Efficient Optimisation Framework for Convolutional Neural Networks with Secure Multiparty Computation

Cate Berry and Nikos Komninos[a,1]

[a]*School of Mathematics, Computer Science and Engineering, City, University of London*

Abstract

Secure multiparty computation (MPC) has the potential to enable the machine learning analysis of data previously unavailable due to privacy concerns, however current implementations are limited in their applications due to the prohibitively high cost of computation and communication. Current MPC frameworks do not take full advantage of potential optimisation methods from the wider field of privacy-preserving deep learning. In this paper we present various optimisation approaches, including batch normalisation and polynomial approximation of activation functions, and evaluate their performance when applied to a privacy-preserving convolutional neural network (CNN), discussing the trade-off each offers in terms of their accuracy and efficiency. We experiment with parametric polynomial (*PPoly*) activations by deriving polynomial approximations to activation functions and allowing the network to tune the coefficients as learning weights. We will show that, in shallow CNNs, the application of batch normalisation in combination with a *PPoly* activation layer can result in faster convergence with testing accuracy exceeding that achieved with an unencrypted network, at the cost of longer running times.

Keywords: secure multiparty computation, convolutional neural network, polynomial approximation, activation function, batch normalisation

1. Introduction

In recent years, the potential power of machine learning (ML) and artificial intelligence (AI) to positively impact decision-making has become

[1]Email: catherine.statham@city.ac.uk, nikos.komninos.1@city.ac.uk

increasingly apparent across a range of application domains. Due to advances in computer hardware and the increased availability of large amounts of data, the application of deep learning with neural networks in particular has become more feasible, generating impressive results in fields from machine translation to medical imaging [1], [2]. Deep learning algorithms can infer features and structure in the input data that may not be discernible to the human eye and use these to produce abstract representations that capture the variation in the input data. One of the most popular deep learning algorithms is the convolutional neural network (CNN). These networks can achieve state of the art performance in various applications and are particularly suited to imaging tasks due to their ability to learn the underlying patterns and features of an image without human supervision or input. In this study, we focus on training and testing privacy-preserving CNNs in particular. This allows us to present results that more closely resemble those that may be obtained in a real-world scenario in industry or academia, where it is not uncommon to train or fine-tune a CNN. This also allows us to compare our results to existing work in the field, where CNN architectures and the MNIST dataset [3] often provide a baseline set of results. It should be noted, however, that the techniques we develop are not limited to CNNs and may be similarly applied in other deep learning architectures.

In most cases, the power of a machine learning model is heavily dependent on the availability of a reliable source of data that the model may be trained and tested upon. Training a model on larger quantities of data usually results in better generalisation to unseen data, making the model more accurate and trustworthy when applied in practice. In many application domains, however, the nature of the problems to be addressed can prohibit the acquisition and usage of such data. For example, medical researchers may wish to build an ML model to predict patient diagnoses based on their reported symptoms but may be prevented from doing so by the need to preserve the privacy and confidentiality of patient medical history data.

The privacy of sensitive data has historically been ensured through the use of cryptographic techniques such as encryption, with different schemes providing varying levels of security. In an ML setting, however, most of the classical encryption techniques are not applicable due to the need to perform computations over private input data, which is necessarily obscured by the encryption process. There exist several cryptographic primitives, however, that are suited to privacy-preserving ML and AI. These include Secure Multiparty Computation (MPC), a generic cryptographic primitive which allows a number of independent parties to perform computations over private data held in a distributed fashion between them [4].

In real-world applications of privacy-preserving deep learning with neural networks, the most important considerations, alongside security, are the accuracy and efficiency of the model to be deployed. Parties participating in MPC schemes generally incur far higher computation and communication costs than in traditional ML schemes, prohibiting the usage of MPC in most practical applications. In this study, we aim to develop and evaluate approaches for optimising the training of privacy-preserving CNNs with MPC. We hope that the results we present and the conclusions we draw give fresh insight into the improvements that may be made to the feasibility and practicality of implementing privacy-preserving neural networks.

1.1. Contributions

We show that the choice of both the activation function used in privacy-preserving CNNs as well as the method of approximation used has a large impact on the performance of the neural network in practice, in terms of both the efficiency and accuracy of the network. We provide insights into the effect of using polynomials of differing degrees and methods of initialisation, some of which are novel in a secret-sharing-based MPC scheme for privacy-preserving CNNs.

As the first study to evaluate various methods of polynomial approximation in conjunction with different types of activation functions in the context of secret-sharing-based MPC, we provide experimental results which demonstrate the strengths and limitations of these different combinations. We additionally show that the techniques used may be applied in the both the training and inference phases of the application of privacy-preserving CNNs, preserving the accuracy of computations and mitigating the impact on the efficiency of the network to a greater degree than previous implementations. We also implement and evaluate parametric activation functions, presenting an effective technique, novel in the context of MPC schemes, where an activation function is initialised with a polynomial approximation and tuned alongside the other weights of the CNN during the training phase. This method is shown to outperform the other methods tested in our experiments, achieving the highest recorded accuracy with a low-degree polynomial.

The results achieved contribute to the body of research focused on improving the practicality of applying MPC for ML in practice. In feed-forward neural networks and CNNs in particular, the activation function may be considered the most important element in enabling models to effectively learn the underlying patterns and features in data. The results we present thus provide valuable insights into the most effective activation functions that may be applied in MPC schemes in terms of both accuracy and efficiency, and lay a basis of empirical findings for future research in this field to build upon.

1.2. Paper Organisation

The sections of this paper are organised as follows. Chapter 2 provides an overview of existing work in this domain. In Chapter 3 we present a formal description of our proposed framework and detail the theoretical background and the methods used in its development. Chapter 4 outlines the testing methodology, the experimental results achieved, and a discussion of our findings. Conclusions and proposals for future work are given in Chapter 5.

2. Related Work

There exists a large and still growing body of research into the optimisation of non-privacy-preserving neural networks, however research into the application of these methods to privacy-preserving neural networks is sparse. In this section we outline some ways in which neural networks may be optimised, the limitations of MPC with respect to machine learning and optimisation, and the ways in which prior work has attempted to overcome these limitations. A summary of the results achieved by papers discussed in this section can be seen in Table 1.

2.1. Optimisation of Neural Networks

As mentioned previously, a convolutional neural network (CNN) is a specific type of feed-forward neural network which possesses certain architectural features. The first and defining layer of a CNN is the convolutional layer. Layers of this type are comprised of a set of learnable 'filters'. These are small windows that are passed over the width and height of the input volume, with the weights of the filter convolved with the input values at each spatial location.

An activation layer is usually placed after each convolutional layer. These are non-linear functions which perform a computation over every individual input value and feed the output forward to the next layer. These layers therefore allow the network to filter out information being passed from neurons that do not meet the activation threshold, with the combination of multiple non-linear layers enabling the network to develop more expressive representations of the data. If a linear activation function is used then, even with multiple layers, the network can still only learn linear representations of the data. This severely limits the possibilities of the network to learn, since the power of a neural network is derived from its ability to detect and represent complex and non-linear patterns and features in data. In fact, the use of a non-linear activation function means that neural networks with as few as two hidden layers can be proven to be universal function approximators [5]. Thus, the choice of activation function may ultimately be considered the most important factor in the successful implementation of a neural network.

As the research in the field of deep learning has progressed, a number of different activation functions have been proposed. One of the most significant advances came with the introduction of the Rectified Linear Unit (ReLU) activation function:

$$ReLU(x) = \max(0, x) \tag{1}$$

Variations of the ReLU activation function have since been developed, including the Leaky ReLU [6] and Swish [7] activation functions, which are described in further detail in section 3.3.

Additionally, parametric activation functions have been proposed, which allow the parameters of the activation function to be treated as learning weights that may be tuned by the model alongside the other learnable parameters of the neural network during training. This has been demonstrated with the PReLU activation function, which has the same mathematical structure as the Leaky ReLU function, except that the gradient parameter a is treated as a learnable value rather than a constant [8]. The PReLU function has been shown to improve model fitting with nearly zero extra computational cost and little risk of overfitting [8].

