Deep ReLU Programming

Peter Hinz

December 1, 2020

Abstract

We present the class of deep ReLU programming optimization problems. These are minimization problems of feed-forward ReLU neural networks with a real-valued output. Since every linear programming problem can be reformulated as a deep ReLU programming problem the latter are a generalization. We develop a mathematical theory for their analysis and propose an efficient iterative solver algorithm. It is based on gradient descent with a variable step-size that is automatically determined maximally such that no neuron activation changes. Similar to the simplex algorithm in linear programming, this guarantees a monotonic decrease of the target function in every step.

1 Introduction

Linear programming plays an important role in the field of mathematical optimization and is used in operations research as a main tool to solve many large-scale real-world problems. A linear programming problem is an optimization problem restricted by linear conditions. In its canonical form, for $n \in \mathbb{N}$, $c \in \mathbb{R}^{n_0}$, a matrix $A \in \mathbb{R}^{n \times n_0}$ and a vector $b \in \mathbb{R}^n$ the goal is to

$$\min \langle x, c \rangle \text{ such that } x \in \mathbb{R}^{n_0}_{\geq 0} \text{ and } (Ax)_j \leq b_j \text{ for } j \in \{1, \ldots, n\}. \quad (1)$$

This means that linear programming aims to minimize an affine map over a convex set of feasible arguments that is determined by a number of linear inequalities. In this work we present a generalization of linear programming, namely deep ReLU programming which aims to minimize real-valued feed-forward ReLU neural nets locally.

1.1 Feed-forward ReLU neural nets

A real-valued ReLU feed-forward neural network $f : \mathbb{R}^{n_0} \to \mathbb{R}$ with input layer dimension $n_0 \in \mathbb{N}$ is a composition of $L \in \mathbb{N}$ layer transition functions of possibly different output dimensions $n_1, \ldots, n_{L+1} \in \mathbb{N}_+$ with $n_{L+1} = 1$ and a subsequent affine mapping, i.e.

$$f : \begin{cases} \mathbb{R}^{n_0} & \to \mathbb{R} \\ x & \mapsto W_{L+1} g^{(L)} \circ \cdots \circ g^{(1)}(x) + b_{L+1} \end{cases}, \quad (2)$$
where $g^{(l)} : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}, x \mapsto \text{ReLU}(W_l x + b_l)$ with the activation function function $\text{ReLU} : \mathbb{R} \to \mathbb{R}, x \mapsto \max(x, 0)$, matrices $W_l \in \mathbb{R}^{n_l \times n_{l-1}}$ and bias vectors $b_l \in \mathbb{R}^{n_l}, l \in \{1, \ldots, L\}$. Note that the dot notation "\cdot" denotes element-wise application of the ReLU function. The restriction $n_{L+1} = 1$ reflects the fact that we consider real-valued neural networks. We denote the class of all such functions by

$$F := \{ f : \mathbb{R}^{n_0} \to \mathbb{R} \mid f \text{ is of the form } (2) \}, \quad (3)$$

where the number of layers $L$ and their widths $n_0, \ldots, n_L$ are implicit. The functions $f \in F$ are piece-wise affine and continuous. They can be represented as a sum

$$f = \sum_{i=1}^{M} \mathbf{1}_{A_i}(x)\langle b_i, x_i \rangle x + c_i \quad (4)$$

with vectors $b_1, \ldots, b_{M+1} \in \mathbb{R}^{n_0}$ and real numbers $c_1, \ldots, c_M \in \mathbb{R}$ and a disjoint partition of the input space $\mathbb{R}^{n_0} = \bigcup_{i=1}^{M} A_i$ into convex sets $A_i, i \in \{1, \ldots, M\}$ each being the solution of linear inequalities. The works [1], [2] and [3] derive upper bounds on $M \in \mathbb{N}$ such that this representation is still possible for all $f \in F$, for example for $M \leq \prod_{i=1}^{L} \sum_{j=0}^{\min(n_0, \ldots, n_{i-1})} \binom{n_i}{j}$, see Section [3]

### 1.2 Deep ReLU programming

#### 1.2.1 Definition and relation to linear programming

Deep ReLU programming problems are minimization problems of functions $f \in F$. Some of the functions $f \in F$ are not bounded from below, i.e. attain arbitrarily small values and in general multiple local minima may exist. Hence, in contrast to linear programs which are convex optimization problems, deep ReLU programs are not necessarily convex.

Given a ReLU neural network $f \in F$, we define the deep ReLU programming problem as the optimization problem to find a local minimum of $f$. Compared to [1], this might seem as a completely different problem, but every linear program can be written as a deep ReLU program. In a linear program we can restrict to vertices as possible candidates for the minimization. Intuitively these are the intersection points, as depicted in Figure [1]. If it exists, the global minimum of the objective function $x \mapsto \langle x, c \rangle, c \in \mathbb{R}$ will be attained in one of the vertices, more precisely in one of those vertices that are contained in the feasible region. For a given linear program with fixed $n, n_0, c, A$ and $b$ as in equation [1] and a parameter $\alpha > 0$ we can construct the neural network $f_\alpha : \mathbb{R}^{n_0} \to \mathbb{R}$ by

$$f_\alpha(x) = \text{ReLU}(\langle c, x \rangle) - \text{ReLU}(\langle -c, x \rangle) + \alpha \sum_{i=1}^{n} \text{ReLU}(\langle A_i, x \rangle - b_i) + \alpha \sum_{i=1}^{n_0} \text{ReLU}(-x_i).$$

Indeed this is of the form (2) for $L = 1$ and appropriate matrices and bias vectors. Furthermore $f_\alpha(x) = \langle c, x \rangle$ whenever $x$ is in the feasible region, i.e. for $x \geq 0$ and $Ax \leq b$ element-wise. Now note that there are only finitely many vertices and for any vertex $x^* \in \mathbb{R}^{n_0}$ that
Figure 1: Qualitative illustration of the argument space of a linear program with \( n_0 = 2 \).
The vertices are depicted as dots and the feasible region is represented by the gray polygon.

is not contained in the feasible region, the function value of \( f_\alpha(x) \) can be made arbitrarily large by increasing \( \alpha \). Hence for \( \alpha \) large enough it is clear that the original linear program and the minimization problem of \( f_\alpha \) have the same solutions. This shows that deep ReLU programs are indeed a generalization of linear programs.

Moreover, by equation (4) minimizing the restriction \( f|_{A_i} \) of \( f \) onto the domain \( A_i \) for some \( i \in \{1, \ldots, M\} \) yields a linear programming problem since the restriction is affine, i.e. \( f|_{A_i}(x) = \langle x, a_i \rangle + c_i \) for some \( a_i \in \mathbb{R}^{n_0} \) and \( c_i \in \mathbb{R} \). In this setting \( A_i \), which is given by linear inequalities as noted above, takes the role of the feasible region over which the function \( x \mapsto \langle x, a_i \rangle = f|_{A_i}(x) - c_i \) shall be minimized. In this sense linear programming can be seen as a minimization problem of a ReLU feed-forward neural net for inputs restricted to one of its affine regions, and by dropping this condition and allowing to consider other affine regions of the neural net in the minimization problem we arrive at deep ReLU programming.

1.2.2 Solvers

In this paper we present an iterative algorithm applicable to functions \( f \in \mathcal{F} \) that are firstly regular and secondly have vertices. We will define these concepts in Section 2. For these functions \( f \), our algorithm a local minimum or detects that the \( f \) is not bounded from below after a finite number of steps. It is based on the ideas of the simplex algorithm from linear programming and of the gradient descent algorithm. The step-size is automatically determined and the value of objective function \( f \) is guaranteed to decrease in every step up to numerical uncertainty.

This is possible due to the nature of the objective function \( f \in \mathcal{F} \) that we want to minimize. As noted above, \( f \) is a piece-wise affine function. The standard gradient descent and similar algorithms do not use the property that the gradient is piece-wise constant because they are not specifically designed for this kind of functions. However, when restricting to such functions, a closer look at the problem reveals that one can relatively easily and computationally efficiently obtain the maximal step size such that the gradient of the neural net is
still identical when moving into a given direction. We exploit this fact and the structure of
the objective function to construct a number of algorithmic building blocks that are then
used to construct an iterative solver algorithm for deep ReLU programming problems. Simi-
larly to the simplex algorithm in linear programming, it traverses vertices and keeps track
of a local basis corresponding to the axis directions but in contrast to linear programming
problems, these axes do not correspond to edges of the feasible region. Instead they are
edges separating regions of different affine behaviour of the objective function.

The essential extension of our algorithm over the simplex algorithm is the firstly the ability to
change the considered affine region, thus allowing it to be used for deep ReLU programming,
and secondly to efficiently update the set of axes during this process. A careful analysis shows
that only one axis and the new gradient needs to be updated during this process of changing
the region such that the overall runtime for each iteration of our presented algorithm is only
\( O(\sum_{i=1}^{L} n_i n_{i-1} + n_0^2) \). Note that the number of weight and bias parameters of the considered
neural network is \( \sum_{i=1}^{L+1} n_i n_{i-1} \) and \( \sum_{i=1}^{L+1} n_i \) with output dimension \( n_{L+1} = 1 \) which is the
reason for the order \( O(\sum_{i=1}^{L} n_i n_{i-1}) \) in the runtime per step. But also the complexity term
\( O(n_0^2) \) is unavoidable for a simplex-like algorithm because the whole simplex tableau has
to be considered. In this sense, the complexity per step is optimal and the extension of
being able to change the considered affine region in our algorithm compared to simplex-like
iterations in one fixed feasible region of the objective function comes at zero cost concerning
the computational complexity per step.

1.3 Notation and Structure

We consider a vector \( a \in \mathbb{R}^n \), \( n \in \mathbb{N} \) also as a matrix with \( n \) rows and one column where
appropriate. Similarly the transpose \( a^T \) is treated as a matrix with one row and \( n \) columns.
Where appropriate, we define matrices block-wise based on smaller matrices and zero-filled
matrices such as \( C = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \) for \( A \in \mathbb{R}^{a \times b} \), \( B \in \mathbb{R}^{c \times d} \), \( a, b, c, d \in \mathbb{N}_{>0} \). In this case, \( C \)
would have \( a + c \) rows and \( b + d \) columns. For \( n \in \mathbb{N} \), the \( n \times n \) identity matrix is denoted by \( \text{Id}_n \)
and for \( a \in \mathbb{R}^n \) the diagonal matrix with values \( a \) on its diagonal is denoted by \( \text{diag}(a) \). The
dimension of a vector space \( V \) and the linear span of vectors \( v_1, \ldots, v_k \in V \) are denoted by
\( \text{dim}(V) \) and \( \text{span}(v_1, \ldots, v_k) \) respectively.

When composing functions \( \varphi^{(i)} \), \( i \geq 1 \) we use the convention that \( \varphi^{(l)} \circ \cdots \circ \varphi^{(1)}(x) \) means
\( \varphi^{(1)}(x) \) for \( l = 1 \) and \( x \) for \( l = 0 \). Furthermore for a univariate function \( \varphi : \mathbb{R} \to \mathbb{R} \) the dot “.”
denotes element-wise application. For example for \( x \in \mathbb{R}^{p_1} \times \cdots \times \mathbb{R}^{p_K} \), \( K, p_1, \ldots, p_K \in \mathbb{N} \),
write \( \varphi(x) \) for \( (\varphi(x_{1,1}), \ldots, \varphi(x_{1,p_1})), \ldots, (\varphi(x_{K,1}), \ldots, \varphi(x_{K,p_K})) \).

This paper is structured as follows: In Section 2 we analyze mathematical properties of
ReLU feed-forward neural networks and derive results needed in the subsequent sections.
We focus on iterative solvers for deep ReLU programming problems in Section 3 where we
first present algorithmic primitives which are then composed in our Deep ReLU Programming
(DRLP)-algorithm. Here we will make use of the previously derived mathematical results.
In Section 4 we give examples of deep ReLU programming problems. Section 5 compares
deep ReLU programming with linear programming and gradient descent-like procedures.
Finally, we summarize our findings in Section 6. The Appendices A and C contains proofs and details on our Julia implementation.

2 Mathematical Analysis

In this section we introduce the theoretical foundations that are needed in the subsequent sections. Recall that $f$ is a feed-forward neural network with ReLU activations with $L \in \mathbb{N}$ hidden layers and in particular a piece-wise affine function. We index the neurons by tuples $(l, j)$ where the $l \in \{1, \ldots L\}$ specifies the layer number and $j \in \{1, \ldots, n_l\}$ the neuron index within the $l$-th layer. The index set of all neurons is denoted by

$$\mathcal{I} = \{(l, j) \in \mathbb{N}^2 | l \in \{1, \ldots, L\}, j \in \{1, \ldots, n_l\}\}.$$  \hfill (5)

For our analysis we need to consider the activation of the neurons of the neural network $f$ at a given input $x \in \mathbb{R}^{n_0}$. To this end we define a function that computes the arguments of the ReLU activation functions of the individual neurons. More precisely we define the ReLU arguments by

$$A : \mathbb{R}^{n_0} \to \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_L}$$

$$x \mapsto (W_1x + b_1, W_2g^{(1)}(x) + b_2, \ldots, W_Lg^{(L-1)} \circ \cdots \circ g^{(1)}(x) + b_L)$$  \hfill (6)

using the notation from equation (2). This is a vector of vectors such that for every $(l, j) \in \mathcal{I}$ $A(x)_{lj}$ is the input of the activation function of the $j$-th neuron in the $l$-th layer.

2.1 Hyperplane and Activation Patterns

For every $(l, j) \in \mathcal{I}$ and every $x \in \mathbb{R}^{n_{l-1}}$, the value $(W_lx + b_l)_j$ can be positive, zero or negative. If the $j$-th row of $W_l$ is non-zero, $\{x \in \mathbb{R}^{n_{l-1}} | (W_lx + b_l)_j = 0\}$ defines a hyperplane in $\mathbb{R}^{n_{l-1}}$ and the three aforementioned possibilities correspond to $x$ lying on the positive half-space of this hyperplane, on the hyperplane itself or on its negative half-space. In order to describe the hyperplane relation of the ReLU arguments from equation (6) we define the hyperplane pattern by

$$H : \mathbb{R}^{n_0} \to \mathcal{H}$$

$$x \mapsto \text{sign}(A(x))$$  \hfill (7)

with $\mathcal{H} = \{-1, 0, 1\}^{n_1} \times \cdots \times \{-1, 0, 1\}^{n_L}$ and the signum function defined by $\text{sign}(x) = x/|x|$ for $x \neq 0$ and $\text{sign}(0) = 0$. The ReLU-function is the identity for arguments greater that 0, else it maps to 0. This is why we call a neuron active if its argument is positive and otherwise inactive. In particular, for $(l, j) \in \mathcal{I}$ the $j$-th neuron in the $l$-th layer is active if $H(x)_{lj} = 1$ and inactive if $H(x)_{lj} \in \{-1, 0\}$. We encode this information in the activation pattern defined by

$$S : \mathbb{R}^{n_0} \to \mathcal{S}$$

$$x \mapsto \text{ReLU}(H(x))$$  \hfill (8)
with $S = \{0, 1\}^{n_1} \times \cdots \times \{0, 1\}^{n_L}$. Furthermore we define the attained activation patterns by $\mathcal{S}_f = \{S(x) \mid x \in \mathbb{R}^{n_0}\}$. Note that $(l, j) \in \mathcal{I}$, the $j$-th neuron in the $l$-th layer is active if and only if $S(x)_{lj} = 1$.

**Example 1.** Let $L = 2$, $(n_0, n_1, n_2, n_3) = (2, 2, 1, 1)$, with weight matrices

$$W_1 = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, W_2 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, W_3 = (1), b_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, b_2 = -1, b_3 = 0.$$  

The neurons are indexed by $I = \{(1,1), (1,2), (2,2)\}$, the input space $\mathbb{R}^{n_0}$ is partitioned into $|\mathcal{S}_f| = 7$ regions and the attained activation patterns are $\mathcal{S}_f = \{0, 1\}^2 \times \{0, 1\} \setminus \{(0,1), 1\}$ as depicted in Figure 2. The ReLU arguments at $x = (x_1, x_2) \in \mathbb{R}^2$ are

$$A(x) = \begin{pmatrix} x_1 - x_2 \\ x_1 + x_2 \end{pmatrix}, \text{ReLU}(x_1 - x_2) + \text{ReLU}(x_1 + x_2) - 1 \in \mathbb{R}^2 \times \mathbb{R}.$$  

It follows that $H(x)_{1,1} = \text{sign}(x_1 - x_2)$ such that $\{x \in \mathbb{R}^2 \mid H(x)_{1,1} = 0\} = \{x \in \mathbb{R}^2 \mid x_1 = x_2\} = \mathbb{R}(1) \in \mathbb{R}^2$ is a line or more generally, a hyperplane in $\mathbb{R}^2$. Similarly, $\{x \in \mathbb{R}^2 \mid H(x)_{1,2} = 0\} = \mathbb{R}(-1) \in \mathbb{R}^2$. However $H(x)_{2,1} = 0$ if and only if $\text{ReLU}(x_1 - x_2) + \text{ReLU}(x_1 + x_2) = 1$ such that $\{x \in \mathbb{R}^2 \mid H(x)_{2,1} = 0\}$ is not a hyperplane due to the non-linearity induced by the ReLU-function. Instead it is a line with two kinks, see Figure 2.

![Figure 2: Qualitative illustration of the partition of the input space $\mathbb{R}^2$ into 7 regions with different activation patterns as explained in Example 1.](image)

### 2.2 Fixed Activations

The non-linearity of the feed-forward ReLU neural network $f$ stems from the fact that the activity of the neurons may vary for different inputs. For our theory, it is important to consider the quantities defined above for a fixed activity $s \in S$ of the neurons. For easier
notation, they will consistently be denoted with a tilde. For \( s \in \mathcal{S} \) we define the fixed activation network \( \tilde{f}_s \) by

\[
\tilde{f}_s : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_{L+1}} \quad \begin{cases} x & \mapsto W_{L+1} \tilde{g}_{s_L}^{(L)} \circ \cdots \circ \tilde{g}_{s_1}^{(1)}(x) + b_{L+1} \\
\end{cases}
\]

with

\[
\tilde{g}_{s_l}^{(l)} : \mathbb{R}^{n_{l-1}} \rightarrow \mathbb{R}^{n_l} \quad y \mapsto \text{diag}(s_l)(W_l y + b_l) \quad \text{for } l \in \{1, \ldots, L\}.
\]

Note that for all inputs \( x \in \mathbb{R}^{n_0} \) it holds that \( \tilde{f}_{S(x)}(x) = f(x) \) since each neuron’s ReLU-mapping can be represented by a multiplication of 1 if the neuron is active or by a multiplication of 0 if the neuron is inactive, which is precisely encoded in \( S(x) \).

