# PRIVACY PRESERVING MACHINE LEARNING

CRYPTOGRAPHIC COMPUTING

ARNAR THORDUR HARDARSON , 201303540

CHRISTIAN KOLLER NIELSEN , 201303527

STEFFEN SKOVSGAARD JENSEN , 201303578

COURSE PROJECT

April 2020

Advisor: Claudio Orlandi

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

# PRIVACY PRESERVING MACHINE LEARNING

ARNAR THORDUR HARDARSON
CHRISTIAN KOLLER NIELSEN
STEFFEN SKOVSGAARD JENSEN

Cryptographic Computing

Course Project
Department of Computer Science
Science & Technology
Aarhus University

April 2020

## ABSTRACT

In machine learning more data results in better models, therefore multiple parties can have interest in training a model collaboratively, whilst keeping their input private. In order to evaluate techniques for private collaborative machine learning, the protocol from *Private Collaborative Neural Network Learning* [2] was implemented. The protocol uses differential privacy in order to protect the privacy of individual records in each peers disjoint datasets. In this work the effects of adding controlled noise from differential privacy using logistic regression was explored.

The results of our experiments were as expected, with a drop in accuracy the more noise that was added to the model. In contrast to the results of *Private Collaborative Neural Network Learning* [2], our results show that the Gaussian mechanism with $(\epsilon, \delta)$-DP performed better than the Laplace mechanism with $(\epsilon, 0)$-DP in terms of $\epsilon$.

# CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## LISTINGS

# 1 INTRODUCTION

Machine learning gives computers the ability to learn without being explicitly programmed, this means giving computers data, they themselves can learn mathematical models to make sense of the data. In order to use machine learning one must: have a lot of data, the data must have a pattern and the pattern should not be known. Generally the more data available to train the machine learning model on, the better performance the machine learning model will have. These datasets are not limited to any single domain and are flexible depending on the data it is given.

Machine learning algorithms have been around for a long time and have in recent years gained a lot of attention, due to i.a. improvements in computing power and improvements of the computational machine learning models. This development have lead to improvements in everything from search queries at google to self-driving cars and fraud detection.

In many industries problems emerge within machine learning due to having regulations that require that the data is kept private. An example of this could be in the healthcare industries, where the patients health records must be kept private. Other examples include industries where competetion is the main factor for secrecy and not are willing to share individual records, or simply a companies interest to keep a users data private to maintain plausible deniability or as a marketing point.

As a contrast, since larger datasets gives better machine learning models one could be interested in data from competitive or data regulated companies, to create better machine learning models. Such companies would not want to reveal their data and give their competitors an advantage. Introducing privacy to machine learning algorithms can solve these issues.

The real world examples motivate the goal to achieve privacy in machine learning, such that multiple parties can jointly compute functions, while keeping their inputs private. This can be done with secure multiparty computation (MPC). MPC insures that computation of a function can be done jointly without revealing information about the input, but it comes with some drawbacks:

1. It is infeasible to run generic MPC protocols over many remote parties, due to high communication costs.

2. The trained model might leak information about the dataset that was used to train it.

An approach to address this problem is differential privacy (DP). The goal of DP is to evaluate a function, such that even with auxiliary information, it is hard to tell if a record was in the dataset or not. The high level idea is to add noise to the result, such that the trained model has a probability distribution over the outputs, such that if person j changed their input, the relative probabilities of any output does not change by much.

This project is an investigation of preserving privacy in collaborative machine learning. It will present a recent construction of privacy preserving in a collaborative machine learning setup and look into how differential privacy interacts with machine learning.

# 2 RELATED WORK

In this chapter the work related to privacy preserving machine learning is presented.

## 2.1 PRIVATE COLLABORATIVE NEURAL NETWORK LEARNING

In 2017 Chase et al. presented a way of doing private collaborative machine learning with a neural network [2]. For the privacy aspect they utilize three aspects of cryptography, namely multiparty computation, differential privacy and secret sharing. They presents two algorithms, an alogorithm called *Private Gradient Descent* and an algorithm called *Collaborative Gradient Computation*, which both addresses the privacy concern.

### *Private Gradient Descent*

The private gradient descent algorithm is not a collaborative algorithm, which is why multiparty computation and secret sharing are omitted, since these aspects of cryptography address privacy concerns in collaborative setting.

**Algorithm 1** Private Gradient Decent (PGD)

```
1: Input: $C > 0$, a bound size of the gradients
2: Input: $\epsilon^*, \delta^*$, differential privacy parameters
3: Input: $T$, number of epochs
4: Input: $w_0$, initial weight vector the neural network
5: Input: $m > 0$, size of the mini-batches
6: Output: $\hat{w}$
7: function PGD
8:     Compute: $\epsilon, \delta$
9:     Compute: $b$ based on $\epsilon, \delta$
10:    Compute: $n$ which is dataset size divided by the norm used
11:    for $t = 0, 1, 2, ..., nT$ do
12:        Let: $Z_T$ be the next mini-batch of size m
13:        Compute: $g_t = \sum_{z \in Z_t} \mathrm{Clip}(C, F'(w_{t-1}, z)) + \mathrm{rand}(b)$
14:        Compute: $(w_t, a_t) \leftarrow \phi(w_{t-1}, a_{t-1}, g_t)$.
```

In algorithm 1 the gradient descent algorithm is shown with differential privacy. Gradient descent is an algorithm used to optimize the machine learning model according to the weights of the model. The algorithm starts with a random sample of weights, noted $w_0$. For optimizing the weights a gradient is computed, this gradient is pointing to the next local minimum of the function. The computation of the gradient along with applying differential privacy is seen in line 13 of algorithm 1. For computing the gradient without applying differential privacy it will look as follows

$$g_i = \sum_{z \in Z} F'(w_{i-1}, z)$$

where $F$, the cost function, is the function to be optimized. For applying differential privacy, there is a function called *Clip*, which is used to make sure that each term $F'(w_{i-1}, z)$ is bounded, since differential privacy requires bounding the sensitivity of the output with respect to changing a single input record. The clipping function works in the following way

$$\mathtt{Clip}(C, x) = \min(1, \frac{C}{||x||}) \cdot x,$$

if the norm of $x$ is greater than the sensitivity bound $C$, then $x$ is being clipped, otherwise $x$ is allowed to remain as it was, since it does not violate the sensitivity bound of the output. The last part of line 13 is adding the noise. The noise is drawn from a probability density distribution, either the Laplace or the Gaussian distribution. The Laplace distribution is used with respect to norm $L_1$ which gives $(\epsilon, 0)$-DP and $L_{inf}$ which gives $(\epsilon, \delta)$-DP, and the Guassian distribution is used with respect to norm $L_2$ which gives $(\epsilon, \delta)$-DP. After this step, the weights are updated by using the Adam optimization algorithm.

### *Collaborative Gradient Computation*

For the collaborative Gradient Computation to preserve privacy, the cryptographic aspects differential privacy, secret sharing and multiparty computation will be utilized. It is assumed that all parties follows the protocol and that they do not collude.

It should be noted, that there are two types of actors in algorithm 2, namely the peers and the hosts. The peers have individually private data, and wants to collaborate in making a machine learning model. The hosts help the peers in computing the gradient of the collaborative model by multiparty computation.

In the first stage of the algorithm it is noted, that each peer is for every data-point $z$, in their respective dataset $Z^i$, computing a gradient with respect to differential privacy, meaning the clip function is used to make sure, that the norm of each gradient would not exceed the sensitivity bound on the function for computing gradients. These gradients are summed together to create a gradient $g^i$, that is representing the whole $Z^i$. For the purpose of secret sharing, they draw a random vector $r^i$, which is uniformly distributed, with the same dimensions as the gradient. The secret sharing is seen in line 11 and 12 of algorithm 2, and it is a simple secret sharing scheme, where one host is receiving $g^i + r^i$ and the other host is just receiving $r^i$.

In the second stage of the algorithm, the two hosts sums their secret shares respectively

$$H_1 : \tilde{g_1} = \sum_i (g_i + r_i) \text{ smod } mC$$
$$H_2 : \tilde{g_2} = \sum_i r_i \text{ smod } mC$$

where smod is defined as $x \text{ smod } C = ((x + C) \mod 2C) - C$ and $m$ is the batch size of the data. The hosts then compute a seed for a random number generator $s_i$. Then the two hosts perform a multiparty computation protocol, which in the paper is based on Yao's garbled circuit[15]. Briefly, Yao's garbled circuit is a way to evaluate an encrypted function on encrypted inputs, which will be explained in chapter 3. The output of the garbled circuit is the gradient of the collaborative model, which is computed in the following way

$$g = ((\tilde{g_1} - \tilde{g_2}) \text{ smod } mc) + \text{Rand}_{s_1 \oplus s_2}(b)$$

The noise of differential privacy is added in this step, since if it was added earlier, then there would have been added an unnecessary amount of noise, which would make the machine learning model worse.

