Ehsan Hesamifard*, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright

Privacy-preserving Machine Learning as a Service

Abstract: Machine learning algorithms based on deep Neural Networks (NN) have achieved remarkable results and are being extensively used in different domains. On the other hand, with increasing growth of cloud services, several Machine Learning as a Service (MLaaS) are offered where training and deploying machine learning models are performed on cloud providers' infrastructure. However, machine learning algorithms require access to the raw data which is often privacy sensitive and can create potential security and privacy risks. To address this issue, we present CryptoDL, a framework that develops new techniques to provide solutions for applying deep neural network algorithms to encrypted data. In this paper, we provide the theoretical foundation for implementing deep neural network algorithms in encrypted domain and develop techniques to adopt neural networks within practical limitations of current homomorphic encryption schemes. We show that it is feasible and practical to train neural networks using encrypted data and to make encrypted predictions, and also return the predictions in an encrypted form. We demonstrate applicability of the proposed CryptoDL using a large number of datasets and evaluate its performance. The empirical results show that it provides accurate privacy-preserving training and classification.

Keywords: Privacy, Machine Learning as a Service, Homomorphic Encryption, Deep Learning

DOI 10.1515/popets-2018-0024
Received 2017-11-30; revised 2018-03-15; accepted 2018-03-16.

1 Introduction

Machine learning algorithms based on deep Neural Networks (NN) have attracted attention as a breakthrough in the advance of artificial intelligence (AI) and are the mainstream in current AI research. These techniques are achieving remarkable results and are extensively used for analyzing big data in a variety of domains such as spam detection, traffic analysis, intrusion detection, medical or genomics predictions, face recognition, and financial predictions [9, 10, 24, 27, 32, 38, 44, 54]. However, training the models requires access to the raw data which is often privacy sensitive and can create potential privacy risks.

Furthermore, with increasing growth of cloud services, machine learning algorithms can be run on the cloud providers' infrastructure where training and deploying machine learning models are performed on cloud servers. Once the models are deployed, users can use these models to make predictions without having to worry about maintaining the models and the service. In a nutshell, this is Machine Learning as a Service (MLaaS), and several such services are currently offered including Microsoft Azure Machine Learning [33], Google Prediction API [16], GraphLab [18], and Ersatz Labs [12]. Machine learning algorithms typically consist of two phases: (i) the training phase during which the algorithm learns a model w from a data set of labeled examples, and (ii) the classification phase that runs a classifier C over a previously unseen feature vector x, using the model w to output a prediction $C(x, w)$. Either of these phases or both could be outsourced to the cloud. In applications that handle sensitive data, it is important that the training data, the feature vector x and the model w remain secret to one or some of the parties involved.

In this paper, we present CryptoDL, a client-server solution to apply neural network algorithms to encrypted data and allow the parties to provide/receive the service without having to reveal their sensitive data to the other parties. The main components of CryptoDL are homomorphic encryption and neural networks.

Threat Model: We consider a client-server model where the client owns the data and the server uses this

***Corresponding Author: Ehsan Hesamifard:** Department of Computer Science and Engineering, University of North Texas, Denton, TX, USA, E-mail: ehsanhesamifard@my.unt.edu
Hassan Takabi: Department of Computer Science and Engineering, University of North Texas, Denton, TX, USA, E-mail: takabi@unt.edu
Mehdi Ghasemi: Department of Mathematics and Statistics, University of Saskatchewan, Saskatoon, Canada, E-mail: mehdi.ghasemi@usask.ca
Rebecca N. Wright: Department of Computer Science, Rutgers University, Piscataway, NJ, USA, rebecca.wright@rutgers.edu

data to build a model and provide prediction service to the client. In the training phase, the server gets the training dataset from the client in an encrypted format, and can train a model on that data without learning anything about the training dataset. Since training is performed on encrypted data, the server doesn't learn the parameters of the trained model or the final model. The trained model is encrypted under the client's public key, and only this client can use the final model for classification of new instances. In the classification phase, the client sends encrypted input to the server, the server performs an inference on the encrypted data, and the client gets the encrypted prediction. Hence, the server cannot access the input data or the prediction.

The original fully homomorphic encryption (FHE) scheme presented in Gentry's seminal paper [14] was highly inefficient but since then, significantly more practical schemes have been developed. In order to have efficient and practical solutions for computations in encrypted domain, we typically use somewhat homomorphic schemes or leveled homomorphic schemes instead of fully homomorphic encryption. The most notable shortcoming of practical homomorphic encryption schemes is that operations in practical schemes are limited to addition and multiplication. Consequently, we need to adopt neural network algorithms within these limitations. The computation performed over sensitive data by neural network algorithms is very complex and neural networks cannot simply be translated to encrypted versions without modification. Some examples of activation functions in neural networks are Sigmoid function ($\frac{1}{1+e^{-x}}$) and Rectified Linear Unit (ReLU, $f(x) = max(x, 0)$) and they should be replaced by a function that only uses addition and multiplication such as polynomials.

However, a solution that builds upon homomorphic encryption schemes should be restricted to computing low degree polynomials in order to be practical [49]. Hence, presenting the desired computation as low-degree polynomials is an important task in making practical use of homomorphic encryption schemes for applying neural network algorithms to encrypted data. Polynomials of degree 2 are used to substitute the Sigmoid function in neural networks [10] and polynomials of degree 3 are used to estimate the natural logarithm function [44]. However, these are arbitrary solutions which enable us to work around certain problems, but there is no generic solution to the problem of approximating a function with low degree polynomials. Our goal is to provide a framework capable of handling general cases.

Once the polynomial approximations are generated, we design and evaluate privacy-preserving training and classification of neural networks by replacing the activation functions with these polynomial approximations. Our goal is to adopt neural networks within practical limitations of homomorphic encryption while keeping accuracy as close as possible to the original model.

Our main contributions in this paper are as follows:

- To the best of our knowledge, our proposed CryptoDL is the first work that provides a solution for training neural network models using homomorphic encryption.
- We provide the theoretical foundation and prove that it is possible to find lowest degree polynomial approximation of a function within a certain error range.
- We propose two methods for approximating continuous functions -ReLU, Sigmoid and Tanh functions- with low degree polynomials and utilize these polynomials in neural networks and analyze the performance of the new algorithms.
- We implement full neural networks over homomorphically encrypted data where activation functions (Sigmoid and ReLU) are replaced with polynomial approximations. To avoid costly bootstrapping of homomorphic encryption, our proposed approach allows some communications between client and server to manage the noise.
- We use a large number of datasets from the UC Irvine Machine Learning Repository [29], MNIST dataset [51] and CIFAR-10 dataset [28] to perform experiments and present empirical results of both training and classification phases.
- We provide detailed comparison with state-of-the-art approaches based on homomorphic encryption (HE) and secure multi-party computation (SMC). The results show that CryptoDL provides accurate and privacy-preserving training and classification and outperforms both HE-based and SMC-based approaches.

The rest of this paper is organized as follows: In Section 2, we provide the theoretical foundation for polynomial approximation and describe our proposed solution in Section 3. In Section 4, we provide results for building models based on encrypted datasets. In Section 5, we review related work. In Section 6, we conclude the paper and discuss future work.

2 Theoretical Foundation: Polynomial Approximation

Since our goal is to adopt a neural network to work within practical homomorphic encryption constraints, we focus on operations performed inside a neural network rather than its structure. Besides activation functions inside neurons (e.g. the "Sigmoid function"), all other operations in a neural network are addition and multiplication, so they can be implemented over encrypted data. Activation functions, on the other hand, cannot be implemented within practical homomorphic encryption schemes. Hence, in order to operate a complete neural network over encrypted data, we replace activation functions with approximations that are compatible with practical HE schemes as discussed below.

Among continuous functions, perhaps polynomials are the most well-behaved and easiest to compute. Thus, it is no surprise that mathematicians tend to approximate other functions by polynomials. Materials of this section are mainly folklore knowledge in numerical analysis and Hilbert spaces. For more details on the subject refer to [4, 50].

Let us denote the family of all continuous real valued functions on a non-empty compact space X by $\mathrm{C}(X)$. Suppose that among elements of $\mathrm{C}(X)$, a subfamily A of functions are of particular interest. For simplicity, one can think of X as a closed, bounded interval $[a, b]$ in $\mathbb{R}$ and A as the set of polynomials in a single variable with real coefficients. Since linear combination and product of polynomials are also polynomials, we assume that A is closed under addition, scalar multiplication and product and also a non-zero constant function belongs to A (This actually implies that A contains all constant functions).

We say an element $f \in \mathrm{C}(X)$ can be approximated by elements of A, if for every $\epsilon > 0$, there exists $p \in A$ such that $|f(x)-p(x)| < \epsilon$ for every $x \in X$. The following classical results guarantee when every $f \in \mathrm{C}(X)$ can be approximated by elements of A.

Theorem 1 (Stone–Weierstrass)**.** *Every element of $C(X)$ can be approximated by elements of A if and only if for every $x \neq y \in X$, there exists $p \in A$ such that $p(x) \neq p(y)$.*

Despite the strong and important implications of the Stone-Weierstrass theorem, it leaves computational details out and does not give a specific algorithm to generate an estimator for f with elements of A, given an error tolerance ϵ. We address this issue here.

