants $x_1 \leq 0 \cap x_2 \leq 0$ and $x_1 \leq 0 \cap x_2 \geq 0$ can be done in space (x_1, x_2, y_2), without involving y_1. Indeed we know that in both of these quadrants $y_1 = 0$, thus we can take a convex relaxation of these two groups in space (x_1, x_2, y_2) without involving y_1.

Formally, quadrants $\leq\leq$ and $\leq\geq$ in the space (x_1, x_2, y_2) are:

$$A^{y_2}_{\leq\leq} = \begin{bmatrix} A_{\leq\leq} \\ y_2 = 0 \end{bmatrix} \qquad A^{y_2}_{\leq\geq} = \begin{bmatrix} A_{\leq\geq} \\ y_2 = x_2 \end{bmatrix}$$

$$R^{y_2}_{\leq\leq} = \begin{bmatrix} | & | & | \\ a_1 & a_2 & 0 \\ | & | & | \end{bmatrix} \qquad R^{y_2}_{\leq\geq} = \begin{bmatrix} | & | & | \\ b_1 & b_2 & b_2 \\ | & | & | \end{bmatrix}$$

Then, using the algorithm for fast computation of convex hull of two PDDs from the previous chapter 6, we get:

$$PDD_{\leq\leq} = (A^{y_2}_{\leq\leq}, R^{y_2}_{\leq\leq}) \quad PDD_{\leq\geq} = (A^{y_2}_{\leq\geq}, R^{y_2}_{\leq\geq})$$

$$PDD_{\leq} = ComputeRelaxation(PDD_{\leq\leq}, PDD_{\leq\geq})$$

Note that $R_{\leq}$ in $PDD_{\leq}$ is trivially concatenation of $R_{\leq\leq}$ and $R_{\leq\geq}$.

$$R_{\leq} = \begin{bmatrix} | & | & | \\ a_1 & a_2 & 0 \\ | & | & | \\ b_1 & b_2 & b_2 \\ | & | & | \end{bmatrix}$$

Similarly, from the second group $\geq\leq$ and $\geq\geq$ we compute $A_{\geq}$ and $R_{\geq}$. Note that because dimension of polyhedra on (x_1, x_2, y_2) is 3, by proposition 4 both $A_{\leq}$ and $A_{\geq}$ are exact convex hulls. Both $A_{\leq}$ and $A_{\geq}$ are trivially projected on higher dimensional space (x_1, x_2, y_2, y_1):

$$A^{y_1}_{\leq} = \begin{bmatrix} A_{\leq} \\ y_1 = 0 \end{bmatrix} \qquad A^{y_1}_{\geq} = \begin{bmatrix} A_{\geq} \\ y_1 = x_1 \end{bmatrix}$$

$$R^{y_1}_{\leq} = \begin{bmatrix} | & | & | & | \\ a_1 & a_2 & 0 & 0 \\ | & | & | & | \\ b_1 & b_2 & b_2 & 0 \\ | & | & | & | \end{bmatrix} \qquad R^{y_1}_{\geq} = \begin{bmatrix} | & | & | & | \\ c_1 & c_2 & 0 & c_1 \\ | & | & | & | \\ d_1 & d_2 & d_2 & d_1 \\ | & | & | & | \end{bmatrix}$$

Finally with help of algorithm 5, we compute the final relaxation:

$$PDD_{\leq} = (A^{y_1}_{\leq}, R^{y_1}_{\leq}) \quad PDD_{\geq} = (A^{y_1}_{\geq}, R^{y_1}_{\geq})$$

$$PDD^{*} = ComputeRelaxation(PDD_{\leq}, PDD_{\geq})$$

Thus, the set of constraints would A^* from PDD^*. Note that the final relaxation is an approximation of convex hull and is not guaranteed to be optimal. As seen, the algorithm has a tree structure and is easy to implement recursively. Recursion goes through every orthant i and splits execution in two paths - where $x_i \leq 0$ and where $x_i \geq 0$. When we reach the final orthant, the procedure returns a precomputed double description for the corresponding quadrant. When the procedure goes up the recursion stack, it trivially computes the projection that includes y_i for two PDDs and then uses the algorithm 6 to compute convex relaxation.

Note although final set of constraints A^* is not guaranteed to be a convex hull, it is guaranteed to be over-approximation of the convex hull. This algorithm for computing relaxation is not precise. However, there is a handy way to trade precision for speed. Notice that it is possible to do multiple passes of decomposition, with different orderings of variables. The total number of orderings is $k!$.

Every pass might produce new constraints; however, the runtime of relaxation grows linearly with the number of passes, and every next pass has less chance to discover new constraints. If multiple passes are done, it is important to remove redundancies with the algorithm 2 since likely many constraints will be rediscovered. In order to do multiple passes, there is no need to recompute initial double descriptions of quadrants. For the case $k = 3$, we choose to do 3 out of 6 possible passes:

$$\{x_0, x_1, x_2\} \quad \{x_1, x_2, x_0\} \quad \{x_2, x_0, x_1\}$$

This way to compute final relaxation have significantly reduced the runtime of the algorithm which will be shown in the chapter on evaluation.

4.3 Fast vertex enumeration of quadrants

Through decomposition, the runtime for computing constraints from quadrant vertices reduced the runtime from dominating to partial. Thus, in this section, we propose a fast algorithm for computing the initial double descriptions for all quadrants.

The original procedure has constructed a set of constraints for every quadrant and solved vertex enumeration problem separately for every quadrant (Fig. 4.2). This approach, however, is suboptimal since the sets of constraints for quadrants differ insignificantly from each other - they consist of the input set A together with k constraints on x_i that define a quadrant. Since most of the constraints in every quadrant are shared - it is possible to share computation.