**Algorithm 2** Collaborative Gradient Computation

```
1: Input: $k$ parties, each one with $Z^i$
2: Input: $C > 0$, a bound size of the gradients
3: Input: $b$, a parameter on the random noise$Z^i$
4: Input: $w$, the current weight vector
5: Input: $m > 0$, the batch size
6: Output: $g = \sum_i \sum_{z \in Z^i} \text{Clip}(mC, F'(w_t, z)) + \text{rand}(b)$
7: function STAGE I
8:     each party performs:
9:     Compute: $g^i = \sum_{z \in Z^i} \text{Clip}(mC, F'(w_t, z))$
10:    Compute: $r^i \leftarrow \text{Uniform}[-mC, mC]$, a random vector with the
       same dimensions as $g^i$ based on
11:    Send: $g^i + r^i$ to host $H_1$
12:    Send: $r^i$ to host $H_2$
13: function STAGE II
14:    $H_1$ :
15:    Compute: $\tilde{g^i} = \sum_i (g_1 + r_i) \text{ smod } mC$
16:    Compute: $s_1$ = a seed for the random number generator
17:    $H_2$ :
18:    Compute: $\tilde{g^i} = \sum_i (r_i) \text{ smod } mC$
19:    Compute: $s_2$ = a seed for the random number generator
20:    Host 1 and host 2 use the garbled circuits protocol to compute $((\tilde{g_1} -$
       $\tilde{g_2})\text{smodmC}) + \text{Rand}_{s_1 \oplus s_2}(b)$
```

*Results*

The algorithm was tested on the MNIST dataset, which consist of 60000 training examples of hand-written testing samples and 10000 testing samples. Two experiments were constructed, one with a small network of 3 layers and one with a large network of 4 layers. Each classifier was trained with 50 epochs with $\delta = 10^{-3}$.

| | $L_{inf}$ | $L_1$ | $L_2$ |
|---|---|---|---|
| $\epsilon = 0.5$ | 58.5% | 89.1% | 82.3% |
| $\epsilon = 2$ | 73.7% | 91.8% | 87.8% |
| $\epsilon = 8$ | 88.1% | 94.8% | 92.1% |
| No noise | 98.5% | | |

Table 1: Precision of the small neural network

| | $L_{inf}$ | $L_1$ | $L_2$ |
|---|---|---|---|
| $\epsilon = 0.5$ | 24.0% | 84.7% | 51.8% |
| $\epsilon = 2$ | 61.1% | 90.3% | 88.4% |
| $\epsilon = 8$ | 80.3% | 92.9% | 90.4% |
| No noise | 98.9% | | |

Table 2: Precision of the large neural network

In the both networks the $L_1$ norm gets the best accuracy and in all but one case the smaller network have better accuracies. A larger network has a larger number of parameters and thereby a larger amount of noise is injected. More rounds will increase the number of gradient steps, but also increases the amount of noise added. In the protocol the amount of data communicated is independent of the size of datasets. The experiments showed that accuracies drop when the number of parameters in the network gets large.

## 2.2 GENERATIVE ADVERSERIAL NETWORKS(GAN)

For highlighting some leakage of information Hitaj et al. describes an effective attack, especially on privacy preserving collaborative deep-learning [8]. This shows that despite adding differential privacy, the attack will still be able to obtain some information about the training data from network, based solely on the gradients. In contrast to what has been showed in section 2.1, it uses a different security model dealing with an active adversary, in which he modifies his gradient inputs to learn more about a specific data class from other parties in a collaborative gradient descent computation. This shows despite having privacy preserving machine learning by adding differential privacy, that the data cannot be kept completely secret, and describes different use-cases where privacy can become an issue.

The attacking strategy with a GAN network works as follows, and is taken from [8]:

1. Assume two participants A and V. Establish and agree on the common learning structure and goal.
2. V declares labels $[a, b]$ and A labels $[b, c]$.
3. Run the collaborative deep learning protocol for several epochs and stop only when the model at the parameter server (PS) and both local models have reached an accuracy that is higher than a certain threshold.
4. First, the Victim trains the network:
   a) V downloads a percentage of parameters from PS and updates his local model.
   b) V 's local model is trained on $[a, b]$.
   c) V uploads a selection of the parameters of his local model to PS.
5. Second, the Adversary trains the network:
   a) A downloads a percentage of parameters from the PS and update his local model.
   b) A trains his local generative adversarial network (unknown to the victim) to mimic class a from the victim.
   c) A generates samples from the GAN and labels them as class $c$.
   d) A's local model is trained on $[b, c]$.
   e) A uploads a selection of the parameters of his local model to PS.
6. Iterate between 4) and 5) until convergence.

The essential part of this attacking strategy is that the adversary is using the learned distribution of class $a$ to create fake samples, which are labelled as class $c$ and then injected into the distributed learning process. The victim is thereby being deceived into working harder in order to distinguish between the two classes, and therefore will the victim reveal more information than intended about class $a$. This means that the generated samples for class $c$, will eventually converge and be indistinguishable from the data in class $a$. Thus being able to generate data with the same distribution as the data in a. This may have different security implications based on the kind of data it is. As if the classes specified are images, and you want to keep an identity secret. Being able to generate the same distribution as the normal image, it will give, perhaps a bit blurred, image that reflects the actual person. While if it is general samples of medical records, containing data like weight, height, age and perhaps which disease they have, it will generate data from the same distribution as the real data in such a way the correlations appear, like the older a person is the higher the probability is for cancer.

In the [8] they provide plentiful of experiments in which they show, that they can with the method above create a GAN network that can create closely resembled numbers from the MNIST dataset, without seeing data from that class itself. This shows that the attack can be used in practice with a collaborative gradient descent computation. It is also shown that they can get results, also with some loss in precision e.g. the images become more blurred, with applied differential privacy techniques.

Whenever a collaborative gradient computation is performed, [8] states that this attack would work, despite good efforts of providing differential privacy in a collaborative gradient computation. This means if the discriminator keeps learning then the generator will also keep learning, and become better at creating records within the same distribution.

As an overall conclusion to the paper they finish off by still having uncertainties how such a GAN network would perform against device or user-level privacy, which they leave as future work.

Through the experiments of [8], they evaulated the GAN network on two different datasets, the MNIST dataset and a AT&T dataset which is a database consisting of images of faces.

## 2.3 MEMBERSHIP INFERENCE ATTACK

Differential privacy can protect a trained model from leakage of information about the individual records which it was used to train on. Hence it is interesting to investigate attacks that do just that, and look at how differential privacy affects that. The membership inference attack by Shokri et al.[14] can, given a trained target model, determine if a given record is in the training set or not. The attack is not based on a specific dataset or model. The model is looked upon as a black box and the adversary can only give input to the model and see the outputs. In [14] a classification model is used and it is *assumed* that the model outputs the probability vector, which contains the probability of the input data belonging to the classes. The probability vector have one probability per class. Normally it is only the label/class of the highest value in the probability vector that is outputted. If a the probability vector is high in one of the classes we say that the confidence is high.

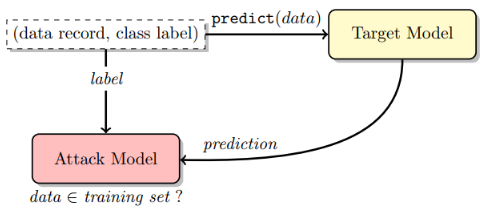

Figure 1: Membership Inference Attack [14]

In figure 1 the target model outputs a prediction consisting of a probability vector. The probability vector along with a label is given to the attack model, which then will decide if the input record that in the training set or not. The goal for the attacker is to construct the attack model. The attack model is created by exploiting that machine learning models behave differently on the data they are trained on, oppose to new data. The target model produce different distributions depending of that class is the inputs true class. Hence the attack model is a collection of models, one for each class of the target model. The attack model is trained using shadow models that is intended to behave like the target model.

