University of Amsterdam
April 2018

Privately Training CNNs using Two-Party SPDZ

Ruben Seggers & Koen van der Veen
Supervisor: Christian Schaffner

Abstract. The success of deep learning algorithms is greatly attributed to the increased availability of large datasets. However, there exist many tasks where privacy concerns prevent data sharing. Some efforts to solve this problem use secure multi-party computation, with the cost of extra computation and communication. A recently developed multi-party computation protocol is SPDZ. In this work, deep convolutional neural nets are trained in a simulated SPDZ environment. Various methods are proposed to enable training, as SPDZ does not support all necessary functionalities used in typical CNNs. Training is evaluated on computation and communication overhead and the trained models are compared to their non-private equivalents. The proposed models achieve similar accuracy and training is successfully optimized to minimize communication overhead, increasing the feasibility of training deeper architectures.

1 Introduction

Convolutional Neural Networks (CNNs) achieve state of the art performance in various image and other machine learning tasks. Large amounts of data need to be gathered to train these networks. However, there exist many tasks where the need for privacy complicates or prevents data sharing, e.g. in the medical discourse. In recent research this problem is addressed using secure Multi Party Computation (MPC) protocols to share computations over multiple parties without needing to share the actual data. Various MPC protocols designed for arithmetic circuit evaluation support efficient addition and multiplication methods. However, nonlinear functions are very expensive to compute, preventing the use of e.g. the ReLU activation function. Therefore, activation functions and loss functions have to be approximated by continuous polynomials. This decreases the efficiency of training and the performance of the models. It should be noted that the privacy vulnerabilities due to the data learning capabilities of the models themselves are not addressed using MPC; to this end differential privacy techniques are currently the state of the art solution. However, this is out of the scope of this research.

2 Related work

Efforts to train machine-learning models without the need for sharing data typically use one of the following approaches: Either i) the data is encrypted using Homomorphic Encryption (HE) and

sent to a server to train models; Or ii) the the model is trained jointly while keeping the inputs private using an MPC protocol.

2.1 HE approach

An example of a CNN trained using HE is CryptoNets [6]. CryptoNets shows high accuracy and throughput. However, it does lose on accuracy in deeper architectures due to the vanishing gradients problem caused by the use of the sigmoid activation function. This problem is addressed in subsequent work by approximating the ReLU by polynomials that can be of low order by approximation on a small interval, combined with batch normalization [4]. CryptoDL explores this idea even further, but focuses on HE inference: The model is trained on a plain text public dataset. Afterwards, predictions are generated on encrypted data. These predictions can only be decrypted by the data owner [8]. For inference CryptoDL shows state of the art performance on inference in both accuracy and efficiency.

Deep learning using HE shows the best accuracy efficiency trade-off. However, as models are trained on encrypted data, this approach will only work on data that is encrypted with the same key used during training. Therefore, either the model can only be used by the data owner, or the key has to be shared with the model owner, meaning the data can be decrypted by the model owner, mitigating all benefits gained with encryption in the first place. In other words, this approach is only feasible for a limited scenario with only one data owner who also is the only one who can use the trained model.

2.2 MPC approach

MPC is well established subfield of cryptography research. However, using MPC for privacy-preserving machine learning is a novel approach and considerably less researched compared to HE. MPC protocols typically use some or all of the following techniques: Circuit Garbling (GC), secret sharing and Oblivious Transfer (OT).

SecureML is a learning scheme in which an NN is trained using MPC [10]. The MPC implementation of SecureML makes use of both GC and OT, used to evaluate the ReLU activation function. For weight matrix multiplications, techniques similar to SPDZ are used, which is explained below. The only results reported are the accuracy achieved by a relatively shallow network without convolutional layers. As there is no information on CPU time, communication cost and synchronization cost, it is hard to judge how their approach compares to others. Furthermore, their implementation is based on a use-case where two non-colluding servers share the user data, which requires trust. SecureML proves security against a semi-honest adversary.

