1 Introduction

In this report we consider following task: how to evaluate neural networks on encrypted data. Neural networks have been shown to outperform other learning models for many tasks. We want to consider the case where the neural network is held by the third party and user wants to use its predictions without compromising privacy. This means that third party must not learn anything about user’s input and user may not learn anything about neural network except the final prediction. For example this may be useful for hospitals. Hospitals may want to predict something based on patient data, but this data is private, so they can not just send it to third party. Also model owner may not want to share the model, because he may want to ask money for every prediction.

This report presents ways to construct neural networks, which can be evaluated on encrypted data using fully homomorphic encryption. These constructions are presented in articles [1] and [2].

Sections 2 and 3 give brief overview of neural networks and fully homomorphic encryption. Section 4 presents results from articles [1] and [2].

2 Neural networks

By neural network we mean artificial feedforward neural network. Neural network is machine learning model, which can be used for wide variety of tasks. Like other machine learning models, neural network must be first trained using labeled data. After neural network is trained, it can be used to receive results about new data.

Neural networks consists of neurons and connections between them. Each connection is directed. We have some input neurons, which does not have any incoming connections and we have output neurons, which does not have any outgoing connections. Feedforward neural networks are such that there are no directed cycles in network. In this report we consider only feedforward neural networks.

Neural networks can be split into layers. First we have input layer, which contains all the input neurons. After that we have first hidden layer. This contains neurons to
which input layer has connections. After that we have second hidden layer, which contains neurons to which first hidden layer has connections. We may have only one hidden layer, but we may have really many of them. Finally we have output layer which contains of output neurons. We can order layers so that input layer is the first one, then comes first hidden layer, then second and so on until after last hidden layer is the output layer. The connections are usually between adjacent layers, but sometimes they can skip layers. In this report we look only the cases where connections are between adjacent layers.

Each neuron has activation value, which in our case can be arbitrary real number. Each connection has weight. Activations of input layer are taken from input of neural network. Activations of other layers are calculated based on activations of previous layer and connections between layers. Output of neural network is activations of output neurons.

### 2.1 Neural network layers

Neural network can consist of different layers. We now introduce fully connected, activation, convolutional and pooling layers.

#### 2.1.1 Fully connected layer

The most common layer is fully connected layer. In this layer there are connections from every neuron in previous layer to every neuron in this layer. Every neuron also has its bias value.

Assume that the weight between \( i \)-th neuron in previous layer and \( j \)-th neuron in current layer is \( w_{ij} \) and bias of \( j \)-th neuron in current layer is \( b_j \). The activation of \( i \)-th neuron in previous layer is denoted by \( x_i \). Then the activation of \( j \)-th neuron in current layer is

\[
b_j + \sum_i w_{ij}x_i.
\]

#### 2.1.2 Activation layer

Fully connected layer is usually followed by activation layer. Activation layer contains as many neurons as previous layer. Activation layer uses activation function. If activation
function is $f$ and activation of $i$-th neuron in previous layer is $x_i$, then activation of $i$-th neuron in current layer is $f(x_i)$.

Historically one of the most popular activation functions has been sigmoid function. Sigmoid is defined by the formula

$$f(x) = \frac{1}{1 + e^{-x}}.$$

ReLU is activation function, which have become popular more recently. But at the moment it is one of the most used activation function. ReLU stands for rectified linear unit. It is defined by the following formula

$$f(x) = \max(0, x).$$

There are of course many other activation functions and in principle you can use almost anything as long as it has not too ugly derivative. The main reason why we need activation function, is that it makes our network non-linear. Otherwise we could learn only linear functions. So using $f(x) = x$ as activation function is not good idea. However many activation functions have problems, which make training more difficult.

2.1.3 Convolutional layer

Fully connected layer contains many weights, which all need to be adjusted during training. If previous layer contains $m$ neurons and current layer contains $n$ neurons, then there is $mn$ weights. If there is too many neurons in layers, then training becomes too slow.