An optional but common technique to accelerate the convergence of a neural network is to apply a batch normalisation layer [9]. This normalises the output of the previous layer by subtracting each individual value by the mean of the mini-batch and dividing by the standard deviation. This has the effect of mapping the data to a normal distribution, with a mean of zero and unit variance. A scale coefficient and offset are then applied to the result, which may be learnt by the network during training to allow for additional tuning of the model to best fit the data. We define batch normalisation with the following expression:

$$BN(x_i) = \gamma \cdot \frac{x_i - \mu_\beta}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \tag{2}$$

Here, μ_B is the sample mean and σ_B is the sample standard deviation of the mini-batch B, while γ and β are learnable parameters and ϵ is a small constant. Although it is not entirely understood why batch normalisation can provide such a significant improvement to model training, the authors of the seminal paper that introduced the concept postulate that maintaining a similar distribution and range of values between the different layers of the network enables the network to convergence more efficiently [9].

2.2. Limitations of Machine Learning with MPC

The major challenge in the application of MPC to neural networks in particular is the need to compute non-linear functions securely and efficiently. Neural networks rely on the non-linear properties of the function applied in the activation layer to enable them to learn complex patterns and features in the training data. Since non-linear operations cannot be performed natively with MPC, alternative methods must be employed to emulate the output of these operations.

One such method is to use garbled circuits (GCs) [10]. Mohassel and Zhang [11] apply this technique in a study that proposes new and efficient protocols for privacy-preserving linear regression, logistic regression, and neural network training. Methods are proposed for secure fixed-point arithmetic, making use of an offline phase to generate random multiplication triples to allow for efficient multiplication of secret shared integers during the online phase with no impact on accuracy. The authors use GCs to evaluate the ReLU function and approximate the logistic and softmax functions using combinations of ReLU functions, addition, and division. However, this approach proves to be highly inefficient and impractical in most settings, motivating us to consider alternative methods.

A separate study from Seggers et al. [12] proposes various methods for performing privacy-preserving training of CNNs. Here, the ReLU activation function is approximated with a polynomial, with experimental results showing that the final testing accuracy does not differ for the exact and approximated ReLU functions, although the time taken for the network to converge is increased. This paper does not consider alternative approximation methods or activation functions, considering only a degree-3 approximation in their experiments. We believe that further research and experimentation may reveal methods that allow for better approximations and faster convergence of the network.

2.3. Polynomial Approximation

While addition and multiplication operations are relatively inexpensive to compute in MPC, more complex non-linear functions cannot be computed using arithmetic circuits, and so present a greater challenge. One approach to mitigating this challenge is to approximate these complex functions using polynomials, which by definition are evaluated using only addition and multiplication operations, which are relatively inexpensive in MPC with secret-sharing [12].

This has been previously demonstrated by Gilad-Bachrach et al. [13], who used the lowest degree non-linear polynomial function, the square function $f(x) = x^2$, as the activation function in their implementation. While the low degree of this polynomial allows for efficient computation with MPC, in a deep CNN the unbounded derivative of the function leads to unstable training and significant accuracy loss [14].

Hesamifard et al.[15] present methods for the approximation of the ReLU, *sigmoid* and *tanh* activation functions with low-degree polynomials. While this research was conducted in the context of privacy-preserving ML, it utilises a technique known as homomorphic encryption (HE) rather than MPC. These methods are fundamentally different in their application, however both require that computations be performed using only addition and multiplication operations for efficiency, meaning that some optimisation techniques may be transferable between them. In particular, the authors focus on approximation with low-degree polynomials, accepting poorer accuracy as a trade-off for computational performance. The study compares various approximation methods for activation functions including numerical analysis, Taylor series, and Chebyshev polynomials. The authors show that for the ReLU approximation, least squares regression outperforms Taylor series, with 56.8% accuracy compared to 40.3%. Chebyshev polynomials far outperform both of these, however, with 69.0% for a standard Chebyshev polynomial and 88.5% for a Chebyshev polynomial modified to better simulate the structure of the ReLU function [15].

2.4. Batch Normalisation in Combination with Polynomial Approximation

Another study in the field of privacy-preserving ML that utilises least squares regression to approximate the ReLU function presents a novel method for improving the quality of the approximation. Chabanne et al. [14] introduce the approach of combining the polynomial approximation of the ReLU function with a batch normalisation layer. Applying a batch normalisation layer before each activation layer means that the data that is received as an input to the activation layer has a restricted stable distribution. This is because, due to the central limit theorem, we know that the output of the preceding convolutional layer has a normal distribution, and thus the application of a batch normalisation layer to this data will result in the output data having a standard normal distribution. A distribution with zero mean and unit variance will have 99.73% of its data between the range $[-3, 3]$, reducing the size of the interval where the polynomial approximation needs to be accurate to a small interval around zero, allowing for a more accurate approximation, particularly with lower-degree polynomials [14].

Chabanne et al. [14] only implement polynomial approximation for the prediction phase, meaning that the networks are trained on unencrypted data using the exact forms of the activation functions, rather than approximations. In this project we aim to implement MPC in both the training and prediction phases, meaning that we both train and test the network in an encrypted way using activation functions approximated by polynomials. The authors also base their scheme on HE rather than MPC, meaning that their results may not be directly comparable to ours. However, it is still notable that the authors achieved a classification accuracy of 98.18% with a CNN trained using the aforementioned methods, which is close to the accuracy achieved by an equivalent unencrypted network (99.59%).

Batch normalisation cannot be directly applied in an MPC scheme due to the division and square root operations required in its computation. Recent work by Wagh [16], however, presents a novel protocol for performing batch normalisation in an MPC setting, making use of numerical methods to approximate the non-linear operations. Numerical methods are techniques that approximate the output of mathematical functions or protocols where it is impossible or intractable to solve them analytically. Wagh [16] defines a series of algorithms that perform the necessary steps of the approximations, including calculating the range over which the approximations will be performed and the initial approximation that will be used as a starting point.

2.5. Parametric Polynomial Activation Functions

Rather than directly approximating an existing activation function such as ReLU with a polynomial, it is possible to initialise a polynomial activation function with arbitrary coefficients and then treat these coefficients as learning weights that may be tuned by the model alongside the other learnable parameters during training.

This idea has been explored in a privacy-preserving ML context by Wu et al. [17], who build upon the framework presented by Hesamifard et al. [15] with the aim of improving predictive accuracy while maintaining efficiency in HE schemes for the secure training and evaluation of CNNs. In experiments conducted over encrypted data, the authors demonstrate that it is possible to achieve better accuracy using shallower CNNs when using parametric polynomial (*PPoly*) activation functions than that attained in previous works. Wu et al. [17] show that in a shallow network, *PPoly* activation functions outperform the exact ReLU activation function, achieving 99.33% and 99.05% testing accuracy respectively. Being able to achieve comparable or better accuracy with a shallower CNN can greatly improve the practicality of a model, particularly in an MPC context where computational costs are exacerbated by a significant factor.

Framework	Activation Function	Method	Accuracy
Seggers et al. [12]	ReLU (order 3)	Regression	83.2%
Hesamifard et al. [15]∗	ReLU	Regression	56.87%
		Taylor series	40.28%
		Standard Chebyshev	68.98%
		Modified Chebyshev	88.53%
Chabanne et al. [14]∗∗	ReLU (order 2)	Regression	98.18%
Wu et al. [17]∗	Polynomial (order 2)	*PPoly*	99.33%

Table 1: Summary of related work. All experiments were performed on the MNIST dataset [3] using comparably sized CNN architectures.
∗ Model implemented with HE
∗∗ Model implemented with HE, polynomial approximation used in prediction phase only