The proposed algorithm starts with enumerating the vertices of the input set of constraints A. Then the step of the algorithm "splits" the vertex set using an orthant x_i into negative path $x_i \leq 0$ and positive path $x_i \geq 0$. The procedure repeats recursively for both paths for the next orthant. Splitting a vertex set in two parts is done by adapting one step of double procedure method - the new hyperplane $x_i = 0$ splits vertices into three sets R_-, R_0 and R_+ based on whether they are on one side of the hyperplane, incident to it or on the other side. Similar to one step of the double description method described in chapter 3, the algorithm creates a new vertex set R^* from pairs $(r_-, r_+) \in (R_-, R_+)$. Then the split would be:

$$R_{x_i \leq 0} = R_- \cup R_0 \cup R^* \quad R_{x_i \geq 0} = R_+ \cup R_0 \cup R^*$$

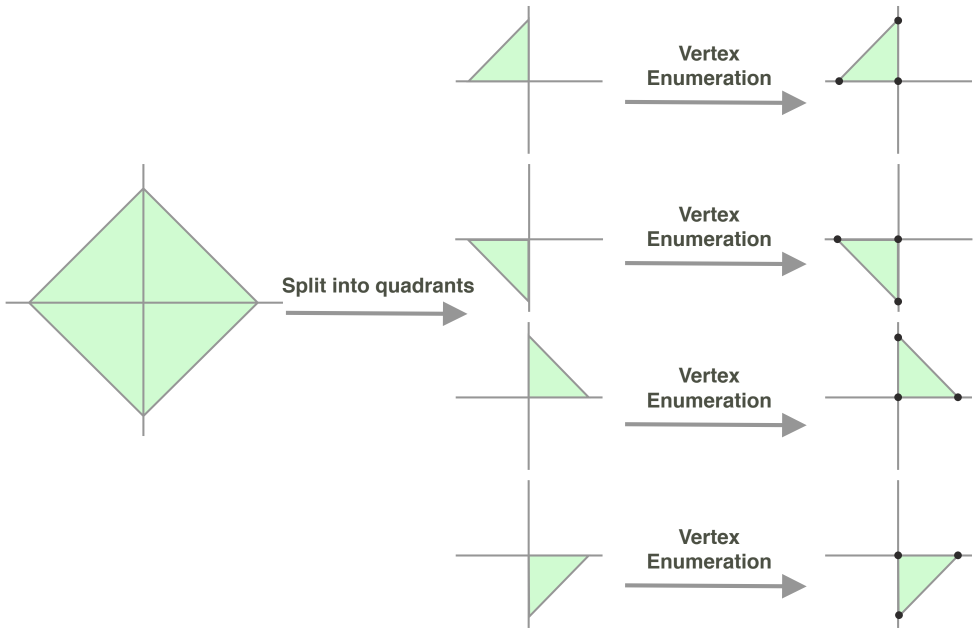


Figure 4.2: Original procedure for enumerating vertices of quadrants

It is easy to prove this approach's correctness - the set of vertices the algorithm would produce would be equivalent to doing a step of double description method for both $x_i \leq 0$ and $x_i \geq 0$. The algorithm uses Fukuda's [8] version of the double description method and only considers adjacent pairs for creating new vertices to avoid creating redundancies. Note that in this algorithm, we only have to use the vertex enumeration algorithm once (Fig. 4.3) compared to 2^k times in the original version. After computing the initial vertex set, the algorithm is computationally equivalent to computing $\approx 2^k$ steps of double description, which is significantly simpler than solving 2^k vertex enumeration problems.

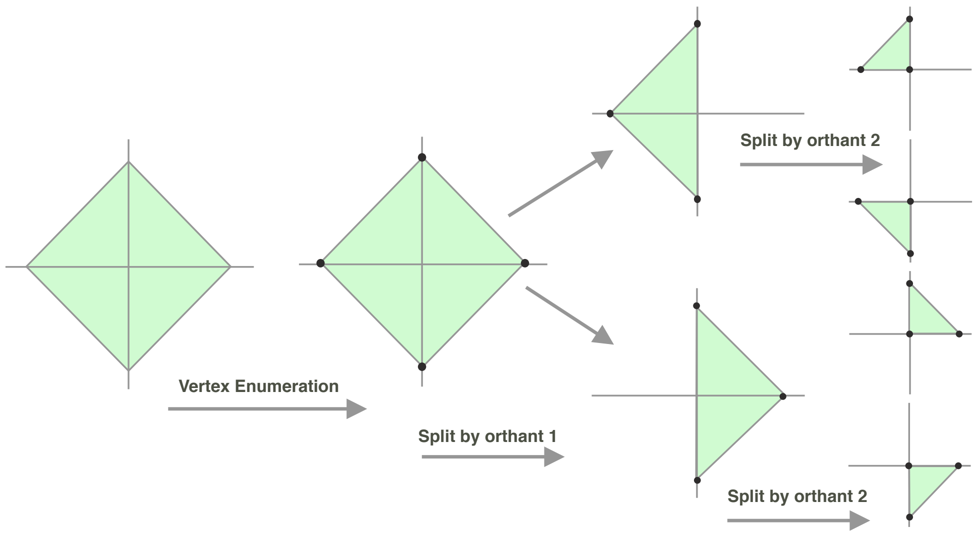


Figure 4.3: Fast procedure for enumerating vertices of quadrants

Apart from the unit tests, the algorithm's correctness was verified on actual network verification problems with integration testing. Both original and new procedures were run for enumerating quadrants' vertices and compared for every input during verification, thus confirming the correctness of the algorithm.

