The Inside Story: Towards Understanding Privacy Leakage of Neural Networks

Masterarbeit

im Rahmen des Studiengangs

Informatik

der Universität zu Lübeck

vorgelegt von

Thorsten Peinemann

ausgegeben und betreut von

Prof. Dr. Esfandiar Mohammadi

Lübeck, den 01. Oktober 2021

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Ort, Datum Unterschrift

Zusammenfassung

In dieser Arbeit entwickeln wir Werkzeuge, die helfen zu verstehen, wie und was neuronale Netze auf einem synthetischen Datensatz lernen. Dieser Datensatz ist zweidimensional, da die Visualisierung bei hochdimensionalen Daten schwierig ist. Der Datensatz ist eng danach modelliert, wie reale Datensätze höchstwahrscheinlich aussehen. Als neue mögliche Leakage leiten wir daraus Polytopfunktionen ab. Ein neuronales Netz lernt diese Funktionen für einzelne Polytope im Eingaberaum. Mit zwei neuartigen Membership-Inference-Attacken zeigen wir, dass es tatsächlich Leakage durch unseren Ansatz gibt, aber weniger als wir hofften. Wir diskutieren Covers Theorem [1] [2] als möglichen Grund dafür. Wir argumentieren, dass in unserem Setting, die Confidence eines neuronalen Netzes mehr Leakage bietet. Mithilfe unserer Visualisierung beobachten wir erste Hinweise darauf, dass neuronale Netze zu Beginn des Trainings einen einzigen Bergrücken mit großen Polytopfunktionsparametern aufbauen, wobei in späteren Epochen weitere Bergrücken hinzukommen. Wir erklären diesen Effekt mathematisch durch Forward Weight Dependency und Backward Weight Dependency. Wir beobachten auch, dass nach dem Training die Polytopfunktionsparameter mit Entscheidungsgrenzen korrelieren. Darüber hinaus zeigen wir, dass Gradient-Boosted-Decision-Trees eine sinnvolle Option als Angriffsmodell für Membership-Inference-Attacken sind, indem wir die Angriffsgenauigkeit des MLLeaks-Angriffs [3] leicht verbessern.

Abstract

In this thesis, we develop tools that help in understanding how and what neural networks learn on a synthetic dataset. This dataset is two dimensional, since visualization is hard on high dimensional data. It is closely modeled after what real datasets most likely look like. We derive a new possible source of leakage: Polytope functions, i.e. the individual functions the neural network has learned for a specific polytope in input space. With two novel membership inference attacks we show that there is in fact leakage through our approach but less than we had hoped. We discuss Cover's theorem [1] [2] as a possible reason for this. We argue that in our setting, the confidence of a neural network is more promising for leakage extraction. Using our visualization, we observe first indications of neural networks building a single mountain ridge of large polytope function parameters in the beginning of training with more mountain ridges rising up in later epochs. We mathematically explain this effect through forward weight dependency and backward weight dependency. We also observe that after training, the polytope function parameters correlate with decision boundaries. Furthermore, we show that gradient boosted decision trees are a viable option as an attack model for membership inference by slightly improving the attack accuracy of the MLLeaks [3] attack.

Table of Contents

1 Introduction

Artificial intelligence is part of many services, some of which are accessed multiple billion times a day [4]. Be it an internet search engine, asking a car where to drive when heading towards a specific location, or shopping online and going through suggested items. This social and economic development has raised a series of questions on whether the data that individuals contribute to such artificial intelligence is safe with the data aggregator. Even if the data aggregator itself is not susceptible to a direct attack with theft of data per se, privacy research in the field of artificial intelligence has made some startling discoveries in recent years.

Neural networks are a very popular form of artificial intelligence. This is due to their applicability in nearly any field where statistics or data has been aggregated as well as their empirically proven effectiveness on many real world problems. Despite their popularity, classically trained neural networks leak information about their training data. Fredrikson et al. [5] are the first ones to introduce such an attack on a neural network (or model) by using the certainty output by the model under attack. Attacks like MLLeaks by Salem et al. [3] and The Secret Revealer by Zhang et al. [6] are respectively able to safely infer whether a given datapoint is part of training data and to even completely reconstruct large parts of training data. Through exposing the weaknesses of neural networks these attacks teach people to be more concerned about their privacy and inform the research community about the main vulnerabilities of neural networks that need to be fixed.

Most privacy attacks from the literature focus on extracting leakage by looking at the output of the targeted models. Jayaraman et al. [7] introduce a threshold based attack that rather looks at the loss function.

We began our work with the hypothesis that activation polytopes leak information about training data. These polytopes subdivide the input space of a neural network and every one of them describes an affine linear function the neural network has learned, as Raghu et al. [8] state. In comparison to a classical approach that uses the prediction or the confidence of the target model, our approach is not able to extract as much leakage.

There is still a huge gap between the theoretical point of view, trying to understand neural networks, and the practical point of view, trying to create effective attacks. The present thesis lessens the size of this gap by bringing the two sides together: Through an in-depth analysis of the behavior of neural networks, accompanied by easy to use visualization tools, and also the creation of attacks that are supported by observations made through the use of our tools.

Our Contribution

The construction of privacy attacks can be a road that is paved with trial and error and a lot of hardship. Finding new leakage possibilities and the evaluation of attacks is mostly done empirically, in that the resulting attack is run on a number of standard datasets to show that it is in fact capable of leaking sensitive information of the targeted model. On the one hand, this makes it harder to analyze why a specific attack is actually working. On the other hand, if the attack fails, it is more often than not hard to explain why this attack is not functioning properly. Generic frameworks for the evaluation of privacy attacks and visualization tools for neural networks could help with this. On top of that, to this day, the questions of how exactly neural networks learn what they learn and how they encode their knowledge are still mostly unanswered but definitely highly relevant for the discovery of new weak spots of neural networks.

We introduce a new set of tools and insights that will hopefully help to overcome these shortcomings of privacy research on neural networks. We create a synthetic two dimensional dataset that is supposed to resemble the layout of high dimensional data in section 4.2. On this dataset we apply our novel visualization tool that samples over the input space and generates images that showcase the output, the parameters of polytope functions and even the layout of hyperedges of associated neurons of the target model over the input space (see section 4.2 and section 5.3). The polytope functions stem from the work by Raghu et al. [8] who mention that neural networks with piecewise linear activation functions have their input space divided into convex polytopes that are bordered by hyperedges, each belonging to a single neuron of the neural network. Each of these polytopes has an associated linear function (the polytope function), describing the output of the neural network in this polytope.

We derive these polytope functions as a possible new source of leakage. There is in fact leakage through our approach, but not as much as we had hoped. As a reason for this, we discuss Cover's theorem [1] [2] in section 7.3 and in section 7.5.

In section 5.4 we are able to observe first indications of an interesting effect: Neural networks seem to create mountain ridges between data clusters of different classes, using its output and its polytope function parameters. We notice that in the beginning of the training phase of the neural network, it starts by creating a single mountain ridge for an initial split of training data and in later epochs this mountain ridge splits up or new mountain ridges rise up. We give a possible explanation for this in a mathematical way.

We use our framework and our insights to construct two new privacy attacks (see chapter 4 and chapter 6) that aim at membership inference (classifying whether a given datapoint is part of training data), one of which is able to perform on par on some datasets with latest research in the field by Choquette et al. [9] in the best case.

Summary of Contributions

1. We develop a visualization tool for neural networks trained on two dimensional datasets that helps in evaluating and implementing privacy attacks (see chapter 5).
2. We thoroughly analyze the effect of training data on the shape and layout of activation polytopes of neural networks and use our visualization to understand how and what neural networks learn and where there is possible leakage. In section 5.4, we are able to observe first indications of the way neural networks train on data. For these indications we give a mathematical explanation.
3. We deploy a membership inference attack in section 4.5 that exploits the shape of activation polytopes that we observe.
4. We deploy a gradient descent based decision boundary attack for membership inference in section 6.4. This attack optimizes towards low confidence, small output and large polytope function parameters in finding the closest boundary.
5. We conduct extensive experiments to evaluate the performance of our attacks in chapter 7. We find that the gradient descent based attack performs on par with latest research on some datasets in the best case. We also conclude that in our framework, confidence based attacks are more performant and complete than any attack that uses the output of the neural network or its polytope function parameters.

6. We assess the usage of gradient boosted decision trees as an attack model for membership inference in section 7.4 and succeed in slightly improving the MLLeaks attack by Salem et al. [3].

Structure of this Work

Chapter 2 looks at some reseach that is relevant for the present thesis. Chapter 3 then lays a ground work of understanding of neural networks. In chapter 4 we present our first attack, the activation polytopes attack. In chapter 5 we discuss some new observations we made on the inner workings of a neural network. Chapter 6 introduces our second attack, a gradient descent based boundary distance attack. Experiments and evaluation of the attacks take place in chapter 7. Lastly, a conclusion and an outlook are given in chapter 8.

2 Related Work

This chapter looks at some research which is closely related to our work or is something that we build upon and compare ourselves to.

2.1 Membership Inference Attacks

Shokri et al. [10] were the first ones to demonstrate that membership inference attacks can be carried out in a real life setting where the target model (for example a neural network) under attack is used for a classification task and the attacker only has black box access. For membership inference, an attacker tries to infer for a given datapoint whether the target model has used this datapoint in training (the datapoint is a member) or not (the datapoint is a nonmember). Shokri et al. assume, that the attacker has access to the prediction of the target model, i.e. a certainty value for each of the possible classes. In this thesis, we will also take a look at the internal structure of the target model. In preparation for their attack, multiple so called shadow models are trained. These shadow models are all trained independently and supposed to behave similar to the target model, so they should have a similar structure and employ a similar learning technique. With the trained shadow models, the attacker generates multiple input / prediction pairs for members and nonmembers of the shadow models' datasets and then tries to learn specific patterns in the prediction that correlate with membership / nonmembership. Shokri et al. use multiple attack models here: One attack model per class of the target model.

The MLLeaks paper by Salem et al. [3] improves on Shokri et al.'s results. Salem et al. introduce three different adversaries resulting in three different types of attack strengths: The first adversary has access to a dataset from the same distribution that the training dataset of the target model comes from. Here, the authors show, that even with a single shadow model and a single attack model, the accuracy is very similar to the one by Shokri et al. We will use this adversary as well when comparing our activation polytopes attack to the MLLeaks attack.

We use different metrics than just the top three certainties from the target model output: We look at the structure the target model has formed through its weights and biases. To the best of our knowledge, this has been completely disregarded in membership inference attacks so far.

The second adversary neither has access to a dataset from the same distribution as the training dataset of the target model, nor does it know, what the target model's structure looks like. This only results in a small drop of attack accuracy but leads to a more realistic scenario. The third adversary completely refrains from usage of a shadow model and simply distinguishes members and nonmembers based on whether the biggest certainty value in the output of the target model is above or below a previously chosen threshold.

For picking the threshold, Salem et al. propose the following approach: The target model is queried with a number of random datapoints and the maximum certainty value of their predictions is noted. These random datapoints are assumed to be nonmembers, so a top percentile of their maximum certainties are averaged and taken as threshold.

2.2 Loss Function Membership Inference

Jayaraman et al. [7] investigate membership inference based on the loss function of a target model. Doing so, they take a different approach than leakage through the output of a target model which has been common thus far. For a given datapoint, their attack perturbs the datapoint and then measures the loss in comparison to the loss at the original datapoint. If the given datapoint is a member of the training dataset, then it is likely to be positioned at a minimum with regard to the loss function. Most of the perturbations will therefore induce a higher loss function value. If the given datapoint is a nonmember, the perturbations will more or less have an equal probability of inducing a greater or smaller loss function value. Measuring this behavior for a datapoint and comparing with a threshold is used to infer membership.

We try to leak information about training data through the structure of activation polytopes and through polytope function parameters (for details see section 3.3 and section 3.4. Thus we also take a different approach than leakage through the output, but one that is different to Jayaraman et al. We compare our approach to the classic approach of using the output or the confidence of the target model.

2.3 Label Only Black Box Membership Inference

A recent paper by Choquette et al. [9] introduces a black box membership inference attack that has minimal assumptions: Only the output label of the target model is given to the attacker. However, multiple queries to the target model are required.

Choquette et al.'s attack is a boundary distance attack, meaning that it tries to determine a given datapoint's distance to the closest decision boundary, i.e. the boundary where the target model's prediction changes from the originally predicted label on the datapoint to a different label. For a given datapoint, the attack starts with a random perturbation of that datapoint, so that the perturbation is classified with a different label. Then they finetune on the boundary with a binary search like approach, namely the HopSkipJumpAttack by Chen et al. [11].

We deploy a boundary distance attack as well, but with a novel approach: We use a gradient descent based strategy and use this optimization algorithm to move from a given datapoint to low confidence, small output and to parts of the neural network where its function has large parameters.

2.4 Model Inversion

Fredrikson et al. [5] are the first ones to excerpt leakage from a target model using the target model's certainties from its output and putting it to use in a model inversion attack.

The recent construction of The Secret Revealer by Zhang et al. [6], introduces a novel model inversion attack that tries to gain information about training data in a white box setting with possible auxiliary information like nonsensitive parts of training data. The attack consists of two phases.

In the first phase of this attack, a generative adversarial network (GAN) is trained on public data. This GAN is able to produce manifold but realistic looking images in accordance with training data, e.g. images of animals if the training data consists of image of cats and dogs and so on. In this phase, the target model's internals are used to optimize the GAN to generate more diverse images instead of just a small collection of images on repeat.

In the second phase, the actual attack takes place: Realistic images are generated using the GAN, but the images should achieve high confidence under the target model, since this is thought to correlate with membership (belonging to training data). Using gradient descent as an optimization algorithm, the produced images are updated over multiple iterations. The images will then very likely resemble images from training data.

For our gradient descent based attack, we build on this paper by Zhang et al. However, there are some key differences: Firstly, we use gradient descent for membership inference instead of model inversion. Secondly, we optimize towards low confidence, rather than high confidence, as we already are given a datapoint and are looking for a nearby decision boundary. And thirdly, we utilize different metrics like the norm of the target model's function's parameters, which is motivated by visualization of a target model and a thorough investigation of possible leakage beforehand. To the best of our knowledge, this represents a part of neural networks that has been neglected in membership inference so far.

3 Preliminaries and Definitions

Before we are able to present our attacks, we need to define some basics, connected to neural networks and associated topics. Since we analyze neural networks in depth and develop new insights, it is important that here, we lay a good ground work of understanding of neural networks. Any reader that has knowledge in the following topics can safely skip them and continue to our contributions of the next chapters.

