CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy

Nathan Dowlin*[1,2], Ran Gilad-Bachrach[1], Kim Laine[1], Kristin Lauter[1], Michael Naehrig[1] and John Wernsing[1]

[1]Microsoft Research
[2]Department of Mathematics, Princeton University

December 17, 2015

Abstract

Applying machine learning to a problem which involves medical, financial, or some other type of sensitive data, not only requires accurate predictions but also careful attention to maintaining data privacy and security. Legal and ethical requirements may prevent the use of cloud-based machine learning solutions for such tasks. In this work, we present a method to convert learned neural networks to *CryptoNets*, neural networks that can be applied to encrypted data. This allows a data owner to send their data in encrypted form to a cloud service that hosts the network. The encryption ensures that the data remains confidential since the cloud does not have access to the keys needed to decrypt it. Nevertheless, we will show that the cloud service is capable of applying the neural network to the encrypted data to make encrypted predictions, and return them, also in encrypted form. These encrypted predictions can be sent back to the owner of the secret key who can decrypt them. Therefore, the cloud service does not gain any information about the raw data nor about the prediction it made.

We demonstrate *CryptoNets* on the MNIST optical character recognition tasks. CryptoNets achieve 99% accuracy and can make more than 51000 predictions per hour on a single PC. Therefore, they allow high throughput, accurate, and private predictions.

1 Introduction

Consider a hospital that would like to use a cloud service to predict the probability of readmission of a patient within the next 30 days, in order to improve the quality of care and to reduce costs. Due to ethical and legal requirements regarding the confidentiality of patient information, the hospital might be prohibited from using such a service. In this work we present a way by which the hospital can get this valuable service without sacrificing patient privacy. In the protocol we propose, the hospital encrypts the private information and sends it in encrypted form to the prediction provider, referred to as *the cloud* in our discussion below. The cloud is able to compute the prediction over the encrypted data records and sends back the results that the hospital can decrypt and read. The encryption scheme uses a public key for encryption and a secret key (private key) for decryption. It is important to note that the cloud does not have access to the secret key, so it cannot decrypt the data, nor can it decrypt the prediction. The only information it obtains during the process is that it did perform a prediction on behalf of the hospital. Hence, the cloud can charge the hospital for its service but does not learn anything about the patient's medical files or the predicted outcomes. This procedure allows for private and secure predictions without requiring the establishment of trust between the data owner and the service provider. This may have applications in fields such as health, finance, business, and possibly others.

It is important to note that this work only enables the inference stage. We make the assumption that the cloud already has a model, in our case it would be a neural network, that was trained in some way, for example using a set of unencrypted data. Training models for these kinds of applications might be a challenge as well due to the same concerns regarding privacy and security. The problem of training such a model is sometimes referred to as *privacy preserving data-mining* (Agrawal & Srikant, 2000). One possible solution to training while preserving privacy lies in the concept of *differential privacy* (Dwork, 2011). Working with a statistical database, differential privacy allows to control the amount of information leaked from an individual record in a dataset. Therefore, when training, one can use this concept to ensure privacy for any entity whose

*The work was conducted while the author was visiting Microsoft Research.

information is contained in the dataset, as well as to create models that do not leak this information about the data they were trained on. However, the notion of differential privacy is not useful in the inference phase since at this stage, we are interested in examining a single record. Other options include working on encrypted data for training as well, in which case either simple classification techniques should be used (Graepel et al., 2013), or other assumptions on the data representation should be made (Aslett et al., 2015a).

The main ingredients of CryptoNets are *homomorphic encryption* and *neural networks*. Homomorphic encryption was originally proposed by Rivest et al. (1978) as a way to encrypt data such that certain operations can be performed on it without decrypting it first. In his seminal paper Gentry (2009) was the first to present a *fully* homomorphic encryption scheme. The term ”fully homomorphic” means that the scheme allows arbitrarily many operations to be performed on the encrypted data. Gentry's original scheme was highly inefficient, but since then the work of several researchers has produced significantly more practical schemes. In this work, in particular, we use the homomorphic encryption scheme of Bos et al. (2013). This scheme is a *leveled* homomorphic encryption scheme, which allows adding and multiplying encrypted messages but requires that one knows in advance the complexity of the arithmetic circuit that is to be applied to the data. In other words, this cryptosystem allows to compute polynomial functions of a fixed maximal degree on the encrypted data. High degree polynomial computation requires the use of large parameters in the scheme, which results in larger encrypted messages and slower computation times. Hence, a primary task in making practical use of this system is to present the desired computation as a low-degree polynomial. We refer the reader to Dowlin et al. (2015); Bos et al. (2013) for details on the encryption scheme, and only give a brief introduction to it in Section 3. We used the *Simple Encrypted Arithmetic Library (SEAL)* for homomorphic encryption[1].

To allow accurate predictions we propose using neural networks, which in recent years have shown great promise for a wide variety of learning tasks. Much of the revival in the interest in neural networks is due to the unprecedented accuracies achieved in tasks such as image classification (Krizhevsky et al., 2012) and speech recognition (Dahl et al., 2012). In Section 2 we present a brief background on neural networks, as well as the necessary adjustments for them to work with homomorphic encryption, thus creating CryptoNets.