DeepSecure is based on Yao's GC protocol and enables deep learning with multiple decentralized data owners and a single model owner [13]. This variety of GC requires more computation, as all functions are rewritten to binary circuits. It does yield the advantage that there is no need for communication between layers; only the inputs and outputs have to be communicated. DeepSecure protects against honest-but-curious adversaries.

2.3 SPDZ

SPDZ is an MPC protocol aimed at arithmetic circuit evaluation based on secret sharing. The SPDZ MPC software package developed by the University of Bristol is based mainly on [2] and [12] for secret sharing and message authentication codes, and [11] to perform preprocessing aimed at reducing the communication during the actual execution. The latter is achieved in an offline phase where so called triples are created and pre-shared that can be used during the execution of a desired computation. The communication cost when using SPDZ is relatively high. However, the computational cost during execution is low, as SPDZ is specifically designed for arithmetic circuit evaluation. Furthermore, the protocol is suited for a generic setup similar to DeepSecure. Moreover, where DeepSecure offers protection against honest-but-curious adversaries, SPDZ is UC-secure

against a dishonest majority. To the best of our knowledge there exist no scientific publication on using SPDZ for machine learning besides SecureML, which has limitations as described in the previous section. However, Morten Dahl wrote a blog post on this approach together with a proof of concept implementation [1].

Shared arithmetic circuit evaluation in a finite field
In SPDZ, participants secret share all inputs among all parties to evaluate an arithmetic circuit. In the case of machine learning, this translates to training a shared model on shared data. To enable sharing, a private value x is split in a privacy-preserving manner into N pieces x_i, for which holds $\sum_i x_i = x$. The mechanism behind the generation of secret shares, or the generation of triples in the offline phase, is not described in this report. However, it is important to understand how secret shared values and triples are used to do joint computation, as they are extended to achieve efficient training of a CNN.

When a computation is performed on shared values, the participants each perform an applicable local computation on their shares of the involved values. The notation of shared computation is as follows: when the desired global computation would be e.g.: $x+y = z$, the values x and y are shared to create secret-shared values $[x]$ and $[y]$. The intended MPC computation is now represented as: $[x]+[y] = [z]$. Each individual i performs operations on x_i and y_i to obtain z_i. To obtain the value of z from $[z]$, the participants broadcast their shares z_i to a designated location, where summing over the shares yields the actual result.

The two main operations defined in SPDZ are addition and multiplication. Addition is performed by simply adding the shares locally, i.e. the shared computation $[x] + [y] = [z]$ is achieved by local computation of $x_i + y_i = z_i$. Multiplication is more complex: To compute $[x] \cdot [y] = [z]$ a triple (a,b,c) is constructed for which holds: $a \cdot b = c$. This triple is secret shared such that every participant can compute a masked version of their shares $\alpha_i = (x_i - a_i)$, and $\beta_i = (y_i - b_i)$. Subsequently, all individuals broadcast their masked shares α_i and β_i. Now α and β are publicly available to every user, z_i can be computed by:

$$\begin{aligned} z_i &= c_i + \alpha b_i + \beta a_i \\ &= c_i + (x - a)b_i + (y - b)a_i \end{aligned}$$

To retrieve the actual value of z_i, an extra term $\alpha\beta$ has to be added to the sum over z_i:

$$\begin{aligned} z &= \alpha\beta + \sum_i z_i = \alpha\beta + c + \alpha b + \beta a \\ &= c + (x - a)b + (y - b)a + (x - a)(y - b) = xy \end{aligned}$$

Computations in NNs are performed on real numbers represented as floats. In SPDZ, real numbers have to be encoded as either fixed point or floating point representations. The former is the more common approach, where the real numbers are scaled by a certain precision value and subsequently rounded to an integer:

$$x_{encoded} = \text{round}(x_{real} * 10^{precision}) \mod Q,$$

where modulo Q defines a finite field and Q is some large prime number. The precision should be low enough compared to Q, such that the maximum value of x does not 'wrap around' when encoded. When two values in the field are multiplied, the maximum precision required by the resulting value is doubled. Therefore, truncation is used to reduce the precision of the result after every multiplication [3].