We define $\|f\|_\infty$ (the sup norm of f) of a given function $f \in \mathrm{C}(X)$ by $\|f\|_\infty = \sup_{x\in X} |f(x)|$,

Then the above argument can be read as: *For every $f \in C(X)$ and every $\epsilon > 0$, there exists $p \in A$ such that $\|f - p\|_\infty < \epsilon$.* It is easy to see that $\|0\|_\infty = 0$, $\|\lambda f + g\|_\infty \leq |\lambda|\|f\|_\infty + \|g\|_\infty$ (subadditivity) and $\|f \times g\|_\infty \leq \|f\|_\infty \times \|g\|_\infty$. The function $\| \cdot \|_\infty$ on $\mathrm{C}(X)$ is an instance of the *norm* on the function space $\mathrm{C}(X)$, which also resembles the structure of inner product spaces which have nice geometry and one can define and develop intuitive concepts over them easily. Let V be an $\mathbb{R}$-vector space, an *inner product* on V is a function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ satisfying the following requirements:

1. $\langle f, f\rangle \geq 0$;
2. $\langle f, f\rangle = 0$ if and only if $f = 0$;
3. $\langle \alpha f + \beta g, h\rangle = \alpha\langle f, h\rangle + \beta\langle g, h\rangle \geq 0$ for every $\alpha, \beta \in \mathbb{R}$;
4. $\langle f, g\rangle = \langle g, f\rangle$.

The pair $(V, \langle \cdot, \cdot \rangle)$ is called an inner product space and the function $\|v\| = \langle v, v\rangle^{\frac{1}{2}}$ induces a norm on V. A basis $\{v_\alpha\}_{\alpha\in I}$ is called an orthonormal basis for V if $\langle v_\alpha, v_\beta\rangle = \delta_{\alpha\beta}$, where $\delta_{\alpha\beta} = 1$ if and only if $\alpha = \beta$ and is equal to 0 otherwise. Every given set of linearly independent vectors can be turned into a set of orthonormal vectors that spans the same sub vector space as the original. The following well-known theorem gives us an approach for producing such orthonormal vectors from a set of linearly independent vectors:

Theorem 2 (Gram–Schmidt)**.** *Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space. Suppose $\{v_i\}_{i=1}^n$ is a set of linearly independent vectors in V. Let $u_1 := \frac{v_1}{\|v_1\|}$ and (inductively) let $w_k := v_k - \sum_{i=1}^{k-1}\langle v_k, u_i\rangle u_i$ and $u_k := \frac{w_k}{\|w_k\|}$. then $\{u_i\}_{i=1}^n$ is an orthonormal collection, and for each k, $span\{u_1, u_2, \cdots, u_k\} = span\{v_1, v_2, \cdots, v_k\}$.*

Note that in this theorem, we can even assume that $n = \infty$.

Let $B = \{v_1, v_2, \dots\}$ be an ordered basis for $(V, \langle \cdot, \cdot \rangle)$. For any given vector $w \in V$ and any initial segment of B, say $B_n = \{v_1, \dots, v_n\}$, there exists a unique $v \in \mathrm{span}(B_n)$ such that $\|w - v\|$ is the minimum as shown in the following theorem.

Theorem 3. *Let $w \in V$ and B be a finite orthonormal set of vectors (not necessarily a basis). Then for $v = \sum_{u\in B}\langle u, w\rangle u$, $\|w - v\| = \min_{z\in span(B)} \|w - z\|$.*

For proof see [4, 50].

Now, let μ be a finite measure on X and for $f, g \in \mathrm{C}(X)$ define $\langle f, g\rangle = \int_X fg d\mu$. This defines an inner product on the space of functions. The norm induced by the inner product is denoted by $\|\cdot\|_{2,\mu}$. It is evident that $\|f\|_{2,\mu}\leq\|f\|_\infty\mu(X),\ \forall f \in \mathrm{C}(X)$,

which implies that any good approximation in $\|\cdot\|_\infty$ gives a good $\|\cdot\|_{2,\mu}$-approximation. But generally, our interest is the other way around. Employing Gram–Schmidt procedure, we can find $\|\cdot\|_{2,\mu}$ within any desired accuracy, but this does not guarantee a good $\|\cdot\|_\infty$-approximation. The situation is favorable in finite dimensional case. Take $B = \{p_1, \ldots, p_n\} \subset \mathrm{C}(X)$ and $f \in \mathrm{C}(X)$, then there exists $K_f > 0$ such that for every $g \in \mathrm{span}(B \cup \{f\})$,

$$K_f\|g\|_\infty \leq \|g\|_{2,\mu}\leq\|g\|_\infty\mu(X). \tag{1}$$

Since X is assumed to be compact, $\mathrm{C}(X)$ is separable, i.e., $\mathrm{C}(X)$ admits a countable dimensional dense subvector space (e.g. polynomials for when X is a closed, bounded interval). Thus for every $f \in \mathrm{C}(X)$ and every $\epsilon > 0$, one can find a big enough finite B, such that (1) holds. In other words, *"good enough $\|\cdot\|_{2,\mu}$-approximations of f give good $\|\cdot\|_\infty$-approximations"*, as desired. This is particularly useful, since the Gram–Schmidt procedure provides a concrete algorithm to compute best $\|\cdot\|_2$-approximations, while just computing $\|\cdot\|_\infty$ in general is an NP-complete task.

In practice, $X = [a, b]$ and the countable dimensional subspace is the algebra of polynomials which satisfies the assumption of the Stone–Weierstrass theorem and the set of monomials is admissible in the Gram–Schmidt process. Different choices of μ, gives different systems of orthogonal polynomials. Two of the most popular measures are $d\mu = dx$ and $d\mu = \frac{dx}{\sqrt{1-x^2}}$. By using $d\mu = dx$ on $[-1, 1]$, the generated polynomials are called Legendre polynomials and by using $d\mu = \frac{dx}{\sqrt{1-x^2}}$ on $[-1, 1]$ the generated polynomials are called Chebyshev polynomials.

These two polynomial sets have different applications in approximation theory. For example, the nodes we use in polynomial interpolation are the roots of the Chebyshev polynomials and the Legendre polynomials are the coefficient of the Taylor series. For more details about these polynomials, see [4, 50].

In this paper, we experiment with polynomial approximations of the Sigmoid function $\frac{1}{1+e^{-x}}$ over a symmetric interval $[-l, l]$ using two different orthogonal system of polynomials. As the first choice, we consider Chebyshev polynomials on the stretched interval which come from the measure

$$d\mu = \frac{dx}{l\sqrt{1-(x/l)^2}}. \tag{2}$$

Our second choice comes from the measure.

$$d\mu = e^{-(l/x)^2}dx. \tag{3}$$

We note that the measure for Chebyshev polynomials mainly concentrates at the end points of the interval which causes interpolation at mostly initial and end points with two singularities at both ends. While the second measure evens out through the whole real line and puts zero weight at the center. This behavior causes less oscillation in the resulting approximation and hence more similarities of derivatives with the Sigmoid function.

We provide two different output sets from our approximation algorithms in Tables 8 and 9 of Appendix. We first choose the interval $[-10^3, 10^3]$ for approximation and approximate the Sigmoid function with four different degrees, $\{3, 5, 7, 9\}$. For each degree, we have two polynomials from two orthogonal systems of polynomials. The reason we choose these specific intervals is that homomorphic encryption schemes only accept integers. We use SageMath [8] to find the coefficients of polynomial approximations for the Sigmoid function (code is available on [15]).

As can be seen in Table 8, the higher degrees give a better approximation and we expect that a better approximation to be a better replacement for the Sigmoid function in the neural network. However, when the degree increases, the coefficients become very small, for example $9.888 * 10^{-31}$. We truncate the coefficients and only show three digits precision.

Next, we keep the degree fixed (equal to 5) and approximate the function on intervals $[-10^i, 10^i]$ for $1 \leq i \leq 4$. As we can see in Table 9, the approximation is more accurate on small intervals. Another observation is that we have smaller coefficients on larger intervals.

3 Polynomial Approximation in Neural Networks

Once the polynomial approximations are calculated, we replace the activation function in the neural network with polynomial approximations and analyze the performance of the neural network shown. We utilize Neural Network Toolbox [37] to implement the neural network and use a large number of datasets from the UC Irvine Machine Learning Repository [29] as shown in Table 1 to evaluate the polynomial approximations.

Table 1. Datasets (I, F and C represent Instances, Features and Classes respectively)

	Name	#I	# F	#C
1	Arcene	900	10000	2
2	Banknote Authentication	1372	4	2
3	Blood Transfusion Service Center	748	4	2
4	Breast Tissue	106	9	6
5	Cardiotocography (3 classes)	2126	21	3
6	Cardiotocography (10 classes)	2126	21	10
7	Climate Model Simulation Crashes	540	17	2
8	CNAE-9	1080	857	9
9	Connectionist Bench (Sonar, Mines vs. Rocks)	208	60	2
10	Connectionist Bench (Vowel Recognition)	528	10	2
11	Crab	200	6	2
12	Daphnet Freezing of Gait	1048576	10	2
13	Fertility	100	10	2
14	First-Order Theorem Proving	6118	51	6
15	Ovarian	216	100	2
16	Wine	178	13	3
17	Sensorless Drive Diagnosis	58509	49	11

3.1 Performance of Neural Networks with Polynomial Approximations

We compute polynomial approximations for the Sigmoid function based on four different parameters: degree, error, intervals and precision of coefficients. We change the degree of polynomials from 2 to 9. Our approximation method gets an error values as the input. We consider the following values for the error parameter: $\{0.1, 0.01, 0.001, 1e-07, 1e-11\}$. We also choose intervals as follows: $[-10^i, 10^i]$, $0 \leq i \leq 5$. The coefficients are very small and we have to truncate them. For this reason, we truncate coefficients with different precisions: 10, 20, 30 and 40 digits. We also calculate the $\|\cdot\|_{2,\mu}$ for each polynomial for a more precise analysis. We generate 1920 polynomials (by assigning all the possible values to all the parameters) and train the neural network using each polynomial.