One line of criticism against homomorphic encryption is its inefficiency, which is commonly thought to make it impractical for nearly all applications. However, combining together techniques from cryptography, machine learning and software engineering, we show that CryptoNets may be efficient and accurate enough for real world applications. We show that when CryptoNets are applied to the MNIST dataset, an accuracy of 99% can be achieved with a throughput of 51739 predictions per hour on a single PC, and a latency of 570 seconds. Note that a single prediction takes 570 seconds to complete, however, the same process can make 8192 predictions simultaneously with no added cost. Therefore, over an hour, our implementation can make 51739 predictions on average. Hence, CryptoNets are accurate, secure, private, and have a high throughput - an unexpected combination in the realm of homomorphic encryption.

[1]Freely available at `http://sealcrypto.codeplex.com`

2 Neural Networks

The goal of this work is to demonstrate the application of neural networks over encrypted data. We use the term neural networks to refer to artificial feed-forward neural networks. These networks can be thought of as leveled circuits. Traditionally, these levels are called layers and are visualized as being stacked so that the bottom-most layer is the input layer. Each node of the input layer is fitted with the value of one of the features of the instance at hand. Each of the nodes in the following layers computes a function over the values of the layer beneath it. The values computed at the top-most layer are the outputs of the neural network.

Several common functions can be computed at the nodes. We have listed some of them here:

1. *Weighted-Sum* (convolution layer): Multiply the vector of values at the layer beneath it by a vector of weights and sum the results. The weights are fixed during the inference processes. This function is essentially a dot product of the weight vector and the vector of values of the feeding layer.

2. *Max Pooling*: Compute the maximal value of some of the components of the feeding layer.

3. *Mean Pooling*: Compute the average value of some of the components of the feeding layer.

4. *Sigmoid*: Take the value of one of the nodes in the feeding layer and evaluate the function $z \mapsto {}^{1}\!/_{(1+\exp(-z))}$.

5. *Rectified Linear*: Take the value of one of the nodes in the feeding layer and compute the function $z \mapsto \max(0, z)$.

Since homomorphic encryption supports only additions and multiplications, only polynomial functions can be computed in a straight forward way. Moreover, due to the increased complexity in computing circuits with nested multiplications, it is desired to restrict the computation to low-degree polynomials. The weighted-sum function can be directly implemented since it uses only additions and multiplications. Moreover, the multiplications here are between precomputed weights and the values of the feeding layer. Since the weights are not encrypted, it is possible to use the more efficient plain multiplication operation, as is described in Section 3.2.1. Some networks also add a bias term to the result of the weighted sum. To add this bias term a plain addition can be used since, again, the value of this bias term is known to the cloud.

One thing to note is that the encryption scheme does not support floating-point numbers. Instead, we use fixed precision real numbers by converting them to integers by proper scaling, although there are also other ways to do this (Dowlin et al., 2015). Furthermore, the encryption scheme applies all of its operations modulo some number t, which is why it is important to pay attention to the growth in the size of the numbers appearing throughout the computation, and to make sure that reduction modulo t does not occur. Otherwise the results of the computation might be unexpected. In our experiments, $5-10$ bits of precision on the inputs and weights of the network were sufficient in maintaining the accuracy of the neural network. All the numbers computed were smaller than 2^{80}, which guided us in selecting the parameters for the encryption scheme as seen in Section 3.2.5.

Max pooling cannot be computed directly since the max-function is non-polynomial. However, powers of it can be approximated due to the relation $\max(x_1, \ldots, x_n) = \lim_{d\to\infty} \left(\sum_i x_i^d\right)^{1/d}$. To keep the degree small, d should be kept reasonable small, with the smallest meaningful value $d = 1$ returning a scalar multiple of the mean pooling function. We will use this scaled mean-pool function instead of the max-pool function, as the sum $\sum x_i$ is easy to compute over encrypted data. The reason we use the scaled mean-pool instead of the traditional mean-pool is that we prefer not having to divide by the number of elements, although this could in principle be done (Dowlin et al., 2015). The only effect of not dividing is that the output gets scaled by a factor, which then propagates to the next layers.

The sigmoid and the rectified linear activation functions are non-polynomial functions. The solution of Xie et al. (2014) was to approximate these functions with low-degree polynomials, but we take a different approach here. We try to control the trade-off between having a non-linear transformation, which is needed by the learning algorithm, and the need to keep the degree of the polynomials small, to make the homomorphic encryption parameters feasible. We chose to use the lowest-degree non-linear polynomial function, which is the square function: $\operatorname{sqr}(z) := z^2$. It is interesting to note that Livni et al. (2014) have recently suggested a theoretical analysis of the problem of learning neural networks with polynomial activation functions and devoted much of their study to the square activation function.

As a conclusion, to make a network compatible with homomorphic encryption some modifications are needed. Preferably, these modifications should be taken into account while training. The activation functions should be replaced by polynomial activation functions and the max pooling replaced by scaled mean pooling. For the sake of time-efficient evaluation, consecutive layers that use only linear transformations, such as the weighted-sum or mean pooling, can be collapsed.

