UCLouvain

epl

École polytechnique de Louvain

On the number of linear regions in a Rectified Network

Author: **Alex Montens**
Supervisor: **Julien Hendrickx**
Readers: **John Lee, Helene Verhaeghe**
Academic year 2025–2026
Master [120] in Computer Science

Abstract

Neural networks have become ubiquitous in our modern world. This widespread success can largely be attributed to advances in deep learning, a field that has demonstrated a remarkable capacity for universal learning across a wide range of tasks and domains. The way deep neural networks do this is not well understood, but for a certain class of neural networks, known as rectified networks, which possess piecewise linear activation functions, the expressivity can be linked to the number of linear regions they partition the input space into. Several works have derived upper bounds on the number of regions a rectified network can achieve and a formula for the expected value was conjectured in 2019. In this work we prove this formula, showing it holds under mild assumptions. This formula does not depend on the number of layers in the network, only the number of neurons, which is in conflict with the long-standing assumption that deep networks are exponentially more expressive than shallower ones. We demonstrate both theoretically and experimentally that this is not case.

Contents

Introduction

Machine learning has existed and been present in all kinds of professional fields for decades now, but for the past few years, a single type of model has emerged as the dominant paradigm due to its versatility, broad applicability, and strong performance across a wide range of tasks: neural networks. Neural networks are a powerful tool in machine learning that can very efficiently learn patterns in the data without requiring manual feature engineering. This paradigm shift from manual feature selection and extraction to simply collecting massive amounts of data and leaving the hard work to the computer has allowed these models to now be used everywhere in the world, for many different purposes, but one problem we cannot solve is how or why they work so well. For neural networks with piecewise linear activation functions, also called rectified networks, they partition the input space into different convex polytopes. Each of these polytopes represents a different linear region where the entire network is collapsed into a single equivalent affine function.

The question of how many linear regions a neural network contains is an understudied part of deep learning. Some advances have been made in bounding the number of regions the network can have ([MPCB14], [Mon17], [STR17]), but little research has currently asked how many they have on average ([HR19b]). Answering this question would help us understand what makes them so powerful, how different architectures of networks change the behaviour, why some work for some tasks and others do not, and much more. It could also allow us to plan in advance what size a network must have in order to solve a given task before training even starts.

This question is hard to test as computing the number of linear regions in a network is an expensive task, practically infeasible for any decently sized network (with thousands of neurons or more). As will be demonstrated in this paper, this grows with respect to the input size, which nowadays can reach thousands, making it intractable.

[Ayt22] proved that rectified networks could be transformed into decision trees. Once pruned, this leaves us with a reduced tree where the number of leaves is the number of linear regions of the original network. This tree decomposition not only allows us to count these linear regions for small networks, but also offers a

representation perfectly suited for the task of deriving a closed-form solution to this problem for the first time.

In this work, we will start by giving a background on relevant subjects in Chapter 2, then we will establish a proper notation and mathematical description of the problem in Chapter 3. We will follow in Chapter 4 with a review of the current standing on the problem before deriving our solution in Chapter 5. Finally, in Chapter 6, we demonstrate empirically the predictions from this formula.

Goal: Find a closed-form solution for the expected value of the number of linear regions in rectified networks.

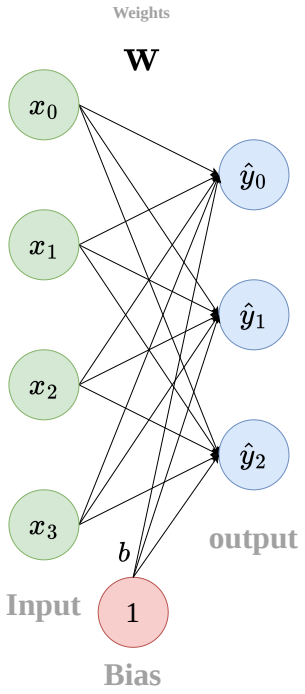
Contributions:

1. Improved algorithm for transforming a rectified network into a decision tree.
2. Closed-form formula for the expected number of linear regions.
3. Empirical and theoretical evidence that for rectified networks with a fixed number of neurons, the expressivity is independent of depth.

Background Material

1 Neural Networks

Neural networks are a type of learning model based on the composition of linear and non-linear layers. They are composed of nodes which are either the input to the model, the intermediary nodes which take as input the nodes from the previous layer, or the output nodes which also take the previous layer of nodes as input, but generally have a different activation function than the other sets of nodes. The activation function is the non-linear part of the neural network.



$$\hat{y} = \sigma(x\mathbf{W}^T + b)$$

Figure 2.1: Single layer Neural Network.

$$\mathbf{NN}(x) = x\mathbf{W}^T + b \quad \text{for a single layer without activation} \tag{2.1}$$

where $\mathbf{W}^T$ is the weight matrix and b is the bias vector.

For binary classification, we could define the final activation function as the sigmoid (see Figure 2.2), this choice simplifies the gradient computation which is often used in deep learning:

$$\begin{aligned} &\sigma(x) = \frac{1}{1+e^{-x}} \\ &\Rightarrow \mathbf{NN}(x) = \sigma(x\mathbf{W}^T + b) \quad \text{for a single layer with activation} \end{aligned} \tag{2.2}$$

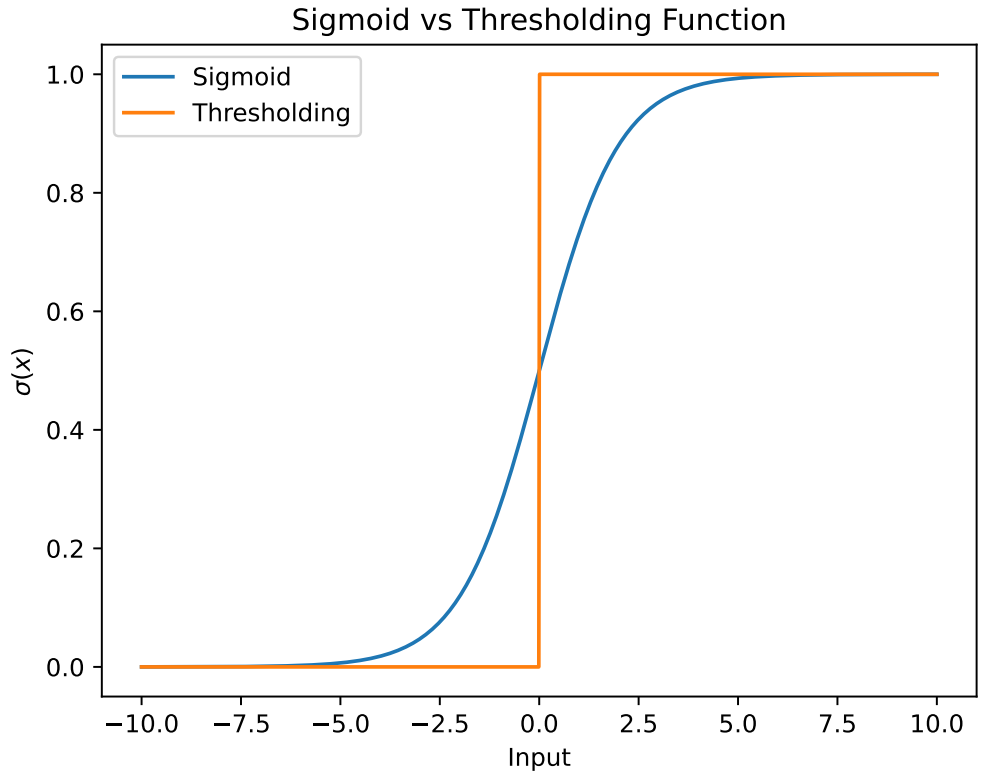


Figure 2.2: Comparison between the thresholding function and the smoothed sigmoid function.

These single layer neural networks (see Figure 2.1), called perceptrons, are very elementary models which are not used much in practice because of the lack of expressiveness. A linear layer can only separate points which are divided by a plane. Only the left dataset in Figure 2.4 can be successfully classified by a perceptron. To solve this issue, we introduce depth to the network to create deep neural networks

(see Figure 2.3). This is done by composition: the output of a layer is fed as input to the next. These intermediary layers are commonly called hidden layers.

$$\mathbf{NN}(x) = \sigma_o(\sigma_h(x\mathbf{W}_0^T + b_0)\mathbf{W}_1^T + b_1) \tag{2.3}$$

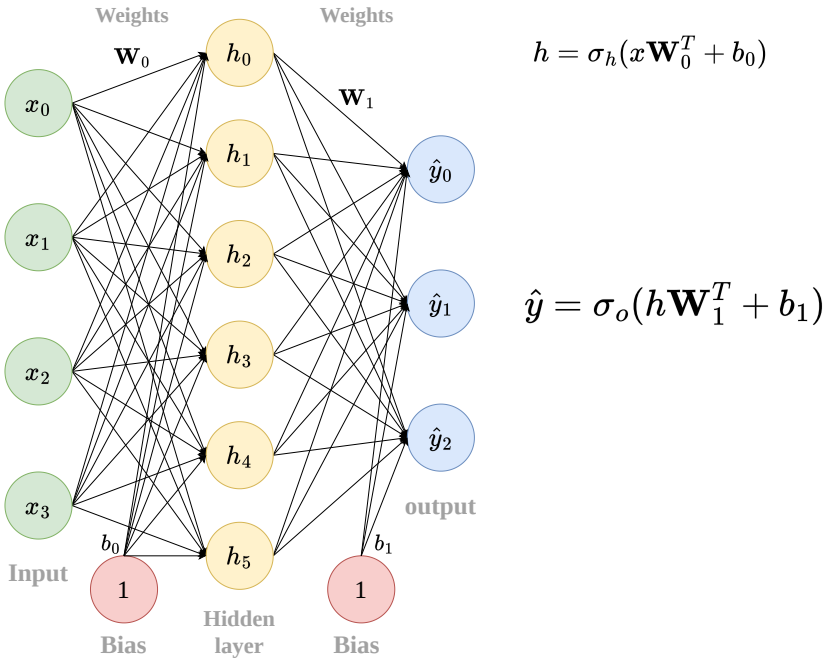


Figure 2.3: Deep Neural Network with a hidden layer.

Note that this time, the activation function σ is not unique, there is one for hidden layers and one for the final output. Different activation functions will lead to the networks having different properties. What we are interested in for this work is a family called rectified functions.

Neural networks often learn from data using first order methods like Adam [KB15], Adagrad [DHS11] or Stochastic Gradient Descent (SGD) which use the derivative of an objective function f with respect to the network's parameters. This function f can for example be the mean squared error for regression or the cross-entropy in the case of classification. The data is used to define a parametrized function as an objective to minimize, this function is called the loss. The general SGD formula for a parametrized function f_θ is:

$$\theta^{t+1} = \theta^t - \eta \nabla f_{\theta^t}(x_i; y_i) \tag{2.4}$$

where θ are the parameters and η is the step size, known as the learning rate.

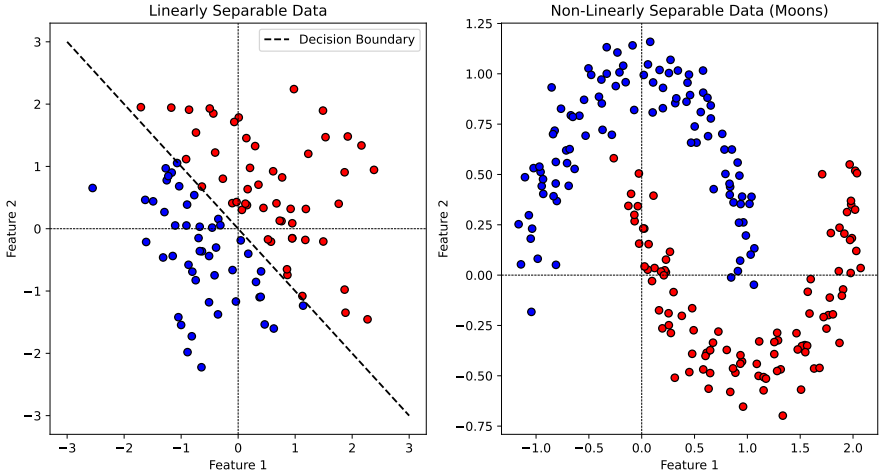


Figure 2.4: Comparison between linearly separable and non-separable points. Single layer Neural Networks cannot classify the non-separable dataset with perfect accuracy.