4.4 Fast vertex enumeration of octahedra

Another improvement concerns speeding up the computation of enumerating the vertex set of polytope bounded by input set of constraints A that further gets split into quadrants. The input constraints are computed as octahedral constraints using DeepPoly. From the experiments, we observed that octahedron is a polyhedron with no or few degeneracies. Every vertex of an octahedron of dimension k is incident to k or slightly more constraints. In such a setting, pivoting-based algorithms [4] perform better and, in case of non-degeneracy, can even have polynomial running time in size [3].

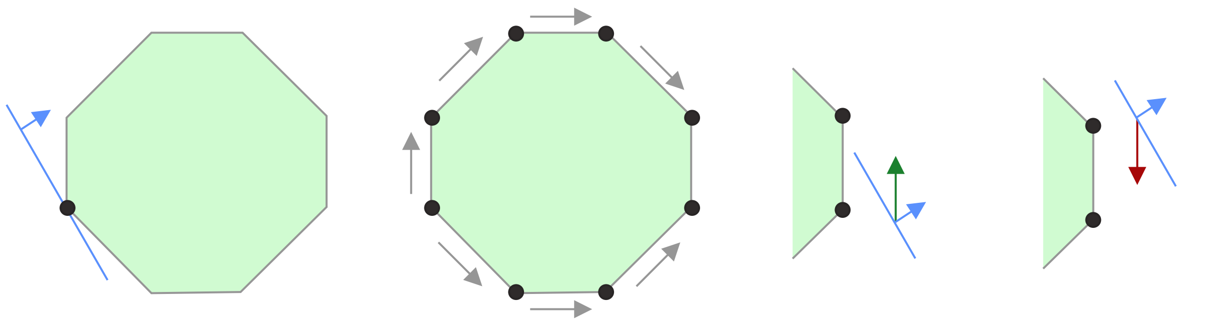

(a) The first vertex of the octahedron is found by solving LP (b) Using pivoting for traversing the vertices of octahedron (c) Traversal proceeds in the *positive* direction (d) No traversal in the *negative* direction

Figure 4.4: Pivoting algorithm for enumerating vertices of the octahedra

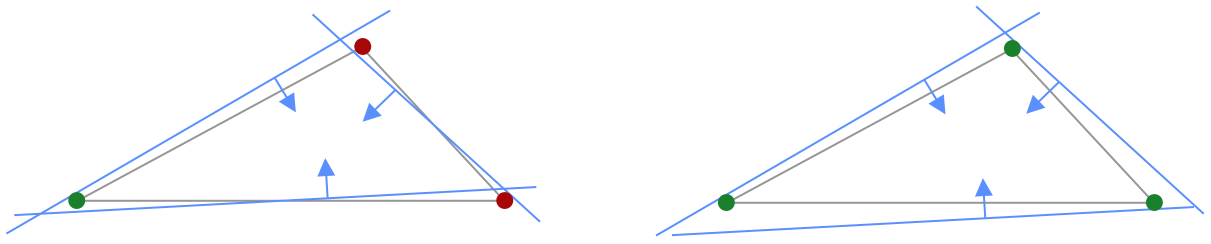
We have implemented a pivoting-based algorithm for computing vertex representation of the octahedron. The algorithm starts with one of the extremal vertices of the octahedron. The first extremal vertex is found by solving a linear program with the help of CDD solver. Then the algorithm traverses octahedron as a graph - visits the vertex and with pivoting operation traverse to the next vertex. To avoid visiting the same vertex multiple times, all visited vertices are stored in a cache.

In the algorithm, we implement some of the commonly used optimizations for the pivoting method. In particular, to traverse octahedron efficiently, we define a traversal direction $\vec{r}$. The first vertex of the octahedron is found by solving LP with the help of CDD by minimizing vertex's scalar product with traversal direction. Then the traversal goes only in the *positive* direction (Fig. 4.4c) - we only pivot to the vertex if the scalar product of the direction towards this vertex is positive. Note that since octahedron is a system of constraints of a certain form, it is possible to precompute the traversal direction such that it is not collinear to any edge of the octahedron.

The computation was done in exact precision. The big computational part of the pivoting algorithm is traversing towards the next vertex. Once the direction is selected, the problem consists of finding the closest hyperplane that intersects the direction - it is done in the operation analogical to ray shooting. This operation was implemented in the mixed floating-point and exact arithmetic. The algorithm uses exact precision arithmetic only when the plane cannot be rejected based on floating-point precision.

4.5 Adjustment of constraints for soundness

The only strict requirement for convex relaxation is *soundness* - none of the constraints should violate the vertex set. The algorithm for computing the initial vertex set for octahedron and splitting it into orthants is extremely fast and can be done precisely. The conversion from quadrant vertices into constraints through decomposition can benefit from using floating-point precision. However, this can lead to minor unsoundness - computed constraints can slightly violate some of the vertices due to imprecise floating-point operations as shown on Fig. 4.5.