3 Homomorphic Encryption

Encrypting data is a prominent method for securing and preserving privacy of data. Homomorphic encryption (HE) (Rivest et al., 1978) adds to that the ability to act on the data while it is still encrypted. In mathematics, a *homomorphism* is a *structure-preserving* transformation. For example, consider the map $\Phi : \mathbb{Z} \to \mathbb{Z}_7$ such that $\Phi(z) := z \pmod 7$. This map Φ preserves both the additive and multiplicative structure of the integers in the sense that for every $z_1, z_2 \in \mathbb{Z}$, we have that $\Phi(z_1 + z_2) = \Phi(z_1) \oplus \Phi(z_2)$ and $\Phi(z_1 \cdot z_2) = \Phi(z_1) \otimes \Phi(z_2)$ where $\oplus$ and $\otimes$ are the addition and multiplication operations in $\mathbb{Z}_7$. The map Φ is a ring homomorphism between the rings $\mathbb{Z}$ and $\mathbb{Z}_7$.

In the context of homomorphic encryption, we will be interested in preserving the additive and multiplicative structures of the *rings* of plaintexts and ciphertexts in the encryption and decryption operations. Since the first such encryption scheme was introduced (Gentry, 2009), there have been many advances in this field (see e.g. Naehrig et al. (2011); Gentry et al. (2012a,b)). Technically speaking, (fully) homomorphic encryption allows for an arbitrary number of addition and multiplication operations to be performed on the encrypted data. For the sake of efficiency, we will instead use a weaker variant of this idea often called leveled homomorphic encryption, where the parameters of the encryption scheme are chosen so that arithmetic circuits of (roughly speaking) a predetermined depth can be evaluated. In our case this amounts to knowing the structure of the neural network, including the activation functions. The particular encryption scheme that we employ is YASHE', described in Bos et al. (2013).

3.1 Description of the method

The encryption scheme of Bos et al. (2013) maps plaintext messages from the ring $R_t^n := \mathbb{Z}_t[x]/(x^n + 1)$ to the ring $R_q^n := \mathbb{Z}_q[x]/(x^n + 1)$. See Appendix A.1 for a brief introduction to rings and their properties. The encryption scheme chooses random polynomials $f', g \in R_q^n$, and defines $f := tf' + 1$. The public key h is defined to be $h := tgf^{-1}$, while f is the secret key. Since not every element in R_q^n is invertible, these steps are iterated until the corresponding f has an inverse and h can be computed.

A message $m \in R_t^n$ is encrypted by computing

$$c := \left[\lfloor q/t \rfloor m + e + hs\right]_q$$

where e and s are random noise polynomials in R_q^n, with coefficients of small absolute value. We use the notation $[a]_q$ (resp. $[a]_t$) to denote the reduction of the coefficients of a modulo q (resp. t) to the symmetric interval of length q (resp. t) around 0. Decrypting is done by computing

$$m := \left[\left\lfloor \frac{t}{q} fc \right\rceil\right]_t .$$

Here the product fc is first computed in R_q^n, the coefficients are interpreted as integers, scaled by t/q, and rounded to the nearest integers. Finally they are interpreted modulo t.

Two ciphertexts c_1 and c_2, with underlying messages m_1 and m_2, can be added together in R_q^n to yield the encryption of $m_1 + m_2$. This works because

$$\begin{aligned} c_1 + c_2 = \lfloor q/t \rfloor (m_1 + m_2) + (e_1 + e_2) \\ + h(s_1 + s_2) , \end{aligned} \tag{1}$$

which decrypts to $m_1 + m_2 \in R_t^n$.

To multiply two messages we first compute

$$\left\lfloor \frac{t}{q} c_1 c_2 \right\rceil = \lfloor q/t \rfloor (m_1 m_2) + e' + h^2 s_1 s_2 \tag{2}$$

where e' is a noise term that under the right conditions is still small. It is easy to see that the term above decrypts to $m_1 \cdot m_2$, but under the secret key $f^2 \in R_q^n$. Using a process called *relinearization* (Bos et al., 2013) it is possible to modify the result so that it will be decryptable under the original secret key.

3.2 Practical considerations

The first thing to note is that the method described above works as long as the noise terms appearing in the encryptions of m_1 and m_2 are small enough. Otherwise the decryptions might not yield correct answers. The security level of the system depends on the parameters n, q, t, and the amount of noise added. The maximum amount of noise that a ciphertext can have and still be decryptable depends on the parameters q and t.

When ciphertexts are added or multiplied, the noise in the resulting ciphertext is typically larger than in the inputs. Noise growth is particularly strong in multiplication. This essentially means that the parameter q should be selected to be large enough to support the increased noise, which necessitates choosing a larger n for security reasons.

If the computation to be performed is expressed as an arithmetic circuit with addition and multiplication nodes, the main limitation to using the scheme is the number of multiplication gates in the path from the inputs to the outputs. This number we refer to as the *level*. Keeping the level low allows for selecting smaller values for the parameters, which results in faster computation and smaller ciphertexts. Note that the level is not the same as the degree of a polynomial to be evaluated, and instead behaves like the logarithm of the degree.