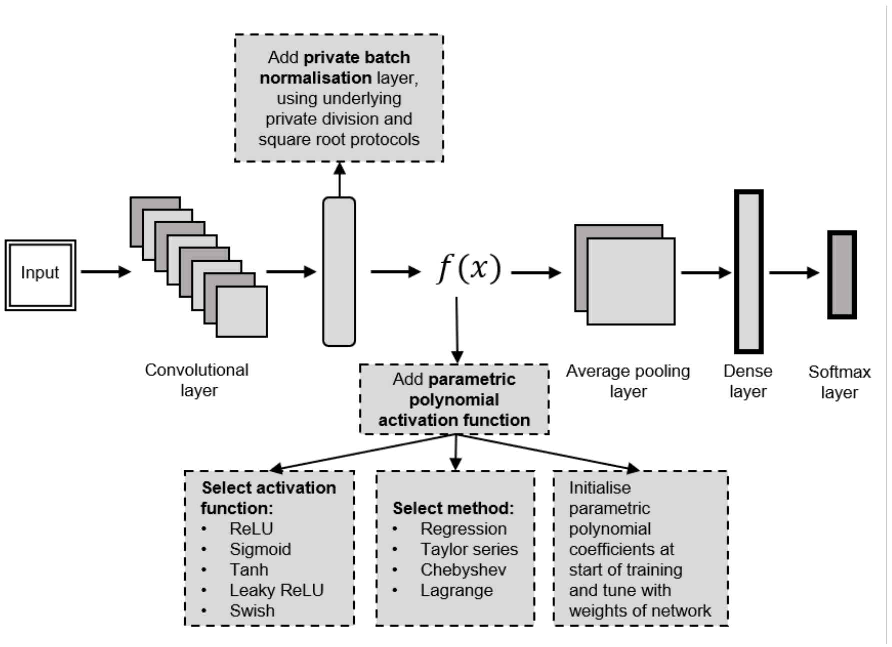

Add private batch normalisation layer, using underlying private division and square root protocols
Input
Convolutional layer
f(x)
Average pooling layer
Dense layer
Softmax layer
Add parametric polynomial activation function
Select activation function: ReLU, Sigmoid, Tanh, Leaky ReLU, Swish
Select method: Regression, Taylor series, Chebyshev, Lagrange
Initialise parametric polynomial coefficients at start of training and tune with weights of network

Figure 1: Efficient Optimisation Framework of CNNs with MPC

3. Efficient Optimisation of CNNs with MPC

In this section we will present our proposed framework for optimising the efficiency of convolutional neural networks implemented with secure multi-party computation. We will present the requisite theoretical background to our research and describe the protocols and methods we have developed, and outline how to apply our findings in practice.

3.1. Framework Setting

While several techniques exist for performing MPC, in our experiments we utilise an additive secret-sharing approach. Here, each individual party's input data is split into "shares" and distributed between all of the parties participating in the scheme. In an m-party MPC scheme, a value x is secret-shared between parties $p_1, ..., p_m$ by choosing $x_{p_1}, ..., x_{p_m} \in \mathbb{F}_q$ uniformly at random such that:

$$x = \sum_{i=1}^{m} x_{p_i} \, mod \, q \tag{3}$$

and then revealing x_i to p_i. We denote such a secret-sharing by $[x]_q$. It should be noted that no information about the true value of x is learnt by any individual party, and that no proper subset of parties can reconstruct x. The secret-shared value can be revealed to one of the parties if all of the other parties in the scheme send their shares to that party. In MPC schemes based on secret-sharing, computations are generally performed using arithmetic circuits over a finite field modulo Q or over a ring modulo 2^n, with Boolean circuits being the special case when $n = 1$.

When implementing an MPC protocol or framework in practice, there exist various model architectures that may be applied which determine how the scheme is executed, such as how many parties are involved in the scheme and the nature of their interactions with each other. A common approach is to apply a 'client-server' architecture, where the parties who perform the actual computations in the MPC scheme are a number of distributed servers who are unaffiliated with the clients who own the data [18]. These servers receive each client's data in its secret-shared form, so that they never see the true values of the inputs, and then perform the necessary calculations to produce the result, which is also returned to the parties in a secret-shared form. The client-server model is especially strong in a machine learning context, since many ML models and algorithms require a large number of calculations, which may be computation and storage heavy. By outsourcing computation to external servers, which may have more resources to expend, we save the clients themselves from having to perform many local computations. It should be noted that if a client-server architecture is utilised, then the addition of more parties (servers, in this case) does not necessarily present any advantage, and in fact increases the complexity of the scheme. Any number of clients can take part in the scheme by supplying their data in a secret-shared form to the servers who will perform the MPC protocols and return the results. Going forward, we develop our framework in the 2-party setting, assuming a client-server design.

3.2. Batch Normalisation

As mentioned previously, it has become relatively commonplace to include a batch normalisation layer in CNN architectures as an additional optimisation technique that can allow the network to converge more quickly to an optimal configuration of weights.

Implementing batch normalisation in a secret-sharing MPC scheme presents a challenge due to the complex operations required by the algorithm, the details of which may be seen in Algorithm 1. While the mean and variance of a mini-batch can be calculated using just addition and multiplication

Algorithm 1 Batch Normalisation

```
Input: Mini-batch of data $X = (x_1, ..., x_m)$
Output: Normalised data $Y = (y_1, ..., y_m)$
1: Calculate mean: $\mu \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$
2: Calculate variance: $\sigma^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu)^2$
3: Normalise: $\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$
4: Scale and shift: $y_i \leftarrow \gamma \hat{x}_i + \beta$
5: return $Y = (y_1, ... y_m)$
```

operations for the secret-shared values, as well as division by an unencrypted value (the mini-batch size), the normalisation step requires division by the square root of an encrypted value. Neither division nor square root can be computed natively using MPC, meaning that batch normalisation cannot be directly applied.

We adapt the methods outlined by Wagh [16] to implement approximated division and square root protocols in *Python*. We apply numerical methods to approximate these functions using algorithms that use only the operations that are compatible with MPC, which in turn allows us to perform the batch normalisation algorithm securely over private data.

Algorithm 2 Private Division

```
Input: $P_1, P_2$ hold shares of $a, b$
Output: $P_1, P_2$ hold shares of $a/b$
1: Compute $w_0 \leftarrow 2.9142 - 2b$
2: Compute $\epsilon_0 \leftarrow 1 - b \cdot w_0$
3: Compute $x_0 \leftarrow a \cdot w_0 \cdot (\epsilon_0 + 1)$
4: for $i = 1$ do
5:     $\epsilon_i \leftarrow \epsilon_{i-1}^2$
6:     $x_i \leftarrow x_{i-1} \cdot (\epsilon_i + 1)$
7: end for
8: return $x_1 \approx a/b$
```

3.2.1. Division Protocol

The division protocol outlined by Wagh [16] begins with $w_0 \cdot (1 + \epsilon_0)$ as an initial approximation for $\frac{1}{x}$, where $w_0 = 2.9142 - 2x$ and $\epsilon_0 = 1 - x \cdot w_0$. This choice of initial values was suggested by Catrina and Saxena [19], who

Algorithm 3 Private Square Root

```
Input: $P_1, P_2$ hold shares of $a$
Output: $P_1, P_2$ hold shares of $\sqrt{a}$
1: Set $x_0 \leftarrow 0.5480842735a + 0.3875541065$
2: for $i = 0$ to 3 do
3:     $x_{i+1} \leftarrow \frac{1}{2}(x_i + \frac{a}{x_i})$
4: end for
5: return $x_3 \approx \sqrt{a}$
```

develop a similar division protocol for MPC over data held in fixed-point representation. We proceed iteratively by setting $\epsilon_i = \epsilon_{i-1}^2$ and multiplying the previous approximate result by $(1 + \epsilon_i)$. Each iteration improves the accuracy of the approximation, but also increases the round complexity, i.e. the number of communication rounds required between parties, by 2, which in turn increases the running time of the protocol. Experiments with the number of iterations of the protocol are described in section 4.3. The full algorithm can be seen in Algorithm 2.

3.2.2. Square Root Protocol

In order to compute the denominator in the normalisation step of the batch normalisation algorithm, we must calculate the square root of an encrypted value. Wagh [16] approaches this in a similar fashion to the division protocol by applying the Newton-Raphson method to approximate the output of the square root operation. The Newton-Raphson method is used to approximate the zero of a continuous, once differentiable function f [20]. Starting from an initial value x_0, we compute an approximation with the iterative formula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{4}$$

With $f(R) = R^2 - x$, whose zero is $\sqrt{x}$, we obtain the iterative formula for computing the square root of a value a:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n}) \tag{5}$$

The choice of the initial value x_0 is important ensuring efficient convergence of the algorithm to an accurate approximation. Following the approach de-

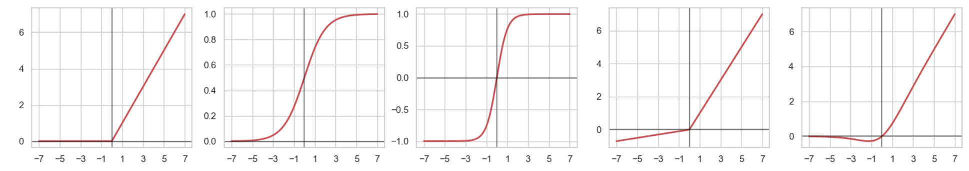


Figure 2: (from left to right) ReLU, *sigmoid*, *tanh*, Leaky ReLU and Swish functions

scribed in [20], we derive values for the starting value through linear approximation:

$$L(x) = \alpha \cdot x + \beta \tag{6}$$

By minimising the relative error function and solving a system of equations, we arrive at the values $\alpha = 0.5480842735$ and $\beta = 0.3875541065$, which allow us to compute an initial linear approximation to $\sqrt{x}$ using the above formula. The full square root algorithm can be seen in Algorithm 3.