2 Rectified Networks

At first, deep neural networks were not practical to use, the activation function they used was a thresholding function $x > 0$ which only gave binary outputs. The sigmoid function introduced earlier was a later attempt at creating a continuous candidate. The reason the function should be continuous is to allow for the computation of gradients. Neural networks are trained using gradient descent, which requires computing gradients. A thresholding function has a gradient of 0 everywhere except at its discontinuity, meaning no information can be propagated further. The sigmoid function solved this issue, but created the vanishing gradient problem: as networks got deeper, the gradient at earlier layers got smaller and smaller until no signal was left for training. Rectified activation functions were introduced to solve this problem. The simplest one, which quickly became ubiquitous is ReLU:

$$\mathbf{ReLU}(x) = \begin{cases} x & if \quad x \geq 0 \\ 0 & otherwise \end{cases} \tag{2.5}$$

Until recent advances, this function was used in every network as it is incredibly simple to compute and behaves very nicely for gradients. A more general formulation of ReLU is LeakyReLU (see comparison in Figure 2.5) defined as:

$$\mathbf{LeakyReLU}(x) = \begin{cases} x & if \quad x \geq 0 \\ \alpha x & otherwise \end{cases} \tag{2.6}$$

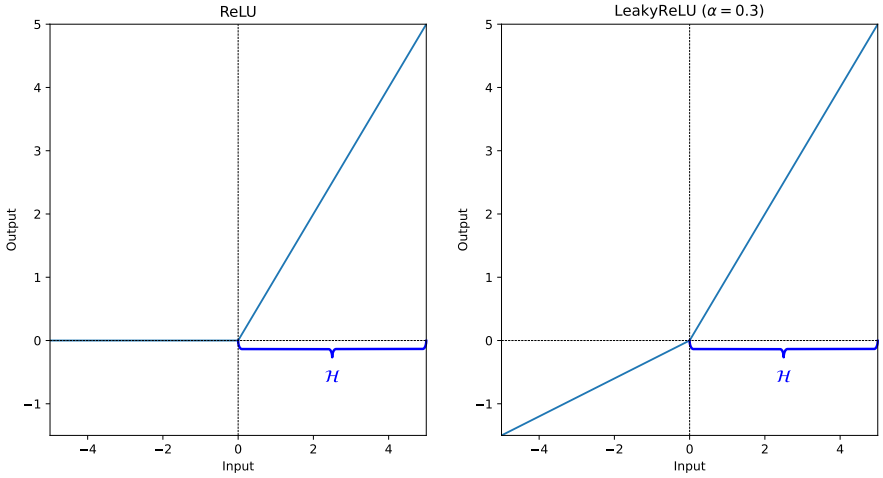


Figure 2.5: Comparison between the ReLU activation function and LeakyReLU. $\mathcal{H}$ is the half-space in Equation (2.7).

where α is typically set to 0.3 in many deep learning libraries such as TensorFlow [AAB+15].

3 Linear regions

ReLU and LeakyReLU, being piecewise linear functions, are defined through a condition. The condition being applied to every node, at every depth of the network, creates an activation pattern for each point in the input space. The different activation patterns in the network create what are called linear regions, which are a global property of the network. The expressivity of a neural network, its capacity to separate points, is closely related to the number of linear regions it creates. The more it can partition the space, the more points it can differentiate. A closely related concept is the activation region. This is sometimes interchangeably used with linear regions in the literature, but it represents a slightly different concept. Linear regions are a property of the network as a whole, the pattern of activation of every node in the network, whereas an activation region is for a single node. Examples of linear regions from rectified networks can be seen in Figures 2.6 and 2.7.

A node in a rectified network computes $\mathbf{LeakyReLU}(x\mathbf{v}^T + b)$ which defines a condition $x\mathbf{v}^T + b \geq 0$, where $\mathbf{v}$ is the row of the weight matrix associated with this node and b is the bias associated with the node. Its activation region is a half-space of the input region. Each node defines a half-space:

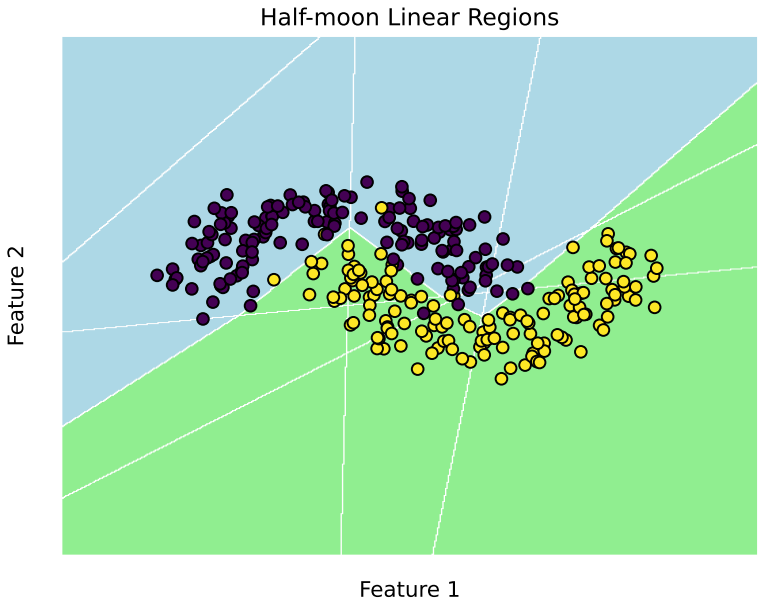


Figure 2.6: Half-moon dataset and the linear regions of a Neural Network trained on it. The colours of the points correspond to their labels and the colours of the regions correspond to the network's labelling. This plot is obtained from the example in Section 1.1.

$$\begin{aligned} K &\equiv x\mathbf{v}^T + b \geq 0 \\ \mathcal{H} &= \{x \in \mathbb{R}^{n_0} : K\} \end{aligned} \tag{2.7}$$

4 Decision Trees

Decision trees are a type of model that execute their decisions through a sequence of conditions. These conditions are usually linear and often operate on a single variable at a time. This allows for these models to partition the space into distinct regions that allow for separate treatment of the data. Each of these regions assigns a value to that space, whether it is the label for classification or the value for regression. This partitioning of space can be seen in Figure 2.8.

5 Linear Programming

Linear Programming is an optimization technique where a linear objective function is optimized subject to linear constraints [Chv83]. Many optimization problems

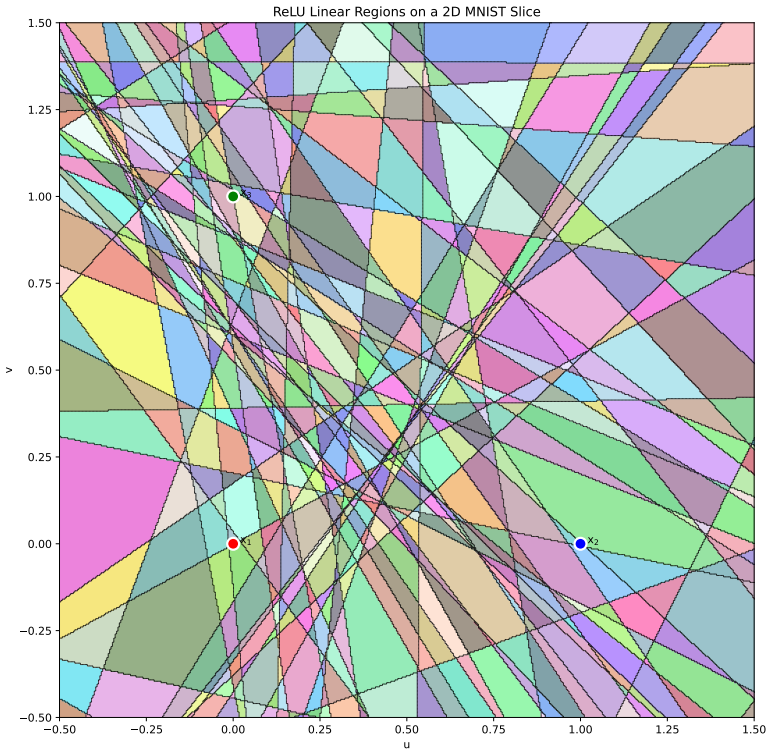


Figure 2.7: 2D slice of linear regions of a Neural Network trained on MNIST.

are expressed as linear programs across many fields such as in economics, logistics, manufacturing, telecommunication and many others [Dan02]. Linear programs can be expressed in standard form as:

$$\begin{aligned} &\textbf{Find vector} && \mathbf{x} \\ &\textbf{that maximizes} && \mathbf{c}^T\mathbf{x} \\ &\textbf{subject to} && A\mathbf{x} \leq \mathbf{b} \\ &\textbf{and} && \mathbf{x} \geq 0 \end{aligned} \tag{2.8}$$

where $\mathbf{c}$ and $\mathbf{b}$ are vectors and A is the constraint matrix.

A sub-problem of linear programming is the feasibility problem, which is determining if such a vector $\mathbf{x}$ exists, without finding any explicit value for it or optimizing for any specific objective.

A harder extension of Linear Programming is Mixed-Integer Linear Programming where values are no longer in $\mathbb{R}$, but are instead in $\mathbb{Z}$ [Wol98]. This problem is NP-Hard, meaning that there is no efficient way to solve these problems in general.

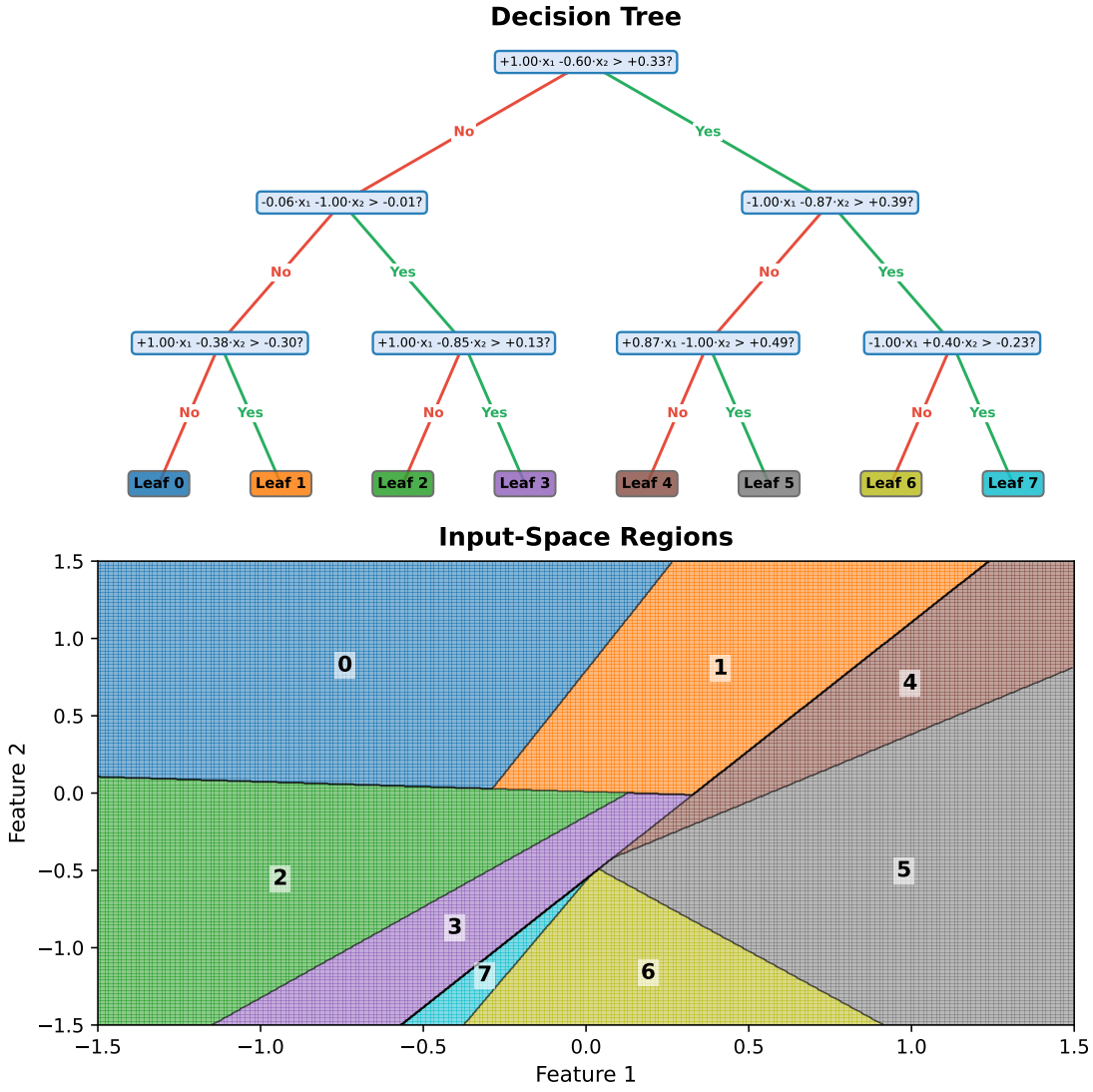


Figure 2.8: Decision tree and the induced space partition.

Problem Statement

1 Definitions and notation

To avoid confusion, depth will solely refer to the depth of the tree and layer to that of the network. Some symbols will serve as generic indices or placeholder variables, but some will have consistent meaning throughout this work:

- n_l: Number of neurons at layer l
- n_0: Input dimension (taking layer 0 as being the input)
- L: Number of layers in the network
- n: Number of neurons in a network $= \sum_{i=1}^{L-1} n_l$
- N_L: Number of linear regions

2 Network statement

Let $\mathbf{NN}_\theta : \mathbb{R}^{n_0} \to \mathbb{R}^m$ denote a feed-forward neural network with input dimension n_0 and L layers, parametrized by weights $\theta = \{\mathbf{W}_l, b_l\}_{l=0}^{L-1}$. The function is constructed as a composition of affine transformations and piecewise linear activation functions $\sigma(\cdot)$.

$$\begin{aligned} \mathbf{NN}_\theta(x) &= \sigma(\dots(\sigma(\sigma(x\mathbf{W}_0^T + b_0)\mathbf{W}_1^T + b_1)\dots)\mathbf{W}_{L-2}^T + b_{L-2})\mathbf{W}_{L-1}^T + b_{L-1} \\ x_l &= \sigma(\dots(\sigma(\sigma(x\mathbf{W}_0^T + b_0)\mathbf{W}_1^T + b_1)\dots)\mathbf{W}_{l-1}^T + b_{l-1}) \end{aligned} \tag{3.9}$$

Specifically, for each layer l, the transformation is defined as:

$$x_l = \sigma(x_{l-1}\mathbf{W}_{l-1}^T + b_{l-1}) \tag{3.10}$$

where, in the case of rectified networks, $\sigma : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ is a rectified activation function (e.g., ReLU or LeakyReLU), defined pointwise by:

$$\sigma(x) = \begin{cases} x & if \quad x \geq 0 \\ \alpha x & otherwise \end{cases} \tag{3.11}$$

3 Linear regions statement

Due to the piecewise linearity of σ, the function $\mathbf{NN}_\theta$ induces a partition of the input space $\mathbb{R}^{n_0}$ into disjoint connected convex polytopes, denoted by the set $\mathcal{P}_{\mathbf{NN}_\theta}$. A region $P \in \mathcal{P}_{\mathbf{NN}_\theta}$ is a linear region if and only if for all $x \in P$, the entire network output can be represented as a single affine function:

$$\mathbf{NN}_\theta(\mathbf{x}) = x\mathbf{A}^T + c \tag{3.12}$$

where $\mathbf{A}$ and c are constant and independent of x within that region.