While keeping the parameters small improves performance, for our tasks, we would like to make t large to prevent the coefficients of the plaintext polynomials from reducing modulo t at any point during the computation. To better understand this point, note that the atomic objects used in a neural network are real numbers. The neural network takes as its input a vector of real numbers and, through a series of additions, multiplications, and other real functions, it computes its outputs, which are also real numbers. However, the homomorphic encryption scheme works over the ring $R_t^n := \mathbb{Z}_t[x]/(x^n + 1)$. This means that some conversion process between real numbers and elements of R_t^n is needed. We refer to such conversions as encodings (real numbers to R_t^n) and decodings (R_t^n to real numbers). If the coefficients of a polynomial in R_t^n are reduced modulo t after say, an addition, there is usually a problem with decoding it correctly to the sum of the real numbers, and instead the result is likely to be unexpected. This is why we need to keep track of how large the coefficients of the plaintext polynomials grow throughout the entire computation, and choose the parameter t to be larger.

To make the computations faster, it is also important to keep track of what parts of the data need to be secured. A common task that is repeatedly performed in the neural network is computing the weighted some of the inputs from the previous layer. While the data from the previous layer is encrypted, the weights are known to the network in their plain form. Therefore, when multiplying the data by the weights we can use a more efficient form of multiplication, described below.

3.2.1 Plain operations

In applying neural networks a common operation is to add or multiply some value, which is derived from the data, with some known constant. The naive way to implement such operations is to first encrypt the constant and than perform the addition or multiplication operation. However, this process is both computationally intensive and adds a large amount of noise if the operation is multiplication. However, this is not necessary. Let $c = \lfloor q/t \rfloor m + e + hs$ be the encrypted message and w the known constant. Addition can be achieved by multiplying w by $\lfloor q/t \rfloor$ and adding that to c, which results in $\lfloor q/t \rfloor (m + w) + e + hs$. This is essentially just encrypting w with no noise and performing normal homomorphic addition.

For multiplication, even the scaling is not needed since $cw = \lfloor q/t \rfloor mw + e' + hs'$. This is very efficient, especially if w is a sparse polynomial. For example, if w is a scalar (as it would be in the scenario below), then this multiplication is computed in linear time in the degree of c, which is $n-1$.

3.2.2 Encoding

As we already discussed above, there is a mismatch between the atomic constructs in neural networks (real numbers), and the atomic constructs in the homomorphic encryption schemes (polynomials in R_t^n). An encoding scheme should map one to the other in a way that preserves the addition and multiplication operations. Such an encoding scheme can be constructed in several ways. For example, it is possible to convert the real numbers to fixed precision numbers, and then use their binary representation to convert them into a polynomial with the coefficients given by the binary expansion. This polynomial will have the property that when evaluated at 2 it will return the encoded value. Another option is to encode the fixed precision number as a constant polynomial. This encoding is simple, but might seem inefficient in the sense that only one coefficient of the polynomial is being used. In Section 3.2.4 we show how multiple such instances can be encoded simultaneously to make use of the entire space. One problem with such a *scalar encoding* is that the only coefficient of the message polynomials grows very rapidly when homomorphic operations are performed

3.2.3 Encoding large numbers

As we have already explained, a major challenge for computing in this encryption scheme lies in preventing the coefficients of the plaintext polynomials from overflowing t. This forces us to choose large values for t, which causes the noise to grow more rapidly in the ciphertexts and decreases (with q fixed) the maximum amount of noise tolerated. Therefore, we need to choose a larger q, and subsequently a larger n for security reasons. One way to partially overcome this issue is by using the Chinese Remainder Theorem (CRT) (see Section A.2). The idea is to use multiple primes $t_1, \ldots, t_k$. Given a polynomial $\sum a_i x^i$ we can convert it to k polynomials in such a way that the j-th polynomial is $\sum [a_i \pmod{t_j}] x^i$. Each such polynomial is encrypted and manipulated identically. The CRT guarantees that we will be able to decode back the result, as long as its coefficient do not grow beyond $\prod t_j$. Therefore, this method allows us to encode exponentially large numbers while increasing time and space linearly in the number of primes used.

3.2.4 Parallel Computation

The encryption uses polynomials of high degree. For example, in our case $n = 8192$, both ciphertext and plaintext polynomials can have degree up to 8191. If the data is encoded as a scalar, only one out of the 8192 coefficients is being used, while all the operations (additions and multiplications) act on the entire 8192 coefficient polynomials. Therefore, the operations are slow due to the high degree, but the result contains only a single significant coefficient. Another application of the CRT allows us to perform Single Instruction Multiple Data (SIMD) operations at no extra cost Gentry et al. (2012b). Assume that t is selected such that $x^n + 1 \equiv \prod (x - \alpha_i) \pmod{t}$. In this case the CRT can be used to show that $R_t^n \cong \mathbb{Z}_t^{\times n}$. The isomorphism is explicit and easy to compute, which means that we can encode n values into a single polynomial, operate on this polynomial, and decode the n different results.

Note that we use here the CRT in an opposite direction to how we use it when encoding large numbers in Section 3.2.3). When encoding large numbers, we take a single number and break it into multiple small numbers that are being processed in parallel and joined together at the end. On the other hand, here we take multiple scalars and join them together to form a single polynomial. This polynomial is being processed as a single unit and only upon completing the computation is it broken into its components.

