DOI: 10.15514/ISPRAS-2019-1(2)-1

Design methods for privacy-preserving neural networks

[1] *Maria A. Lapina*, ORCID: 0000-0001-8117-9142 <*mlapina@ncfu.ru*>
[1]*Nikita Fisenko* ORCID 0009-0003-7313-9580< *mr.niksemen-05@yandex.ru* >
[1]*Egor Shiriaev, ORCID: 0000-0002-2359-1291 <ea_or@list.ru>*
[2]*S Neelakandan, ORCID: 0000-0001-8583-0019, <drsneelakandan@ieee.org >*

[1] *North Caucasus Federal University,*
1, Pushkina st., Stavropol, 355017, Russia
[2] *RMK Engineering College affiliated to Anna University,*
RSM Nagar, Gummidipoondi Taluk, Kavaraipettai, Tamil Nadu 601206, India

Abstract. The paper presents methods of designing neural networks focused on ensuring data privacy. The research conducted by a team of authors from the North Caucasus Federal University focuses on the integration of homomorphic encryption into the architecture of convolutional neural networks. The main goal of the research is to develop efficient data processing techniques that preserve privacy at all stages of learning and inference. The work considers the application of standard convolutional neural network architectures to image classification tasks, followed by the integration of homomorphic encryption using the TenSEAL library. Special attention is paid to the approximation of activation functions, which is a key aspect for compatibility with homomorphic encryption and data privacy. The results demonstrate the potential of adapting neural networks to handle encrypted data, highlighting the importance of further research to optimize the performance and security of the models. This work represents a significant contribution to the development of machine learning methods with privacy and data security in mind.

Keywords: neural networks, homomorphic encryption, data privacy, convolutional neural networks, approximation of activation functions.

For citation: Lapina Maria Fisenko Nikita, Shiriaev Egor, S Neelakandan Design methods for privacy-preserving neural networks. *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2019. pp. 15-19 (in Russian). DOI: 10.15514/ISPRAS-2019-1(2)-1

Acknowledgements. This work was supported by the Russian Science Foundation 19-71-10033, https://rscf.ru/project/19-71-10033/

1. Introduction

In recent years, the field of machine learning and artificial intelligence has seen a significant increase in interest in the development and application of neural networks, especially in tasks that require processing large amounts of data [1]. These advanced technologies find applications in various fields, including automated image analysis and the development of complex decision-making systems. However, as their use grows, the need to ensure the security and confidentiality of the processed information increases. This is especially relevant in the context of the tightening of legislation in the field of personal data protection in Russia in connection with the adoption of the Federal Law "On Personal Data" No. 152-FZ, aimed at strengthening control over the processing and dissemination of personal information of citizens [2]. In this context, one of the promising directions in the field of neural network design is the development of methods capable of ensuring data confidentiality at all stages of data processing [3]. To increase the level of information protection, the use of homomorphic encryption is proposed. This approach allows data to be processed in such a way that it remains encrypted throughout the learning process, thus providing a balance between extracting valuable information and preserving its confidentiality [4]. The basic principles and methodologies for designing neural networks that consider data privacy requirements, including techniques, advantages, and limitations, will be reviewed, and practical examples of their application will be discussed. The aim of this study is to provide a comprehensive overview of existing approaches and identify promising directions for further research in this area.

2. Privacy-preserving neural networks

2.1. General approaches

Homomorphic encryption is an encryption method that allows one to perform computations on encrypted data without requiring decryption [5]. The results of these computations remain in encrypted form and once decrypted, are identical to the results of operations on the data in plaintext [5]. This encryption approach can be used to provide privacy in outsourcing data storage and processing, allowing encrypted data to be transferred to commercial cloud environments for further processing, keeping it encrypted [6]. Homomorphic encryption is a sophisticated approach for processing encrypted data that allows various computational operations to be performed on it without prior decryption [7]. This method includes different encryption schemes, each of which has a unique ability to process the data while keeping it encrypted. The computational operations can be represented in the form of Boolean or arithmetic operations, which gives this technology flexibility in application. Depending on the level of support for different operations, there is a classification of homomorphic encryption schemes. First, note partially homomorphic encryption, which is limited to performing only one type of operation (e.g., only addition or only multiplication). Then we consider schemes that are restricted but still homomorphic to some extent: they are capable of handling operations of two types, but with constraints on the complexity of the computational circuits. Next, we consider level-complete homomorphic encryption, which allows arbitrary computational operations with limited depth. This ensures that more complex data processing algorithms can be realized. Finally, Fully Homomorphic Encryption (FHE) is considered, which provides the ability to perform an unlimited number of operations of different types on encrypted data. It is FHE that is the most powerful tool in the field of information security [8].

Within the framework of neural network development, the application of the CKKS (Cheon-Kim-Kim-Song) homomorphic encryption method, which belongs to the fourth generation of homomorphic ciphers, is due to several key advantages that make this approach particularly relevant for data processing in the context of machine learning [9]. The CKKS scheme, designed to work efficiently with real and complex numbers, provides a high degree of accuracy in computations over encrypted data, which is critical for achieving adequate results in deep learning tasks, given that the data and weights of the neural network are often real numbers [10]. One of the significant advantages of CKKS is its ability to process encrypted data using approximate computations, which can optimize performance and reduce the number of computational resources required to process complex neural network models while maintaining an acceptable level of accuracy [11]. In addition, CKKS can efficiently scale computational results through the inclusion of a scaling operation, which allows controlling the growth of errors during repeated arithmetic operations on encrypted data and provides the ability to perform complex computational procedures without significant loss of accuracy [12]. The use of CKKS in neural networks is also motivated by its high degree of security and its ability to ensure data confidentiality during processing, which is critical in applications that require the protection of sensitive information, such as medical or financial applications [13]. Thus, the choice of CKKS as a homomorphic encryption method for use in neural networks is motivated by its unique combination of accuracy, efficiency, and security, making it an ideal tool for solving machine learning problems where these aspects are key to successful implementation.