Similar to equation (9) we can define the fixed activation ReLU arguments by

\[
\tilde{A}_s : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_L} \quad \begin{cases} x & \mapsto (W_1 x + b_1, W_2 \tilde{g}_{s_1}^{(1)}(x) + b_2, \ldots, W_L \tilde{g}_{s_{L-1}}^{(L-1)} \circ \cdots \circ \tilde{g}_{s_1}^{(1)}(x) + b_L) \\
\end{cases}
\]

for an activation pattern \( s \in \mathcal{S} \). The corresponding fixed activation hyperplane pattern is given by

\[
\tilde{H}_s : \mathbb{R}^{n_0} \rightarrow \mathcal{H} \quad \begin{cases} x & \mapsto \text{sign}(\tilde{A}_s(x)) \\
\end{cases}
\]

### 2.3 Compatibility

In equation (8) we arbitrarily defined a neuron to be inactive if its input is exactly 0. This is a borderline case and it could just as well be defined active for this input. To reflect this we introduce the concept of compatibility. We say that a hyperplane pattern \( h \in \mathcal{H} \) and an activation pattern \( s \in \mathcal{S} \) are compatible if

\[
\forall (l,j) \in \mathcal{I} \quad h_{ij}(s_{ij} - 0.5) \geq 0.
\]

In other words \( s \) and \( h \) are compatible if every neuron that is considered active has a non-negative input and every neuron that is considered inactive has a non-positive input. This seemingly simple concept will play an important role in our theory. For convenience we define

\[
\mathcal{S}^C(x) = \{ s \in \mathcal{S} \mid s \text{ is compatible with } H(x) \},
\]

\[
\tilde{\mathcal{S}}^C(x) = \{ s \in \mathcal{S} \mid s \text{ is compatible with } \tilde{H}_s(x) \}
\]

and say that \( s \in \mathcal{S} \) is compatible with \( x \) if it is compatible with \( H(x) \), i.e. if \( s \in \mathcal{S}^C(x) \). The first quantity \( \mathcal{S}^C(x) \) above is simply the set of all activation patterns that are compatible with the hyperplane pattern \( H(x) \) at the input \( x \). The second quantity \( \tilde{\mathcal{S}}^C(x) \) is a bit more involved because \( s \) is appearing twice in the set condition. It is the set of all activation patterns that are compatible with the their induced fixed activation hyperplane pattern. Luckily, both quantities are always equal as the following theorem states.
**Theorem 2.** For \( x \in \mathbb{R}^{n_0} \) the following are true:

1. \( \mathcal{S}^C(x) = \tilde{\mathcal{S}}^C(x) \)
2. For \( s \in \mathcal{S}^C(x) \) it holds that \( A(x) = \tilde{A}_s(x) \).

The above theorem is an important tool for the subsequent sections. Below we present two immediate consequences.

**Corollary 3.** For all \( x \in \mathbb{R}^{n_0} \) and compatible activation patterns \( s \in \mathcal{S}^C(x) \), it holds that \( \tilde{f}_s(x) = f(x) \).

*Proof.* This follows from Theorem 2 because \( \tilde{f}_s(x) = W_{L+1} \text{ReLU}(\tilde{A}_s(x)_L) + b_{L+1} \) and \( f(x) = W_{L+1} \text{ReLU}(A(x)_L) + b_{L+1} \). \( \square \)

**Corollary 4.** For \( x \in \mathbb{R}^{n_0} \) and a compatible activation pattern \( s \in \mathcal{S}^C(x) \), it holds that
\[
\{ y \in \mathbb{R}^{n_0} \mid H(y) = H(x) \} = \{ y \in \mathbb{R}^{n_0} \mid \tilde{H}_s(y) = \tilde{H}_s(x) \}.
\]

*Proof.* Let \( y \in \mathbb{R}^{n_0} \) with \( H(y) = H(x) \). Then \( H(y) \) and \( H(x) \) are compatible with \( s \) such that Theorem 2 implies that \( \tilde{H}_s(y) = \text{sign}(\tilde{A}_s(y)) = \text{sign}(A(y)) = H(y) \) and similarly \( \tilde{H}_s(x) = H(x) \). In particular \( \tilde{H}_s(y) = \tilde{H}_s(x) \). For the other direction assume \( y \in \mathbb{R}^{n_0} \) with \( \tilde{H}_s(y) = \tilde{H}_s(x) \). By assumption and again Theorem 2 it holds that \( s \in \tilde{\mathcal{S}}^C(x) \), i.e. \( s \) is compatible with \( \tilde{H}_s(x) \) and hence also with \( \tilde{H}_s(y) \) such that \( s \in \tilde{\mathcal{S}}^C(y) \). Like above, it follows from the same theorem that \( H(y) = \tilde{H}_s(y) = \tilde{H}_s(x) = H(x) \). \( \square \)

The algorithm we describe in Section 3 requires information about active neurons \( s \in \mathcal{S} \) and a current position \( x \in \mathbb{R}^{n_0} \) in each iteration step. Simply put, it sees the neural network as if its activation was \( s \) at input \( x \) whether or not this is actually true. In this context the first statement “\( \mathcal{S}^C(x) = \tilde{\mathcal{S}}^C(x) \)” of Theorem 2 states that we can tell whether \( s \in \mathcal{S}^C(x) \) just by checking whether \( \tilde{H}_s(x) \) is compatible with \( s \). This is useful since we can conclude statements about the compatibility of \( s \) with the true hyperplane pattern \( H(x) \) by checking for compatibility of \( s \) with the hyperplane pattern \( \tilde{H}_s(x) \) that is induced by the assumed choice of \( s \) without actually knowing the true hyperplane pattern \( H(x) \).

This is an example of how properties about the true neural network can be obtained from the one that is obtained by fixing the activations to some specific \( s \in \mathcal{S} \). Corollaries 3 and 4 are further examples. Results of this kind are important to establish a connection between the true network \( f \) and our algorithm where we assume a specific activation \( s \in \mathcal{S} \) in every iteration to describe the activity of the neurons.

### 2.4 Local behaviour

In this subsection we want to define concepts and derive results about the local behaviour of feed-forward neural networks with ReLU activation functions. Note that for \( x \in \mathbb{R}^{n_0} \) and \( \varepsilon > 0 \) the \( \varepsilon \)-ball around \( x \) is \( B_\varepsilon(x) = \{ y \in \mathbb{R}^{n_0} \mid \| x - y \|_2 < \varepsilon \} \).
2.4.1 Affected Indices

At every input \( x \in \mathbb{R}^n \) the hyperplane \( H(y) \) pattern might or might not be constant for \( y \) in an infinitely small neighbourhood around \( x \). If it is not constant, there are one or more indices \((l, j) \in I\) describing the positions of corresponding neurons whose activation is not constant. We call these indices the **affected indices** and denote them by

\[
C(x) = \{(l, j) \in I \mid \forall \varepsilon > 0 \mid \{|H(y)_{l,j}| y \in B_\varepsilon(x)\} > 1\} \subset I,
\]

where \( P \) denotes the power set. Similarly we define the **fixed activation affected indices** of \( x \in \mathbb{R}^n \) by

\[
\tilde{C}_s(x) = \{(l, j) \in I \mid \forall \varepsilon > 0 \mid \{\tilde{H}_s(y)_{l,j}| y \in B_\varepsilon(x)\} > 1\} \quad \text{for } s \in S.
\]

**Lemma 5.** For \( x \in \mathbb{R}^n \) and all \((l, j) \in C(x)\), it holds that \( H(x)_{l,j} = 0 \) and for all \( s \in S \) and \((l, j) \in \tilde{C}_s(x)\) it holds that \( \tilde{H}_s(x)_{l,j} = 0 \).

Proof. If \( H(x)_{l,j} = c \in \{-1, 1\} \) for some \( x \in \mathbb{R}^n \) and \((l, j) \in I\), then by continuity of \( y \mapsto A(y)_{l,j} \) for \( \varepsilon \) small enough \( \{|H(y)_{l,j}| y \in B_\varepsilon\}| = |\{c\}| = 1 \neq 1 \) such that \((l, j) \notin C(x)\). The case for \( \tilde{H}_s \) is along similar lines.

Unfortunately, it does not hold that \( C(x) = \tilde{C}_s(x) \) for all \( x \in \mathbb{R}^n \) and compatible \( s \in S^C(x) \) as the following two examples demonstrate.

**Example 6.** Let \( L = 2, (n_0, n_1, n_2, n_3) = (1, 2, 1, 1), b_1 = (0, 0)^T, W_1 = (1, 1)^T, W_2 = (1, -1), b_2 = 0, W_3 = (1), b_3 = 0 \). It holds that

\[
C(0) = \{(1, 1), (1, 2)\} \subset I = \{(1, 1), (1, 2), (2, 1)\},
\]

\[
S^C(0) = \left\{ s \in \{0, 1\}^2 \times \{0, 1\} \mid s \text{ is compatible with } H(0) = \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right) \right\} = \{0, 1\}^2 \times \{0, 1\},
\]

\[
\tilde{A}_{(0,1)}(x) = \left(\begin{smallmatrix} x \\ -x \end{smallmatrix}\right),
\]

\[
\tilde{C}_{(0,1)}(0) = \{(1, 1), (1, 2)\}, (2, 1) = I.
\]

In particular \( C(0) \subset \tilde{C}_s(0) \) for \( s = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right) \in S^C(0) \).

**Example 7.** Let \( L = 2, (n_0, n_1, n_2, n_3) = (1, 2, 1, 1), b_1 = (0, 0)^T, W_1 = (1, -1)^T, W_2 = (1, -1), b_2 = 0, W_3 = (1), b_3 = 0 \). It holds that

\[
C(0) = \{(1, 1), (1, 2), (2, 2)\} = I
\]

\[
S^C(0) = \left\{ s \in \{0, 1\}^2 \times \{0, 1\} \mid s \text{ is compatible with } H(0) = \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right) \right\} = \{0, 1\}^2 \times \{0, 1\},
\]

\[
\tilde{A}_{(0,1)}(x) = \left(\begin{smallmatrix} x \\ 0 \end{smallmatrix}\right),
\]

\[
\tilde{C}_{(0,1)}(0) = \{(1, 1), (1, 2)\} \subset I.
\]

In particular \( C(0) \supset \tilde{C}_s(0) \) for \( s = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right) \in S^C(0) \).
The examples 6 and 7 show that neither \( C(x) \subset \tilde{C}_s(x) \) always holds for any \( x \in \mathbb{R}^{n_0} \) \( s \in \mathcal{S}^C(x) \) nor does \( C(x) \supset \tilde{C}_s(x) \). We will therefore introduce an additional concept. For \( x \in \mathbb{R}^{n_0} \) we define the possibly affected indices by
\[
C^*(x) = \bigcup_{s \in \mathcal{S}^C(x)} \tilde{C}_s(x).
\] (18)

Lemma 25 in the appendix shows that \( C^*(x) \) is always a superset of both \( C(x) \) and \( \tilde{C}_s(x) \) for \( s \in \mathcal{S}^C(x) \).

### 2.4.2 Hyperplane Pattern Preservation

We now want to focus on how the hyperplane pattern changes locally. For \( x \in \mathbb{R}^{n_0} \) we define its hyperplane kernel by
\[
\text{Ker}^H(x) = \{ y \in \mathbb{R}^{n_0} \mid \exists \varepsilon^* > 0 \ \forall t \in (-\varepsilon^*, \varepsilon^*) \ H(x + ty) = H(x) \}.
\] (19)

This is the set of directions in which the hyperplane pattern does not change. Similarly, for a specific activation pattern \( s \in \mathcal{S} \) we define the fixed activation hyperplane kernel by
\[
\text{Ker}^H_s(x) = \{ y \in \mathbb{R}^{n_0} \mid \exists \varepsilon^* > 0 \ \forall t \in (-\varepsilon^*, \varepsilon^*) \ \tilde{H}_s(x + ty) = \tilde{H}_s(x) \}.
\] (20)

These two quantities are always equal for activation patterns \( s \) compatible with \( x \) as the following lemma shows.

**Lemma 8.** For all \( x \in \mathbb{R}^{n_0} \) and \( s \in \mathcal{S}^C(x) \) it holds that \( \text{Ker}^H(x) = \text{Ker}^H_s(x) \)

**Proof.** This follows immediately from Corollary 4.

The above result states that we can find directions in which the hyperplane pattern of the neural network \( f \) does not change locally by only considering the affine functions \( \tilde{A}_s(x)_{lj}, \ (l, j) \in \mathcal{I} \) for some compatible \( s \in \mathcal{S}^C(x) \). In other words, for \( s \in \mathcal{S}(x) \) the quantity \( \text{Ker}^H_s(x) \) is a universal property and not specific to \( s \). The following reformulation makes this result more useful.

**Lemma 9.** For all \( x \in \mathbb{R}^{n_0} \) and \( s \in \mathcal{S} \) it holds that
\[
\text{Ker}^H_s(x) = \bigcap_{(l, j) \in \tilde{C}_s(x)} \left\{ v \in \mathbb{R}^{n_0} \mid \tilde{A}_s(x + v)_{lj} = 0 \right\}.
\] (21)

This lemma shows that for \( s \in \mathcal{S} \) the fixed activation hyperplane kernel is a subspace of \( \mathbb{R}^{n_0} \). In connection with Lemma 8 it states that for every \( x \in \mathbb{R}^{n_0} \) the local hyperplane kernel \( \text{Ker}^H(x) \) is the solution set of all \( y \in \mathbb{R}^{n_0} \) satisfying the linear equation system \( \tilde{A}_s(x + y)_{lj} = 0 \) for all \( (l, j) \in \tilde{C}_s(x) \). It is remarkable that this holds irrespective of the choice of \( s \in \mathcal{S}^C(x) \). Despite the fact that firstly for different \( s, s' \in \mathcal{S}^C(x) \) the fixed activation affected indices might not have the same cardinality \( |\tilde{C}_s(x)| \neq |\tilde{C}_{s'}(x)| \) and secondly, the affine functions \( \tilde{A}_x \) and \( \tilde{A}_{s'} \) can differ, the fixed activation hyperplane kernels are the same, i.e. \( \text{Ker}^H_s(x) = \text{Ker}^H_{s'}(x) \).
Lemma 10. For $x \in \mathbb{R}^{n_0}$ and $S^C(x)$ there exists an $\varepsilon^* > 0$ such that

$$B_{\varepsilon^*}(0) \cap \{v \in \mathbb{R}^{n_0} | H(x + v) = H(x)\} = B_{\varepsilon^*}(0) \cap \text{Ker}^H(x)$$

(22)

The above lemma states that locally around $x \in \mathbb{R}^{n_0}$ the solutions $x' \in \mathbb{R}^{n_0}$ to $H(x') = H(x)$ are precisely the affine subspace $x + \text{Ker}^H(x)$. We call its dimension the degrees of freedom

$$\text{df}^H(x) = \dim (\text{Ker}^H(x)).$$

(23)

Informally, this is the dimension of the directions which do not change the sign of any neuron output before the ReLU activation function is applied. We have shown that we can compute this dimension by picking an arbitrary $s \in S^C(x)$ and considering the kernel of linear equation system (21). Note that by Lemma 9 we have the following lower bound:

Lemma 11. For $x \in \mathbb{R}^{n_0}$ and $s \in S^C(x)$ it holds that $\text{df}^H(x) \geq n_0 - |C_s(x)|$.

2.5 Normal vectors

For $s \in \mathcal{S}$ the fixed activation neural network $\tilde{f}_s$ is an affine function. The same holds for the fixed activation ReLU arguments $x \mapsto \tilde{A}_s(x)_{lj}$ for every $(l, j) \in \mathcal{I}$. By equation (11) it follows that

$$\tilde{A}_s(x)_{lj} = \langle x, \tilde{v}_{s,l,j} \rangle + \tilde{w}_{s,l,j}$$

(24)

with $\tilde{w}_{s,l,j} = b_{lj} + \left(\sum_{i=1}^{l-1} W_i \text{diag}(s_{i-1}) \cdots W_{i+1} \text{diag}(s_1) b_i\right)_j$ and

$$\tilde{v}_{s,l,j} = (W_l \text{diag}(s_{l-1}) W_{l-1} \cdots \text{diag}(s_1) W_1)_j.$$  

(25)

for $s \in \mathcal{S}$ and $(l, j) \in \mathcal{I}$. We call $(\tilde{v}_{s,l,j})_{(l,j) \in \mathcal{I}}$, the normal vectors for $s \in S^C(x)$. Figure 3 illustrates this concept. For $(l, j) \in \mathcal{I}$, the set $\{x \in \mathbb{R}^2 | \tilde{A}_s(x)_{lj} = 0\}$ is a $(n_0 - 1)$-dimensional hyperplane if and only if the normal vector $\tilde{v}_{s,l,j}$ is non-zero. In contrast, the version without a fixed activation $\{x \in \mathbb{R}^2 | A(x)_{lj} = 0\}$ is in general not a hyperplane as explained previously in example 1. Below we present two results about these normal vectors.

Figure 3: Qualitative illustration of normal vectors for $n_0 = 2$. 

11
Lemma 12. For $x \in \mathbb{R}^{n_0}$, $s \in \mathcal{S}(x)$ and $(l,j) \in \tilde{C}_s(x)$ and $v \in \mathbb{R}^{n_0}$ it holds that $\tilde{A}_s(x + v)_{lj} = \langle v, \tilde{v}_{s,l,j} \rangle$.

Proof. Equation (24) and Lemma 5 imply $\tilde{A}_s(x + v)_{lj} = \tilde{A}_s(x)_{lj} + \langle v, \tilde{v}_{s,l,j} \rangle$. □

Lemma 13. Let $x \in \mathbb{R}^{n_0}$ and $s \in \mathcal{S}$. For all $(l,j) \in \tilde{C}_s(x)$ it holds that $\tilde{v}_{s,l,j} \neq 0$.

Proof. By equation (17), $x \mapsto \tilde{A}_s(x)_{lj}$ cannot be constant. The statement now follows from equation (24). □

Figure 4: Illustration of the intuition behind oriented normal vectors for $n_0 = 2$. The points $x \in \mathbb{R}^{n_0}$ in the gray area $R_s \subset \mathbb{R}^2$ have constant hyperplane pattern $H(x)$ that is compatible with the activation pattern $s \in \mathcal{S}$. Let $(l_1, j_1), (l_2, j_2) \in \mathcal{I}$ and assume that $s_{l_1,j_1} = 0$ and $s_{l_2,j_2} = 1$ such that due the compatibility implies $\tilde{A}_s(x)_{l_1,j_1} \leq 0$ and $\tilde{A}_s(x)_{l_2,j_2} \geq 0$ for $x \in R_s$. Equation (24) implies that intuitively, the normal vector $\tilde{v}_{s,l_1,j_1}$ of the oriented hyperplane $\tilde{A}_s(x)_{l_1,j_1} = 0$ points away from the gray region $R_s$ because all $x \in R_s$ are on its negative half-space. Similarly the normal vector $\tilde{v}_{s,l_2,j_2}$ of the oriented hyperplane $\tilde{A}_s(x)_{l_2,j_2} = 0$ points in the direction where $R_2$ lies. The oriented normal vectors always do so. This is useful for the construction of feasible axes in Section 2.7.

For $s \in \mathcal{S}$ and $(l,j) \in \mathcal{I}$ we define the oriented normal vector $\tilde{u}_{s,l,j} \in \mathbb{R}^{n_0}$ by

$$\tilde{u}_{s,l,j} = \begin{cases} \tilde{v}_{s,l,j} & \text{if } s_{lj} = 1 \\ -\tilde{v}_{s,l,j} & \text{if } s_{lj} = 0 \end{cases} \quad (26)$$

Figure 4 illustrates the intuition behind this concept.