In image processing one solution to this problem is to use convolutional layer. In this approach we do not look neurons of layer as one-dimensional array, but as multidimensional array. Usually we use three-dimensional array, because images are naturally three-dimensional. One dimension for width, one dimension for height and last dimension for colors.

![Convolutional layer](image)

Convolutional layer then has filter, which has same amount of dimensions as previous layer and which contains weights. Activation of neuron in current layer is found when filter is applied to some part of the previous layer. For convolutional layer we must also specify stride. Stride says how much we move filter in every step for finding activations.
Assume the previous layer has size \( n_1 \times n_2 \), the filter has size \( k_1 \times k_2 \) and the stride is \((m_1, m_2)\). Then size of the current layer is \((\frac{n_1-k_1}{m_1} + 1) \times (\frac{n_2-k_2}{m_2} + 1)\). Assume activation of neuron in \( i\)-th row and \( j\)-th column in previous layer is \( x_{ij} \) and weight in \( i\)-th row and \( j\)-th column in filter is \( w_{ij} \). Then activation of \( a\)-th row and \( b\)-th column in current layer is

\[
\sum_{i=1}^{k_1} \sum_{j=1}^{k_2} w_{ij} x_{cd},
\]

where \( c = (a - 1)m_1 + i \) and \( d = (b - 1)m_2 + j \).

Often multiple different filters are used resulting in that many more neurons in current layer. For example if we apply \( K \) filters, we have \( K \) times more output neurons.

### 2.1.4 Pooling layer

Main goal of the pooling layer is to reduce number of neurons. Again stride must be chosen and also size of the window. Most used pooling is max pooling. Assume that previous layer is with size \( n_1 \times n_2 \), window is with size \( k_1 \times k_2 \) and stride is \((m_1, m_2)\). Assume activation of neuron in \( i\)-th row and \( j\)-th column in previous layer is \( x_{ij} \). Then activation of \( a\)-th row and \( b\)-th column in current layer is

\[
\max_{i \in A, j \in B} x_{ij},
\]

where \( A \) is set of integers from \((a - 1)m_1 + 1\) to \((a - 1)m_1 + k_1\) and \( B \) is set of integers from \((b - 1)m_2 + 1\) to \((b - 1)m_2 + k_2\).

![Max pool layer](image)

Other poolings like average pooling is also used sometimes, but they are not so popular. Average pooling just takes average instead of maximum from same elements.

### 2.2 Training neural networks

To train neural network we first need data for which we know expected outputs. During training our goal is to change neural network weights and biases so it returns outputs as close to expected outputs as possible. First we must define, what does it mean for outputs to be close. For that we use cost function. Different cost functions can be used. One of the most popular cost function is mean squared error. Assume \( x_1, x_2, \ldots, x_n \) are inputs
with corresponding outputs \( y_1, y_2, \ldots, y_n \). Mean squared error cost function is defined by formula

\[
C(w, b) = \frac{1}{n} \sum_{i=1}^{n} ||y_i - f(x_i, w, b)||^2,
\]

where \( w \) is the vector of weights, \( b \) is the vector of biases and \( f(x_i, w, b) \) is output vector of neural network.

Goal of the training is to minimize cost function. Minimizing cost function can be done with gradient descent. Gradient of function is vector, which shows in what direction the function grows the fastest. So if we change the inputs of function by some small enough step in opposite direction of gradient, we will decrease the value of function. Assume \( v_{old} \) is the vector containing current weights and current biases. Then we can find new weights and biases by using formula

\[
v_{new} = v_{old} - \gamma \nabla C(v_{old}),
\]

where \( \gamma \) is learning rate and \( \nabla C \) is gradient of \( C \). Learning rate is small positive number, which shows how big steps we take towards the opposite of the gradient. Learning rate can not be too big, because then we may not actually decrease the cost function. It also can not be too small, because then the number of steps needed to reach the minimum may get very large, which would make algorithm slow.