(a) Vertices for which soundness if violated are marked with red color

(b) Constraints are shifted to ensure that the relaxation is sound

Figure 4.5: Convex relaxation can have minor unsoundness due to FP precision

In order to ensure that relaxations are sound, in the end the algorithm adjusts constraints s.t. they don't violate any vertices (Fig. 4.5). The constraints are given in the form:

$$\vec{a} = [c, a_1, ..., a_k, b_1, ..., b_k]$$

That is equivalent to a constraint:

$$c + a_1x_1 + ... + a_nx_n + b_1y_1 + ... + b_ny_n \geq 0$$

Simple way to make a constraint sound is increasing coefficient c sufficiently s.t. that it doesn't violate any constraints. The vertices are given in form:

$$\vec{r} = [1, c_1, ..., c_n, d_1, ...d_n]$$

The constraint $\vec{a}$ violated $\vec{r}$ if and only if $\vec{a} \cdot \vec{r} \leq 0$. Let $\vec{a} \cdot \vec{r} = e$, where $e < 0$. Then increasing coefficient c by $|e|$ will give will give a sound constraint $\vec{a^*}$. Indeed then:

$$\vec{a^*} \cdot \vec{r} = \vec{a} \cdot \vec{r} + 1 \cdot |e| = e + |e| = 0$$

This means that the constraint $\vec{a^*}$ became sound for $\vec{r}$. To make constraint sound for all vertices, free coefficient c see should be increased by the absolute value of the biggest negative scalar product. Note that to compute a sound scalar product, one does not have to use exact precision and instead use floating-point interval arithmetic described by Miné in [11].

4.6 Fkrelu: fast algorithm for computing k-ReLU relaxations

After we have explained different parts of the algorithm, we show how they are put together to create fast and precise algorithm for computing k-ReLU relaxation. The algorithm starts with enumerating the vertex set of octahedron using a pivoting-based algorithm for vertex enumeration. Then the vertex set is split into quadrants. Convex relaxation is computed for quadrant vertices with an algorithm based on decomposition. More formally, the procedure is described in Alg. 7

Algorithm 7 Fkrelu: fast algorithm for computing k-Relu relaxation

```
1: procedure FKRELU(input constraints of the octahedron)
2:     Enumerate vertices of octahedra with pivoting method in exact precision
3:     Split vertices into quadrants in exact precision
4:     Let $A^*$ be final list of constraints
5:     for ordering of variables for decomposition do
6:         Compute constraints with convex relaxation algorithm through decomposition for the variable ordering
7:         Add constraints to the list of constraints $A^*$
8:     end for
9:     Remove redundancies from $A^*$ with Alg. 2
10:    Adjust constraints $A^*$ for soundness using vertices
11:    return final set of constraints $A^*$
12: end procedure
```

4.7 Mixed precision CDD-based algorithm

The proposed approach to combine exact and floating-point arithmetic can also accelerate computing k-ReLU relaxation based on CDD library. Indeed, most of the runtime is spent on computing convex hull of quadrant vertices in exact precision. Thus, it is also possible to compute the quadrant vertices with CDD precisely, compute convex hull in floating-point precision, and then adjust all constraints for soundness.

However, this approach doesn't work straightforwardly due to numerical issues. Compared to computing convex relaxation through decomposition, the double description method is more numerically unstable. The constraints are added iteratively - that leads to multiple intermediate vertices created and eliminated. The vertices at the end of the algorithm can be a result of multiple operations with intermediate vertices. Due to floating-point errors, the imprecision accumulates and leads to numerical issues.

Because of this, CDD library can abort the computation due to numerical errors for some inputs when the final convex hull. This occurs in about $\approx 5\%$ of the inputs for $k = 3$. For $k = 4$, all attempts to compute convex hull in floating-point precision have failed due to numerical errors.

Thus, we propose a hybrid approach to achieve the speedup that attempts to solve the problem in floating-point precision, and if it failed, it switches back to exact precision (Alg. 8). If computation succeeded in floating-point, the constraints are adjusted for soundness.

Algorithm 8 Algorithm for computing k-Relu relaxation in Hybrid-arithmetic CDD

```
1: procedure HybridArithmeticCDD(input constraints)
2:     Compute all vertices of quadrants with CDD in exact precision
3:     Attempt computing constraints with CDD floating point precision
4:     if computation succeeds then
5:         Adjust constraints for soundness for vertex representation (Alg. )
6:     else
7:         Compute constraints with CDD exact precision
8:     end if
9:     return constraints
10: end procedure
```

Note that computing convex hull through decomposition suffers much less through numerical issues since the number of intermediate vertices is significantly less. For example, the number of vertices in one quadrant for $k = 3$ is 48. Since there are in total $2^k = 8$ quadrants, every constraint in the final result can theoretically be a result of go through up to $48 \cdot 8 = 384$ intermediate vertices. In the case of decomposition based algorithm, the number of intermediate of vertices limited for every vertex in the result is limited to $k = 3$. The contrast becomes even more stark for the case of $k = 4$ and higher.

5

Convex relaxation framework

The fast algorithm for computation of k-ReLU relaxations described in Chapter 4 is significantly faster than the state-of-the-art implementation of the exact method. However, processing the whole layer *jointly* is infeasible. The complexity of computing relaxation with increasing k still grows quickly. Moreover, the approximation operations described in Chapter 3 become less precise with increasing dimension. This chapter proposes effective methods to process layers with multiple neurons with a small k. We note that good heuristics for computing relaxations for the whole layer significantly affect the effectiveness of verification. Heuristics proposed here contributed towards achieving the best results in multiple categories in the neural network verification competition VNN 2020.

5.1 Sparse groupings of neurons

When the layer has the size n that is greater k, the straightforward way to process it is to group neurons into n/k disjoint groups of size k and to compute relaxation for each of these groups. Although fast, this method might not be precise. Indeed, every neuron appears in exactly one group. However, if the neuron has a tight relational constraint with some other neuron - it has a low chance to be discovered with this approach.

The most precise way to compute the layer's relaxation with k-ReLU is the computing relaxation of every possible group of k neurons. However, the number of groups grows as $\binom{n}{k}$, and the number of groups quickly becomes infeasible with the growing size of the layer.