To compare the performance of our polynomial approximations with other studies, we use several activation functions for training: the Sigmoid function $f_1(x) = \frac{1}{1+e^{-x}}$, another variation of the Sigmoid function $f_2(x) = \frac{2}{1+e^{-4x}} - 1$, and the square function $f_3 = x^2$ which was proposed in [31] and [10]. We use the four different activation functions and if we choose the interval properly based on the dataset, our model could achieve the best accuracy among all four activation functions. In order to find the interval that gives the best accuracy, we extract three values from each dataset. We first build a vector for each dataset where each item in the vector is the mean of the feature values. Then, we compute maximum (maxMean), minimum (minMean) and mean (meanMean) of values in this vector and generate polynomials in these intervals: [-maxMean, maxMean], [-minMean, minMean] and [-meanMean, meanMean]. We train the neural network for each interval and the results show that the interval [-meanMean, meanMean] has the best accuracy. As it can be seen in Table 2, in our experiments, neural networks with polynomial approximation

```
Input: Dataset, MaxDegree, ActivationFunction
Output: A set of Polynomials

$MeanArray \leftarrow \{\}$;
$Mean = 0$;
$PolynomialSet \leftarrow \{\}$;
for $i \leftarrow 1$ to $\#Features$ do
    $mean \leftarrow$ mean of $Feature[i]$;
    $MeanArray \leftarrow MeanArray \cup \{mean\}$;
end
$Mean \leftarrow$ mean of $MeanArray$;
for $i \leftarrow 1$ to $MaxDegree$ do
    $poly \leftarrow$ Approximate the ActivationFunction
      in the interval $[-Mean, Mean]$ with
      precision 10 and degree $i$;
    $PolynomialSet \leftarrow PolynomialSet \cup \{poly\}$;
end
return $PolynomialSet$;
```

Algorithm 1: Generating Polynomial Approximations

Table 2. Performance of the neural network model based on the following activation functions in layer 2 only: $f_1(x) = \frac{1}{1+e^{-x}}$, $f_2(x) = \frac{2}{1+e^{-4x}} - 1$, the square function $f_3 = x^2$ and the polynomial approximation. a, i, d and n represent accuracy -$\frac{(TP+TN)}{(TP+TN+FP+FN)}$-, number of iterations, degree of polynomial and $\| \cdot \|_{2,\mu}$ respectively. Precision -the value for truncating the coefficients of the polynomial- is equal to 10, except Datasets 3 and 9 which are 20 and 15. We report the polynomial approximation with minimum degree. Refer to Table 1 for the list of datasets.

Nom	$f_1(x)$		$f_2(x)$		$f_3(x)$		Polynomial			
	a	i	a	i	a	i	a	d	i	n
1	86.67%	18	56.67%	7	30.00%	1000	90.00%	2	40	0.005
2	100.00%	69	100.00%	85	40.29%	6	100%	3	34	0.118
3	79.46%	15	81.25%	25	83.93%	13	83.93%	3	38	0.005
4	67.00%	30	68.75%	16	18.75%	1000	75.00%	2	23	0.001
5	98.58%	30	89.34%	43	79.31%	1000	99.37%	2	42	0.001
6	82.45%	55	83.07%	74	15.67%	1000	89.97%	2	45	0.001
7	95.37%	21	91.36%	11	87.65%	11	96.30%	2	23	0.001
8	96.30%	179	66.05%	107	7.40%	1000	97.53%	2	106	0.001
9	77.42%	15	83.87%	22	54.84%	1000	93.55%	3	27	0.000
10	86.67%	33	75.84%	66	12.08%	52	96.30%	2	23	0.001
11	100%	45	100%	31	66.66%	1000	100%	2	30	0.001
12	90.14%	448	90.19%	243	90.27%	49	90.26%	3	63	0.000
13	86.67%	24	86.67%	13	40.00%	8	86.67%	2	7	0.001
14	100%	48	100%	51	16.45%	1000	100%	2	48	0.001
15	93.75%	25	96.87%	1000	56.25%	41	96.87%	2	13	0.001
16	96.29%	37	96.29%	45	44.44%	7	96.29%	2	20	0.001
17	86.29%	359	86.29%	368	42.22%	1000	89.87%	2	373	0.001

as activation function achieve the best accuracy in all cases.

Algorithm 1 shows the process of choosing the best polynomial approximation based on the dataset and its features. Informally, we first extract the mean of values from features (meanMean) and then generate the polynomial approximation in the [-meanMean, meanMean] interval. Based on our extensive experiments, this approach always lead to the best polynomial approximation [45].

3.2 Error Rate of Neural Networks with Polynomial Approximations

In the previous section, we showed that current activation functions in neural networks could be replaced with polynomial approximation without causing any accuracy loss on the model. To further confirm the suitability of the polynomial approximations, we also compare the error rate of neural networks with the original activation function (Sigmoid function) and with polynomial approximations as replacement. To provide a precise and fair comparison, we calculate the error rate over the same test set for all activation functions. This approach was previously used in [26] for comparing performances of different activation functions. We first split the dataset into training set and test set. We then train the neural network using the training set phase by phase. We start with the first 10% of the training set to build the model and calculate the error rate, then add another 10% of the training data and repeat the process until we have used the entire training data. The training set is kept the same during all the corresponding phases for all the activation functions. We report the results for four datasets, and as shown in Figure 1, the error rate in the case of the polynomials is almost the same as in the case of other activation functions. Now, we conclude with higher confidence that approximation of an activation function with a polynomial using our proposed method is a practical replacement for the currently used activation functions in neural networks.

4 Experimental Results

In this section, we present results of implementing neural networks over encrypted data. We used the HELib for implementation and all the computations were run on a virtual machine with 48GB RAM, 12 CPU cores and Ubuntu 14.04. First, we train the models based on the encrypted data and measure the running time for the training process. Then, we use the trained model to

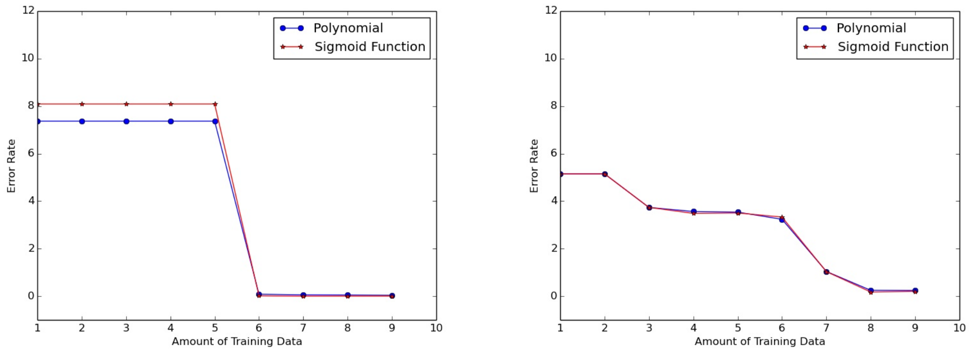


Fig. 1. Comparing the error rate of the models generated using polynomial approximation and Sigmoid function for four different datasets.

classify encrypted instances and measure the accuracy of the model as well as the running time.

For generating encryption schemes in the HELib, we set values for input parameters as $k = 80$ (security level which is equivalent to AES-128) and $s = 0$ (number of slots in the ciphertext) which allows the HELib to automatically pick the best number of slots. Another important value is L which is set to 20 in our experiments. We report the average running time from different rounds of algorithms and the standard deviation of running times is less than 3.0% in all cases.

In this paper, we focus on fully connected feed-forward neural networks. Current implementations which work with plain data, usually have three classes: *net*, *layer* and *neuron*. Each neuron stores input values, output values, weights and other required parameters. A layer is an array of neurons and neural network is a combination of these layers in a specific order. During the feed-forward or back-propagation functions, we go through all layers and in each layer, we process all the neurons in that layer.

We should also consider the growing noise in the ciphertexts during computations. As mentioned before, one technique to deal with the noise is bootstrapping. However, bootstrapping comes with heavy computation cost. To address this problem, we use an alternative approach where the server checks the level of the noise in the ciphertext after each operation. If the noise level is above the threshold, the server sends the ciphertext to the client and the client decrypts and encrypts it again and sends the fresh ciphertext back to the server. Therefore, in each round of communication, one cipher refreshed by the client for reducing the noise.

The number of operations allowed on the ciphertext depends on the value of L. If we set a small value for L, we need more communications between the client and the server whereas for higher values of L, less communications are required. However, by increasing the value of L, both the size of ciphertexts and the amount of transferred data increase.

4.1 Results of Training Phase

The training phase in neural network is a one time process and requires heavy computation. The process of training is based on two main functions: *feed-forward* and *back-propagation*. When the network gets encrypted instances, first it runs the feed-forward and then the back-propagation functions on them. The network goes through all instances and at the end, outputs a model. Since we feed a large dataset to the network for training, noise growth is a big challenge. To efficiently implement the training phase, we define a neuron class and define a ciphertext for each variable like weight or output. In this approach, we can use paralleled structure and the noise growth is manageable. However, the size of network grows quickly.

We implement neural networks over encrypted data with different numbers of hidden layers (1, 2, 3, 4 and 5). We implement the neural networks in the HELib and use three different datasets: Crab, Fertility and Climate Model (see Table 1) to perform the experiments. The results are shown in Tables 3a, 3b and 3c. The results show that if we use batch learning, the performance of the training phase is acceptable.

The HELib supports SIMD feature and the running time for one instance is the same as the time for a batch of instances. We also train the neural network with different sizes of batch as an input (282, 576, 1420, 3668 and 6144) and calculate the running time for feed-

Table 3. Training Neural Network over Encrypted Data (batch input size of 576 and $L = 20$; standard deviation is less than 2.9%). NC and #C represent Network Creation and Number of Communications.

(a) Crab dataset.

#Hidden layer(s)	NC(s)	Feedforward(s)	Backpropagation(s)	#C	Noise Reduction(s)	Total Time(s)
1	42.33	59.80	109.81	77	149.03	217.07
2	56.97	130.26	202.12	192	312.27	394.38
3	96.22	289.88	412.85	452	682.82	803.95
4	119.22	383.72	548.42	600	912.07	1056.5
5	125.63	407.64	582.13	640	969.58	1120.70

(b) Fertility dataset.