Formally, let $\mathcal{A}$ be the set of all possible binary activation patterns ϵ. Each pattern corresponds to a system of linear inequalities:

$$\begin{aligned} C^+(\epsilon) &\equiv x\mathbf{W}_{l,k}^T + b_{l,k} \geq 0 \quad \forall l, k \in S_{\text{on}}(\epsilon) \\ C^-(\epsilon) &\equiv x\mathbf{W}_{l,j}^T + b_{l,j} < 0 \quad \forall l, j \in S_{\text{off}}(\epsilon) \end{aligned} \tag{3.13}$$

where S_{on} and S_{off} denote the sets of neurons activated and non-activated for pattern ϵ, respectively. The corresponding linear region P_ϵ is non-empty if and only if the following system of inequalities is feasible, which corresponds to a Linear Program, and more specifically to a satisfiability problem:

$$P_\epsilon \neq \emptyset \iff \exists x \in \mathbb{R}^{n_0} : C^+(\epsilon) \wedge C^-(\epsilon) \tag{3.14}$$

The cardinality of the partition is defined as:

$$N_L(\mathbf{NN}_\theta) = |\mathcal{P}_{\mathbf{NN}_\theta}| = \sum_{\epsilon \in \mathcal{A}} \mathbb{I}(P_\epsilon \neq \emptyset) \tag{3.15}$$

Goal: The objective of this paper is to estimate the expected number of linear regions and derive a closed-form expression for the expectation:

$$\mathbb{E}[N_L] \tag{3.16}$$

Literature Review and Context

Counting, bounding, or otherwise deriving an analytic expression for the number of linear regions is a research area pioneered by [MPCB14]. In this first work, they applied a simple heuristic: each neuron contributes a binary decision, meaning a network with n neurons would have at most 2^n patterns. This upper bound was the first derived, but far from tight. In [Mon17], this bound was further improved to:

$$N_L \leq \prod_{l=1}^{L} \sum_{j=0}^{m_{l-1}} \binom{n_l}{j} \tag{4.17}$$

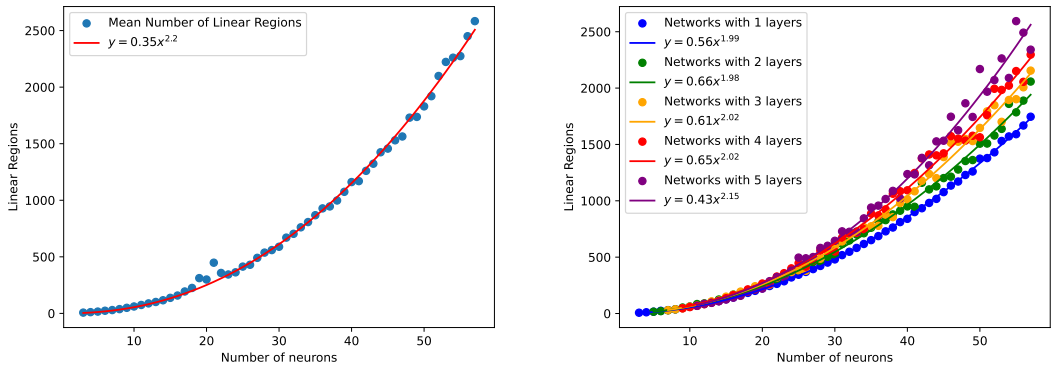
for a neural network with n_0 inputs and L layers of $n_1, \ldots, n_L$ neurons with $m_{l-1} = min\{n_0, ..., n_{l-1}\}$.

The field suffered from its inability to compare theoretical results with empirical data. Bounds were derived with no means to test them on real neural networks. [STR17] offered the first method to count these linear regions. This method worked to give the first observations and compare them to theoretical results, but was limited: the input regions had to be bound and it used Mixed-Integer Linear Programming which is computationally expensive. In the same work, they derived a stronger bound of:

$$N_L \leq \sum_{(j_1,\ldots,j_L)\in J} \prod_{l=1}^{L} \binom{n_l}{j_l} \tag{4.18}$$

where $J = \{(j_1, ..., j_L) \in \mathbb{Z}^L : 0 \leq j_l \leq min\{n_0, n_1 - j_1, \ldots, n_{l-1} - j_{l-1}, n_l\} \quad \forall l = 1, \ldots, L\}$.

The ability to measure the number of linear regions now made it possible to guide theoretical work with the help of observations. By training a neural network on the half-moon dataset and on the iris dataset, we start to see a trend in Figures 4.9 and 4.10: the number of linear regions scales polynomially with respect to the number of neurons. Depth, which was thought to increase the expressivity of neural networks exponentially, only has a tiny effect, increasing the exponent from 1.99 to 2.15 over 5 layers on the half-moon dataset.



(a) All network configurations averaged. (b) All network configurations averaged by number of layers.

Figure 4.9: Number of linear regions with respect to the number of neurons. Every network configuration with a given number of neurons was trained 5 times on the half-moon dataset. The measured data and the best ax^b fits are plotted.

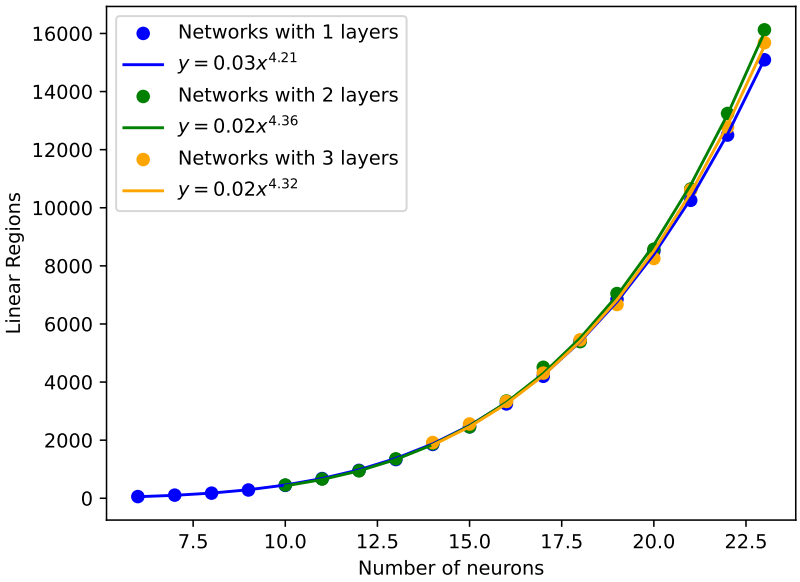


Figure 4.10: Number of linear regions with respect to the number of neurons. Every network configuration with a given number of neurons was trained 5 times on the iris dataset. The measured data and the best ax^b fits are plotted, averaged by number of layers. Computational costs limited the plot to a more restrained number of neurons than the half-moon dataset one.

All prior works derived upper bounds on the number. The estimation of the typical behaviour, the average number of linear regions, has not been as explored. The same year, in 2019, two papers from the same authors came out, [HR19a] and [HR19b] which cover the subject. They successfully derive an upper bound on the expected value and from this, they suggested a general form of $\frac{n^{n_0}}{n_0!}$ for the expected value, a polynomial form matching observations and precisely what we prove later.

Solution

1 Decision Tree Decomposition Method

1.1 Running example

To illustrate this section, we will keep a running example of the method using a neural network trained on the half-moon dataset, the same one from Figure 2.6. This neural network has a single hidden layer with 5 neurons. The weights are:

$$\mathbf{W}_0 = \begin{bmatrix} 3.6098 & -0.0624 \\ 0.6497 & -0.7927 \\ 4.3684 & -0.5638 \\ 1.8314 & -1.2996 \\ 0.2922 & -1.9378 \end{bmatrix}, b_0 = \begin{bmatrix} -0.1995 \\ -0.5587 \\ -4.5972 \\ 4.1683 \\ -0.2310 \end{bmatrix}$$

$$\mathbf{W}_1 = \begin{bmatrix} 3.7594 & -0.5889 & -4.2003 & -2.0180 & -0.7174 \\ -4.0120 & 0.7478 & 3.9010 & 2.1633 & 1.1049 \end{bmatrix}, b_1 = \begin{bmatrix} 0.3990 \\ -0.3027 \end{bmatrix}$$

1.2 Effective weight decomposition

As was shown in [Ayt22], if we ignore the bias parameters of the network, we can rewrite the recurrent form of the network into a formula with no non-linearity where x_l is the transformed input at layer l, $\mathbf{A}$ is a matrix that depends on the activation pattern created by the input x_0 and $\odot$ is element-wise multiplication:

$$\sigma(x_l)\mathbf{W}_l^T = \sigma(x_{l-1}\mathbf{W}_{l-1}^T)\mathbf{W}_l^T = x_{l-1}\mathbf{W}_{l-1}^T(\mathbf{W}_l^T \odot \mathbf{A}) \tag{5.19}$$

Recursively applying this formula and multiplying all the matrices together, we see that an effective matrix can be obtained for each input. This effective matrix encapsulates the entire neural network for a given linear region. We start by defining the initial effective matrix as $\mathbf{W}_e^0 = \mathbf{W}_0$, then iterate the following for each layer:

$$
\begin{aligned}
h &= x(\mathbf{W}_e^l)^T \\
\mathbf{A}_{\mu\nu} &= \begin{cases} 1 & \text{if } h_\mu \geq 0 \\ \alpha & otherwise \end{cases} \quad \forall \nu \\
\mathbf{W}_e^{l+1} &= (\mathbf{W}_{l+1} \odot \mathbf{A}^T)\mathbf{W}_e^l
\end{aligned}
\tag{5.20}
$$

The product $x(\mathbf{W}_e^{L-1})^T$ is then equivalent to $\mathbf{NN}(x)$.

The bias can be accounted for by including it in the weights by concatenating $\mathbf{W}_l|b_l$ and adding a dimension to the input equal to a constant value of 1.

$$
\begin{aligned}
\mathbf{W}_l &= \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \cdots & \vdots \\ \vdots & \vdots & \ddots & w_{n-2,m-1} \\ w_{n-1,0} & \cdots & w_{n-1,m-2} & w_{n-1,m-1} \end{bmatrix}, b_l = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n_i-1} \end{bmatrix} \\
\Rightarrow & \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} & b_0 \\ w_{1,0} & w_{1,1} & \cdots & \vdots & b_1 \\ \vdots & \vdots & \ddots & w_{n-2,m-1} & \vdots \\ w_{n-1,0} & \cdots & w_{n-1,m-2} & w_{n-1,m-1} & b_{n-1} \end{bmatrix}
\end{aligned}
\tag{5.21}
$$

And to compensate for the additional dimension in the weight matrices, we concatenate a one-hot row vector with a 1 in last position to every weight matrix but the last.

$$
\hat{\mathbf{W}}_l = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} & b_0 \\ w_{1,0} & w_{1,1} & \cdots & \vdots & b_1 \\ \vdots & \vdots & \ddots & w_{n-2,m-1} & \vdots \\ w_{n-1,0} & \cdots & w_{n-1,m-2} & w_{n-1,m-1} & b_{n-1} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}
\tag{5.22}
$$

For our half-moon example, we get the following pre-processed weights:

$$
\hat{\mathbf{W}}_0 = \begin{bmatrix} 3.6098 & -0.0624 & -0.1995 \\ 0.6497 & -0.7927 & -0.5587 \\ 4.3684 & -0.5638 & -4.5972 \\ 1.8314 & -1.2996 & 4.1683 \\ 0.2922 & -1.9378 & -0.2310 \\ 0 & 0 & 1 \end{bmatrix}
$$

$$
\hat{\mathbf{W}}_1 = \begin{bmatrix} 3.7594 & -0.5889 & -4.2003 & -2.0180 & -0.7174 & 0.3990 \\ -4.0120 & 0.7478 & 3.9010 & 2.1633 & 1.1049 & -0.3027 \end{bmatrix}
$$

1.3 Converting neural networks into decision trees

To create a tree from the effective weights $\mathbf{W}_e$, we create a set of inequalities at each layer l of the network from the x_l and the $\mathbf{W}_e^l$ by doing the following:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \\ 1 \end{bmatrix}^T, \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} & b_0 \\ w_{1,0} & w_{1,1} & \cdots & \vdots & b_1 \\ \vdots & \vdots & \ddots & w_{n-2,m-1} & \vdots \\ w_{n-1,0} & \cdots & w_{n-1,m-2} & w_{n-1,m-1} & b_{n-1} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}^T \tag{5.23}$$

$$\Rightarrow \begin{cases} x_0 w_{0,0} + x_1 w_{0,1} + \cdots + x_{m-1} w_{0,m-1} & > -b_0 \\ x_0 w_{1,0} + x_1 w_{1,1} + \cdots + x_{m-1} w_{1,m-1} & > -b_1 \\ & \vdots \\ x_0 w_{n-1,0} + x_1 w_{n-1,1} + \cdots + x_{m-1} w_{n-1,m-1} & > -b_{n-1} \end{cases}$$

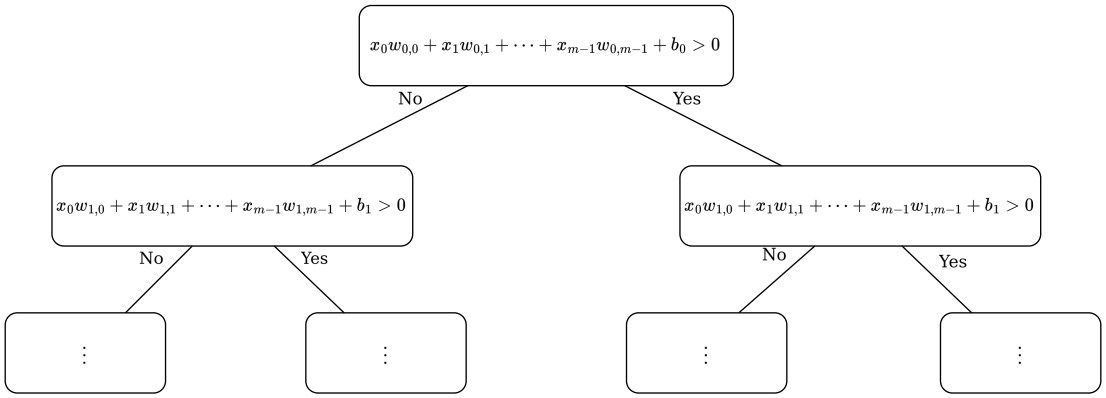


Figure 5.11: Tree built from inequalities in Equation (5.23).