There is efficient algorithm, which trains neural networks layer by layer using gradient descent. It is called backpropagation.

### 3 Fully homomorphic encryption

Fully homomorphic encryption scheme enables to do additions and multiplications on ciphertexts. Assume \( x \) and \( y \) are plaintexts. Then if \( E \) is encryption function of fully homomorphic encryption scheme, it holds that

- \( E(x + y) = E(x) \oplus E(y) \)
- \( E(xy) = E(x) \otimes E(y) \)

where \( \oplus \) and \( \otimes \) are some operations defined by the encryption scheme.

First fully homomorphic encryption scheme was proposed by Gentry in 2009 [3].

Very high level idea of all known fully homomorphic encryption schemes is similar. In this report we will give high level description of fully homomorphic encryption scheme presented in [4]. This is just to give idea how fully homomorphic encryption scheme works. Approaches described later in article work as good with any other fully homomorphic encryption scheme as long as it is reasonably fast.

#### 3.1 Learning with errors

Fix \( q, n \) and some distribution \( \chi \) over \( \mathbb{Z} \). Take \( n \)-dimensional vector \( s \) over \( \mathbb{Z}_q \). LWE distribution \( A_{n,q,s,\chi} \in \mathbb{Z}_q^{n+1} \) is sampled by taking \( a \) uniformly randomly from \( \mathbb{Z}_q^n \), taking \( e \) from \( \chi \) and outputting \( \langle a, (s, a) + e \mod q \rangle \).

LWE_{n,q,\chi} problem is to distinguish following two distributions:
1. uniformly random $s$ is taken from $\mathbb{Z}_q^n$ and then samples are taken from $A_{n,q,s,\chi}$;
2. samples are taken uniformly randomly from $\mathbb{Z}_q^{n+1}$.

$LWE_{n,q,\chi}$ assumption is that the $LWE_{n,q,\chi}$ problem is infeasible.

### 3.2 Fully homomorphic encryption scheme

We now give high level overview of the fully homomorphic encryption scheme presented in [4]. This scheme is based on the $LWE_{n,q,\chi}$ assumption.

Fix some $q$ and $N$. Then ciphertext $C$ is $N \times N$ matrix over $\mathbb{Z}_q$ with small entries relative to the $q$. Secret key $v$ is $N$-dimensional vector over $\mathbb{Z}_q$ with at least one big entry relative to $q$. Assume this entry is in $k$-th position. Then $C$ encrypts message $\mu$ if $Cv = \mu v + e$, where $e$ is error vector with small entries.

Assume $C_k$ is $k$-th row of $C$. Then $\langle C_k, v \rangle = \mu v_k + e_k$. Using the knowledge that $v_k$ is large and $\mu$ and $e_k$ are small we can find $\mu$ from $\langle C_k, v \rangle$. So we can decrypt.

Assume $C_1v = \mu_1 v + e_1$ and $C_2v = \mu_2 v + e_2$. Then

$$(C_1 + C_2)v = (\mu_1 + \mu_2)v + (e_1 + e_2).$$

So $C_1 + C_2$ is encryption of $\mu_1 + \mu_2$. Also

$$(C_1C_2)v = C_1(\mu_2 v + e_2) = \mu_2(\mu_1 v + e_1) + C_1e_2 = \mu_2\mu_1 v + (\mu_2 e_1 + C_1e_2).$$

So $C_1C_2$ is encryption of $\mu_1\mu_2$. However notice that we had requirement that error vector must be small. Since we had assumption that $\mu$ and $C$ are with small entries, it means that it is reasonable to hope that $e_1 + e_2$ and $\mu_2 e_1 + C_1 e_2$ are also small. But obviously error increases with every addition and multiplication. So if we apply multiple additions and multiplications in a row, then in one moment the error becomes too big and it is not possible to decrypt anymore. Also notice that multiplications increase error much more than additions. So the growth of the error depends mainly on the multiplicative depth of circuit, which means we want multiplicative depth to be small.