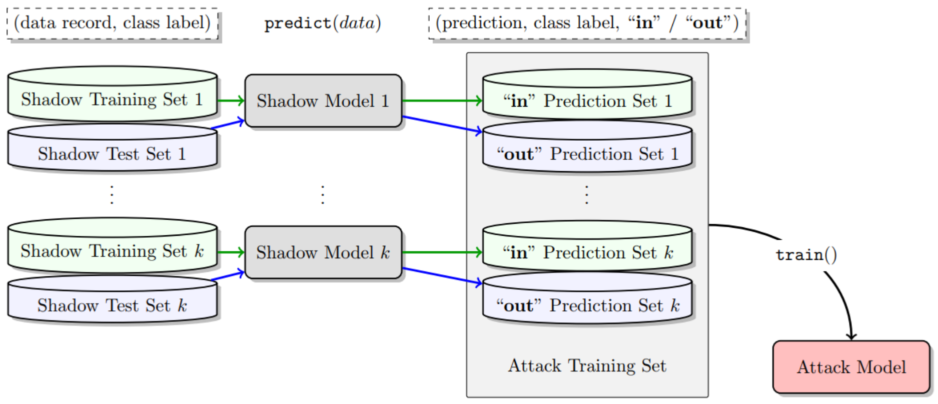

Figure 2: Training of Shadows Models and the Attack Model [14]

The shadow models are trained with data of the same format as the target function and it is assumed this data is disjoint the target models training data. The shadow models are trained with the same model as the target model, e.g. logistic regression. The idea is that similar models, trained on similar data behave similarly. This results in the adversary now having trained shadow models, where the training set is known. Before an adversary can create the shadow models, he needs training data with similar distribution to the target models training data. Data generated by the target model with high confidence should be statistically similar to the targets training data set. In [14] an algorithm called synthesize can use a target function and can create data similar to the dataset. The trained shadow models is then used to generate a datasets consisting of probability vectors and a label stated whether it was data in the training or not. These new datasets with labels (in/out) and records (probability vectors) can be used to create a new supervised attack model. Now an attack model is created, that given a probability vector can output whether a data record was in the data set or not.

In experiments Google's prediction API was used a black box machine learning service on the MNIST dataset is and had training accuracy 98.4%, testing accuracy 92.8% and attack precision 51.7%, while on other datasets an attack precision of 93.5% was reached. By definition differential privacy limits the success rate of membership inference attacks, performed as described above.

# 3 BACKGROUND

This chapter presents in depth background information on machine learning techniques and principles, as well as cryptographic techniques relevant for doing private collaborative machine learning.

## 3.1 MACHINE LEARNING

### 3.1.1 *What is Machine Learning*

Machine learning works with an input domain $X$, a target $Y$, and some unknown target function $f : X \to Y$ and the goal is to learn a hypothesis $h$ that estimates the target function on new unknown data.

To do machine learning training data is needed, which is given to a learning algorithm. The learning algorithm finds the best $h$ from a set of hypothesis. The hypothesis set restricts the search space and adds solution preferences. A final hypothesis $h \approx f$ is chosen, such that it has low error on training data and such that the out of sample error, $E_{out}$, is close to the in sample error, $E_{in}$.

### 3.1.2 *Machine Learning techniques*

#### *Classification*

In machine learning, classification [11] uses the features of the data, $X$, to identify what class it belongs to. An example could be to classify *spam* and *non-spam* mail. For a classification problem data points are classified by splitting the data with a hyperplane. In a two class classification problem all points on one side of the hyperplane will be in one class, while the remaining are in the other class.

Linear models often use a signal $s$, and in linear classification $h(x) = sign(s)$, where $s = \sum_{i=0}^{d} w_i x_i$. $w$ is a weight vector that is learned from a set of labelled training samples. $w$ ends up showing which features of the input, that is important for the classification, and $d$ is the dimension of the input and weights, which means each dimension in the input data has a corresponding weight.

If the data is non-linear separable the space of the data can be transformed into a higher dimension, and this will preserve the linearity in $w$, and then one can do the classification.

*Logistic Regression*

Logistic regression finds the probability of how certain the model is about something, i.e. the probability of an input belonging to a class. For example, if a bank wants to check whether a user could get credit approval, a classification would simply output yes or no as an answer, while a logistic regression [11] problem would output the probability of the answer, so the bank could make the judgement themselves based on the certainty. These probabilities can be transformed into classifications by picking the class with the highest probability.

In logistic regression $h(x) = \theta(s)$, where the logistic function $\theta$, see equation 1, goes between 0 and 1, and says how certain you are about the answer.

$$\theta(s) = \frac{e^s}{1 + e^s} \tag{1}$$

This is also called sigmoid because the function looks like a flattened s, equation 1. The final hypothesis $h$ is chosen s.t. $h \approx f$, by grading different hypothesis according to the likelihood, equation 2. If the probability of $y$ given $x$ is high, then the hypothesis is good.

$$p(y|x) = \begin{cases} h(x) & \text{for } y = +1; \\ 1 - h(x) & \text{for } y = -1; \end{cases} = \theta(yw^T x) \tag{2}$$

The likelihood of an entire dataset is computed by taking the product of equation 2 for all $x, y$ in the dataset.

$$\prod_{x,y \in D} P(y|x) = \prod_{x,y \in D} \theta(yw^T x) \tag{3}$$

Maximizing the likelihood with respect to $w$ in equation 3, can be done by minimizing the $E_{in}$.

$$\begin{aligned} E_{in} &= -\frac{1}{N} \prod_{n=1}^{N} \ln(\theta(yw^T x) \\ &= \frac{1}{N} \sum_{n=1}^{N} \ln(\frac{1}{\theta(yw^T x)}) \\ &= \frac{1}{N} \sum_{n=1}^{N} \ln(1 + e^{-y_n w^T x_n}) \end{aligned} \tag{4}$$

To train the logistic regression model, algorithm 3, an algorithm called gradient descent is used. Gradient descent is a non-linear optimization algorithm for finding the minimum of a function. The algorithm starts with $w_0$ and will then in each iteration take a step along the steepest slope. The direction is noted $v$ and step size is noted $\eta$ and the next weight will be $w_1 = w_0 - \eta \cdot v$.

**Algorithm 3** Logistic Regression Algorithm

```
1: for $t = 0, 1, 2, \dots$ do
2:     Compute the gradient: $\Delta E_{in} = -1/N \sum_{n=1}^{N} \frac{y_n x_n}{1 + e^{-y_n w^T x_n}}$
3:     Update the weights: $w(t+1) = w(t) - \eta \Delta E_{in}$
4:     Iterate to until it is time to stop
5: Return the finals weights $w$
```

*Neural Networks*

Over the last couple of years a machine learning technique known as neural networks [9] or artificial neural networks has become increasingly popular. This is due to an increase in computational power along with advances in the field for optimizing networks, such that deep neural networks have become feasible. Neural networks is a model to some degree based on the idea of how the brain works.

From the brain analogy, a neural network is built upon neurons who are connected, in different fashions. For the purpose of this work only the feed-forward neural networks will be discussed, where the data propagation only moves in a single direction between the layers. A simple network with a single hidden-layer would contain a input layer, the hidden layer and an output layer. Each neuron in the input layer is connected to each neuron in the hidden-layer and each neuron in the hidden-layer is connected to the output layer.

Each of these connections between the neurons have an associated weight, so when feeding data through the input layer of the neural network, which has the same dimension as the input, the neural network passes it through the network, each subsequent neuron sums up the input value multiplied by the weight of the connection, and applies an *activation function* and then forwards the value to the next layer. These activation functions introduce some non-linearity to the model, which lets the output of the network have a value that could not be produced by a linear function.

Commonly used activation functions are sigmoid, $\sigma(x) = 1/(1 + e^x)$, ReLu $f(x) = max(0, x)$ and tanh, $tanh(x) = 2 \cdot \sigma(2x) - 1$. Example of how single neuron receives values is illustrated in figure 3, where a single neuron receives inputs from the previous layer with $\phi$ being the activation function.

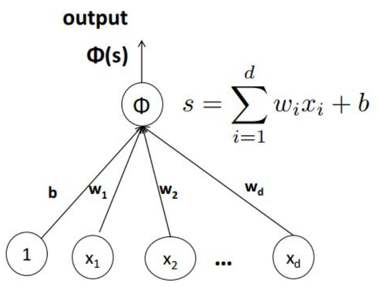

Figure 3: An example of the computation a single neuron performs in a neural network

A neural network model is initialized with random weights, and trained through *backpropagation*. In backpropagation, the propagation starts from the output activations and goes back through the network. All the output and hidden neurons computes the difference between the target and output value, which is noted δ. Then the input activation and the output delta of the weights are multiplied to find the gradient of the weight. Finally the learning weight, a percentage of the weights gradient, is subtracted from the weights.

Regular neural networks, which we just described don't scale well to images, as the input size quickly becomes unmanageable with regards to the amount of weights added to the network, which can be considered wasteful and may lead to overfitting of the model. As a solution to this, convolutional networks are often used with images to constrain the architecture in a sensible way. Convolutional neural networks have neurons arranged in 3 dimension, height, width and depth, where depth usually is the color channels R,G,B. The convolutional layer will then compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. An activation layer will then apply an elementwise activation function such as ReLu. A pooling layer will then be applied as a down-sampling operation along the width and height. The output of this will then be used as an input to a previously described neural network, which finally outputs the labels.