The research has considered the principle of convolutional neural network design focused on data privacy preservation. The research starts with local data processing using the MNIST dataset, which is downloaded and processed locally [13]. This reduces the risks associated with data leakage as the data is not transmitted across the network and is not stored on external servers. The study employs a standard convolutional neural network architecture for image processing, which provides training efficiency and sufficient accuracy for image classification tasks.

The study focuses on the integration of homomorphic encryption using the TenSEAL library [14]. This allows encrypted computation while providing a high level of data privacy protection [15]. An encrypted version of the neural network model is implemented, which is able to perform inference on encrypted data, allowing the model to make predictions without revealing the contents of the data [16].

The study also establishes encryption parameters, including global scale and polynomial modulus sizes, to control the balance between encryption accuracy and security [17]. In addition, the Galois key generation process provides the necessary infrastructure to perform operations on encrypted data [18].

The presented approach demonstrates the possibility of combining traditional machine learning techniques with advanced encryption technologies to create systems capable of processing data while preserving its confidentiality. This is especially relevant for areas where sensitive data such as medical images or financial information needs to be processed.

2.2. Secure matrix multiplication

Computation of homomorphic matrices is a fundamental operation for privacy-preserving machine learning. One of the many algorithms for multiplying homomorphic matrices, Halevy and Shoup's algorithm is based on a sequence of matrix-to-vector multiplications that are encoded as plaintext. (see the original, maybe add something from there in the form of water). In matrix-vector multiplication, the input matrices are encoded in their diagonal representation.

Let a matrix A of size $m \times m$ will be represented in the form $a_0, \dots, a_{m-1}$, where $a_i = (A_{0,i}, A_{1,i+1}, \dots, A_{m-1,m+1})$.

There are two paths to follow. The first is a vector of weights in unencrypted form, which will be computed using the algorithm described below. First, $a_i[j] = A_{j,j+1}$. Next product $w = vA$, where $v = \{v_0, v_1, \dots, v_{n-1}\}$ – input vector can be calculated as

$$v_0 = x_0\{a_0, a_1, \dots, a_{n-1}\}$$
$$v_1 = x_1\{a_{n-1}, a_0, \dots, a_{n-2}\}$$
$$\dots.$$
$$v_{n-1} = x_{n-1}\{a_1, a_2, \dots, a_{n-1}, a_0\}$$

This method requires m rotation, multiplication, and addition.

In case the vector of weights will be represented in encrypted form, the multiplication of two encrypted matrices is performed by the second way, the description of the algorithm of which is presented below:

The multiplication of a matrix by a vector can be represented by combining the operations of rotation and multiplication by a constant

Let a matrix A of size $m \times m$ will be presented in encrypted form виде $y_{-}, \dots, y_{m-1}$, where $a_i = (A_{0,i}, A_{1,i+1}, \dots, A_{m-1,m+1})$.

First and foremost, $y_i[j] = A_{j,j+1}$. Then there is a piecewise product between the vector of weights and the matrix, the product of the $w = vA$, where $v = \{v_0, v_1, \dots, v_{n-1}\}$ – input vector can be calculated as

$$v_0 = \{x_0, x_1, \dots, x_{n-1}\} \odot \{y_0, y_1, \dots, y_{n-1}\} = \{x_0 y_0, x_1 y_1, \dots, x_{n-1} y_{n-1}\}$$
$$v_1 = \{x_{n-1}, x_0, \dots, x_{n-2}\} \odot \{y_{n-1}, y_0, \dots, y_{n-2}\} = \{x_{n-1} y_{n-1}, x_0 y_0, \dots, x_{n-2} y_{n-2}\}$$
$$\dots.$$
$$v_{n-1} = \{x_1, x_2, \dots, x_{n-1}, x_0\} \odot \{y_1, y_2, \dots, y_{n-1}, y_0\} = \{x_1 y_1, x_2 y_2, \dots, x_{n-1} y_{n-1}, x_0 y_0\},$$

where $\odot$ - component product between vectors.

2.3. Approximation of activation functions and their realization in a homomorphic cipher

The research on the design of encrypted neural networks focuses on adapting activation functions to work with homomorphic encryption [19]. Homomorphic encryption, characterized by its ability to perform computations on encrypted data, is a powerful tool to ensure data privacy during data processing [20]. However, its application in the context of neural networks faces some difficulties, especially in the implementation of nonlinear activation functions, which are critical for the training efficiency of deep learning models [21].

In the current research, work has been done to adapt neural networks for processing encrypted data using homomorphic encryption. The basis for this was the TenSEAL library designed for homomorphic encryption, which allows to perform computations on encrypted data while ensuring its confidentiality. An important part of the research was the development of an encrypted version of the neural network represented by a class. This class copies the weights and biases from the trained model and uses them to create an encrypted model capable of handling encrypted data.

The sigmoid activation function, also known as the logistic sigmoid, is one of the most widely used activation functions in the field of neural networks, especially in the context of binary classification and logistic regression. This function takes a real number as input and compresses it between 0 and 1, making it ideal for tasks where it is necessary to predict a probability belonging to a certain class.