The lion's share of computation in training NNs are matrix multiplications, which consist of scalar multiplications and additions, both readily available in SPDZ. However, various other

common computations are non-linear functions, e.g. the widely used ReLU activation function. These functions cannot be evaluated as an arithmetic circuit, and thus not using SPDZ. A common approach to address this shortcoming is to approximate these functions using polynomials, as they only require multiplication and addition; This is also the approach used in this research.

3 Proposed architecture

The goal of this project is to implement all necessary components to train a CNN on SPDZ to enable training that is UC-secure against a dishonest majority. This solution will build on existing components of NNs on SPDZ already developed, as described below. The proposed method enables two-party MPC between two non-colluding servers, one with data and one with a model; as well as a hub-spoke setup, where the model owner has multiple two-party MPC connections with independent data owners. In the former setup, there remains a choice in whether the model share is broahttps://www.sharelatex.com/project/5ab2803fe2f64a7a8b625b8ddcast from the data-holding server to the model owner making it fully available, or to keep the model share such that the model owner can only make predictions when the data server allows it to do so.

3.1 Existing components

The idea to apply SPDZ to machine learning is not novel. A recent informal research blog post describes how a shared model can be trained by two non-colluding parties on shared data [1]. Table 1 shows the architecture of this implementation; the encoding column lists which components make use of SPDZ, i.e. whether components are privately or publicly evaluated.

Table 1: Model architecture of the existing implementation.

Layer	Encoding
conv2d (32, (3,3))	Public
sigmoid	Public
conv2d (32, 3,3))	Public
sigmoid	Public
average_pooling2d (2,2)	Public
dropout	Private
dense(128, 6272)	Private
dropout	Private
dense(5, 128)	Private
softmax	Public
cross-entropy loss	Private

Convolutional layers can be pre-trained: a model is trained on a certain task and dataset. Subsequently, the weights of the convolutional layers are used when training another model. This method is known as *transfer learning*, as knowledge gained in training one model is transferred to another model. Often, these pre-trained weights are used as a starting point and fine-tuned by training on the new task. Note that the first layers of the architecture are public. Here, the convolution layers are fixed, i.e. not fine-tuned on the new task. In this setting, the convolution layers are used to extract features from the data. These features and the remaining part of the model are shared among two servers and a shared output is produced. This output is broadcast and used to compute the final softmax layer and cross-entropy loss.

The private components of this architecture make use of two main operations: the dot product (for dense layers) and the sigmoid function (for activation layers). Dot products consist of multiplications and additions, which are supported by SPDZ. However, a naive implementation of the dot product masks every value x in a matrix multiple times, resulting in a high communication cost.

To prevent this, an optimized triple of matrices (A,B,C) can be used such that $A \cdot B = C$. For two matrices A and B of size $M \times N$ and $N \times L$ this reduces the number of communicated values from $M \times N \times L$ to $M \times N + N \times L$ [1]. For a simple dense layer in a neural network with 100 input nodes and 10 output nodes and a batch of 128 examples this results in $128 \cdot 100 + 100 \cdot 10 = 13,800$ communicated values instead of $128 \cdot 100 \cdot 10 = 128,000$ communicated values, which is a reduction in communication by nearly a factor 10.

The sigmoid function contains exponentiation, which is not defined in SPDZ. Therefore, the sigmoid function is approximated using a polynomial. The used polynomial is of order 9, where only the uneven powers are used as they influence the precision of approximation more than the even powers, and using all powers would drastically increase communication cost.

In this model, the shares of the output of the dense layer are revealed before the softmax function is applied. This is a substantial privacy leak as at least one of the two parties now sees the predictions of the network on the data. However, if these outputs are only revealed to the data owner, the privacy leak is relatively small, since the outputs of the model do not reveal much about its parameters. After the softmax function, the outputs are combined with the privately shared labels in the cross-entropy loss function, yielding a shared output.