2.6 Regularity

We call $x \in \mathbb{R}^{n_0}$ a vertex if $df^H(x) = 0$. Furthermore, $x \in \mathbb{R}^{n_0}$ is a regular point if $df^H(x) = n_0 - |C^*(x)|$. If both conditions hold, $x$ is a regular vertex. Furthermore, we call the ReLU feed-forward neural network $f$ regular if all points $x \in \mathbb{R}^{n_0}$ are regular points. Examples 6 and 7 showed such that for a general $x \in \mathbb{R}^{n_0}$, neither $C^*(x) \subset \tilde{C}_s(x)$ nor $C^*(x) \supset \tilde{C}_s(x)$. However, if $x$ is regular equality holds.
Lemma 14. Let \( x \in \mathbb{R}^{n_0} \) be a regular point. Then for all \( s \in \mathcal{S}^C(x) \) it holds that \( C^*(x) = \tilde{C}_s(x) \)

The above lemma implies that for \( x \in \mathbb{R}^{n_0} \) all \( \tilde{C}_s(x), s \in \mathcal{S}^C(x) \) are equal, i.e. the choice of \( s \in \mathcal{S}^C(x) \) is irrelevant.

Lemma 15. For every regular point \( x \in \mathbb{R}^{n_0} \) and \( s \in \mathcal{S}^C(x) \) the normal vectors \( (\tilde{v}_{s,l,j})_{(l,j) \in \mathcal{C}_s(x)} \) are linearly independent.

Below we present a probabilistic result which guarantees that a random neural network \( f \) is regular almost surely for specific distributional assumptions on its parameters.

Theorem 16. Assume the \( \sum_{i=2}^{L+1} n_i n_{i-1} \) weight parameters of the matrices \( W_1, \ldots, W_L \) and the \( \sum_{i=2}^{L+1} n_i \) bias parameters of the vectors \( b_1, \ldots, b_L \) of the neural network \( f \) are sampled from a distribution such that conditionally on the weight parameters, the bias parameters are independent with a conditional marginal distribution that assigns probability zero to all finite sets. Then

\[ \mathbb{P}[f \text{ is regular}] = 1, \]

i.e. almost surely all arguments \( x \in \mathbb{R}^{n_0} \) of \( f \) are regular points.

The condition is satisfied for example if all parameters of the neural network are independent random variables each with a continuous distribution or if the weight matrices are deterministic and the bias parameters are independent random variables with a continuous distribution. In such cases, the above result states that all points \( x \in \mathbb{R}^{n_0} \) in the input space are regular points.

2.7 Feasible directions

For each input value \( x \in \mathbb{R}^{n_0} \) and compatible \( s \in \mathcal{S}^C(x) \) we consider directions \( v \in \mathbb{R} \) such that locally \( s \) is still compatible for arguments in direction \( v \) from \( x \) i.e. such that for some \( \varepsilon > 0 \), \( s \in \mathcal{S}^C(x + \varepsilon v) \). We call the class of such directions feasible directions, formally defined by

\[ \tilde{D}_s(x) := \{ v \in \mathbb{R}^{n_0} \mid \exists \varepsilon > 0 \ s \in \mathcal{S}^C(x + \varepsilon v) \}. \]  

(27)

The following lemma provides a different formulation in terms of the fixed activation ReLU arguments \( \tilde{A}_s \).

Lemma 17. For \( x \in \mathbb{R}^{n_0} \) and \( s \in \mathcal{S}^C(x) \) it holds that

\[ \tilde{D}_s(x) = \{ v \in \mathbb{R}^{n_0} \mid \forall (l,j) \in \mathcal{C}_s(x) \quad \tilde{A}_s(x + v)_{lj} (s_{lj} - 0.5) \geq 0 \}. \]  

(28)

In particular for each \( x \in \mathbb{R}^{n_0} \) and \( s \in \mathcal{S}^C(x) \), the feasible directions form a convex cone.
2.7.1 Feasible axes and region separating axes for regular vertices

In the special case where $x \in \mathbb{R}^n_0$ is a regular vertex, the feasible directions $\tilde{D}_s(x)$ are the positive span of $|C^*(x)|$ linear independent vectors as we will show below. By Lemmas 14 and 15, the oriented normal vectors $(\tilde{u}_{s,l,j})_{(l,j) \in C^*(x)}$ are linearly independent. In particular, when put as columns in a matrix, the resulting square matrix $A \in \mathbb{R}^{n_0 \times n_0}$ has an inverse $A^{-1}$ with rows denoted by $\tilde{a}_{x,s,l,j} \in \mathbb{R}^{n_0}$, $(l,j) \in C^*(x)$ satisfying

$$\langle \tilde{a}_{x,s,l,j}, \tilde{u}_{s,l',j'} \rangle = \begin{cases} 1 & \text{if } (l,j) = (l',j') \\ 0 & \text{if } (l,j) \neq (l',j') \end{cases} \quad \text{for } (l,j), (l',j') \in C^*(x) \quad (29)$$

For convenience we define the set of feasible axes $\tilde{A}_s(x)$ by

$$\tilde{A}_s(x) := \left\{ \tilde{a}_{s,x,l,j} \mid (l,j) \in \tilde{C}_s(x) \right\}. \quad (30)$$

We now show that indeed the feasible directions are the positive span of the feasible axes.

**Lemma 18.** Let $x \in \mathbb{R}^{n_0}$ be a regular vertex and $s \in S^C(x)$. For any $v \in \mathbb{R}^{n_0}$ it holds that $v \in \tilde{D}_s(x)$ if and only if there exist non-negative coefficients $\alpha_{l,j}$, $(l,j) \in \tilde{C}_s(x)$ such that $v = \sum_{(l,j) \in \tilde{C}_s(x)} \alpha_{l,j} \tilde{a}_{s,x,l,j}$.

**Proof.** First note that $v = \sum_{(l,j) \in \tilde{C}_s(x)} \alpha_{l,j} \tilde{a}_{s,x,l,j}$ for possibly negative coefficients $\alpha_{l,j}$, $(l,j) \in \tilde{C}_s(x)$ by Lemma 15. We use the formulation in equation (28). By assumption and Lemma 12 it holds that

$$v \in \tilde{D}_s(x) \iff \forall (l,j) \in \tilde{C}_s(x) \quad \langle v, \tilde{a}_{s,l,j} \rangle \geq 0$$

$$\iff \forall (l,j) \in \tilde{C}_s(x) \quad \langle v, \tilde{u}_{s,l,j} \rangle \geq 0$$

$$\iff \forall (l,j) \in \tilde{C}_s(x) \quad \sum_{(l',j') \in \tilde{C}_s(x)} \alpha_{l',j'} \langle \tilde{a}_{s,x,l',j'}, \tilde{a}_{s,l,j} \rangle \geq 0$$

By equation (29), the latter condition evaluates to $\alpha_{l,j} \geq 0$ for $(l,j) \in \tilde{C}_s(x)$.

The following result shows that different compatible signatures share the same axes at indices with the same activations.

**Theorem 19.** Let $x \in \mathbb{R}^{n_0}$ be a regular vertex, $(l,j) \in C^*(x)$ and $s, s' \in S^C(x)$ with $s_{l,j} = s'_{l,j} =: r \in \{0,1\}$. It holds that $\tilde{a}_{s,x,l,j} = \tilde{a}_{s',x,l,j}$.

The above theorem is an important result that we use in Section 3.1.8 to compute the axes $\tilde{A}_s(x)$ for a new activation pattern $s' \in S^C(x)$ when we already know the axes $\tilde{A}_s(x)$ for another activation pattern $s \in S^C(x)$ that only differs at one specific neuron position $(l^*, j^*) \in I$ from $s'$. By the above result, in this case all but one axis are the same. Furthermore, the theorem allows us to define a set

$$\mathcal{A}(x) = \{ a_{r,x,l,j} \in \mathbb{R}^{n_0} \mid r \in \{0,1\}, (l,j) \in C^*(x) \}. \quad (31)$$
of $2n_0$ different region separating axes satisfying
\[ a_{x,s,l,j} = \tilde{a}_{x,s,l,j} \quad \text{for} \ (l,j) \in C^*(x), \ s \in \mathcal{S}^C(x). \] (32)

Note that feasible axes $\tilde{A}_s(x)$ are a concept specific to $s \in \mathcal{S}^C(x)$ whereas the region separating axes $A(x)$ are not.

**Example 20.** Let $L = 2$, $(n_0, n_1, n_2, n_3) = (2, 2, 1, 1)$, $b_1 = (0,0)^T$, $W_1 = \begin{pmatrix} 0 & 1 \end{pmatrix}$, $W_2 = (1,-1)$, $b_2 = -1$, $W_3 = (1)$, $b_3 = 0$. In this case it holds that
\[ h_1(x_1, x_2) = (\text{ReLU}(x_1), \text{ReLU}(x_2)), \] (33)
\[ A(x)_{2,1} = \text{ReLU}(x_1) - \text{ReLU}(x_2) - 1. \] (34)

In particular $A(x)_{2,1} = 0$ if and only if either $x_2 < 0$ and $x_1 = 1$ or $x_2 \geq 0$ and $x_1 = 1 + x_2$. This is the reason for the kink in Figure 5 at $x^* = (1,0)$. The corresponding possibly affected indices are $C^*(x^*) = \{(1,2), (2,1)\}$ and the region separating axes are given by $a_{x^*,0,1,2}(0,-1)$, $a_{x^*,1,1,2} = (1,1)$, $a_{x^*,0,2,1} = (1,1)$, $a_{x^*,1,2,1} = (1,0)$.

![Figure 5](image-url)

Figure 5: Illustration of the region separating axes. The black lines represent the sets $A(x_1, x_2)_{lj} = 0$ for $(l,j) \in \{(1,1), (1,2), (2,1)\}$. The tuples indicate the activation pattern of affine regions of the neural network. The arrows are the region separating axes at position $x^* = (1,0)$. The red region separating axes represent the feasible axes $\tilde{A}_s(x^*)$ for the activation pattern $s = ((1,1), 1)$ whereas the green axes correspond to $\tilde{A}_{s'}(x^*)$ for $s' = ((1,0), 0)$.

### 2.8 Local minima in regular networks

Gradient descent algorithms are usually based on a computed gradient at the current position in the argument space $\mathbb{R}^{n_0}$. However our considered function $f$ is piece-wise affine and therefore the gradient is constant on each such affine region and not well-defined at the boundaries of these regions. It solely depends on which neurons are active such that we define the gradient specific to the activation pattern $s \in \mathcal{S}$ by
\[ \nabla_s = (W_{L+1} \text{diag}(s_L) W_L \cdots \text{diag}(s_1) W_1)^T. \] (35)
With this definition it holds that
\[
\hat{f}_s(x) = \langle x, \nabla_s \rangle + \sum_{l=1}^{L+1} W_{l+1} \text{diag}(s_L) \cdots W_l \text{diag}(s) b_l
\]  
(36)

for \(x \in \mathbb{R}^{n_0}\) and a compatible activation pattern \(s \in S^C(x)\) by equation (9). In particular by Corollary 3 it follows that \(\nabla_s\) is indeed equal to the gradient \(\nabla f(x)\) at values \(x \in \mathbb{R}^{n_0}\) where it is well-defined and the activation pattern is \(S(x) = s\). Note that \(\nabla_s\) depends on the weight matrices \(W_1, \ldots, W_{L+1}\) but not on the bias vectors \(b_1, \ldots, b_{L+1}\) of the neural network \(f\). Below we present a sufficient and necessary condition when a regular vertex is a local minimum of \(f\)

**Proposition 21.** Let \(x \in \mathbb{R}^{n_0}\) be a regular vertex. Then
\[
\exists \varepsilon > 0 : f(x) = \inf \{ f(y) \mid y \in B_{\varepsilon}(x) \}
\]
\[\iff \forall a \in A(x) \exists s \in S^C(x), (l, j) \in \hat{C}_s(x) \text{ with } a = \tilde{a}_{x,s,l,j} \text{ and } \langle \tilde{a}_{x,s,l,j}, \nabla_s \rangle \geq 0.\]

Intuitively the above result states that a regular vertex \(x \in \mathbb{R}^{n_0}\) is a local minimum of \(f\) if and only if for any region separating axis \(a \in A(x)\), \(f\) is non-decreasing locally in the direction \(a\). Furthermore, for every axis \(a_{x,r,l,j}\) for \(r \in \{0, 1\}, (l, j) \in C^*(x)\) the test whether \(f\) is non-decreasing locally in direction of the axis \(a_{x,r,l,j}\) can be carried out by checking \(\langle \tilde{a}_{x,s,l,j}, \nabla_s \rangle > 0\) with some \(s \in S^C(x)\) with \(s_{lj} = r\). In particular, this result again links properties that are not specific to an activation pattern to those that are. We will use Proposition 21 in the stopping criterion for our algorithm presented in Section 3.2.4.

For regular networks, this condition will exactly identify vertices that are local minima.

**Lemma 22.** Every strict local minimum of \(f\) is a vertex and for every non-strict local minimum \(x \in \mathbb{R}^{n_0}\) there exist a vertex \(x' \in \mathbb{R}^{n_0}\) with \(f(x) = f(x')\).

### 3 Solvers for Deep ReLU Programming Problems

In this section we will describe how solver algorithms for deep ReLU programming problems can be implemented. We first describe basic algorithmic routines in Section 3.1 that will be used as building blocks in Section 3.2 where we describe our algorithm in detail and provide pseudo-code. We also provide an implementation in the Julia programming language, see Appendix C for details.

#### 3.1 Algorithmic Primitives

**3.1.1 Computation of the gradient**

For a given activation pattern \(s \in S\) we can compute the gradient \(\nabla_s \in \mathbb{R}^{n_0}\) in \(O(\sum_{l=1}^{L} n_l m_{l-1})\) by using equation (35). Below we will denote the function that computes this gradient by \(\text{Gradient}\).
3.1.2 Computation of an oriented normal vector

For an activation pattern $s \in S$ and a neuron position index $(l, j) \in I$ the complexity for the computation of the normal vector $\tilde{v}_{s.l,j}$ is of order $O(n_0 n_1 + \ldots + n_{l-1} n_l)$. This follows from equation (25) if we compute the matrix product from the left side starting with the $j$-th row of $W_l$. The same holds for the oriented normal vector $\tilde{u}_{s.l,j}$ by equation (26). In our pseudo-code we will denote this procedure by ORIENTEDNORMALVEC taking as parameters the activation pattern $s \in S$ and the neuron position $c = (l, j) \in I$.

3.1.3 Computing inner products with all oriented normal vectors

Assume we have an activation pattern $s \in S$ and a vector $w \in \mathbb{R}^{n_0}$. We want to compute the inner products $\langle w, \tilde{v}_{s.l,j} \rangle$ for all $(l, j) \in I$. Then each of these inner products can be computed as explained in Section 3.1.2 in $O(\sum_{l=1}^{L}(n_l n_{l-1}))$. However there exists a more efficient way to compute all such inner products at once in the same complexity order. From equation (25) it follows that

$$\langle w, \tilde{v}_{s.l,j} \rangle = (W_l \text{diag}(s_{l-1}) W_{l-1} \cdots \text{diag}(s_1) W_1 w)_j \quad \text{for } (l, j) \in I \quad (37)$$

In particular the desired inner products can be computed by pushing the vector $w$ through the network with fixed activation pattern $s$ but with bias terms set to zero and observing the computed values at all neuron positions $I$. More precisely, we successively compute the above matrix vector product from the right as $u_1 := W_1 w$, $u_{l+1} := W_{l+1} \text{diag}(s_l) u_l$ for $l \in \{1, \ldots, L\}$ and read the corresponding entries $(u_l)_j$ for $(l, j) \in I$. According to equation (26), depending on the activation $s_{lj}$ the sign in equation (37) needs to be flipped to obtain the inner products with the oriented normal vectors. Since this change of the sign, does not increase the overall complexity order, the computation of all inner products is of order $O(\sum_{l=1}^{L} n_l n_{l-1})$.

3.1.4 Projection using the pseudoinverse of some oriented normal vectors

Assume we have a number $m \leq n_0$ of distinct neuron positions $(l_1, j_1), \ldots, (l_m, j_m) \in I$, an activation pattern $s \in S$ and the pseudoinverse $A^+ \in \mathbb{R}^{m \times n_0}$ of the matrix whose linearly independent columns are the oriented normal vectors $\tilde{u}_{s.l_1,j_1}, \ldots, \tilde{u}_{s.l_m,j_m}$. We can project a vector $v \in \mathbb{R}^{n_0}$ onto the linear span of $\tilde{u}_{s.l_1,j_1}, \ldots, \tilde{u}_{s.l_m,j_m}$ by computing $(A^+)^T w$ where $w = (\langle v, \tilde{u}_{s.l_1,j_1} \rangle, \ldots, \langle v, \tilde{u}_{s.l_m,j_m} \rangle) \in \mathbb{R}^{n_0}$. By Section 3.1.3 $w$ can be computed in $O(\sum_{l=1}^{L} n_l n_{l-1})$ such that the computational complexity for the computation of this projection is of order $O(\sum_{l=1}^{L} n_l n_{l-1} + n_0^2)$. Below we will denote this procedure by PROJECT($A^+, C, S, V$) where $A^+$ is the pseudoinverse matrix, $C$ is the list of neuron positions, $S$ is the activation pattern and $V$ is the vector to project.

3.1.5 Updating the pseudoinverse for an additional column

Given $n \in \{0, \ldots, n_0 - 1\}$, a matrix $A \in \mathbb{R}^{n_0 \times n}$ with rank$(A) = n$ and its pseudoinverse $A^+ = (A^T A)^{-1} A^T$ we can add a given additional linear independent column $w \in \mathbb{R}^{n_0}$ to $A$
and compute the corresponding pseudoinverse in computational complexity $O(nn_0)$. This can be done by first projecting $w$ onto the column space of $A$ by computing $w_\parallel = (A^+)^\top A^\top w$ and taking the orthogonal part $w_\perp = w - w_\parallel$. If we denote by $\tilde{A} \in \mathbb{R}^{n_0 \times (n+1)}$ the matrix that has the same columns as $A$ and an additional $(n+1)$-th column equal to $w$ then its pseudoinverse $\tilde{A}^+$ can be constructed as follows. The $(n+1)$-th row of $\tilde{A}^+$ is equal to $\langle w_\perp, w \rangle^{-1} w_\perp$. Furthermore, if we denote the $i$-th row of $\tilde{A}^+$ by $a_i^+$ then the $i$-th row of $\tilde{A}^+$ is equal to $a_i^+ - \langle a_i^+, w \rangle \langle w_\perp, w \rangle^{-1} w_\perp$ for $i \in \{1, \ldots, n\}$. One easily checks that with this definition $\tilde{A}^+ \tilde{A} = \text{Id}_{n+1}$ and that the column space of $\tilde{A}$ and the row space of $\tilde{A}^+$ are equal. Obviously, given $A, A^+$ and $w$ this construction of $\tilde{A}^+$ is of order $O(nn_0)$.