The trade-off between these two approaches has been proposed by Singh et al. in [12]. There, heuristic splits the layer into n/l bigger disjoint groups of size l, where $l \geq k$. Then within every bigger group, all possible combinations of k neurons are computed. This approach is more precise then splitting layer into disjoint groups of k because more neuron combinations are tried. However, with this approach, l has to be relatively small, because the number of groups increases sharply with growing l - as $n/l \cdot \binom{l}{k}$. Another drawback is that this strategy has low *diversity* - each neuron is only tried in combination with a small subset of layer's neurons.

To address these restrictions, we propose a *sparse* heuristic for grouping neurons. The intuitive explanation of this strategy is that there are no two groupings in the final set that overlap too much. More formally, let G be the set of groupings of neurons of size k. Then:

$$\forall g_1 \in G \quad \forall g_2 \in G \quad |g_1 \cap g_2| < k - l$$

Where l is a *sparsity* parameter $l \geq 1$. One way to construct such set of groups is iteratively - iterate through combinations and only add group if no group with big overlap existed (Alg. 9).

Algorithm 9 Sparse heuristic for grouping neurons of the layer

```
1: procedure SparseHeuristic(Set of input neurons)
2:     Let current selection of groups be an empty list
3:     for new group ← all combinations of input neurons of size $k$ do
4:         if new group doesn't overlap significantly with any of the selected groups then
5:             Add the group to the selection
6:         end if
7:     end for
8:     return current selection of groups
9: end procedure
```

The example of such sparse covering of a group of 6 neurons with $k = 3$ is:

$$(1, 2, 3), (1, 4, 5), (2, 4, 6), (3, 5, 6)$$

Note that none of pairs of groups overlap by more than one neuron.

Although we don't have the analytical formula to show the growth of the number of groups with the increasing n, the number of groups is significantly smaller than a heuristic that selects all combinations and thus allows to use big n.

The sparse heuristic significantly improves the certification procedure. Our explanation of why it might happen is the following. We model the network layer as a hypergraph, where each node represents a neuron, and hyperedge represents a neuron group. Then, in case of sparse heuristic, all of the neurons would have small distances in such a graph, which means that there would exist *implicit* relations between pairs of neurons even if the pair was not processed jointly.

5.2 Adaptive relaxation

The only neurons for which relaxations have to be computed are the neurons where the lower bound is less than zero, and the upper bound is greater than zero. Indeed, if both bounds are less or greater than zero, then ReLU is a linear transformation. However, neurons, where the lower bound is less than zero, and the upper bound is greater, can have a vastly different area of their triangle relaxation. The area is proportional to the loss of precision; thus, it is more important to computing precise relaxations for high-area neurons.

To improve the algorithms' computational efficiency, we propose only to compute joint relaxations for neurons with a big area. To achieve this, we introduce the *cutoff* parameter and only compute relaxations jointly for neurons with triangle area above cutoff. Note that based on our experiments for fully-connected networks, almost all neurons have a significant area; however, in convolutional networks, even a small cutoff parameter can drastically reduce the number of neurons to be processed.

5.3 Selection of group size k

Exact relaxation does not allow scaling beyond $k = 3$. Already for $k = 4$, the original implementation's runtime in exact precision based on CDD exceeds 20 minutes. The Hybrid CDD always falls back to the exact precision due to numerical errors. Fkrelu implementation scales to $k = 4$, and it usually takes 200-300 ms to process one group. However, as we have discussed earlier in Chapter 4, the algorithm based on decomposition loses more precision in higher dimensions, and, although still fast, it is significantly slower then Fkrelu for $k = 3$. Based on experiments, the best results are achieved for the case $k = 3$.

Unfortunately, since it is infeasible to compute volume for relaxations $k \geq 4$, it is hard to substantiate why case $k = 3$ gives the best results. However, we can propose our intuitive explanation. Computation of relaxation already for $k = 4$ is already at least three times more expensive than the computation of $k = 3$. Indeed, three times more input constraints have to be computed for $k = 4$, and the algorithm itself is significantly slower than for $k = 3$. Thus, instead of computing relaxation for one group with $k = 4$, it is possible to compute at least three groups with $k = 3$. Since these groups are selected according to the sparse heuristic, it is possible to capture relations between more neurons and potentially discover tight constraints.

Increasing size, k, is similar to the heuristic proposed by Singh et al. in [12], where all neurons are split into disjoint groups of size $l > k$ and then all possible combinations of neurons are computed within this group. The computation produces tight constraints within this group of neurons. However, it is computationally expensive and doesn't connect neurons implicitly as a sparse heuristic, which has shown to be effective. In all of our experiments, using $k = 3$ showed the best results.

6 Evaluation

For evaluation, we use the ERAN framework for neural network verification that supports DeepPoly. We compare our work against state-of-the-art system *kPoly* [12]. In the first part of the chapter, we compare the Fkrelu algorithm with the existing implementation based on CDD [2]. In the second part of the chapter, we analyze our methods' performance on challenging network verification problems. All experiments were run on a Linux machine with Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz and 100 GB of RAM.

We note the code is publicly available and integrated with ERAN framework [1] for network verification.

6.1 k-ReLU relaxation

One of the main bottlenecks in neural network certification is the computation of convex relaxation of k-ReLU. Thus we will analyze improvements to the k-ReLU relaxation algorithm in isolation. Since runtime slightly differs for different inputs, to make a fair comparison, we have run the network verification on a benchmark - MNIST 6x100 network with L_∞ perturbation to the input. We have measured the runtime of different methods on two images from the input - one where verification succeeds and one where it fails. We note that verification requires the computation of several thousand relaxations.