To be able to test the feasibility of the setup proposed in this project, the experimental setup from [1] is used. This implementation simulates SPDZ by splitting the values of a tensor into shares and storing them in separate attributes of a tensor class, i.e. there will be a simulated shared computation rather than actual separate parties communicating over a network. There are three different types of tensor classes defined to compare experimental results: a native tensor, used to do tensor operations as usual on 32 bit floats; a public encoded tensor, used to do tensor operations in a finite field on 128 bit integers used as fixed-point representation of real numbers; and a private encoded tensor, used to emulate the secret-sharing mechanism from SPDZ, also using 128 bit integers. All tensors are implemented as NumPy classes to ensure fair comparison. The public encoded tensor shows the effects of training with integers in a finite field instead of floats; The private tensor shows the effects on training using SPDZ.

3.2 Desirable extensions

As stated above, the existing implementation does not allow private training of convolutional layers, i.e. the convolutional layers can only be used as a feature extractor and cannot be fine-tuned for the task at hand. Furthermore, for some tasks there is no large similar public dataset available and weights of the convolutional layers have to be trained from scratch, increasing the need to be able to privately train these. When using convolutional layers, it is common to also decrease the number of parameters by using max pooling, greatly reducing training time. Including these convolutional and pooling layers results in a deeper architecture, which gives rise to the vanishing gradients problem: As the sigmoid activation function downscales the values between every two layers, many values decrease to zero. The most common approach to tackle this problem, is to use the ReLU activation layer, which simply sets all negative values to zero and leaves positive values unchanged. Especially combined with convolutional layers, the ReLU has become a powerful standard in machine learning.

3.3 Implementation

To fully privately train a CNN on SPDZ the existing implementation has to be extended with SPDZ implementations of: i) Convolutional layer; ii) Average pooling layer; iii) ReLU activation function; And iv) loss function.

i) Convolutional layer
A convolution layer computes the output of sliding a set of convolution filters over an input space. One implementation of the convolutional layer extracts patches from the input, subsequently flat-

tening and concatenating these patches into a single matrix. The weights of the filters are flattened and transposed accordingly. This technique enables an efficient computation of the convolution layer by a single dot product between the two resulting matrices. This process of extracting patches from the input into a single matrix is called the im2col operation [5].

ii) Average pooling layer
Most CNNs use max pooling to down-sample the resolution of feature maps, where for every $M \times N$ values the maximum value is selected. The max pooling operation is a non-linear function, which is not compatible with SPDZ. A popular alternative for max pooling is average pooling, where the average value of every $M \times N$ pixels is selected using a sum and a division by a public constant.

iii) ReLU activation function
As for the max pooling, the ReLU function is non-linear and not compatible with SPDZ. A solution is to approximate this function with a polynomial. Where the regular ReLU simply sets all negative values to zeros, a rather cheap operation, a polynomial requires computation of several powers of x according to the order of the polynomial, and a dot product with the public coefficients of the approximation. Preliminary experiments showed that higher orders only slightly increase model performance (i.e. prediction accuracy), for a high price in computation and communication cost. Figure 1 shows the approximations of orders 3 and 9 next to the actual (exact) ReLU function in the relevant domain [-1,1].

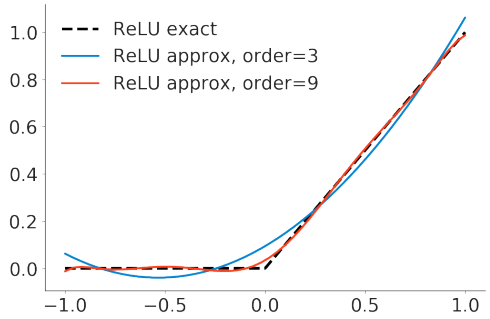

Fig. 1: Approximation of the ReLU activation function.

iv) Loss function
The cross-entropy loss function is commonly used for classification tasks, as it conveniently yields outputs that sum to 1, so the outputs are interpretable as probabilities. However, as it contains exponentiation which is not available in SPDZ, again a solution has to be found. Unfortunately, approximating an exponent with a polynomial is hard. Both other loss functions or other approximation methods can be used to solve this problem. Due to time constraints, these solutions are not evaluated in this research. To be able to run simulations, the output of the model is broadcast and the softmax is computed publicly. The cross-entropy loss is computed privately. Indeed, this yields a privacy risk and solutions have to be considered in future work.

3.4 Optimization

Many frameworks exist for machine learning, for a great deal to enable efficient computation using CPU/GPU instructions that massively speed up training. However, as the encoded tensors

make use of integers in a finite field instead of regular floats, these optimization cannot be used directly. Therefore, other optimization techniques can drastically improve CPU time of training. Furthermore, using specialized triples for several matrix operations can decrease the amount of information that has to be communicated significantly.

CPU time optimization
For the convolutional layer the col2im and im2col operations are used. As these operations are the most expensive parts of the network, they are performed in Cython, which compiles the operations on the NumPy tensors to C code.

Communication-complexity optimization
As in the communication optimization for dot product and convolutions, the main purpose of specialized triples is that every value that is to be kept private is only masked once. The specialized triple for the dot product is used in all experiments. Furthermore, specialized triples are developed for several operations as further described below.

As mentioned earlier, the convolutional layer uses the im2col operation to extract patches from the input. As patches are often overlapping, the resulting matrix of the im2col operation often contains many duplicates of the input. When specialized dot-product triples are used, this still incurs an extra communication cost as the unique values in the input are masked more than once. Therefore, a convolution triple (A,B,C) is proposed, for which holds: $im2col(A) \cdot B = C$. Using this triple, no input value is masked more than once and a minimal communication cost is achieved. For the backward phase of a convolutional layer, a similar problem holds. The im2col operation is again applied to the input of the layer, before a dot product with the backpropagated gradient matrix resulting in the weight-update matrix. Therefore, a similar triple is introduced for this operation for which holds $im2col(A) \cdot B = C$, which again results in minimal communication cost.

When the ReLU function is approximated by a polynomial, x^2 is evaluated. As the value of x is multiplied by itself, it can be masked twice with the same value, resulting in less communication. The corresponding triple is $A \cdot A = B$.

In the backpropagation phase through the average-pooling layer, values in the backpropagated gradient on the output side of the layer are copied into the backpropagated gradient on the input side of the layer. Subsequently, these values are multiplied with the input and weight matrices of the previous (conv) layer to compute the weight update and backpropagation matrices. However, by masking the backpropagated gradients before copying communication can be reduced. Two specialized triples are introduced: For the weight update $convbackward(A) \cdot im2col(B) = C$. And for the backpropagated gradient $convbackward(A) \cdot B = C$

Communication complexity optimization: re-using of masks
The last optimization considers the re-using of masks. Whenever there exists a matrix which occurs in two operations: $[x] \cdot [y] = [z]$ and $[x] \cdot [q] = [w]$ the values that are used to mask x can be re-used. This occurs in many places across the network. In the forward phase this is used for the ReLU approximations as to evaluate $x^2 = x \cdot x$ and $x^3 = x^2 \cdot x$, where x is used twice.

For the backward phase this idea can be exploited even further: In the forward phase, each layer with weights masks the input x and the weights w to evaluate the output. In the backward phase, the weight update and backpropagated gradient are computed by matrix multiplications of the incoming backpropagated gradients with the layer input x and layer weights w, from which the masks can be re-used. As the backpropagated gradient is also used twice, in the second operation, its mask can be re-used also. Finally, for the backward phase of the ReLU, the incoming gradient is multiplied by the derivative of the ReLU approximation. This derivative contains all but the last powers of the original polynomial approximating the ReLU function, which already have been evaluated and can therefore be re-used.

4 Experiments and results

4.1 Setup

For all experiments one or both of the following two architectures are used: one with a single-convolutional layer as described in Table 2, and one with double-convolutional layers as described in Table 3. For brevity, these will also be referred to as the shallow convnet and deep convnet.

The models are trained with the following details: The models are trained on batches of size 128 of the MNIST dataset with plain stochastic gradient descent (SGD) without momentum and a learning rate of 0.01 [9]. For the softmax function the log-sum-exp function is used, where the maximum activation value in a layer is subtracted from all activation values in order to prevent exploding gradients [7]. In the backward pass through the network, the backpropagated gradient with respect to the input through the first layer of the model is not computed, since it is never used and requires extra communication for private tensors. The ReLU activation function is placed after the average pooling, as preliminary experiments showed that this does not yield a significant decrease in performance in this specific setup, while reducing communication cost and CPU time. A design choice specific to the deep architecture is the pooling layer between the two convolution layers, decreasing both CPU time and communication cost. As there is no private solution implemented for the loss function, the non-scaled output before the softmax is broadcast, and the output is privately shared during the computation of the loss.

The hardware used for the experiments is an Amazon EC2 c5.xlarge server with 4 cores and 8GB RAM. All software is written in Python, making use of the NumPy library when appropriate and Cython for the im2col and col2im operations.

Various parts of the system will be evaluated on one or more of the following measures: i) Model performance: the accuracy of the final model; ii) Time complexity: The CPU time it takes to train the neural net; and iii) Communication complexity: The number of values sent around the network.

Table 2: Single convlayer architecture (shallow convnet).

Layer	Encoding
conv2d (32, (3,3))	Private
average_pooling2d (2,2)	Private
ReLU (approx)	Private
dense(10, 6272)	Private
softmax	Public
cross-entropy loss	Private

Table 3: Double convlayer architecture (deep convnet).

Layer	Encoding
conv2d (32, (3,3))	Private
average_pooling2d (2,2)	Private
ReLU (approx)	Private
conv2d (32, (3,3))	Private
average_pooling2d (2,2)	Private
ReLU (approx)	Private
dense(10, 1568)	Private
softmax	Public
cross-entropy loss	Private

4.2 Experiment 1: Approximating the ReLU activation function

This experiment aims to show whether a reasonable model performance can be achieved using a low-order (3) polynomial approximation of the ReLU activation function. Figure 2 shows the training accuracy, which is smoothed by taking the average training accuracy per every 10 training steps. Figure 3 shows the accuracy on the test set after every epoch. The final test accuracy does not differ for the exact and approximated ReLU functions. However, for the single convlayer architecture it takes 2 epochs instead of 1 epoch to converge; and for the two convlayer architecture it takes 4 instead of 2 epochs. It is, of course, suboptimal to train longer, especially when there is

an extra communication cost to consider. However, for many tasks the advantage of being able to train on private data could outweigh this extra cost. Furthermore, it should be noted that the final accuracy could suffer from the ReLU approximation when introducing other model-performance-increasing measures.

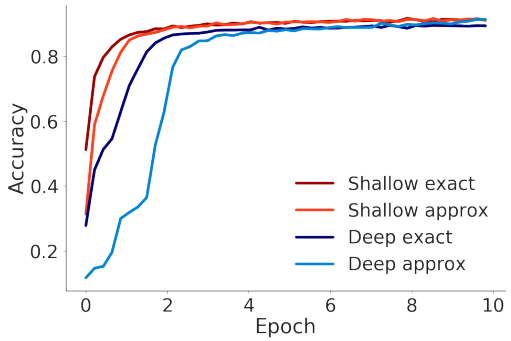

Fig. 2: Train accuracy.

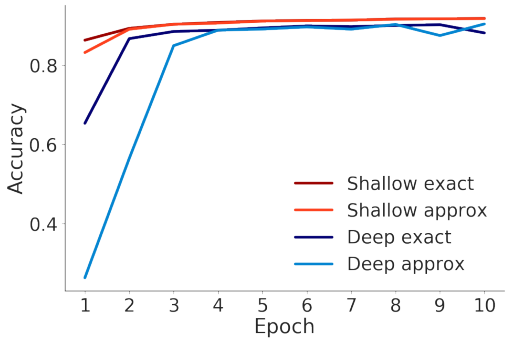

Fig. 3: Test accuracy.

4.3 Experiment 2: Computation in the finite field

In this experiment, the increase in time and difference in accuracy after one epoch is measured when using public and private computations in the finite field. For the private computations, an approximation of the ReLU is necessary. As shown in Table 4, the accuracy after one epoch of training the single convlayer architecture in the SPDZ finite field encoding is the same when using the same ReLU function. This shows that the precision is high enough to adequately encode the floating point values of the weights in the network. The execution time is increased by a factor 99.0 when taking the exact same network, and a factor 114.4 when also encoding the values used for the polynomial approximation of the ReLU. This time increase is mostly due to the 128 bit representation of integers, preventing use of many CPU/GPU instructions and the possibility to chain operations. The latter entails that the python interpreter has to wait for the result after every matrix operation before being able to call a new matrix operation (in optimized C). The listed CPU time increase is problematic. However, it is not a theoretical increase, but rather a technical one; Methods and/or hardware have to be developed to compute operations on these representations more efficiently.

Table 4: CPU time overhead of finite field encoding.

Tensor	Time	Time increase	Accuracy
Native	00h 00m 59s	-	86.3%
Public encoded (ReLU exact)	01h 37m 22s	99.0x	86.3%
Public encoded (ReLU approx)	01h 52m 28s	114.4x	83.2%

4.4 Experiment 3: Computation on secret-shared values

In order to research the actual effect on CPU time of secret-sharing rather than finite-field encoding, the private encoding is compared against the public encoding. The CPU time for one epoch is listed for both encodings using the single convlayer architecture. A time increase of a factor 7.5 is acceptable for many application. It should be noted that other aspects in a real-world setting would

increase training time for other reasons than computations, i.e. waiting on another participant that hasn't finished computation or the time it takes to communicate the secret-shared values.

Table 5: CPU time overhead of secret-sharing computation.

Tensor	Time	Time increase	Accuracy
Public encoded (ReLU approx)	01h 52m 28s	-	83.2%
Private encoded (ReLU approx)	14h 05m 24s	7.5x	83.2%

4.5 Experiment 4: Communication cost and optimization

To measure the effect of the communication optimization, the number of values that is exchanged from one party to the other for one iteration is reported. The iteration consists of one forward and one backward phase through the network with a batch size of 128. For one epoch, these numbers have to be multiplied by $\frac{60K}{128}$, as there are 60K examples in the MNIST training dataset. The communication is reported for the shallow and the deep convnet, both with and without the proposed communication optimization in Table 6 and Table 7. The models without the optimization only use a specialized triple in the dot product.

For the shallow convnet the amount of communication is reduced from 13.2M to 4.2M: a factor 3.3; For the deep convnet it is reduced from 29.6M to 5.1M: a factor 5.8.

When comparing the shallow convnet to the deep convnet, the difference in extra communication when using the proposed optimization is reduced from a factor 2.24 to a factor 1.23. This improvement is largely achieved by the optimization of the convolutional layer, which communicates 1M values instead of 16M values. This result shows that the proposed optimization make a large difference when expanding the use of SPDZ to deeper CNNs.

Table 6: Communicated # of 128 bit values per iteration in the single convlayer architecture.

	Baseline				Optimized			
Layer	Forward	Backward	Total		Forward	Backward	Total	
conv2D (32,3,3)	903K	4,114K	5,018K	(38%)	101K	803K	903K	(22%)
avg_pooling2D (2,2)	-	-	-		-	-	-	
ReLU (approx)	3,211K	3,211K	6,423K	(49%)	1,605K	803K	2,408K	(58%)
dense (10,6272)	866K	868K	1,734K	(13%)	866K	1K	867K	(21%)
total	4,980K	8,194K	13,174K	(100%)	2,572K	1,607K	4,179K	(100%)

Table 7: Communicated # of 128 bit values per iteration in the double convlayer architecture.