#Hidden layer	NC(s)	Feedforward(s)	Backpropagation(s)	#C	Noise Reduction(s)	Total Time(s)
1	55.24s	65.30	121.83	77	160.17	247.89
2	72.42s	138.17	220.69	192	331.60	436.99
3	112.66s	305.51	440.44	452	718.36	864.60
4	131.20s	390.96	562.10	600	926.35	1089.86
5	140.52s	419.80	602.63	640	995.61	1168.47

(c) Climate Dataset

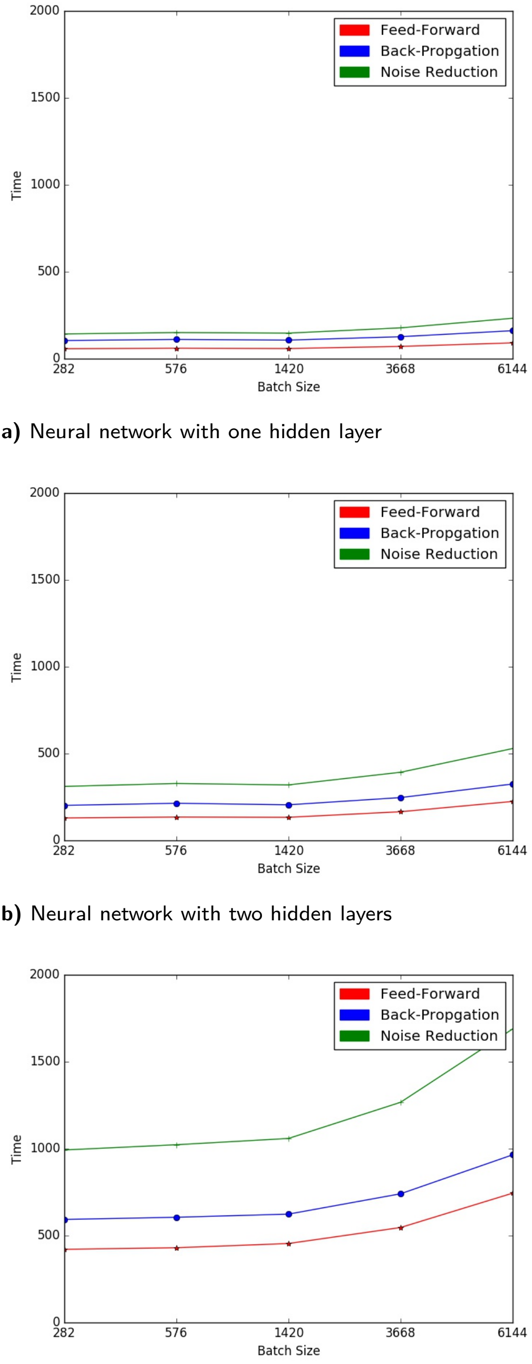
#Hidden layer(s)	NC(s)	Feedforward(s)	Backpropagation(s)	#C	Noise Reduction(s)	Total Time(s)
1	92.73	86.69	170.85	77	210.98	357.40
2	11.34	161.42	273.74	192	388.69	553.61
3	112.66	333.65	497.64	452	780.07	989.26
4	165.77	416.37	638.44	600	1011.63	1228.38
5	175.25	430.35	843.84	640	1229.50	1456.7

forward, back-propagation and noise reduction for one, two, and five hidden layers. Figures 2a, 2b and 2c show the result for one, two and five hidden layers respectively. As it can be seen, when we increase the size of the batch by 30 times, the running time only doubles. In the above experiments, we only run the experiment for one iteration. To analyze how the the number of iterations affects the feed-forward, back-propagation, noise reduction times and the number of communications, we repeat the neural network training with varying number of iterations.

Our empirical results show that training over encrypted data is efficient when batch learning is used, and the network performance is acceptable. For example, we reach the training rate of 0.68 seconds per instance (6 features) for a batch size of 576 and two hidden layers (see Table 3a). By increasing the batch size to 6144, the training rate decreases to 0.10 second per instance for a neural network with two hidden layers. It is worth mentioning that although larger batch sizes lead to faster training, they also increase the size of the network. Hence, a trade-off between the memory and running time is needed and we should choose a proper batch size based on the size of the dataset.

4.2 Results of Classification Phase

In the classification phase, we only run the feed-forward step and can perform the computation in parallel for a batch of unseen instances. Note that we implement classification over encrypted data while the model is also encrypted. In this phase, noise growth is not a big challenge and the run-time is more important. The results are shown in Tables 4a, 4b and 4c. For the classification with a batch size of 576, we reach 0.04 second per instance for one hidden layer and 0.1 second per instance for two hidden layers. If we choose a batch of size of 6144, we reach 0.014 second per instance for one hidden layer and 0.036 second per instance for two hidden layers. The model is encrypted in all training and classification experiments and also we did not use any parallelization techniques in our implementation.



(a) Neural network with one hidden layer

(b) Neural network with two hidden layers

(c) Neural network with five hidden layers

Fig. 2. Running time for different batch sizes.

Table 4. Classification over Encrypted Data (batch size 576 and $L = 20$; standard deviation is less than 3%.) #HL and #C represent for number of hidden layers and number of communications.

(a) Crab dataset.

#HL(s)	Classification(s)	#C	Noise Reduction(s)
1	28.62	0	0
2	61.17	0	0
3	154.20	0	0
4	229.28	10	155.87
5	234.65	21	171.20

(b) Fertility dataset.

#HL(s)	Classification(s)	#C	Noise Reduction(s)
1	35.94	0	0
2	68.86	0	0
3	160.49	0	0
4	236.72	10	152.34
5	255.70	21	169.84

(c) Climate dataset.

#HL(s)	Classification(s)	#C	Noise Reduction(s)
1	55.85	0	0
2	61.78	0	0
3	178.25	0	0
4	249.54	10	149.10
5	269.39	21	166.89

4.3 Comparison with state-of-the-art HE-based Approaches

In this section, we compare our results with the state-of-the-art privacy-preserving neural networks based on homomorphic encryption. CryptoNets is the closest work to ours where it aims to implement neural networks classification using HE [10].

The main difference between our approach and CryptoNets is that our approach performs both training and classification phases over encrypted data whereas CryptoNets only considers the classification phase which is much simpler compared to the training. The authors of CryptoNets assume that the model is trained based on plain data and then use this model for classifying encrypted instances. In our approach, the model is built using encrypted data and the final model is encrypted. When the model is trained using plain data, the final model is also in plain format and works much faster than an encrypted model.

In order to provide a fair comparison, we change our implementation discussed earlier and follow the same process as the CryptoNets: train a model based on the plain data and then classify encrypted instances. In CryptoNets, the authors implement a Convolutional Neural Network (CNN) with two convolutional layers and two fully connected layers. They replace the Max Pooling layer with Average Pooling layer and replace the activation function (i.e., Sigmoid function) with the square function. We design a CNN with the same depth (the number of multiplication required) and similar structure. We use an average pooling layer similar to CryptoNets. The MNIST dataset [51] is used for experiments which includes 60000 images where 50000 images are used for training and 10000 images for testing, each image us a 28×28 pixel, and each pixel is an integer value in the interval $[0, 255]$.

We train the same model with ReLU as the activation function instead of the Sigmoid function. In this case, the accuracy of the trained model is 99.02% and when we replace the activation function (i.e., RELU) with polynomial approximation, the accuracy is 99.00% whereas the accuracy of the similar model in CryptoNets is 98.95%. When we used more complex neural network, we were able to achieve 99.25% accuracy which is very close to the accuracy of the same model that uses the ReLU function and significant improvement over CryptoNets. The accuracy of this model with the Sigmoid function as activation function is 99.10% and when we replace the Sigmoid function with the polynomial approximation, we achieve 99.15% accuracy. For the Tanh function as the activation function, the original accuracy is 97.27% and the accuracy with polynomial approximation is 98.15%.

In terms of throughput, our approach significantly outperforms CryptoNets. When we classify ciphertext with the batch size of 8192 (the same batch size used in CryptoNets), our approach can make 163840 predictions per hour whereas CryptoNets makes only 51739 predictions per hour. The running time for classifying this batch input size is 570 for CryptoNets whereas in our approach, the running time is 320 seconds. The size of the data transferred from the client to the server in CryptoNets is 570Mb and in our approach is 336.7MB. Note that to provide a fair comparison, we use machines with similar configuration (Intel Xeon E5-1620 CPU running at 3.5GHz with 16GB of RAM in CryptoNets and Intel Xeon E5-2640, 2.4GHz with 16GB RAM in our case) for the experiments.

To further show applicability of our proposed approach for more complicated network architectures, we use the CIFAR-10 [28] which is one of the widely used benchmark datasets for deep learning, to train a neural network and implement it over encrypted data. The CIFAR-10 dataset consists of 60000 32×32 colour images categorized in 10 classes, with 6000 images per class. There are 50000 training and 10000 test images. We trained convolutional neural networks using the CIFAR-10 dataset and achieved 91.5% accuracy with polynomials as the activation functions whereas the accuracy with the original activation function (ReLU) is 94.2%. As expected, the CIFAR-10 is much slower compared to the MNIST since both the dataset and the CNN are much more complex.

4.4 Comparison with state-of-the-art SMC-based Approaches

Although our approach is based on homomorphic encryption, it does not use bootstrapping due to huge computation cost and instead requires some communications between the client and the server. One may argue that why not use secure multi-party computation (SMC) instead of homomorphic encryption. Although our approach requires some communications, it has several advantages over SMC-based approaches as discussed below.