3.1 Notation Table

To make things as easy as possible for the reader, we present a notation table which specifies notations for all the basics we need in this thesis.

Notation	**Meaning**
M	Neural network model
P	Activation polytope
x	Input to neural network model
y	Output of neural network model
η	Neuron / Perceptron
$\nu_i^{(l)}$	Output of the i'th neuron of layer l
$w_{ij}^{(l)}$	Weight at the connection of the i'th neuron of layer $l-1$ and the j'th neuron of layer l of neural network model
b	Neural network bias
A_p, b_p	Polytope function weight and bias

3.2 Basics of Neural Networks

The topic this whole thesis revolves around is neural networks. Neural networks can be used to handle many tasks: A simple task like learning how to classify a flower type based on some metrics like, for instance, stem size. Or a much harder task like learning to differentiate human beings by looking at some known set of images of these persons. We will also use the term model, when talking about neural networks.

3.2.1 Data Encoding for Training of a Neural Network

To handle any of the data mentioned above – stem size of a flower or pictures of human beings – neural networks need some form of data encoding. Data on a computer is usually stored in a binary fashion but this is not useful for learning in most cases: A string of 0s and 1s may not convey the characteristics of a colored image of a person as well as its color value magnitudes for red, green and blue color of each pixel. Therefore, choosing a good encoding of input data is crucial to successfully training a neural network on a dataset.

A binary representation can be usefull if the data at hand is actually a measurement of binary features, like whether a given person has shopped specific e-commerce items. For image datasets, usually, individual pixels are encoded with a floating point value between 0 and 1 for each color channel, indicating the magnitude of this color. Text datasets require a more sophisticated approach: To grasp their structure it is often useful to measure the frequency of features like specific words. We will go into more detail on how to encode a dataset in a manner that allows a neural network to actually learn something, when we introduce the datasets used in this thesis in section 7.2.

We interpret inputs and outputs of neural networks as vectors. For example, data obtained from text with 1024 features extracted, will result in 1024 dimensional vectors.

Once a dataset has been preprocessed to be available in a useful encoding, the next question that arises, is how neural networks can learn the data from the dataset.

3.2.2 Training a Neural Network

In the training phase of supervised learning, the neural network is given an assortment of data (training data) which is already labeled correctly. In the case of images of a person, the label of an image could be the person's name. Once training is done, the task given to the neural network would be to output a label which correctly identifies a person when we show an unlabeled image (test data) to the neural network as input.

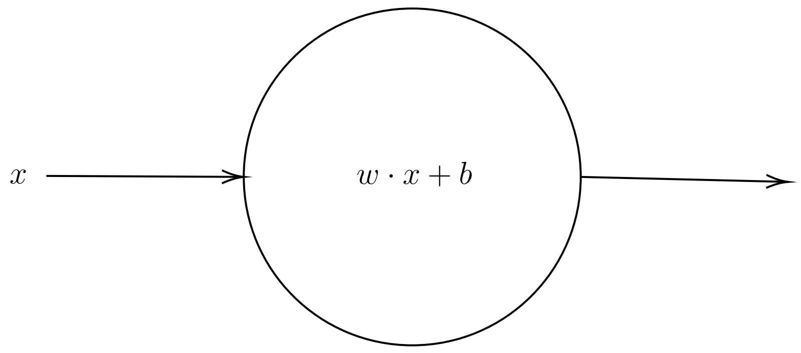

Figure 1: Neuron of a neural network, parameterized with a weight vector w and a scalar bias b.

We need to define what state of the neural network, i.e. its weights and biases, is considered as good with regard to the task at hand. To this end, a loss function calculates an error from the output of the neural network on training data and the correct labels. The output is based on the parameters of the model (weights and biases) whereas the correct label stems from training data.

A possible candidate for a loss function is the MSE loss which measures the element wise mean squared error from the model's output on training data and the correct labels.

To achieve a progression in training, weights and biases of the model should be adjusted in a way, that the loss is decreased. This forms an optimization problem which can be solved with different strategies. We will take a look at one of them in section 3.5, namely gradient descent, since we will need the details of this approach in chapter 6 when we try to optimize towards a heap in the model's parameters' norm or a valley in the output norm and confidence of a neural network.

3.2.3 The Structure of Neural Networks

As a basic building block, neural networks use neurons. Neurons combine numeric inputs, factor them with weights, add a bias and thus produce a numeric output, which can then be fed into another neuron, see figure 1.

The number of neurons needed for a given task, depends on the dimensionality of input data, but also the complexity of the task is a huge factor: The harder the task, the more neurons are usually needed. Multiple neurons are combined in a layer and these layers can be stacked onto each other.

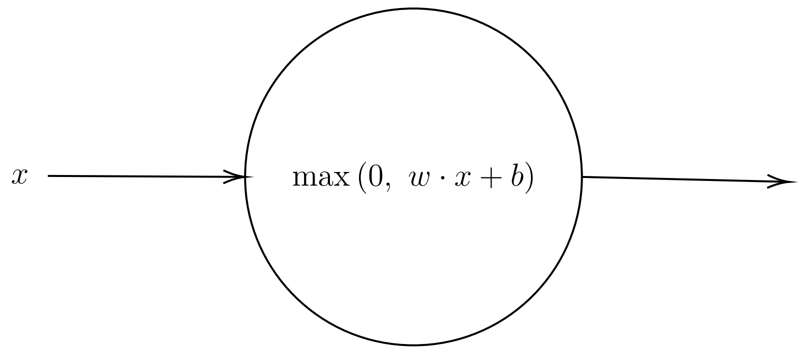

Figure 2: Perceptron of a neural network, parameterized with a weight vector w and a scalar bias b. Here, the $\max$ function computes the element-wise maximum in comparison with 0.

By adjusting the weights to the training data accordingly, a fixed number of neurons can only learn an affine linear relationship between inputs and labels. This is the case, since the function of a single neuron is affine linear and thus the same goes for any combination of neurons.

Learning linear relations is insufficient for many tasks. This is why we need to introduce nonlinearity in the form of an activation function. In this thesis, we focus on the rectified linear unit (ReLU) activation function, as it is widely used and also adheres to our constraints of section 3.3. ReLUs work in such a way, that they take the input and cut of anything that has a value below zero. We define this in a mathematical way:

Definition 3.1
Let $x \in \mathbb{R}^m$ be an arbitrary input to a neural network or any intermediate result. A recitified linear unit is defined as the function $f(x) = (\max(x_i, 0))_{i=1,\dots,n}$.

When extending a neuron with a ReLU, we will call this a perceptron, see figure 2.

We obtain the form of neural networks (see figure 3) we use from here on out. It consists of a number of hidden layers of perceptrons, an input layer which represents input data and an output layer which is a simple linear layer (no ReLU) and produces the output of the neural network, possibly with some alterations like mapping the individual values to the range $[0, 1]$, for instance. The output of a neural network is simply a vector and, in our case, will consist of a certainty value for each class. The bigger the value, the higher the certainty of the model that the input belongs to this class.

In this thesis, we only analyze feed forward neural networks (FNNs), rather than also taking into account convolutional neural networks

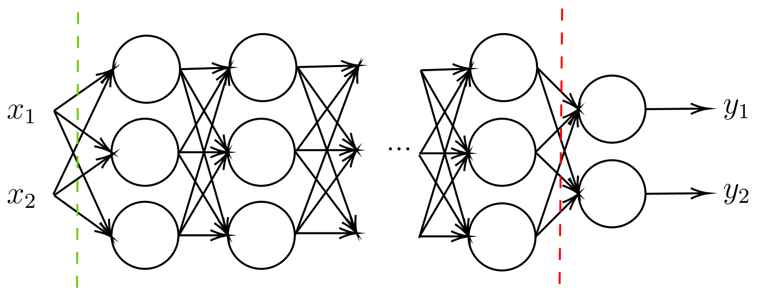

Figure 3: A feed forward neural network (FNN) with a two dimensional input layer (left off the green dashed line), multiple hidden layers with perceptrons and a two dimensional output layer (right off the red dashed line)

(CNNs) which are most often used for image recognition. Putting our focus on one type of target models allows us to go even deeper in analysis and attack development. Still, FNNs are highly relevant as many classification tasks can be performed by FNNs as section 7.2 demonstrates. Interestingly enough, image filters used in CNN that do not use some nonlinear features like pooling layers can easily be represented as sparse weight matrices of a regular FNN.

3.3 Partial Linearity Theorem and Activation Polytopes

An interesting result we need in our reasearch is due to Raghu et al. [8], who stated the following.

Theorem 3.2 (Partial Linearity Theorem)
Let M be neural network. If M uses a piecewise linear activation function, then the input space of M is subdivided into convex polytopes where M behaves piecewise linearly.

An example of this can be seen in figure 4.

The reason for this behavior of neural networks is as follows: A single perceptron in the neural network is either active or not, i.e. its output value is greater than zero or equal to zero. We then define the activation pattern (or simply activation) of the whole network as an indicator function, which for each perceptron indicates whether this perceptron is active or not.

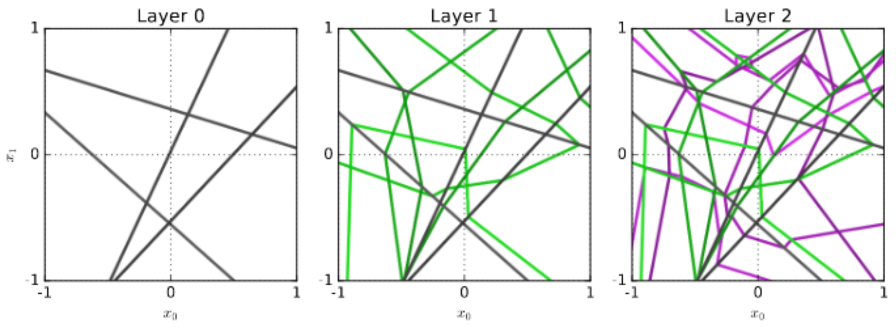

Figure 4: A simple neural network with a ReLU activation function, three layers of perceptrons and two dimensional input. From left to right, more layers of the model over the input space are shown. Each line indicates the activation of a specific neuron: For every point from input space above this line, the neuron is active, for every point below it, the neuron is inactive. Taking all the layers together gives the convex activation polytopes the input space is subdivided into. This image is taken from Raghu et al.[8].

Definition 3.3
Let M be a neural network consisting of a set of perceptrons $N = \{\eta_1, \eta_2, ..., \eta_n\}$ and with inputs $x \in \mathbb{R}^m$. Define the output of perceptron η of M on input x as ν_η. We then define the activation pattern as a function $f : \mathbb{R}^m \times N \to \{0, 1\}$ where $f(x, \eta) = \begin{cases} 1 & \nu_\eta(x) > 0 \\ 0 & \text{else} \end{cases}$.

When given a specific activation, for each perceptron, the ReLU dissolves since we know whether this perceptron is active or not. The output of each perceptron then is either zero or changes linearly for this activation, because the neuron itself consists of an affine linear function with weight and bias.

The form of piecewise linear convex areas is explained in figure 4. Here, we see a model that was trained on two dimensional input. For neurons in the first layer, we can easily draw a line which indicates the activation. Every input above this line activates the neuron, every input below it, does not.

For deeper layers, Raghu et al. use an inductive argument: Since the output of the lower layers is linear in any given polytope, the lines of a deeper layer can only take turns at lines of lower layers, since here, an input to the deeper layer has a break in linearity.

We refer to the polytopes of the Partial Linearity Theorem as activation polytopes, because they are related to a specific activation of the neural network at hand.

We use the results of Raghu et al. and go a lot of steps further: We describe a general way to compute the exact activation polytopes and then improve on this by providing a different approach that is much more time efficient but is still able to compute a good picture of activation polytopes, given a desired resolution. From this, we are able to observe some first indications of an interesting effect of how neural networks learn. Furthermore we use Raghu et al.'s basic results to propose two novel attacks based on our observations.

3.4 Polytope Functions and the Gradient of a Neural Network

The last section concluded, that the input space of a neural network with a piecewise linear activation function is subdivided into convex polytopes. For our attack proposed in chapter 4 and chapter 6 we will need the actual functions of these polytopes. We call any of those functions a polytope function and it is in the form $M_P(x) = A_P \cdot x + b_P$ with weight $A_P \in \mathbb{R}^{l \times m}$, and bias $b_P \in \mathbb{R}^l$ for some polytope P of a given neural network M. How to obtain these functions is discussed in section 4.2.

Note that A_P not only describes the function of M in the current polytope, but is also equivalent to the gradient of M in said polytope: So when we are talking about the function of M in a specific polytope, we are also talking about the gradient in this polytope. This offers somewhat more intuition to the reader, for instance a large gradient will imply a great change in the output of the neural network.

The proof of the following theorem shows the equality between polytope function weight and gradient.

Lemma 3.4
Let $x \in \mathbb{R}^m, A \in \mathbb{R}^{l \times m}, b \in \mathbb{R}^l$ and define $M(x) = Ax + b$. It holds, that $\nabla M(x) = A$, i.e. the gradient of an affine linear matrix function is equal to the matrix itself.

Proof. To get the gradient of our function M, we need to calculate $\frac{\partial M(x)}{\partial x}$. Fortunately, we can split up $M(x) = Ax + b$ into l different functions $a_i x + b_i$ where a_i is the i-th row of A and b_i is the i-th entry of b, thus we get that $M(x) = (a_i x + b_i)_{i=1,2,\ldots,m}$.

Now, the gradient can easily be obtained:

$$\frac{\partial M(x)}{\partial x} = \begin{pmatrix} \frac{\partial\, a_1x+b_1}{\partial x} \\ \frac{\partial\, a_2x+b_2}{\partial x} \\ \vdots \\ \frac{\partial\, a_mx+b_m}{\partial x} \end{pmatrix} = \begin{pmatrix} \frac{\partial\, a_1x+b_1}{\partial x_1} & \frac{\partial\, a_1x+b_1}{\partial x_2} & \cdots & \frac{\partial\, a_1x+b_1}{\partial x_m} \\ \frac{\partial\, a_2x+b_2}{\partial x_1} & \frac{\partial\, a_2x+b_2}{\partial x_2} & \cdots & \frac{\partial\, a_2x+b_2}{\partial x_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial\, a_lx+b_l}{\partial x_1} & \frac{\partial\, a_lx+b_l}{\partial x_2} & \cdots & \frac{\partial\, a_lx+b_l}{\partial x_m} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix},$$

which is, in fact, equal to A. $\square$

3.5 Optimizing a Loss Function with Gradient Descent

When we try to optimize a loss function, this implies that we want to move towards an optimum of this function. Gradient descent does this by calculating the gradient of the loss function with respect to the inputs into the function. Since the gradient indicates the direction of steepest ascent, Gradient Descent takes a step in the opposite direction of the gradient, thus decreasing the loss function value. In every iteration the inputs are adjusted to the values the algorithm stepped to in the last one. Hopefully, with each iteration, the input will get closer to something that induces an optimum in the loss function.