As a result a common architecture for convolutional neural network, when working with images, is first having a convolutional layer followed by a Pooling Layer and finally a regular neural network.

Overall convolutional layers can be seen as a preprocessing step, where above the simplest model with a single convolutional layer is described. Having multiple convolutional layer is very common in practice and has had very good results when classifying images.

## 3.2 SECURE MULTIPARTY COMPUTATION

In private collaborate machine learning MPC protocols are interesting, because they allow $k$ parties whom each hold a private value $x_i$ and to evaluate $f(x_1, ..., x_n)$ such that each party only learns the result.

Some MPC primitives are generic in the sense that they can be applied to a broad range of functions, such as the garbled schemes introduced by Andrew Yao [15], and others are designed for specific function. A garbled scheme allows two parties to evaluate an encrypted function on some encrypted inputs. The definition of the scheme is from note [13] and is as follows: $G = (Gb, En, De, Ev, ev)$. The algorithm $ev$ is used to evaluate the circuit without encoding $ev(f, x) = f(x)$. The remaining algorithms are described below:

CIRCUIT GENERATION To generate a garbled circuit $(F, e, d)$ from a boolean circuit $f$ with $T$ wires the algorithm $Gb$ does:

- For each wire $i \in [1..T]$ in the boolean circuit: choose two random strings $(K_0^i, K_1^i) \leftarrow \{0,1\}^k\{0,1\}^k$. Let $T$ be the index of the output wire: then define $d = (Z_0, Z_1) = (K_0^T, K_1^T)$. If the circuit has $n$ input wires define $e = \{K_0^i, K_1^i\}_{i \in [1..n]}$
- For all $i \in [n+1, T]$ define a garbled table $(C_0^i, C_1^i, C_2^i, C_3^i)$
  - For all $(a, b) \in \{0,1\}\{0,1\}$ compute $C'_{a,b} = G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus (K^i_{\neg(a,b)}, 0^k)$
  - Choose a random permutation $\pi : \{0,1,2,3\} \to \{0,1\}\{0,1\}$ and add $(C_O^i, C_1^i, C_2^i, C_3^i) = (C'_{\pi(0)}, C'_{\pi(0)}, C'_{\pi(0)}, C'_{\pi(0)})$ to $F$

ENCODING $En(e, x)$ parses $e = \{K_0^i, K_1^i\}_{i \in [1,...,n]}$ and outputs $X = \{K_{x_i}^i\}_{i \in [1,...,n]}$

EVALUATE $Ev(F, X)$ parses $X = \{K^i\}_{i \in [1,...,n]}$ and for all $i \in [n+1, T]$:

1. Recover $(C_O^i, C_1^i, C_2^i, C_3^i)$ from $F$
2. For $j = 0, 1, 2, 3$ compute: $(K'_j, \tau_j) = G(K^{L(i)}, K^{R(i)}, i) \oplus C_j^i$
3. If there is a unique $j$ s.t. $\tau_j = 0^k$, then define $K_i = K'_j$, otherwise abort and output $\perp$.

Output $Z = K^T$

DECODING $De(d, Z)$ parses $d = (Z_0, Z_1)$ and outputs $0$ if $Z = Z_0$ and $1$ if $Z = Z_1$, or $\perp$ if $Z \notin \{Z_0, Z_1\}$

The following protocol for garbled circuits is from note [13] and works as follows.

1. Alice and Bob have inputs $x, y \in \{0,1\}^n$ and want to compute $f(x,y)$, which is a circuit where the first $n$ input bits are given by Alice and the second $n$ input bits are given by Bob. The encoding information will be noted as $e = (e_x, e_y)$, where $e_x$ is the encoding information for Alice's input and $e_y$ is the encoding information for Bob's input.

2. Alice runs $(F, e_x, e_y, d) \leftarrow Gb(1^k, f)$ and sends $F$ to Bob.

3. Alice runs $X \leftarrow En(e_x, x)$ and sends $X$ to Bob.

4. Alice and Bob run a secure two party computation e.g. oblivious transfer [12], where Alice inputs $e_y$, Bob inputs $y$, and Bob learns $Y \leftarrow En(e_y, y)$.

5. Bob computes $Z \leftarrow Ev(F, X, Y)$.

6. Alice sends $d$ to Bob who outputs $z \leftarrow De(d, Z)$

## 3.3 DIFFERENTIAL PRIVACY

*Why Differential Privacy*

Differential privacy is a tool developed to avoid leakage of private information in the result of an evaluated function. The result of the evaluated function could be a trained machine learning model, which could leak information about the dataset used to train it [2]. Assume that an adversary has a model trained and that feature $i$ is has-red-hair and the learned weight $w_j \neq 0$. If there is only one person in town with red hair, the adversary knows that he/she was in the dataset.

*Distance Between datasets*

The $L_p$-norm[6] of vectors $x = x_1, x_2, ..., x_n$ is used in differential privacy and it is defined as follows:

$$||x||_p := (\sum_{i=1}^{n} |x_i|^p)^{1/p}$$

The $L_1$-norm can be used to tell the size of a single dataset or the distance between datasets $||D - D'||_1$, i.e. how many records differ between $D$, $D'$.

### *What is Differential Privacy*

**Definition 1.** *A function* $f : X^* \rightarrow Y$ *provides* $(\epsilon, \delta)$*-DP if for any set of possible outputs* $S \subseteq Y$*, and any pair of datasets* $D, D' \in X^*$ *that* $\|D - D'\|_1 \leqslant 1$

$$\Pr[f(D) \in S] \leqslant e^{\epsilon} \cdot \Pr[f(D') \in S] + \delta$$

*Where the privacy guarantee is parametrized by* $\epsilon, \delta \geqslant 0$ *[2].*

The definition of differential privacy insures that the absence or presence of single records in the dataset does not change the output of the algorithm by much.

### *Achieving Differential Privacy*

Differential privacy is achieved by adding noise to the evaluation of $f(X)$. A simple protocol that illustrates the usage of randomness to produce differential privacy is the following game[6]: Asked the question "Did you clean last week?", respond in the following way.

1. Flip a coin.
2. If tails respond truthfully.
3. If heads then flip the coin again and respond "yes" if head and "no" if tails.

The randomness creates plausible deniability and $\epsilon$ can be calculated from the definition of differential privacy.

$$\frac{\Pr[\text{Response} = \text{Yes}|\text{Truth} = \text{Yes}]}{\Pr[\text{Response} = \text{Yes}|\text{Truth} = \text{No}]} = \frac{3/4}{1/4} = \ln 3 = \epsilon$$

$$\frac{\Pr[\text{Response} = \text{Yes}|\text{Truth} = \text{No}]}{\Pr[\text{Response} = \text{Yes}|\text{Truth} = \text{Yes}]} = \frac{3/4}{1/4} = \ln 3 = \epsilon$$

Here the truth would correspond to the dataset and when the datasets differs in one record the same $\epsilon$ is gotten. Hence the protocol gives a guarantee of $(\epsilon, 0)$ differential privacy.

### *Sensitivity*

Consider the previous coin flip example, where function $f$ outputs the result of a query on a dataset and imagine the dataset had data from multiple participants, who each had information stored about whether they cleaned their room last week or not. The $L_1$ sensitivity of $f$ is largest change a single participant could have on the output. The amount of noise added to make $f$ differential privacy can be calibrated to how sensitive the $f$ is. The $L_1$ sensitivity of function $f : X^* \rightarrow Y$ is defined as follows:

$$\Delta f = \max_{D,D' \in X^*, ||D-D'||_1=1} ||f(D') - f(D')||_1$$

The smaller $\Delta f$ is, the less noise is needed.

*Laplace distribution*

One probability distribution that can be used to add random noise is Laplace distribution $Lap(b)$, with density function:

$$P(x|b) = \frac{1}{2b} e^{-\frac{|x|}{b}} \tag{5}$$

*Laplace mechanism*

To add noise to a function $f$, a privacy mechanism is used. Laplace Mechanism is a mechanism that adds controlled noise by scaling it the the sensitivity of the function: $b = \frac{\Delta f}{\epsilon}$. The Laplace Mechanism is defined as such:

$$M_L(x, f(\cdot), \epsilon) = f(x) + (R_1, \ldots, R_k) \tag{6}$$

where $R_i$ are i.i.d random variables from $Lap(b)$.

Laplace distribution flattens out as $\epsilon$ decreases. The smaller $\epsilon$ the more noise the and the better privacy. If the true answer to a query is 100 or 101, then the distribution on outputs look like the following figure.