It is possible to use method called bootstrapping to make error vector smaller, but this is really expensive. So doing it too many times makes operations inefficient.

### 4 Evaluating neural network on encrypted data

Assume we have two parties. One party has some data and he wants to predict something based on it without anybody learning anything about this data. We will call this party "client". The second party has trained neural network, which can be used to predict the thing first party wants to know. Second party does not want to reveal this neural network to anybody. We will call this party "server". Our goal is to enable first party to make predictions without compromising privacy. In particular we have three goals:

- **Privacy.** We want to ensure that server does not learn anything about client’s data. We also want to ensure that client does not learn anything about server’s model except the result it returns.


• Efficiency. We want to use protocol which runs in reasonable time.

• Accuracy. We want the prediction accuracy to be close to the prediction accuracy of non-private neural network.

In this report we are looking the protocol where client sends data to server in encrypted form. Server has trained neural network, which he uses to make prediction. Neural network must be such that it includes only additions and multiplications. Then it is possible to use fully homomorphic encryption for evaluating neural network on encrypted data. Server sends back encrypted result.

In this case our first goal, privacy, is easy. Client sends only encrypted version of his data. So privacy of client is ensured by the security of encryption protocol. Server sends only encrypted result of neural network, so client can not learn anything else about neural network.

Efficiency is more interesting. As discussed in the section 3.2, speed of doing additions and multiplications using fully homomorphic encryption protocols is mainly determined by the multiplicative depth of the arithmetic circuit. We can look our neural network as polynomial, because it can contain only additions and multiplications. Multiplicative depth of our neural network is then equal with the degree of this polynomial. So we want to make the degree of this polynomial as small as possible.

Accuracy is more tricky and with neural networks, it usually can not be said how good it will be without testing. So we are going to look tests made on some networks to find how good accuracy we can get.

4.1 Constructing neural network containing only additions and multiplications

In general neural networks can contain very different and complex functions. We have to replace them somehow with functions containing only additions and multiplications. Because these are the only operations, which can be done using fully homomorphic encryption. We are going to look at each layer separately. Activations of our input layer are going to be encrypted, so activations of all the other layers are also going to be encrypted. But since we use already trained network, then weights and all the other parameters are not going to be encrypted.

4.1.1 Fully connected layer

The weight between $i$-th neuron in previous layer and $j$-th neuron in current layer is denoted by $w_{ij}$ and bias of $j$-th neuron in current layer is denoted by $b_j$. The activation of $i$-th neuron in previous layer is denoted by $x_i$. Then the activation of $j$-th neuron in current layer is

$$b_j + \sum_i w_{ij} x_i.$$ 

Activations of previous layer are encrypted, but weights and biases are not. This means that in this layer we have only additions and multiplications with non-encrypted values. So this layer does not increase multiplicative depth.
4.1.2 Convolutional layer

Convolutional layer is similar to the fully connected layer in the sense that it too contains only additions and multiplications with non-encrypted values. So this layer also does not increase multiplicative depth.

4.1.3 Pooling layer

Max pool layer needs ability to take maximum over some activations of previous layer. Since activations are encrypted, we can not do it. Instead it is possible to use average pooling or sum pooling. Sum pooling is just taking sum of values instead of maximum. In some cases this may decrease our accuracy, but in most of the cases, it should not have big impact. Taking sum or average of course does not increase our multiplicative depth.

4.1.4 Activation layer

This is the most tricky layer. Most commonly used activation functions (for example ReLu, sigmoid) are not polynomials. So we have to somehow replace them with something polynomial. Also notice that we can not take anything linear here. Otherwise our whole neural network would be linear. This would mean that we could only train it to approximate linear functions. However, in most of the cases we want to train it to approximate non-linear functions. So we have to use at least degree 2 polynomial, which means that this layer will increase multiplicative depth of our neural network. Later on we are going to look two approaches for choosing activation function.