For example, in neural network training, inputs to the loss function are most often dependant on the weights and biases of the model. So, optimizing this loss function results in adjustment of weights and biases of the model to better fit the training data.

When we are interested in increasing the loss function value, we take a similar approach, only that we walk in the direction of gradient when trying to optimize our inputs. We call this gradient ascent.

In both gradient descent and gradient ascent, fixing a good learning rate, i.e. the amount of distance allowed to be traveled in one optimization step and a number of iterations (or epochs) that is large enough is critical for a positive outcome of the algorithm. More parameters include: Momentum, which causes gradients of past iterations to be accumulated instead of just using the gradient of the current epoch. Weight decay draws the focus away from only minimizing the loss function, but also trying to keep the weights small, typically measured by the euclidean norm of the weights. Dampening reduces the learning rate after each iteration, that is, in later epochs, smaller steps are taken.

3.6 Membership Inference Attacks on Neural Networks

In membership inference, an attacker is given white box or black box access to a target model M and is given a datapoint d which is either drawn from the training set of M or not. The attacker then must decide, if d is a member, i.e. belonging to the training set, or a nonmember. The accuracy of such an attack is considered to be the fraction of samples the attacker classifies correctly.

3.7 Gradient Boosted Decision Trees

Friedman [12] introduces gradient boosted decision trees (GBDT). In section 7.4 we show that they are a viable option as an attack model for membership-inference-attacks.

A simple decision tree splits training data at each node by looking at one or more values from the input datapoint, trying to reduce the value of a given loss function. When asked for a prediction on a given input, the input is pushed down through the decision tree and at each node takes a turn into the split, the input belongs to. The leaf node can then be used to obtain the result of the prediction.

In GBDT, multiple stages of decision trees are used. In the first stage, the decision trees will only try to fit their output to the labeled training data as good as possible. Decision trees at a second stage will try to fix their output to the difference of the label and the output of the decision trees of the first stage. At the third stage, new decision trees take the label and the outputs of the first and the second stage into account and so on. Consequently, a new stage only trys to change and improve output on those samples from training data, that are currently being falsely classified, according to the labels of training data.

3.8 Model Extraction and Federated Learning

We discuss potential applications for our observations on the training of neural networks in chapter 5. This includes model extraction and federated learning.

A model extraction attack aims at reconstructing the weights and biases of a neural network. The attacker has some access to the model, e.g. the attacker might be able to query the model to gain some input / output pairs to help with the task. In this attack, one will either be interested in fidelity extraction, meaning that the extracted parameters of the targeted network M should closely match the function that M represents in that it produces almost the same output every time. Or, one will be interested in task accuracy extraction, where the extracted function should be able to solve the same classifaction task that M tries to solve, even though the extracted function might not produce exactly the same predictions as M.

In federated learning, multiple actors with individual datasets try to train a global model on all of these datasets, without ever sharing the datasets. This is often accomplished by a central server that receives updates from the actors and – based on these updates – constructs a model for all.

4 Exploiting Leakage of Activation Polytopes in Neural Networks

We present our first privacy attack on neural networks: This attack exploits a specific part of neural networks that has been neglected in security analysis so far, to the best of our knowledge: The layout of activation polytopes and their behavior in the training of neural networks. The results indicate that there is in fact leakage – but further research is needed to distil an even better, more powerful attack.

4.1 Introduction and Motivation

In neural network training, the model tries to fit its parameters, i.e. its weights and biases to the training data as good as possible. The accuracy on the training data should reach an acceptable level, but furthermore, the accuracy on test data should be satisfying as well. Test data is not known to the model in training, so generalizing to this is a very important but also a very tough task.

Neural networks are prone to overfitting. When this happens, the gap between training accuracy and test accuracy is very large. This is undesirable: On the one hand, a bad performance on test data makes the model rather useless. On the other hand, it opens up many possibilities for attacks. For instance, when a model overfits, its certainty will quite possibly be lower for test data than training data. Even a simple threshold attack can then classify given datapoints based on a threshold value ϑ fairly successfully into members and nonmembers: If the certainty of the target model on a given datapoint d is above ϑ, the attacker would conclude that d is a member, otherwise it would conclude that d is a nonmember.

Infering membership of datapoints is a highly relevant field of research and poses a huge threat to privacy. Medical history, geographical positions or shopping preferences that are reflected in datasets all represent sensitive data. Individuals that contributed to the dataset will most often be concerned that the fact of their contribution alone will be held private by the data aggregator.

As chapter 2 on related work has shown, many attacks already exist in the area of membership inference. One very popular example is the MLLeaks attack by Salem et al. [3] which uses an attack model that learns on the top three values from the certainty of the target model to infer membership. The authors' work resulted in a highly successful attack in a setting where the structure of the target model is known to the attacker, and even in the case where this structure is not known, i.e. a strict black box attack.

It is an open question though, whether the results of Salem et al. could be improved even further when moving towards a white box setting and exploiting the structure as well as parameters of the target model. This is definitely a vastly interesting topic to look at: Better results in membership inference attacks raise awareness for sensible handling of private data and highlight the main vulnerabilities to the privacy research community.

We propose a new membership inference attack which exploits a completely unregarded part of neural networks with great potential for leakage: We try to exploit the structure of activation polytopes of the target model.

As explained by the partial linearity theorem in section 3.3, activation polytopes are piecewise linear convex polytopes that subdivide the input space of any neural network that uses a piecewise linear activation function, like a ReLU. The question is, why should we bother looking at the polytopes? The reason is, that these polytopes are built during training. And since all the model sees in training is the training data, we suspect leakage here. Also, we assume that an attack exploiting this leakage would be even more robust to less overfitting of the target model or defense mechanisms since this leakage stems from the structure of the target model rather than its output. We propose the following hypothesis.

Hypothesis 4.1
The layout and the form of activation polytopes of a neural network with a piecewise linear activation function are both susceptible to leaking information about the training data.

With our attack we try to evaluate the validity of this hypothesis.

4.2 Visualizing Activation Polytopes

We visualize activation polytopes to further support our hypothesis. Before diving into the actual attack, we conduct a number of experiments to evaluate the influence of training data on activation polytopes.

When visualizing the activation polytopes of a neural network, we have to restrain ourselves to the two dimensional case for obvious reasons. In higher dimensions we will use different means to get some insights into a model in later chapters, but for plotting the full activation polytopes we need to draw them on a plane.

Since datasets of two dimensional datapoints are not so relevant in machine learning and thus are very rare, we introduce a new synthetic dataset we call Clusters2D. This dataset is constructed by randomly sampling pairs of cluster centers over the unit square with a uniform distribution and then spreading random datapoints around a cluster center. For each pair of cluster centers, one cluster will contain only members of class 0 and the other will only contain members of class 1. This approach tries to resemble the arrangement of natural datasets of higher dimensions where we often find clusters of datapoints that belong to the same class. But to make the classification task of the target model somewhat harder, we only keep a small gap between neighboring clusters of different classes. This way, the target model is urged to draw fine decision boundaries between the two. An example Clusters2D dataset is shown in figure 5.

We need to obtain a mathematical representation of activation polytopes before we can actually visualize them. For a mathematical representation of a single activation polytope, given by the corresponding activation pattern, we want an equation for each of the adjacent hyperedges. In the two dimensional case, these hyperedges are simple two dimensional lines, in three dimensions, they are planes in the three dimensional space and in higher dimensions they are hyperplanes.

The challenge at hand can now be specified as follows: for a fixed neural network M and given an arbitrary point p, compute the bounding activation polytope P of M for p.

To solve this challenge, firstly, we recall, that for a given activation pattern, we argued in section 3.3 that all ReLUs dissolve. And so, each perceptron of M either contributes a linear function – described by its weights and bias – to the overall function of M or nothing at all, e.g. the zero function.

As a first step, we compute the activation pattern f of M, given p. A detailed description on how to do this, is given in algorithm 1. Again, f

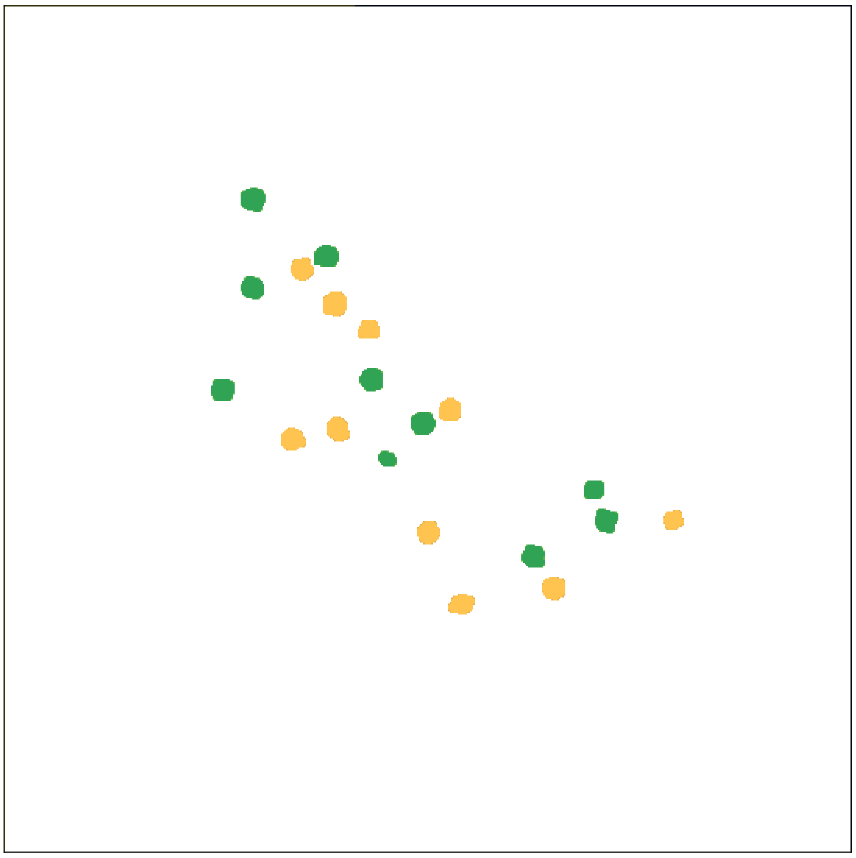

Figure 5: An example of a Clusters2D dataset. The dataset features two classes, shown as green and orange clusters. All datapoints are sampled over the unit square.

is an indicator function, which tells us for each perceptron whether it is active or not.

Algorithm 1: ActivationPattern

```
Input   : model, point
Output: Activation pattern of model for point
set activation to the zero function
for each layer of model do
    output = outputAfterLayer(model, layer, point)
    for each perceptron of layer do
        if output[perceptron] $> 0$ then
            activation(perceptron) $:= 1$
        else
            activation(perceptron) $:= 0$
        end
    end
end
return activation
```

Given that we are able to compute an activation pattern, we want to calculate for each perceptron ν the actual weights and the adapted bias which describe the output of M up until after ν. We call these actual weights and biases the *adapted* weights and biases and denote this with w^* and b^* as opposed to the *basic* parameters w, b of the perceptron, as defined in M.

The adapted parameters of a model inside of some activation polytope P form an affine linear function like $g(x) = w^* \cdot x + b^*$ for any input x. When all the points inside of P that cause g to go from negative values to zero or above are connected, this describes the hyperedge of ν at P. This is true since g is greater than zero if and only if ν is active and so, the hyperedge obtained from g in said way, indicates this change in activation. Note that, some perceptrons might not contribute a hyperedge to P but we will see how to sort these out in just a bit. We give an example of a hyperedge of an activation polytope in figure 6.

For the calculation of adapted weights and biases, we start by taking a coarse grained look at M and disregard any ReLU. The output M_1 of the first layer of M is made up of the basic weight matrix w_1 and the basic bias vector b_1 of the first layer: $M_1(x) = w_1 \cdot x + b_1$. The output of the second layer then is $M_2(x) = w_2 \cdot M_1(x) + b_2 = w_2 \cdot (w_1 \cdot x + b_1) + b_2 = w_2 \cdot w_1 \cdot x + w_2 \cdot b_1 + b_2$. Consequently, the adapted weight matrix of the second layer is simply $w_2^* = w_2 \cdot w_1$, and the adapted bias vector is $b_2^* = w_2 * b_1 + b_2$. This gives us a way to easily compute the adapted parameters of any layer

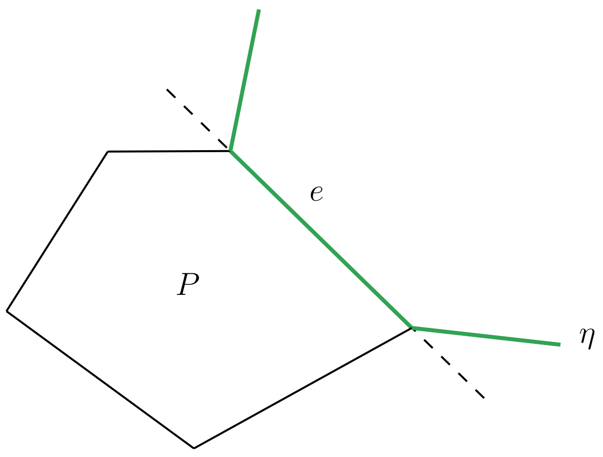

Figure 6: An activation polytope P of some neural network M. The green lines indicate where perceptron η changes from inactive (all points below the line) to active (all points above the line). The edge e of P that is due to η is only a part of that. If η's behavior inside of P is given by $g(x) = w^* \cdot x + b$ for some input x, then e is obtained by connecting all the points inside of P that make g go from negative values to zero and above. When continued outside of P this gives the dotted line.

and any perceptron. Note that, adapted weights and biases of any layer can be constructed independently from one another as we do not need one of the two to calculate the other.