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

where x - input value, e – base of the natural logarithm

The sigmoidal activation function, due to its unique characteristics, plays a significant role in the field of machine learning, especially in the context of neural networks. One of the key features of the sigmoid function is its smoothness and differentiability [22]. These properties make it ideally suited for use in gradient descent and error back propagation algorithms, which ensures efficient updating of weights during neural network training. Transforming the input values by a sigmoid function into a range from 0 to 1 allows the outputs to be interpreted as probabilities. Thus, a value close to 1 indicates a high probability of belonging to a certain class, while a value close to 0 indicates a low probability. This feature makes the sigmoid function particularly useful in binary classification tasks. Introducing nonlinearity using the sigmoidal activation function allows neural networks to model complex and nonlinear relationships between input and output data. Without nonlinearity, neural networks would be limited to linear operations and would not be able to approximate a wide range of functions or solve problems requiring generalization of nonlinear relationships. However, the sigmoid function is not symmetric about the origin, which can lead to a bias in the gradients during the learning process. This bias can have an impact on the speed and stability of convergence of training, since the gradients propagated back through the network may not be optimally distributed, which in turn affects the process of updating the weights.

The key to adapting neural networks to handle homomorphic encryption is the approximation of activation functions. In this study, a quadratic activation function was used; being polynomial, it adapts more easily to the constraints of homomorphic encryption, unlike traditional activation functions such as ReLU or sigmoid [22]. This allowed the encrypted model to perform inference on encrypted data while retaining learning and classification ability [23]. Additionally, the study tuned the encryption parameters, including global scale and modulus sizes of the polynomials, which is a key element to customize the homomorphic encryption process [24].

The study confirms the feasibility of adapting neural networks to handle encrypted data using homomorphic encryption and activation function approximation [25]. However, further research is needed to optimize the balance between security, privacy, and model performance, considering the impact of activation function approximation on accuracy and overall model performance. However, the use of approximated activation functions entails certain trade-offs. On the one hand, it allows the model to handle encrypted data, which is a key requirement for privacy preservation. On the other hand, the approximation may result in lower model accuracy compared to using traditional activation functions. This is because the approximated functions may not fully reproduce the behavior of their nonlinear counterparts, which may have an impact on the learning process and the model's ability to generalize.

The study demonstrates how neural networks can be adapted to handle encrypted data using an approximation of activation functions for compatibility with homomorphic encryption. This opens new possibilities for developing secure and privacy-preserving machine learning systems, although further research is needed to optimize the balance between security, privacy, and model performance.

3 Modeling

3.1 Description of algorithms

The ConvNet class represents an essential element for experimenting and analyzing neural networks in the context of deep learning. This class serves an important purpose in defining and structuring neural network models, providing the necessary tools for exploring different architectures and learning algorithms.

First, the ConvNet class plays a key role in defining the model architecture. Here, the types of layers, their parameters, and the connections between them are defined, allowing researchers to create and adapt models to solve specific problems. It facilitates code organization and structuring. Putting all model components inside a class makes the code cleaner and more modular, which makes it easier to understand and maintain, especially in the case of complex models. In addition, the ConvNet class utilizes PyTorch functionality, allowing you to use a wide range of built-in methods and functions to work with the model. This includes optimization, loss computation, and other functions needed to effectively train and evaluate models.

The ConvNet class includes two functions - def __init__ and def forward. The detailed description of the algorithm of each function will be given below.

The __init__ function represents the constructor of the ConvNet class, which is a subclass of torch.nn.Module. It performs initialization of the neural network object, defining its structure and parameters. At the beginning of the function, the constructor of the parent class is called with super(ConvNet, self).__init__(), which allows the methods and attributes of the parent class to be used. Then the attributes of the neural network such as convolution layers (conv1) and full-link layers (fc1, fc2) are defined.

The related neural network objects torch.nn.Conv2d and torch.nn.Linear represent neural network layers that have different parameters such as the number of input and output channels, convolution kernel sizes, input and output dimensions, and others.

In this case, self.conv1 is a convolutional layer that takes as input images with one channel (since kernel_size=7), performs convolution with 4 filters and uses parameters padding=0 and stride=3, and self.fc1 and self.fc2 are full-connected layers that take as input vectors of dimension 256 (after leaving the convolutional layer) and output vectors of dimension hidden and output, respectively.

The __init__ function initializes the neural network structure, defining the layers and their parameters to be used in the training and prediction process.