4.2 Testing accuracy

We are going to test accuracy of our neural network on MNIST database [6]. MNIST dataset consists of images of handwritten digits. Images are $28 \times 28$ greyscale images. There is 60000 training images and 10000 images for testing. We will train our neural networks on those 60000 images and then measure accuracy on 10000 testing images. Accuracy is then the percent of testing images the neural network classifies correctly. According to [7] the state of the art accuracy for this dataset is 99.77%.

4.3 Using $f(x) = x^2$ as activation function

In [1] it is proposed to use $x^2$ as activation function. For classifying MNIST dataset they use following neural network consisting of following layers:

- Convolutional layer. The input is of size $28 \times 28$. Filter has size $5 \times 5$. Stride is $(2, 2)$ and 5 filters is used. So the output of this layer is $5 \times 13 \times 13$.

- Activation layer. Using $f(x) = x^2$ as activation function.

- Sum pool layer. The layer uses $1 \times 3 \times 3$ windows and stride $(1, 1, 1)$. Size of the output is $5 \times 13 \times 13$.

- Convolutional Layer. Filter size is $1 \times 5 \times 5$. Stride is $(1, 2, 2)$ and 10 filters are used. Output of the layer is $50 \times 5 \times 5$. 

8
• Sum pool layer. Same as the first sum pool layer.

• Fully Connected Layer. This layer fully connects the incoming $50 \cdot 5 \cdot 5 = 1250$ neurons to the outgoing 100 neurons.

• Activation layer. Using $f(x) = x^2$ as activation function.

• Fully connected layer. This layer connects incoming 100 neurons to outgoing 10 neurons.

• Activation layer. Using sigmoid as activation function.

To this network we give $28 \times 28$ image as input and it returns 10 values. Each value corresponds to one digit. In training our goal is to train network so that if digit $a$ is in input image then activation of neuron corresponding to $a$ is 1 and activations of other neurons are 0. This means that later if we evaluate the network we classify it as the digit corresponding to the neuron with the highest activation value.

Notice that last layer uses sigmoid activation function, which can not be evaluated on encrypted data. But sigmoid activation function is actually strictly increasing. This means that if we remove this in evaluation step, the neuron with the highest value stays the same. So we do not need it in evaluation step. All the other layers are either linear or use square activation function. We have two layers using square activation function, so multiplicative depth of our network is two. This makes this network efficient enough.

In [1] it is said that the accuracy of this network is 98.95%, which is not too bad, but still relatively far from the state of the art accuracy 99.77%.

This approach however have some disadvantages. Derivative of $x^2$ is unbounded. This means that during training gradients can go really big, which can make training unstable.

4.4 Using approximation of ReLU as activation function

In [2] different approach to choosing activation function is proposed. Their idea is to train network using ReLU and then replace ReLU with low degree polynomial approximation of ReLU. This should solve the problem, which we had with $x^2$ activation function approach, where training could go unstable with deep networks.

There obviously is not any polynomial which approximates ReLU well on the entire $\mathbb{R}$. So in [2] they came up with an idea to put batch normalization layer before every activation layer. Batch normalization layer was introduced in [5]. The goal of the batch normalization layer is to ensure that outputs of this layer are always from same distribution. In many cases this makes training neural network easier.

Batch normalization layer can be used to ensure that the outputs of this layer are from standard normal distribution. This means that it is enough to find polynomial, which approximates ReLU well in small part around point 0. In [2] the approximate polynomials are found by minimizing square error between ReLU and polynomial, so that points are picked from standard normal distribution. In particular they take large enough $N$ and then $N$ points $x_1, x_2, \ldots, x_N$ from standard normal distribution. Then they find
$n$ degree polynomial $P$ such that

$$\sum_{i=1}^{N}(P(x_i) - \text{ReLU}(x_i))^2$$

is minimal. In Figure 4 resulting polynomials can be seen.