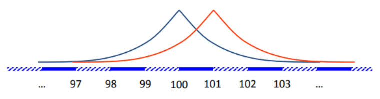

Figure 4: Output distributions with true answer 100 and 101 [5]

The Laplace mechanism preserves $(\epsilon, 0)$-differential privacy this follows from theorem 3.6 in [6].

$$\begin{aligned}
\frac{p_D(r)}{p_{D'}(r)} &= \prod_{i=1}^{k} \frac{e^{\frac{-\epsilon|f(D)_i - r_i|}{\Delta f}}}{e^{\frac{-\epsilon|f(D')_i - r_i|}{\Delta f}}} \\
&= \prod_{i=1}^{k} e^{\frac{\epsilon(|f(D')_i - r_i| - |f(D)_i - r_i|)}{\Delta f}} \\
&\leqslant \prod_{i=1}^{k} e^{\frac{\epsilon|f(D')_i - f(D)_i|}{\Delta f}} \\
&= e^{\frac{e^{\epsilon||f(D') - f(D)||_1}}{\Delta f}} \\
&\leqslant e^{\epsilon}
\end{aligned} \tag{7}$$

The first inequality comes from the triangle inequality and the last follows from the definition of sensitivity.

### $L_2$-*Sensitivity*

The Gaussian Mechanism uses the $L_2$ sensitivity of a function is to add to each of the outputs of a function[6].

$$\Delta_2 f = \max_{||D-D'||_1=1} D, D' \in X * ||f(D') - f(D')||_2 = 1 \tag{8}$$

### *Composition bounds*

Usage of an $\epsilon$-differential private mechanism $t$ times would yield in the result to be $(t\epsilon, 0)$-differential private. With $n$ independent mechanisms $M_1, ..., M_n$ with privacy $\epsilon_1, ..., \epsilon_n$ then a function $g(M_1, ..., M_n)$ is $\sum_{i=1}^{n} \epsilon_i$-differential private[10].

For any sequence of $r$ of outcomes $r_i \in (M_1, ..., M_n$ we write $M_i^r$ for the $M_i$ supplied with $r$. The probability of output $r$ from the sequence of $M_i^r(D)$ is:

$$\begin{aligned} Pr[M(D) = r] &= \prod_i Pr[M_i^r(A) = r_i] \\ &\leqslant \prod_i Pr[M_i^r(B) = r_i] \prod_i e^{\epsilon_i} \end{aligned}$$

Reconstructing the first product into $Pr[M(B) = r]$ gives the definition of $\sum_i \epsilon_i$-differential privacy.

If $f_1$ is DP and $f_2$ is DP then the union $(f_1(D), f_2(D))$ is $(\epsilon_1 + \epsilon_2) -$ DP.

# 4 ANALYSIS

## 4.1 THREAT MODEL

Throughout our experiments, as the work is largely based on [2], the threat model remains the same. This threat model is a weak one considering the protocol is working with sensitive data. The threat model consists of passive security with the addition that the none of the parties are colluding. This means that the adversaries all follow the protocol and is not allowed to modify their inputs, besides what is specified in the protocol nor will they make use of the attack from [8], since it is described as an active attack, which does not follow the protocol. As neither of the parties are colluding, this means that all parties only have their own *view* after the execution of the protocol. To formalise this, after the protocol execution each peer $i$ has the $view_i = \{\mathtt{config}, \mathtt{data}_i, g_i^1, \dots, g_i^{nT}, g_s^1, \dots, g_s^{nT}\}$, where $\mathtt{data}_i$ is all of the data the peer holds throughout the protocol, $g_i^j$ for $j \in \{1, \dots, nT\}$ is the gradient calculated by the peer on the each iteration of his data, and $g_s^j$ for $j \in \{1, \dots, nT\}$ is the summed gradients calculated by the trusted party with the added differential privacy. Whereas $n$ is the amount of rounds and $T$ is the amount of epochs. To state it clearly, each $g_i^j$ up to $g_i^n$ are calculated on disjoint data and each epoch is how many iterations are done over the full dataset. Additionally it is assumed before the protocol begins, the peers agree on parameters used throughout the protocol, meaning the sensitivity bound, the amount of epochs run, initial weights of the model, batch size and a noise scale parameter.

## 4.2 DIFFERENTIAL PRIVACY IN MACHINE LEARNING

Multiparty computation is not enough to make machine learning algorithms private, because it only promises that computation can be done jointly without revealing information about the input. It is seen that the output of the computation, namely the gradient and the trained model, can leak information about the datasets that was used to train the model. Given a model if there is a sparse amount of data on a single feature, as described in the introduction of section 3.3, the model could heavily base some correlation between the feature and classification. This could potentially expose information about individual data in the training data, which is something we want to avoid.

Hence you also need differential privacy to ensure that the output of the computed functions do not leak too much information about individuals in the dataset. If differential privacy is added to the gradient as in section 2.1, the weights would be noisy and imposing a strict sensitivity bound on the individual records s.t. each data entry does not say too much about the final model.

Differential privacy promises to protect individual records from leaking too much information and it gives participators in the dataset plausible deniability. The two different guarantees that is used in this work, is $(\epsilon, 0)$-DP and $(\epsilon, \delta)$-DP. In [6] it is stated, that the first guarantee says, the observed output is almost equally likely to be observed on a neighbouring dataset, and the second guarantee says, given an output it may be possible to find a dataset, where the output is more likely to be produced. Hence, the $(\epsilon, 0)$-DP is a stronger privacy guarantee than $(\epsilon, \delta)$-DP is, but a strong privacy guarantee comes with a trade-off, namely that the machine learning model is becoming less accurate.

*Noise distribution*

For differential privacy some controlled noise have to be added to the gradient. In [2] three noise functions are proposed, named $L_1$, $L_2$ and $L_\infty$. In this work $L_1$, and $L_2$ were quickly chosen as the $L_\infty$ performed badly in the initial tests and created results equal to random guesses in a trained model, as well as observing this distribution performing the worst in the results of [2].

The $L_1$ distribution performed the best in [2] despite having a stronger privacy guarantee, as it is $(\epsilon, 0) - DP$, and the authors also claimed they did not have a clear intuition behind their results, investigating that result and possibly trying to recreate it would be interesting to see if same results appeared. Despite not having the same setup, the overall performance of the noise distribution will still be correlated to [2], as well as compared to the $L_2$ distribution.

$L_2$ was chosen, as both the [2] paper and [**abadi**], which the work of [2] was heavily based on used. It also gave us the possibility to contrast the two distributions to each other compared to the results of [2].

The $L_1$ is the Laplace distribution with parameter $b = \frac{C}{\epsilon}$, which makes the gradient $(\epsilon, 0) - DP$ and $L_2$ is the Gaussian distribution with parameter $b = C\sqrt{2\ln(\frac{1.25}{\delta})}/\epsilon$. Both of these mechanisms were both proposed by Cynthia Dwork as described in section 3.3.

## 4.3 MACHINE LEARNING MODEL

Initially trying to replicate the results from [2] a convolutional neural network was created. This was done with the TensorFlow framework[1]. But during the implementation we ran into some obstacles. This was due to issues with extracting the gradients from the framework for each iteration of the collaborative gradient computation, which is required in the [2] paper. As a solution to the problem, a different model was selected and used to perform the collaborative gradient descent. This was a simpler model such that no framework was required to implement it, and thereby the same problem as with TensorFlow did not occur. The model chosen was the Logistic regression model, section 3.1.2, which has a few tradeoffs in terms of the the model used. Convolutional neural networks are often targeted image classification and therefor also result in very good prediction ratios on images, like in the [2] paper they get prediction precision of 98.5% and 98.9% depending on the neural network architecture used. The logistic regression model is simpler but with the trade-off of lower precision, on around 83%-85% with no noise.

The machine learning model was implemented with the one-vs-all technique. Where each of the classifiers are implemented as a two-class classifier, matching to each class. To keep the gradients disjoint and not use the same data throughout each of the two-class gradient steps, each batch was separated randomly into the 10 classes and trained, much like a variation of a stochastic gradient descent. This meant for each epoch, each gradient would still be completely independent of each other.

From Theorem 1 [2], there are two requirements to the cost function of the machine learning model. The function has to be Lipschitz smooth, and it has to be convex. As a result it will be shown that the Logistic regression have these qualities. The cost function of logistic regression is implemented as a cross-entropy cost function. In [11] it is said, that if the logistic regression is implemented with a cross-entropy cost function, then there is only one valley with regard to the gradient descent, which means the gradient descent will not be trapped in a local minimum. This is due to the cost function being a convex function of the weights.