While privacy-preserving machine learning based on secure multi-party computation techniques have been studied in the literature, those approaches focus on traditional machine learning algorithms such as linear regression [6], decision trees [2, 5, 53] and linear classifiers [5, 19]. There are a few recent works that focus on neural networks and aim to develop privacy-preserving training [34] and prediction [25, 39, 41].

To the best of our knowledge, SecureML [34] is the only work based on SMC techniques that considers both training and classification phases. The others only focus on classification phase. In the following, we provide a detail comparison of our proposed approach with these efforts.

The recent work of Mohassel and Zhang [34] aims to develop privacy-preserving training and classification of neural networks using SMC techniques. In their proposed approach which uses HE as one building block of SMC, a data owner shares the data with two servers and the two servers run the machine learning algorithm using two-party computation (2PC) technique. They focus on three different algorithms: linear regression, logistic regression and neural network. Their protocol is divided into two phases: online and offline. In the offline phase, they use oblivious transfer (OT) for generating multiplication triplets and in the online phase, they securely compute the activation function in logistic regression and neural network training. We only look at their neural network algorithm which is closely related to our work.

Table 5. Comparing the number of communications in our approach and SMC-based approach for different number of hidden layers and different batch sizes for the input (HL represents for Hidden Layers).

	1 HL		2 HLs		3 HLs		4 HLs		5 HLs	
Batch Size	**CryptoDL**	**[34]**	**CryptoDL**	**[34]**	**CryptoDL**	**[34]**	**CryptoDL**	**[34]**	**CryptoDL**	**[34]**
282	77	6042	192	12216	452	29940	600	77148	640	129144
576	77	7507	192	15151	452	37095	600	95543	640	159919
1420	77	10452	192	21036	452	51420	600	132348	640	221484
3668	77	12778	192	25714	452	62850	600	161762	640	270706
6144	77	13924	192	28036	452	68548	600	176452	640	295300

The main advantage of our approach over SMC-based solutions is the communication overhead which consists of the number of communications between the client and the server and the amount of data transferred in the communications. In SMC protocols, we need a number of communications between the client and the server for each operation whereas in our approach, the server does not need to communicate with the client unless the amount of noise reaches a threshold. If the same neural network is implemented using SMC protocols, the number of communications is much higher compared with the HE based protocols.

The SMC-based approach of [34] implements two solutions based on linearly homomorphic encryption (LHE) and oblivious transfer (OT) and results are provided for both LAN and WAN settings. In their LHE-based solution, the running time for LAN and WAN is almost the same. However, the accuracy of the model is very low. To improve the accuracy, higher degree polynomials should be used which will introduce significant computation and communication overhead. Alternatively, they present an OT-based solution which achieves better accuracy. However, there is a huge gap between communication overhead of LAN and WAN settings (e.g., 0.86s for LAN and 43.2s for WAN, see Table 2 from [34]). As mentioned by the authors, the reason for this gap is the communication overhead since the number of communications increases drastically as the neural networks become more complex. Since we assume a client-server model, the WAN setting is more important and we use the number of communications as one of the metrics for comparison.

To provide a comparison for the number of communications, we look at the approach proposed in [34]. The number of communications is $O(t \cdot \sum_{i=1}^{m}(|B| \cdot d_{i-1} + d_{i-1} \cdot d_i))$ where t, m, $|B|$ and d_i are the number of iterations, number of layers, size of the batch input and the number of neurons in layer i relatively (refer to section III-D of [34]). It is not as straightforward in our approach to calculate the number of communications, because the number of communications depends on the level of the noise in the ciphertext. However, it is possible to estimate the number of communications based on the architecture of the neural network.

In our approach, when the level of the noise gets close to the threshold, the server sends the ciphertext to the client and receives the refreshed ciphertext. Therefore, one communication needed between client and server for reducing noise in each ciphertext. This ciphertext could be a weight or a variable in the process of the algorithm. Suppose the network has l layers and d_i is the number of neurons in layer i. D is the degree of the polynomial used as the activation function and based on the input parameters for the HE scheme, the number of allowed multiplication is $\mathcal{L}$. Each iteration (one feed-forward and one back-propagation) includes $(l-1) + (2D-1)(l-2)$ multiplications (call it $\mathcal{M}$). Therefore, $\frac{\mathcal{M}}{\mathcal{L}}$ is the number of communications for each variable to reduce the noise in one iteration. On the other hand, the number of weights in a network is $\sum_{i=1}^{l-1} d_i \cdot d_{i+1}$ (call it $\mathcal{W}$). We can estimate the total number of communications for t iterations with this equation: $t \cdot \frac{\mathcal{M}}{\mathcal{L}} \cdot \mathcal{W}$. In our solution, the number of communications is independent of the batch size whereas in [34], the size of the batch contributes to the number of communications. Table 5 shows the number of communications for one iteration of our approach and SMC-based approach for different number of hidden layers and different batch sizes. As shown, there is a huge difference in the number of communications in our solution and the proposed solution in [34].

In addition to the the number of communications, we compare the amount of data transferred between the client and the server. When we have simpler neural networks, SMC-based approach has smaller communication overhead but our approach is much better when the neural network becomes more complex. For example, for a neural network with 2 hidden layers and batch size of 512, the amount of data transferred in [34] is 12MB whereas for our approach, it is 33MB. However,

Table 6. Training Neural Network over the Encrypted MNIST Data (batch input size of 192 and $L = 20$).

#Hidden layer(s)	Feedforward(s)	Backpropagation(s)	# of Communications	Noise Reduction(s)	Total Time(s)
2	2884.31	6301.26	6740	8939.40	10476.29

Table 7. Training different Neural Network architectures over the Encrypted Data ($L = \{5, 10, 20\}$).

Topology	L	Feedforward(s)	Backpropagation(s)	Noise Reduction(s)	# of Communications	Total Time(s)
[6 1 1 10]	5	9.58	24.62	34.65	164	37
[12 2 2 10]	5	17.39	46	63.31	314	67.25
[24 4 4 10]	5	44.28	113.23	157.46	788	165.36
[48 8 8 10]	5	134.22	331.64	465.74	2432	488.53
[93 16 16 10]	5	654.98	1357.19	2012.08	8504	2086.27
[6 1 1 10]	10	23.45	40.27	63.64	106	71.49
[12 2 2 10]	10	42.53	71.29	113.68	190	125.8
[24 4 4 10]	10	105.77	160	265.5	448	289.52
[48 8 8 10]	10	315.25	464.39	779.48	1324	842.675
[93 16 16 10]	10	1157.17	1631.88	2788.78	4516	2999.14
[6 1 1 10]	20	28.46	60.64	86.89	34	118.93
[12 2 2 10]	20	58.83	105.44	158.8	62	204.07
[24 4 4 10]	20	127.19	214.33	322.55	130	419.02
[48 8 8 10]	20	608.87	882.1	1339.14	314	1860.77
[93 16 16 10]	20	1527.85	1927.16	3020.29	874	4469.96

for a neural network with 5 hidden layers and batch size of 512, the amount of data in [34] is 159MB while in our solution, it is 155MB. If we use larger batch sizes (i.e., 3668), our approach outperforms [34] in neural networks with 4 hidden layers and higher (151 MB for our approach v.s. 161 MB for [34]). These results show that [34] has much higher communication overhead compared to our approach and the main reason is that the number of communications in [34] grows significantly as the neural networks become more complex.

We implemented a neural network similar to the one used in [34] and performed experiment using the encrypted MNIST dataset. The results shown in Table 6 for one iteration of training and indicate that the communication has a huge overhead on the running time. Besides the number of communications, the size of data transferred in each communication impacts the running time. The size of data in each communication is the same as the size of a ciphertext. Although we can decrease the size of the ciphertext by decreasing the value of L, decreasing the value of L results in higher number of communications. Table 7 shows the result for $L = \{5, 10, 20\}$ and different topologies of the neural network for one iteration of training. As it can be seen, although we have higher number of communications for $L = 5$, the whole run time is less than $L = 10$. The reason is that the size of the ciphertext is larger when $L = 10$. To improve the run time, we can run the code in parallel or using Graphics Processing Unit (GPU) which is part of our future work. Note that we did not use any parallelization techniques and the performance can be further improved if we we use parallel implementation.

In terms of the accuracy, the model trained by our proposed approach can reach 95.15% using polynomial as the activation function over plain-text MNIST dataset while the model trained by the protocol in [34] can only reach 93.4% accuracy. The running time for privacy preserving prediction in our model is better compared with the solution in [34].

Another advantage of our approach is the polynomial approximation method we used for approximating the Sigmoid function. In our approach, the degree of polynomial is 2 or 3 in all cases which allows the implementation to be efficient. However, in [34], the performance of the model drops considerably when low degree polynomials are used and it requires a high degree polynomial (i.e. 10) to achieve a reasonable accuracy (refer to Table 1 in [34]). As discussed before, computing the high degree polynomial over encrypted data is very inefficient.

The approach in [34] needs two different servers in addition to the client and assumes that the servers do not collude with each other. This is a weaker security assumption compared to ours and if the servers collude with each other, the privacy of the client's data is not preserved anymore. In addition, to improve the performance of their model, they proposed a client aided approach where the client operates part of the computation. In this scenario, if a subset of clients collude with one of the servers, their approach leaks information about the data from honest clients (see [34], Section V).

Darvish et al. present DeepSecure that enables distributed clients (data owners) and cloud servers to jointly evaluate a deep learning network on their private assets [41]. It uses Yao's Garbled Circuit (GC) protocol to securely perform deep learning. Liu et al. propose a privacy-preserving classification protocol based on the Oblivious Transfer (OT) and HE schemes [25]. Riazi et al. propose a solution based on Garbled Circuits (GC) and Aditive Secret Sharing (SS) [39]. They all perform experiments on the MNIST dataset and report the results.