3.2.5 Parameter Selection

The main parameters defining the cryptosystem are the plaintext modulus t, the coefficient modulus q and the degree n of the polynomial modulus $(x^n + 1)$. To allow for encoding large enough numbers for the purposes of the network, we used two plaintext moduli and both of the CRT techniques described above. The values used are $t_1 = 1099511922689$ and $t_2 = 1099512004609$. They were selected so that their product is greater than 2^{80}, which is large enough for applying the network. Moreover, they are small enough so that with the coefficient modulus $q = 2^{383} - 2^{33} + 1$ and the polynomial modulus $x^{8192} + 1$ they allow for computing the desired network correctly, i.e. so that the noise does not grow too large. Finally, the plaintext moduli are chosen such that

$$x^{8192} + 1 = \prod_{i=0}^{8191} \left(x - \alpha_i^{1,2} \right) \pmod{t_{1,2}}$$

In other words, the polynomial modulus breaks into linear components, which allows for optimal use of the SIMD technique described in Section 3.2.4.

4 Empirical Results

We have tested CryptoNets on the MNIST dataset (LeCun et al., 1998). This dataset consists of 60,000 images of hand written digits. Each image is a 28x28 pixel array, where each pixel is represented by its gray level in the range of 0-255. We used the training part of this dataset, consisting of 50,000 images, to train a network and the remaining 10,000 images for testing. The details of the network used are presented in Table 1. The accuracy of the training network is 99% (it mislabels only 105 out of the $10,000$ test examples).

4.1 Timing analysis

Since the network can accept batches of size 8192 (due to the choice of the degree $n = 8192$ in the encryption parameters) we timed the network on the first 8192 images of the test set to match the batch size. The latency of the network is governed by the time to process a batch while the throughput is also a function of the batch size. Therefore, we separated the report for these two parameters and also reported on the time per instance. These results are presented in Table 2.

Applying the network takes 570 seconds on a PC with a single Intel Xeon E5-1620 CPU running at 3.5GHz, with 16GB of RAM, running the Windows 10 operating system. Since applying the network allows making 8192 predictions simultaneously using the SIMD operations as described in Section 3.2.4, this PC can sustain a throughput of $8192 \times 3600/570 \approx 51739$ predictions per hour. Encrypting the data takes 122 seconds and additional 0.060 seconds for every parallel instance to be encoded. Therefore, if 8192 instances are encoded, a throughput of 48068 instances per hour can be encrypted and encoded. Decrypting the data takes 5 seconds and additional 0.046 seconds to decode predictions for each instance. Therefore, a throughput of 77236 decryptions and decoding per hour is achievable with our setup.

4.2 Description of the Network

The network has two forms: the model that is the direct output of training, and the simplified version which is actually used for making predictions. The trained network has 9 layers, and the simplified version has 5 layers. A visualization of the latter is given in Table 1, though we will describe both of them here.

Here is a description of the network used for training:

1. *Convolution Layer*: The input image is 28×28. The convolution has windows, or kernels, of size 5×5, a stride of $(2, 2)$, and a mapcount of 5. The output of this layer is therefore $5 \times 13 \times 13$.
2. *Square Activation Layer*: This layer squares the value at each input node.
3. *Scaled Mean Pool Layer*: This layer has $1 \times 3 \times 3$ windows, and again outputs a multi-array of dimension $5 \times 13 \times 13$.
4. *Convolution Layer*: This convolution has a kernel size of $1 \times 5 \times 5$, a stride of $(1, 2, 2)$, and a mapcount of 10. The output layer is therefore $50 \times 5 \times 5$.
5. *Scaled Mean Pool Layer*: As with the first mean pool, the kernel size is $1 \times 3 \times 3$, and the output is $50 \times 5 \times 5$.
6. *Fully Connected Layer*: This layer fully connects the incoming $50 \cdot 5 \cdot 5 = 1250$ nodes to the outgoing 100 nodes, or equivalently, is multiplication by a 100×1250 matrix.
7. *Square Activation Layer*: This layer squares the value at each input node.
8. *Fully Connected Layer*: This layer fully connects the incoming 100 nodes to the outgoing 10 nodes, or equivalently, is multiplication by a 10×100 matrix.
9. *Sigmoid Activation Function*: This layer applies the sigmoid function to each of the 10 incoming values.

The sigmoid activation function is necessary for the training stage in order to get reasonable error terms when running the gradient descent algorithm. However, we don't have a good way of dealing with the sigmoid in the encrypted realm. Luckily, once we have our weights fixed and want to make predictions, we can simply leave it out. This is because the prediction of the neural network is given by the index of the maximum value of its output vector, and since the sigmoid function is monotone increasing, whether or not we apply it will not affect the prediction.

The other change that we make to the network is just for an increase in efficiency. Since layers 3 through 6 are all linear, they can all be viewed as matrix multiplication and composed into a single linear layer corresponding to a matrix of dimension 100 by $5 \cdot 13 \cdot 13 = 865$. Thus, our final network for making predictions is only 5 layers deep.

One obstacle to training networks using the square activation function is that, unlike the rectified linear and sigmoid functions, its derivative is unbounded. This can lead to strange behavior when running the gradient descent algorithm. Especially for deeper nets it sometimes blows up or overfits. The overfitting issue can be partially resolved by the addition of convolution layers without activation functions (layers 4 and 5 in our network). This allows us to reduce the number of degrees of freedom in the output polynomial. However, for even deeper nets (10 to 20 layers) something else will be needed to aid in training.