In our pseudo-code we will use a procedure ADDPSEUDOROW that takes the following parameters. The first parameter $A^+$ is the pseudoinverse of the matrix $A$ with the oriented normal vectors $\bar{u}_{s,l}|i,j|, \ldots, \bar{u}_{s,l}|j,j|$ as columns where the ordered collection $C = ((l_1, j_1), \ldots, (l_{|l|}, j_{|l|}))$ of neuron indices and the activation pattern $s \in \mathcal{S}$ are the second and third parameters. The last parameter $u$ is the additional column that shall be added to $A$. The return value is the updated pseudoinverse $\tilde{A}^+$ computed as described above. Note that we use the PROJECT procedure from Section 3.1.4 to compute $w_\parallel$ in order $O(\sum_{l=1}^L n_l m_{l-1} + n_0^2)$. The rest of the construction can also be covered by this computational complexity order.

Upon this procedure we build the ADDAXIS procedure in Algorithm 1 which, instead of a specified an additional column $u$, takes the neuron position $c$ as its last parameter from which the oriented normal vector will be derived and then used in the above ADDPSEUDOROW procedure. From the pseudo-code and the above discussion it is clear that the whole ADDAXIS procedure is of computational complexity order $O(\sum_{l=1}^L n_l m_{l-1} + n_0^2)$.

**Algorithm 1 Add Axis**

1: procedure ADDAXIS($A^+$: matrix, $s$: activation pattern, $c$: neuron position)
2: $u \leftarrow$ ORIENTEDNORMALVEC($s, c$)
3: $A^+ \leftarrow$ ADDPSEUDOROW($A^+, C, s, u$)
4: return $A^+$
5: end procedure

### 3.1.6 Updating the pseudoinverse for an omitted column

Given $n \in \{1, \ldots, n_0\}$, a column index $i \in \{1, \ldots, n\}$, a matrix $A \in \mathbb{R}^{n_0 \times n}$ with rank($A$) = $n$ and its pseudoinverse $A^+ = (A^\top A)^{-1} A^\top$ we can omit the $i$-th column of $A$ and compute the corresponding pseudoinverse in computational complexity $O(nn_0)$. If we denote the rows of $A^+$ by $a_1^+, \ldots, a_n^+$ and the matrix that is obtained by omitting the $i$-th column in $A$ by $\bar{A} \in \mathbb{R}^{(n-1) \times n_0}$ then for $k \in \{1, \ldots, n - 1\}$ the $k$-th row of its pseudoinverse $\bar{A}^+ \in \mathbb{R}^{(n-1) \times n_0}$ is equal to $a_j^+ - \langle a_i^+, a_j^+ \rangle a_i^+$ where $j(k) = k$ if $k < i$, $j(k) = k + 1$ if $k \geq i$.

Note that this construction of $\bar{A}^+$ is of complexity order $O(nn_0)$ and does not involve $A$ but only its pseudoinverse $A^+$. Again one easily verifies that indeed $A^+ A = \text{Id}_{n-1}$ and that the column space of $\bar{A}$ and the row space of $\bar{A}^+$ are equal.
In our pseudo-code we will denote this procedure by REMOVEPSEUDOROW taking as parameters a matrix $A^+$ and an index $i$. It returns the modified pseudoinverse as described above where $A^+$ and $i$ take the roles of the pseudoinverse $A^+$ and $i$ respectively.

### 3.1.7 Flip one activation

For an activation pattern $s \in \mathcal{S}$ and a neuron index $(l^*, j^*) \in \mathcal{I}$ we define the function that flips the activation of $s$ at position $(l^*, j^*)$ by $\text{FLIP} : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S}$ with

$$\text{FLIP}(s, (l^*, j^*))_{lj} = \begin{cases} s_{lj} & \text{if } (l, j) \neq (l^*, j^*) \\ 1 - s_{lj} & \text{if } (l, j) = (l^*, j^*) \end{cases}.$$  

(38)

In an implementation with constant random access time, this function can be implemented as it accesses and changes only one specific bit.

### 3.1.8 Updating the feasible axes when changing one activation

We now explain the essential ingredient that allows the change of the affine region in a computationally efficient way. It exploits the fact that flipping the activation pattern at one specific position only causes one feasible axis to change as depicted in Figure 6.

Assume that $x$ is a regular vertex, i.e. $|C^*(x)| = n_0$ and let $s \in C^*(x)$ be a compatible activation pattern. Applying Lemma 14 we identify the affected indices $\tilde{C}_s(x) = C^*(x) = \{(l_1, j_1), \ldots, (l_{n_0}, j_{n_0})\}$. Let $A_s \in \mathbb{R}^{n_0 \times n_0}$ be the matrix with columns $\tilde{u}_{s,l_1,j_1}, \ldots, \tilde{u}_{s,l_{n_0},j_{n_0}}$ and denote its pseudoinverse by $A_s^+ = (A_s^T A_s)^{-1} A_s^T$. Now we change the activation at neuron index $(l_{n_0}, j_{n_0})$, i.e. let $s' = \text{FLIP}(s, (l_{n_0}, j_{n_0}))$ as defined in equation (38). We want to find the pseudoinverse $A_{s'}^+ \in \mathbb{R}^{n_0 \times n_0}$ of the matrix $A_{s'} \in \mathbb{R}^{n_0 \times n_0}$ with columns $\tilde{u}_{s',l_1,j_1}, \ldots, \tilde{u}_{s',l_{n_0},j_{n_0}}$. The key insight here is the fact that the first $n_0 - 1$ rows of $A_{s'}^+$ and $A_s^+$ are equal. Since $A_{s'}^+$ is the pseudoinverse of $A_s$, the unknown axis $a_{s,n_0}^+ \in \mathbb{R}^{n_0}$ in the last row of $A_{s'}^+$ satisfies the following three conditions. Firstly, it is an element of the linear span of $\tilde{u}_{s',l_1,j_1}, \ldots, \tilde{u}_{s',l_{n_0-1},j_{n_0-1}}$, secondly it is orthogonal to $\tilde{u}_{s',l_1,j_1}, \ldots, \tilde{u}_{s',l_{n_0-1},j_{n_0-1}}$ and finally it satisfies $\langle a_{s,n_0}^+, \tilde{u}_{s',l_{n_0},j_{n_0}} \rangle = 1$. These three conditions uniquely determine $a_{s,n_0}^+ \in \mathbb{R}^{n_0}$ and suggest the following computation. First compute

$$w := \tilde{u}_{s',l_{n_0},j_{n_0}} - \sum_{i=1}^{n_0-1} \langle \tilde{u}_{s',l_1,j_1}, \tilde{u}_{s',l_{n_0},j_{n_0}} \rangle a_{s,n_0}^+ \in \mathbb{R}^{n_0},$$  

(39)

which then needs yields $a_{s,n_0}^+ = \langle w, \tilde{u}_{s',l_{n_0},j_{n_0}} \rangle^{-1} w$. One easily checks that indeed the three conditions are satisfied. Since $\tilde{u}_{s',l_{n_0},j_{n_0}}$ and the inner products can be computed in $\mathcal{O}(\sum_{i=1}^{L} n_i n_{i-1})$ as explained in Sections 3.1.2 and 3.1.3 the overall computational effort in equation (39) is of order $\mathcal{O}(\sum_{i=1}^{L} n_i n_{i-1} + n_0^2)$ because we need to subtract $n_0 - 1$ vectors with $n_0$ entries each. In turn this implies that $a_{s,n_0}^+$ and hence the matrix $A_{s'}^+$ can be constructed in $\mathcal{O}(\sum_{i=1}^{L} n_i n_{i-1} + n_0^2)$ if we already know $A_s^+$. Above we assumed for simplicity that the axis that needs to be recomputed corresponds to the last row of the matrix $A_{s'}^+$. We will denote the above procedure by UPDATEAXISNEWREGION and allow for an arbitrary row index.
Its arguments \((\mathbf{A}^+, \mathbf{i}, \mathbf{s}, \mathbf{c})\) specify the matrix \(\mathbf{A}^+\) whose \(\mathbf{i}\)-th row needs to be updated when the activation pattern \(\mathbf{s}\) is flipped at neuron index \(\mathbf{c} \in \mathcal{I}\), where \(\mathbf{c}\) corresponds to \((l_{n0}, j_{n0})\) above. This procedure returns the updated matrix \(\mathbf{A}^+\).

We want to stress again that this algorithmic primitive is the important building block which allows to change the considered affine region. It is therefore the main ingredient that distinguishes our algorithm presented in Section 3.2 from linear programming algorithms where the considered affine region is the fixed feasible region. The fact that this change of the considered region is computationally not more expensive than the other algorithmic differences makes deep ReLU programming interesting for practical problems, see Table 1.

**Figure 6:** Illustration of the concept for the algorithmic primitive in Section 3.1.8 for \(n_0 = 3\).

If we know the feasible axes \(\tilde{\mathbf{u}}_{\tilde{s}, l_1, j_1}, \tilde{\mathbf{u}}_{\tilde{s}, l_2, j_2}, \tilde{\mathbf{u}}_{\tilde{s}, l_3, j_3}\) of the red region with activation pattern \(\tilde{s} \in \mathcal{S}\) and want to compute the feasible axes \(\tilde{\mathbf{u}}_{\tilde{s}, l_1, j_1}, \tilde{\mathbf{u}}_{\tilde{s}, l_2, j_2}, \tilde{\mathbf{u}}_{\tilde{s}, l_3, j_3}\) of the green region with activation pattern \(\tilde{s}\) then we can reuse the \(n_0 - 1 = 2\) axes \(\tilde{\mathbf{u}}_{\tilde{s}, l_1, j_1} = \tilde{\mathbf{u}}_{\tilde{s}, l_1, j_1}\) and \(\tilde{\mathbf{u}}_{\tilde{s}, l_2, j_2} = \tilde{\mathbf{u}}_{\tilde{s}, l_2, j_2}\), only one axis \(\tilde{\mathbf{u}}_{\tilde{s}, l_3, j_3}\) needs to be recomputed.

### 3.1.9 Advance maximally without leaving the current affine region

Assume we have an argument \(x \in \mathbb{R}^{n_0}\) and we want to advance into a direction \(v \in \mathbb{R}^{n_0}\) maximally such that the hyperplane pattern does not change. More precisely we want to find

\[
t^* = \sup \{t \geq 0 \mid H(x + tv) = H(x)\}
\]

as illustrated in Figure 7. The value \(t^*\) above can be computed efficiently as follows. Let \(x_v(t) = x + tv\) for \(t \in \mathbb{R}\). If we use the notation from equation (2) and define \(\alpha_1 = W_1 x + b_1\) and \(\beta_1 = W_1 v\) then \(g^{(1)}(x_v(t)) = \text{ReLU}.(\alpha_1 + t \beta_1)\) such that

\[
t_1 := \min \left\{-\frac{(\alpha_1)_j}{(\beta_1)_j} \middle| j \in \{1, \ldots, n_1\} \land (\beta_1)_j \neq 0 \land -\frac{(\alpha_1)_j}{(\beta_1)_j} > 0\right\}
\]

is the infimum \(t_1 = \inf \{t > 0 \mid \exists j \in \{1, \ldots, n_1\} : H(x)_{1,j} \neq H(x_v(t))_{1,j}\}\). Similarly, if we set \(s = S(x)\) and recursively define \(\alpha_{l+1} = W_{l+1} \text{diag}(s_l) \alpha_l\), \(\beta_{l+1} = W_{l+1} \text{diag}(s_l) \beta_l\) for \(l \in\)
Figure 7: Illustration of the setting in 3.1.9 for $n_0 = 2$. Given a starting point $x \in \mathbb{R}^{n_0}$, a compatible activation pattern $s \in S^C(x)$ and a direction $v \in \mathbb{R}^{n_0}$ we consider the ray $x + tv$, $t \geq 0$ and find $t$ maximally such that the affine region corresponding to $s$ is not left. In the figure, $x' = x + t^*v$.

Figure 8: Illustration of a problem caused by numerical uncertainty for a two-dimensional input $n_0 = 2$. At the position $x_1$ we want to advance in the direction parallel to the boundary of two affine regions until we encounter a change of the affine behaviour at position $x_2$. However, when using finite precision arithmetic, it can happen that $x_1$ itself is not exactly on the boundary and that the direction we take for the next step is not parallel to the boundary line as seen on the magnification of the of the corresponding rectangle section on the left. This causes the dotted line to intersect with the region boundary line such that the next position would be $x'_2$. In order to avoid this artifact, we need to keep book about which region boundaries we are assuming to stay on and ignore activation changes of the corresponding neurons.
\{1, \ldots, L - 1\}, then
\[
t_l := \min \left\{ -\frac{(\alpha_l)_j}{(\beta_l)_j} \mid j \in \{1, \ldots, n_l\} \land (\beta_l)_j \neq 0 \land -\frac{(\alpha_l)_j}{(\beta_l)_j} > 0 \right\}
\] (42)
satisfies \(t_l = \inf \{ t > 0 \mid \exists j \in \{1, \ldots, n_l\} : H(x_{l,j}) \neq H(x_{v}(t_{l,j})) \} \) for \(l \in \{1, \ldots, L\}\). In particular it holds that \(t^* = \min(t_1, \ldots, t_L)\).

This basic idea needs to be extended in some ways to fit our needs. Specifically, we need the following modifications:

1. **Explicit specification of activation pattern**: Since we will track the activation pattern in a dedicated state variable in our algorithm in Section 3.2 instead of using \(s = S(x)\) as above, the activation pattern \(s\) to be used in the above construction will be explicitly specified as a parameter which is assumed to be compatible with \(x\), i.e. to be an element of \(S^C(x)\).

2. **Information about the corresponding neuron position**: In our implementation below the neuron index at which the activation pattern changes at the point \(x_v(t^*)\) is needed. In other words, we also require the argmin neuron position to be returned.

3. **Controllable insensitivity**: We need to specify some neuron positions where the sign change will be ignored, they shall not be considered in the construction \(t^*\). This is necessary to avoid numerical problems illustrated in Figure 8.

4. **Mismatch robustness**: Due to numerical uncertainty, it can happen that our specified activation pattern is not compatible with the hyperplane pattern at position \(x\), i.e. \(s \not\in S^C(x)\). This means that \(x\) is outside the affine region of the objective function \(f\) corresponding to \(s\). In such a case it makes sense to ignore a sign change or to allow for negative values for \(t^*\), see Figure 9. This is achieved in line 10 of Algorithm 2.

In Algorithm 2 we provide pseudo-code that implements the computation of \(t^*\) and these modifications. If the list of neurons \(C\) is provided as a boolean array which allows random access, then the check if a specific neuron index \((i, j) \in I\) is an element of \(C\) requires constant time and therefore, the computational complexity of the whole procedure is of order \(O(\sum_{l=1}^{L} n_l m_{l-1})\).

### 3.2 Proposal for a solver algorithm

In this section, we describe our **Deep ReLU Linear Programming (DRLP) algorithm**, a basic iterative solver for deep ReLU linear programming problems. Where appropriate we also provide pseudo-code. This is the basis for our implementation in the Julia programming language, see Appendix C. The objective feed-forward neural network \(f\) is assumed to be implicitly given. We require \(f\) to satisfy the following two conditions:

1. There need to exist vertices because we aim to first find a vertex and will then iterate on vertices.
Figure 9: Illustration of the mismatch robustness implemented in Algorithm 2. Assume that the activation pattern $s$ is compatible with arguments $\tilde{x}$ in the gray area, i.e. $s \in \mathcal{S}^C(\tilde{x})$ for those $\tilde{x}$. However the current position $x$ is not in the gray area, $s \not\in \mathcal{S}^C(x)$. We distinguish two cases. If the specified direction $v = v_2$ points away from the gray area we allow for negative $t^*$ to get back to the region boundary (black line). If the specified direction $v = v_1$ points towards the gray area, then we ignore this region boundary in the construction of $t^*$. This way we protect against slight disturbances of the specified position $x$ and the assumption $s \in \mathcal{S}^C(x)$.

Algorithm 2 Advance maximally in a specified direction

1: \textbf{procedure} ADVANCEMAX($x$: position, $v$: direction, $s$: activation pattern, $C$: list of neuron positions to ignore)
2: \hspace{1em} $\alpha_1 \leftarrow W_1 x + b_1$
3: \hspace{1em} $\beta_1 \leftarrow W_1 v$
4: \hspace{1em} $c \leftarrow (1, 1)$
5: \hspace{1em} $t^* \leftarrow \infty$
6: \hspace{1em} \textbf{for} $l = 1, \ldots, L$ \textbf{do}
7: \hspace{2em} \textbf{for} $j = 1, \ldots, n_l$ \textbf{do}
8: \hspace{3em} \textbf{if} $(l, j) \notin C$ \textbf{and} $\beta_{lj} \neq 0$ \textbf{then}
9: \hspace{4em} $t \leftarrow \frac{a_{lj}}{\beta_{lj}}$
10: \hspace{4em} \textbf{if} $(s_{lj} = 1 \land \beta_j < 0) \lor (s_{lj} = 0 \land \beta_j > 0)$ \textbf{then}
11: \hspace{5em} \textbf{if} $t < t^*$ \textbf{then}
12: \hspace{6em} $t^* \leftarrow t$
13: \hspace{6em} $c \leftarrow (l, j)$
14: \hspace{5em} \textbf{end if}
15: \hspace{4em} \textbf{end if}
16: \hspace{3em} \textbf{end if}
17: \hspace{2em} \textbf{end for}
18: \hspace{1em} $\alpha_{l+1} \leftarrow W_{l+1}\text{diag}(s_l)\alpha_l + b_{l+1}$
19: \hspace{1em} $\beta_{l+1} \leftarrow W_{l+1}\text{diag}(s_l)\beta_l$
20: \hspace{1em} \textbf{end for}
21: \hspace{1em} \textbf{return} $(x + t^*v, c)$
22: \textbf{end procedure}
2. The network $f$ needs to be regular in the sense of Section 2.6 because we rely on the linear independence of the normal vectors $(\tilde{v}_{s,l,j})_{(l,j)\in \tilde{C}_s(x)}$ as provided by Lemma 15 and on the existence of exactly $2n_0$ region separating axes at every visited vertex.

3.2.1 State Variables

The state of the algorithm involves the following quantities. The current position in the argument space $\mathbb{R}^{n_0}$ will be saved in the state variable $x \in \mathbb{R}^{n_0}$. Furthermore, the variable $s \in S$ will always hold an activation pattern that is compatible with $x$, i.e. it shall always hold $s \in S^C(x)$. The variable $C$ is an ordered collection of elements in $\mathcal{I}$. Its purpose is to keep track of the affected indices $\tilde{C}_s$. We assume that $C$ contains at most $n_0^\alpha$ such indices. Note that this assumption is true by our regularity assumption by Lemma 14.

Finally, the algorithm will also keep track of feasible axes in the rows of a matrix $A^+ \in \mathbb{R}^{n_0 \times n_0}$. This matrix will always be the pseudoinverse of the matrix whose columns are the oriented normal vectors $\tilde{u}_{s,l,j_1}, \ldots, \tilde{u}_{s,l_{|C|},j_{|C|}}$ where $((l_1,j_1), \ldots, (l_{|C|},j_{|C|})) = C$. The pseudoinverse will not be recomputed in every step but incrementally updated using the algorithmic primitives from Sections 3.1.5, 3.1.6 and 3.1.8.