At the very first layer, we pick the first inequality as root and then we create a left and right child from the next inequality and repeat for every one of them. Once a layer is done, we compute the effective matrix for every path. Then, a new subtree is created with its set of inequalities, which is why parts of the tree are self-similar as can be seen in Figure 5.11.

Since a new depth is created for every element in each output vector, including the intermediary ones, the maximum depth is given by:

$$n = \sum_{l=1}^{L-1} n_l \tag{5.24}$$

where n_l is the size of x_l, the number of neurons at layer l, making n equal to the total number of neurons.

The set of inequalities for the first layer of the half-moon example are:

$$\begin{cases} 3.6098x_0 - 0.0624x_1 & \geq 0.1995 \\ 0.6497x_0 - 0.7927x_1 & \geq 0.5587 \\ 4.3684x_0 - 0.5638x_1 & \geq 4.5972 \\ 1.8314x_0 - 1.2996x_1 & \geq -4.1683 \\ 0.2922x_0 - 1.9378x_1 & \geq 0.2310 \end{cases} \tag{5.25}$$

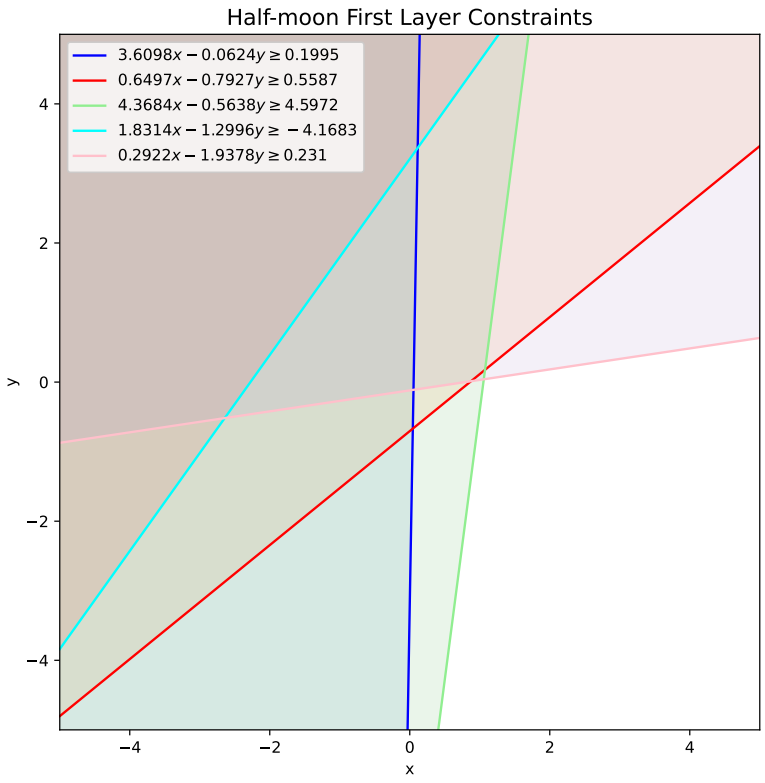


Figure 5.12: Regions from the constraints in Equation (5.25).

The intermediary tree for the half-moon example, intermediary because we have yet to account for the classification layer, can be seen in Figure 5.13.

1.4 Classification layer

By transforming a neural network into a decision tree as we have done here, we have successfully obtained an effective weight matrix for each partitioning of the

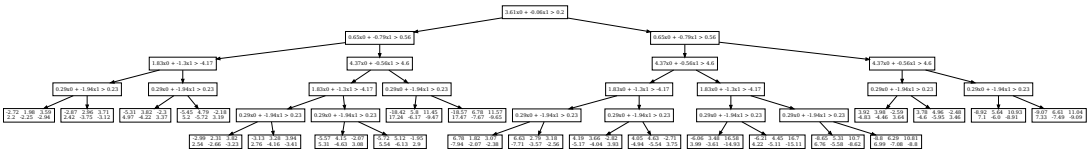


Figure 5.13: Equivalent tree derived from the example neural network trained on the half-moon dataset. Each leaf node represents the effective weights for that branch. (Full page version available for readability in Figure 8.26.)

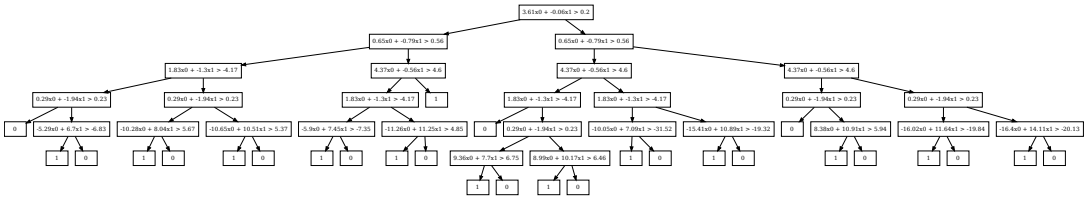


Figure 5.14: Equivalent tree derived from the example neural network trained on the half-moon dataset. Each leaf node represents the label assigned by the network for that branch. (Full page version available for readability in Figure 8.27.)

space, but for classification tasks, this is not what we want. The required output for this task is a label. To get the label, we need to make a final tree conversion from $\mathbf{W}_e^{L-1}$. The final label is defined as:

$$\hat{y} = \underset{i}{\operatorname{argmax}} \left(\left(x(\mathbf{W}_e^{L-1})^T \right)_i \right) \tag{5.26}$$

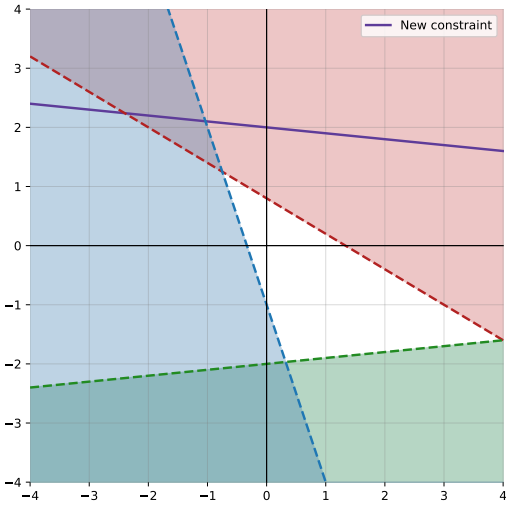
The argmax function can easily be defined as a sequence of comparisons, meaning that creating a tree is very straightforward. You select an element $\hat{y}_i$ of the output vector and create an inequality with another element $\hat{y}_j$ as $\hat{y}_i \geq \hat{y}_j$. By chaining these comparisons for each element in the vector, we ensure that each branch is assigned a label where its associated value is greater than that of all others and can become the sole defining element of the whole branch.

1.5 Pruning The Tree

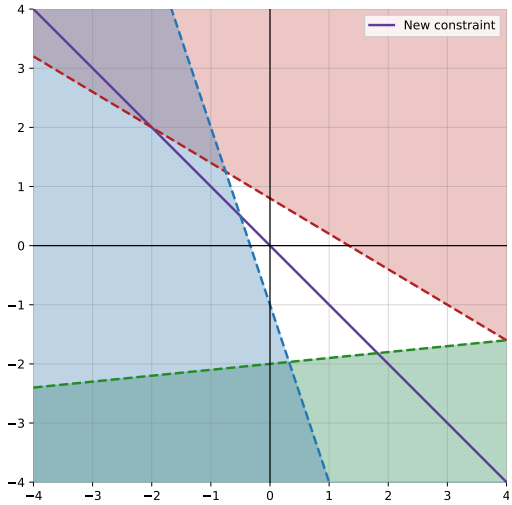
Not all the paths of the tree from the root to a node are reachable. Some paths are unattainable because of unsatisfiable constraints, thus the actual tree is a heavily pruned version of the base tree.

Since each constraint is linear, each branch of the tree encloses a polytope. Pruning occurs when the hyperplane of the new linear constraint does not pass through the enclosed polytope (see 2D example in Figure 5.15). Each node of the

tree splits into a true and a false case, but here, only one can happen because the polytope is fully on one side of the hyperplane, meaning only a single valid child node exists.



(a) Example where pruning happens, the new constraint does not pass through the polygon defined by the previous constraints.



(b) Example where pruning does not happen, the new constraint does pass through the polygon defined by the previous constraints.

Figure 5.15

For each path in the tree, we simply add each condition encountered to a linear program and run a feasibility check (solving with no particular objective, just finding a first solution to confirm it is feasible, or fail to do so). Each node in the path ϵ either contributes a condition $K_i \equiv x\mathbf{v}_i^T + b_i \geq 0$ if the right branch is taken, meaning the neuron is activated, or a condition $K_i \equiv -x\mathbf{v}_i^T - b_i \geq 0$ if the left branch is taken, meaning the neuron is non-activated for this path, $\mathbf{v}_i$ and b_i being the weights and bias associated with the i-th node along the path. The feasibility check is then:

$$\bigcap_{i \in \epsilon} K_i \neq \emptyset \tag{5.27}$$

This pruning can be done during the creation of the tree by solving the feasibility problem before creating the left and right nodes (see difference between Figures 5.16 and 5.17). This optimisation is exponentially faster than the base version and allows for the conversion to be done on much larger networks.

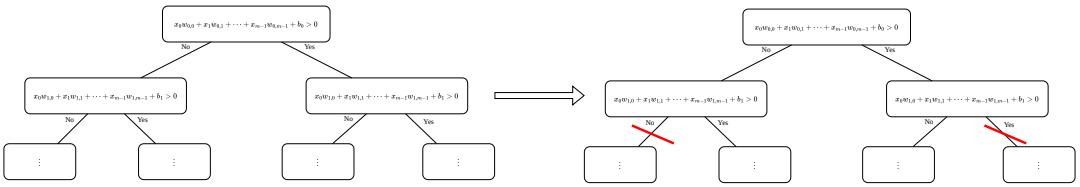


Figure 5.16: Passive pruning example: the tree is first fully built, then branches with unsatisfiable constraints are pruned.

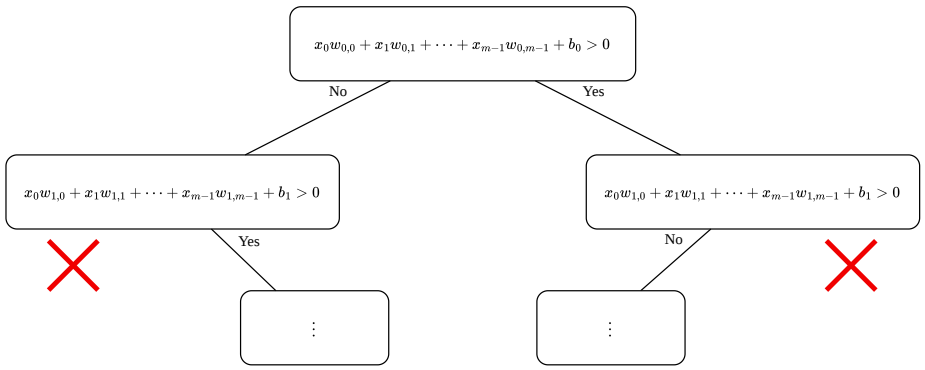


Figure 5.17: Active pruning example: the tree is built branch by branch, checking every time if all the constraints up to this point are satisfiable, pruning the branch if that is not the case by simply not creating it in the first place.

2 Distribution of Weights in a Neural Network

Knowing the distribution of weights in a neural networks could drastically help in deriving a closed-form solution to the expected number of linear regions. If we need to account for any possible distribution, it is not even sure an analytical solution does exist. This is why, if we know the distribution and this distribution has symmetries or other helpful properties, we can use them to derive our formula.