For the cost function to be Lipschitz smooth, the following should hold for the derivative of the function $f(\cdot)$

$$|f(xy) - f(x'y')| \leqslant |xy - x'y'| \tag{9}$$

The derivative of the cost functions is as follows

$$f(xy) = -yx \cdot \theta(-yx \cdot w)$$

where $\theta(\cdot)$ outputs a number between 0 and 1. The cost function takes $w, x, y$ as input, but since $w$ is maintained, when computing the private gradients, it will be shown for two different pairs of $(x, y)$. In the case, where $\theta(\cdot)$ is returning 1, it will be shown that the equality holds.

$$\begin{aligned} |f(xy) - f(x'y')| &= |(-yx \cdot \theta(-yx \cdot w)) - (-y'x' \cdot \theta(-y'x' \cdot w))| \\ &= |(-yx \cdot 1) - (-y'x' \cdot 1)| \\ &= |-yx + y'x'| \\ &= |xy - x'y'| \end{aligned}$$

If the ouput of $\theta(\cdot)$ is less than 1, then the distance between $f(xy)$ and $f(x'y')$ will become smaller than the distance between $xy$ and $x'y'$. For the case where $w$ is updated due to multiple epochs, [3] states that the cross-entropy cost function is continuously differentiable, which means it has differentiable continuous gradients, and thereby it makes sense to assume that it is Lipschitz smooth.

## 4.4 MULTIPARTY COMPUTATION ALTERNATIVES

In the Collaborative Gradient Computation algorithm the hosts $H_1$ and $H_2$ use garbled circuits protocol to compute:

$$((\tilde{g}_1 \tilde{g}_2) \text{ smod } mC) + \text{Rand}_{s_1 \oplus s_2}(b) \tag{10}$$

where $x \text{ smod } C = ((x + C) \mod 2C)C$. This can be done by creating a boolean circuit for equation 10 and follow Yao's protocol.

HOMOMORPHIC ENCRYPTION An alternative to GB could be homomorphic encryption (HE) because it allows computation on ciphertexts and generates an encrypted result that when decrypted correspond to the result of the operations, if they were done with plain text. The randomness could be generated such that both host affect it and HE would allow for the evaluation of the function such that no party would know the randomness added.

SECRET SHARING Dwork et. al. [7] have created an efficient distributed protocol for generating random noise in a way that is secure against malicious participants. The purpose of this random noise is to add differential privacy. Privacy is added through Gaussian or Exponentially distributed noise. The distributed implementation removes the need for a trusted party.

The overall structure the ODO (Our Data Ourselves) protocol is that on a query the holders shares out this value using a *non-malleable verifiable secret sharing scheme*. Then it is verified that the values are legitimate according to that the secret sharing scheme. Then noise shares are generated cooperatively. Then all shares are summed up and obtains their share of the evaluation of the query+noise. Then the noisy sum is reconstructed using the reconstruction technique of the verifiable secret sharing scheme.

BEDOZA The computation can transform to a boolean circuit and be evaluated with the unconditional secure multiparty computation protocol BeDOZa. But it needs a trusted dealer for randomness. But as long a one has to create a boolean circuit where it is hard to do division one might as well use garbled circuits, where you are not dependent on a dealer.

# 5 IMPLEMENTATION

As a part of fully understanding and evaulating the private collaborative neural network learning, the final algorithm was implemented. In this section the implementation will be discussed, and related to the original work described in [2]. The code was implemented in python, and computations were done on a single machine. To split up the responsibilities of the parties in the code, each party were created as a separate class. Additionally the garbled circuit was implemented as a trusted party, as the garbled circuit is deterministic and would provide the same overall output of the computation as a trusted party would. For the overall evaluation the same dataset as the one described in [2] was used; the MNIST dataset. This means the input consisted of 784 dimensions i.e. 28x28 pixels in grayscale and labeled from 0 to 9. The datasize overall has 70000 data entries with matching labels. The model was trained using 60000 datapoints and evaluated with the remaining 10000, meaning the 10000 evaluation data was not used for training.

## 5.1 MACHINE LEARNING MODEL

As described in chapter 4, the overall machine learning model was changed. Instead of using a convolutional neural network a simpler model, Logistic regression, was used. The model is separated into two files *classifiers.py* and *logistic_regression.py*, one that handles the one-vs-all classification and one that handles underlying logistic regression logic.

The classifier, holds the overall model meaning the weights, as well as the operations generally done on the model. The methods for training the model are gradient_step(X,y), apply_gradient(gradient) and predict(X). The gradient_step(X,y), takes the datapoints for the batch and the accompanying labels and returns a computed gradient. To avoid using the the same point in multiple gradient computations in the one-vs-all classifiers, at each round the batch is shuffled and split into 10 parts, one for each two-class classifier. This should give a result similar to a stochastic gradient descent, in which random samples are selected during the batch computations. The implementation for this can be seen in listing 1.

```
def gradient_step(self, X, y, reg=1e-4):
    permutation = np.random.permutation(X.shape[0])
    X = X[permutation]
    y = y[permutation]
    size = len(X) // 10
    gradient_list = []
    for idx, model in enumerate(self.models):
        y_iter = y[size*idx:size*(idx+1)]
        X_iter = X[size*idx:size*(idx+1), :]
        assert len(X_iter) == len(y_iter)
        y_clasi = [1 if val == idx else 0 for val in y_iter]
        gradient = model.gradient_step(X_iter, y_clasi, reg=reg)
        gradient_list.append(gradient)
    return gradient_list
```

Listing 1: gradient_step implementation

In the apply_gradient(gradient), a gradient is applied to the weights to update the model. This means when training the model a single train step is not used, but instead is split into two different method, which is used when doing the collaborative gradient descent. Finally to review the model with the evaluation data the predict(X) takes a list of inputs X, and makes a prediction on each input.

The logistic_regression simply holds the specified model logic, meaning the specific model used for training, which in our case is the logistic regression. This is the methods for training the models. For our specific use case the cost function is not used and only the gradient is computed. This is done inside the *log_cost(X, y, w, C, order)*, where X is the data for that batch, y is the associated label, w are the current weights, C is the clipping boundary and order is which noise mechanism is used. For each data point in the batch, X, a gradient is computed, and then clipped with the with regards to the order(to to set which normalisation is used), and C. These are summed up and is the resulting gradient for the batch. The implementation of the log_cost can be seen in listing 2.

```
def log_cost(X, y, w, C, order, reg=0):
    gradients = None
    grad_reg = reg * w
    grad_reg[0] = 0
    for i, data in enumerate(X):
        grad = -(data * (y[i]-logistic(np.dot(data, w))))
        gradients = util.clip(C, grad, order) if gradients is
            None else gradients + util.clip(C, grad, order)
    gradients = gradients + grad_reg
    assert gradients.shape == w.shape
    return gradients
```

Listing 2: log_cost implementation

Where logistic simply is the sigmoid function calculated $\text{logistic}(x) = 1/(1 + np.exp(-x))$.

## 5.2 PRIVATE GRADIENT DESCENT

As mentioned previously the player has methods for performing the gradient descent, the calculate_gradient(i), which calculates a gradient on the i'th round, where the i is used to determine which disjoint batch to use. Here it calls the underlying model to calculate the gradient, after which the player calculates the secret shares based on the gradient from the model, and returns the gradient + randomness and the randomness, this can be seen in listing 3. The update gradient simply, takes a gradient parameter and updates the weights within the classifier.

```
def calculate_gradient(self, i):
        gradient = self.model.gradient_step(self.data_train[i],
            self.label_train[i])
        share_rand = [np.random.uniform(-self.mC, self.mC, len(
            arr)) for arr in gradient]
        rand_gradient = [gradient[j] + share_rand[j] for j in
            range(len(gradient))]
        return rand_gradient, share_rand
```

Listing 3: calculate_gradient implementation

As for the loop running the private gradient descent. As it has no hosts to do computations and given as it is only a single peer who is allowed to know the gradient in clear, the randomness is subtracted from the gradient with the randomness. The randomness from differential privacy is then added to the gradient and the update_gradient(gradient), method is called to update the gradient. This can be seen in listing 4

```
def single_gradient_descent(peer, config):
    C = config['C']
    epsilon = config['epsilon']
    delta = config['delta']
    order = config['order']

    for _ in range(config['epochs']):
        for i in range(config['r']):
            rand_gradient, randomness = peer.calculate_gradient(i
                )
            gradient = [arr-randomness[idx] for idx, arr in
                enumerate(rand_gradient)]
            size = len(gradient[0])
            b = util.calculate_b(C, epsilon, delta, size, order)
            rand = [util.randomness(size, b, order) for _ in
                range(len(gradient))]
            peer.update_gradient([gradient[j] + rand[j] for j in
                range(len(gradient))])
    return peer
```

Listing 4: private_descent implementation

## 5.3 COLLABORATIVE GRADIENT DESCENT

For the collaborative aspect, we still use the same underlying player, as described in the private gradient descent. But still need two hosts as described in the collaborative gradient descent algorithm from [2]. These host take gradients from each peer and sum them together, while one simply gets a random value of the same dimension as the gradient, and the other gets the random value added to the gradient. From the hosts view these gradients have the exact same form, and are indistinguishable, a host class is created which simply sums up the gradients received by the peers. After summing the gradients, the symmetric mode operation is applied to the sum, and the host returns the result.