3.2.2 Initialization

Our DRLP algorithm takes an initial position $x = x_0$ as input. For simplicity, we assume that this initial position is not at the boundary of two affine regions, i.e. $C^*(x_0) = \emptyset$. Hence we initialize $C$ to be empty and the matrix $A^+ \in \mathbb{R}^{n_0 \times n_0}$ to have no rows. We then compute the activation pattern $s = S(x)$ in a forward pass in $O(\sum_{l=0}^{L} n_l n_{l-1})$. Below we will denote this procedure by INITIALIZE($x$) and assume it returns the initialized variables $(s, C, A^+)$. 

3.2.3 Finding a vertex

After the initialization we need to perform $n_0$ iterations to find a regular vertex of the affine region corresponding to the initial activation pattern $s$. In this phase the activation pattern $s$ and hence affine region is not changed. We first compute the gradient $\nabla_s \in \mathbb{R}^{n_0}$ as explained in Section 3.1.1 and set the initial direction to $v = -\nabla_s$. We then move into the direction opposite to the gradient until we encounter the border of the current affine region. This is done maximally such that the current affine region is not left by using the ADVANCEMAX subroutine from Section 3.1.9. The matrix $A^+$ is updated by using the ADDAXIS procedure from Section 3.1.5 and the neuron index $c$ returned by ADVANCEMAX is added to the set $C$ of affected indices in line 7 of Algorithm 3. Then the direction for the next step is adjusted by projecting it onto the space orthogonal to span of all normal vectors with indices in $C$. Note that for any $v \in \mathbb{R}^{n_0}$ and subspace $U \subset \mathbb{R}^{n_0}$, the projection $v_U$ of $v$ onto $U$ satisfies $\langle v_U, -v \rangle \leq 0$ such that the objective function will not increase in the next iteration. This procedure of moving maximally into a direction and the adjustment of used direction, $A^+$ and $C$ is repeated $n_0$ times until we arrive at a vertex, see Figure 10 and the pseudo-code of the procedure FINDVERTEX in Algorithm 3.
Figure 10: Illustration of the procedure of finding a vertex $x_3$ for $n_0 = 3$ starting at position $x_0$. The vector $x_2 - x_1$ is orthogonal to the normal vector of the yellow hyperplane, the vector $x_3 - x_2$ is orthogonal to normal vectors of both, the yellow and the blue hyperplanes.

---

**Algorithm 3** Find a vertex

1: **procedure** FindVertex($x$: position, $s$: activation pattern, $C$: list of neuron positions, $A^+$: matrix)
2:  $\nabla_s \leftarrow \text{Gradient}(s)$
3:  $v \leftarrow -\nabla_s$
4:  **while** $|C| < n_0$ **do**
5:     $(x, c) \leftarrow \text{AdvanceMax}(x, v, s, C)$
6:     $A^+ \leftarrow \text{AddAxis}(A^+, s, c)$
7:     $C \leftarrow \text{AddItem}(C^+, c)$
8:     $v \leftarrow v - \text{Project}(A^+, C, s, v)$
9:  **end while**
10: **return** $(x, s, C, A^+)$
11: **end procedure**
3.2.4 Traversing different affine regions

After a vertex has been found by the FINDVERTEX procedure, the returned position $x$ is in a vertex and the rows of the matrix $A^+$ are the feasible axes. Similar to the simplex algorithm in linear programming, we want to choose one of these axes to proceed the iteration on edges. For our purposes we use the procedure CHOOSEAXIS from Algorithm 4 which selects the axis with the strongest correlation with an input direction $v$. In line 2 the expression

\begin{verbatim}
Algorithm 4 Choose Axis
1: procedure CHOOSEAXIS($A^+$: matrix, $v$: direction)
2:   $i \leftarrow \text{argmax}(\{1, \ldots, \text{nrows}(A^+)\} ; i \mapsto \langle v, A^+_i \rangle / \sqrt{\langle A^+_i, A^+_i \rangle})$
3:   $a^+ \leftarrow A^+_i$
4:   return $(a^+, \langle v, a^+ \rangle, i)$
5: end procedure
\end{verbatim}

“nrows($A^+$)” returns the number of rows of the matrix $A^+$. In line 4 we return the selected axis $a^+$, i.e. row of $A^+$, its inner product $\langle v, a^+ \rangle$ with the input direction $v$ and the row index $i$ corresponding to $a^+$.

Now we can compose the subroutines we previously described to form the final DRLP-Algorithm 5. If we use the negative gradient $-\nabla s$ for the direction $v \in \mathbb{R}^{n_0}$ in Algorithm 4 then the sign of the returned inner product $\langle v, a^+ \rangle$ tells us if there exists a feasible axis which has a negative inner product with the gradient $\nabla s$ depending on which we are taking the following actions:

- If there is such an axis $a^+$, the row with the corresponding index $i$ will be removed from the matrix $A^+$ and we proceed by moving into the direction of this axis using the ADVANCEMAX procedure. Either there is no neuron that changes its activation in the direction $a^+$. In this case the resulting function can be made arbitrarily small and the algorithm terminates\(^1\). Or the procedure returns the new position $x \in \mathbb{R}^{n_0}$ and the index $c \in I$ of the corresponding neuron that changes its activation at that position. Based on this information we modify the matrix $A^+$ using the ADDAXIS function from Section 3.1.5 such that its rows are the feasible axes $\tilde{A}_s(x)$. The old neuron index corresponding to the hyperplane that was abandoned in the ADVANCEMAX step is removed from the ordered collection $C$ in line 11 and the new index $c$ is appended in line 12.

At this stage, the considered affine region of $f$ shall be changed to allow to continue in a different region. To this end the following two state variables need to be adjusted.

1. The state variable $s$ which specifies the activation pattern is flipped at the position $c$ corresponding to the critical neuron. This is achieved by the FLIP function from Section 3.1.7 in line 13 of Algorithm 5.

\(^1\)For simplicity of the exposition, we did not consider this case where $t^* = \infty$ in Algorithm 2.
2. The set of feasible axes need to be updated. Since only one activation changes, we can exploit the fact that only one axis needs to be recomputed as described in Section 3.1.8 by using the UpdateAxisNewRegion in line 14.

We can now continue with the next iteration as above by computing the gradient $\nabla s$ for the new $s$. However before the next iteration is made, we assign 1 to an additional variable nextFlipIndex which will explained below.

- If there is no axis $a^+$ which has negative correlation with the gradient then the current vertex $x$ is a local minimum of the affine region corresponding to the activation pattern $s$. In this case we will change $s$ to a different compatible activation pattern in $\mathcal{S}^C(x)$. After $s$ was changed using the FLIP function in line 21 we update the axes in $A^+$ for the new activation pattern using UpdateAxisNewRegion in line 22. Then we recheck whether there is a feasible axis among the new axes which has negative inner product with the new gradient $\nabla s$. In each retry we flip $s$ at a different neuron. More precisely we subsequently flip at $C[1], C[2], \ldots, C[n_0]$. This is achieved by increasing the index counter nextFlipIndex in line 23. After $n_0$ such flips without finding a new direction to continue with, Proposition 21 guarantees that $x$ is a local minimum of $f$ and the procedure finishes returning $x$ in line 18.

### 3.2.5 Computational Complexity

Table 1 summarizes the computational complexity orders for the subroutines used of the previous and this section. Note that all these subroutines can be executed in complexity order $O(\sum_{l=1}^{L} n_l n_{l-1} + n_0^2)$. In particular every loop iteration in Algorithms 3 and 5 is of the same complexity order. We call these loop iterations steps of our algorithm, since either the position of $X$ or the considered activation pattern $s$ changes. Every such step is of order $O(\sum_{l=1}^{L} n_l n_{l-1} + n_0^2)$.
Algorithm 5 DRLP Algorithm

1: procedure DRLP-Algorithm(x: starting position)
2:   (s, C, A⁺) ← INITIALIZE(x)
3:   (x, s, C, A⁺) ← FINDVERTEX(x, s, C, A⁺)
4:   while true do
5:     ∇s ← GRADIENT(s)
6:     (a⁺, α, i) ← CHOOSEAXIS(A⁺, C, s, ∇s)
7:     if α < 0 then
8:       A⁺ ← REMOVEPSEUDOROW(A⁺, i)
9:       (x, c) ← ADVANCEMAX(x, a⁺, s, C)
10:      A⁺ ← ADDAXIS(A⁺, c)
11:     C ← REMOVEATINDEX(C, i)
12:      C ← ADITEM(C, c)
13:     s ← FLIP(s, c)
14:     A⁺ ← UPDATEAXISNEWREGION(A⁺, i, s, c)
15:    nextFlipIndex ← 1
16:   else
17:     if nextFlipIndex > n₀ then
18:       return x
19:     end if
20:     c ← C[nextFlipIndex]
21:     s ← FLIP(s, c)
22:     A⁺ ← UPDATEAXISNEWREGION(A⁺, nextFlipIndex, s, c)
23:    nextFlipIndex ← nextFlipIndex + 1
24:   end if
25: end while
26: end procedure
3.2.6 Possible modifications

In our pseudo-code we focussed on an easy to understand proof-of-concept implementation and there are plenty possible extensions and improvements.

**Position correction** In our DRLP-algorithm, we keep track of the position $\mathbf{x} \in \mathbb{R}^{n_0}$ and the activation pattern $\mathbf{s} \in \mathcal{S}$ separately, these variables are only synchronized at the beginning in the *Initialize* function. Due to numerical uncertainty, they can diverge, i.e. $\mathbf{s} \notin \mathcal{S}^C(\mathbf{x})$. To avoid this, it may be beneficial to force the position to be exactly in a vertex whenever this is implicitly assumed. More precisely, after the *FindVertex* procedure in Algorithm 5, the position $\mathbf{x}$ should only iterate on vertices. This means that at the neuron positions $(l, j) \in \mathcal{C} \subset \mathcal{I}$ the ReLU arguments need to be zero, i.e. $A(\mathbf{x})_{lj} = 0$. If $\mathbf{s} \in \mathcal{S}^C(\mathbf{x})$ as assumed in our algorithm, $\tilde{A}_x(\mathbf{x})_{lj} = 0$ for $(l, j) \in \mathcal{C}$ by Theorem 2. By equation (24) this yields a linear equation system involving inner products with normal vectors corresponding to the neuron positions $\mathcal{C}$. Using the inner product computation from Section 3.1.3, equation (26) and the state matrix $A^+$ this equation system can be solved in complexity order $\mathcal{O}(\sum_{l=1}^{L} n_l n_{l-1} + n_0^2)$. Hence, this position correction does not increase the overall complexity order per step.

**Axis correction** Again due to numerical uncertainty and the fact that the feasible axes in the state matrix $A^+$ are recursively modified, numerical errors can accumulate and it might be necessary to do a fresh computation of $A^+$ as the pseudoinverse of the corresponding oriented normal vectors. However, this step would require a computational complexity of $\mathcal{O}(n_0^3)$.

**Other activation function** Our theoretical and algorithmic considerations of Sections 2 and 3 can be easily extended to feed-forward neural networks using activation functions of the form

$$
\sigma_{a,b} : \begin{cases} 
\mathbb{R} & \to \mathbb{R} \\
 x & \mapsto a\text{ReLU}(-x) + b\text{ReLU}(x)
\end{cases}
$$

for $a, b \in \mathbb{R}$. (43)

To apply our algorithm for networks using this activation function, one can adjust the relevant parts of the algorithm such as the gradient computation.

Another possibility is the representation of a neuron using this activation functions by a linear combination of two neurons each using a ReLU activation function. These pairs of neurons then induce the same hyperplanes and the overall neural network is not be regular anymore. One therefore has consider only one representative of each such pair in the AdvanceMax procedure and flip their activation synchronously in the Flip procedure. We use this technique in our Julia implementation such that we can successfully demonstrate our algorithm in quantile regression, where final layer’s activation function is of the form (43).
Alternatives to the iteration on vertices. Our DRLP-Algorithm first finds a vertex and then proceeds on vertices in subsequent iterations. The theory in Section 2.7.1 on feasible axes for regular vertices can easily be extended for regular points. More precisely, the definition of the feasible axes are also valid for regular points \( x \in \mathbb{R}^{n_0} \) by Lemmas 14 and 15. Furthermore, from the proof of Lemma 18 it is clear that

\[
v \in \tilde{D}_s(x) \iff \forall (l,j) \in \tilde{C}_s(x) \langle v, \tilde{u}_{s,l,j} \rangle \geq 0 \quad \text{for } s \in S^C(x) \tag{44}
\]

such that \( v \in \tilde{D}_s(x) \) if and only if \( v = v_\perp + v_\parallel \) with \( v_\parallel = \sum_{(l,j) \in \tilde{C}_s(x)} \alpha_{l,j} \tilde{a}_{s,x,l,j} \) for nonnegative coefficients \( \alpha_{l,j} \), \( (l,j) \in \tilde{C}_s(x) \) and \( \langle v_\perp, \tilde{u}_{s,l,j} \rangle = 0 \) for \( (l,j) \in \tilde{C}_s(x) \). This decomposition allows to find directions \( v \in \tilde{D}_s(x) \) that have negative inner product with the gradient \( \nabla_s \) which can then be used instead of \( a^+ \) in line 9 of Algorithm 5.

We can then drop the first condition on \( f \) we required at the beginning of Section 3.2 since we do not need iterate on vertices anymore.

Definite quadratic forms. For \( m \in \mathbb{R} \), a positive definite \( m \)-ary quadratic form \( q : \mathbb{R}^m \rightarrow \mathbb{R}_+ \) is a degree 2 polynomial of the form \( q(x) = x^\top A x \) for a positive definite matrix \( A \in \mathbb{R}^{m \times m} \). In particular for \( x, v \in \mathbb{R}^m \) it holds that

\[
q(x + tv) = at^2 + bt + c \quad \text{for } t \in \mathbb{R} \tag{45}
\]

with real constants \( a = (v^\top Av) \), \( b = (x^\top (A + A^\top)v) \) and \( c = x^\top Ax \). If \( v \) is non-zero, then the minimum \( t^* \) of the parabola (45) is given by the linear equation \( 2at^*_q + b = 0 \), for negative definite quadratic forms, the maximum can be computed similarly. One can therefore extend our algorithm to be applicable to a combination of definite quadratic forms and ReLU feed-forward neural networks.

For example let \( f \) be as in equation (2), \( q \) a definite quadratic form with \( m = n_0 \) and define the objective

\[
\tilde{f} : \begin{cases} 
\mathbb{R}^{n_0} & \rightarrow \mathbb{R} \\
x & \mapsto f(x) + q(x) \end{cases}.
\]

For a position and a direction \( x', v \in \mathbb{R}^{n_0} \) with \( \lim_{h \rightarrow 0} (\tilde{f}(x + hv) - \tilde{f}(x))/h < 0 \) we can use the ADVANCEMAX procedure to obtain \( t^* > 0 \) maximally such that \( t \mapsto \tilde{f}(x + tv) \) is affine on \([0, t^*]\). Then by equation (45), \( \tilde{f} \) is a parabola on \([0, t^*]\) whose extreme value position \( \tilde{t} \) restricted to this interval can be easily computed by a linear equation and a comparison against 0 and \( t^* \). By the assumption on \( v, \tilde{t} > 0 \) and \( \tilde{f}(x) > \tilde{f}(x + \tilde{t}v) \). This shows how to automatically and efficiently select the step size in this case. At position \( x \in \mathbb{R}^{n_0} \) with compatible activation pattern \( s \in S^C(x) \) the direction \( v \) for the next iteration needs to be selected to have negative inner product with the gradient

\[
\nabla_{\tilde{f},s}(x) := \nabla q(x) + \nabla_s \tag{46}
\]

within \( \tilde{D}_s(x) \) defined as in (44) for the state variable \( s \in S(x) \). If this is not possible, the compatible activation pattern \( s \in S(x) \) has to be changed to consider a different adjacent affine region of \( f \) as described in Section 3.2.4. Note that local minima of \( \tilde{f} \) are not necessarily on the vertices such that we have to resort to alternative iterations as described above.
4 Examples

In this section we want to present how our algorithm can be applied in practice. First we carried out experiments using neural networks with weights randomly sampled from a continuous distribution\(^2\). By Theorem 16 the resulting networks are almost surely regular. As noted in the introduction, the function \( f \) may attain infinitely small values and the DRLP algorithm would terminate in the \textsc{AdvanceMax} step. In Figures 11 and 12 we depict the objective function centered around the local minimum found by our algorithm for input dimensions \( n_0 = 1 \) and \( n_0 = 2 \).

Figure 11: After 18 iterations, our DRLP-Algorithm successfully finds a local minimum of a feed-forward ReLU neural network with random parameters and topology \( n_0, \ldots, n_{L+1} = (1, 50, 10, 10, 10, 10, 10, 10) \) starting at position \( x_0 = 0 \). The red vertical lines represent the iteration steps. The green vertical line indicates the final position at termination.

Figure 12: Surface plot of the objective function around a local minimum found with our DRLP-Algorithm applied to a random feed-forward ReLU neural network with topology \( (n_0, \ldots, n_{L+1}) = (2, 10, 10, 10, 10, 10, 10) \).

\(^2\)Here we ignore the fact, that finite precision floating point arithmetic only allows discrete distributions.
We now want to present some optimization problems that can be reformulated in a way such that deep ReLU programming is applicable. This is achieved by the construction of a neural net with a one-dimensional output that computes exactly the objective function that we want to minimize. We can then directly apply our solver algorithm.

### 4.1 Lasso penalized quantile regression

For \( p, N \in \mathbb{N} \), consider predictor and response samples \( x_1, \ldots, x_N \in \mathbb{N}^p \), \( y_1, \ldots, y_N \in \mathbb{R} \). Quantile regression as introduced in [4] aims to minimize

\[
\theta \mapsto \sum_{i=1}^{N} \rho_\alpha(y_i - \theta_0 - \sum_{j=1}^{p} \theta_j x_{ij})
\]

over \( \theta \in \mathbb{R}^{p+1} \) with \( \rho_\alpha : x \mapsto \alpha \max(x, 0) + (1-\alpha) \max(-x, 0) \), \( \alpha \in [0, 1] \). For more generality, we also incorporate a Lasso penalty term weighted by \( \lambda \geq 0 \) in our loss function \( L \):

\[
L : \begin{cases} \mathbb{R}^{p+1} & \rightarrow \mathbb{R}_\geq 0 \\ \theta & \mapsto \sum_{i=1}^{N} \rho_\alpha(y_i - \sum_{j=1}^{p} \theta_j x_{ij} - \theta_0) + \lambda \sum_{j=1}^{p} |\theta_j| 
\end{cases}
\]

(47)

For \( \lambda = 0 \) and \( \alpha = 0.5 \) we obtain the loss function of Least Absolute Deviation (LAD) regression [5]. Such optimization problems are usually solved using specialized linear programming algorithms such as the Barrodale-Roberts algorithm [6].