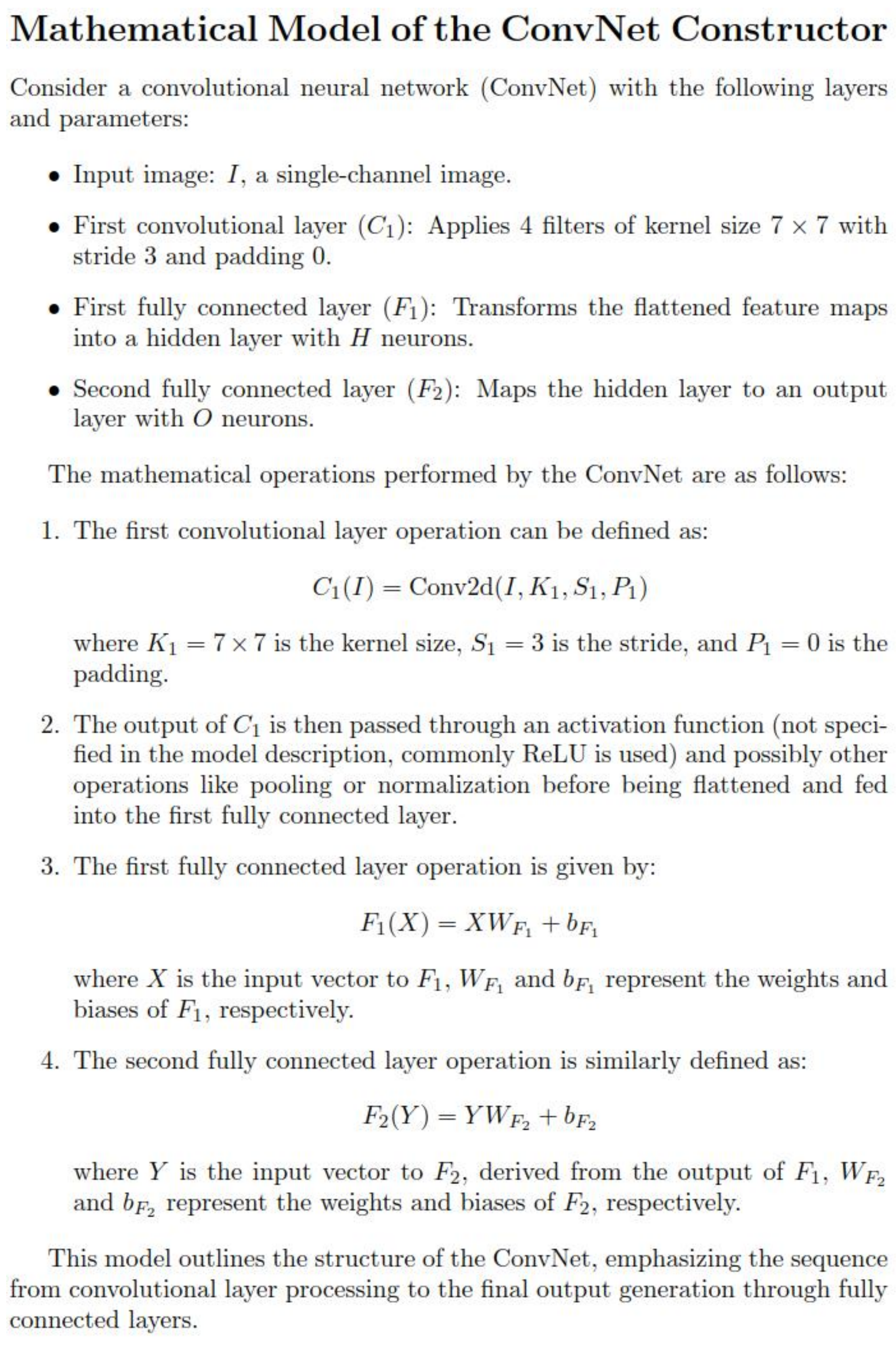
The forward function is to define the process of direct passage of data through the neural network. This function represents the main part of the neural network operation, in which input data is fed to the input of the network, passes through various layers and is returned as predictions or output values.

The input data x, representing images, is first passed through the convolution layer self.conv1, which performs convolution with known filters on the input data, creating a set of feature maps. This extracts important features from the images.

Next, the convolution result is subjected to a quadratic activation function, which in this context means a piecewise multiplication of the convolution result by itself. This can help to enhance the activation of certain features in the data.

The data is then transformed using the view method into a form compatible with the expected input dimensionality for the next fully connected layer self.fc1. This converts the multidimensional data into a one-dimensional vector that can be fed as input to the fully linked layer. The transformed data is passed through the full-link layer self.fc1, where linear combinations of the input data and their weighting factors are computed.

Like the previous step, the result of passing through self.fc1 is also subjected to a quadratic activation function. The data passes through the last full-link layer self.fc2, which converts the output from self.fc1 into expected predictions or task-specific output values.



Mathematical Model of the ConvNet Constructor

Consider a convolutional neural network (ConvNet) with the following layers and parameters:

- Input image: I, a single-channel image.
- First convolutional layer (C_1): Applies 4 filters of kernel size 7×7 with stride 3 and padding 0.
- First fully connected layer (F_1): Transforms the flattened feature maps into a hidden layer with H neurons.
- Second fully connected layer (F_2): Maps the hidden layer to an output layer with O neurons.

The mathematical operations performed by the ConvNet are as follows:

1. The first convolutional layer operation can be defined as:

$$C_1(I) = \text{Conv2d}(I, K_1, S_1, P_1)$$

   where $K_1 = 7 \times 7$ is the kernel size, $S_1 = 3$ is the stride, and $P_1 = 0$ is the padding.

2. The output of C_1 is then passed through an activation function (not specified in the model description, commonly ReLU is used) and possibly other operations like pooling or normalization before being flattened and fed into the first fully connected layer.

3. The first fully connected layer operation is given by:

$$F_1(X) = XW_{F_1} + b_{F_1}$$

   where X is the input vector to F_1, W_{F_1} and b_{F_1} represent the weights and biases of F_1, respectively.

4. The second fully connected layer operation is similarly defined as:

$$F_2(Y) = YW_{F_2} + b_{F_2}$$

   where Y is the input vector to F_2, derived from the output of F_1, W_{F_2} and b_{F_2} represent the weights and biases of F_2, respectively.

This model outlines the structure of the ConvNet, emphasizing the sequence from convolutional layer processing to the final output generation through fully connected layers.