4.3 Message sizes

The images consist of 28×28 pixels. Each pixel is encrypted as 2 polynomials (two values for t are used together with CRT to allow for the large numbers needed). Each coefficient in the polynomial requires 48 bytes and therefore, each image requires $28 \times 28 \times 8192 \times 2 \times 48$ bytes or 588 MB. However, the same message can contain 8192 images and therefore, the per image message size is only 73.5 KB. The response of the classifier contains only 10 values (for the 10 possible digits) and therefore the message size is $10 \times 8192 \times 2 \times 48$ which is 7.5 MB or 0.94 KB per image, when 8192 images are encoded together. These numbers are summarized in Table 3.

It is interesting to put the message sizes used in comparison to natural raw representations of these messages. The size of the message depends on the representation used. For example, if each image is represented as an array of size 28x28 and each pixel is represented as a double precision floating point number, then the size of each image is approximately 6 KB, which is 12 times smaller than the encrypted version. More concise representation is possible if only a single byte is used to represent each pixel, which will bring the ratio between the encrypted size to the unencrypted size to 96. The sparsity of the data allows compressing the data even further, and indeed the compressed version of this dataset has an average of only 165 bytes per instance. Therefore, comparing to that, the encrypted version would be 456 times larger. In conclusion, the encrypted data is one to three orders of magnitude larger than the unencrypted data. The exact factor depends on what is considered a natural representation of the data in its raw form.

5 Discussion and Conclusions

The growing interest in *Machine Learning As a Service* (MLAS), where a marketplace of predictors is available on a pay-per-use basis, requires attention to the security and privacy of this model. Not all data types are sensitive, but in many applications in medicine, finance, and marketing the relevant data on which predictions are to be made is typically very sensitive.

Different methods can be used to protect the data. For example, the prediction provider and the data owner can encrypt the data while in transit using traditional cryptography. These methods are promising in terms of throughput, latency, and accuracy, but they require some way to establish trust between the cloud and the data owner. The provider also needs to guarantee the safety of the keys, and the safety of the data against attackers while it is stored in the cloud.

Another possible approach would be using secure *Multi-Party Computation* (MPC) techniques (Goldreich, 1998). Most MPC methods establish a communication protocol between the parties involved, such that if the parties follow the protocol they will end with the desired results while protecting the security and privacy of their respective assets (Barni et al., 2006; Orlandi et al., 2007; Piva et al., 2008; Chen & Zhong, 2009). Barni et al. (2006) presented a method of this type, where the data owner encrypts the data and sends it to the cloud. The cloud computes an inner product between the data and the weights of the first layer, and sends the result to the data owner. The data owner decrypts, applies the non-linear transformation, and encrypts the result before sending it back to the cloud. The cloud can apply the second layer and send the output back to the data owner. The process continues until all the layers have been computed. In Orlandi et al. (2007) they also noted that this procedure leaks much of the information of the weights of the network to the data owner, and added a method to obscure the weights. The main difference between these methods and the method we describe in this paper is that in our method the data owner does not have to maintain a constant presence while the neural network is evaluated. For example, the data owner can encrypt the data and store it in the cloud in its encrypted form. The cloud can apply one or several networks to the data while the data owner is offline. Whenever the data owner wishes to read the predictions, it can retrieve the information and decrypt it, allowing the data owner to maintain a much simpler infrastructure. Moreover, since intermediate results are not shared, less information is leaked from the cloud to the data owner.

Graepel et al. (2013) suggested the use of homomorphic encryption for machine learning algorithms. They focused on finding algorithms where the training can be done over encrypted data, and therefore were forced to use learning algorithms in which the training algorithm can be expressed as a low degree polynomial. As a result, most of the algorithms proposed were of the linear discrimination type. Several authors also looked at nearest neighbor classification (Zhan et al., 2005; Qi & Atallah, 2008). However, linear classifiers and nearest neighbor classifiers do not deliver the same level of accuracy that neural networks are capable of.

Aslett et al. (2015a,b) presented ways to train machine learning models over data encrypted with homomorphic encryption. They presented both simple algorithms, such as naive Bayes classifiers, as well as more involved random models such as random forests and some variations of it. Their work differs from our work in several major aspects: The models they propose work well on some tasks, but do not compete well with neural networks on tasks such as recognizing objects in images. They also had to use a unique coding scheme, in which values are compared to threshold before encryption, to allow the learning algorithm to work. CryptoNets imposes fewer requirements on the data owner,

Table 1: Breakdown of the time it takes to apply CryptoNets to the MNIST network

Layer	Description	Time to compute
Convolution layer	Weighted sums layer with windows of size 5×5, stride size of 2. From each window, 5 different maps are computed and a padding is added to the upper side and left side of each image.	46 seconds
1st square layer	Squares each of the 835 outputs of the convolution layer	290 seconds
Pool layer	Weighted sum layer that generates 100 outputs from the 835 outputs of the 1st square layer	195 seconds
2nd square layer	Squares each of the 100 outputs of the pool layer	36 seconds
Output layer	Weighted sum that generates 10 outputs (corresponding to the 10 digits) from the 100 outputs of the 2nd square layer	3 seconds

Table 2: The performance of CryptoNet for MNIST

Stage	Latency	Additional latency per each instance in a batch	Throughput
Encoding+Encryption	122 seconds	0.060 seconds	48068 per hour
Network application	570 seconds	0	51739 per hour
Decryption+Decoding	5 seconds	0.046 seconds	77236 per hour

Table 3: Message sizes of CryptoNet for MNIST

	Message size	Size per instance
Owner $\rightarrow$ Cloud	588 MB	73.5 KB
Cloud $\rightarrow$ Owner	7.5 MB	0.94 KB

and allows the use of neural networks, however it does not support training on the encrypted data.