Definition 2.1 (Alpha-stable distribution)**.** *A distribution is stable if a linear combination of two independent random variables sampled from this distribution has the same distribution, up to location and scale parameters, the alpha-stable distribution is the complete family of these distributions.*

This distribution of the weights in a neural network has been successfully obtained in [LYK24] which demonstrates an alpha-stable distribution. Their derivation is as follows: we refer to the earlier SGD update in Equation (2.4), and consider that training does not use the entire data, but is done in batches instead.

The difference between the batch gradient and full gradient is $\mathbf{U}_\theta$. Following this, the update rule becomes:

$$\theta^{t+1} = \theta^t - \eta \nabla f_{\theta^t}(x_i; y_i) - \eta \mathbf{U}_{\theta^t} \tag{5.28}$$

$$\theta^T = \theta^0 - \eta \sum_{t=0}^{T-1} \nabla f_{\theta^t}(x_i; y_i) - \eta \sum_{t=0}^{T-1} \mathbf{U}_{\theta^t}. \tag{5.29}$$

Taking η to be small, this becomes the Lévy-Langevin equation with the stable distribution as the general solution:

$$d\theta_t = -\nabla f_{\theta^t}(x_i; y_i)dt - d\mathbf{U}_t. \tag{5.30}$$

The general density function cannot be expressed analytically, but its characteristic function, the Fourier transform of the density, can:

$$\varphi(t; \alpha, \beta, c, \mu) = e^{it\mu - |ct|^\alpha (1 - i\beta \operatorname{sgn}(t)\Phi)} \tag{5.31}$$

with Φ being:

$$\Phi(x) = \begin{cases} \tan(\frac{\pi\alpha}{2}) & \text{if } \alpha \neq 1 \\ -\frac{2}{\pi}\log(|t|) & \text{if } \alpha = 1 \end{cases}. \tag{5.32}$$

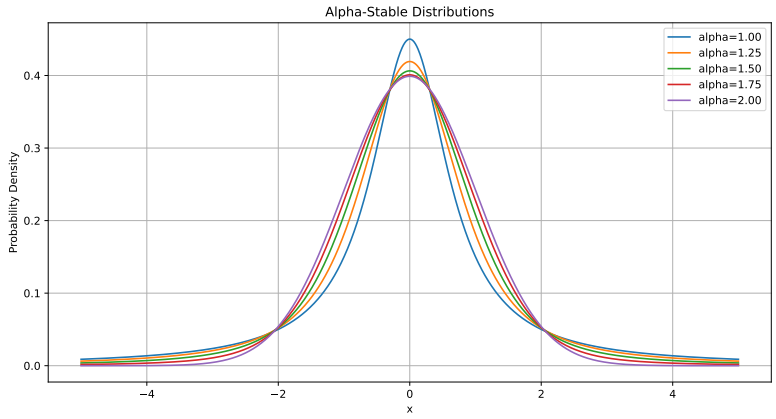


Figure 5.18: Different parametrizations of the alpha-stable distribution, going from a Cauchy distribution at $\alpha = 1$ to a Gaussian distribution at $\alpha = 2$.

To test if this theoretical result holds in practice, we fit an alpha-stable distribution to all weights of different large, open-source models of different architectures and modalities to ensure this observation is as general as possible. The selected

models are ResNet50 [HZRS15], Whisper Base [RKX+22] and Bert Base Uncased [DCLT18].

By fitting this distribution to these models (Figures 8.28 to 8.30), we observe that the alpha-stable distribution is a near perfect fit. It holds for all the weights with generic, centred shapes, but also the skewed histograms from ResNet50. It also shows a clear trend in the alpha parameter which increases with the size of the weight matrices (see Figure 5.19).

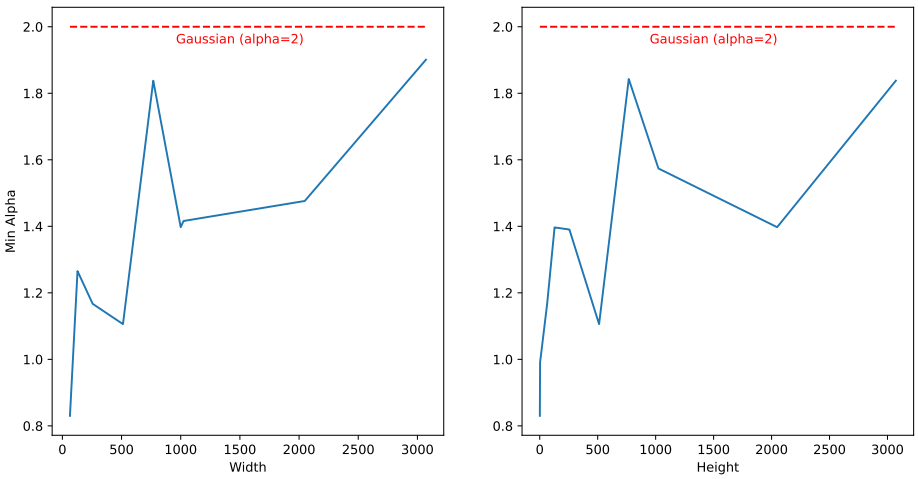


Figure 5.19: Minimum of the alpha parameter with respect to the width and height of the parameters of the large NNs. The minimum value for alpha increases both as width and height increases, meaning the more parameters a network has, the closer to Gaussian the weights are.

As the matrices of parameters get larger, the value for alpha increases and seems to approach 2, which corresponds to the normal distribution (see Figure 5.18). We will assume from this point forward that the distribution we are working with is a centred normal distribution. This distribution has several properties that will be useful:

1. **Rotational invariance**: The normal distribution, like the Cauchy or Student-t distributions, is symmetric about the origin. This means the distribution, when normalized, is uniform on $\mathbf{S}^d$, the d-dimensional sphere.

2. **General position holds almost surely**: General positioning is a statement about the values being spread randomly. A simple example would be in 2D, where general positioning means no 3 points are in a line. This property usually holds for continuous distributions.

3 Deriving the Formula

Lemma 3.1. *For a rectified network with isotropic normally distributed weights and input dimension $n_0 = min\{n_0, ..., n_{l-1}\}$, the expected number of linear regions N_L after the first layer is:*

$$\mathbb{E}[N_L] = 2\sum_{j=0}^{n_0} \binom{n_1 - 1}{j}$$

If we consider each node of the tree independently, then the path to that node defines an activation pattern ϵ and a system of constraints K_i in the same way as for pruning the tree in Equation (5.27), with active nodes contributing $K_i \equiv x\mathbf{v}_i^T + b_i \geq 0$ and non-active nodes contributing $K_i \equiv -x\mathbf{v}_i^T - b_i \geq 0$. These constraints define a polytope P and the node itself represents an additional constraint K_{i+1}.

$$\mathcal{H}_i = \{x \in \mathbb{R}^{n_0} : K_i\} \tag{5.33}$$

$$P_\epsilon = \bigcap_{i \in \epsilon} \mathcal{H}_i. \tag{5.34}$$

In Equation (3.14), we defined the number of linear regions with respect to the satisfiability of activation patterns. What we show here is that the expression for $P_\epsilon = \emptyset$ is equivalently stated as a geometry problem and we frame this geometry problem through the use of decision trees. Every node in the tree represents a polytope and the leaves are the linear regions of the network.

Lemma 3.2. *The expected number of linear regions can be expressed only from the probability that the constraint at depth $k+1$ intersects the polytope created by the k constraints that come before it:*

$$\mathbb{E}[N_L] = \prod_{i=1}^{k} (1 + p(cut_i))$$

Proof. Making use of the properties of the tree representation, the expected number of leaves N_L, and thus the expected number of linear regions, at a given depth k of the tree is the expected number at depth $k-1$ multiplied either by 1 or by 2 depending on $p(\text{cut}_k)$, the probability that the new constraint K_{i+1} passes, or cuts, through the polytope P_ϵ:

$$\begin{aligned}\mathbb{E}[N_L] &= \Big(2 \times p(\text{cut}_k) + 1 \times (1 - p(\text{cut}_k))\Big) \times \mathbb{E}[N_L|k-1] \\ &= (1 + p(\text{cut}_k)) \times \mathbb{E}[N_L|k-1] \\ &= \prod_{i=1}^{k} (1 + p(\text{cut}_i)).\end{aligned} \tag{5.35}$$

□

This is because the tree is a binary tree, if both sides are satisfiable, the node divides into two new nodes, but if the constraint does not pass through the polytope, then only one side is satisfiable and a single node is created.

3.1 Characterizing the cutting probability

At the first layer, the random polytope $P \subset \mathbb{R}^{n_0}$ is defined by the intersection of k half-spaces $\mathcal{H}$ where the coefficients $a_{i,j}, b_i \sim \mathcal{N}(0, \sigma^2)$ are independent Gaussian random variables, as established in Section 2. Such a space is called a Gaussian half-space. Given a new constraint defining a new half-space $\mathcal{H}_{i+1}$, the goal is to find the probability $p(P \cap \mathcal{H}_{i+1} \neq \emptyset \wedge P \cap \overline{\mathcal{H}_{i+1}} \neq \emptyset | P \neq \emptyset)$ which is the probability that the new constraint cuts the existing, non-empty polytope.

3.2 Derivation of the Probability

Wendel's theorem [Wen62] states that if k points are distributed uniformly on a sphere $\mathbf{S}^d$, then the probability that they all lie on a same hemisphere is:

$$S(k, d) = \frac{1}{2^{k-1}} \sum_{i=0}^{d-1} \binom{k-1}{i}. \tag{5.36}$$

Lemma 3.3. *The probability that the intersections of both a Gaussian half-space and its inverse with a non-empty intersection of k Gaussian half-spaces are also non-empty is:*

$$p(cut_k) = 2\frac{S(k+1, d)}{S(k, d)} - 1$$

This probability $S(k, d)$ is also the probability that the intersection of k random Gaussian half-spaces in $\mathbb{R}^d$ is non-empty. The argument goes as follows: we can consider the coefficients $(\mathbf{v}_i, b_i)$ to be vectors. Since their magnitudes do not matter, only their signs and directions, we can assume they are normalized, making them lie on the sphere $\mathbf{S}^d$. The intersection $\bigcap_{i \in \epsilon} \mathcal{H}_i$ is non-empty if and only if the origin

is not in the convex hull formed by the set $\{(\mathbf{v}_i, b_i) : i \in \epsilon\}$ and this event is equivalent to saying they all lie on the same hemisphere. This means we can use Wendel's theorem as the probability of $P_\epsilon \neq \emptyset$.

Consider a non-empty polytope P_k bounded by k hyperplanes. We define A_k as the event $\{P_k \neq \emptyset\}$. When a new random half-space $\mathcal{H}_{k+1}$ is introduced to the system to form $P_k \cap \mathcal{H}_{k+1}$, the probability that this intersection remains non-empty is the conditional probability $p(A_{k+1} \mid A_k)$. Since $A_{k+1} \implies A_k$, $p(A_{k+1} \wedge A_k) = p(A_{k+1})$:

$$p(P_k \cap \mathcal{H}_{k+1} \neq \emptyset \mid A_k) = \frac{p(A_{k+1} \wedge A_k)}{p(A_k)} = \frac{S(k+1,d)}{S(k,d)}. \tag{5.37}$$

A constraint cuts the polytope if both $\mathcal{H}_{k+1}$ and its complement $\overline{\mathcal{H}_{k+1}}$ have a non-empty intersection with the polytope. Let E^+ and E^- denote these respective intersection events. Due to the central symmetry of the Gaussian distribution, $p(E^+ \mid A_k) = p(E^- \mid A_k)$. Given that $p(E^+ \vee E^- \mid A_k) = 1$ for a non-empty P_k, the probability of a cut is derived via the principle of inclusion-exclusion:

$$\begin{aligned} p(\text{cut}_k) &= p(E^+ \wedge E^- \mid A_k) \\ &= p(E^+ \mid A_k) + p(E^- \mid A_k) - p(E^+ \vee E^- \mid A_k) \\ &= 2\frac{S(k+1,d)}{S(k,d)} - 1. \end{aligned} \tag{5.38}$$

This formula is in perfect alignment with simulations in Figure 5.20.

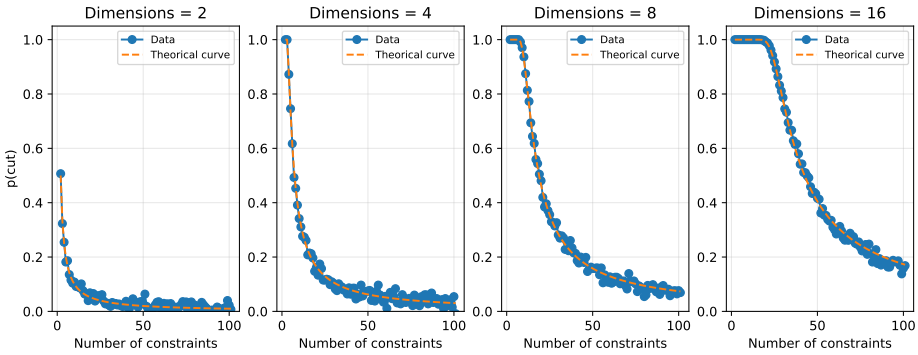


Figure 5.20: Comparison of the formula with simulations. The simulation was made by creating a random system of inequalities, ensuring it is satisfiable, then adding a new constraint. For each depth k, we run the solver on 5000 random different sets of constraints.