3.3. Activation Functions

As described previously, the activation function used in a neural network is arguably the most crucial component for enabling the network to learn deeper features and underlying patterns in data. For this reason, many different types of activation functions have been developed and implemented over time.

One of the classical activation functions is the *sigmoid* function:

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \tag{7}$$

The form of the *sigmoid* function, as well as the other activation functions we describe in this section, may be seen in Figure 2. The *sigmoid* function was the activation function primarily used in early research into deep learning due to its being non-linear, continuously differentiable, monotonic, and bounded. In particular, the output of the *sigmoid* function is bounded in the range $(0, 1)$, meaning that extremely large or small input values are clipped or 'squashed' to this range. This has the advantage of imitating biological synapses, however it also results in very high and very low values being saturated, with the function only being sensitive to changes in input values of around 0. *sigmoid* functions also carry the disadvantage of the gradient being close to zero at the tails of the curve. This can result in the 'vanishing gradient problem' during the training of deep neural networks, where the gradient computed and backpropagated through the network becomes increasingly small, preventing the network from learning effectively.

Another common activation function is the hyperbolic tangent (*tanh*) function:

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{8}$$

The *tanh* function is similar to the *sigmoid* function, however it operates over the range $(-1, 1)$ with a centre at 0. *tanh* has come to be generally preferred over the *sigmoid* function in recent years, with networks using the *tanh* activation functions being shown to provide faster convergence and lower classification error than those using *sigmoid* activation functions [21]. Despite these improvements, the *tanh* function still suffers from the same issues as the *sigmoid* function, namely the saturation of extreme values and the vanishing gradient problem.

In recent years, the ReLU function (Equation 1) has come to be favoured over the *sigmoid* and *tanh* functions. ReLU is a piecewise function, with the output being zero for input values less than or equal to zero, and the output being equivalent to the input for values above zero. This has the advantage of enabling sparse activations in the network, with around 50% of hidden neurons forced to zero in a randomly initialised network. Unlike *sigmoid* and *tanh*, ReLU does not saturate values and thus does not suffer from the same vanishing gradient problem, resulting in a much faster rate of convergence of the network. For this reason, the ReLU function has quickly become the state-of-the-art activation function used in deep learning applications.

However, ReLU still has some limitations that can result in difficulties during training. Since the range of the ReLU function is $[0, \infty)$, some activations can 'blow up' and become extremely large, since they are not bounded in the same way as with the *sigmoid* or *tanh* functions. We may also encounter the 'dying ReLU' problem, which occurs when the input to a neuron is consistently negative, meaning that the neuron never activates. This prevents gradients from flowing backward through the neuron during backpropagation, meaning that the weights are not updated, and the neuron becomes effectively useless. If a large number of neurons become inactive, then the capacity of the network is reduced, and its learning capabilities impeded.

One approach to mitigating the dying ReLU problem is to implement a Leaky ReLU activation function [6]:

$$f(x) = \begin{cases} x, & \text{if } x > 0. \\ ax, & \text{otherwise.} \end{cases} \tag{9}$$

The Leaky ReLU is a variation of the ReLU function that allows for a small, positive gradient (typically $a = 0.01$) when the unit is not active, rather than zero gradient. In theory this could prevent neurons from dying by ensuring that the gradient flowing backwards is not zero. Some studies have successfully implemented the Leaky ReLU, achieving lower test error than with a regular ReLU [22], however results have generally been inconsistent across different algorithms and tasks, making its overall benefit unclear. Pedamonti [23] compares the accuracy achieved on an image classification task using an artificial neural network with various different activation functions. The author achieved validation accuracy of 95.5% with *sigmoid*, 97.7% with ReLU and 97.6% with Leaky ReLU, showing that ReLU-like functions significantly outperform the *sigmoid* function, and that the loss in accuracy from applying Leaky ReLU over ReLU is negligible.

The final activation function we consider is the Swish activation function [7]:

$$f(x) = x \cdot \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \tag{10}$$

Here, σ is the *sigmoid* function as defined previously, and β is either a constant or a trainable parameter (typically $\beta = 1$). Swish can be loosely viewed as a smooth function which nonlinearly interpolates between the linear function and the ReLU function [7]. Compared to the ReLU function, Swish has a non-monotonic "bump" when $x < 0$. The authors of the article introducing the Swish function present results that show that Swish consistently matches or outperforms both the ReLU and Leaky ReLU functions when applied to deep CNNs across a variety of tasks [7]. A study from Hayou et al. [24] analyses the performance of the ReLU, *tanh* and Swish activation functions in neural networks of varying width and depth. Experiments show that *tanh* can outperform ReLU on shallower networks, however the vanishing gradient problem results in *tanh* achieving very poor testing accuracy on deeper networks. The authors hypothesise that Swish will outperform both *tanh* and ReLU, due to possessing characteristics similar to those of *tanh* that enable efficient information propagation, while avoiding the vanishing gradient problem. Numerical results confirm this; while the difference in testing accuracy between ReLU and Swish for shallower networks is small (94.01%

compared to 94.46%), for deeper networks with up to 40 layers, the Swish activation function achieves higher accuracy by almost 6% (91.45% compared to 97.14%) [24].

3.4. Polynomial Approximation

We implement various polynomial approximation methods in *Python* in order to test their effect on the accuracy and efficiency of privacy-preserving CNNs. We build upon the existing *PrivateML* framework, which provides primitives for training neural networks with MPC and secret-sharing in *Python*. The implementation environment is described in further detail in section 4.1.

The *PrivateML* framework natively defines activation layers which approximate the *sigmoid* and ReLU functions with least squares regression. The *sigmoid* function is approximated with a degree-9 polynomial with 5 terms, which means that a series of powers of the encrypted data must be computed. For such a high-degree polynomial, this means that we will incur a significant cost in computation and communication. We aim to show that a better trade-off between the accuracy and efficiency of the approximation may be achieved using alternative polynomial approximation methods that do not require high-degree polynomials.

We implement a least squares regression approximation using the *polyfit* function from the *Python* package *NumPy* [25]. This function takes as arguments a set of sample points (x, y) and the polynomial degree n and returns the coefficients of the polynomial that minimises the squared error. In the basic implementation, (x, y) are uniformly sampled from a given domain. We treat the domain as a hyperparameter of the network and perform experiments with different values to obtain an optimal range from which to sample the points.

We also implement the methods described by Chabanne et al. [14]. In this case, we randomly sample the set of points x from a normal distribution and pass these through the exact ReLU function to obtain the corresponding y values. This can be done offline and without encryption because the data used to derive the polynomial approximation is not private. We begin by sampling from a standard normal distribution with mean zero and unit variance but include the mean and variance of the distribution as hyperparameters that can be altered for testing purposes. Chabanne et al. [14] in fact show that sampling from a distribution with a mean of zero and a standard deviation of slightly higher than 1 can produce more accurate results, particularly for lower-degree polynomials.

We compute Taylor series approximations using the *Python* package *SciPy* [26]. This function takes a callable function f to be approximated, the point x at which the polynomial is to be evaluated, the degree n of the polynomial, and a scale value, which defines the width of the interval that is to be used to evaluate the polynomial. The function outputs the estimated Taylor series polynomial of f at x.

Alongside these we also test the use of Chebyshev polynomials. We develop our implementation using symbolic computation on top of code that defines the underlying mathematical theory of Chebyshev polynomial approximation [27]. The implementation can approximate any of the activation functions defined previously with a polynomial of specified degree over a given interval.