Let us take the ReLUs into account now, as well. Our algorithm to calculate adapted weights steps through M layer by layer: At the first layer, the adapted weights are equal to the basic weights. At the second layer, for neuron ν, we simply construct a matrix of all weights of neurons of the first layer and multiply it by the weight of ν. But for any perceptron of the first layer which is inactive, we discard it's weights, i.e. replace them with the zero vector. The overall result matches the weights of the function which produces output at ν when the input is send through from the beginning of M. With deeper layers we go on analogously: Construct a matrix of adapted weights from the former layer, set all inactive perceptrons' weights to zero and multiply the weight of the current perceptron to obtain the adapted weight of this perceptron. Pseudocode for this approach is given in algorithm 2.

The adapted bias of any layer is generated in a similar fashion, as can be seen in algorithm 3. Again, at the first layer, the adapted bias is equal to the basic bias. At the second layer we multiply the weight matrix (with discarded weights of inactive perceptrons) of the second layer by the bias vector of the first layer and add the bias vector of the second layer. Deeper layers are handled in an analogous fashion.

Algorithm 2: AdaptedWeights

```
Input   : model, point, myPerceptron
Output: Adapted weight of myPerceptron at activation polytope of
        model at point
set myPerceptronLayer to the layer of myPerceptron in model
set adaptedWeights to the identity matrix
activation = ActivationPattern(model, point)
for each layer before myPerceptronLayer do
    layerWeights = LayerWeightMatrix(model, layer)
    for each perceptron of layer do
        if activation(perceptron) = 0 then
            layerWeights(perceptron) = $(0, 0, ..., 0)$
        end
    end
    adaptedWeights = layerWeights · adaptedWeights
end
adaptedWeights = PerceptronWeightVector(model,
 myPerceptron) · adaptedWeights
return adaptedWeights
```

Algorithm 3: AdaptedBias

```
Input   : model, point, myPerceptron
Output: Adapted bias of myPerceptron at activation polytope of
        model at point
set myPerceptronLayer to the layer of myPerceptron in model
adaptedBias = LayerBiasVector(model, first layer)
activation = ActivationPattern(model, point)
for each layer after first layer up to myPerceptronLayer do
    layerWeights = LayerWeightsMatrix(model, layer)
    layerBias = LayerBiasVector(model, layer)
    for each perceptron of layer do
        if activation(perceptron) = 0 then
            layerWeights(perceptron) = $(0, 0, ..., 0)$
        end
    end
    adaptedBias = layerWeights · adaptedBias + layerBias
end
return adaptedBias(myPerceptron)
```

It remains to show, how to get the exact bounding polytope for an activation, once the hyperedge equations have all been set up. We already mentioned, that some perceptrons might not even contribute a hyperedge to the polytope at all. To sort these out and get a simple point wise representation of the polytope we do the following. Firstly, we construct a series of constraints that bound our activation polytope: For any perceptron, from the activation pattern we either get that any point of the polytope must lie above or below this perceptron's hyperedge. For our constraints, this results in the hyperedge equation of this perceptron with a greater than or a less than sign in between. Now, again for any single perceptron and its hyperedge, we construct an optimization problem which tries to get any point which lies on the given hyperedge but with respect to all the constraints just built. By doing this optimization twice with a slightly different target function, we get two points on the hyperedge and still inside the activation polytope. These two points are sufficient to describe the hyperedge's contribution to the activation polytope in the two dimensional case. Also, if this perceptron's hyperedge is not adjacent to the current activation polytope, we will simply not get any solution to the optimization problem and we can discard this perceptron. The procedure described here is depicted in detail in algorithm 4. The function `Solve` is considered to be an optimizer of some sort: It takes an equation for which it must propose a solution, keeping some constraints in mind and trying to maximize or minimize to a specific goal. See section 4.7 for details on the solver we used.

We are now able to construct any hyperedge belonging to an activation polytope at a specific point in the input space. Applying this result to our Clusters2D dataset, we obtain such pictures as in figure 7.

There are many things we are able to observe here. The density of the polytopes seems to be higher right between two clusters of different classes, whereas with a bit of distance to training data, the density seems to drop. Furthermore, many perceptrons form hyperedges which are all in parallel in between the two clusters. The overall appearance of polytopes seems more structured nearby training data, while they look less structured and less homogenous when going further away from training data, at least in most directions.

From these experiments we derive 4.1 which states that the layout and the form of activation polytopes are susceptible to leaking information about the training data. Furthermore, we suspect density and parallelism of activation polytopes to be even more concentrated around training data on higher dimensional data: Here, the neural network still has to focus its effort on separating clusters and classifying training data but anywhere where there is not much to be done, i.e. where there is no training data,

Algorithm 4: ActivationPolytope

Input : model, point

Output: Boundary points of activation polytope of model at point

```
 1 activation = ActivationPattern(model, point)
 2 constraints = ()
 3 polytope = ()
 4 for each perceptron of model do
 5     adaptedWeights = AdaptedWeights(model, point, perceptron)
 6     adaptedBias = AdaptedBias(model, point, perceptron)
 7     if activation(perceptron) = 1 then
 8         Append(constraints, adaptedWeights · $x$ + adaptedBias $\geq 0$)
 9     else
10         Append(constraints, adaptedWeights · $x$ + adaptedBias $\leq 0$)
11     end
12 end
13 for each perceptron of model do
14     adaptedWeights = AdaptedWeights(model, point, perceptron)
15     adaptedBias = AdaptedBias(model, point, perceptron)
16     solution = Solve(adaptedWeights · $x$ + adaptedBias $= 0$,
         constraints, maximize $x_1 + x_2$)
17     if solution $\neq \perp$ then
18         Append(polytope, solution)
19         solution = Solve(adaptedWeights · $x$ + adaptedBias $= 0$,
             constraints, minimize $x_1 + x_2$)
20         Append(polytope, solution)
21     end
22 end
23 return polytope
```

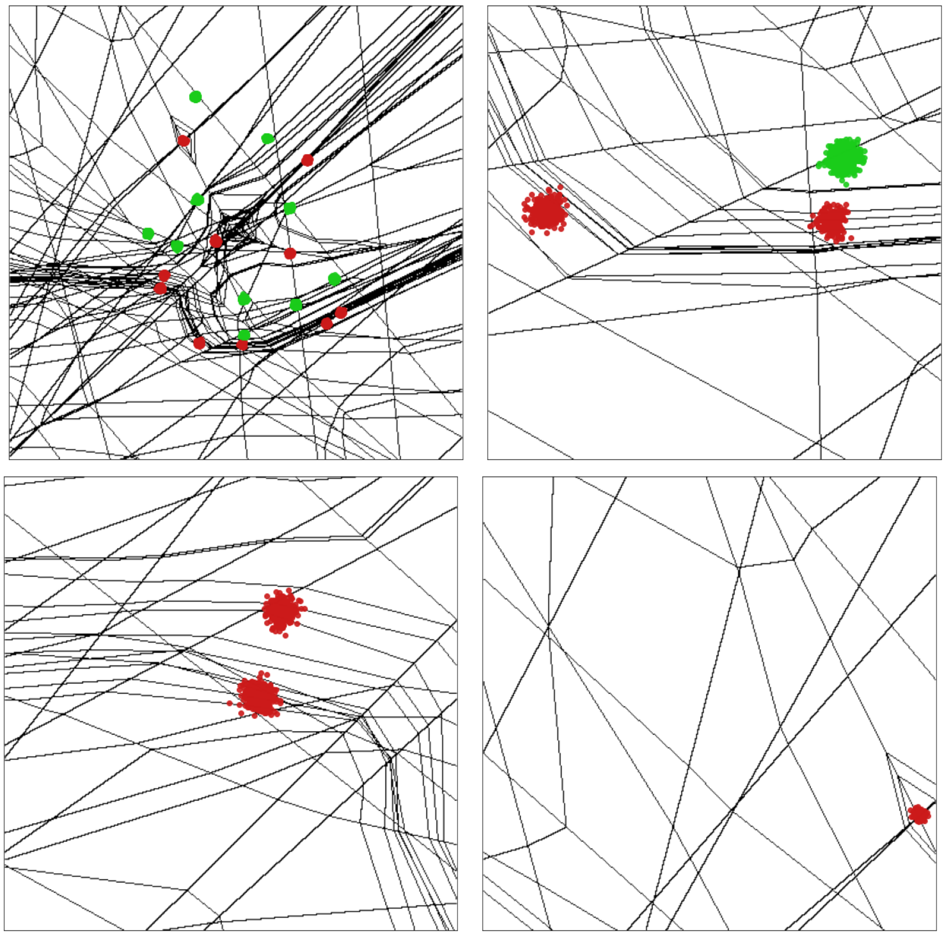

Figure 7: Activation polytopes of a neural network with 4 layers of 25 perceptrons per layer. The model has been trained on a random Clusters2D dataset which is indicated by the green and red clusters of members. From top left to bottom right the pictures show an overview and three detailed views. The last picture was taken at the top left corner of the first picture. As can be seen, activation polytopes tend to be more dense and feature hyperedges that are more parallel nearby training data. When moving away from training data, the activation polytopes become less structured and less dense.

the artifacts which we can observe in the two dimensional case might just vanish since they spread out into a lot of dimensions except for just two.

To summarize, we get two assumptions that are a requirement for hypothesis 4.1, as follows.

Assumption 4.2
High dimensional data behaves in a similar fashion as Clusters2D, in that it forms clusters of data points from the same class and produces a lot of empty space between clusters or pairs of clusters that are from different classes.

Assumption 4.3
Additionally, the structure of activation polytopes observed on Clusters2D is similar on high dimensional data as well, but vanishes when more distant to training data.

4.3 Obtaining Attack Training Data in a High Dimensional Setting

All datasets evaluated in the experimental chapter in section 7.2 consist of data with the number of dimensions ranging from 600 up to a couple of thousands. Our attack needs to measure metrics in the surroundings of a given point (applicable metrics are discussed in section 4.4) to mimic the creation of a detailed view around a given point like in figure 7. Such a detailed view can then be used to infer membership, i.e. separate views into those that look like the surrounding of a member and those that do not. Any attempt to do this, must consider that sampling over a certain radius around a given point is impracticable because of high dimensionality. Furthermore we can not simply compute all of the activation polytopes: A target model with n neurons can have up to 2^n activation polytopes over training data. The reason for this is that we have a distinct activation polytope for each activation and since there are n neurons with two different states – either active or inactive – we have an exponential amount of different activations, thus an exponential amount of different activation polytopes.

To cope with thess problems, we introduce a ray shooting algorithm which tries to get the most out of the surrounding structure around a given point. This algorithm shoots rays from the given point for which membership shall be inferred. Once a first activation polytope boundary is crossed, the exact position of the crossing point is finetuned by taking a binary search like approach walking backward with a smaller step size and changing direction every time the boundary is crossed again. The ray's initial direction is chosen uniformly at random for every dimension. After the first boundary is hit and for every subsequent boundary, the ray continues in the direction of the normal standing on the closest hyperedge of the activation polytope the ray currently is in. The algorithm moves forward in the direction which takes it further away from the initial starting point. Taking the normal of the closest hyperedge ensures that the algorithm does not simply walk randomly but somewhat normalizes the way distances between boundaries are measured. Algorithm 5 shows these thoughts in pseudocode. Values chosen for the number of boundaries computed and the number of steps taken to determine a boundary precisely, both influence running time and precision of the algorithm. Larger values inflict a greater runtime but also greater detail in the obtained boundaries.

There are a few things to note about the functions used in algorithm 5. `ClosestPerceptron` is supposed to get the perceptron which is closest to a given point, which means that it should find the perceptron corresponding to the hyperedge the point is closest to in the current activation polytope. This is fairly easy, because this perceptron will be right at the verge of flipping to activated / deactivated. Therefore, the output of this perceptron will have the smallest absolute value over all perceptrons in the target model, when sending the point through the model.

Once the closest perceptron has been determined, calculating the normal of this perceptron's hyperedge is something already covered by the calculation of adapted weights: The hyperedge's equation is made up of $w^* \cdot x + b^*$, so this already is an edge representation with normal w^* and displacement b^*.

With the ray shooting algorithm, we introduced a way to perform measurements around a given point without needing to go through all the dimensions. In the next section we discuss what could possibly be measured in those surroundings.

4.4 Metrics for an Activation Polytopes Attack

We define what parts of activation polytopes we want to analyze for possible leakage. The following metrics are measured for a given datapoint, to generate data for an attack model.

4.4.1 Boundary Distance

The first metric is called the boundary distance metric. It simply looks at the distances from the starting point to any point along the ray that sits at

Algorithm 5: HyperedgeRayShooting

Input : model, point, numBoundaries, numSteps

Output: Points at Boundaries encountered

```
for $i \in \{1, 2, ..., \text{numDims}\}$ do
    directionVec$_i$ $= \in_{\mathcal{R}} (-1, 1, 0, 0)$
end
directionVec = Normalize(directionVec)
lastClass = PredictClass(model, point)
direction = 1
boundaryPoints = ()
for boundaryIndex $\in \{1, 2, ..., \text{numBoundaries}\}$ do
    stepWidth = 1
    finetuning = False
    for $\in \{1, 2, ..., \text{numSteps}\}$ do
        point = point + direction · stepWidth · directionVec
        newClass = PredictClass(model, point)
        if newClass $\neq$ lastClass then
            direction = -direction
            finetuning = True
            lastClass = newClass
            boundaryPoint = point
        end
        if finetuning then
            stepWidth = stepWidth/2
        end
    end
    Append(boundaryPoints, boundaryPoint)
    perceptron = ClosestPerceptron(model, boundaryPoint)
    directionVec = AdaptedWeight(model, perceptron)
    directionVec = Normalize(directionVec)
    set direction to whatever takes us further away from the initial point
end
return boundaryPoints
```

a boundary. By doing this, it expresses the density of activation polytopes around a given point: The higher the density, the more likely the gaps between boundaries encountered from the ray shooting algorithm will be small.

4.4.2 Activation Polytope Derivative

Another metric we look at, takes all the activation polytopes discovered and computes the target model's respective gradients at these polytopes. Keep in mind, that this gradient is equal to the weight of this polytope's function. This has potential leakage: Since the polytopes arrange themselves according to training data, the polytope functions' parameters quite possibly leak some information about training data. Their computation has already been covered in section 3.4. The gradient of the neural network is a matrix, so applying the Frobenius norm – which is an extension of the Euclidean norm to matrices – gives a scalar value which is the output of this metric.