We compare with the original implementation based on CDD introduced by Singh et al. in [12]. When measured in isolation on one computational thread our proposed algorithm based on Hybrid-arithmetic CDD achieves, on average, ten times speedup compared to the original version (Tab. 6.1). The improvement comes from the significant acceleration of conversion times of orthant vertices into constraints. Originally, this computation takes about 90% of over-

	Hybrid with CDD	Fkrelu
min	0.86	31.92
max	23.34	430.86
mean	9.05	171.20

Table 6.1: The speedup statistics computed on 3752 inputs in comparison with the original CDD-based 3-ReLU implementation (greater is better)

	1-ReLU	Fkrelu
min	3.52	1.00
max	8.46	1.12
mean	4.89	1.03

Table 6.2: The statistics of ratio of 3-ReLU approximate relaxation volume to optimal on 2291 inputs (smaller is better, 1 is the best)

all runtime and is significantly accelerated by switching to floating-point precision. Note that the constraints are adjusted to make the relaxation sound. However, in about $\approx 5\%$ cases, the computation fails due to the double description method's numerical instability - thus having to fall back to the expensive computation in exact precision.

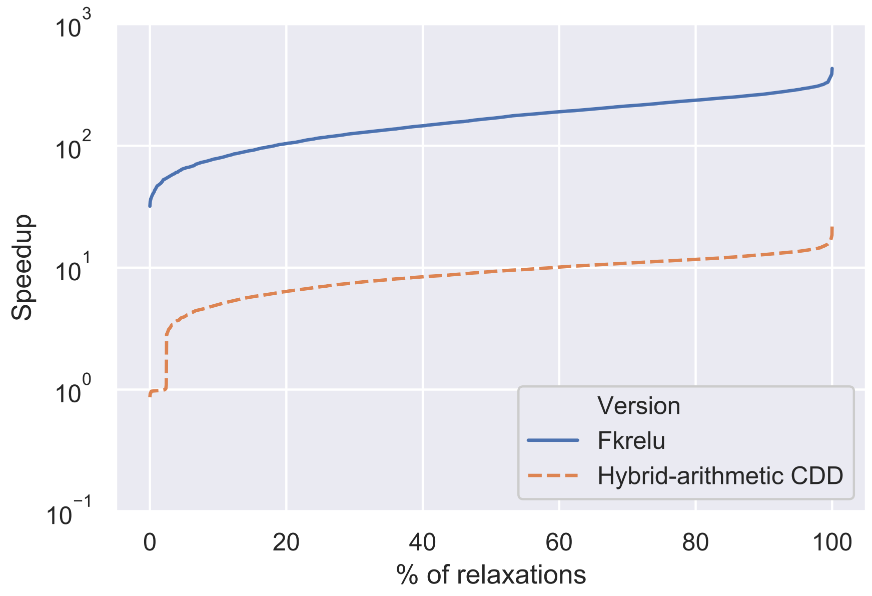


Figure 6.1: Distribution of speedup of proposed k-ReLU relaxation implementations over the original CDD-based implementation computed on 3752 inputs sampled on network verification benchmark (greater is better)

Computing relaxation with Fkrelu is, on average, ≥ 150 times faster than the original implementation (Fig. 6.1). We note that relaxation approximates the solution. Thus the direct comparison is unfair. However, relaxations computed with Fkrelu only slightly deviate from the optimal - on average, their volume is only 3% greater than the optimal (Tab. 6.2). In comparison, the volume of relaxation of neurons computed separately has, on average, five times greater volume compared to optimal. It allows us to conclude that Fkrelu has negligible loss of precision compared to exact relaxation computed with CDD for the case $k = 3$.

We note that computing volume of relaxation for $k = 3$ is a hard computational problem. For this computation, we use QHull implementation included in *scipy* package [17]. However, due to numerical issues, the computation fails for a percentage of inputs and thus it could only be computed for part of relaxations. The volume computation for relaxation $k = 4$ is computationally infeasible - at all our attempts, the implementation was failing due to numerical issues.

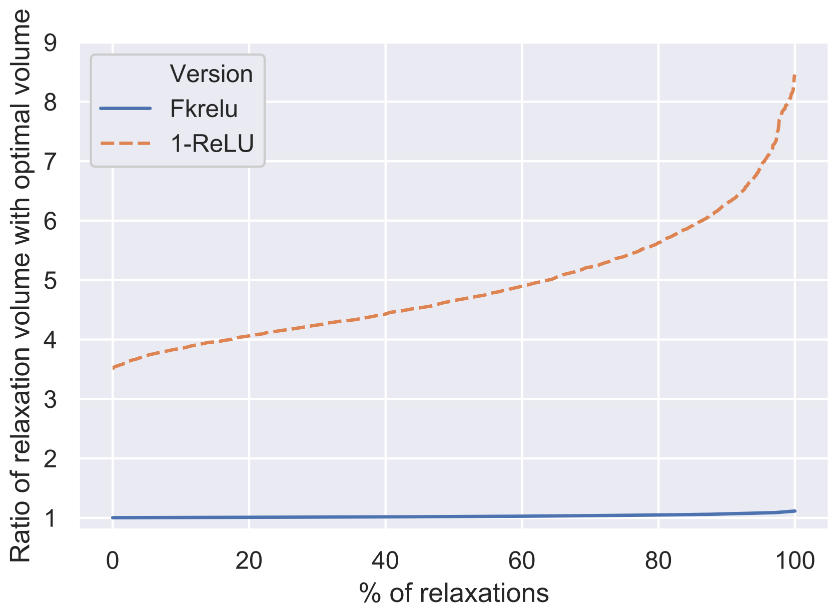


Figure 6.2: The distribution of ratio of 3-ReLU approximate relaxation volume to optimal computed on 2291 inputs (smaller is better, 1 is the best)

6.2 Network verification

For evaluation of our work for neural network verification, we use the ERAN framework. We instantiate the ERAN framework with DeepPoly abstract domain that and provides the input constraints for relaxations. We compare our work against state-of-the-art system k-Poly [12]. We show that our method allows us to verify significantly more regions compared to k-Poly for some of the benchmarks while having a similar runtime.