Training neural networks over encrypted data is still possible. If all the activation functions are polynomials, and the loss function is polynomial too, back-propagation can be computed using additions and multiplications only. However, there are several challenges in doing so. Computational complexity is a major challenge. Even when trained on plaintext, neural networks are slow to train. Today, much of the effort in the field of machine learning goes towards accelerating this training process by using sophisticated hardware such as GPUs. However, adding homomorphic encryption to the process will make the process at least an order of magnitude slower. It is more likely that the slowdown would be much worse since the level of the computed polynomial is proportional to the number of back-propagation steps made, and therefore using leveled homomorphic encryption does not seem to be feasible. Another challenging aspect in the presence of encryption is the lack of ability of a data scientist to inspect the data and the trained models, to correct mislabeled items, to add features, and to tune the network.

The main contribution of this work is a method that enjoys the accuracy of neural networks with the simplicity of use of homomorphic encryption. By combining techniques from cryptography, machine learning, and engineering, we were able to create a setup in which both accuracy and security are achieved, while maintaining a high level of throughput. This work leaves much room for improvement, however. For example, the throughput and latency can be significantly improved by using GPUs and FPGAs to accelerate the computation. Another direction for further progress would be finding more efficient encoding schemes that allow for smaller parameters, and hence faster homomorphic computation.

References

Agrawal, Rakesh and Srikant, Ramakrishnan. Privacy-preserving data mining. In *ACM Sigmod Record*, pp. 439–450. ACM, 2000.

Aslett, Louis JM, Esperança, Pedro M, and Holmes, Chris C. Encrypted statistical machine learning: new privacy preserving methods. *arXiv preprint arXiv:1508.06845*, 2015a.

Aslett, Louis JM, Esperança, Pedro M, and Holmes, Chris C. A review of homomorphic encryption and software tools for encrypted statistical machine learning. *arXiv preprint arXiv:1508.06574*, 2015b.

Barni, Mauro, Orlandi, Claudio, and Piva, Alessandro. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pp. 146–151. ACM, 2006.

Bos, Joppe W, Lauter, Kristin, Loftus, Jake, and Naehrig, Michael. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding*, pp. 45–64. Springer, 2013.

Chen, Tingting and Zhong, Sheng. Privacy-preserving backpropagation neural network learning. *Neural Networks, IEEE Transactions on*, 20(10):1554–1564, 2009.

Dahl, George E, Yu, Dong, Deng, Li, and Acero, Alex. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

Dowlin, Nathan, Gilad-Bachrach, Ran, Laine, Kim, Lauter, Kristin, Naehrig, Michael, and Wernsing, John. Manual for using homomorphic encryption for bioinformatics. Technical report, Microsoft Research, 2015. `http://research.microsoft.com/apps/pubs/default.aspx?id=258435`.

Dwork, Cynthia. Differential privacy. In *Encyclopedia of Cryptography and Security*, pp. 338–340. Springer, 2011.

Eisenbud, David. *Commutative Algebra: with a view toward algebraic geometry*, volume 150. Springer Science & Business Media, 1995.

Gentry, Craig. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pp. 169–178, 2009.

Gentry, Craig, Halevi, Shai, and Smart, Nigel P. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology–EUROCRYPT 2012*, pp. 465–482. Springer, 2012a.

Gentry, Craig, Halevi, Shai, and Smart, Nigel P. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pp. 850–867. Springer, 2012b.

Goldreich, Oded. Secure multi-party computation. *Manuscript. Preliminary version*, 1998.

Graepel, Thore, Lauter, Kristin, and Naehrig, Michael. Ml confidential: Machine learning on encrypted data. In *Information Security and Cryptology–ICISC 2012*, pp. 1–21. Springer, 2013.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

LeCun, Yan, Cortes, Corinna, and Burges, Christopher J.C. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

Livni, Roi, Shalev-Shwartz, Shai, and Shamir, Ohad. On the computational efficiency of training neural networks. In *Advances in Neural Information Processing Systems*, pp. 855–863, 2014.

Naehrig, Michael, Lauter, Kristin, and Vaikuntanathan, Vinod. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pp. 113–124. ACM, 2011.

Orlandi, Claudio, Piva, Alessandro, and Barni, Mauro. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007:18, 2007.

Piva, Alessandro, Orlandi, Claudio, Caini, M, Bianchi, Tiziano, and Barni, Mauro. Enhancing privacy in remote data classification. In *Proceedings of The Ifip Tc 11 23rd International Information Security Conference*, pp. 33–46. Springer, 2008.