3.3 Closed-form solution for the expected number of leaves

Now that we have obtained the probability of a cut at a given depth k in Lemma 3.3, denoted as $p(\text{cut}_k)$, we can plug this result back into the recursive formula from Lemma 3.2. We must also lift the formula into $d+1$, since Wendel's theorem applies to hyperplanes that pass through the origin. Lifting into $d+1$ enables us to fulfil this condition. This is the same procedure that allowed us to include the bias in the tree decomposition in Equation (5.21). Bias is added to the weights and a new dimension is added to the input with a constant value of 1.

$$1 + p(\text{cut}_i) = 1 + \left(2\frac{S(i, d+1)}{S(i-1, d+1)} - 1\right) = 2\frac{S(i, d+1)}{S(i-1, d+1)}. \tag{5.39}$$

$$\begin{aligned}
\mathbb{E}[N_L] &= \prod_{i=1}^{k} (1 + p(\text{cut}_i)) \\
&= \prod_{i=1}^{k} \left(2\frac{S(i, d+1)}{S(i-1, d+1)}\right) \\
&= 2^k \prod_{i=1}^{k} \frac{S(i, d+1)}{S(i-1, d+1)}.
\end{aligned} \tag{5.40}$$

Notice that the product of probabilities forms a telescoping product. Assuming $S(0, d+1) = 1$ (the intersection of zero constraints is the entire space $\mathbb{R}^d$, which is inherently non-empty), the product simplifies drastically:

$$\begin{aligned}
\prod_{i=1}^{k} \frac{S(i, d+1)}{S(i-1, d+1)} &= \frac{S(1, d+1)}{S(0, d+1)} \times \frac{S(2, d+1)}{S(1, d+1)} \times \cdots \times \frac{S(k, d+1)}{S(k-1, d+1)} \\
&= \frac{S(k, d+1)}{S(0, d+1)} \\
&= S(k, d+1).
\end{aligned} \tag{5.41}$$

Thus, the expected number of leaves becomes:

$$\mathbb{E}[N_L] = 2^k S(k, d+1) = 2^{n_1} S(n_1, n_0 + 1). \tag{5.42}$$

Finally, substituting Wendel's Theorem for $S(k, d)$ yields a clean, closed-form solution:

$$\begin{aligned}
\mathbb{E}[N_L] &= 2^{n_1} \left(\frac{1}{2^{n_1 - 1}} \sum_{j=0}^{n_0} \binom{n_1 - 1}{j}\right) \\
&= 2 \sum_{j=0}^{n_0} \binom{n_1 - 1}{j}.
\end{aligned} \tag{5.43}$$

4 Extension to deeper networks

Theorem 4.1. *For a rectified network with isotropic normally distributed weights, input dimension $n_0 = min\{n_0, ..., n_{l-1}\}$ and n neurons, the expected number of linear regions N_L is:*

$$\mathbb{E}[N_L] = 2\sum_{j=0}^{n_0} \binom{n-1}{j}$$

The analysis so far assumes that the constraint coefficients $a_{i,j}$ and b_i are independently sampled from $\mathcal{N}(0, \sigma^2)$. We have shown in Section 2 that this holds for the first layer where the effective weights are simply the layer's weights. In deeper networks, however, the effective constraints at layer $l > 0$ are instead compositions of the transformations of the preceding layers, the effective matrices $\mathbf{W}_e$ from the tree decomposition. We now examine the distribution at a second layer and show that, under a mild and realistic asymptotic assumption, the Wendel-based formula remains valid and that this stays valid for the next layers.

4.1 Distribution of higher layers

Lemma 4.2. *The probability of a new constraint at deeper layers created from the effective weight $\mathbf{W}_e^l$ intersecting the polytope created by the existing constraints asymptotically becomes:*

$$p(cut_k) \xrightarrow{n_l \to \infty} 2\frac{S(k+1, d)}{S(k, d)} - 1$$

In the decision tree decomposition, the linear constraints defining the polytopes at depth k correspond to the rows of the effective weight matrices W_e^l. For the first layer ($l = 0$), these coefficients are simply the model's weights, which we postulated as i.i.d. $\mathcal{N}(0, \sigma^2)$. For deeper layers ($l > 0$), the effective weights are constructed recursively according to Equation (5.20):

$$\mathbf{W}_e^{l+1} = (\mathbf{W}_{l+1} \odot \mathbf{A}_\epsilon^T)\mathbf{W}_e^l \tag{5.44}$$

where $\mathbf{A}_\epsilon \in \{\alpha, 1\}^{n_l \times n_l}$ is the matrix encoding the activation pattern ϵ along a specific tree branch.

Consider the j-th coefficient $a_{k,j}$ of a constraint from layer l. It is formed by the dot product between the j-th row of $\mathbf{W}_l \odot \mathbf{A}_\epsilon^T$ and the corresponding column of $\mathbf{W}_e^{l-1}$:

$$a_{k,j} = \sum_{i=1}^{n_l} (\mathbf{W}_{l,ji} \cdot \mathbf{A}_{\varepsilon,ii}) \cdot \mathbf{W}_e^{l-1,ij}. \tag{5.45}$$

Since $A_{\varepsilon,ii}$ is fixed for the branch, each term in the sum is an independent random variable with mean zero and finite variance. By the Central Limit Theorem, as the layer width n_l grows, the distribution of $a_{k,j}$ converges to a Gaussian $\mathcal{N}(0, \sigma^2_{\text{eff}})$, regardless of the exact distribution of the previous effective weights. The same argument applies to the bias coefficients b_k.

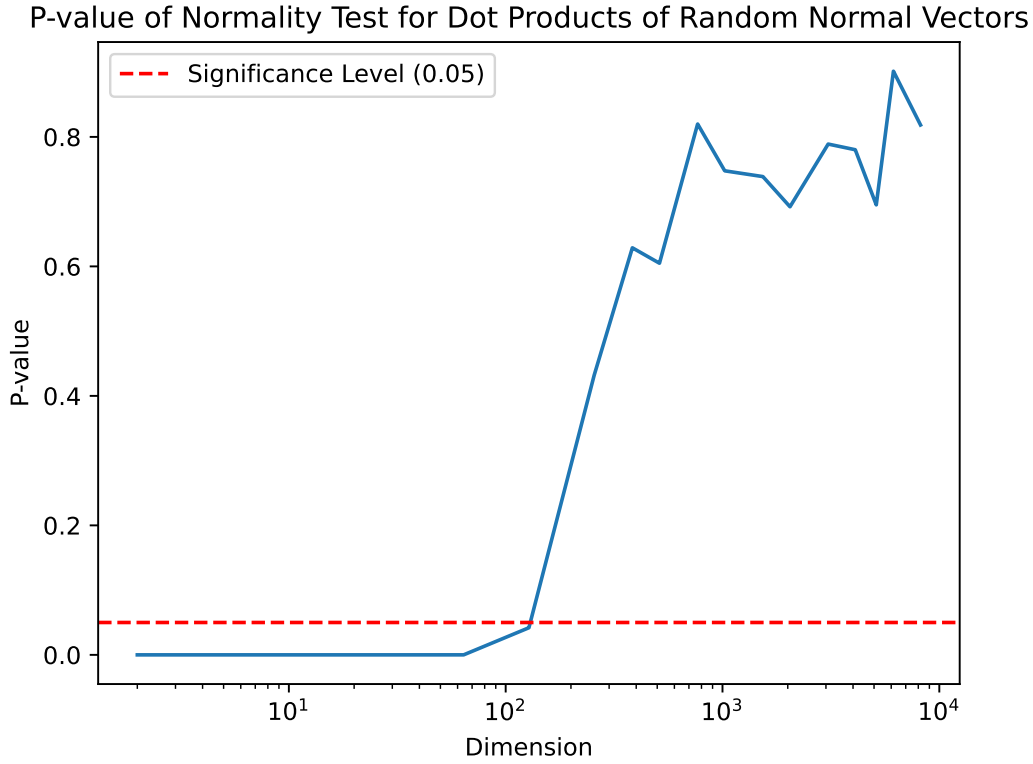


Figure 5.21: p-values for Kolmogorov-Smirnov tests comparing the distributions of the dot products of vectors with normally distributed elements and a normal distribution with a mean of 0 and the same standard deviation as the data.

The large width assumption holds for values of n_l well within modern standards for neural networks. We see from Figure 5.21 that there is a phase transition around 10^2 where the dot products go from non-Gaussian to firmly normally distributed. Modern neural networks operate with layers of thousands of neurons, meaning this shift in the hundreds is well passed and the effective weights we obtain for the tree decomposition, and consequently the coefficients for the constraints forming the polytopes, are normally distributed.

4.2 Scale Invariance of the Half-space

Let us suppose that a new constraint has coefficients $a_{k+1,j}, b_{k+1} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma'^2)$ with $\sigma' \neq \sigma$. The cutting condition depends on the new constraint only through the pair $(\hat{\mathbf{n}}, t)$. Writing $a_{k+1,j} = \sigma' Z_j$ and $b_{k+1} = \sigma' Z_0$ with $Z_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$:

$$\begin{aligned} \hat{\mathbf{n}} &= \frac{a_{k+1}}{\|a_{k+1}\|} \\ t &= \frac{b_{k+1}}{\|a_{k+1}\|} = \frac{\sigma' Z_0}{\sigma' \sqrt{\sum_{j=1}^{d} Z_j^2}} = \frac{Z_0}{\sqrt{\sum_{j=1}^{d} Z_j^2}} \end{aligned} \tag{5.46}$$

which is a Student-t distribution with d degrees of freedom, independent of σ'. Since $\hat{\mathbf{n}}$ is also uniform on S^{d-1} for any $\sigma' > 0$, the distribution of the half-space $\mathcal{H}_{k+1}$ is identical under $\mathcal{N}(0, \sigma'^2)$ and $\mathcal{N}(0, \sigma^2)$. Consequently, the probability $p(\text{cut}_k)$ stays the same, independently of both σ and σ' (see Figure 5.22).

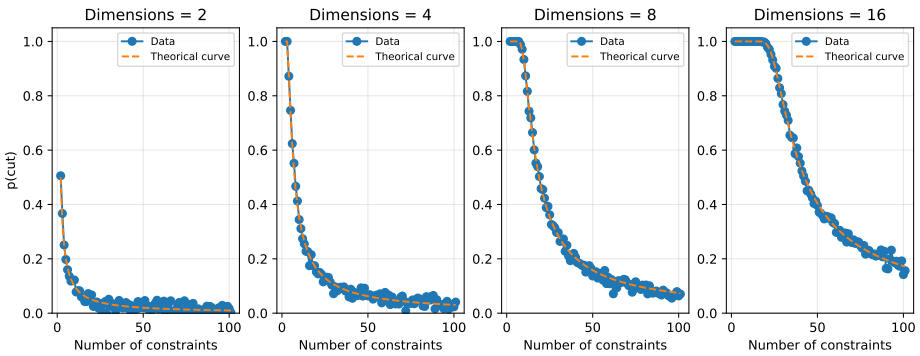


Figure 5.22: Comparison of the formula with simulations. The simulation was made by creating a random system of inequalities, each constraint with a different standard deviation, ensuring it is satisfiable, then adding a new constraint. For each depth k, we run the solver on 5000 random different sets of constraints.

4.3 Recovering the Wendel formula

We now set the following assumption: the dimension n_l, corresponding to the number of neurons of the hidden layer, is large, larger than the phase transition point.

As stated in Section 4.1, the distribution of a_{k+1} and b_{k+1} becomes an isotropic Gaussian distribution in the limit. As shown in Section 4.2, the cutting probability is independent of the variance, and so:

$$p(\text{cut}_k) \xrightarrow{n_l \to \infty} 2\frac{S(k+1,d)}{S(k,d)} - 1 \tag{5.47}$$

recovering exactly the formula derived for the first layer.

4.4 Number of linear regions

From Lemma 4.2, since the probability is unchanged, the entire derivation of the first layer therefore also carries through unchanged, and the expected number of leaves at depth k stays the same. The final formula for the expected number of linear regions a neural network with n neurons and n_0 inputs contains is:

$$\mathbb{E}[N_L] = 2\sum_{j=0}^{n_0} \binom{n-1}{j} \tag{5.48}$$

5 Asymptotic Analysis

The formula for the expected number of linear regions a neural network with n neurons and n_0 inputs is $2\sum_{j=0}^{n_0}\binom{n-1}{j}$. This formula simplifies asymptotically for several cases, most notably for the case where there are more constraints than input dimensions, giving the formula proposed in [HR19b].

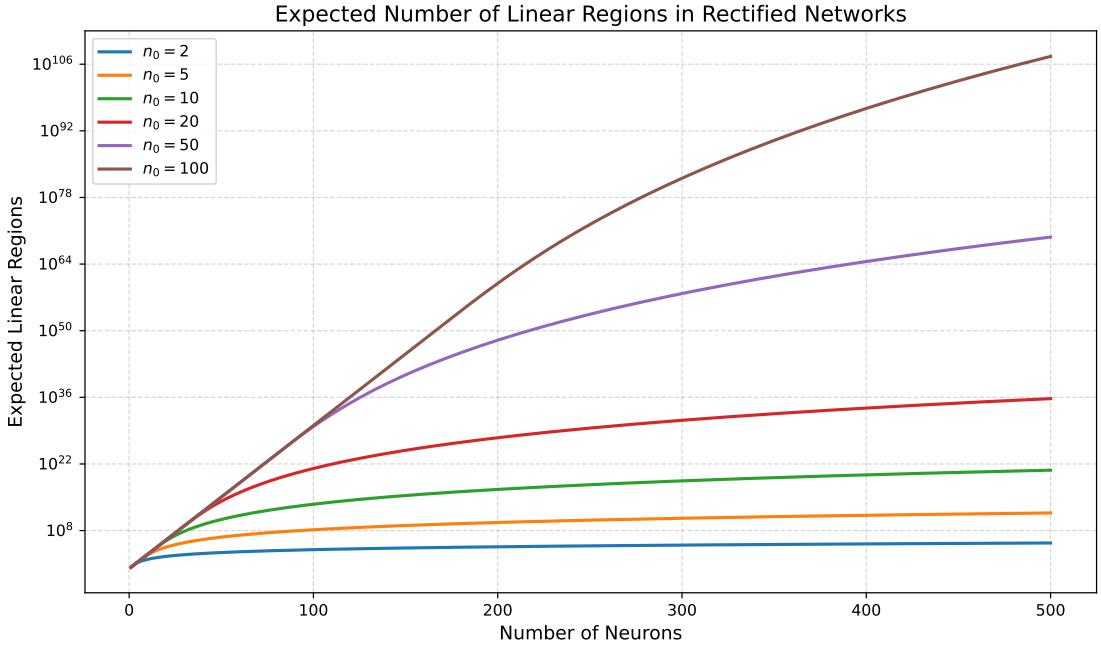


Figure 5.23: Expected number of linear regions with respect to the number of neurons for different input dimensions.