The accuracy of the models trained with the MNIST dataset in these approaches is 98.95%, 99.31%, and 99% respectively whereas in our approach, the accuracy is 99.25%. The throughput of their approaches are 2759 instances per hour, 2812 instances per hour, and 1333 instances per hour respectively (these numbers calculated by extrapolation), whereas in our solution the throughput is 163840 instances per hour. [25] implements a deeper CNN and achieves a slightly better accuracy (99.0%) but the throughput drops to only 386 instances per hour.

For classifying a batch input with size 8192, the running times for [41] and [25] are 10649 and 10485.76 seconds respectively, while in our solution, the running time is only 320 seconds. The amount of data transferred in one round of protocol in these two approaches are 722GB and 390GB respectively, whereas in our approach it is only 336.7MB. These papers provide results for one instance and do not discuss the impact of higher batch input size on the running time; we assume that the running time increases linearly and calculate these numbers by extrapolation.

For classifying one instance, the running time and the amount of data transferred in [39] are 2.70 seconds and 12.9MB. However, for classifying a batch input with 100 instances, the running time and the amount of data transferred increase to 153.47 seconds (392 seconds in our solution) and 1289.4MB (320MB in our solution). For a batch input with size 8192, the amount of transferred data and the running time in our solution remains the same, however in [39], they increase drastically and make the solution inefficient for large input size. As can been seen, our approach significantly outperforms all three approached in all aspects.

In addition to having a much better performance, our approach has several advantages over SMC-based approaches. Unlike SMC-based approaches, the structure of the client does not need to be changed in our approach when the architecture of the neural network is changed. The only operations in the client side are encryption and decryption and the server can perform different machine learning algorithms with the same client. Another advantage of our approach is that it preserves the privacy of the model compared with the SMC-based solutions. In SMC-based solutions, the client participates in the computations and information about the model could possibly leak to the client. For example, the client can learn information such as the number of layers in the neural network, the structure of each layer and the activation functions. Although there are some solutions like adding more dummy layers to the model to mitigate this issue, these will add to the already large computation cost. On the other hand, in our approach, the only operations in the client side are encryption and decryption (and possibly division in the case of networks with high number of layers). Hence, the client cannot learn any information about the structure of the neural network.

4.5 Discussion and Limitations

We do not use any parallel programming techniques (such as openmp library [35]) or Graphics Processing Unit (GPU) in our implementation. In each layer of the neural network, the computations inside neurons are independent of each other, and the computations for the neurons in each layer could be performed in parallel. This will significantly reduce the running time, especially, when the number of neurons in each layer is high. The performance can also be further improved using GPU-based implementation. These are left to future work.

A limitation of our proposed work is that our approximation only considers continuous functions. Although our approximation covers a number of widely used activation functions in deep neural networks, there are non-continuous functions used in deep learning algorithms that are not covered. We will extend our approximation work to include non-continuous functions.

There are several attacks like membership inference [43], model inversion [13] and model extraction [47] against machine learning algorithms. For example, in membership inference attack, the client registers for a machine learning service and uses the model as a black-box. A malicious client aims to understand that an instance was part of the training set or not. Both SMC-based and HE-based approaches are susceptible to these attacks, because they target underlying machine learning algorithms regardless of whether the data is encrypted or not. There are countermeasure proposed in the literature that could be used to mitigate these attacks [13, 43, 47]. However, defending against attacks that target the underlying machine learning algorithms is outside the scope of this work and the main focus of this work is to provide privacy-preserving training and classification in neural networks.

5 Related Work

Graepel et al. use a somewhat homomorphic encryption scheme to train two machine learning classifiers: Linear Mean and Fisher's Linear Discriminate (FLD) [19]. They propose division-free algorithms to adopt to limitations of homomorphic encryption algorithms. They focus on simple classifiers such as the linear means classifier, and do not consider more complex algorithms. Also, their approach considers a weak security setting in which the client can learn the model. Bost et al. use a combination of three homomorphic systems (Quadratic Residuosity, Piallier, and BGV schemes), and garbled circuits to provide privacy-preserving classification for three different machine learning algorithms, namely Hyperplane Decision, Naive Bayes, and Decision trees [5]. Their approach only considers the classification phase, is based on SMC, and is only efficient for small data sets. Our proposed protocol is based only on homomorphic encryption and we consider both training and classification phases. Mohassel and Zhang propose a protocol for privacy-preserving training and classification based on SMC techniques [34]. A detailed comparison with this work is presented in section 4.4.

Xie et al. discuss theoretical aspects of using polynomial approximation for implementing neural network in encrypted domain [49]. Building on this work, Dowlin et al. implement a neural network classifier on encrypted data [10]. They use a leveled homomorphic encryption scheme which supports SIMD. They replace *max pooling* with scaled mean-pooling for solving the issue of having no division operation available for encrypted values and use function $f(z) := z^2$ as an activation function. They build the model using plain data and then use this model for classifying encrypted data. Our approach, on the other hand, provides a solution for both training and classifying encrypted instances. A detailed comparison with this work is presented in section 4.3. Yuan et al. in [52] propose a protocol for privacy-preserving training and classification based on SMC techniques and HE schemes. Three types of components participate in the protocol: clients, system and cloud. The clients encrypt the data using the public key of the system and send it to the cloud. As part of their protocol, they implement secure sharing of scalar product and sum using secret sharing.

Aslett et al. propose methods for implementing statistical machine learning over encrypted data and implement extremely random forests and Naive Bayes classifiers over 20 datasets [2]. The majority of operations are addition and multiplication and they show that performing algorithms over encrypted data without any multi-party computation or communication is practical. They also analyze current homomorphic encryption tools for use in statistical machine learning [3]. Several methods have been proposed for statistical analysis over encrypted data, specifically for secure computation of a χ^2-test on genome data [27, 32, 54].

Shortell et al. use the Taylor expansion of $ln(x)$ to estimate the natural logarithm function by a polynomial of degree 5 [44]. Although Taylor expansions are more convenient and easier to compute, the accuracy of estimation is not always consistent. We used Taylor expansions to estimate the Sigmoid function and compared the result with other approximation techniques. While in some cases they produce accuracies as high as other approximations, often times the accuracy drops significantly when using Taylor expansions. Livni et al. analyzed the performance of polynomial as an activation function in neural networks [31]. However, their solution cannot be used for our purpose because they approximate the Sigmoid function on the interval [-1,1] while the message space in homomorphic encryption schemes is integers. Our methods is able to generate polynomial approximation for an arbitrary interval by using Chebyshev polynomials to find the best approximation based on the $\|\cdot\|_{2,\mu}$.

There is another line of research which is not directly related to our problem, but is worth mentioning here. Togan et al. propose an algorithm for finding a maximum over encrypted data [46]. In their implementation, the client sends the encrypted data to the server, which computes the maximum without learning the value of the maximum or its index. Liu et al. propose a protocol for computing scalar product, Secure Scalar Product in MapReduce (S^{2}PM) [30]. They use the BGN homomorphic encryption scheme and the HELib library in implementation.

There has also been some work on adding division to homomorphic encryption schemes. Veugen considers a semi-honest model where client A has some encrypted number $[x]$, and the server B has the decryption key K [48]. Party A would like to divide the integer x by some integer d. Chen et al. analyze the performance of integer arithmetics over encrypted data and propose an algorithm for the division operation [7]. They use the HELib for implementation and their results show that division for large numbers cannot be supported without using bootstrapping. The division operation has more limitations than advantages for our solution, because we should use a FHE which is not practical. We assume that homomorphic encryption schemes do not support division and try to find a solution based only on addition and multiplication operations.

There are also approaches based on differential privacy to privacy-preserving deep learning. Shokri and Shmatikov in [42] propose a privacy-preserving approach for learning from distributed data that allows multiple parties to jointly learn a neural network model without sharing their input datasets. Their approach is based on differential privacy where each party builds the model using its own data, and only shares a small part of the model's parameters with other parties. Abadi et al. in [1] propose a framework for differentially private training of neural networks. It includes a differentially private stochastic gradient descent (SGD) algorithm, the moments accountant, and hyperparameter tuning. They develop new algorithmic techniques, provide an analysis of privacy costs within the framework of differential privacy, and implementation strategies based on the TensorFlow software library for machine learning. Differential privacy based approaches should make a trade-off between privacy and utility which affects accuracy of the trained model. Our threat model is different than differential privacy. In our approach, the server does not learn anything about the model whereas in differential privacy, the server will learn the model. Since the model is encrypted, reverse engineering cannot be used either.

6 Conclusion and Future Work

In this paper, we developed new solutions for running neural network algorithms over encrypted data. In order to implement neural networks within limitations of homomorphic encryption schemes, we introduced new techniques to approximate activation functions with low degree polynomials. We developed a theoretical foundation for this purpose and provided an approach to generate those approximations based on Chebyshev polynomials. We then used these approximation to train neural networks. We also performed experiments to evaluate computation cost of operating these polynomials over encrypted data and implemented the neural networks with polynomial approximation as activation function over encrypted data. Our results show that polynomials, if chosen properly, are suitable replacements for activation functions to adopt neural networks within homomorphic encryption schemes limitations. Our proposed approach provides accurate and privacy-preserving training and classification and outperforms state-of-the-art approaches based on homomorphic encryption and secure multi-party computation.

For future work, we plan to study approximation of non-continuous functions used in deep neural network algorithms. We also plan to investigate possibility of using polynomial approximation and rational functions for more complex computations such as statistical analysis, solving partial differential equations, etc. over encrypted data.

7 Acknowledgment

Part of this work was done while the first author was visiting the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), Rutgers University. The support was provided by the National Science Foundation under grant number CNS-1523467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov. Deep Learning with Differential Privacy. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), Vienna, Austria. pp 308–318.

[2] L. J. M. Aslett, P. M. Esperança, C. C. Holmes. Encrypted statistical machine learning: new privacy preserving methods. CoRR abs/1508.06845 (2015).