4.4.3 Normal Cosine Similarity

The last metric looks at all the hyperedges of the boundaries encountered. Since, in the two dimesional case, we noticed more parallelism around training data, this metric computes the cosine similarity of subsequent boundaries' normalized normals. The cosine similarity is a measure of similarity between two vectors and shows off where there is high parallelism.

4.5 The Activation Polytopes Attack

We introduce the activation polytopes attack. This attack takes a similar approach as Salem et al. [3] did in their MLLeaks attack: The attack model is trained on data obtained from a priorly constructed shadow model and the actual attack is then carried out on the target model. Recall, that the shadow model is used so that the attacker has labeled training data, since queries to the target model with a given datapoint do not safely reflect whether this datapoint is a member or not.

Our approach differs from Salem et al. in that we try to exploit the form of activation polytopes of the target model instead of simply looking at the confidence which is output by the target model.

We concentrate on the first and simplest but nonetheless realistic and interesting adversary as defined in the work of Salem et al., where the shadow model uses the same architecture as the target model and is trained on data from the same distribution as the data of the target model.

Once both the shadow model and the target model have been trained, our attack takes batches of the shadow model's training data and test data, i.e. members and nonmembers, and for each of these batches' points performs the hyperedge ray shooting from algorithm 5 on the shadow model. At the boundaries discovered by the algorithm, the attacker then applies the metrics discussed in section 4.4. Algorithm 6 depicts this training phase of our attack.

Algorithm 6: Activation Polytopes Attack – Training Phase

```
Input   : shadowModel, shadowTrainingData, shadowTestData
Output: attackModel
attackTrainingData = ()
for point ∈ shadowTrainingData do
    boundaries = hyperedgeRayShooting(shadowModel, point,
      numBoundaries, numSteps)
    for boundary ∈ boundaries do
        apply chosen metric to this boundary
    end
    append(attackTrainingData, (boundaries, 1))
end
for point ∈ shadowTestData do
    boundaries = hyperedgeRayShooting(shadowModel, point,
      numBoundaries, numSteps)
    for boundary ∈ boundaries do
        apply chosen metric to this boundary
    end
    append(attackTrainingData, (boundaries, 0))
end
train attackModel on attackTrainingData
return attackModel
```

The attack model used in the algorithm can basically be any kind of classifier which predicts members and nonmembers with a high certainty based on the outputs of the hyperedge ray shooting algorithm in combination with our metrics. Like Salem et al., we use a feed forward neural network to keep up the comparability to their work. We also evaluate the performance of gradient boosted decision trees, since they seem well suited for this classification task, but have not been investigated in such a manner anywhere before, to the best of our knowledge.

When running the attack on the target model with a given datapoint, this datapoint goes through the same procedure as the training data for our attack did with the shadow model: The hyperedge ray shooting algorithm is performed on the target model with the datapoint and then sent to the attack model for a classification into members and nonmembers. Algorithm 7 summarizes this.

Algorithm 7: Activation Polytopes Attack – Attack Phase

```
Input  : attackModel, targetModel, point
Output: Membership classification
1 boundaries = hyperedgeRayShooting(targetModel, point,
    numBoundaries, numSteps)
2 for boundary ∈ boundaries do
3     apply chosen metric to this boundary
4 end
5 return predictMembership(attackModel, boundaries)
```

4.6 Limitations of the Activation Polytopes Approach

There are some limitations of our first approach for the reader to note. The first limitation which we already mentioned, is the fact that there is an exponential number of activation polytopes. So any visualization technique – even in the two dimensional space – has a hard time at getting a complete representation of a given neural network in an efficient way. Time consuming visualization hinders the research process in that it takes too long to try out different specific strategies for visualization like color schemes or line strengths, and it takes too long to visualize a greater amount of different target models to validate any observations made on a single target model once. We overcome this problem in the next chapter.

As our experiments in section 7.3 indicate, there is in fact leakage through our approach, but less strong as we would have hoped. We discuss Cover's theorem [1] [2] as a reason for this. In chapter 6 we introduce a different attack with a more promising outlook.

4.7 Implementation Details

When implementing our activation polytopes attack, there are a few things to note.

Firstly, when using PyTorch – as we did – it is important to turn off gradient computation when running the training phase or the attack phase of the algorithm. PyTorch usually automatically constructs a tree from all executed operations for later gradient computation. So when a lot of batches are processed to generate the attack's training data or to run the attack itself, PyTorch will always keep track of what is computed and in what order. However, this is completely unnecessary, since the activation polytopes attack does not optimize in any way over the input data or the models used, except for the attack model, which is trained in a follow up step, as can be seen in algorithm 6. Disabling PyTorch gradient computation through the command `pytorch.no_grad()` not only erases this computational overhead and therefore saves time, but also heavily decreases memory usage: We noticed large drops in GPU utilization as well as occupied memory space.

Implementing the hyperedge ray shooting of algorithm 5 requires some care as to when a new boundary is discovered. If the first boundary is crossed and finetuned, the algorithm, in its last step for this boundary, might go back over the boundary again, closer to the starting point. When going for the next boundary, the algorithm definitely should not consider the same boundary again. An implementation should rather keep track of the last point which was actually over the boundary when looking from the starting point. From this point the algorithm should continue when moving to the next boundary.

In section 4.2 we argue, that a solver of some sort is needed to compute activation polytopes with algorithm 4. This solver needs to do some optimization with potentially many constraints, as their amount grows linearly in the size of the target model. Therefore, some thought should be put in to what optimizer to use here. We decided to utilize Gurobi [13] which is an industry standard tool and has proven to be highly efficient in comparison to other software suites. Gurobi takes a fairly easy approach to optimization, as there is little setup required and adding constraints and equations follows a simple and intuitive schema as can be seen in their documentation.

5 New Observations on the Inner Workings of a Neural Network

As preparation for our next attack and similar to the visualization of activation polytopes in chapter 4, we spend a lot of time trying to deeply understand what is happening inside of the targeted neural network. More specifically, we want to move on from the computationally expensive calculation of individual activation polytope hyperedges to the visualization of polytope functions. As mentioned in the last chapter, the exponential amount of activation polytopes is one main problem for visualization and attack development. In this chapter, we move past this problem and use our newly developed visualization tool to gain new insights on the inner workings of a neural network.

5.1 Depicting the Influence of Training Data in Neural Networks

We generate images that for each point on a two dimensional grid plot the polytope function's parameters' norms (i.e. the norm of its weight and bias) of the target model at this specific point, see algorithm 8. To compare, we also plot the norm of the output as well as the confidence of the target model at this point. The target model in this scenario has been trained on our synthetic dataset Clusters2D so that we are actually able to see something meaningful in 2D.

In algorithm 8 the function `PolytopeFunction` is assumed to return the adapted weights and the adapted bias of the polytope function of the given model at the given point. This function simply acts as an interface to algorithms 2 and 3 where weights and bias are computed respectively.

By creating these images, we are hoping to understand how output and polytope function of the target model are influenced by the position of training data. On the one hand, this could help us in understanding how

Algorithm 8: VisualizationSampling

```
Input  : model, areaOfInterest, resolution
Output: Image of model over areaOfInterest
1 Generate a list of points that are spread equidistantly in areaOfInterest
    according to resolution
2 image = EmptyImage()
3 for point ∈ list do
4     outputnorm = $||\text{model}(\text{point})||$
5     A, b = PolytopeFunction(model, point) Anorm = $||A||$
6     bnorm = $||b||$
7     Paint pixel in image at point with desired value out of the three
        above
8 end
9 return image
```

neural networks learn the classification task at hand, which is an utterly relevant field of research and has not been understood so good so far. On the other hand, we want to figure out what is leaked from both of these parts of a neural network, possibly to be used in an attack.

One example of the aforementioned plots can be seen in figure 8. We are able to observe first indications of an interesting effect: The neural network seems to build some sort of mountain ridge of large polytope function weights and bias to separate training data of different classes. The bias also tends to carve small norms into plains of large norms. Other than that, there is less activity of the model when moving away from training data. The output norm of the model behaves inversely, in that it features narrow valleys of small output vectors where the separation of different classes is done. Both of these lines – the valleys of the output norm and the mountain ridges of the polytope function's weight norm – correspond to the decision boundaries.

5.2 Conclusions for Leakage and Neural Network Understanding

We draw two conclusions from these observations: First off, neural networks use these valleys and mountain ridges two separate training data and therefore this effect has a high risk of leakage of sensible information about training data. And secondly, this effect could give a precise description of a neural network and what it has learned, while still being way more dense, i.e. with less overhead than, for instance, activation polytopes.

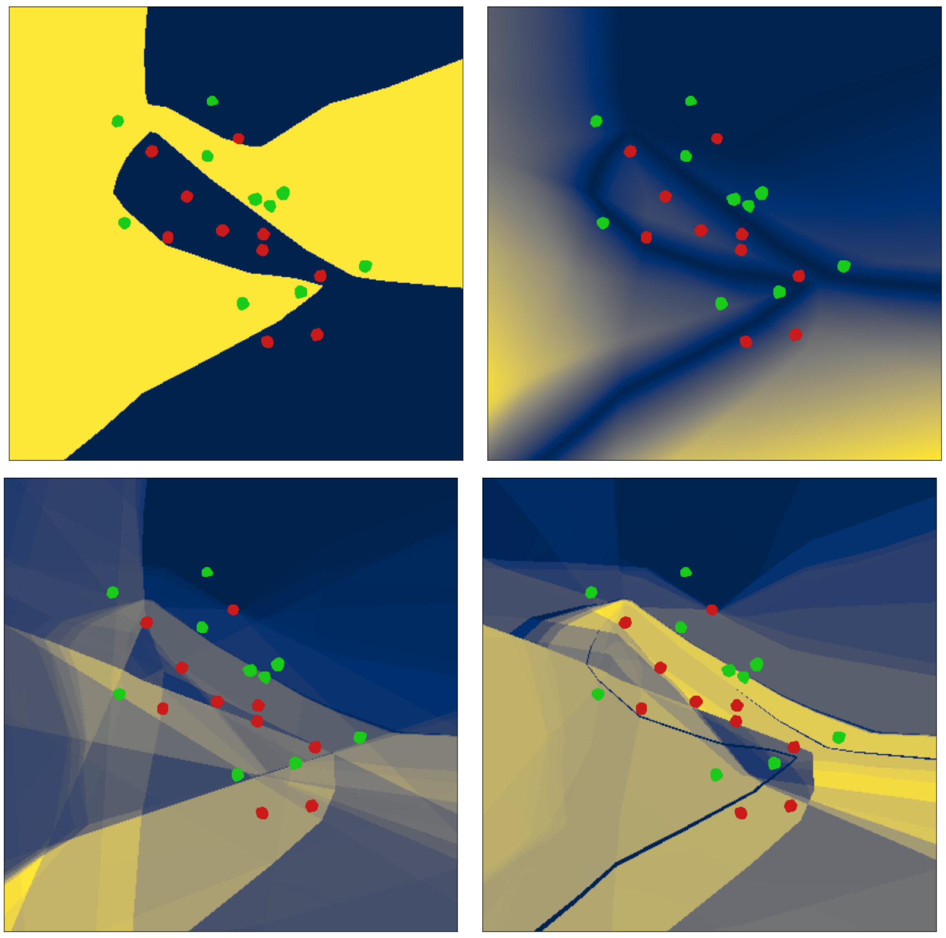

Figure 8: Different plots on a two dimensional grid of a neural network that has been trained on a random Clusters2D dataset. Shown from top left to bottom right are decision boundaries, output norm, polytope function weight norm and polytope function bias norm. Green and red dots represent clusters of training data of different classes. Yellow areas indicate large values, blue areas indicate small values. The examined models seem to build a sort of mountain ridge of using polytope function parameters to seperate training data. The output norm behaves inversely. Both effects – mountain ridges and valleys – correspond to the decision boundaries.

This could be used in membership inference or model inversion attacks and could even be used to make model extraction more efficient: If there is a less complex description of a neural network, we could try to extract this instead of trying to reconstruct all the hyperplanes making up the activation polytopes which quite possibly has a far greater query complexity. Another example where our observations could be applied, is the field of federated learning. Here, exchange of local mountain ridges / valleys with other actors provides a great way of sharing what has been learned so far but possibly reducing the amount of data that has to be transmitted, thus making the overall approach more efficient.

Comparing to the paper by Raghu et al. [8] who originally talked about activation polytopes, our approach described here also gives a great opportunity to visualize activation polytopes without having to compute the exponential lot of them. Our approach allows researchers to pick any desired resolution and any desired spot in the neural network (at least in the two dimensional case) and the visualization computes the surrounding polytope functions but still is much more time efficient. The images of figure 7 that was shown in the last chapter were created using the method described here: For each pixel, the activation of the model is computed. For two adjacent pixel it is then checked if their activation differs. If so, we fill in one of the pixels and in the end, we get all the activation polytopes at the current level of resolution.

5.3 An Animated Visualization of the Training Process of Neural Networks

The observations and implications we have gathered so far become even more apparent, when we create animated views of the mountain ridges over time: Every five batches of every epoch we take a snapshot just like the ones in figure 8 and append these to a GIF to get a glance at what training of a neural network looks like in motion. The reader can take a look at figure 9 for this, the discussion of those images takes place in the next section.