The final approximation method we implement is Lagrange polynomial approximation. Following mathematical theory [28], we apply the *lagrange* function from the *SciPy* package to obtain the Lagrange interpolating polynomial [26].

The points used to evaluate the polynomials should be chosen carefully to avoid large oscillations and inaccuracies between the points. This may be achieved by using lower-degree polynomials or by using Chebyshev nodes as the interpolation points. We implement a functionality to generate both equidistant points and Chebyshev nodes to be used as the interpolation points, enabling us to evaluate how the polynomials generated with these different approaches compare in terms of the accuracy and efficiency of the network.

As an alternative to interpolation, we may also apply the least squares method to obtain a Lagrange approximating polynomial, which leads to an exact approximation if the function to be approximated lies in the space spanned by the basis functions [28]. We implement a generic least squares protocol to perform a least squares fit to a given function over a given interval.

3.4.1. Parametric Polynomial Activation Functions

In order to implement a parametric polynomial activation function in our framework, we build upon the methods described by Wu et al. [17], who implemented such an activation function in an HE-based protocol. Wu et al. define their degree-2 parametric polynomial (*PPoly*) activation function as:

$$p(x_i) = c_{i1}x_i + c_{i2}x_i^2 \tag{11}$$

Where x_i is the input of the activation function in the i^{th}channel and c_{i1} and c_{i2} are learnable parameters. In their implementation, Wu et al. randomly initialise the polynomial coefficients c_{ij} in the range $[-1.5, 1.5]$. A range including negative and positive values was chosen so that the resulting polynomial may be non-monotonic. We explore this approach with a *PPoly* layer in our *Python* framework, which allows the network loss to be backpropagated and used to gradually tune the polynomial coefficients.

Extending the work of Wu et al. [17], we propose that higher accuracy may be achieved by instead initialising the polynomial coefficients with those taken from a baseline polynomial approximation to an existing activation function. We therefore implement a functionality that initialises a *PPoly* activation function with a polynomial approximation to an existing activation function, and then treats the coefficients as learnable weights. In theory, this effectively allows us to derive a polynomial approximation to a function as described in the previous sections, and then tune this approximation during the training phase to better fit the data and potentially produce more accurate results. This also reduces over-parameterisation in the network, as only a few coefficient values need to be learnt by the network, rather than potentially hundreds as is the case in the approach used by Wu et al. [17].

3.5. Efficient Optimisation Framework of CNNs with MPC

We have outlined some of the key bottlenecks that are encountered when implementing neural networks with MPC and have identified some potential areas of improvement over current state of the art methods. We now present our framework for performing more efficient and scalable training and inference with privacy-preserving CNNs. A pictorial diagram of our proposed framework can be seen in Figure 1.

The application of numerical approximation techniques including the Newton-Raphson method have enabled the implementation of private division and square-root operations. Following the protocols and algorithms outlined by Wagh [16] we have been able to implement these protocols in a *Python* environment where their efficacy has been tested and verified. The trade-off between the accuracy and efficiency of these protocols has also been explored, with the findings presented in section 4.3.

A primary consequence of the secure and efficient implementation of these operations is the ability of performing privacy-preserving batch normalisation. Batch normalisation layers have become commonplace in neural network architectures in recent years due to their effect of increasing the speed and stability of convergence of the network. This is no less true in a privacy-preserving network, where any gains made in the efficiency and practicality of the network are in fact even more important.

Alongside the inclusion of a privacy-preserving batch normalisation layer, we also propose the use of polynomial approximated activation layers. While the optimisations provided by applying parametric activation functions may be considered marginal in non-privacy-preserving contexts, in an MPC scheme for training privacy-preserving CNNs the gains in accuracy and efficiency are all the more valuable, leading us to conclude that their implementation is worthwhile and beneficial.

The type of activation function to be approximated, the method of polynomial approximation, and the degree of the polynomial used are the most influential factors that must be determined at this stage. In section 4 we present and discuss the results of experiments with a privacy-preserving CNN where each of these factors are varied alongside other relevant hyperparameters of the network. It is evident from these results that the most successful implementation in terms of accuracy and efficiency was obtained using a parametric polynomial activation function initialised with a degree-3 least squares regression polynomial approximation to the Leaky ReLU function. This approach achieved the highest accuracy of the combinations tested, with 87.5% accuracy on the test set. These results were also achieved in fewer epochs than the other activation functions tested in our experiments. When a *PPoly* activation layer is used, the coefficients of the polynomial activation function are trained alongside the other weights of the network in a privacy-preserving way. The optimal values, corresponding to the state of the network weights at the point of lowest validation error, may be fixed for use in a privacy-preserving inference phase.

4. Experimental Results and Discussion

In this section we present and discuss the experimental results attained by models trained and evaluated with the various combinations of activation functions and polynomial approximation methods described in the previous section. We also perform complexity analysis of the protocols implemented and analyse the implications of our implementation on the security of the networks we train and test.

4.1. Implementation Environment

There exist a number of libraries and frameworks in *Python* which allow users to train neural networks with a combination of simple commands. In order to implement an MPC scheme with secret-sharing, however, we must redefine the operations used to evaluate the network from first principles. This includes the basic arithmetic operations of addition and multiplication, as well as each individual layer and the computations performed within them. Developing a comprehensive framework from the ground up would be out of scope for this project, so we build upon *PrivateML*, an existing secret-sharing-based MPC framework designed specifically for training and evaluating privacy-preserving neural networks [29]. We also make use of a number of *Python* modules for processing and manipulating data, including *NumPy* and *SymPy* [25], [30].

PrivateML provides the basic components necessary for training a CNN in *Python* with secret-sharing-based MPC in a simulated client-server model. We do not implement a functioning client-server model as this would require the configuration of communication with remote servers. This project does not intend to focus on the logistics of the MPC scheme itself, but on the impact of various optimisation approaches on the performance of the CNN model, which may be evaluated in an offline test environment in *Python*.

4.2. Testing Details

The models tested were trained and evaluated on the MNIST dataset [3], a popular image recognition dataset commonly used for evaluating the performance of CNNs.

As mentioned previously, we build upon the *PrivateML* framework [29] and the open-source implementation of this framework in *Python*. Due to

Layer	Hyperparameter settings
Conv2D	32 filters, size=3x3, strides=1, padding=1
BatchNorm	-
PolyActivation	user defined
AveragePooling2D	size=2x2
Dense	10 hidden neurons
Softmax	-

Table 2: CNN architecture

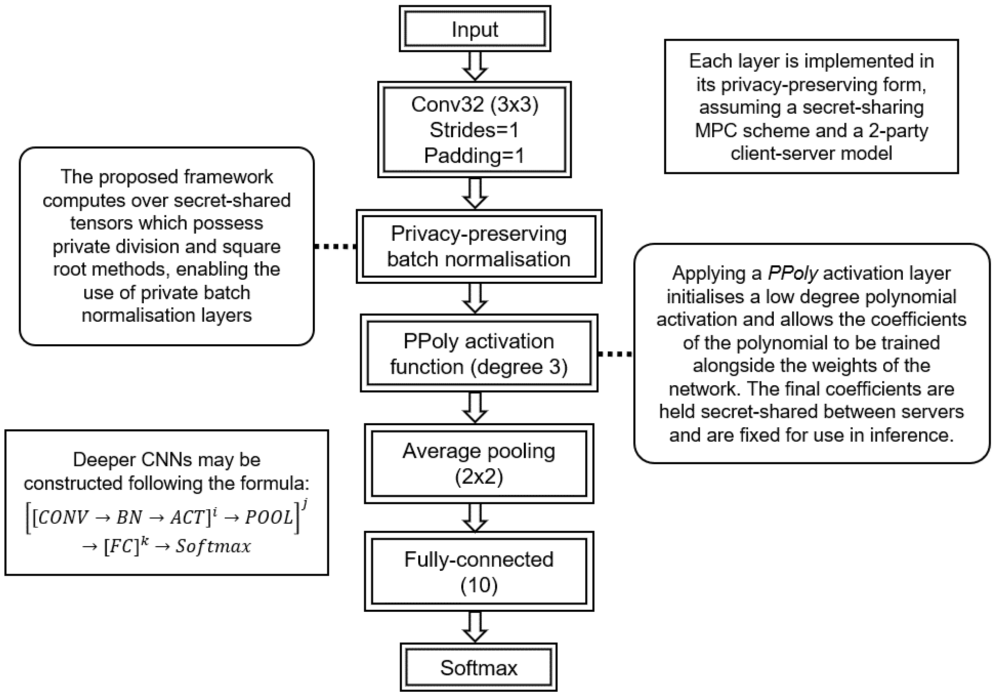

Input
Conv32 (3x3) Strides=1 Padding=1
Privacy-preserving batch normalisation
PPoly activation function (degree 3)
Average pooling (2x2)
Fully-connected (10)
Softmax
Each layer is implemented in its privacy-preserving form, assuming a secret-sharing MPC scheme and a 2-party client-server model
The proposed framework computes over secret-shared tensors which possess private division and square root methods, enabling the use of private batch normalisation layers
Applying a PPoly activation layer initialises a low degree polynomial activation and allows the coefficients of the polynomial to be trained alongside the weights of the network. The final coefficients are held secret-shared between servers and are fixed for use in inference.
Deeper CNNs may be constructed following the formula: $\left[[CONV \to BN \to ACT]^i \to POOL\right]^j \to [FC]^k \to Softmax$