[3] L. J. M. Aslett, P. M. Esperanca, C. Holmes. A review of homomorphic encryption and software tools for encrypted statistical machine learning. Tech. rep., University of Oxford, 2015.

[4] K. Atkinson, W. Han. Theoretical Numerical Analysis: A Functional Analysis Framework. Texts in Applied Mathematics. Springer New York, 2009.

[5] R. Bost, R. A. Popa, S. TU, S. GOLDWASSER. Machine learning classification over encrypted data. In 22nd Annual Network and Distributed System Security Symposium, NDSS, San Diego, California, USA (2015).

[6] D. Boneh, Wu. David, H Jacob. Using homomorphic encryption for large scale statistical analysis. 2012.

[7] Y. Chen, G. Gong. Integer arithmetic over ciphertext and homomorphic data aggregation. In Communications and Network Security (CNS), IEEE Conference on (Sept 2015), pp. 628–632.

[8] T. S. Developers. SageMath, the Sage Mathematics Software System (Version 7.1), 2016. http://www.sagemath.org.

[9] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauther, M. Naehrig, J. Wernsing. Manual for using homomorphic encryption for bioinformatics. Tech. Rep. MSR-TR-2015-87, November 2015.

[10] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. L. M. Naehrig, J. Wersning. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. Proceedings of The 33rd International Conference on Machine Learning, in PMLR 48:201-210.

[11] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In Proceedings of CRYPTO 84 on Advances in Cryptology (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 10–18.

[12] Ersatz Labs. http://www.ersatzlabs.com/, Accessed: 2017-02-20.

[13] M. Fredrikson, S. Jha, T. Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. ACM Conference on Computer and Communications Security 2015, Denver, CO, USA, October 12-6, 2015, 1322-1333.

[14] C. Gentry. A Fully Homomorphic Encryption Scheme. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[15] M. Ghasemi. https://github.com/mghasemi/pyProximation.

[16] Google Prediction API, https://cloud.google.com/prediction/, Accessed: 2017-02-20.

[17] S. Goldwasser, S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (New York, NY, USA, 1982), STOC '82, ACM, pp. 365–377.

[18] GraphLab, http://www.select.cs.cmu.edu/code/graphlab/, Accessed: 2017-02-20.

[19] T. Geaepel, K. Lauter, M. Naehrig. ML confidential: Machine learning on encrypted data. In Proceedings of the 15th International Conference on Information Security and Cryptology (Berlin, Heidelberg, 2013), ICISC'12, Springer-Verlag, pp. 1–21.

[20] S. Halvei, V. Shoup. Algorithms in HElib. In Advances in Cryptology - CRYPTO - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, Proceedings, Part I (2014), pp. 554– 571.

[21] E. Hesamifard, H. Takabi, M. Ghasemi. CryptoDL: Towards Deep Learning over Encrypted Data. Annual Computer Security Applications Conference (ACSAC 2016), Los Angeles, California, USA.

[22] E. Hesamifard, H. Takabi, M. Ghasemi, C. Jones. Privacy-preserving Machine Learning in Cloud. In Proceedings of the 2017 on Cloud Computing Security Workshop (CCSW '17), ACM, New York, NY, USA, pp. 39-43.

[23] E. Hesamifard, H. Takabi, M. Ghasemi. CryptoDL: Deep Neural Networks over Encrypted Data. arXiv preprint arXiv:1711.05189, 2017.

[24] N. Islam, W. Puech, K. HAYAT, R. BROUZET. Application of Homomorphism to Secure Image Sharing. Optics Communications 284, 19 (Sept. 2011), 4412–4429.

[25] L. Jian, J. Mika, L. Yao, A. N. Oblivious Neural Network Predictions via MiniONN Transformations. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, 2017. Dallas, Texas, USA.

[26] B. Karlik, A. V. Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. International Journal of Artificial Intelligence and Expert Systems 1, 4 (2011), 111–122.

[27] M. Kim, K. E. Lauther. Private genome analysis through homomorphic encryption. IACR Cryptology ePrint Archive 2015 (2015), 965.

[28] A. Krizhevsky, V. Nair, G. Hinton. CIFAR-10 (Canadian Institute for Advanced Research), http://www.cs.toronto.edu/ kriz/cifar.html

[29] M. LICHMAN, UCI machine learning repository, 2013.

[30] F. Liu, W. K. Ng, W. Zhang. Secure scalar product for big-data in MapReduce. In Big Data Computing Service and Applications (BigDataService), IEEE First International Conference on (March 2015), pp. 120–129.

[31] R. Livni, S. Shalev-Shwartz, O. Shamir. On the computational efficiency of training neural networks. CoRR abs/1410.1141 (2014).

[32] W. J. LU, Y. YAMADA, J. SAKUMA. Privacy-preserving genome-wide association studies on cloud environment using fully homomorphic encryption. BMC Medical Informatics and Decision Making 15, 5 (2015), 1–8.

[33] Microsoft Azure Machine Learning, https://azure.microsoft.com/en-us/services/machine-learning/, Accessed: 2017-02-20.

[34] P. Mohassel, Y, Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. IACR Cryptology ePrint Archive, 2017.

[35] OpenMP, http://www.openmp.org/, Accessed: 2017-02-22.

[36] P. PAILLIER. Public-key cryptosystems based on composite degree residuosity classes. In Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques (Berlin, Heidelberg, 1999), EUROCRYPT'99, Springer-Verlag, pp. 223–238.

[37] P. PILOTTE. Neural Network Toolbox, 2016.

[38] S. RANE, W. SUN, A. VETRO. Secure distortion computation among untrusting parties using homomorphic encryption. In ICIP (2009), IEEE, pp. 1485–1488.

[39] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, F. Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications.

Cryptology ePrint Archive, Report 2017/1164, 2017.

[40] R. Rivest, L. Adleman, M. Dertouzos. On data banks and privacy homomorphisms. In Foundations on Secure Computation, Academia Press (1978), pp. 169–179.

[41] Bita. D. Rouhani, M. Sadegh Riazi, F. Koushanfar. DeepSecure: Scalable Provably-Secure Deep Learning. CoRR abs/1705.08963 (2017). http://arxiv.org/abs/1705.08963

[42] R. Shokri, V. Shmatikov. Privacy-Preserving Deep Learning. Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). Denver, Colorado, USA. pp 1310–1321.

[43] R. Shokri, M. Stronati, V. Shmatikov. Membership Inference Attacks against Machine Learning Models. arXiv preprint. arXiv: abs/1610.05820 (2016).

[44] T. Shortell, A. Shokoufandeh.Secure signal processing using fully homomorphic encryption. In Advanced Concepts for Intelligent Vision Systems - 16th International Conference, ACIVS, Catania, Italy, October 26-29, Proceedings (2015), pp. 93–104.

[45] H. Takabi, E. Hesamifard, M. Ghasemi. Privacy Preserving Multi-party Machine Learning with Homomorphic Encryption. Private Multi-Party Machine Learning, NIPS 2016 Workshop, Barcelona, Spain.

[46] M. Togan, C. Plesca. Comparison-based computations over fully homomorphic encrypted data. In Communications (COMM), 10th International Conference on (May 2014), pp. 1–6.

[47] F. Tramèr, F. Zhang, A. Juels, M.K. Reiter, T. Ristenpart. Stealing Machine Learning Models via Prediction APIs. 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. 601–618.

[48] T. Veugen. Encrypted integer division and secure comparison. Int. J. Appl. Cryptol. 3, 2 (June 2014), 166–180.

[49] P. Xie, M. Bilenko, T. Finley, R. Gilda-Bachrach, K. E. Lauter, M. Naehrig. Crypto-nets: Neural networks over encrypted data. CoRR abs/1412.6181 (2014).

[50] Y. Xu, Orthogonal polynomials of several variables. Encyclopedia of Mathematics and its Applications 81 (2001).

[51] L. Yann, C. Corinna, C. J. C. Burges. MNIST handwritten digit database. 2010.

[52] J. Yuan, S. Yu. Privacy Preserving Back-Propagation Learning Made Practical with Cloud Computing. Security and Privacy in Communication Networks: 8th International ICST Conference, SecureComm 2012, Padua, Italy, September 3-5, 2012. Revised Selected Papers. 2013, Springer Berlin Heidelberg, Berlin, Heidelberg, 292–309.

[53] J. Zhan. Using Homomorphic Encryption For Privacy-Preserving Collaborative Decision Tree Classification. 2007 IEEE Symposium on Computational Intelligence and Data Mining, 637-645.

[54] Y. Zhang, W. Dai, X. Jiang, H. Xiong, S. Wang. Foresee: Fully outsourced secure genome study based on homomorphic encryption. BMC Medical Informatics and Decision Making 15, 5 (2015), 1–11. Yuchen Zhang, Wenrui Dai contributed equally to this work.

Appendix

Homomorphic Encryption: Homomorphic encryption (HE) schemes preserve the structure of the message space such that we can perform operations such as addition and multiplication over the ciphertext space. Like other types of encryption schemes, an HE scheme has three main functions, *Gen*, *Enc*, and *Dec*, for key generation, encryption, and decryption, respectively. However, an HE scheme also has an evaluation function, *Eval*. Suppose we have a set of plaintext messages $\{m_i\}$ and relative ciphertexts $\{c_i\}$ for $i = \{1, \cdots, n\}$. Now, consider a circuit C. The evaluation function processes the public key pk, a set of ciphertexts $\{c_i\}$ and a circuit C such that (see [14])

$$Dec(sk, Eval(pk, C, c_1, \cdots, c_n)) = C(m_1, \cdots, m_n) \quad (4)$$

Homomorphic encryption (HE) was first introduced in 1978 by Rivest et al. [40]. Other researchers followed to introduce several other HE schemes [11, 17, 36]. The idea behind the encryption function *Enc* is to add a small value, called *noise*, to m for encrypting. Therefore, each ciphertext has a small amount of noise. When we add two ciphertexts c_1 and c_2, the result is also a ciphertext, but with noise that has grown. The *Dec* function works correctly if this amount is less than a threshold. This threshold leads to a bound on the number of computations that can be performed over encrypted data. If an entity wants to decrease the noise, it should decrypt the ciphertext and encrypt it again, and for decryption, it needs the secret key sk. For years, the community was trying to find out if there is a way to decrease the noise without having the secret key.