Algorithm 1: Forward Pass Function

```
Input  : An input tensor $x \in \mathbb{R}^{n \times c \times h \times w}$ representing a batch of
         images.
Output: The output tensor $x$ after processing through the neural
         network.
// Apply the first convolutional layer.
$x \leftarrow conv1(x)$
// Apply quadratic activation function.
$x \leftarrow x \times x$
// Reshape the tensor for the fully connected layer.
$x \leftarrow x.view(-1, 256)$
// Apply the first fully connected layer.
$x \leftarrow fc1(x)$
// Apply quadratic activation function again.
$x \leftarrow x \times x$
// Apply the second fully connected layer.
$x \leftarrow fc2(x)$
return $x$
```

The train function represents the process of training a neural network on training data. During training, the model sequentially goes through several steps, including passing input data through the neural network, calculating losses, updating model parameters based on the calculated gradients and repeating this process for each training epoch. At the end of each epoch, the current loss on the training data is output. When training is complete, the function returns the trained model for later use.

Algorithm 2: Training Process of a Neural Network

```
Input  : A model $M$, training data loader $D$, loss function $L$,
         optimizer $O$, number of epochs $E$.
Output: Trained model $M$.
$M.train()$
for $e = 1$ to $E$
    $L_{total} \leftarrow 0$
    for each batch $(x_b, y_b)$ in $D$
        $O.zero\_grad()$
        $output_b \leftarrow M(x_b)$
        $loss \leftarrow L(output_b, y_b)$
        $loss.backward()$
        $O.step()$
        $L_{total} \leftarrow L_{total} + loss.item()$
    $L_{epoch} \leftarrow L_{total}/\text{len}(D)$
    print "Epoch: ", $e$, "Loss: ", $L_{epoch}$
$M.eval()$;
```

The test function is intended for testing the trained model on a test data set. First, it puts the model into eval mode to ensure that no learning occurs during testing. It then traverses through each test data loader batches, passing the data through the model and calculating the loss using the specified loss function. As the data passes through, it also counts the correct answers for each class and the total number of correct answers for the entire test dataset. At the end of the test, it outputs the average test loss and prediction accuracy for each class, as well as the overall test accuracy.

```
Algorithm 3: Model Testing and Evaluation
Input  : A model $M$, test data loader $D_{test}$, loss function $L$.
Output: Test loss and accuracy for each class and overall.

$M.eval()$
$L_{total} \leftarrow 0$
$class_{correct} \leftarrow$ list of zeros with length equal to number of classes
$class_{total} \leftarrow$ list of zeros with length equal to number of classes

for each batch $(x_j, y_j)$ in $D_{test}$ do
    $output \leftarrow M(x_j)$
    $loss \leftarrow L(output, y_j)$
    $L_{total} \leftarrow L_{total} + loss.item()$
    // Prediction and accuracy calculation
    $pred \leftarrow \arg\max(output)$
    $correct \leftarrow pred == y_j$
    for $i \leftarrow 0$ to $length(y_j)$ - 1 do
        $class_{correct}[y_j[i]] \leftarrow class_{correct}[y_j[i]] + correct[i]$
        $class_{total}[y_j[i]] \leftarrow class_{total}[y_j[i]] + 1$
    end
end
$L_{mean} \leftarrow \frac{L_{total}}{length(D_{test})}$
print "Test Loss: ", $L_{mean}$

for label in 0 to number of classes - 1 do
    $accuracy \leftarrow \frac{class_{correct}[label]}{class_{total}[label]} \times 100$
    print "Test Accuracy of ", label, ": ", $accuracy$, "%"
end
$overall_{accuracy} \leftarrow \frac{\sum(class_{correct})}{\sum(class_{total})} \times 100$
print "Test Accuracy (Overall): ", $overall_{accuracy}$, "%"
```

The EncConvNet class is an encrypted version of the ConvNet neural network designed to work with encrypted data. Inside the class are stored parameters of the original ConvNet model, such as weights and offsets of convolutional and full-link layers copied from the trained model. These parameters are stored in specially organized class variables for later use when passing data directly through the encrypted neural network. It is a tool for working with encrypted data using neural networks, providing the ability to perform convolution and direct data transformation operations using the stored parameters.

The __init__ function is intended for initialization of the class object. It takes an instance of the torch_nn model as input, assuming it is a trained ConvNet model. The function then copies the weights and offsets from the convolutional and full-connection layers of this model, converting them to a Python list. Each convolution layer weight is represented as a three-dimensional array, and each offset is represented as a one-dimensional array. The weights of the fully connected layers are represented as two-dimensional arrays, and the offsets are represented as one-dimensional arrays.

```
Algorithm 4: Initialization of the Model with Pre-trained ConvNet
Weights
Input: A pre-trained ConvNet model $M_{torch}$ with layers $conv1$, $fc1$,
       and $fc2$.
Output: A new model structure $M_{new}$ initialized with weights and
        biases from $M_{torch}$.

// Copying weights and biases from the first convolutional
   layer.
$M_{new}.conv1_{weight} \leftarrow$
  $M_{torch}.conv1.weight.data.view(M_{torch}.conv1.out\_channels, M_{torch}.conv1.kernel\_size[0], M_{torch}.conv1$

$M_{new}.conv1_{bias} \leftarrow M_{torch}.conv1.bias.data.tolist()$

// Copying weights and biases from the first fully connected
   layer.
$M_{new}.fc1_{weight} \leftarrow M_{torch}.fc1.weight^T.data.tolist()$
$M_{new}.fc1_{bias} \leftarrow M_{torch}.fc1.bias.data.tolist()$

// Copying weights and biases from the second fully
   connected layer.
$M_{new}.fc2_{weight} \leftarrow M_{torch}.fc2.weight^T.data.tolist()$
$M_{new}.fc2_{bias} \leftarrow M_{torch}.fc2.bias.data.tolist()$
```