The collaborative gradient descent implementation consists of an outer loop for the amount of epochs, and an inner round for the batch used. For each round each peer calculates r, and g+r on the disjoint batches, which are given to the respective hosts. These are summed up and passed to the trusted party, which calculates ((g+r)-r) + noise, depending on the distribution set in the config. For each peer they update their model with the trusted calculated gradient. The implementation can be seen in listing 5.

```
def collaborative_gradient_descent(peer_list, config):
    print("running collaborative gradient descent")
    host_1 = Host(config)
    host_2 = Host(config)
    for _ in range(config['epochs']):
        for i in range(config['r']):
            for peer in peer_list:
                rand_gradient, randomness = peer.
                    calculate_gradient(i)
                host_1.feed_gradient(rand_gradient)
                host_2.feed_gradient(randomness)
            gradient_sum = host_1.calculate_gradient()
            randomness_sum = host_2.calculate_gradient()
            trusted_gradient = trusted_gradient_calculation(
                gradient_sum, randomness_sum, config)
            for peer in peer_list:
                peer.update_gradient(trusted_gradient)
        print(".")
    return peer_list
```

Listing 5: collab_descent implementation

# 6 EVALUATION

In this chapter evaluation of the differential private machine models are run.

The algorithms are tested on the MNIST dataset that contains images of hand-written digits. Despite expecting the same results on both the private gradient descent and the collaborative gradient descent, both algorithms were tested out to verify the results. This does not mean that the results are exactly the same as there are randomness mixed into both of them, both through shuffling the data as well as the noise added at each round. Before running both the algorithms a parameter sweep was performed to find good values for the learning rate and batch sizes. After some experimentation the learning that was used in the experiments was 0.3, and the batch size was 20000, the effect of the batch size is that the smaller batch sizes the more noise will be added on each round. While smaller batch sizes lets the algorithm take more steps in the gradient descent. From preliminary tests, higher batch size resulted in a better precision. This is most likely also due to having multiple epochs, in which the same values as the ones in [2] was used; 50 epochs. When the gradient descent is computed with the L2 norm, the privacy guarantee is $(\epsilon, \delta)$-DP, where $\delta = 10^{-3}$.

## 6.1 PRIVATE GRADIENT DESCENT

For the private gradient descent algorithm it should be noted, that the precision without any noise is at 85.59%.

In table 3 the varying precision for the L1 norm is observed, when changing $\epsilon$, and thereby the privacy guarantee. It is noted that the best precision is at 82.41% with $\epsilon$ at 0.7, but the precision is converging around an $\epsilon$ at 0.4.

| $\epsilon$ | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
|---|---|---|---|---|---|
| Precision | 60.25% | 70.80% | 76.67% | 78.20% | 77.18% |
| $\epsilon$ | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
| Precision | 81.48% | 82.41% | 79.39% | 82.18% | 82.33% |

Table 3: Private gradient descent for L1

In table 4 the varying precision for the L2 norm is observed, when changing $\epsilon$. It should be noted that when $\epsilon$ is 0.16, then the model has the best observed precision, but the precision starts to converge when $\epsilon$ is 0.12.

| $\epsilon$ | 0.02 | 0.04 | 0.06 | 0.08 | 0.10 |
|---|---|---|---|---|---|
| Precision | 53.18% | 65.93% | 74.17% | 75.53% | 77.54% |
| $\epsilon$ | 0.12 | 0.14 | 0.16 | 0.18 | 0.20 |
| Precision | 80.58% | 80.08% | 83.99% | 82.82% | 82.19% |

Table 4: Private gradient descent for L2

For the private gradient descent algorithm, it is noted that the best precision from L1 and L2 is nearly the same, but the values for $\epsilon$ is very different. The $\epsilon$-values for L2 is five times smaller than the $\epsilon$-values for L1. This is due to L2 having a weaker privacy guarantee, since it also uses $\delta$.

## 6.2 COLLABORATIVE GRADIENT DESCENT

For the collaborative gradient descent algorithm it should be noted that the precision without any noise is at 83.56%.

In 5 it should be noted, that the precision increases as $\epsilon$ increases, but it converges when $\epsilon$ is 0.5. The best precision observed for this algorithm is 82.76% and it is when $\epsilon$ is 0.72.

| $\epsilon$ | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
|---|---|---|---|---|---|
| Precision | 58.28% | 69.76% | 74.52% | 77.88% | 79.98% |
| $\epsilon$ | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
| Precision | 82.04% | 82.76% | 81.84% | 80.40% | 79.48% |

Table 5: Collaborative gradient descent for L1

In 6 it should be noted, that the best precision is 82.25% and it is when $\epsilon$ is 0.14. The precision starts to converge, when $\epsilon$ is 0.10.

| $\epsilon$ | 0.02 | 0.04 | 0.06 | 0.08 | 0.10 |
|---|---|---|---|---|---|
| Precision | 55.26% | 67.97% | 74.53% | 73.86% | 79.28% |
| $\epsilon$ | 0.12 | 0.14 | 0.16 | 0.18 | 0.20 |
| Precision | 81.68% | 82.25% | 79.82% | 80.94% | 81.01% |

Table 6: Collaborative gradient descent for L2

For the collaborative gradient descent algorithm, it should be noted, as for the private gradient descent, that the best observed precision from L1 and L2 is nearly the same. And that the $\epsilon$-values is five times smaller in L2 than in L1, since L2 having a weaker privacy guarantee.

## 6.3 ILLUSTRATIONS

To give a better illustration of the results and the effects on the final weights of the model, some illustrations of the above described results will be shown. In figure 5 and figure 6, four curves are seen. The blue curve is the collaborative gradient descent with noise added, the red curve is the private gradient descent with noise added, the orange curve is the collaborative gradient descent without noise added, and the green curve is the private gradient descent without noise added.

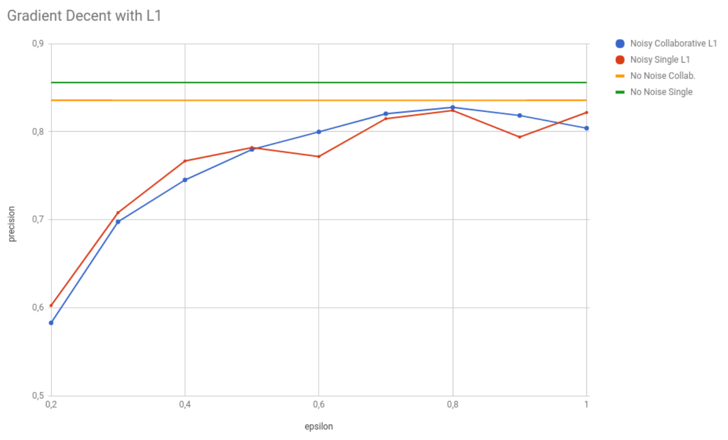

Figure 5: Precision of the trained models using L1

In both figure 5 and figure 6 it is noted, that the collaborative gradient descent and the private gradient descent is much alike in their results, due to the construction of the collaborative gradient descent algorithm.

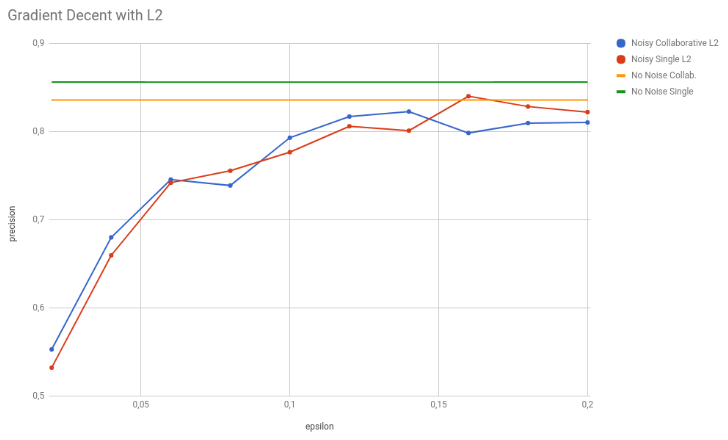

Figure 6: Precision of the trained models using L2

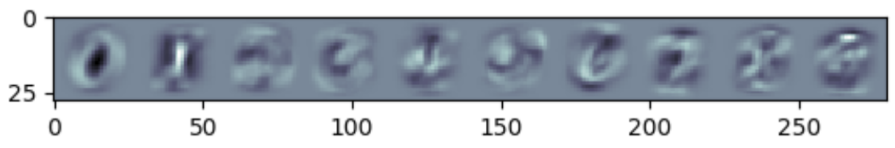

Figure 7: Hand-drawn digits with collaborative gradient descent, no noise and and 83.56% precision

In figure 7 the weight of the model is visualised and it created the shape of the 10 digits that the model have learned.