![Figure 4: Polynomial approximations of ReLU [2].](image)

It can be noticed that degree $2n + 1$ polynomials approximate similarly as $2n$ degree polynomials. So we have no point to use odd degree polynomials, because they increase multiplicative depth, but do not increase quality of approximation.

They first tested this approach on small neural network (see Figure 5). This network consisted of two convolutional layers each followed by average pool layer, then one fully connected layer followed by batch normalization layer, ReLU activation layer and finally one more fully connected layer.

Without replacing ReLU it achieved accuracy 97.95% on MNIST dataset. Replacing ReLU with either degree 2, degree 4 or degree 6 polynomial gave accuracy 97.55%, 97.84% or 97.91%. So with degree 6 polynomial the accuracy was already really close to the accuracy of non-private version of this network. Accuracy of non-private version of this network is however quite far from state of the art accuracy of 99.77%. So we need to use deeper network to get closer to the state of the art accuracy.

To get closer to the state of the art accuracy [2] introduces more complex neural network (see Figure 6). It consists of six convolutional layers. Each convolutional layer is followed by batch normalization and ReLU activation layer. After every second ReLU activation layer there is average pool layer. Finally after last average pool layer there is one fully connected layer followed by dropout layer and then one more fully connected layer. Convolutional layers use filter with size $(n \times 3 \times 3)$. Average pool layers use window
size \((2 \times 2)\). First fully connected layer has 256 neurons and last fully connected layer has 10 neurons.

Dropout layer is technique for reducing overfitting. During training, it removes every node with its connections with some fixed probability. Then remaining weights are adjusted and after that removed nodes are added back to the network with previous weights. This layer makes training easier, but does not affect neural network evaluation.

Without replacing ReLU this network achieved accuracy 99.59% on MNIST dataset. This is very close to the state of the art accuracy. However, replacing ReLU with either degree 2, degree 4 or degree 6 polynomial gave accuracy 59.14%, 97.91% or 36.94%. So it is quite far from the initial accuracy. This happens because we now have 6 ReLU layers which we replace with polynomial approximation. Small error in first activation layers causes the outputs of later batch normalization layers to not be from standard distribution anymore. This introduces even bigger errors in following activation layers.

In [2] the distribution of last batch normalization layer outputs were investigated and it came out that it had standard deviation slightly greater than 1. So they tried with different polynomial approximations. They got the best result using degree 4 polynomial,
which was found so that it approximates the best if inputs are from normal distribution with mean 0 and standard deviation 1.1. This way they got accuracy 98.18%. This is still quite far from the accuracy of non-private network.

To improve further they replaced the ReLU function with the degree 2 polynomial learned on a normal distribution with mean 0 and standard deviation 1.2. Then they did some further training with really small learning rate. It could not be done with degree 4 polynomial because training got unstable, because of the infinite derivative. This way they got accuracy of 99.28%, which is quite close to the accuracy of non-private network.

Finally they treated the coefficients of polynomial as training weigths and did some further training. This way the accuracy of 99.30% was achieved.

The multiplicative depth is 6 if degree 2 polynomial is used and 12 if degree 4 polynomial is used. So the efficiency is still reasonable.

In the end this approach got better accuracy than previous approach with $x^2$ activation function. But to achieve this accuracy some training on network with polynomial activation function had to be made. So it did not really remove the problem with the previous approach that with deeper networks the training becomes unstable. However for networks with really few activation layers, the accuracy of this approach is really close to the non-private accuracy.

5 Conclusions

In this report we studied the problem of evaluating neural network on encrypted data using fully homomorphic encryption. This requires that all operations must be additions or multiplications. The hardest part was to use such activation function. We studied two approaches. First approach used $f(x) = x^2$ as activation function. This approach achieved 98.95% accuracy on the MNIST database. Second approach trained neural network using ReLU activation function. To evaluate neural network ReLU was replaced with low degree polynomial approximation of ReLU. This way accuracy 99.30% was achieved. However both approaches have some problems, especially with deeper networks.

References