The forward method in the EncConvNet class represents the direct passage of data through an encrypted neural network. First, the encrypted input data passes through the convolution layer, where a convolution transform is applied to each data window using the kernels and offsets stored in the class object. The resulting convolution outputs are then combined into a channel list. Next, a quadratic activation function is applied to the resulting encrypted channels, after which the data is fed to the full-link layers fc1 and fc2. For each layer, the encrypted data is multiplied by the corresponding weights stored in the class object and the corresponding offsets are added. After each full-link layer, a quadratic activation function is also applied.

```
Algorithm 5: Training Process of a Neural Network
Input  : A model $M$, training data loader $D$, loss function $L$,
         optimizer $O$, number of epochs $E$.
Output: Trained model $M$.

$M.train()$
for $e = 1$ to $E$
    $L_{total} \leftarrow 0$
    for each batch $(x_b, y_b)$ in $D$
        $O.zero\_grad()$
        $output_b \leftarrow M(x_b)$
        $loss \leftarrow L(output_b, y_b)$
        $loss.backward()$
        $O.step()$
        $L_{total} \leftarrow L_{total} + loss.item()$
    $L_{epoch} \leftarrow L_{total}/\text{len}(D)$
    print "Epoch: ", $e$, "Loss: ", $L_{epoch}$
$M.eval()$
```

The enc_test function is designed to evaluate the performance of the encrypted model on a test dataset. The initial part of the function execution includes initialization of variables to store the total loss and statistics on correctly classified objects for each category. During the model testing process, the following algorithm is executed on each data batch from the test loader, which is described below

```
Algorithm 6: Testing an Encrypted Neural Network Model
Input: Encryption context $C$, encrypted model $M_{enc}$, test data loader
       $D_{test}$, loss function $L$, kernel shape $K_{shape}$, stride $S$.
Output: Test loss and accuracy metrics for each class and overall.

Initialize total test loss $L_{total} \leftarrow 0$
Initialize correct and total counters for each class, $class_{correct} \leftarrow$ list of
 zeros(10), $class_{total} \leftarrow$ list of zeros(10)

foreach $(data, target)$ in $D_{test}$ do
    // Prepare data: encrypt input data
    $x_{enc}, windows_{nb} \leftarrow$
     $ts.im2col\_encoding(C, data.view(28, 28).tolist(), K_{shape}[0], K_{shape}[1], S)$

    // Encrypted prediction
    $enc\_output \leftarrow M_{enc}(x_{enc}, windows_{nb})$
    // Decrypting the result
    $output \leftarrow enc\_output.decrypt()$
    $output \leftarrow torch.tensor(output).view(1, -1)$
    // Loss computation
    $loss \leftarrow L(output, target)$
    $L_{total} += loss.item()$
    // Convert output probabilities to predicted class
    $,pred \leftarrow torch.max(output, 1)$
    // Compare predictions with the true label
    $correct \leftarrow np.squeeze(pred.eq(target.data.view\_as(pred)))$
    // Update test accuracy for each class
    $label \leftarrow target.data[0]$
    $class_{correct}[label] += correct.item()$
    $class_{total}[label] += 1$
end

// Calculate and print the average loss on the test set
$L_{mean} \leftarrow L_{total}/\sum(class_{total})$
print("Test Loss: ", $L_{mean}$)

for $label \leftarrow 0$ to $9$ do
    print("Test Accuracy of ", $label$, ": ",
     $100 \times class_{correct}[label]/class_{total}[label]$, "% (", $class_{correct}[label]$,
     "/", $class_{total}[label]$, ")")
end

print("Test Accuracy (Overall): ", $100 \times \sum(class_{correct})/\sum(class_{total})$,
 "% (", $\sum(class_{correct})$, "/", $\sum(class_{total})$, ")")
```

3.2 Results of the experimental study

As part of the research, experiments were conducted to train and test the neural network, as well as its encrypted version, using the MNIST dataset. The aim of the experiment was to evaluate the performance of the model in normal and encrypted modes, and to study the effect of homomorphic encryption on the performance and accuracy of the model. The hardware configuration consists of an Intel(R) Xeon(R) CPU E5-2696 v3 CPU clocked at 2.30 GHz, 32 GB of DDR4 RAM at 2133 MHz, and a 1 TB SSD. The average time was measured by running the algorithms 100 times on the platform. During the experiment process, data was collected on training and testing losses as well as classification accuracy for each class. Additionally, the CPU utilization during training was measured.

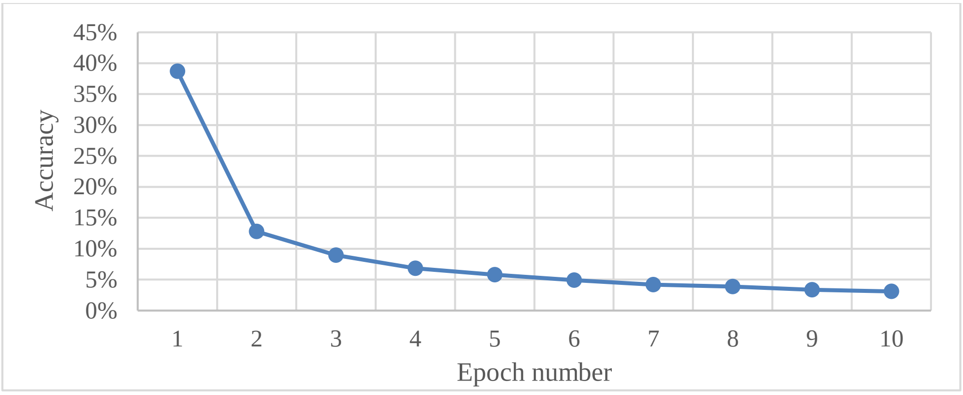


Figure 1: Learning Losses graph

Figure 1 shows the dynamics of neural network losses during training over 10 epochs. The X axis represents the numbers of epochs (from 1 to 10), and the Y axis represents the number of losses. The graph shows a decrease in loss with each subsequent epoch, indicating an improvement in the model's ability to learn and adapt to the data.