5.4 Further Observations and Mathematical Explanations

On the newly obtained animated view of a neural network (see figure 9) we make another completely new observation: The neural network's hyperedges all seem to work on the same task in the beginning, we call this coalescing neurons. The reader can see this as a big mountain ridge that goes through the middle and takes care of a very basic split of training data in the second image (epoch 1) of figure 9. Then, after some time,

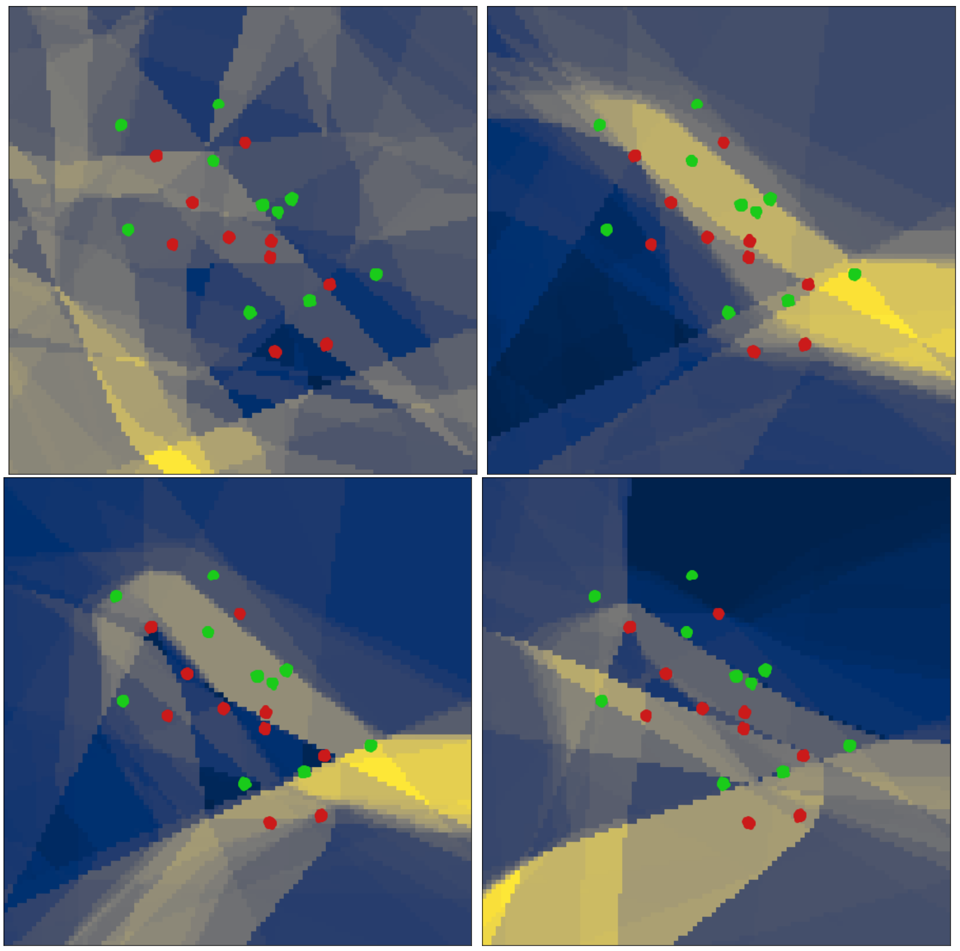

Figure 9: Progression of polytope function weight norm of a neural network that has been trained on a random Clusters2D dataset. Yellow areas indicate large values, blue areas indicate small values. Green and red dots represent clusters of training data of different classes. From top left to bottom right the state of the model is shown in pretraining, epoch 1, epoch 2 and epoch 9. We observe that the examined models start by creating a single mountain ridge of large polytope function weights and bias that generates an initial split of training data. After some time, this mountain ridge either splits up, or new ones rise up somewhere. This process keeps on going until there are ridges in all useful spots.

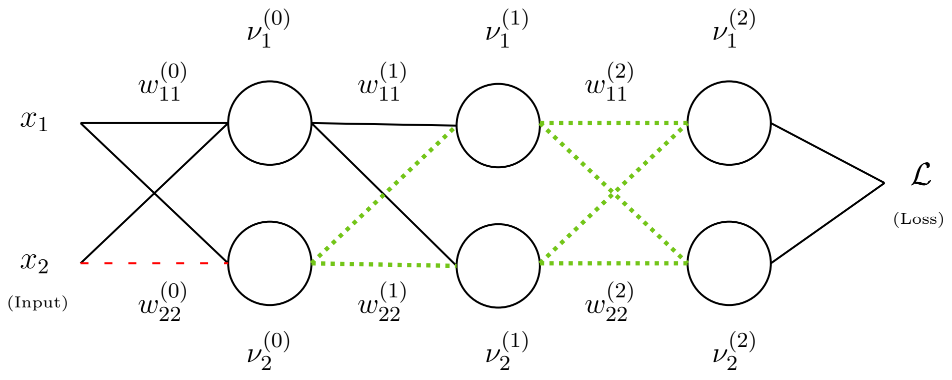

Figure 10: Explanation of backward weight dependency on an example neural network. Weights of layer l are denoted by $w_{ij}^{(l)}$, whereas $(\nu_1^{(l)}, \nu_2^{(l)})$ describes the output of the model after layer l. When computing the weight update in stochastic gradient descent for weight $w_{22}^{(0)}$ (red dashed line), all the weights that can be reached from this weight until the end of the model (green dotted lines) influence the update. Combined with forward weight dependency (neurons from deeper layers need neurons from upper layers to bend and perform hyperedge breaks), this gives a probable reason why neurons of upper layers and neurons of deeper layers are so intertwined in the beginning of training the model.

in our observations, this mountain ridge either splits up into another mountain ridge, or another mountain ridge rises up somewhere. Over the course of multiple epochs, this process keeps on going until there are ridges in all useful spots.

We hold two influences responsible for this observed depedency between weights of deeper layers and upper layers. The first one is the forward weight dependency of the neural network: When hyperedges of neurons from deeper layers want to bend at some point, they need to have hyperedges of upper layers to perform a break.

The second one is the backward weight dependency: To explain this, we have a look at the example neural network of figure 10. This model features three layers with two neurons each and two dimensional input (x_1, x_2). The weights of layer l are denoted by $w_{ij}^{(l)}$ and the output of layer l is referred to as $(\nu_1^{(l)}, \nu_2^{(l)})$. The output of the last layer is fed into the loss function $\mathcal{L}$.

We calculate the gradient $\frac{\partial \mathcal{L}(x)}{\partial w_{22}^{(0)}}$ which is used in stochastic gradient descent to update the weight $w_{22}^{(0)}$. The output of the neural network after layer l is denoted as $\nu^{(l)}$. The function $\Phi^{(l)\prime}$ is the derivative of the ReLU function over the output of layer l.

$$\frac{\partial \mathcal{L}(x)}{\partial w_{22}^{(0)}} = \sum_i 2(\mathrm{NN}(x)_i - y_i)\ \sum_j \Phi^{(2)\prime}\ \underline{\sum_k w_{kj}^{(2)}}\ \Phi^{(1)\prime}\ \underline{\sum_l w_{l2}^{(1)}}\ \Phi^{(0)\prime}\ \nu_2^{(0)}$$

The green and dotty underlined terms are the weights which are green and dotted in figure 10. Weight $w_{22}^{(0)}$ is updated dependant on all the weights that can be reached when going from $w_{22}^{(0)}$ to the end of the neural network. This is true in general as well: Weights of upper layers are updated depending on all of the weights from deeper layers.

Combining forward weight dependency and backward weight dependency, this gives a probable reason why neurons from deeper layers and neurons from upper layers are so intertwined at the beginning.

Through our visualization we notice another effect: Hyperedges associated with neurons of deeper layers move quicker than those associated with neurons of upper layers. This is due to the fact that in SGD, updates of weights of upper layers become dependant on more and more factors (as can be seen in the gradient that has been computed in this section). When these factors are small (for examples the derivatives Φ' in said formulas), the overall gradient tends to diminish and therefore the update for the weight becomes rather small.

It is outside the scope of this master thesis to dive even deeper into this topic or to even use it for model extraction or federated learning. Hopefully, our findings and conclusions will be of use to other researchers in the future. Most certainly, they will be of use to us in the next chapter.

6 A Gradient Descent Based Boundary Distance Attack

Our next attack combines any insights we have made so far or are made in recent papers. We form a new white box attack which uses gradient descent / ascent to optimize towards low confidence, small output and large polytope function parameters of the target model to determine the distance to the closest decision boundary of said model. Then, a threshold is used to compare the distance and to infer membership. Our approach differs from recent approaches in that it uses gradient descent as an optimization tool and furthermore exploits not only the output of the target model but its structure (given by its polytope functions' parameters) as well. In the best case, this attack is able to perform on par with the latest research results on black box membership inference attacks by Choquette et al. [9] on some datasets.

6.1 Introduction and Motivation

Our activation polytopes attack tries to get a good picture of the structure of the target model around a given datapoint. From this, it tries to infer whether the datapoint is a member or not. When we evaluate the performance of this attack in numerous experiments in section 7.3 we conclude, that for the most part the leakage extracted is not as strong as we had hoped. However, one metric does induce a reasonable attack performance: The decision boundary metric, which looks at the distance from a given datapoint to the nearby activation polytope hyperedges. This is not so surprising: Decision boundaries for membership inference have been researched far and wide in recent papers, take the work by Choquette et al. [9] for instance.

Our first approach includes the ray shooting from algorithm 5. Coming from an analysis on two dimensional synthetic data, we were hoping that this ray shooting captures the essence of what makes a member a member by simply doing the same thing we did in 2D: Taking a kind of 'picture' around a given datapoint. Sadly, this approach is not able to get results as good as results from current research. We would like to improve on that.

We craft a new attack which is able to infer membership better through a more targeted approach than our ray shooting and also through using boundary distance as a metric.

6.2 Using Gradient Descent To Find The Closest Boundary

It would be possible to just use our ray shooting approach once again to find the nearest boundary. In chapter 4 the task for this algorithm was to take a kind of 'picture' of the surroundings of a given datapoint. Now, this task changes as we do not care about the overall surroundings but we need to find a close to optimal path from the datapoint which leads to the closest boundary. Again, in a high dimensional setting this is anything but trivial, since there are countless possible paths to choose from, and picking a random one is not likely to give very good results.

To manage this solution space complexity, we employ a standard algorithm in optimization, the gradient descent algorithm. This will be similar to the work by Zhang et al. [6] who used this algorithm for model inversion on CNNs trained on image datasets. We use gradient descent for membership inference instead and evaluate FNNs and associated datasets.

Gradient descent is an iterative algorithm that starts at some point x and progresses into the negative direction of the gradient of some loss function which describes how valuable x is to us, resulting in a change of point and a decrease in value. Gradient ascent takes the opposite direction, therefore it tries to increase the value. We refer the reader to section 3.5, which describes gradient descent / ascent in more detail.

6.3 Putting Our Observations On The Inner Workings Of Neural Networks To Use

We already identified the parameters of polytope functions and the output of the neural network as two indicators for leakage in chapter 5. In the following, we derive three possible parts of the neural network that leak information about training data when looking at some activation polytope P with polytope function $M_P(x) = A_P \cdot x + b_P$. Here, softmax defines the softmax function that maps the output values of the target model to the range [0,1].

Confidence
$\|\text{softmax}(A_P \cdot x + b_P)\|_\infty$

Output
$\|A_P \cdot x + b_P\|_2$

Polytope Function
$\|A_P\|_2, \|b_P\|_\infty$

Research up to this point has shown its main interest in leakage through confidence of the target model, this applies to the paper by Zhang et al. [6] as well. Output is a bit broader as it looks at the whole output rather than just the most significant value. Leakage through polytope functions is a completely new approach. All of the three leakage options are very similar, yet it is the fine differences that result in the manifold observations we make in experiments in section 7.5.

For leakage through polytope functions, we use the Frobeniusnorm $\|.\|_2$ for the polytope function weight and the max norm $\|.\|_\infty$ for the polytope function bias. In our experiments we find these to be the most helpful in running our attack (see overall results in section 7.5).

In our attack, when given a datapoint on which membership should be inferred, we use gradient ascent / descent to optimize towards large polytope function parameters, small output or low confidence of the target model respectively and move the datapoint accordingly. This way, we move closer to the valleys and mountain ridges we have seen in chapter 5 and we are able to compare both strategies.

Our approach differs distinctly from Zhang et al. [6]: Firstly, we optimize towards low confidence since we start at a possible training data point and try to find a nearby decision boundary. Zhang et al. optimize towards high confidence of the target model, since they start somewhere random and try to find a training data point. And secondly, we also want to evaluate whether the polytope function of the target model can be used to leak even more than the output or confidence of the model can.

Other than that, we use another approach similar to the finetuning we did in algorithm 5: When a boundary is crossed, we finetune on this boundary by going back and forth with a smaller step size.

The loss function for gradient ascent / descent in this use case consists of either the norm of a polytope function's parameter at the datapoint, the norm of the output, or the confidence of the target model.

Algorithm 9 provides some pseudocode for our approach to use gradient ascent / descent. A loss function for polytope function bias is used here, but it can be replaced by any other useful loss function. In the algorithm, `PolytopeFunction` returns weights and bias of the polytope function of a model at a specific point by using algorithms 2 and 3. The function `GAOptimize` represents an implementation of gradient ascent optimization, which is presumed to return a gradient, i.e. a vector that indicates the direction of steepest ascent of polytope function weight for the input point. Details on the practical use of gradient ascent optimization are provided in section 6.5.

Algorithm 9: ConfidenceBoundaryDistance

```
Input  : model, point, numIterations
Output: Boundary distance of point in model
direction = 1
finetuning = False
stepWidth = 1
optPoint = Copy(point)
lastClass = PredictClass(model, optPoint)
for iteration ∈ $\{1, 2, ..., $numIterations$\}$ do
    ,= PolytopeFunction(model, optPoint)
    loss = norm()
    gradient = GAOptimize(loss, optPoint)
    optPoint = optPoint + direction · stepWidth · gradient
    newClass = PredictClass(model, optPoint)
    if newClass ≠ lastClass then
        direction = -direction
        finetuning = True
        lastClass = newClass
        boundaryPoint = optPoint
    end
    if finetuning then
        stepWidth = stepWidth/2
    end
end
return EuclideanDistance(point, boundaryPoint)
```

When using this algorithm, we want to make sure, that the utilization of gradient ascent is in fact possible here, which is not obvious at all: Gradient ascent expects a loss function which is differentiable. Is this the case here?

Recall that, for the loss function using polytope function weights, we use algorithm 2, where we need to mix the weights of individual layers of the target model with regard to the activation pattern in these layers. Mixing these weights are simple multiplications which can be differentiated. Getting the activation pattern of a specific layer is a bit tricky: Precise clamping of the output of a layer to the values 0 and 1 makes it hard in differentiation of this step. Therefore, we use the sigmoid function which continuously maps values to the range [0,1] and clamps everything above 1 and below 0. This function is well differentiable.

The same logic applies to the bias loss function.

Let us move on to the confidence loss function. Ordinarily, this loss function would not be differentiable, because the maximum function is usually not continuous. The same goes for any ReLU that is part of the neural network under scrutiny, since a ReLU is basically a maximum function as well. However, one can approximate the derivative of discontinuous functions: For instance, the derivative of the function $f(x) = \max(x, 0)$ for some input $x \in \mathbb{R}$ is often approximated by

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \\ 0.5 & \text{else.} \end{cases}$$

PyTorch uses this approximation and many more like these for other functions which are not differentiable by default. This helps us a lot: With the maximum function being differentiable, all that remains in the confidence loss function is the function of the neural network M itself. We already know that M is differentiable – at least in PyTorch – because this is used throughout the training phase of M when gradients are computed to update the weights and biases of the network. The only difference being that here, we do not update those parameters, but update the input instead.