Note that \( L(x) = W_2 \text{ReLU}(W_1 x + b_1) \) with matrices

\[
W_1 = \begin{pmatrix} -\alpha X \\ (1-\alpha)X \\ \lambda Id_p \\ -\lambda Id_p \end{pmatrix}_{2(N+p) \times p+1}, \quad X = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ x_1^\top \\ \vdots \\ x_N^\top \end{pmatrix}_{N \times p+1}, \quad b_1 = \begin{pmatrix} (\alpha Y - (1-\alpha)Y) \\ 0_{2p \times 1} \end{pmatrix}_{2(N+p) \times 1}, \quad Y = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}_{N \times 1},
\]

and \( W_2 = (1, \ldots, 1) \in \mathbb{R}^{1 \times 2(N+p)} \) is a one hidden layer feed-forward neural network with layer widths \( (n_0, n_1, n_2) = (p+1, 2(N+p), 1) \). Each absolute value and \( \rho_\alpha \) function is replicated by a linear combination of two ReLU neurons. They are inducing the same hyperplanes such that the resulting network is not regular. However our algorithm can be applied if we only consider the activation of one representative for each such pair, see Section 3.2.6.

For a small sample data set we have tested our algorithm starting from a random position and compared its final position to the output of the \texttt{rq.fit.lasso} R function. Indeed our algorithm terminates with the correct minimizer, see Section C.

While our implementation will not provide an advantage over the existing specialized optimization methods, this demonstrates the universality of our algorithm. Furthermore, the proposed extension for definite quadratic forms from Section 3.2.6 would even cover the classical Lasso [7] where the loss function consists of a quadratic loss and a L1 penalty component.

### 4.2 Censored Least Absolute Deviation

The censored least absolute deviation (CLAD) estimator was introduced 1984 by J. Powell in [8]. For \( p, N \in \mathbb{N} \), and predictor and response samples \( x_1, \ldots, x_N \in \mathbb{N}^p \), \( y_1, \ldots, y_N \in \mathbb{R} \) it
aims to minimize the loss

\[ \mathcal{L} : \begin{cases} \mathbb{R}^p \rightarrow \mathbb{R} \\ \theta \mapsto \sum_{i=1}^N |y_i - \max(\langle \theta, x_i \rangle, 0)| \end{cases} \]  \tag{48}

Quantile regression from Section 4.1 involved no composition of nonlinearities and such that the structure can be easily understood and efficient linear programming based solver algorithms similar to the Barrodale-Roberts algorithm can be formulated. In contrast for the above loss function, a rewrite as a neural network requires two hidden layers since there is a composition of the absolute value function and the nonlinear mapping induced by the maximum. Furthermore the loss function \( \mathcal{L} \) is non-convex such that practical solvers for this problem are not apparent.

Powell originally proposed direct programming solvers which only consider function evaluations and ignore the underlying structure of the loss function. Later, Buchinsky [9] proposed his Iterative Linear Programming Algorithm (ILPA) which iteratively performs firstly a standard quantile regression for the previously selected uncensored observations and then checks for the newly estimated parameter \( \theta \) which observations are actually uncensored and will be considered in the next iteration, starting with all observations uncensored in the first iteration. Unfortunately, this procedure does not always converge and does not necessarily provide a local minimum as shown by Fitzenberger [10]. In this work, he also presents his BRCENS algorithm, an adaptation of the Barrodale-Roberts algorithm to CLAD regression iterating on the vertices induced by the hyperplanes corresponding to the absolute value and maximum functions in equation (48).

We can obtain this algorithm as a special case of deep ReLU programming by rewriting the loss function \( \mathcal{L} \) as a neural network. For the data matrix \( X := (x_1, \ldots, x_N)^\top \in \mathbb{R}^{N \times p} \) and the response vector \( Y := (y_1, \ldots, y_N)^\top \in \mathbb{R}^N \) it holds that

\[ \mathcal{L}(\theta) = \sum_{i=1}^N |y_i - \max(\langle \theta, x_i \rangle, 0)| \]

\[ = \sum_{i=1}^N \text{ReLU}(y_i - \text{ReLU}(\langle x_i, \theta \rangle)) + \sum_{i=1}^N \text{ReLU}(-y_i + \text{ReLU}(\langle x_i, \theta \rangle)) \]

\[ = (1, \ldots, 1)_{1 \times N} \text{ReLU}. (Y - \text{ReLU}.(X\theta)) + (1, \ldots, 1)_{1 \times N} \text{ReLU}. (-Y + \text{ReLU}.(X\theta)) \]

\[ = (1, \ldots, 1)_{1 \times 2N} \text{ReLU}. (W_2\text{ReLU}. (W_1\theta) + b_2), \]

where

\[ W_2 = \begin{pmatrix} 1_N \\ -\text{I}_N \end{pmatrix}_{2N \times N}, W_1 = \begin{pmatrix} \text{Y} \end{pmatrix}_{2N \times 1}, b_2 = \begin{pmatrix} \text{Y} \end{pmatrix}_{2N \times 1}. \]

In particular, \( \mathcal{L} \) can be written as a \( L = 2 \) hidden layer neural network in the form \( (2) \) with \( (n_0, n_1, n_2, n_3) = (p, N, 2N, 1) \) such that we can apply our DRLP algorithm.

Note that for \( \theta \in \mathbb{R}^{n_0} \), the function \( x \mapsto \langle \theta, x \rangle \) is a feed-forward neural network of the form \( (2) \) with \( L = 1, W_1 = \theta^\top, W_2 = 1, b_1 = 0 \) and \( b_2 = 0 \). In this sense the application of the DRLP algorithm to the above problem can be seen as training the first layer parameters of a simple neural network with absolute deviation loss. We will leverage this idea to train the first layer of more general, possibly deep networks in the next section with deep ReLU programming.
4.3 First layer L1 loss-optimization for feed-forward neural networks

In this section, we want to use deep ReLU programming train the first layer parameters of a feed-forward neural network $f \in \mathcal{F}$ as in equation [2]. To express the variability of the first layer parameters $\theta = (W, b) \in \mathbb{R}^{n_1 \times n_0} \times \mathbb{R}^{n_0} \equiv \Theta$ we define

$$f_\theta(x) = W_{L+1}g^{(L)}(\cdots g^{(2)}(\text{ReLU} \cdot (W_\theta x + b_\theta)) + b_{L+1}$$

for $x \in \mathbb{R}^{n_0}$. Note that only the first layer is different from $f$ and the only difference is that the weight and bias parameters are replaced by those specified by $\theta$. In particular $f_\theta = f$ for $\theta = (W_1, b_1)$. Using this notation, the L1 loss for given training samples $(x_1, y_1), \ldots, (x_N, y_N) \in \mathbb{R}^{n_0} \times \mathbb{R}$, $N \in \mathbb{N}$ is given by

$$\mathcal{L} : \left\{ \begin{array}{c} \Theta \rightarrow \mathbb{R} \\ \theta \mapsto \sum_{i=1}^N |f_\theta(x_i) - y_i| \end{array} \right.$$ 

We now want to construct a feed-forward neural network $f_L: \mathbb{R}^{n_1(n_0+1)} \rightarrow \mathbb{R}$ that computes this loss $f_L(\bar{\theta}) = \mathcal{L}(\theta)$ for $\theta \in \Theta$. Here we denote by $\bar{\theta} \in \mathbb{R}^{n_1(n_0+1)}$ the rearrangement of the entries of $\theta = (W_\theta, b_\theta) \in \mathbb{R}^{n_1 \times n_0} \times \mathbb{R}^{n_1}$ into a vector according to the rule

$$\bar{\theta}_k = \begin{cases} (W_\theta)_{ij} & \text{if } k \leq n_1n_0 \text{ with } (i,j) \in \{1,\ldots,n_1\} \times \{1,\ldots,n_0\} \text{ such that } k = in_0 + j \\ b_{k-n_1n_0} & \text{if } k > n_1n_0 \end{cases}$$

for indices $k \in \{1,\ldots,n_1(n_0+1)\}$. This means that we first fill the parameters of the weight matrix $W_\theta$ row-wise into the rearrangement $\bar{\theta}$, followed by the entries of the bias vector $b_\theta$. We also need to introduce the corresponding $(Nn_1) \times (n_0 + 1)$ data matrix $M_X = (M(x_1)\cdots M(x_N))\top$, where

$$M(x_i) = \left( \begin{array}{cccccccc} x_i^T & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & x_i^T & 0 & \cdots & 0 & 1 \end{array} \right) \text{ for } i \in \{1,\ldots,N\}.$$ 

Note that for $i \in \{1,\ldots,N\}$, $M(x_i)\bar{\theta} = W_\theta x_i + b_\theta$ expresses the first layer affine transformation of $f_\theta$ applied to the predictor $x_i$ such that $M_X\bar{\theta}$ is a long vector containing groups of the entries $M(x_1)\bar{\theta}, \ldots, M(x_N)\bar{\theta}$. To each of these groups, also the remaining tranformations of $f_\theta$ need to be applied. In order to express this notationally conveniently, for $a,b,l \in \mathbb{N}$ we define the $l$-fold diagonal replication $D_l(W)$ of $W \in \mathbb{R}^{a \times b}$ and $l$-fold stacked vector of $b \in \mathbb{R}^a$

$$D_l(W) = \left( \begin{array}{cccc} W & 0 & \cdots & 0 \\ 0 & W & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & W \end{array} \right) \in \mathbb{R}^{la \times lb}, \quad T_l(b) = \left( \begin{array}{c} b \\ b \\ \vdots \\ b \end{array} \right) \in \mathbb{R}^{la}$$

Using the notation $g_N^{(i)}: \mathbb{R}^{Nn_i-1} \rightarrow \mathbb{R}^{Nn_i}$, $x \mapsto \text{ReLU} \cdot (D_N(W_i)x + T_N(b_i))$ for $i \in \{2,\ldots,L\}$ and $Y = (y_1,\ldots,y_N) \in \mathbb{R}^N$ it follows that

$$\mathcal{L}(\theta) = \sum_{i=1}^N |f_\theta(x_i) - y_i| = \sum_{i=1}^N |W_{L+1}g_N^{(L)}(\cdots g_N^{(2)}(\text{ReLU} \cdot (W_\theta x_i + b_\theta)) + b_{L+1} - y_i|$$

34
\[(1, \ldots, 1)_{1 \times N} \text{ReLU} \left( -Y + T_N(b_{L+1}) + D_N(W_{L+1})g_N^{(L)} \circ \cdots \circ g_N^{(2)} \left( \text{ReLU}(M_X\theta) \right) \right) \\
+ (1, \ldots, 1)_{1 \times N} \text{ReLU} \left( Y - T_N(b_{L+1}) - D_N(W_{L+1})g_N^{(L)} \circ \cdots \circ g_N^{(2)} \left( \text{ReLU}(M_X\tilde{\theta}) \right) \right) \\
= (1, \ldots, 1)_{1 \times 2N} \text{ReLU} \left( \left( -Y + T_N(b_{L+1}) \right) + \left( D_N(W_{L+1}) - D_N(W_{L+1}) \right)g_N^{(L)} \circ \cdots \circ g_N^{(2)} \left( \text{ReLU}(M_X\tilde{\theta}) \right) \right)
\]

It follows that a network \( f_L \) that computes \( \mathcal{L} \) can be realized using \( L + 1 \) hidden layers of widths \((n_0, \ldots, n_{L+1}) = (n_1(n_0 + 1), Nn_1, \ldots, Nn_L, 2N) \). The predictor and response samples affect the first layer matrix and the \( L + 1 \)-th layer bias vector respectively.

Note that for \( NN_1 > n_1(n_0 + 1) \) the origin \( 0 \in \mathbb{R}^{n_1(n_0 + 1)} \) of the above constructed network \( f_L \) will generally be a vertex that is not regular because the first layer bias vector is 0 such that every of the \( NN_1 \) neurons in the first layer may change their activation locally around the origin. Thus, our algorithm will only be applicable as long as the produced iteration positions are regular vertices.

We want to note that similar to the above construction, also the L1 loss as a function of the weight and bias parameters of another layer with index \( \ell \in \{1, \ldots, L\} \) can be written as a feed forward neural for fixed parameters in the remaining layers. In this case, the transformed predictors \( g^{(\ell-1)} \circ \cdots \circ g^{(1)}(x_i), i \in \{1, \ldots, N\} \) take the role of the predictors above and induce the matrix \( M_X \). However, the ReLU activation function applied during this transformation often causes a linear dependence structure among the transformed predictors such that whenever the ReLU arguments are zero for some neurons, automatically those of other neurons are also zero, for example in the case multiple transformed predictors being completely zero. This causes corresponding vertices to be non-regular, thus rendering our algorithm unusable. Due to this reason, we restricted to first layer’s parameter training here.

**Example 23.** For the topology \( L = 3 \), \((n_0, n_1, n_2, n_3, n_4) = (4, 5, 4, 2, 1) \) we sample the network weights and bias vectors from a uniform distribution and apply our DRLP algorithm to the function \( \mathcal{L} \) above re-written as a neural network for 500 training samples also sampled from a uniform distribution. Our algorithm produces a sequence \( n_1(n_0 + 1) = 25 \)-dimensional estimates for the first layer network parameters with strictly decreasing loss depicted in Figure 13, a property that most other training methods do not guarantee. After a finite number of steps convergence is reached.

Interestingly, the decay in training loss is approximately exponential. Note that since our algorithm iterates on vertices, this might empirically provide information about the density of vertices around local minima of the loss \( \mathcal{L} \). An interesting question to ask is whether this decay graph provides information about the quality of the converged local minimum: Assuming an exponential decay, maybe a overly fast convergence might indicate premature trapping in a suboptimal undesirable local minimum. A further question for future analysis is whether such vertex density information can be used to improve the step size control in standard gradient descent in the sense that a high density requires small step sizes and vice versa.

The widths of the hidden layers of \( f_L \) scale with \( N \) such that one step of our algorithm is of order \( O(N^2) \) for increasing \( N \). This undesired blowup is caused by the large replication
matrices and stacked vectors used in the network $f_C$ above which compute the parallel evaluation of the original network $f$ on the predictor samples. Instead, one could consider a modification of our algorithm adapted to such multiple evaluations of the same function $f$ on different samples. This way, the quadratic dependence in $N$ could be improved to a linear dependence. A further modification could be an adaptation for quadratic loss instead of $L_1$ loss following the discussion on definite quadratic forms in Section 3.2.6. We will pursue these ideas in future work.

5 Comparison

In this section we want to discuss key similarities and differences of deep ReLU linear programming with linear programming and gradient descent.

5.1 DRLP and Linear Programming

As noted in Section 1.2.1, deep ReLU programming can be seen as an extension of linear programming because the objective function $f$ is piece-wise affine on convex domains

$$R_s := \{ x \in \mathbb{R}^n \mid s \in \mathcal{S}^C(x) \}, \quad s \in \mathcal{S}$$

(49)

each of which has the form of a feasible region in linear programming. In this sense linear programming focusses only on an affine objective on the feasible region while deep ReLU linear programming considers multiple regions $R_s$ with their own affine objective function $f|_{R_s}$ for $s \in \mathcal{S}$. At the boundaries of these regions their objective functions are equal since the feed-forward ReLU neural network $f$ is continuous. This means that deep ReLU programming can be seen as a patchwork of many linear programming problems which satisfy a continuity condition and the algorithmic difference of our presented algorithm is the ability

Figure 13: Training loss for the setup explained in Example 23 across different epochs.
to change the considered region $R_s$ by switching $s \in S$ in a computationally efficient way that allows to reuse previously computed axes.

Section 1.2.1 also showed that every linear program can be rewritten as a deep ReLU program by constructing a ReLU neural network that has the solution of the linear program as its global minimum. However, this is only a theoretical statement and such a rewrite is of no practical use since our algorithm does not provide any computational runtime advantage over existing solvers for linear programs. Instead, the advantage of our algorithm is the ability to iterate through different affine regions to find local minima of feed-forward ReLU neural networks, whereas in the formulation of linear programming, the feasible region is not left.

For linear programming problems there exist inner point methods which have polynomial runtime for the required number of iterations in terms of the number of conditions and the dimensionality. Their iteration steps are within the feasible region of the considered linear program. In contrast, for deep ReLU programming problems it is very unlikely that the currently considered affine region of the objective function has a local minimum of $f$ as one the vertices on its boundary. Instead probably many different affine regions have to be traversed such that it is questionable whether a transfer of ideas from inner point methods for linear programming to deep ReLU programming is beneficial.

5.2 DRLP and Gradient Descent

Given a differentiable function $\varphi : \mathbb{R}^{n_0} \to \mathbb{R}$ and a starting point $x_0 \in \mathbb{R}^{n_0}$, gradient descent-like algorithms iteratively compute a sequence of points $x_0, x_1, \ldots \in \mathbb{R}^{n_0}$ with an update rule of the form

$$x_{i+1} = x_i - \eta_i \nabla \varphi(x_i), \quad i \in \mathbb{N}$$

for positive step size parameters $\eta_0, \eta_1, \ldots \in \mathbb{R}$. This is motivated by the fact that for every $x \in \mathbb{R}^{n_0}$

$$\mathbb{R}^{n_0} \setminus \{0\} \to \mathbb{R}, v \mapsto \lim_{t \to 0} \frac{\varphi(x + tv) - \varphi(x)}{t \|v\|_2} = \langle -\frac{v}{\|v\|_2}, \nabla \varphi(x) \rangle$$

is minimized by $v = -\alpha \nabla \varphi(x)$, $\alpha > 0$ by Cauchy-Schwartz inequality. However, this is only the optimal direction to decrease $\varphi$ in an infinitely small neighbourhood around $x$ such that the step size in equation (50) has to be chosen with care. The DRLP-algorithm has a similar update rule, however the arguments generated in its iterations are placed on the boundaries between affine-regions where the gradient is not defined and instead of the gradient, an axis separating such regions is used as the direction for the next iteration step.

The step size is automatically determined maximally, such that the assumed affine behaviour is still valid. This is achieved by exploiting the structure of feed-forward ReLU neural networks. In contrast to standard gradient descent-like algorithms this adaptation to such functions allows the update procedure to decrease the function value at every iteration. This is a very strong property since every new iteration will either strictly decrease the objective function or our algorithm stops. For regular networks with vertices, this stopping condition explained in Section 3.2.4 exactly determines local minima such that for these networks, our algorithm finds the exact position of a local minimum after a finite number of iterations.
Other gradient descent-like algorithms don’t have these properties but they are not restricted to piece-wise affine functions.

Whenever it is defined, the gradient $\nabla f(x)$ for a feed forward neural network $f$ as in equation (2) at position $x \in \mathbb{R}^{n_0}$ can be computed in $O(\sum_{l=0}^{L} n_{l+1} n_l)$ by the chain rule in calculus and also the update rule (50) is of this order. In contrast, we have shown in Section 3.2.5 that one iteration in our DRLP algorithm is of order $O(\sum_{l=0}^{L} n_{l+1} n_l + n_0^2)$. In particular, when $n_0^2$ is dominated by one of $n_{i+1} n_i$, $i \in \{0, \ldots, L\}$, one iteration in our algorithm is of the same order as in gradient descent for $\max(n_0, \ldots, n_{L+1}) \to \infty$. Concerning the required number of iterations until convergence it is plausible that for large $n_0, \ldots, n_L$, the automatically selected step size in deep ReLU programming is small because the input space $\mathbb{R}^{n_0}$ is split into many small regions, each with its own affine behaviour. This can slow down the minimization progress, especially at the beginning when large step sizes in equation (50) are appropriate. In contrast, at later stages of the minimization process, when a small step size has to be chosen in gradient descent-like algorithms to further minimize the objective function, the automatic step size-control of the DRLP-Algorithm can be beneficial.