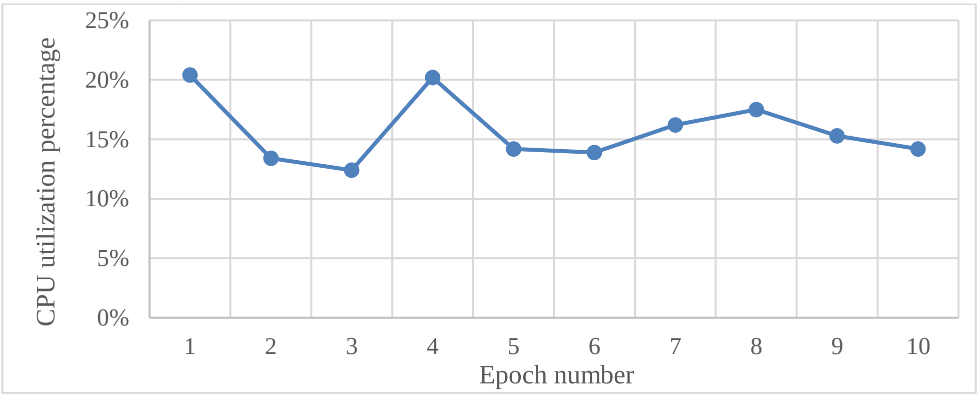


Figure 2: Graph "CPU utilization during training"

Figure 2 illustrates the percentage of CPU utilization during each epoch of model training. The X axis represents the epoch numbers, and the Y axis represents the percentage of CPU utilization. The graph can show changes in CPU utilization depending on the complexity of the computation at each epoch.

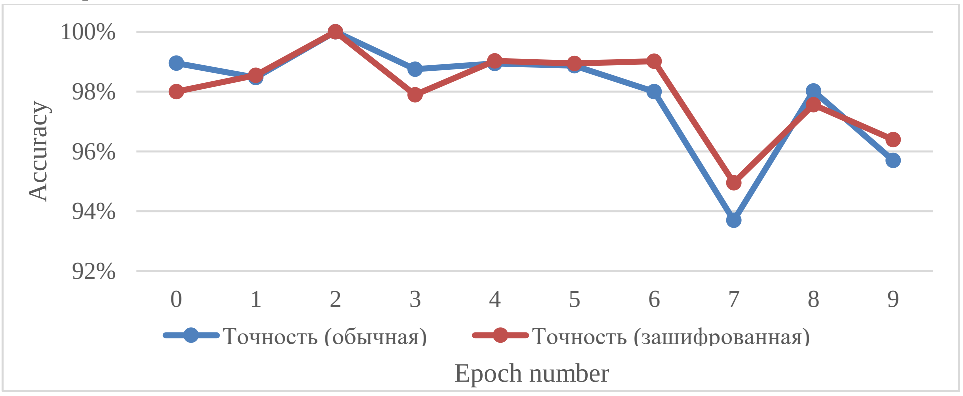


Figure 3: "Test Accuracy" graph

Figure 3 shows the accuracy of the model classification on the test data for each of the 10 classes, as well as the overall accuracy. The X-axis represents the classes (0 through 9) and the Y-axis represents the percentage of accuracy for each class. The graph helps visualize how the model performs in classifying different categories, identifying the classes on which the model performs better or worse.

The experimental results show that the neural network exhibits high accuracy in both the normal and encrypted modes, with a slight increase in overall accuracy in the encrypted mode. This indicates that the application of homomorphic encryption does not have a significant negative impact on the model's classification ability. However, an increase in training loss in the encrypted mode is observed, which may indicate the need for additional optimization of the model parameters to handle encrypted data. The processor load during training remained within normal limits, which confirms the effectiveness of using this neural network architecture for image classification tasks.

4. Conclusion

The result of a study on adapting neural networks to work on encrypted data using homomorphic encryption is significant, highlighting the potential and limitations of this approach in the field of machine learning. The study demonstrates that by approximating activation functions and using homomorphic encryption, neural networks can be successfully trained and tested on encrypted data while maintaining a high level of privacy.

The results of a study on adapting neural networks to handle encrypted data when applying homomorphic encryption present significant findings that highlight the potential and limitations of this approach in the context of machine learning. The analysis demonstrates that by approximating activation functions and using homomorphic encryption, neural networks can be successfully trained and tested on encrypted data while maintaining a high level of privacy.

Examining the training and testing results of the neural network, we conclude that the model exhibits improved performance with each epoch, as evidenced by the reduction of loss during training. This indicates that the neural network successfully adapts to the data and learns efficiently. In addition, the observed CPU load during training remains within normal limits, indicating the effectiveness of using this neural network architecture for image classification tasks.

The results of testing the model on test data demonstrate high classification accuracy for each of the classes, as well as overall accuracy, which confirms the effectiveness of the model in classification tasks. It is interesting to note that the encrypted version of the model shows comparable, and in some cases even higher accuracy, indicating that the use of homomorphic encryption does not negatively affect the model's classification ability.