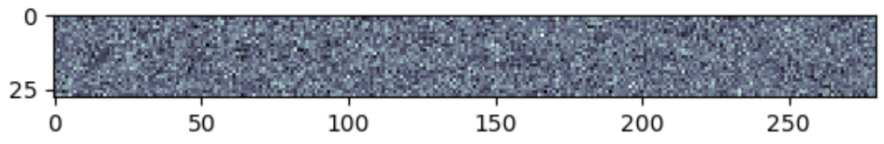

Figure 8: Hand-drawn digits with collaborative gradient descent, differential privacy with $\epsilon = 0.2$ and and 69.76% precision

In figure 8 the human eye sees nothing but noise, and yet the machine learning model can with high probability tell the 10 different digits apart.

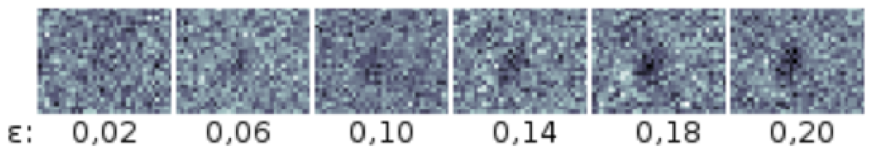

Figure 9: The development of digit 0 as epsilon increases

In figure 9 it is seen that as $\epsilon$ and the noise decrease the shape of the digit 0 slowly emerges.

## 6.4 SUMMARY

As expected the result of the collaborative gradient descent computation is very much like the private gradient descent computation, which is expected based on how the collaborative gradient descent computation. In contrast to the results from [2], which are seen table 1 and 2, the $L_1$ distribution did not do better than the $L_2$. This is expected since the $L_1$ provides better differential privacy guarantees. Overall from the result it seems that it is possible to do machine learning while keeping the data private. But the tighter the parameter is, the more data is required to train a useful model. In this case the dataset consisted of only 60000, and had 784 features, while having more data and less features. This could give more potential to train a differential private model without running as many epochs.

# 7 CONCLUSION

In this work, it was investigated how machine learning and differential privacy is working together. For the investigation a protocol called *Collaborative Gradient Descent* was implemented. This protocol was introduced earlier this year in [2]. The goal of the implemented protocol was to gain a better understanding of differential privacy in machine learning, and to see if similar results could be obtained. As problems were encountered during the development process, a change in machine learning model was necessary, and a simpler model was used. This of course meant that replicating [2] entirely was not an option.

Using the logistic regression algorithm, instead of a convolutional neural network, good results was still provided in terms of evaluating the loss of precision with the added tighter differential privacy bounds. Overall our trained model resulted in much lower $\epsilon$ values, while still maintaining a good amount of privacy. Also compared to the results in [2], the $L_2$ mechanism performed much better than $L_1$ in terms of lower epsilons, while maintaining a high precision, though the $L_2$ has a weaker privacy guarantee, due to the use of $\delta$. In contrast to the results of [2], where $L_1$ outperformed the other two distributions to very high degree, as seen in the table 1 and 2.

In this work some attacks against neural network on privacy was looked into both with an active adversary on gradients 2.2 and on trained models 2.3. While these attacks were not evaluated against the collaborative gradient descent, they give some insight into the privacy issues regarding collaborative trained machine learning model. Evaluating these attacks against [2] will be left as future work.

We have overall learned a lot about differential privacy techniques and their influence in machine learning as a way of making machine learning models more private. It is a popular topic currently and with the huge increase in machine learning, adding privacy when working with sensitive data is highly relevant.

# 8 FUTURE WORK

Currently a lot of research has gone into working with differential privacy in machine learning, this means a lot of interesting new algorithms and ways of applying differential privacy to the models are appearing. In this work we focused very narrowly [2], and evaluating their results with regards to a new model. An obvious choice is of course mimicking their results entirely by also using a convolutional neural network and verifying their results as we had the difference that the $L_2$ distribution performed much better compared to the $L_1$ distribution. This means either picking up a new framework for efficiently creating neural networks, that has an easier way of extracting the gradients or coding a full convolutional neural network from scratch.

As this is a suggested solution to a passive attack, it would be very interesting to find privacy preserving machine learning approaches to active attacks and evaluate those and get a wider view of the different approaches to computing privacy preseving machine learning.

As the active attack [8] still pose questions towards the efficiency of their attack on device-level or user-level differential privacy on how it performs. It could be interesting to evaluate how it performs against the recent approach by apple[4], who recently announced how it uses for user-level differential privacy. As the paper has just recently been released, at the time of writing, it has not been reviewed in depth by us yet.

As differential privacy is a very broad and active area not only in machine learning but also very active in database topics, investigating more distributions and mechanisms to achieve differential privacy could also be very interesting to see if they can be applied to machine learning algorithms to greater success.

Despite not being the primary focus of this work, reviewing more methods of doing multiparty computation or two party computation to securely calculate the noise from differential privacy while summing the gradients in a more efficient could also be a interesting topic to look into for future work.

# BIBLIOGRAPHY

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Melissa Chase, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, and Peter Rindal. "Private Collaborative Neural Network Learning." In: 2017. URL: https://eprint.iacr.org/2017/762.pdf.

[3] Tianyi Chen. "Convergence Analysis on Neural Network." In: 2017. URL: https://www.researchgate.net/profile/Tianyi_Chen/publication/321295526_Convergence_Analysis_on_Neural_Network/links/5a19cdada6fdcc50adeae849/Convergence-Analysis-on-Neural-Network.pdf.

[4] Apple Differential Privacy Team. "Learning with privacy at scale." In: 2017. URL: https://machinelearning.apple.com/docs/learning-with-privacy-at-scale/appledifferentialprivacysystem.pdf.

[5] Cynthia Dwork. "A Firm Foundation for Private Data Analysis." In: *Communications of the ACM* (2011). URL: https://www.microsoft.com/en-us/research/publication/a-firm-foundation-for-private-data-analysis/.

[6] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy." In: (Aug. 2014). URL: https://www.cis.upenn.edu/~aaroth/Papers/privacybook.pdf.

[7] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. "Our Data, Ourselves: Privacy Via Distributed Noise Generation." In: *Advances in Cryptology (EUROCRYPT 2006)*. Springer Verlag, 2006. URL: https://www.microsoft.com/en-us/research/publication/our-data-ourselves-privacy-via-distributed-noise-generation/.

[8] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. "Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning." In: 2017. URL: http://www.freepastry.org.

[9] Andrej Kapathy. "Neural Networks." In: Stanford University. URL: http://cs231n.github.io/neural-networks-1/.

[10] Frank McSherry. "Privacy Integrated Queries." In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, Inc., 2009. URL: https://www.microsoft.com/en-us/research/publication/privacy-integrated-queries/.

[11] Yaser S. Abu Mostafa, Malik Magdon Ismail, and Hsuan Tien Lin. "Learning From Data." In: 2012.

[12] Claudio Orlandi. "Lecture Note for Cryptographic Computing Week 4." In: Aarhus University, 2016. URL: https://blackboard.au.dk/bbcswebdav/pid-1070550-dt-content-rid-1387451_1/courses/BB-Cou-UUVA-54625/Week4%281%29.pdf.

[13] Claudio Orlandi. "Lecture Note for Cryptographic Computing Week 6." In: Aarhus University, 2016. URL: https://blackboard.au.dk/bbcswebdav/pid-1070550-dt-content-rid-1439093_1/courses/BB-Cou-UUVA-54625/Week6%282%29%281%29.pdf.

[14] Congzheng Song Reza Shokri Marco Stronati and Vitaly Shmatikov. "Membership Inference Attacks Against Machine Learning Models." In: 2017. URL: https://arxiv.org/pdf/1610.05820.pdf.

[15] Andrew C Yao. "Protocols for secure computations." In: 1982. URL: https://dl.acm.org/citation.cfm?id=1382751.

Arnar Thordur Hardarson , Christian Koller Nielsen , Steffen Skovsgaard Jensen : *Privacy Preserving Machine Learning,* Course Project, © April 2020