Furthermore the use of an axis that is separating affine regions of $f$ instead of its gradient can reduce the number iteration steps needed in situations as depicted in Figure 14. In this case, gradient descent-like algorithms will oscillate between two affine regions because for every new iteration, they will follow direction of the gradient and will eventually overjump the region boundary. While modifications such as the momentum method can circumvent this problem to some extent, the DRLP algorithm would first find the intersection point and then follow the region separating axis in the direction that decreases the function value until a neuron activation changes, thus completely avoiding unnecessary oscillations.

![Figure 14](image)

**Figure 14:** Qualitative comparison of possible behaviours of the DRLP algorithm (green) versus other gradient descent-like algorithms (red) in three dimensions (two-dimensional input space) in starting at the same position $x_0 \in \mathbb{R}^2$. The dotted lines connect the points corresponding to subsequent iteration steps. The black lines represent the boundaries of the affine regions of $f$.

## 6 Summary

We have introduced the class of deep ReLU programming problems as non-convex optimization problems of finding a local minimum of a real-valued feed-forward neural network $f$ with ReLU activation functions. These functions are piece-wise affine on domains which
are given by sets of linear inequalities. In particular, if restricted to one such domain, the resulting minimization problem has the form of a linear programming problem where the restricted domain takes the role of the feasible region. Hence, deep ReLU programming can be seen as an extension of linear programming with the difference that there are multiple feasible regions, each with its own affine objective function such that the overall function is continuous.

We developed notation and theory suitable for the analysis of these functions in Section 2. For every activation pattern \( s \), the concept of compatibility allows to precisely describe the input domain where replacing the ReLU activation functions by the hard-coded activation pattern \( s \) preserves the output and all intermediate layer values of \( f \), i.e. where the network \( f \) can be described by fixed affine layer maps induced by \( s \). We defined regularity for neural networks and showed that for independent continuous distributions of its parameters, the resulting network is almost surely regular. For such regular networks, we further presented a necessary and sufficient condition for \( x \) being a local minimum based on concepts specific to activation patterns compatible with \( x \).

These theoretical considerations lay the foundation for the discussion on deep ReLU programming solver algorithms in Section 3. We first described algorithmic primitives and then combined them to our DRLP algorithm, an iterative solver for deep ReLU programming problems. In its first phase it first finds a vertex and then iterates on vertices while changing the affine region and keeping track of a compatible activation pattern and a set of critical neurons that define the region separating axes to proceed with. The automatic step size selection exploits the piece-wise constant gradient and ensures that the objective function cannot increase across iterations. Our algorithm stops based on the local minimum condition for regular networks we presented previously. For an \( L \) layer network of widths \( n_0, \ldots, n_L \) we showed that every iteration is of computational complexity \( \mathcal{O}(\sum_{i=1}^{L} n_i n_{i-1} + n_0^2) \), where the first and the second terms reflect the number of parameters and simplex tableau size, such that this complexity order is optimal for an extension of the simplex algorithm to ReLU neural networks. We further proposed modifications for better numeric stability, other activation functions, alternatives to vertex iterations and generalizations for mixtures of feed forward ReLU nets and quadratic forms as objective functions.

In Section 4 we presented optimization problems that can be rewritten in the form of a deep ReLU programming problem. We showed that our algorithm can be applied in lasso penalized quantile regression, censored least absolute deviation and even for training of the first layer of a deep ReLU network using L1 loss. For the first and second application in this list, there exist specialized linear programming solver algorithms that can be seen as special cases of our DRLP algorithm, for the third example one usually resorts to standard neural network training algorithms not specifically designed to take advantage of the piece-wise affine structure of the ReLU network to be trained. Here our algorithm provides a simplex-like iteration on vertices of the loss function such that the decreases in every step. In future work it will be interesting to apply deep linear programming for training of not only the first layer of ReLU neural networks but also other layers iteratively. This would yield a training procedure which monotonically decreases the L1 training loss across iterations, a major advantage compared to other optimization methods.
We finally compared our algorithm to linear programming and gradient descent variants in Section 5. The overall computational complexity for an iteration in our algorithm of $O(n_0^2 + \sum_{i=1}^{L} n_in_{i-1})$ allows it to be considered as a new competitive optimization tool when the objective function can be written as a feed-forward ReLU neural network.
A Proofs and auxiliary results

A.1 Compatibility

Proof of Theorem 2. Fix $s \in S^C(x)$. For $(l, j) \in \mathcal{I}$, if $s_{lj} = 0$ then by equation (14) \( \text{sign}(A(x)_{lj}) = H(x)_{lj} \in \{0, -1\} \) and therefore ReLU\( (A_{lj}(x)) = 0 \). Similarly $s_{lj} = 1$ implies ReLU\( (A(x)_{lj}) = A(x)_{lj} \). Hence $g^{(1)}(x) = \tilde{g}^{(1)}_{l_1}(x)$ and for $l \in \{2, \ldots, L\}$, \[ g^{(l)} \circ \cdots \circ g^{(1)}(x) = \text{ReLU}(A(x)_l) = \text{diag}(s_l)A(x)_l = \tilde{g}^{s_l} \circ g^{(l-1)} \circ \cdots \circ g^{(1)}(x). \]

A recursive application of the above formula yields $A(x) = \tilde{A}_s(x)$. 

Now fix $s \in \tilde{S}^C(x)$. By Definition 15 $s$ is compatible with $\tilde{H}_s(x)$. In particular for $(l, j) \in \mathcal{I}$, $s_{lj} = 0$ implies sign\( (\tilde{A}_s(x)_{lj}) = \tilde{H}_s(x)_{lj} \in \{-1, 0\} \) and ReLU\( (\tilde{A}_s(x)_{lj}) = 0 \). Similarly $s_{lj} = 1$ implies ReLU\( (\tilde{A}_s(x)_{lj}) = \tilde{A}_s(x)_{lj} \). This implies \[ \forall l \in \{1, \ldots, L\} \quad \text{ReLU}(\tilde{A}_s(x)_l) = \text{diag}(s_l)\tilde{A}_s(x)_l \quad (51) \]

We now prove $A(x) = \tilde{A}_s(x)$ by induction. Note that $A(x)_1 = W_{1x} + b_1 = \tilde{A}_s(x)_1$. For the induction step assume $A(x)_l = \tilde{A}_s(x)_l$ for some $l \in \{1, \ldots, L-1\}$. Then by equation (51) \[ g^{(l)} \circ \cdots \circ g^{(1)}(x) = \text{ReLU}(A(x)_l) = \text{ReLU}(\tilde{A}_s(x)_l) = \text{diag}(s_l)(\tilde{A}_s(x)_l) = \tilde{g}^{s_l}_l \circ \cdots \circ \tilde{g}^{s_1}_1(x). \]

This implies that $A(x)_{l+1} = W_{l+1}g^{(l)} \circ \cdots \circ g^{(1)}(x) + b_{l+1} = W_{l+1}\tilde{g}^{s_l}_l \circ \cdots \circ \tilde{g}^{s_1}_1 + b_{l+1} = \tilde{A}_s(x)_{l+1}$. An induction on $l$ yields $A(x) = \tilde{A}_s(x)$. Therefore we have shown that $s \in S^C(x) \cup \tilde{S}^C(x)$ implies $A(x) = \tilde{A}_s(x)$, hence $H(x) = \tilde{H}_s(x)$ and therefore $S^C(x) = \tilde{S}^C(x)$. \( \square \)

A.1.1 Local Behaviour

Lemma 24. For all $x \in \mathbb{R}^n_0$, there exists $\varepsilon > 0$ such that for all $y \in B_\varepsilon(x)$ it holds that $S^C(y) \subset S^C(x)$.

Proof. Let $\varepsilon > 0$ such that $H(x)_{lj} = H(y)_{lj}$ for $y \in B_\varepsilon(x)$ and $(l, j) \in \mathcal{I} \setminus C(x)$. By Lemma 5 $H(x)_{lj} = 0$ for $(l, j) \in C(x)$ such that for all $y \in B_\varepsilon(x)$ any $s \in S$ compatible with $H(y)$ is also compatible with $H(x)$.

This means that for every input $x \in \mathbb{R}^n_0$ there exists a neighbourhood such that for all $y$ in this neighbourhood all $s$ compatible with $H(y)$ are also compatible with $H(x)$.

Lemma 25. For all $x \in \mathbb{R}^n_0$, it holds that $C(x) \subset C^*(x)$ and for all $s \in S^C(x)$, $\tilde{C}_s(x) \subset C^*(x)$. 

41
Proof. The latter part follows immediately from equation (18). For the first part, let \((i, j) \in C(x)\). This implies that for every \(\varepsilon > 0\) there exists \(y_\varepsilon \in B_\varepsilon(x)\) with \(H(y_\varepsilon)_{ij} \neq 0 = H(x)_{ij}\). For \(\varepsilon\) small enough \(\mathcal{S}_C(y_\varepsilon) \subset \mathcal{S}_C(x)\) by Lemma 24, in particular \(\tilde{H}_s(x)_{ij} = H(x)_{ij} \neq H(y_\varepsilon)_{ij}\). Since \(\varepsilon\) can be chosen arbitrarily small and there are finitely many possibilities for \(s \in \mathcal{S}\) there exists some \(s \in \mathcal{S}\) such that for all \(\varepsilon > 0\) there exists some \(y_\varepsilon \in B_\varepsilon(x)\) with \(\tilde{H}_s(x)_{ij} \neq \tilde{H}_s(y_\varepsilon)_{ij}\). In other words \((i, j) \in \tilde{C}_s(x)\). Since this was for arbitrary \((i, j) \in C(x)\) it follows that \(C^*(x) \supset C(x)\).

Proof of Lemma 14. If \(v^* \in \text{Ker}_s H(x)\), then there exists \(\varepsilon > 0\) such that for all \((l, j) \in \tilde{C}_s(x)\) and \(t \in (-\varepsilon, \varepsilon)\) \(\tilde{H}_s(x + tv^*)_{ij} = \tilde{H}_s(x)_{ij} = 0\). In particular for all \(t \in (-\varepsilon, \varepsilon)\) it holds that \(\tilde{A}_s(x + tv^*)_{ij} = 0\). By affinity of \(z \mapsto \tilde{A}_s(x)\), it holds that \(\tilde{A}_s(x + v^*)_{ij} = 0\). For the other direction assume that \(v^* \in \mathbb{R}^{n_0}\) with \(\tilde{A}_s(x + v^*)_{ij} = 0\) for \((l, j) \in \tilde{C}_s(x)\). Let \(\varepsilon > 0\) such that for all \(v \in B_{\varepsilon^2}(0)\) and \((l, j) \in I \setminus \tilde{C}_s(x)\) it holds that \(\tilde{H}_s(x + v)_{ij} = \tilde{H}_s(x)_{ij}\). This is possible by equation (17). Then for all \(t \in \mathbb{R}^{n_0}\) such that \(tv^* \in B_{\varepsilon^2}(0)\) it holds that \(\tilde{H}_s(x + tv^*) = \tilde{H}_s(x)\) since also for indices \((l, j) \in \tilde{C}_s(x)\) it holds that \(\tilde{H}_s(x + tv) = \tilde{H}_s(x)\) by Lemma 5.

Proof of Lemma 25. By Corollary 4 \(\{v \in \mathbb{R}^{n_0} \mid H(x + v) = H(x)\} = \{v \in \mathbb{R}^{n_0} \mid \tilde{H}_s(x + v) = \tilde{H}_s(x)\}\). Let \(\varepsilon > 0\) small enough such that \(H_s(x + v)_{ij}\) is constant for indices \((i, j) \in I \setminus \tilde{C}_s(x)\) and \(v \in B_{\varepsilon^2}(0)\). This is possible by equation (17). For all other indices \((i, j) \in \tilde{C}_s(x)\), the fixed activation hyperplane pattern \(\tilde{H}_s(x)_{ij} = 0\) by Lemma 5. It follows that \(B_{\varepsilon^2}(x) \cap \{v \in \mathbb{R}^{n_0} \mid \tilde{H}_s(x + v) = \tilde{H}_s(x)\} = B_{\varepsilon^2}(x) \cap \tilde{C}_s(x)\{v \in \mathbb{R}^{n_0} \mid \tilde{A}_s(x + v)_{ij} = 0\}\). Lemmas 9 and 8 complete the proof.

A.2 Regularity

A.2.1 Basic results

Proof of Lemma 14. Let \(s \in \mathcal{S}_C(x)\). By Lemma 25 \(\tilde{C}_s(x) \subset C^*(x)\) such that Lemma 11 implies \(|C^*(x)| = n_0 - \text{dim}^H(x) \leq |\tilde{C}_s(x)| \leq |C^*(x)|\). In particular \(C^*(x) = \tilde{C}_s(x)\).

Proof of Lemma 15. Lemma 14, equation (23) and Lemmas 8, 9, 12 imply that
\[
|\tilde{C}_s(x)| = n_0 - \text{dim}^H(x) = n_0 - \text{dim}(\text{Ker}_s^H(x)) = n_0 - \text{dim}(\text{Ker}_s^H(x))
= n_0 - \text{dim}\left(\bigcap_{(l,j) \in \tilde{C}_s(x)} \{v \in \mathbb{R}^{n_0} \mid \tilde{A}_s(x + v)_{ij} = 0\}\right)
= n_0 - \text{dim}\left(\bigcap_{(l,j) \in \tilde{C}_s(x)} \{v \in \mathbb{R}^{n_0} \mid \langle v, \tilde{v}_{s,l,j} \rangle = 0\}\right) = \text{dim}(\text{span}((\tilde{v}_{s,l,j})_{(l,j) \in \tilde{C}_s(x)}))
\]

A.2.2 Probabilistic results on regular vertices

Below we use the notation and concepts of Section 2. For \(s \in \mathcal{S}\), \((l, j) \in I\) let
\[
H^s_{ij} := \left\{ x \in \mathbb{R}^{n_0} \mid \langle x, \tilde{v}_{s,l,j} \rangle + \left(\sum_{i=1}^{l-1} W_i \text{diag}(s_{i-1}) \cdots W_{i+1} \text{diag}(s_i) b_i\right)_j + b_{lj} = 0 \right\}.
\]
Note that \( H^*_i = \{ x \in \mathbb{R}^{n_0} \mid \bar{H}_i(x)_{ij} = 0 \} \).

**Lemma 26.** Assume the \( \sum_{i=2}^{L+1} n_i n_{i-1} \) weight parameters of the matrices \( W_1, \ldots, W_L \) and the \( \sum_{i=2}^{L+1} n_i \) bias parameters of the vectors \( b_1, \ldots, b_L \) of the neural network \( f \) are sampled from a distribution such that conditionally on the weight parameters, the bias parameters are independent with a conditional marginal distribution that assigns probability zero to all finite sets. Then the following holds almost surely: For all \( n \in \mathbb{N}, s_1, \ldots, s_n \in \mathcal{S}, (l_1, j_1), \ldots, (l_n, j_n) \in \mathcal{I} \) with non-zero normal vectors \( \tilde{v}_{s_1, l_1, j_1}, \ldots, \tilde{v}_{s_n, l_n, j_n} \) the hyperplanes \( H_{i_1, j_1}^{s_1}, \ldots, H_{i_n, j_n}^{s_n} \) defined by equation (52) satisfy either \( \cap_{i=1}^{n} H_{i, j_i}^{s_i} = \{ \} \) or the normal vectors \( \tilde{v}_{s_1, l_1, j_1}, \ldots, \tilde{v}_{s_n, l_n, j_n} \) are linearly independent.

**Proof.** We proceed by induction. For \( n = 1 \) the claim is obviously true. Now assume the claim is true for \( n = n^* \in \mathbb{N} \) and let \( s_1, \ldots, s_{n^*+1} \in \mathcal{S}, (l_1, j_1), \ldots, (l_{n^*+1}, j_{n^*+1}) \in \mathcal{I} \). Conditionally on the weight parameters, the normal vectors

\[
\tilde{v}_{s_1, l_1, j_1}, \ldots, \tilde{v}_{s_{n^*+1}, l_{n^*+1}, j_{n^*+1}}
\]

are almost surely constant and by assumption the bias parameters are independent with a continuous distribution. We distinguish two cases:

1. Conditionally on the weight parameters, at least one of the above normal vectors is equal to zero.

2. Conditionally on the weight parameters, all of the above normal vectors are nonzero. In this case let \( k \in \{1, \ldots, n^* + 1\} \) with \( k = \max_{i \in \{1, \ldots, n^*\}} l_i \) and let \( J := \{1, \ldots, n^* + 1\} \setminus \{k\} \). Now we condition on the weight parameters and on \( b_{l_i, j_i}, i \in J \). In this case, the bias vector \( b_{l_k, j_k} \) has a continuous distribution. If \( \cap_{i \in J} H_{i, j_i}^{s_i} \neq \{\} \) then by induction assumption the vectors \( \tilde{v}_{s_i, l_i, j_i}, i \in J \) are linearly independent. If \( \tilde{v}_{s_i, l_i, j_i}, i \in \{1, \ldots, n^*\} \) are linearly dependent, \( \cap_{i=1}^{n^*+1} H_{i, j_i}^{s_i} \neq \{\} \) implies that \( \cap_{i \in J} H_{i, j_i}^{s_i} \subset H_{l_k, j_k}^{s_k} \) and this is only possible when \( b_{l_k, j_k} \) attains a specific value \( b^* \in \mathbb{R} \) which almost surely does not happen.

The above discussion shows that with probability zero the normal vectors in equation (53) are all non-zero, linearly dependent and the hyperplanes satisfy \( \cap_{i=1}^{n^*+1} H_{i, j_i}^{s_i} \neq \{\} \). Since there are only finitely many choices for \( s_1, \ldots, s_{n^*+1} \in \mathcal{S}, (l_1, j_1), \ldots, (l_{n^*+1}, j_{n^*+1}) \in \mathcal{I} \) the claim holds almost surely for \( n = n^* + 1 \). \( \square \)

**Lemma 27.** If \( f \) is not regular then there exist \( n \in \mathbb{N}, s_1, \ldots, s_n \in \mathcal{S}, (l_1, j_1), \ldots, (l_n, j_n) \in \mathcal{I} \) with non-zero normal vectors \( \tilde{v}_{s_1, l_1, j_1}, \ldots, \tilde{v}_{s_n, l_n, j_n} \) and hyperplanes \( H_{i_1, j_1}^{s_1}, \ldots, H_{i_n, j_n}^{s_n} \) defined by equation (52) such that \( \cap_{i=1}^{n} H_{i, j_i}^{s_i} \neq \{\} \) and the normal vectors \( \tilde{v}_{s_1, l_1, j_1}, \ldots, \tilde{v}_{s_n, l_n, j_n} \) are linearly dependent.

**Proof.** If \( f \) is not regular then there exists a point \( x^* \in \mathbb{R}^{n_0} \) that is not regular. In particular \( df^H(x^*) \neq n_0 - |C^*(x)| \). We distinguish two cases.
1. If there exists \( s \in \mathcal{S}^C(x) \) with \( C^*(x) = \tilde{C}_s(x) \) then \( \text{df}^H(x^*) \neq n_0 - |\tilde{C}_s(x)| \) such that Lemma 11 implies that \( \text{dim}(\text{Ker}^H(x)) = \text{df}^H(x^*) > n_0 - |\tilde{C}_s(x)| \). In particular, \( \tilde{v}_s, l_1, j_1, \ldots, \tilde{v}_s, l_n, j_n \) are linearly dependent by Lemmas 9 and 8. Lemma 13 shows that they are non-zero.

2. If \( C^*(x) \neq \tilde{C}_s(x) \) for all \( s \in \mathcal{S}^C(x) \), then let \( \bar{s} \in \mathcal{S}^C \). By Lemma 25 there exists \((l^*, j^*) \in C^*(x) \setminus \tilde{C}_s(x) \neq \{ \} \). Since \((l^*, j^*) \in C^*(x) \) there exists \( s' \in \mathcal{S}^C(x) \) with \((l^*, j^*) \in \tilde{C}_{s'} \). Using the notation of equation (52), Lemmas 8 and 9 imply

\[
\cap_{(l,j) \in \tilde{C}_s(x)} H_{ij}^s = \cap_{(l,j) \in \tilde{C}_{s'}(x)} H_{ij}^{s'} \neq \{ \}.
\]

In particular \( \cap_{(l,j) \in \tilde{C}_s(x)} H_{ij}^s \cap H_{ij}^{s'} \neq \{ \} \) and \((\tilde{v}_{s,l,j})_{(l,j) \in \tilde{C}_s(x)}, \tilde{v}_{s',l',j'} \) are linearly dependent. Lemma 13 shows that they are non-zero.

Proof of Theorem 10. The claim follows from Lemmas 26 and 27.

A.3 Feasible directions

Lemma 28. For \( x \in \mathbb{R}^{n_0} \) it holds that \( s \in \mathcal{S}^C(x) \) if and only if \( \tilde{A}_s(x)_{lj}(s_{lj} - 0.5) \geq 0 \) for all \((l, j) \in \mathcal{I}) \).

Proof. This follows from equations (13) and (12).

Proof of Lemma 14. Let \( v^* \in \mathbb{R}^{n_0} \) with \( \tilde{A}_s(x + v^*)_{lj}(s_{lj} - 0.5) \geq 0 \) for all \( \tilde{C}_s(x) \). Choose \( \varepsilon^* > 0 \) small enough such that for indices \((l, j) \in \mathcal{I} \setminus \tilde{C}_s(x) \), the fixed activation hyperplane pattern is constant, i.e. such that for all \( v \in B_{\varepsilon^*}(0) \) it holds that \( \tilde{H}_s(x + v)_{lj} = \tilde{H}_s(x)_{lj} \). This is possible by equation (11). Now let \( \varepsilon > 0 \) such that \( \varepsilon v^* \in B_{\varepsilon^*}(0) \). In particular at indices \((l, j) \in \mathcal{I} \setminus \tilde{C}_s(x) \) \( \tilde{H}_s(x + \varepsilon v^*)_{lj} = \tilde{H}_s(x)_{lj} \) is compatible with \( s \). For the other indices \((l, j) \in \tilde{C}_s(x) \) it holds by assumption and Lemma 12 that

\[
\tilde{A}_s(x + \varepsilon v^*)_{lj}(s_{lj} - 0.5) = \varepsilon(v^* \cdot \tilde{v}_{s,l,j})(s_{lj} - 0.5) = \varepsilon \tilde{A}_s(x + v^*)_{lj}(s_{lj} - 0.5) \geq 0.
\]

Using Lemma 28 it follows that \( \tilde{H}_s(x + \varepsilon v^*) \) is compatible with \( s \). To prove the other direction let \( v \in \mathbb{R}^{n_0} \) such that for some \((l, j) \in \tilde{C}_s(x + v) \) the condition \( A_s(v)_{lj}(s_{lj} - 0.5) \geq 0 \) does not hold. Similar to above it follows that \( H(x + \varepsilon v) \) cannot be compatible with \( s \) for any \( \varepsilon > 0 \).

Corollary 29. For an input \( x \in \mathbb{R}^{n_0} \) and a compatible activation pattern \( s \in \mathcal{S}^C(x) \) it holds that all directions \( v \in \mathbb{R}^{n_0} \) such that \( s \in \mathcal{S}^C(x + v) \) satisfy \( v \in \tilde{D}_s(x) \).

Proof of Corollary 29. This follows immediately from Lemmas 28 and 17.

The above corollary gives a sufficient criterion for directions being feasible directions in terms of compatibility.
Proof of Theorem 14. Let
\[
\mathcal{W}_s := \left\{ v \in \mathbb{R}^n \mid \forall (l, j') \in I \setminus \{(l, j)\} \; \tilde{H}_s(x + v)v_{j'} = \tilde{H}_s(x)v_{j'} \wedge (r - 0.5)\tilde{A}_s(x + v)_{lj} > 0 \right\}
\]
and define \(\mathcal{W}_{s'}\) similarly. Corollary 4 implies \(\mathcal{W}_s = \mathcal{W}_{\bar{s}}\). By equation (17) and Lemma 14 it is possible to find \(\varepsilon_s, \varepsilon_{s'} > 0\) such that for \(y_s \in B_{\varepsilon_s}(x)\), \(y_{s'} \in B_{\varepsilon_{s'}}(x)\) the hyperplane patterns \(H_s(y_s)_{lj}, \tilde{H}_{s'}(y_{s'})_{lj}\) are constant at indices \((l, j) \in C^*(x) = C_s(x) = C_{s'}(x)\). Since \(B_{\min(\varepsilon_s, \varepsilon_{s'})}(x)\) contains an open ball around \(x\), it follows that
\[
\left\{ v \in \mathbb{R}^n \mid \forall (l, j') \in \tilde{C}_s(x) \setminus \{(l, j)\} \; \tilde{H}_s(x + v)v_{j'} = \tilde{H}_s(x)v_{j'} \wedge (r - 0.5)\tilde{A}_s(x + v)_{lj} > 0 \right\} = \left\{ v \in \mathbb{R}^n \mid \forall (l, j') \in \tilde{C}_{s'}(x) \setminus \{(l, j)\} \; \tilde{H}_{s'}(x + v)v_{j'} = \tilde{H}_{s'}(x)v_{j'} \wedge (r - 0.5)\tilde{A}_{s'}(x + v)_{lj} > 0 \right\}.
\]
Now note that by equation (29), \(\bar{s}_{s,x,l,j}\) is an element of the first of the above sets and by equivalence also of the second. In particular, for \(\varepsilon^* > 0\) small enough \(x + \varepsilon^*\bar{s}_{s,x,l,j} \in \mathcal{W}_s = \mathcal{W}_{\bar{s}}\) such that \(s, s' \in S^C(x + \varepsilon^*\bar{s}_{s,x,l,j})\). Now Lemma 12 and Theorem 2 imply
\[
\varepsilon^*\langle \bar{s}_{s,x,l,j}, v_{s',l'}, \rangle = \tilde{A}_s(x + \varepsilon^*\bar{s}_{s,x,l,j})v_{j'} = A(x + \varepsilon^*\bar{s}_{s,x,l,j})v_{j'} = \tilde{A}_s(x + \varepsilon^*\bar{s}_{s,x,l,j})v_{j'}
\]
for all indices \((l', j') \in \tilde{C}^*(x)\). By equation (29) this means \(\bar{s}_{s,x,l,j} = \bar{s}_{s',x,l,j}\). \(\square\)

A.4 Regular vertices and local minima

Proof of Lemma 22. If a strict local minimum \(x \in \mathbb{R}^n\) is not a vertex then there exists a non-zero \(v \in \text{Ker}^H(x)\). But by equation (19) cannot be a strict local minimum. For the second claim let \(x^*\) be a non-strict local minimum of \(f\), i.e. there exists \(\varepsilon^* > 0\) such that \(f(x^*) = \inf \{ f(x) \mid x \in B_\varepsilon(x^*) \}\). If \(df^H(x^*) > 0\) then choose a non-zero \(v \in \text{Ker}^H(x^*)\). The map \(t \mapsto f(x^* + tv)\) is an affine function for \(t \in \{ t \in \mathbb{R} \mid H(x^* + tc) = H(x^*) \}\). By equation (19) there exists \(\varepsilon > 0\) such that \((-\varepsilon, \varepsilon) \subset \{ t \in \mathbb{R} \mid H(x^* + tv) = H(x^*) \}\). This implies that, the function \(\mathbb{R} \to \mathbb{R}, t \mapsto f(x^* + tv)\) is differentiable at \(t = 0\) with derivative 0. Then for \(t^* = \min \{ t > 0 \mid H(x^* + tv) = H(x^*) \}\) it holds that \(f(x^*) = f(x^* + tv)\) and \(df^H(x^* + tv) < df^H(x^*)\). Repetition of this procedure leads to a point \(x^{**}\) with \(df^H(x^{**}) = 0\), i.e. a vertex with \(f(x^{**}) = f(x^*)\) after a finite number of steps. \(\square\)

Lemma 30. For \(x \in \mathbb{R}^n\) and \(v \in \mathbb{R}^n \setminus \{0\}\) the limit \(\lim_{t \searrow 0} \frac{1}{t} (f(x + tv) - f(x))\) exists. Furthermore assume that \(\varepsilon^* > 0\) such that for all \((l, j') \in I \setminus C(x)\), \(H(y)_{lj}\) is constant for \(y \in B_{\varepsilon^*}(x)\) and let \(t^* > 0\) small enough such that \(x + t^*v \in B_{\varepsilon^*}\). For \(s \in S^C(x + t^*v)\) it holds that
\[
\lim_{t \searrow 0} \frac{1}{t} (f(x + tv) - f(x)) = \lim_{t \searrow 0} \frac{1}{t} \left( \tilde{f}_s(x + tv) - \tilde{f}_s(x) \right) = \langle v, \nabla \rangle.
\]

Proof. Note that \(H(x)_{lj} = H(x + t^*v)_{lj}\) for \((l, j) \in I \setminus C(x)\) and \(H(x)_{lj} = 0\) for \((l, j) \in C(x)\). Hence \(s\) is compatible with \(H(x)\) and by assumption \(s\) is compatible with \(H(x + t^*v)\). By Theorem 2 it follows that \(s\) is compatible with both \(\tilde{H}_s(x)\) and \(\tilde{H}_s(x + t^*v)\). By equation (12)
and the fact that all coordinates of $y \mapsto \tilde{A}_s(y)$ are affine functions it follows that $s$ is compatible with all $\tilde{H}_s(x+tv)$ for $t \in [0,t^*]$ and hence again by Theorem 2 with all $H(x+tv)$ for $t \in [0,t^*]$. Applying the same theorem again yields $A(x+tv) = \tilde{A}_s(x+tv)$ and hence $f(x+tv) = \tilde{f}_s(x+tv)$ for all $t \in [0,t^*]$. The result now follows from the fact that $\tilde{f}_s$ is affine with gradient exactly $\nabla_s$ as in equation (35).

**Corollary 31.** Let $x \in \mathbb{R}^{n_0}$ be a regular vertex and $\varepsilon^* > 0$ such that for all $(l'j') \in I \setminus C(x)$, $H(y)$ is constant for $y \in B_{\varepsilon^*}(x)$. For every $v \in B_{\varepsilon^*}(0)$ there exist non-negative coefficients $\alpha_1, \ldots, \alpha_{n_0} \in [0,\infty)$ and $n_0$ region separating axes $a_1, \ldots, a_{n_0} \in \mathcal{A}(x)$ such that

$$ f(x + v) = f(x) + \sum_{i=1}^{n_0} \alpha_i \lim_{t \searrow 0} \frac{1}{t} (f(x + ta_i) - f(x)). $$

**Proof.** Let $v \in B_{\varepsilon^*}(0)$ and $s \in \mathcal{S}^{C}(x)$ such that $s$ is compatible with $H(x + v)$. Then $v \in D_s(x)$ by Corollary 29. Lemma 18 and equation (32) imply that there exist coefficients $\alpha_{ij} \in [0,\infty), (l, j) \in C(x)$ such that

$$ v = \sum_{(l, j) \in C_s(x)} \alpha_{ij} a_{x, s, l, j} = \sum_{(l, j) \in C_s(x)} \alpha_{ij} a_{x, s, l, j} $$

(54)

This implies that

$$ f(x + v) \overset{(1)}{=} \tilde{f}_s(x + v) \overset{(2)}{=} \tilde{f}_s(x + \sum_{(l, j) \in C_s(x)} \alpha_{ij} a_{x, s, l, j}) \overset{(3)}{=} \tilde{f}_s(x) + \left( \sum_{(l, j) \in C_s(x)} \alpha_{ij} a_{x, s, l, j}, \nabla_s \right) $$

$$ = f(x) + \sum_{(l, j) \in C_s(x)} \alpha_{ij} \left( a_{x, s, l, j}, \nabla_s \right) \overset{(4)}{=} f(x) + \sum_{(l, j) \in C_s(x)} \alpha_{ij} \left( \tilde{f}_s(x + a_{x, s, l, j}) - \tilde{f}_s(x) \right) $$

$$ \overset{(5)}{=} f(x) + \sum_{(l, j) \in C_s(x)} \alpha_{ij} \lim_{t \searrow 0} \frac{1}{t} \left( \tilde{f}_s(x + ta_{x, s, l, j}) - \tilde{f}_s(x) \right) $$

$$ \overset{(6)}{=} f(x) + \sum_{(l, j) \in C_s(x)} \alpha_{ij} \lim_{t \searrow 0} \frac{1}{t} \left( f(x + ta_{x, s, l, j}) - f(x) \right). $$

The justification of the steps is as follows: (1) follows from Corollary 3, (2) from equation (54), (3), (4) and (5) from the fact that the functions $f_{s'}$, $s' \in \mathcal{S}$ are affine and (6) is an implication of Lemma 30. Furthermore $|C_s(x)| = n_0$ by Lemma 15.

**Proof of Proposition 21.** The claim follows from Lemmas 30 and Corollary 31.

---

**B Bounds on the number of affine regions**

In the sequel, let $N(n_0, \ldots, n_L)$ denote the maximal number of affine regions that a ReLU feed-forward neural network of depth $L$ and widths $n_0, \ldots, n_L$ can have. We will now present some bounds on $N(n_0, \ldots, n_L)$.

The work [11] establishes the following bound on the number of affine regions:
Lemma 32 (Montúfar). Let $L \in \mathbb{N}$. A neural network with $L + 2$ layers ($L$ hidden layers) of widths $n_0, \ldots, n_{L+1}$ is a piece-wise affine function with at most

$$N(n_0, \ldots, n_L) \leq \prod_{i=1}^{L} \sum_{j=0}^{\min(n_0, \ldots, n_i)} \binom{n_i}{j}$$

affine regions.

It uses a result from 1975 by Zaslavsky, which bounds the number of regions that are carved out of $\mathbb{R}^{n_0}$ by $n_1$ hyperplanes, see [12]:

Lemma 33 (Zaslavsky). In $\mathbb{R}^d$, $N \in \mathbb{N}$ hyperplanes cut the space $\mathbb{R}^d$ into at most $\sum_{j=0}^{d} \binom{N}{j}$ regions.

This bound was improved in [2] to

$$N(n_0, \ldots, n_L) \leq \sum_{j_1, \ldots, j_L \in J} \prod_{i=1}^{L} \binom{n_i}{j_i},$$

where $J = \{(j_1, \ldots, j_L) \in \mathbb{N}^L | \forall i \in \{1, \ldots, L\} : j_i \leq \min(n_0, n_1 - j_1, \ldots, n_{L-1} - j_{L-1}, n_L)\}$.

Later, an abstract framework for the construction of such upper bounds was presented in [3].

It yields bounds formulated as the L1 norm of a product of matrices

$$N(n_0, \ldots, n_L) \leq \|B_n^\gamma M_{n_{L-1}, n_L} \cdots B_{n_1} M_{n_0, n_1} c_{n_0+1}^{(n_0+1)}\|_1,$$

where $M_{n_0, n_1}, \ldots, M_{n_{L-1}, n_L}$ are defined by $M_{a,b} = (\delta_{i, \min(j, a+1)})_{i \in \{1, \ldots, b+1\}, j \in \{1, \ldots, a+1\}} \in \mathbb{N}^{b+1 \times a+1}$ for $a, b \in \mathbb{N}_+$, $c_{n_0+1}^{(n_0+1)} = (\delta_{n_0+1, i})_{i \in \{1, \ldots, n_0+1\}} \in \mathbb{N}^{n_0+1}$ and $B_n^\gamma \in \mathbb{N}^{a+1 \times a+1}$ are upper triangular matrices depending on some family of vectors $(\gamma_{ij})_{i,j \in \mathbb{N}}$. The bounds (55) and (56) can be derived from (57), see [3].

C Julia Implementation

As a proof-of-concept, we have implemented our DRLP-Algorithm in the Julia programming language[3]. We used this implementation for the numerical experiments in Section 4.

References

[1] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, “On the number of linear regions of deep neural networks,” in Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS’14, (Cambridge, MA, USA), pp. 2924–2932, MIT Press, 2014.

[3] https://github.com/hinzstatmathethzch/DRLP
[2] T. Serra, C. Tjandraatmadja, and S. Ramalingam, “Bounding and counting linear regions of deep neural networks,” *CoRR*, vol. abs/1711.02114, 2017.

[3] P. Hinz and S. van de Geer, “A framework for the construction of upper bounds on the number of affine linear regions of relu feed-forward neural networks,” *arXiv preprint arXiv:1806.01918*, 2018.

[4] R. Koenker and G. Bassett Jr, “Regression quantiles,” *Econometrica: journal of the Econometric Society*, pp. 33–50, 1978.

[5] P. Bloomfield and W. L. Steiger, *Least Absolute Deviations: Theory, Applications and Algorithms*, pp. 37–76. Boston, MA: Birkhäuser Boston, 1983.

[6] I. Barrodale and F. D. Roberts, “An improved algorithm for discrete l1 linear approximation,” *SIAM Journal on Numerical Analysis*, vol. 10, no. 5, pp. 839–848, 1973.

[7] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[8] J. L. Powell, “Least absolute deviations estimation for the censored regression model,” *Journal of Econometrics*, vol. 25, no. 3, pp. 303–325, 1984.

[9] M. Buchinsky, “Changes in the u.s. wage structure 1963-1987: Application of quantile regression,” *Econometrica*, vol. 62, no. 2, pp. 405–458, 1994.

[10] B. Fitzenberger, “Computational aspects of censored quantile regression,” *Lecture Notes-Monograph Series*, vol. 31, pp. 171–186, 1997.

[11] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[12] T. Zaslavsky, “Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes,” vol. 154, 01 1975.