It should be noted that the use of approximated activation functions and homomorphic encryption requires further research to optimize the balance between security, privacy, and model performance. It is important to study the impact of different types of activation function approximation on the accuracy and overall performance of the model, and to develop methods to improve the performance of encrypted neural networks.

The results of this study open new perspectives for the development of secure and privacy-preserving machine learning systems, especially in areas where sensitive data processing is required. They also emphasize the importance of continued research in this area to achieve the optimal combination of security, privacy, and efficiency in neural networks.

References

[1] Azraoui, M., Barham, M., Bozdemir, B., et al. (2019). SoK: Cryptography for Neural Networks.
[2] Madi, A., Sirdey, R., Stan, O. (2020). Computing Neural Networks with Homomorphic Encryption and Verifiable Computing.
[3] Shi, J., Zhao, X. (2023). Anti-leakage method of network sensitive information data based on homomorphic encryption
[4] Yeow, S.-Q., Ng, K.-W. (2023). Neural Network Based Data Encryption: A Comparison Study among DES, AES, and HE Techniques.
[5] V. Subramaniyaswamy, V. Jagadeeswari, V. Indragandhi, R. Jhaveri, V. Vijayakumar, K. Kotecha, Logesh Ravi, "Somewhat Homomorphic Encryption: Ring Learning with Error Algorithm for Faster Encryption of IoT Sensor Signal-Based Edge Devices", 2022.
[6] Alessandro Falcetta, M. Roveri, "Privacy-Preserving Deep Learning With Homomorphic Encryption: An Introduction", 2022.
[7] Craig Gentry, "A fully homomorphic encryption scheme", .
[9] Seungjae Chae, Joon-Woo Lee, Yongwoo Lee, Jong-Seon No, "Ciphertext Refresh Using Communication Cost on CKKS Fully Homomorphic Encryption", 2023.
[10] Sajjad Akherati, Xinmiao Zhang, "Low-Complexity Ciphertext Multiplication for CKKS Homomorphic Encryption", .
[11] Samanvaya Panda, "Principal Component Analysis Using CKKS Homomorphic Scheme", .
[12] Infall Syafalni, Daniel M. Reynaldi, R. Munir, T. Adiono, N. Sutisna, Rahmat Mulyawan, "Complexity Analysis of Encoding in CKKS-Fully Homomorphic Encryption Algorithm", 2022.
[13] T. Wingarz, M. Gomez-Barrero, C. Busch, M. Fischer, "Privacy-Preserving Convolutional Neural Networks Using Homomorphic Encryption", 2022.
[14] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, Alaa Eddine Belfedhal, "TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption", 2021.
[15] Yancho B. Wiryen, Noumsi Woguia Auguste Vigny, Mvogo Ngono Joseph, Fono Louis Aimé, "A Comparative Study of BFV and CKKs Schemes to Secure IoT Data Using TenSeal and Pyfhel Homomorphic Encryption Libraries", 2023.
[16] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, Roman Walch, "Pasta: A Case for Hybrid Homomorphic Encryption", 2023.
[17] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, D. Cousins, Saroja Erabelli, N. Genise, S. Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, D. Micciancio, Ian Quah, Y. Polyakov, Saraswathy R.V., K. Rohloff, Jonathan Saylor, Dmitriy Suponitsky, M. Triplett, V. Vaikuntanathan, Vincent Zucca, "OpenFHE: Open-Source Fully Homomorphic Encryption Library", 2022.
[19] Hongquan Li, Yunfei Cao, "Study on linear optimization of activation function of homomorphic encryption neural network", 2021.
[20] Srinath Obla, Xinghan Gong, Asma Aloufi, Peizhao Hu, Daniel Takabi, "Effective Activation Functions for Homomorphic Evaluation of Deep Neural Networks", 2020.
[21] Kohei Yagyu, Ren Takeuchi, M. Nishigaki, Tetsushi Ohki, "Improving Classification Accuracy by Optimizing Activation Function for Convolutional Neural Network on Homomorphic Encryption", .
[22] F. Temurtas, Ali Gülbağ, N. Yumusak, "A Study on Neural Networks Using Taylor Series Expansion of Sigmoid Activation Function", 2004.
[23] Srinath Obla, Xinghan Gong, Asma Aloufi, Peizhao Hu, Daniel Takabi, "Effective Activation Functions for Homomorphic Evaluation of Deep Neural Networks", 2020.
[24] Hongquan Li, Yunfei Cao, "Study on linear optimization of activation function of homomorphic encryption neural network", 2021.
[25] Kohei Yagyu, Ren Takeuchi, M. Nishigaki, Tetsushi Ohki, "Improving Classification Accuracy by Optimizing Activation Function for Convolutional Neural Network on Homomorphic Encryption", .
[26] Ilsang Ohn, Yongdai Kim, "Smooth Function Approximation by Deep Neural Networks with General Activation Functions", 2019.

Information about authors

Maria A. LAPINA – Candidate of Physical and Mathematical Sciences, Associate Professor of the Department of Information Security of Automated Systems of the North Caucasus Federal University.

Nikita FISENKO – student of the Department of Information Security of Automated Systems of the North Caucasus Federal University.

Egor SHIRIAEV is a graduate student of the North Caucasus Federal University of the Faculty of Mathematics and Computer Sciences Named after Prof. Nikolay Chervyakov. Research interests: modular arithmetic, neurocomputer technologies, cryptographic methods of information protection.

S NEELAKANDAN is PhD, Associate Professor of the Department Computer Science and Engineering of the R.M.K

Engineering College affiliated to Anna University, Kavaraipettai, India