6.4 The Gradient Descent Boundary Distance Attack

Putting together the pieces for our gradient descent boundary distance attack is fairly simple now: For each datapoint on which membership shall be inferred, the attack computes the boundary distance and compares it with a priorly selected threshold. If the distance is above the threshold, this datapoint is classified as a member, otherwise it is classified as a nonmember, see algorithm 10.

Choosing a good threshold is important for attack performance, obviously. There are different approaches to this and we already mentioned a possible strategy in section 2.1. When evaluating our attack in experiments, we pick an optimal threshold.

Algorithm 10: Gradient Ascent Boundary Distance Attack

Input : model, point, threshold

Output: Membership classification

```
1 distance = ConfidenceBoundaryDistance(model, point,
    numIterations)
2 if distance > threshold then
3     return $\top$
4 else
5     return $\bot$
6 end
```

6.5 Implementation Details

Implementing our attack requires an efficient gradient descent / ascent to be available if not implemented anew. Luckily, PyTorch provides `torch.optim.SGD`, an implementation of stochastic gradient descent, which can be used for gradient ascent when defining the loss negatively, so that the polytope function parameters' norms increase during optimization instead of decreasing. Of course, the parameterization of PyTorch's SGD influences the outcome of our algorithm. So picking a learning rate, momentum or number of epochs must be done beforehand. Before applying our attack to a dataset, we evaluate some of the dataset's datapoints to tweak parameters so that gradient ascent takes a meaningful path through the model. For the tuning of parameters, we plot the change of the norm when gradient ascent / descent optimizes. Here, we do not finetune on a single boundary (like our attack does) and we note wherever there is a change of class, i.e. a boundary is detected. These visualizations help us in finding a good set of parameters so that the closest boundary is found with a reasonable number of epochs and with a satisfying numerical precision. Access to the dataset is not imperative here: We only need a set of parameters that makes sure, that the decision boundary is reached from any given datapoint with these parameters. So tuning the parameters on a random point and slightly raising the number of epochs, should suffice to get a working attack on real data as well. Furthermore, we observed that we do not need to change our set of parameters for different datasets.

7 Experiments and Evaluation

This chapter evaluates the algorithms and attacks discussed in this thesis. Beforehand, the experimental setup as to what machinery and technologies were used, are explained in detail.

7.1 Experimental Setup

All experiments were run on a NVIDIA DGX-2 deep learning system [14]. This GPU cluster consists of 16 NVIDIA Tesla V100 GPUs which themselves account for an overall number of roughly 82 thousand Cuda cores that perform at two petaFLOPS. The GPUs are supported by 512 GB of GPU memory and 1.5 TB of main memory.

The algorithms developed in this thesis were implemented using the Python [15] programming language which is most commonly used in the research field of artificial intelligence security due to its intuitive syntax, simple programmability and many extensions available through packages.

Amongst various packages used in this thesis, we most importantly rely on the PyTorch [16] package which has also evolved as one of the most popular libraries used, connected to machine learning. PyTorch features a comprehensive yet easy approach to working with neural networks.

7.2 Evaluated Datasets

We perform an empirical study on various datasets to evaluate the usefulness of our attacks. All of the datasets used, have already been used in prior research on this topic, therefore it is easy for us to compare with results from different papers. We give a short summary of what these datasets are about.

The Purchase [17] dataset contains shopping history of customers for different products. It is represented as 600 dimensional binary vectors indicating for each of the 600 products whether the individual of this datapoint has purchased this product or not. Shokri et al. [10] performed a clustering on this dataset resulting in one hundred clusters with individuals of similar purchasing preferences. By doing this, they constructed a classification problem where the task is to predict the correct cluster given a purchase history.

Shokri et al. further used the Location [18] dataset which originally contains mobile check-ins to a social network in the Bangkok area. The Bangkok map is split into same size pieces and for each user it is noted whether this user has made a check-in in this region, resulting in 446 binary features in this dataset. Again, the dataset is clustered into 30 different clusters based on the check-in behavior of users, to obtain a classification problem.

The News [19] dataset consists of newsgroup documents that are categorized into twenty classes. Since datapoints in this dataset are in text form, they have to be transformed into frequencies of terms. These frequencies are then converted into term-frequency times inverse document frequency (tfidf) values by a tfidf formula, which has proven to be more useful than regular frequencies in text classification. We obtain 134410 dimensions after the tfidf transformation.

The Adult [20] dataset contains 14 individuals' attributes like age and ocupation and poses the task to predict whether a given person makes over 50K $ a year.

The Texas [21] dataset attributes patients 6169 important health factors like injury cause and diagnosis. From this, a prediction as to what procedure this patient will undergo, is demanded.

We use the same splits for datasets to obtain training data for all needed models as in the original papers by Salem et al. [3] and Choquette et al. [9] respectively.

7.3 Results and Evaluation of the Activation Polytopes Attack

For the first results, we take a look at the activation polytopes attack. This attack analyzes the structure of activation polytopes around given datapoints to make an attempt at determining whether they are members or nonmembers. Here, we compare ourselves to the paper by Salem et al. [3], since they proposed a very successful attack in this field and we use a very similar approach: We concentrate on their first adversary, where a separate shadow model uses the same architecture as the target model and is trained on data from the same distribution as the training data of the target model. In the training phase of our attack, only the shadow model is used, whereas the actual attack is then carried out on the target model.

As hypothesis 4.1 states, we want to show that the structure of activation polytopes leaks even more information about training data than the confidence could. Therefore we extend the attack data of Salem et al. by our attack data in hope of more leakage. We also extend any attack model in a natural way so that it is able to handle attack data with more dimensions, i.e. more attachments. The attack models used in this scenario are simple feed forward neural networks with two layers and 64 neurons, parameters like learning rate are picked to be equal to the parameters of Salem et al.

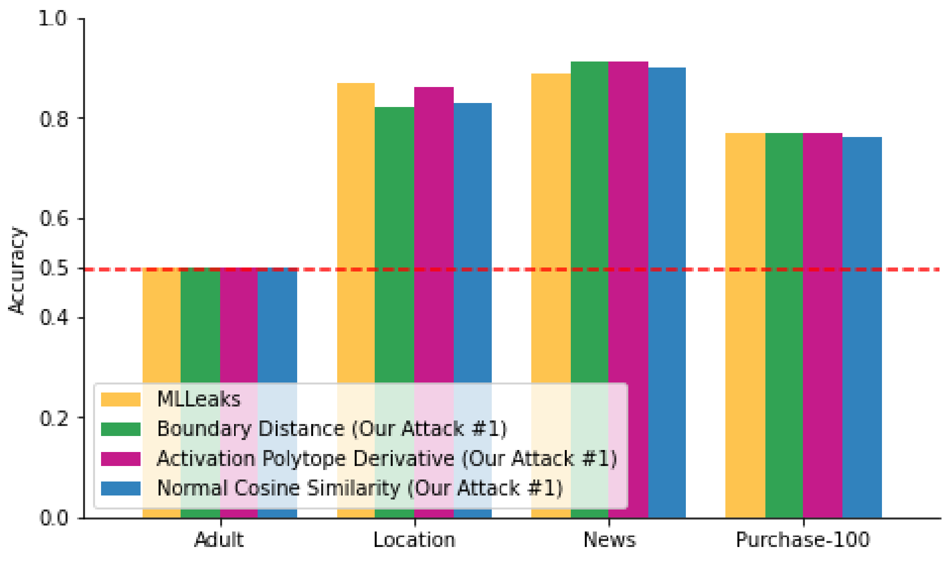

Figure 11: Attack accuracy of the original MLLeaks attack by Salem et al. [3] and the combination of the MLLeaks attack with additional metrics obtained through our activation polytopes attack on different datasets. Attack models are neural networks. Our approach is not able to improve the MLLeaks attack.

The first results of our activation polytopes attack is shown in figure 11. We observe, that our attack does not improve the overall performance of the attack by Salem et al. Different attack metrics perform a bit differently but none of these metrics show a significant increase in attack accuracy.

This means that our first attack – at least in its current state – can not be used to leak any more than already is leaked by the MLLeaks attack when carried out on these datasets. Any decrease in accuracy can be explained by the fact that the neural network used as attack model actually now faces a tougher task in relating the attack's data and the labels in training when our data is not as correlated with membership as is the data by Salem et al. Possible reasons, that this correlation is not so strong could be, that the MLLeaks attack already is almost complete, i.e. it captures most there is to infer membership and any other approach would have a hard time extracting anymore useful information to further improve the attack. Also, it is possible that assumptions 4.2 and 4.3 are not valid assumptions to make: Maybe high dimensional data does not have a similar structure as Clusters2D does. Or maybe, this structure does exist, but it does not vanish when moving away from training data, so that it is harder for our attack to distinguish members and nonmembers.

To evaluate our assumptions, we also carry out our attack as a standalone attack. As attack model, we deploy a neural network, just like in the paper by Salem et al., figure 12 shows the results in this scenario.

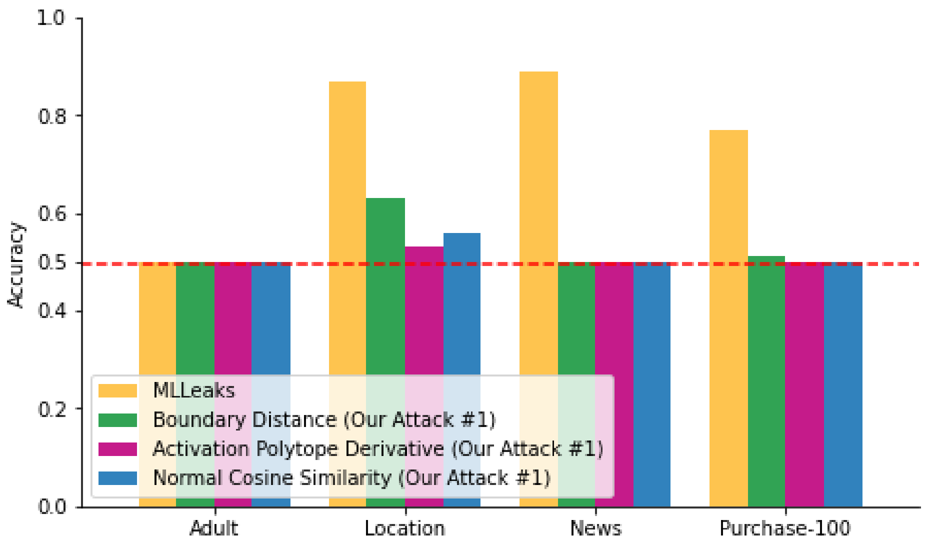

Figure 12: Comparions of attack accuracy between the original ML-Leaks attack by Salem et al. [3] and our standalone activation polytopes attack on different datasets. Attack models are neural networks. Only on one dataset is our attack successfull.

For almost all of the datasets, our attack is not able to leak any information about training data, with the exception of the Location dataset. Our attack's accuracy on the Location dataset is around 60% for the boundary distance metric. The other two metrics reveal some leakage as well, but less. The attack by Salem et al. performs with an attack accuracy of 86% on this dataset.

This result is rather surprising to us, as our two dimensional plots indicated clear leakage and we would have suspected at least some leakage on all of the datasets. Our attack's accuracy on the Location dataset however, is fairly good since it implies definite leakage of sensible information from the target model, meaning that assumptions 4.2 and 4.3 might not be faulty afterall, at least on one dataset so far. However, the other datasets indicate, that our approach is not working yet, either because these datasets are contrary to assumptions 4.2 and 4.3. Or, other metrics than the ones used here or even a different approach than our hyperedge ray shooting from algorithm 5 are necessary to infer membership.

The metric which performs the best, is the boundary distance. This metric is the least experimental of the three, as boundary distance has been used in many attacks so far, for instance the paper by Choquette et al. [9]. So, this result is not so surprising. The other metrics do not seem to capture the structure quite as well.

Overfitting could play a role in the performance of the activation polytopes attack, as it does with many membership inference attacks. Overfitting does occur the strongest on the Location target model, with training accuracy at 100% and test accuracy at 60% which might explain that our attack performs the best here. But other target models overfit similarily, with the Purchase target model performing on training data with 100% accuracy and on test data with 72%, so overfitting alone can not be held accountable for our attack's performance.

Cover's theorem [1] [2] states, that transforming a low dimensional dataset into high dimensions through a nonlinear transformation, makes the resulting dataset more likely to be linearly separable and therefore easier to learn for a neural network. This does not apply directly to our scenario as we do not transform the Clusters2D dataset to get the real datasets evaluated here, but it might still apply to our high dimensional datasets. This would mean that a target model that trains on datasets like Location and Purchase has a simpler task than we would it expect to have when generalizing our Clusters2D dataset to higher dimensions. The target model then does not need to concentrate its activation polytopes as much as it did in the two dimensional case. So even though there is a far greater amount of these polytopes (recall that they grow exponentially with the number of neurons in the target model) they might not form in a way around members that make them distinguishable from nonmembers that good.

To further assess our ray shooting approach of algorithm 5, we increase the number of rays. This only slightly increases the attack accuracy. We conclude, that the ray shooting algorithm is not a useful approach for constructing attack data.

7.4 Improving the MLLeaks Attack Through Gradient Boosted Decision Trees

As mentioned in section 3.7, we want to investigate the usage of gradient boosted decision trees in membership inference. To the best of our knowledge, these classifier have not been thoroughly examined and put to use in this field of research so this is a definitive step into something new.

Aside from the attack model, all other aspects of our attack remain the same as we simply put the attack training and test data into a GBDT. For the sake of completeness, we also deploy a GBDT for the MLLeaks attack.

GBDTs have some parameters that need to be chosen carefully. Most importantly, one has to decide on the number of decision trees and the number of nodes in one of these trees. Both of these parameters influence the performance of the model drastically. The number of decision trees can be chosen somewhat arbitrarily: Each decision tree only tries to improve on the results of the tree before itself, so the overall training error can only decrease when more trees are used. We choose the number of decision trees to be 1024. The second parameter, the number of nodes in a single tree is chosen from the range between one and 60 nodes by whichever gives the best attack performance. The optimal parameter is always below 10 in our experiments. In a realistic attack scenario, choosing this parameter in a gainful manner (regarding the attack accuracy) might pose a problem. Future work should perform an in depth study on how this parameter should be chosen in general.