This question was answered in 2009 when first Fully Homomorphic Encryption (FHE) scheme was designed by Gentry [14]. A Fully Homomorphic Encryption (FHE) scheme is a homomorphic encryption scheme that supports circuits with arbitrary depth. In his dissertation, Gentry introduced a technique for handling an arbitrary depth of computations, called *bootstrapping*. In the bootstrapping technique, the amount of noise is decreased without needing to access sk. However, it has a huge computation cost and is a very slow process. This limitation makes FHE impractical for actual use.

Recent advances in homomorphic encryption have led to a faster HE scheme: Leveled Homomorphic Encryption (LHE), see [14]. LHE schemes do not support the bootstrapping step, so they only allow circuits with depths less than a specific threshold. If the number of operations is known before starting the computations, we can use LHE instead of FHE. The performance of LHE schemes is further improved using Single-Instruction-Multiple-Data (SIMD) techniques [20].

Despite the advantages of using HE schemes, they have some limitations. The first one is message space. Almost all HE schemes work with integers. Therefore, before encrypting data items, we need to convert them to integers and if this conversion is not done properly, it could lead to accuracy loss. The second limitation is ciphertext size. The size of the message increases considerably after encryption. Another important limitation is related to noise. After each operation, the amount of noise in the ciphertext increases. Multiplication increases noise much more than addition. For HE schemes to work properly, the amount of noise should remain less than a predefined threshold. The last and most important limitation is lack of support for the division operation. In summary, only a limited number of additions and multiplications could efficiently be performed over the encrypted data and complex functions such as the Sigmoid function used in neural networks are not compatible with the current HE schemes.

Neural Networks: At a high level of abstraction, a neural network is a combination of neurons arranged in ordered layers. Each neuron gets an input, operates a function on it and outputs the result of the function. The structure of this function depends on the layer to which the neuron belongs. Besides the first layer (input layer) and the last layer (output layer), there is at least one middle layer, called hidden layer. In fully feed-forward neural networks, each neuron has a weighted connection to all neurons in the next layer. Neurons in different layers are of different types. For example, neurons in the input layer only get one input and output which is the same value. Neurons in hidden layers are more complex; they get inputs, compute the weighted summation of inputs, operate a function on the summation and then output the value of the function. These functions could be the Sigmoid ($\sigma = \frac{1}{1+e^{-x}}$), the ReLU ($ReLU(x) = max(0, x)$), Tanh ($2\sigma(2x)-1$), max or mean functions and are called activation functions (or transfer functions). There are two methods for training the neural network: online training and batch training. In online training, the weights are updated after feeding one instance to the network whereas in batch training, a batch of instances is fed to the network to update the weights in each step.

Table 8. Polynomial approximation for the Sigmoid function on the interval $[-10^3, 10^3]$. $p_1(x)$ generated by method 1 (Equation 2) and $p_2(x)$ generated by method 2 (Equation 3).

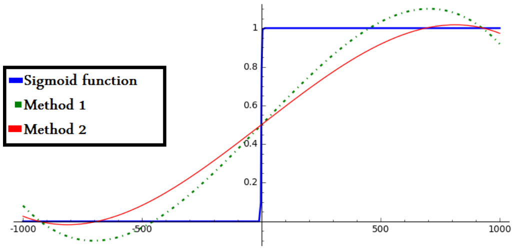
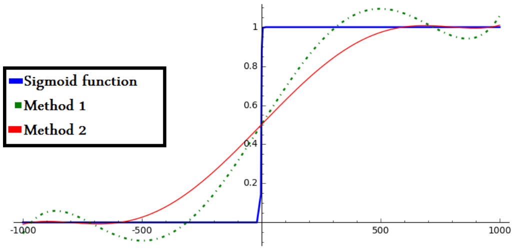
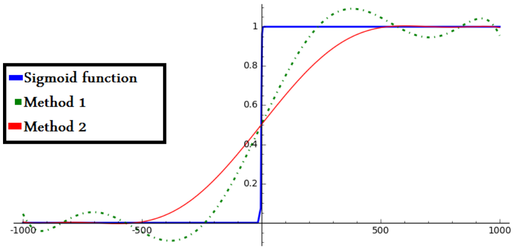
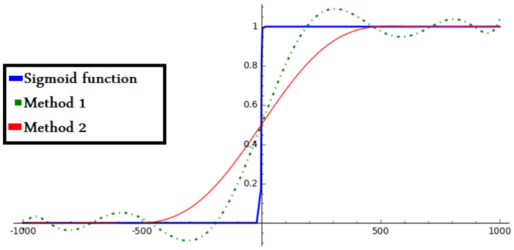
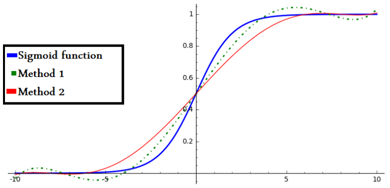
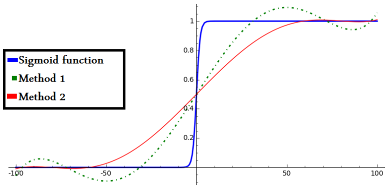
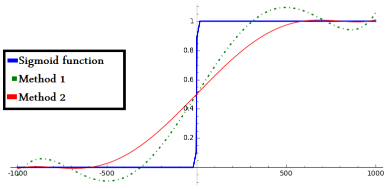
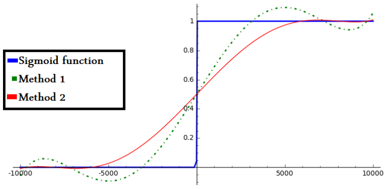
Degree	Polynomial Approximations	Sigmoid function, $p_1(x)$ and $p_2(x)$
3	$p_1(x) = -(8.606e - 10) * x^3 + (1.330e - 17) * x^2$ $+ 0.001 * x + 0.499$ $p_2(x) = -(4.805e - 10) * x^3 - (7.088e - 16) * x^2$ $+ 0.0009 * x + 0.500$ $\|f - p_1\|_2 = 0.276, \|f - p_2\|_2 = 0.270$	
5	$p_1(x) = (2.0694e - 15) * x^5 - (5.319e - 23) * x^4$ $\cdots + 0.001 * x + 0.499$ $p_2(x) = (6.653e - 16) * x^5 + (9.377e - 21) * x^4$ $\cdots + 0.001 * x + 0.500$ $\|f - p_1\|_2 = 0.226704, \|f - p_2\|_2 = 0.094159$	
7	$p_1(x) = -(5.917e - 21) * x^7 + (2.126e - 28) * x^6$ $\cdots + 0.002 * x + 0.499$ $p_2(x) = -(1.098e - 21) * x^7 - (1.027e - 25) * x^6$ $\cdots + 0.001 * x + 0.500$ $\|f - p_1\|_2 = 0.196167, \|f - p_2\|_2 = 0.037212$	
9	$p_1(x) = (1.841e - 26) * x^9 - (8.478e - 34) * x^8$ $\cdots + 0.003 * x + 0.499$ $p_2(x) = (2.002e - 27) * x^9 + (9.866e - 31) * x^8$ $\cdots + 0.001 * x + 0.500$ $\|f - p_1\|_2 = 0.175104, \|f - p_2\|_2 = 0.016006$	

Table 9. Polynomial approximation for the Sigmoid function on different intervals. $p_1(x)$ generated by method 1 (Equation 2) and $p_2(x)$ generated by method 2 (Equation 3).

Interval	Polynomial Approximations	Sigmoid function, $p_1(x)$ and $p_2(x)$
[-10,10]	$p_1(x) = 0.00001 * x^5 - (4.897e - 14) * x^4$ $\cdots + 0.160 * x + 0.499$ $p_2(x) = (6.371e - 6) * x^5 + (9.377e - 13) * x^4$ $\cdots + 0.124 * x + 0.500$ $\|f - p_1\|_2 = 0.066, \|f - p_2\|_2 = 0.008$	
$[-10^2, 10^2]$	$p_1(x) = (2.060e - 10) * x^5 - (5.329e - 19) * x^4$ $\cdots + 0.019 * x + 0.499$ $p_2(x) = (6.653e - 11) * x^5 + (9.376e - 17) * x^4$ $\cdots + 0.012 * x + 0.500$ $\|f - p_1\|_2 = 0.207, \|f - p_2\|_2 = 0.030$	
$[-10^3, 10^3]$	$p_1(x) = (2.069e - 15) * x^5 - (5.319e - 23) * x^4$ $\cdots + 0.001 * x + 0.499$ $p_2(x) = (6.653e - 16) * x^5 + (9.377e - 21) * x^4$ $\cdots + 0.001 * x + 0.500$ $\|f - p_1\|_2 = 0.227, \|f - p_2\|_2 = 0.094$	
$[-10^4, 10^4]$	$p_1(x) = (2.069e - 20) * x^5 - (5.320e - 27) * x^4$ $\cdots + 0.0001 * x + 0.499$ $p_2(x) = (6.653e - 21) * x^5 + (9.376e - 25) * x^4$ $\cdots + 0.0001 * x + 0.500$ $\|f - p_1\|_2 = 0.229, \|f - p_2\|_2 = 0.300$	 Sigmoid function Method 1 Method 2 1 0.8 0.6 0.4 0.2 -10000 -5000 5000 10000