For benchmarks, we use MNIST and Cifar10 datasets. The adversarial region is L_∞-norm - where each pixel is allowed to change it's value within ϵ deviation. We run experiments on several fully-connected and convolutional networks of different sizes (Tab. 6.3). Similar to k-Poly during verification of fully-connected networks, we refined neurons bounds as described in [14].

Our work managed to verify the greater or same number of images on all of the regions compared to k-Poly. Significant speedup allowed to compute joint relaxation for significantly more groups of neurons, thus capturing more constraints between neurons. In particular, the size of the group for the sparse heuristic described in the previous chapter could be increased to

Table 6.3: Neural network architectures and parameters used in experiments.

Dataset	Model	Type	#Neurons	#Layers	Defense	Refine ReLU	Sparse N	Cutoff
MNIST	6×100	fully connected	510	6	None	✓	100	0
	9×100	fully connected	810	9	None	✓	100	0
	6×200	fully connected	1 010	6	None	✓	100	0
	9×200	fully connected	1 610	9	None	✓	200	0
	ConvSmall	convolutional	3 604	3	None	✗	100	0
	ConvBig	convolutional	48 064	6	DiffAI	✗	100	0
CIFAR10	ConvSmall	convolutional	4 852	3	PGD	✗	100	0
	ConvBig	convolutional	62,464	6	PGD	✗	100	0

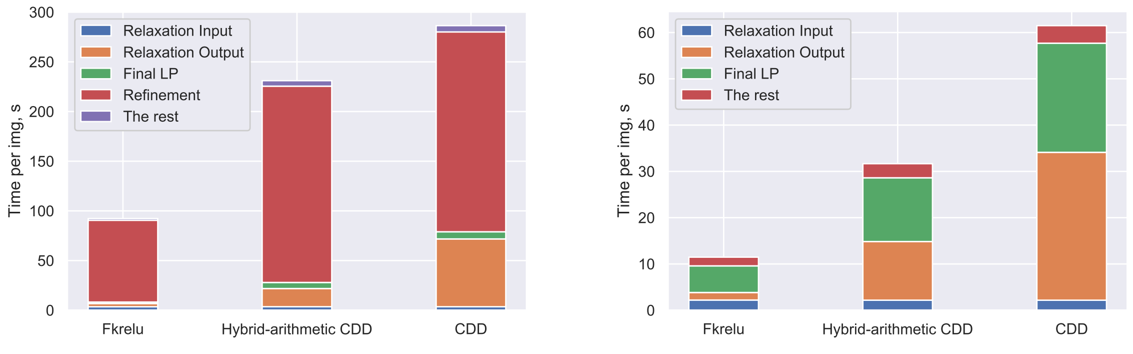
Table 6.4: Number of verified adversarial regions and runtime of our work vs *kPoly*

Dataset	Model	#correct	ϵ	*kPoly*[12]		Our work	
				verified(#)	time(s)	verified(#)	time(s)
MNIST	6×100	99	0.026	47	342	56	121
	9×100	97	0.026	41	321	46	272
	6×200	99	0.015	61	190	75	326
	9×200	97	0.015	56	458	66	591
	ConvSmall	100	0.12	35	483	44	18
	ConvBig	95	0.3	76	36	77	80
CIFAR10	ConvSmall	70	2/255	38	103	38	21
	ConvBig	65	2/255	44	218	44	262

$n = 100$, and $n = 200$, which corresponds to ≈ 1000 and ≈ 5000 groups respectively - with the original implementation computing relaxation for such number of groups is very expensive.

We note that the runtime can differ significantly for different networks. For small networks, for example computation of constraints for relaxation takes only a small part of the overall runtime (Fig. 6.3, 6.4b) and thus significant part of the runtime is spent on computing relaxations. There, the Fkrelu algorithm brings the maximum benefit and allows us to significantly speedup certification.

Although the speedup of the Hybrid-arithmetic CDD version computed in isolation on one thread is ≈ 10 times faster than the original CDD-based implementation, the speedup is smaller during actual network verification. Our hypothesis that it happens because, during network verification, 20 relaxation computations run in parallel on different threads. Because the convex hull algorithm can have a big memory usage due to potentially large intermediate representation and stored adjacency information between vertexes [8]. When 20 computations are run in parallel, memory throughput contributes to the bottleneck. Note that the overall part of the runtime spent on computing relaxations is *significantly* smaller when using the Fkrelu algorithm.

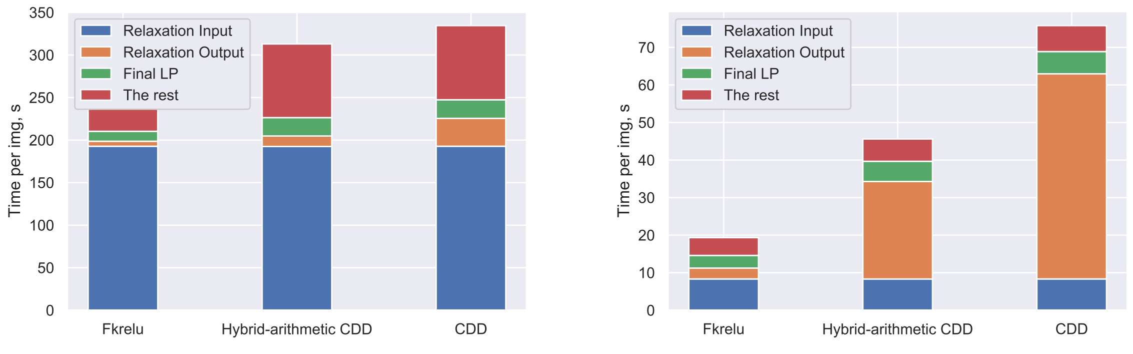


(a) 6x100 network

(b) ConvSmall network

Figure 6.3: Distribution of the runtime of our method for network verification on Mnist dataset computed on 10 images

However, on big networks computing, the input constraints for relaxations is a bottleneck (Fig. 6.4a). Computation of input constraints relies on DeepPoly, and if the number of neurons is large, this computation can be expensive. This, in return, makes the impact of relaxation computation diminished on the overall runtime. However, there is ongoing work for porting DeepPoly on GPU and adding support for computing input constraints for relaxations. The GPU version of DeepPoly would lead to a significant speedup of computation of the input constraints, and then using Fkrelu instead of exact relaxation can lead to a significant speedup. Another bottleneck for networks with large number of neurons is solving the final linear program for proving the desired property. Here we rely on Gurobi [9] LP solver and it is unlikely that this part of the runtime can be significantly improved.



(a) Cifar ConvBig network

(b) Cifar ConvSmall network

Figure 6.4: Distribution of the runtime of our method for network verification on Cifar10 dataset computed on 10 images

During the verification of fully-connected networks, where we do refinement, the biggest part of the runtime is spent on refining the neuron bounds. Unexpectedly, using Fkrelu leads to refinement procedure is significantly faster than using exact relaxation (Fig. 6.3a). The reason behind that is that although Fkrelu relaxation for the case $k = 3$ on average has volume only by a few percent greater than that of exact relaxation, for the case $k = 3$, it produces roughly a third of constraints compared to the exact algorithm. To refine neuron bounds, the linear program that contains constraints generated by relaxations [14]. When using the Fkrelu system, the number of constraints is smaller, so the LP is solved faster.

Counter-intuitively, for the same reason using refinement based on Fkrelu constraints can also be more *precise*. The LP for refining the neurons is run with a fixed time limit. Thus the refinement could succeed within a time limit in the smaller system of Fkrelu constraints and fail in a system with many constraints generated with the exact algorithm.

It is important to note that on networks with adversarial training, the joint computing joint relaxation of ReLU gives significantly less improvement compared to the networks trained without adversarial training. We hypothesize that it is due to adversarial training, making neurons relatively independent from each other, and thus the relational constraints between them are less tight.

7

Future work

The idea of passing the convex barrier with joint relaxation of neurons raises a big number of questions. In this chapter, we propose some ideas for future research.

7.1 Improved DeepPoly bounds

DeepPoly abstract domain associates one upper and one lower relational bound for every neuron. Currently, the upper and lower bound for relaxation $y_1 = ReLU(x_1)$ are chosen to minimize the area in (x_1, y_1) plane. These bounds however can be suboptimal when considering neurons jointly. For example, given $y_1 = ReLU(x_1)$ and $y_2 = ReLU(x_2)$ and relational constraints between x_1 and x_2, the optimal lower and upper bound for y_1 and y_2 in space (x_1, x_2, y_1, y_2) might not be the triangle bounds. Selection of lower and upper bounds that are optimal *jointly* poses an interesting optimization problem and can improve precision of DeepPoly.

7.2 Improved DeepPoly backsubstition

The cornerstone of the verification methods based on neurons' computing joint relaxation depends heavily on the input constraints. Constraints are computed through DeepPoly backsubstition. However, when neurons are considered jointly - the backsubstitution can be improved. For example, during backsubstitution, one can consider relations between pairs of neurons computed through convex relaxation.

Bibliography

ERAN.

pycddlib, 2018.

D. Avis and D. Bremner. How good are convex hull algorithms? *Proceedings of the eleventh annual symposium on Computational geometry - SCG 95*, 1995.

D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. *Automated Technology for Verification and Analysis Lecture Notes in Computer Science*, page 269–286, 2017.

F. Fernandez and P. Quinton. Extension of chernikova's algorithm for solving general mixed linear programming problems. 01 1988.

K. Fukuda. *Lecture: Polyhedral Computation*. 2015.

K. Fukuda and A. Prodon. Double description method revisited. *Combinatorics and Computer Science Lecture Notes in Computer Science*, page 91–111, 1996.

L. Gurobi Optimization. Gurobi optimizer reference manual, 2020.

A. Maréchal and M. Périn. Efficient elimination of redundancies in polyhedra by raytracing. *Lecture Notes in Computer Science Verification, Model Checking, and Abstract Interpretation*, page 367–385, 2017.

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. *Programming Languages and Systems Lecture Notes in Computer Science*, page 3–17, 2004.

G. Singh, R. Ganvir, M. Püschel, and M. Vechev. Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems 32*, pages 15098–15109. 2019.

G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

G. Singh, T. Gehr, M. Püschel, and M. Vechev. Robustness certification with refinement. In *International Conference on Learning Representations*, 2019.

H. R. Tiwary. On the hardness of computing intersection, union and minkowski sum of polytopes. *Discrete & Computational Geometry*, 40(3):469–479, 2008.

H. L. Verge. A note on chernikova's algorithm. Technical report, 1994.

P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Na-*

ture Methods, 17:261–272, 2020.