Figure 3: CNN structure and implementation details

inefficiencies in the original code, limitations on computational power, and time restrictions, we perform scaled-down tests in order to keep training times feasible and prevent memory overflows. This means that we must use smaller training, validation, and testing sets than ideal, while also training for fewer epochs than is standard.

Since we are aiming to improve the efficiency and practicality of MPC for training and evaluating CNNs, it is in our interest to use as small a network as possible while maintaining a high degree of accuracy. We are also limited to using a shallow CNN architecture due to the high memory requirements of the *PrivateML* framework, which has not been optimised for large-scale deep learning tasks. In addition, due to time limitations and the number of tests that need to be performed, using a deeper architecture would make training times infeasible. For these reasons we proceed with a shallow CNN architecture as may be seen in Table 2. A diagram showing the structure of the network and the implementation details is presented in Figure 3. Deeper networks may be constructed with the following formula:

$[[CONV \rightarrow BN \rightarrow ACT]^i \rightarrow POOL]^j \rightarrow [FC]^k \rightarrow Softmax$. In our proposed framework, BN is a privacy-preserving batch normalisation layer, ACT refers to a $PPoly$ activation layer, and $POOL$ is an average-pooling layer (preferable to max-pooling in MPC schemes). The hyperparameters of each layer may be altered and optimised to suit individual implementations.

4.3. Protocol Efficiency

We apply asymptotic complexity analysis to the private division and square root protocols to judge their efficiency and practicality. In the case of the division protocol (Algorithm 2), we derive an upper bound on the asymptotic running time of $\mathcal{O}(n)$, where n refers to the length of the array being computed over. Similarly, the square root protocol we have implemented (Algorithm 3) also has an asymptotic running time of $\mathcal{O}(n)$. In these cases we can say that the algorithms have polynomial time complexity, which is considered to be efficient.

Through experiments we determine the optimal number of iterations to be applied in each protocol, allowing us to obtain an accurate approximation in the fewest iterations possible, reducing the computational cost of the protocols and making their application more feasible. This is crucial when attempting to optimise the performance of a neural network in an MPC setting, as we must consider the trade-off between potential improvements in accuracy with the increased computational and communication costs that results from performing repeated operations over secret-shared data.

We experiment with the number of iterations of both the private division and square root protocols, taking into account the running time of the algo-

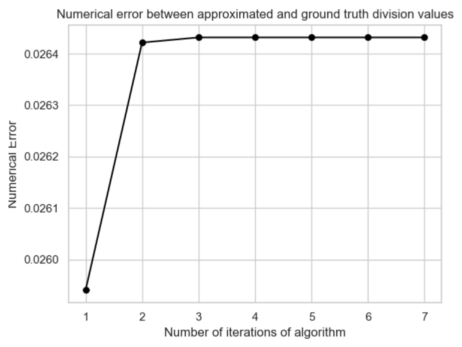
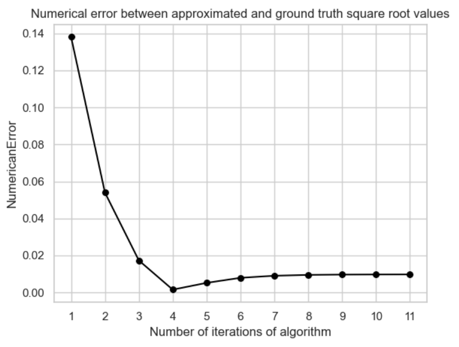
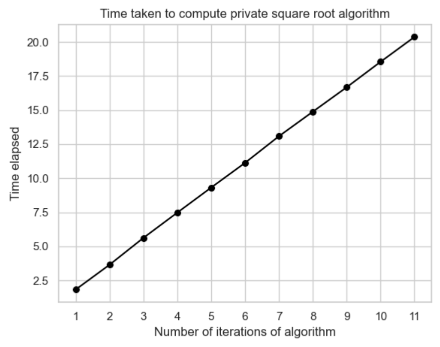


Figure 4: Convergence of division protocol



(a) Convergence of square root protocol



(b) Time taken for square root protocol with additional iterations

Figure 5: Experiments with number of iterations of square root algorithm

rithms and the error between the approximated result and the true result. We evaluate the performance of the square root protocol by measuring the running time and accuracy of the approximation with successive iterations relative to the ground truth. We observe that the square root approximation protocol converges after around 7 iterations (Figure 5a) with the lowest numerical error reported after 4 iterations. The running time of the algorithm also increases linearly with additional iterations, as expected (Figure 5b). The lowest numerical error for the division protocol is found after just 1 iteration (Figure 4), and therefore we fix the number of iterations of the square root and division protocols at 4 and 1 respectively.

4.4. Experimental Results

Table 4 shows the results obtained for each polynomial approximation method with the settings that achieved the highest testing accuracy.

We observe that the highest testing accuracy was attained by the network trained using a degree-3 least squares regression polynomial approximation

Method	Train accuracy	Test accuracy	Runtime (mins)
ReLU (exact)	71.88%	53.13%	0.5
ReLU + BN (exact)	96.88%	81.25%	0.6
ReLU (approx)	43.75%	23.96%	52.8
ReLU + BN (approx)	94.34%	74.35%	156.5

Table 3: Baseline implementation results

Method	Degree	Function	Train acc.	Test acc.	Runtime
Regression	3	Leaky ReLU	62.50%	81.25%	160.3
Taylor series	3	Swish	100.00%	71.88%	155.6
Chebyshev	4	*tanh*	100.00%	69.79%	181.4
Lagrange	4	*tanh*	96.88%	70.83%	181.8

Table 4: Experimental results for best implementation with each approximation method

Sample distribution	Degree	Testing accuracy
Uniform$[-0.5, 0.5]$	3	77.38%
Normal$(\mu = 0.0, \sigma = 1.0)$	3	77.48%
Normal$(\mu = 0.0, \sigma = 1.1)$	3	79.17%
Normal$(\mu = 0.0, \sigma = 1.2)$	3	81.25%
Normal$(\mu = 0.0, \sigma = 1.2)$	2	78.13%

Table 5: Results of regression approximation to Leaky ReLU with different settings

to the Leaky ReLU function. This method achieved 81.25% testing accuracy, which matches that recorded with an equivalent unencrypted CNN implemented without MPC, as may be seen in Table 3.

Based on the approach presented by Chabanne et al. [14], as outlined in Section 2.4, we used a sample of points taken from a standard normal distribution to derive our polynomial approximation. We also experimented with the parameters of the distribution that the sample was taken from by varying the value of the standard deviation, σ. The testing accuracy achieved by networks trained with degree-3 least squares regression approximations of Leaky ReLU with different values of σ may be seen in Table 5. We find an optimal value of $\sigma = 1.2$, which matches the results found by Chabanne et al. in the case of degree-2 polynomial approximations. We may also compare these results to those obtained using a polynomial approximation fitted with a uniformly sampled set of points. Through the hyperparameter tuning process, we found that sampling points uniformly in the range $[-0.5, 0.5]$ provided the most successful approximation, however the accuracy achieved did not exceed that of the implementation mentioned previously.