We see in Figure 5.23 that the expected number of linear regions starts with exponential growth until about twice the input dimension, at which point the growth slows down drastically and even becomes imperceptible on a logarithmic scale.

5.1 Low input dimension, large network ($n \gg n_0$)

To find the asymptotic behaviour, we look at the dominant term in the sum, which occurs at the highest index $j = n_0$:

$$\binom{n-1}{n_0} = \frac{(n-1)(n-2)...(n-n_0)}{n_0!} \tag{5.49}$$

Asymptotically, this becomes:

$$\mathbb{E}[N_L] \sim \frac{n^{n_0}}{n_0!} \tag{5.50}$$

5.2 Input dimension similar to the number of neurons ($n \approx n_0$)

Let $n_0 = \gamma n$ with $0 < \gamma < 1$. From Stirling's approximation, in this regime, we get:

$$\begin{aligned} \mathbb{E}[N_L] &\sim 2^{NH(\gamma)} \\ \text{where } H(\gamma) &= -\gamma \log_2 \gamma - (1-\gamma)\log_2(1-\gamma) \end{aligned} \tag{5.51}$$

1. If $\gamma < 1/2$, it becomes the previous case as γ approaches 0.
2. At $\gamma = 1/2$, $H(1/2) = 1$, marking the transition to exponential growth.
3. If $\gamma > 1/2$, the sum captures most of the binomial mass and approaches 2^n, becoming the next case as γ becomes greater than 1.

5.3 Large input dimension compared to the number of neurons ($n < n_0$)

Since $\binom{n-1}{j} = 0$ for $j > n-1$, the sum truncates naturally:

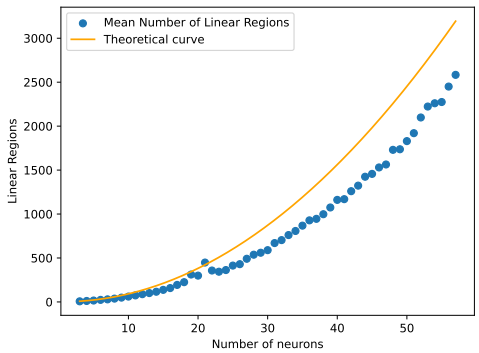
$$\sum_{j=0}^{n_0} \binom{n-1}{j} = \sum_{j=0}^{n-1} \binom{n-1}{j} = 2^{n-1} \tag{5.52}$$

$$E[N_L] = 2 \cdot 2^{n-1} = 2^n \tag{5.53}$$

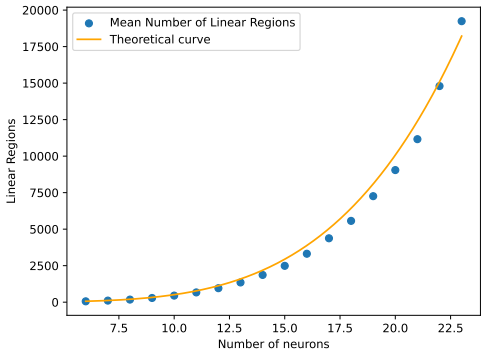
The network saturates at the maximum possible number of regions for n hyperplanes in $\mathbb{R}^{n_0}$.

Validation

1 Comparison Between Formula and Experiments



(a) Comparison of the number of linear regions in neural networks trained on the half-moon dataset, averaged over every network configuration for each size.



(b) Comparison of the number of linear regions in neural networks trained on the iris dataset, averaged over every network configuration for each size.

Figure 6.24

Comparing the formula we found with the data from the neural networks trained on the half-moon and iris datasets (see Figure 6.24), we do not expect a perfect agreement as the dimensions are too low to satisfy the Gaussian distribution for the effective weights at deeper layers. However the curves still match very well. For the half-moon dataset, both curves start very close together, but as the number of neurons increases and the deeper layers weigh more and more, they begin to deviate from each other. On the other hand, the networks trained on the iris dataset follow the theoretical curve very closely with minimal differences at higher number of neurons, once again when the network configurations with several layers start appearing.

2 Linear Regions and Network Expressivity

Our result in Theorem 4.1 implies that the expressivity of a rectified network is independent of its number of layers and only depends on the number of neurons. We ran an experiment to test this claim, taking accuracy as a substitute for expressivity.

2.1 Experimental Setup

We first create a dataset using Scikit's `make_classification` method [PVG+11]. We then train 3 different networks on this dataset:

- A first network with a random number of hidden layers between 3 and 6 with each layer having between 256 and 1024 neurons each to match our observations in Figure 5.21.
- A second network with a number of hidden layers between 2 and one less than the number of layers from the first network, still with a number of neurons per layer of at least 256.
- A network with a single hidden layer.

All three of these networks are made to have the same number of neurons, meaning the same expected number of linear regions. Since deeper networks require more training signal, the number of epochs for each group of configurations is proportional to the depth of the first configuration, the deepest one.

2.2 Results

We observe in Figure 6.25 no significant difference in performance between the shallow and deep configurations, with a mean absolute difference of less than 1% and a mean difference of -0.0094 ± 0.0074, statistically consistent with a difference of 0. The average difference in accuracy between the single layer network and both other configurations is about 1% while the deep configurations have on average $8.11\times$ more parameters and up to $12\times$ more.

Even though all three configurations have very different parameter counts, they all show very close, if not indistinguishable, performances, seemingly validating the prediction that expressivity is independent of the number of layers.

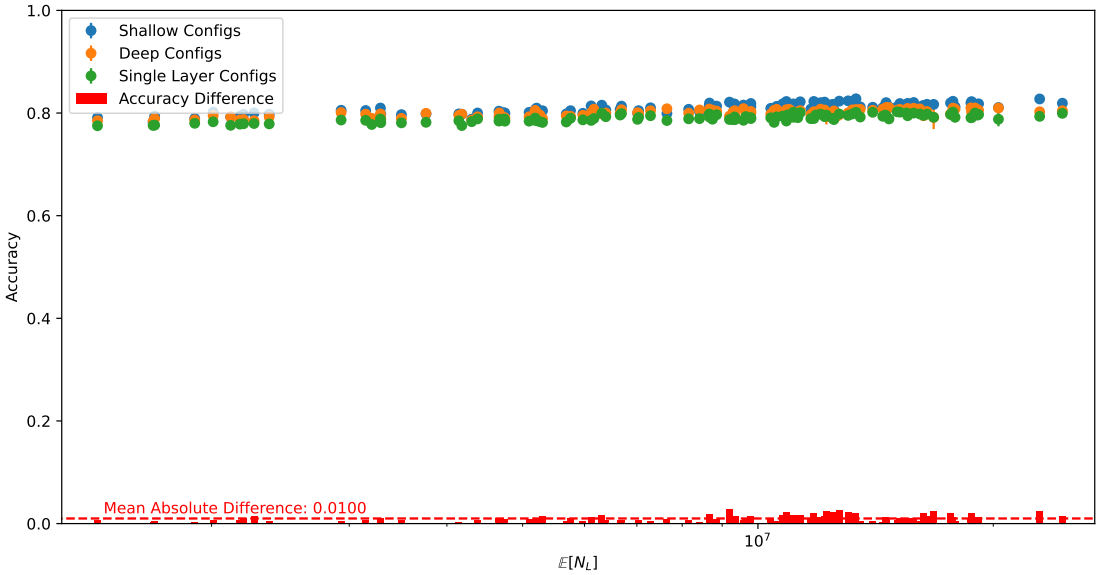


Figure 6.25: This plot shows 4 distinct elements: (1) in orange are the accuracies from deep networks, (2) in blue are the accuracies from shallow networks with fewer layers than their corresponding deep network, (3) in green are the accuracies of single layer networks with equal neuron count and (4) in red is the absolute difference between the corresponding blue and orange points.

Conclusion

We have successfully proven the conjectured formula for the number of linear regions in a rectified network. This holds several implications for our understanding of neural networks and leaves some questions open for later works.

The main implication is on our comprehension of the impact of depth in neural networks. It was long thought that depth was an important factor in the success of these models and provided an exponential increase in their capabilities, but we have shown that for an equal number of neurons, a shallower, but wider network will have the same expressiveness as a deeper network. Where deep networks still hold an advantage is parameter count: for the same number of linear regions, they often require fewer parameters.

This work only covers feed-forward networks, many other architectures exist and are more widely used in production. Working on finding similar scaling laws for architectures such as convolutional neural networks or transformers could be of immense help for the current state of the field.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Ayt22] Caglar Aytekin. Neural networks are decision trees, 2022.

[Chv83] Vašek Chvátal. *Linear programming*. A series of books in the mathematical sciences. W. H. Freeman, New York, 1983 - 1983.

[Dan02] George B. Dantzig. Linear programming. *Operations Research*, 50(1):42–47, February 2002.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[HR19a] Boris Hanin and David Rolnick. Complexity of linear regions in deep networks, 2019.

[HR19b] Boris Hanin and David Rolnick. *Deep ReLU networks have surprisingly few activation patterns*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *ICLR (Poster)*, 2015.

[LYK24] Jipeng Li, Xueqiong Yuan, and Ercan Engin Kuruoglu. Exploring weight distributions and dependence in neural networks with α-stable distributions. *IEEE Transactions on Artificial Intelligence*, 5(11):5519–5529, 2024.

[Mon17] Guido Montufar. Notes on the number of linear regions of deep neural networks, 03 2017.

[MPCB14] Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks, 2014.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RKX+22] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022.

[STR17] Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. Bounding and counting linear regions of deep neural networks. *CoRR*, abs/1711.02114, 2017.

[Wen62] J. G. Wendel. A problem in geometric probability. *MATHEMATICA SCANDINAVICA*, 11:109–112, Jun. 1962.

[Wol98] Laurence Wolsey. *Integer Programming*. Wiley, 1998.

Appendix

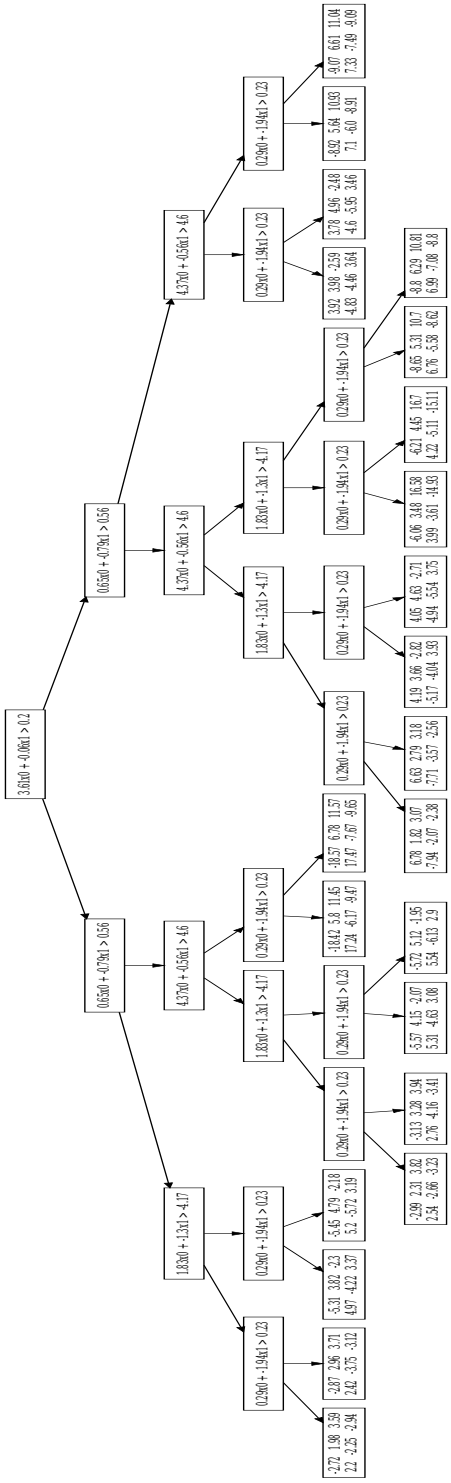


Figure 8.26: Full size tree from Figure 5.13.

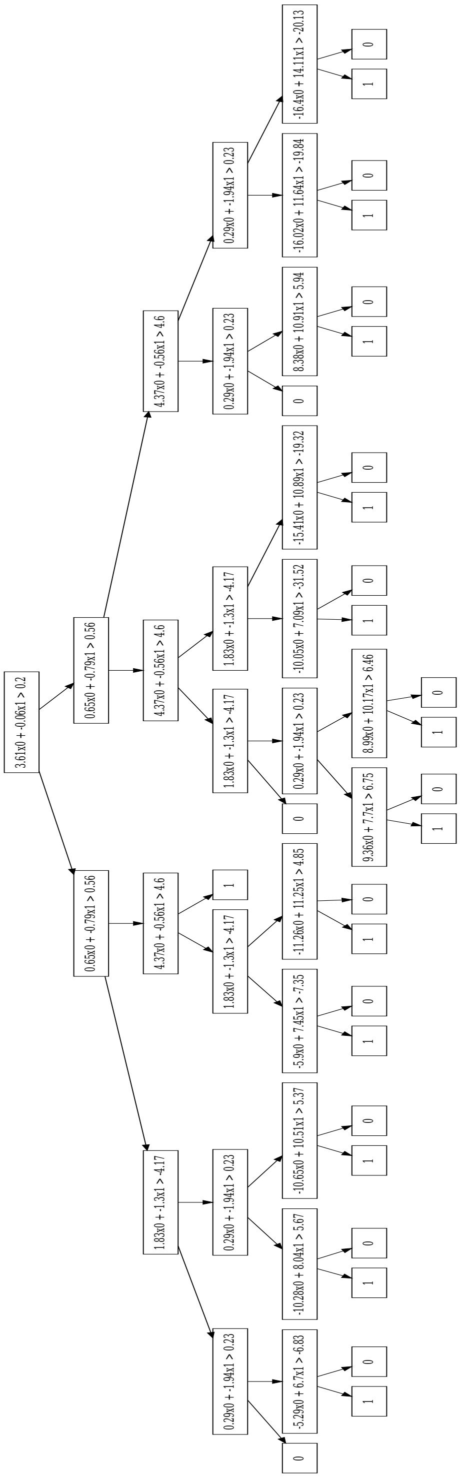


Figure 8.27: Full size tree from Figure 5.14.

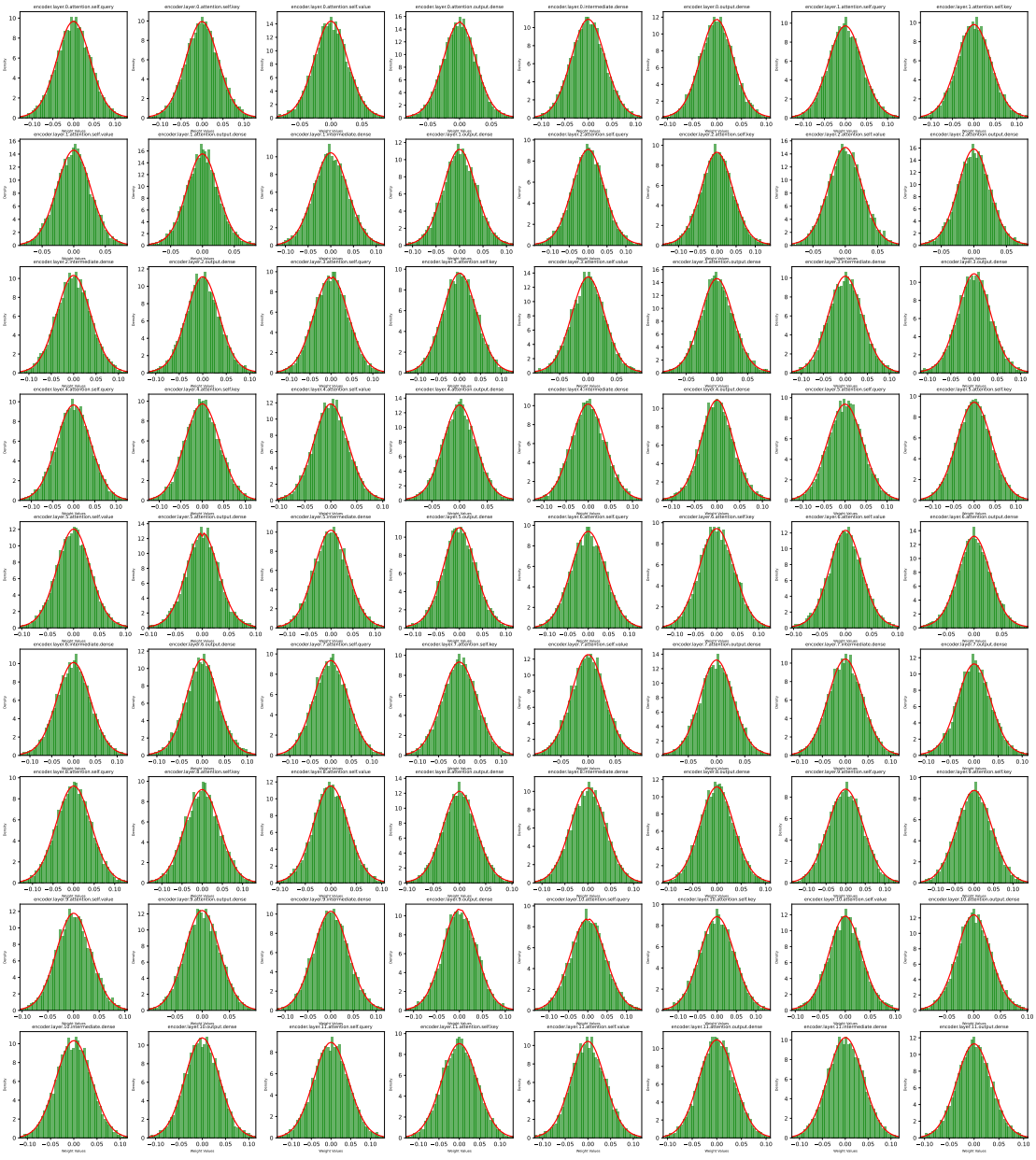


Figure 8.28: Histograms of weights and alpha-stable fit for all weights in the *Bert Base Uncased* model.

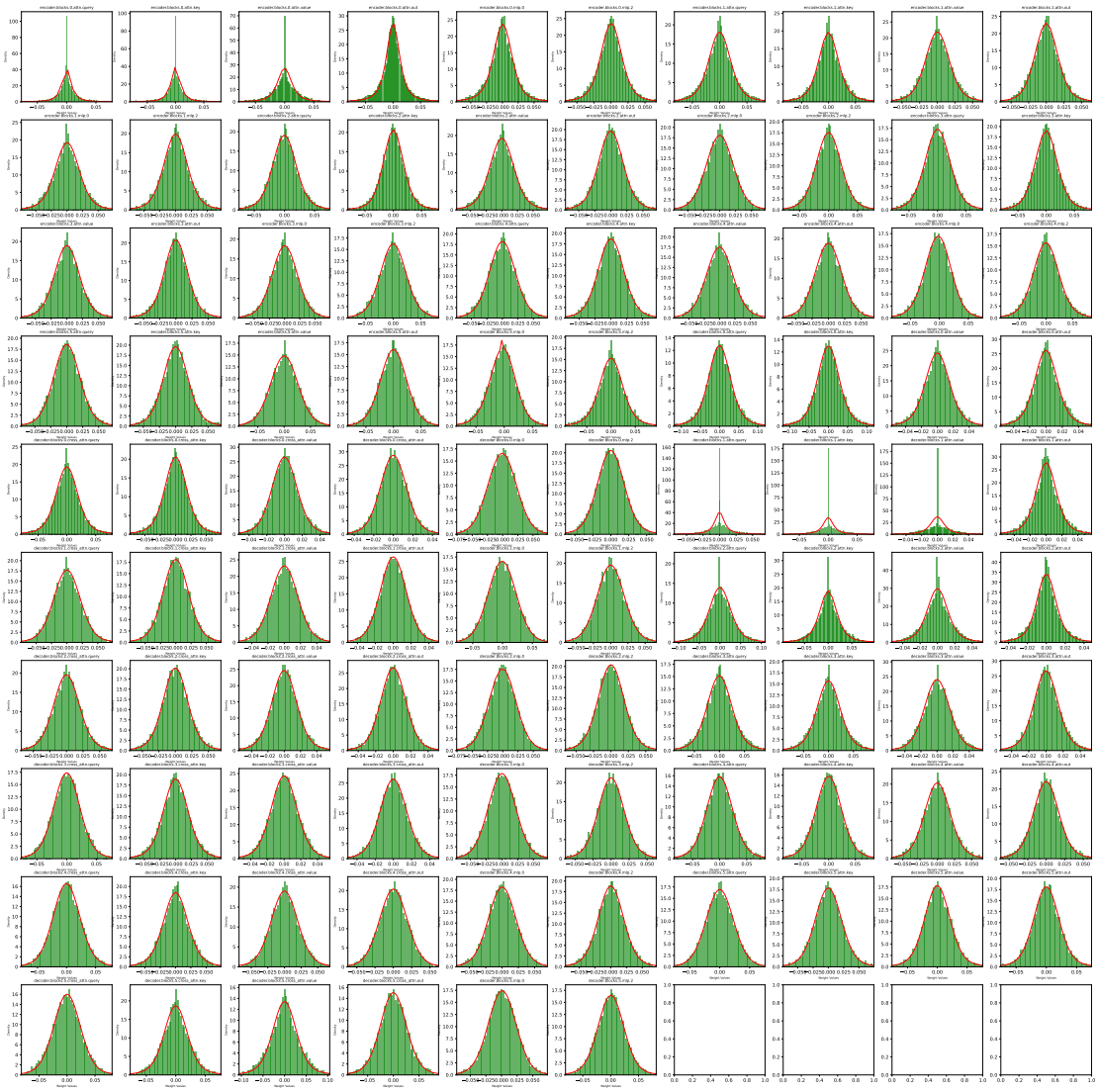


Figure 8.29: Histograms of weights and alpha-stable fit for all weights in the *Whisper Base* model.

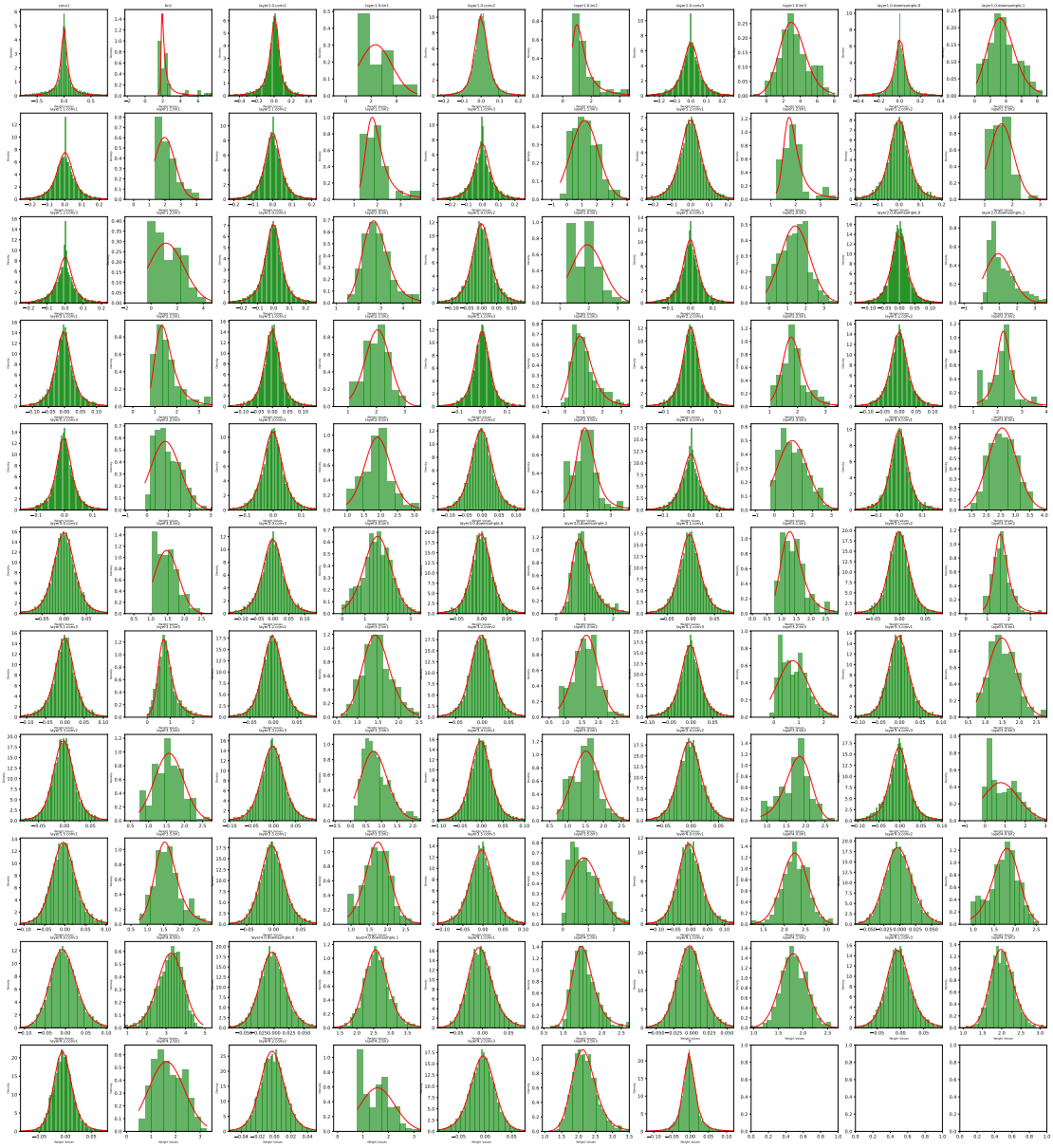


Figure 8.30: Histograms of weights and alpha-stable fit for all weights in the *ResNet50* model.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**