We measured results that can be seen in figure 13. Interestingly enough, with GBDTs we are not only able to improve our activation polytopes attack, but also, we are able to improve the MLLeaks attack on almost all of the datasets. Increase in accuracy ranges up to 5% points for MLLeaks and also for our attack, at least on the Location dataset. We are able show, that GBDTs are in fact a good choice as an attack model in the field of membership inference.

7.5 Results and Evaluation of the Gradient Descent Boundary Attack

As a second result, we evaluate our gradient descent based boundary attack. This attack uses gradient descent / ascent to find a path from a given datapoint by optimizing towards large polytope function parameters, small output norm and high confidence of the target model and then measures the distance between the datapoint and the first decision

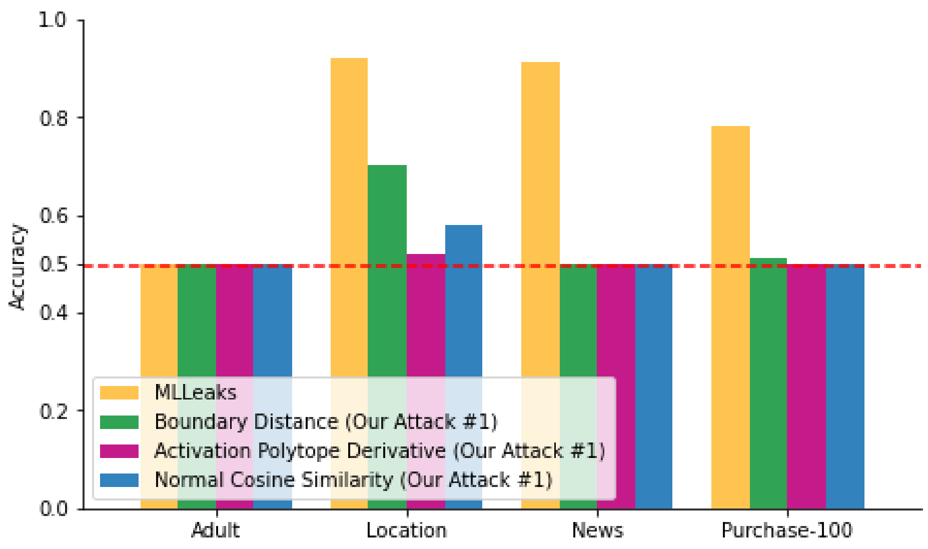

Figure 13: Attack accuracy of the original MLLeaks attack by Salem et al. [3] and our standalone activation polytopes attack on various datasets. Attack models are gradient boosted decision trees (GBDT). Both the ML-Leaks attack and our activation polytopes attack profit from an attack accuracy increase of 5% points. Increasing the number of rays of algorithm 5 only induces a small increase of attack accuracy.

boundary that it discovers. We compare ourselves with one of the latest papers in this field by Choquette et al. [9], who contributed a very successful black box attack that only uses the label that is output by the target model (label only attack).

Attack accuracy of our approach and of the one by Choquette et al. are shown in figure 14. Our attack performs on par with Choquette et al. on the Adult and Location datasets: The label only attack has an accuracy of 59% on Adult, whereas our attack has an accuracy of 58%. On Location, the label only attack has an accuracy of 89%, our attack can even slightly improve on that with an accuracy of 91%. On the Purchase-100 and Texas datasets, our attack underperforms in comparison to Choquette et al.: Their attack has an accuracy of 87% and 80% respectively and our attack has an accuracy of 78% and 58% respectively.

On Adult and Location, we can infer, that our approach is in fact a viable option as an attack. We generally are not surprised, that this attack performs much better than our first approach (the activation polytopes attack of chapter 4) since we crafted this attack from everything we had learned in that chapter and also since boundary attacks in general are often very performant.

In section 6.3 we derived three possible parts of neural networks that leak

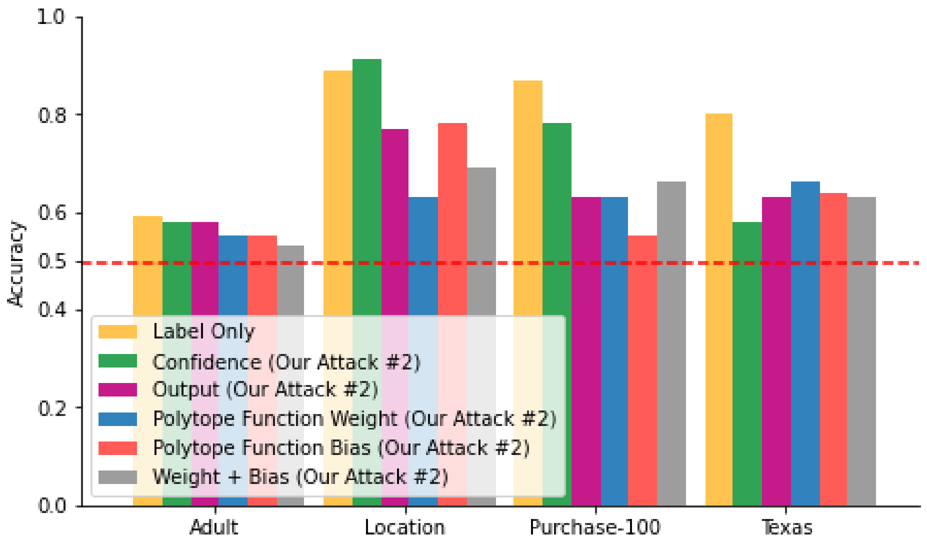

Figure 14: Comparison of attack accuracy between the paper by Choquette et al. [9] (Label Only) and our approach on different datasets. For our attack, we show the performance with an optimal threshold. On two datasets, our attack performs on par with the attack by Choquette et al. We conclude that our attack is in fact a viable option as an attack and that confidence is best for leakage extraction (at least in our attack framework).

information about training data: Confidence, polytope function parameters and output. We conclude that – inside of our attack framework – confidence is the best way to go when constructing a membership inference attack. The polytope function parameters do not extract as much leakage as confidence is able to. Changing the norms of the polytope function parameters does not change attack accuracy that much. Applying an additional softmax transformation does increase the attack accuracy most of the time.

To some extent, we can follow that polytope function parameters and decision boundaries are also aligned on higher dimensional data: We were only observe this correlation on our synthetic two dimensional dataset, but since the gradient ascent attack is able to extract leakage through polytope function parameters, there must be some correlation on real datasets as well.

The question that remains open and needs to be answered, is the following. Since our attack performs in a white box setting, while the attack by Choquette et al. performs in a black box setting: Why does our attack not perform much better than their attack? And why does it only perform as good as their attack on two datasets?

To answer the second question, we observe that Location is the dataset that overfits the most in comparison to the other datasets, followed by Purchase-100, so overfitting is quite possibly helpful for our attack. However, the target model of the Texas dataset overfits more than the target model of Adult, but our attack's accuracy is very similar in both cases, so overfitting can not be the only explanation.

To answer both questions alike, maybe some target models and datasets are more prone to leaking information to our gradient descent / ascent based attack as their confidence / gradient landscape is shaped more suitable for finding the closest boundary through optimizing towards high confidence of the target model. For instance, the Location dataset does induce a slightly better attack performance for our attack than for the attack by Choquette et al. Maybe an optimization strategy different from gradient descent / ascent could do the trick on other datasets as well. Another reason could also be that, the leakage on Location and Adult are 'maxed out', meaning that with decision boundary attacks, no more information about training data and membership can be leaked as current approaches are already too precise.

We consider Cover's theorem [1] [2] to be another reason for our attack's performance (we already mentioned Cover's theorem in detail in section 7.3). If the task of learning a high dimensional datasets is easier for a target model than we would expect from looking at Clusters2D, then it does not need to finetune its individual polytope function parameters to the training data as much. The parameters would then tend not to correlate with decision boundaries as much and leak less information about training data.

8 Conclusion and Outlook

In this thesis, we developed tools that help in understanding how and what neural networks learn on a synthetic two dimensional dataset that is closely modeled after what high dimensional datasets most likely look like. These tools will hopefully help other researchers in finding new opportunities for leakage and then implementing and evaluating privacy attacks like membership inference and model inversion. Using our tools, we derived three main possible leakage opportunities (see section 6.3) given a target model: Its confidence, its output and its polytope functions, i.e. the individual functions the neural network has learned at a specific point, parameterized by weight and bias.

When applying our visualization tools in chapter 5, we were able to observe first indications of an interesting effect of neural network training: Neural networks seem to build a single mountain ridge of large polytope function parameters in the beginning and then after a while, more mountain ridges rise up. We mathematically explained these coalescing neurons through forward weight dependency (which states that hyperedges belonging to neurons of deeper layers need neurons of upper layers to bend and perform breaks) and backward weight dependency (in stochastic gradient descent, weights in upper layers are updated dependant on weights of deeper layers) which, when combined, might cause neurons of all layers to perform a similar task, at least in the beginning of training. We also observed that after training, the norm of polytope function parameters very much correlate with decision boundaries, so the neural network uses this to encode what is has learned. We highlighted possible applications of our observations in the fields of model extraction and federated learning. And we observed that hyperedges associated with neurons of deeper layers change quicker than those of neurons of upper layers. This last observation is due to the fact that the updates in stochastic gradient descent become dependant on more factors when looking at neurons of upper layers. For these neurons, the updates tend to diminish since many small factors might occur.

We crafted two novel membership inference attacks that use more than the confidence of the target model, based on the observations made through the usage of our tools. The first one exploits the layout and structure of activation polytopes (the polytopes over the input space, inside of which neural networks with partial linear activation functions behave like a linear function) and uses neural networks and gradient boosted decision trees as attack models (see chapter 4). The second one uses a gradient descent based approach and optimizes towards large polytope function parameters and low confidence and small output norm for finding the closest boundary (see chapter 6). The boundary distance is compared to a threshold to infer membership.

Through these attacks and our visualization tools we were able to lessen the size of the gap between the theoretical point of view of the likes of Raghu et al. [8] that explain how neural networks learn and a practical point of view like the one of Jayaraman et al. [7] that showcases an attack that uses the loss function of a target model instead of the confidence. We contributed another approach that does not use the confidence of target models as an attack angle and also tied it to observations we made on how neural networks learn.

Our first attack on activation polytopes does in fact reveal leakage but underperforms in comparison with the MLLeaks attack by Salem et al. [3], see section 7.3. In section 7.4, we also assessed the implementation of gradient boosted decision trees (GBDT) as attack models and were able to slightly improve the performance of the MLLeaks attack. Therefore we follow that GBDT are a good fit as an attack model in membership inference attacks

Our second attack on polytope function parameters, output norm and confidence is able to perform on par on some datasets in comparison to the latest research by Choquette et al. [9] in the best case, see section 7.5. Out of polytope function parameters, output and confidence, we conclude that confidence is the best way to go for leakage extraction in our framework.

We discuss Cover's theorem [1] [2] as a possible reason for why the leakage of activation polytopes and activation functions is not that strong, see section 7.3 and section 7.5.

There are a lot of possibilities to expand on our work. Our observations on neural network training – the coalescing neurons at the beginning of training and the correlation between polytope function parameters and decision boundaries – should be tested on datasets of higher dimensional data, to further confirm that any effects observed on Clusters2D also pop up there. Possible applications of our observations, like model extraction and federated learning should also be evaluated.

We give a mathematical explanation for the coalescing neurons in section 5.4. An even deeper mathematical analysis, especially concerning backward propagation, could help in understanding exactly what is happening and why. More analysis on the progression of activation polytopes and polytope functions throughout the training of a neural network could also be helpful. This could furthermore be used to improve our attacks.

Since we proposed one way of visualizing neural networks and assessing possible attack angles for membership inference and model inversion, there should be even more proposals that try to close the gap between the theory and the practice of leakage of neural networks. It would also be interesting to see the effect of combining our insights on the structure of activation polytopes and our observations on polytope function parameters with other attacks, like The Secret Revealer by Zhang et al. [6].

We contributed to the trend of trying to understand what neural networks do and how we can exploit that, aside from taking a look at only the confidence of the target model. This trend will certainly keep on growing in the future and we are looking forward to new intriguing developments in this field.

Bibliography

[1] Thomas M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Trans. Electron. Comput.*, 14:326–334, 1965.

[2] Simon Haykin. *Neural Networks and Learning Machines (Third Edition)*. Pearson, 2009.

[3] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models, 2018.

[4] Google. Google zeitgeist. `https://archive.google.com/zeitgeist/2012/`, 2012.

[5] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. pages 1322–1333, 10 2015.

[6] Yuheng Zhang, Ruoxi Jia, Hengzhi Pei, Wenxiao Wang, Bo Li, and Dawn Song. The secret revealer: Generative model-inversion attacks against deep neural networks, 2020.

[7] Bargav Jayaraman, Lingxiao Wang, Katherine Knipmeyer, Quanquan Gu, and David Evans. Revisiting membership inference under realistic assumptions, 2021.

[8] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks, 2017.

[9] Christopher A. Choquette-Choo, Florian Tramèr, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. *CoRR*, abs/2007.14321, 2020.

[10] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 3–18. IEEE Computer Society, 2017.

[11] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. Hopskipjumpattack: A query-efficient decision-based attack, 2020.

[12] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics Data Analysis*, 38(4):367–378, 2002. Nonlinear Methods and Data Mining.

[13] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[14] NVIDIA Corporation. NVIDIA DGX-2 Datasheet. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf`, 2019. Accessed: 21.08.2021.

[15] Python Software Foundation. Python language reference, version 3.8. `https://www.python.org`, 2021.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[17] Acquire valued shoppers challenge. `https://www.kaggle.com/c/acquire-valued-shoppers-challenge/data`, 2021.

[18] Dingqi Yang, Daqing Zhang, and Bingqing Qu. Participatory cultural mapping based on collective behavior data in location-based social networks. *ACM Trans. Intell. Syst. Technol.*, 7(3):30:1–30:23, 2016.

[19] News category dataset. `https://www.kaggle.com/rmisra/news-category-dataset`, 2021.

[20] Uci adult census income. `https://archive.ics.uci.edu/ml/datasets/adult`, 2021.

[21] Hospital discharge data. `https://www.dshs.texas.gov/THCIC/Hospitals/Download.shtm`, 2021.