The results achieved by networks trained with *PPoly* activation functions of different forms may be seen in Table 6. Of particular note is the degree-3 polynomial, which achieved a testing accuracy of 87.50%, which is significantly higher than that achieved by any of the previously tested polynomial

Polynomial	Method	Train acc.	Test acc.	Runtime
$c_{i1}x_i + c_{i2}x_i^2$	[17]	15.66%	29.17%	205.8
$c_1x_i + c_2x_i^2$	Original	100.00%	79.17%	179.6
$c_1x_i + c_2x_i^2 + c_3x_i^3$	Original	90.63%	87.50%	142.1
$c_0 + c_2x_i^2 + c_4x_i^4$	Original	53.13%	68.75%	212.3

Table 6: *PPoly* activation function results

approximation methods. This also exceeds the testing accuracy achieved by an equivalent CNN computing over plaintext data, although the increase in running time of the model is significant.

We chose to initialise the *PPoly* activation function with a least squares regression approximation to the Leaky ReLU function, which achieved the highest accuracy in tests of the polynomial approximation methods, as described previously. We proposed previously that a better fitting polynomial approximation could be derived by allowing the network to tune the coefficients of an existing approximation, and our results support this hypothesis, showing an increase in testing accuracy of 6.25%. We also note in Table 6 that the method applied by Wu et al. [17] of initialising the polynomial coefficients randomly in the range $[-1.5, 1.5]$ achieved far lower testing accuracy than our proposed method of initialising the polynomial coefficients with those from a separate polynomial approximation of an existing activation function.

The running time of the most successful *PPoly* implementation, 142.1 minutes, is also the lowest of the results presented in Table 4 and Table 6. This is due to the low degree of the polynomial, and the early stopping of training after the second epoch, when we noticed that the validation accuracy was consistently decreasing with each mini-batch of data.

4.5. Discussion of Results

Figure 6 shows the plots of the degree-3 polynomial approximations obtained with each polynomial approximation method in comparison to the exact ReLU function.

As explained previously, our experiments must be performed on a smaller scale, making it difficult to directly compare the accuracy achieved by our models with the results reported in existing work. We therefore primarily compare the results achieved with different combinations of activation functions and approximation methods to judge their relative performance in our

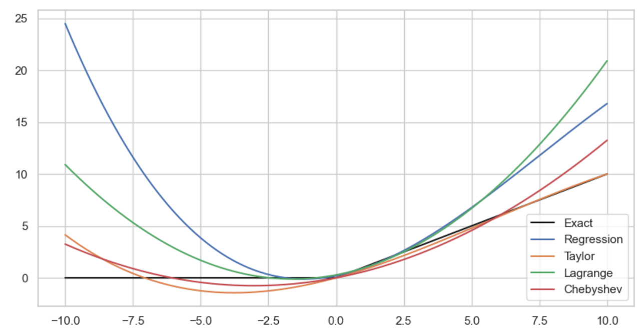


Figure 6: Degree-3 approximations of ReLU with each method in comparison to exact ReLU function

testing environment. Larger scale tests should be performed to confirm the efficacy of these findings and to directly compare them to existing work in the field.

From the results presented in Table 4, we may deduce that, out of the methods tested, least squares regression appears to be the most effective polynomial approximation method in terms of the accuracy achievable on the testing dataset. Through hyperparameter tuning we have been able to determine the optimal settings of the approximation, as shown in Table 5. Combined with the *PPoly* method, this approach may also be further tuned during training to achieve even better accuracy. While the highest testing accuracy was achieved using a degree-3 polynomial, we have also presented results for the equivalent degree-2 polynomial implementation. We note that although this method achieves somewhat lower accuracy than the optimal method, this may be viewed as an acceptable trade-off for the reduction in the computational complexity of the network.

Networks trained using low-degree *PPoly* activation functions generally achieve similar or higher accuracy than those using Taylor series, Chebyshev, and Lagrange polynomial approximations. In particular, our degree-2 *PPoly* network achieved 79.17% accuracy, beating that achieved by other methods which required degree-3 and 4 polynomials.

As mentioned previously, the highest testing accuracy was achieved with a degree-3 least squares regression approximation to the Leaky ReLU function, with *tanh* and Swish providing the best results for the other approximation methods. In the case of *tanh*, however, the optimal implementations required the use of degree-4 polynomials, which incur higher running times. Tests performed with *sigmoid* polynomial approximations consistently stalled during training and did not progress past the point of around 30% testing set accuracy. We recorded similarly poor results using the original *PrivateML sigmoid* approximation layer, which uses a degree-9 polynomial approximation. This may reflect the limitations of the *sigmoid* activation function, such as the vanishing gradient problem.

While the ReLU, Leaky ReLU, and Swish activation functions possess similar structures, we may note that Swish performed the best when tested with the optimal Chebyshev and Lagrange polynomial approximation settings. The Leaky ReLU, on the other hand, outperformed Swish when implemented with Taylor series and least squares regression fitted polynomials. This suggests that the performance of a polynomial approximation method may depend on the type of function that is being approximated, and that different methods may be better suited to different use-cases.

It is evident from the results presented that the approaches we have implemented have the potential to optimise the performance of privacy-preserving CNNs trained and tested using secret-sharing-based MPC. In order to preserve the practicality of the networks we train, the results we present focus on the application of low-degree polynomial approximations. We have demonstrated the significant improvements in accuracy that may be achieved with the application of private batch normalisation and parametric polynomial activation functions in an MPC setting, and have shown that the testing accuracy achieved can match or even exceed that attained by unencrypted models. The additional complexity that results from implementing *PPoly* activation functions appears to be negligible in comparison to the improvements in accuracy that may be achieved, and is also offset to an extent by the low degree of the polynomials that we may use with this method.

4.6. Security Analysis and Discussion

Throughout this paper we have assumed that the framework utilises a client-server protocol design, where secure computations are performed by a number of distributed servers. This approach was formalised by Araki et al. [18], who provide in their article rigorous proofs of the security level of their protocols, which they claim to be secure in the presence of one semi-honest adversarial party. This level of security is sufficient in cases when there is some level of trust between the parties involved in the scheme, but there is still concern about inadvertent information leakage, or the parties cannot reveal their input data to each other due to privacy regulations [18].

Additionally, the security of the private division and square root protocols we have implemented have been proven in the work of Wagh [16] where these protocols were introduced. The combination of these protocols that is needed to enable the application private batch normalisation does not compromise this security.

The use of polynomial approximated and parametric activation functions also does not impact the security or privacy of the networks being trained and tested. The coefficients of the polynomial approximated activation functions need not be kept private, as these values reveal no new information about the data being computed over or the inner weights of the model being trained. In the case of *PPoly* activation functions, the coefficients of the polynomials are stored as private values alongside the weights of the network, and so do not present any additional security risk. The values that these polynomials are initialised with may be kept private or public with no infringement on privacy, as long as the intermediary and final values of the coefficients are kept private, as these may be used to deduce information regarding the nature of the data being used for training.

5. Conclusions and Future Work

Our results give fresh insight into the efficacy of various polynomial approximation methods when applied to the training of privacy-preserving CNNs with MPC. As far as we are aware, ours is the first study in this field to perform a comprehensive evaluation of methods for approximating non-linear activation functions with polynomials, and the first to experiment with parametric polynomial activation functions. Our study may therefore form a reference point for future researchers and practitioners wishing to implement privacy-preserving CNNs with MPC. We are also the first to apply Lagrange polynomials for approximating non-linear activation functions, although the results achieved do not support the use of this method over other approaches, such as least squares regression.

We have demonstrated that it is possible to implement a privacy-preserving batch normalisation layer with MPC and provided empirical results to support our hypothesis that this would greatly increase the accuracy of the network at the cost of longer running times. We have additionally implemented and evaluated the novel approach of initialising a parametric polynomial activation function with a polynomial approximation and allowing the network to tune the coefficients as learning weights. The high testing accuracy and speed of convergence of models trained with this method, which can even exceed that of equivalent unencrypted models, show its potential for providing polynomial approximated activation layers that are highly optimised for the data and task at hand. As an extension to our work we would like to experiment with parametric polynomial activation functions of different forms and with different methods of initialisation.