	Baseline				Optimized			
Layer	Forward	Backward	Total		Forward	Backward	Total	
conv2D (32,3,3)	903K	4,114K	5,018K	(17%)	101K	803K	903K	(18%)
avg_pooling2D (2,2)	-	-	-		-	-	-	
ReLU (approx)	3,211K	3,211K	6,423K	(22%)	1,606K	803K	2,408K	(47%)
conv2D (32,3,3)	7,235K	8,840K	16,075K	(54%)	812K	201K	1,013K	(20%)
avg_pooling2D (2,2)	-	-	-		-	-	-	
ReLU (approx)	803K	803K	1,606K	(5%)	401K	201K	602K	(12%)
dense (10,1568)	216K	219K	435K	(1%)	216K	1K	218K	(4%)
total	12,368K	17,188K	29,556K	(100%)	3,136K	2,008K	5,144K	(100%)

For both baseline and optimized models, the ReLU layer accounts for a substantial part of the communication. This is mainly due to repeated masking of the input to compute the various terms of the polynomial.

For the baseline models, the backward phase is more expensive in terms of communication than the forward phase, which is intuitive, since the backward phase requires computations of both the parameter updates and the backpropagated gradients. However, the optimized models utilize the re-use of masks of layer inputs, layer parameters and incoming backpropagated gradients to reduce communication in the backward phase. As a result, the backward phase is even less expensive in terms of communication than the forward phase for the optimized models.

5 Conclusion and discussion

The proposed architecture was able to successfully privately train a converging convolutional neural network with ReLU approximations and average pooling on the MNIST dataset. The accuracy of the networks are comparable to their non-private equivalents, although they converge somewhat slower.

Training the CNNs using the proposed method is not fully private as the outputs of the softmax layer are public. It is hard to quantify what this entails in either an honest-but-curious or a malicious majority setting, but both encryption methods for exponentiation, loss functions without exponentiation and other ways to secure this part are available. As this research has a focus on the feasibility of using SPDZ for more generic deep-learning architectures, this is left to future research.

The experiments have shown that communication cost can be decreased greatly using specialized triples. Especially for deeper CNNs this is of great importance. On the contrary, the ReLU approximation requires a large amount of communication and alternatives may outperform this method. Furthermore, corresponding SPDZ pre-processing methods to create specialized triples have to be developed, and the computation power and communication complexity this entails in the preprocessing phase of SPDZ has to be researched. Furthermore, where the need for scalar multiplication triples in general shared computation is ubiquitous, it might be less so for e.g. convolution triples, especially when using uncommon filter sizes. These costs have to be researched to further evaluate the feasibility of this approach.

A great optimization possibility is expected in a more efficient encoding of the integers in a finite field. As SPDZ is not developed for matrix multiplication of large floating-point matrices, it is to be expected that the system is not optimized for such computation. Furthermore, many other optimization techniques should be evaluated whether they can be translated to a finite-field representation.

Various other approaches to decrease CPU time and communication complexity can be researched. Both from an AI perspective, e.g. reducing the need for high precision in the finite field; as from an MPC perspective, e.g. combining GC methods with SPDZ. As this is a novel field where various disciplines cross, many interesting paths are yet to be explored.

References

1. Private image analysis with mpc. https://mortendahl.github.io/2017/09/19/private-image-analysis-with-mpc/. Accessed: 2018-01-18.
2. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. Cryptology ePrint Archive, Report 2010/514, 2010. https://eprint.iacr.org/2010/514.
3. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.
4. Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.
5. Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), October 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.
6. Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. Technical report, February 2016.
7. Bolin Gao and Lacra Pavel. On the properties of the softmax function with application in game theory and reinforcement learning. 04 2017.
8. Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *CoRR*, abs/1711.05189, 2017.
9. Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.
10. Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. Cryptology ePrint Archive, Report 2017/396, 2017. https://eprint.iacr.org/2017/396.
11. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. Cryptology ePrint Archive, Report 2011/091, 2011. https://eprint.iacr.org/2011/091.
12. I. Damgård, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.
13. Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. *CoRR*, abs/1705.08963, 2017.