Qi, Yinian and Atallah, Mikhail J. Efficient privacy-preserving k-nearest neighbor search. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pp. 311–319. IEEE, 2008.

Rivest, Ronald L, Adleman, Len, and Dertouzos, Michael L. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

Xie, Pengtao, Bilenko, Misha, Finley, Tom, Gilad-Bachrach, Ran, Lauter, Kristin, and Naehrig, Michael. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

Zhan, Justin Zhijun, Chang, LiWu, and Matwin, Stan. Privacy preserving k-nearest neighbor classification. *IJ Network Security*, 1(1):46–51, 2005.

A Commutative Algebra

Many of our results rely on concepts in commutative algebra that might be unfamiliar to some readers. In this section we provide some background on the concepts used in this paper. We refer the reader to Eisenbud (1995) for a comprehensive introduction to the field.

A.1 Rings

A commutative ring R is a set on which there are two operations defined: addition and multiplication, such that there is $0 \in R$ which is the identity element for addition and $1 \in R$ which is the identity for the multiplication operation. For every element $a \in R$ there exists an element $-a \in R$

such that $a + (-a) = 0$. Furthermore, the following hold for every $a, b, c \in R$:

$$\begin{aligned} a\,(bc) &= (ab)\,c\,; \\ a\,(b+c) &= ab+ac\,; \\ (a+b)\,c &= ac+bc\,; \\ a+(b+c) &= (a+b)+c\,; \\ a+b &= b+a\,; \\ ab &= ba\,. \end{aligned}$$

Since all the rings we discuss in this work are commutative rings, we use the term "ring" to refer to a "commutative ring".

Several rings appear in this work. The set $\mathbb{Z}$ of integers is a ring, as is the set $\mathbb{Z}_m$ of integers modulo m, whose elements can be thought of as sets of the form $\{i + am \,:\, a \in \mathbb{Z}\}$, where i is an integer. When we write $k \in \mathbb{Z}_m$, we mean the set $\{k + am \,:\, a \in \mathbb{Z}\}$. Conversely, we say that $k \in \mathbb{Z}$ represents, or is a representative of, this element of $\mathbb{Z}_m$.

The set $R[x]$ of polynomials with coefficients in a ring R is itself a ring. In this work we deal a lot with the ring $\mathbb{Z}_m\,[x]$ of polynomials with integer coefficients modulo m. Finally, the set $\mathbb{Z}_m\,[x]\,/(x^n+1)$, whose elements can be thought of as sets of the form $\{p\,(x) + q\,(x)\,(x^n+1) \,:\, q\,(x) \in \mathbb{Z}_m\,[x]\}$, where $p(x) \in \mathbb{Z}_m[x]$, is a ring. When we write $r(x) \in \mathbb{Z}_m[x]/(x^n+1)$ we mean the set $\{r\,(x) + q\,(x)\,(x^n+1) \,:\, q\,(x) \in \mathbb{Z}_m\,[x]\}$, and conversely say that $r(x)$ represents, or is a representative of, this element of $\mathbb{Z}_m\,[x]\,/(x^n+1)$. The polynomials with coefficients in some fixed set of representatives of elements of $\mathbb{Z}_m$, and of degree at most $n-1$, form a complete set of representatives of elements of $\mathbb{Z}_m\,[x]\,/(x^n+1)$. To simplify the notation, we refer to the ring $\mathbb{Z}_m\,[x]\,/(x^n+1)$ as R_m^n.

A.2 Chinese Remainder Theorem (CRT)

An element $p \in R$ is said to be prime if for every $f, g \in R$ it is true that if p divides fg, then p divides at least one of f and g. The Chinese Remainder Theorem states that $R \cong \prod_i R/(p_i)$ when the $p_1, \ldots, p_n$ are distinct primes. This should be interpreted as follows: An element $r \in R$ is uniquely represented by elements $r_1, \ldots, r_n$ such that $r_i \in R/(p_i)$. This allows breaking $r \in R$, which might be large (in some sense), into n "small" values $r_1, \ldots, r_n$. At the same time, it is also true that every $r_1, \ldots, r_n$ such that $r_i \in R/(p_i)$ has a unique $r \in R$ that represents it. This allows us to pack n "small" values $r_1, \ldots, r_n$ into a single large value $r \in R$.

The Chinese Remainder Theorem can be written in an explicit "constructive" form. The transformation from R to $\prod_i R/(p_i)$ is the easier one, and is simply given by sending r to the the sets $\{r + qp_i \,:\, q \in R\}$ for each i. In the other direction, given $r_1, \ldots, r_n$, they can be mapped to $\sum q_i r_i$, where $q_i \in R$ are such that for every $j \neq i$, p_j divides q_i, and p_i divides $q_i - 1$. The values of the q_i can be computed as follows: First let $\hat{q}_i := \prod_{j \neq i} p_i$. Next, let $\hat{q}_i^{-1} \in R$ be such that p_i divides $\hat{q}_i \hat{q}_i^{-1} - 1$. This is always possible when the ideals (p_i) and (p_j) are coprime (Eisenbud, 1995), which is the case when R is the ring of integers, or a polynomial ring over integers modulo a prime number. Finally, let $p_i := \hat{q}_i \hat{q}_i^{-1}$.