Due to constraints on time and resources, we were required to perform scaled-down tests in order to keep running times feasible. This resulted in the overall accuracy achieved by our models being far lower than if the models had been trained using a larger training dataset and for a greater number of epochs. Because of this, it is difficult to compare our results to existing work in this field, due to the difference in scale of the values recorded. We were also restricted to using a shallow CNN, which limits the generalisability of our results. It is possible that different polynomial approximation methods may perform better or worse when applied in networks of varying depths and architectures, and this is something that we would like to explore in the near future. That being said, the experiments we have performed allow us to compare the results obtained with different techniques within our testing environment and to draw conclusions on their performance relative to each other and to an unencrypted baseline.

In order to perform more thorough experiments we would ideally implement the framework in a true client-server model, rather than a simulated environment in *Python*, using dedicated remote servers with more powerful hardware to perform computations. We would also reformat the code to be less memory-intensive and better scalable to larger networks. For example, we may integrate our framework with a distributed computation framework such as TensorFlow [31], which provides highly optimised protocols for unencrypted neural network training that could be adapted to an MPC setting.

References

[1] Q. V. Le, M. Schuster, A neural network for machine translation, at production scale, `https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html`, accessed: 2020-10-12 (2016).

[2] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, S. Thrun, Dermatologist-level classification of skin cancer with

deep neural networks, Nat. 542 (7639) (2017) 115–118. doi:10.1038/nature21056.
URL https://doi.org/10.1038/nature21056

[3] Y. LeCun, C. Cortes, C. J. Burges, Mnist handwritten digit database, http://yann.lecun.com/exdb/mnist/, accessed: 2020-10-12 (2010).

[4] R. Cramer, I. Damgård, J. B. Nielsen, Secure Multiparty Computation and Secret Sharing, Cambridge University Press, 2015.
URL http://www.cambridge.org/de/academic/subjects/computer-science/cryptography-cryptology-and-coding/secure-multiparty-computation-and-secret-sharing?format=HB&isbn=9781107043053

[5] K. Hornik, M. B. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (5) (1989) 359–366. doi:10.1016/0893-6080(89)90020-8.
URL https://doi.org/10.1016/0893-6080(89)90020-8

[6] A. L. Maas, A. Y. Hannun, A. Y. Ng, Rectifier nonlinearities improve neural network acoustic models, in: International Conference on Machine Learning (ICML), Vol. 1, 2013.

[7] Q. V. L. Prajit Ramachandran, Barret Zoph, Searching for activation functions (2018).
URL https://openreview.net/forum?id=SkBYYyZRZ

[8] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: 2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015, IEEE Computer Society, 2015, pp. 1026–1034. doi:10.1109/ICCV.2015.123.
URL https://doi.org/10.1109/ICCV.2015.123

[9] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: F. R. Bach, D. M. Blei (Eds.), Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, Vol. 37 of JMLR Workshop and Conference Proceedings, JMLR.org, 2015, pp. 448–456.
URL http://proceedings.mlr.press/v37/ioffe15.html

[10] A. C. Yao, How to generate and exchange secrets (extended abstract), in: 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986, IEEE Computer Society, 1986, pp. 162–167. doi:10.1109/SFCS.1986.25.
URL https://doi.org/10.1109/SFCS.1986.25

[11] P. Mohassel, Y. Zhang, Secureml: A system for scalable privacy-preserving machine learning, in: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, IEEE Computer Society, 2017, pp. 19–38. doi:10.1109/SP.2017.12.
URL https://doi.org/10.1109/SP.2017.12

[12] R. Seggers, K. L. Veen, C. Schaffner, Privately training cnns using two-party spdz, accessed: 2020-10-12 (2018).
URL https://homepages.cwi.nl/~schaffne/projects/reports/RubenSeggers_KoenvdVeen.pdf

[13] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, J. Wernsing, Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy, in: M. Balcan, K. Q. Weinberger (Eds.), Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, Vol. 48 of JMLR Workshop and Conference Proceedings, JMLR.org, 2016, pp. 201–210.
URL http://proceedings.mlr.press/v48/gilad-bachrach16.html

[14] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, E. Prouff, Privacy-preserving classification on deep neural network, IACR Cryptol. ePrint Arch. 2017 (2017) 35.
URL http://eprint.iacr.org/2017/035

[15] E. Hesamifard, H. Takabi, M. Ghasemi, Cryptodl: Deep neural networks over encrypted data, CoRR abs/1711.05189. arXiv:1711.05189.
URL http://arxiv.org/abs/1711.05189

[16] S. Wagh, New directions in efficient privacy-preserving machine learning, Ph.D. thesis, Princeton University, accessed: 2020-10-12 (5 2020).
URL https://snwagh.github.io/CV/thesis.pdf

[17] W. Wu, J. Liu, H. Wang, F. Tang, M. Xian, Ppolynets: Achieving high prediction accuracy and efficiency with parametric polynomial activations, IEEE Access 6 (2018) 72814–72823. doi:10.1109/ACCESS.2018.2882407.
URL https://doi.org/10.1109/ACCESS.2018.2882407

[18] T. Araki, J. Furukawa, Y. Lindell, A. Nof, K. Ohara, High-throughput semi-honest secure three-party computation with an honest majority, in: E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, S. Halevi (Eds.), Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, ACM, 2016, pp. 805–817. doi:10.1145/2976749.2978331.
URL https://doi.org/10.1145/2976749.2978331

[19] O. Catrina, A. Saxena, Secure computation with fixed-point numbers, in: R. Sion (Ed.), Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers, Vol. 6052 of Lecture Notes in Computer Science, Springer, 2010, pp. 35–50. doi:10.1007/978-3-642-14577-3_6.
URL https://doi.org/10.1007/978-3-642-14577-3_6

[20] M. Liedel, Secure distributed computation of the square root and applications, in: M. D. Ryan, B. Smyth, G. Wang (Eds.), Information Security Practice and Experience - 8th International Conference, ISPEC 2012, Hangzhou, China, April 9-12, 2012. Proceedings, Vol. 7232 of Lecture Notes in Computer Science, Springer, 2012, pp. 277–288. doi:10.1007/978-3-642-29101-2_19.
URL https://doi.org/10.1007/978-3-642-29101-2_19

[21] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Y. W. Teh, D. M. Titterington (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010, Vol. 9 of JMLR Proceedings, JMLR.org, 2010, pp. 249–256.
URL http://proceedings.mlr.press/v9/glorot10a.html

[22] B. Xu, N. Wang, T. Chen, M. Li, Empirical evaluation of rectified activations in convolutional network, CoRR abs/1505.00853. arXiv:

1505.00853.
URL http://arxiv.org/abs/1505.00853

[23] D. Pedamonti, Comparison of non-linear activation functions for deep neural networks on MNIST classification task, CoRR abs/1804.02763. arXiv:1804.02763.
URL http://arxiv.org/abs/1804.02763

[24] S. Hayou, A. Doucet, J. Rousseau, On the selection of initialization and activation function for deep neural networks, CoRR abs/1805.08266. arXiv:1805.08266.
URL http://arxiv.org/abs/1805.08266

[25] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362. doi:10.1038/s41586-020-2649-2.
URL https://doi.org/10.1038/s41586-020-2649-2

[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy, Scipy 1.0-fundamental algorithms for scientific computing in python, CoRR abs/1907.10121. arXiv:1907.10121.
URL http://arxiv.org/abs/1907.10121

[27] Chebyshev approximation in python, https://www.excamera.com/sphinx/article-chebyshev.html, accessed: 2020-10-12.

[28] H. P. Langtangen, Approximation of functions, http://hplgit.github.io/num-methods-for-PDEs/doc/pub/approx/html/approx.html, accessed: 2020-10-12 (2016).

[29] K. L. Veen, M. Dahl, R. Seggers, Privateml, https://github.com/koenvanderveen/privateml, accessed: 2020-10-12 (2018).

[30] A. Meurer, C. P. Smith, M. Paprocki, O. Certík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, S. Roucka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. M. Scopatz, Sympy: symbolic computing in python, PeerJ Comput. Sci. 3 (2017) e103. doi:10.7717/peerj-cs.103.
URL https://doi.org/10.7717/peerj-cs.103

[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, Tensorflow: A system for large-scale machine learning, in: K. Keeton, T. Roscoe (Eds.), 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, USENIX Association, 2016, pp. 265–283.
URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi