Fast Homomorphic Evaluation of Deep Discretized Neural Networks

Florian Bourse[1,2*], Michele Minelli[1,2†], Matthias Minihold[3†], and Pascal Paillier[4]

[1] DIENS, École normale supérieure, CNRS, PSL Research University, 75005 Paris, France
[2] INRIA
[3] Horst Görtz Institut für IT-Security, Ruhr-Universität Bochum, Germany
[4] CryptoExperts, Paris, France

Abstract. The rise of machine learning – and most particularly the one of deep neural networks – multiplies scenarios where one faces a privacy dilemma: either sensitive user data must be revealed to the entity that evaluates the cognitive model (*e.g.* in the Cloud), or the model itself must be revealed to the user so that the evaluation can take place locally. The use of homomorphic encryption promises to reconcile these conflicting interests in the Cloud-based scenario: the computation is performed remotely and homomorphically on an encrypted input and the user decrypts the returned result. A typical task is that of classifying input patterns and a number of works have already attempted to implement homomorphic classification based on Somewhat Homomorphic Encryption. The resulting running times are not only disappointing but also quickly degrade with the number of layers in the network. Surely, this approach is not adapted to deep neural networks, that are composed of tens or possibly hundreds of layers.

This paper achieves unprecedentedly fast, scale-invariant homomorphic evaluation of neural networks. Scale-invariance here means that the computation carried out by every neuron in the network is independent of the total number of neurons and layers, thus opening the way to privacy-preserving applications of deep neural networks. We refine the recent R/LWE-based Torus- FHE construction by Chillotti *et al.* (ASIACRYPT 2016) and make use of its efficient bootstrapping to refresh ciphertexts propagated throughout the network. For our techniques to be applicable, we require the neural network to be *discretized*, meaning that signals are binarized values in $\{-1, 1\}$, weights are signed integers in a prescribed interval and neurons are activated by the sign function. We show how to train such networks using a specifically designed backpropagation algorithm. We report experimental results on the MNIST dataset and show-case a discretized neural network that recognizes handwritten numbers with over 92% accuracy and is evaluated homomorphically in just about 0.88 seconds on a single core of an average-level laptop. We believe that this work can help bridge the gap between machine learning's capabilities and its practical, efficient and privacy-preserving implementation.

Keywords: Fully Homomorphic Encryption, Neural Networks, Bootstrapping, MNIST.

* Supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013 Grant Agreement no. 339563 – CryptoCloud)

† This work was supported by European Union's Horizon 2020 research and innovation programme under grant agreement No H2020-MSCA-ITN-2014-643161 ECRYPT-NET. This work was done while the author was visiting CryptoExperts.

1 Introduction

Fully Homomorphic Encryption (FHE). An FHE scheme provides a way to encrypt data while supporting computations through the encryption envelope. Given an encryption of a plaintext x, one can compute an encryption of $f(x)$ for any computable function f. This operation does not require intermediate decryption or knowledge of the decryption key and therefore can be performed based on public information only. Applications of FHE are numerous but one particular use of interest is the privacy-preserving delegation of computations to a remote service. The first construction of FHE dates back to 2009 and is due to Gentry [Gen09]. A number of improvements have followed [vDGHV10, SS10, SV10, BV11a, BV11b, BGV12, GHS12, GSW13, BV14, AP14], leading to a biodiversity of techniques, features and complexity assumptions.

The quest for privacy-preserving machine learning. Machine Learning As a Service (MLAS) is becoming popular because of its versatility. These applications typically have high computation and data-storage requirements, which make them less suitable as client-side technologies. Moreover, since the process of training a cognitive model is time and resource-consuming, the trained prediction algorithm is often considered critical intellectual property by its owner, who is typically not willing to share its technology or proprietary tools, resulting in that machine learning algorithms are most conveniently cloud-based.
However, this setting raises new issues concerning the privacy of the uploaded input data. Users want to send their encrypted data to a cloud service that offers privacy-preserving predictions, and fulfills this task using its powerful yet undisclosed, state-of-the-art predictive models. In this paper, we put forward a new and powerful FHE framework that makes it efficient for the cloud to operate a neural network dedicated to some specific machine learning task. The network, previously trained on plaintext dataset, does not have access to the input data in the clear, but is only given user-provided encrypted inputs and returns encrypted predictions.
Obviously, encrypting the user's data ensures its confidentiality, since the private key under which the data is encrypted is assumed never to leave the owner's controlled domain. In this setting, only the legitimate owner of the secret key can decrypt the result returned by the delegated computation that has been homomorphically performed in the cloud. The cloud service only learns superficial information, but can still charge the user for using the service.
Neural networks (NNs) are often built from medical, financial or otherwise sensitive data. They are usually trained to solve a *classification* problem: all possible observations are categorized into classes and, given a training dataset of observation/class pairs, the network should be able to assign the correct class to new observations. Such framework can be easily applied to problems like establishing a diagnosis from medical observations.
In this work we do not consider the problem of privacy-preserving data-mining, intended as training a neural network over encrypted data and that can be addressed, e.g., with the approach of [AS00]. Instead, we assume that the neural network is trained with data in the clear and we focus on the evaluation part. Another potential concern for the service provider is that users might be sending malicious requests in order to either learn what is considered a company secret (the neural network itself), or specific sensitive information encoded in the weights (which could be a breach into the privacy of the training dataset). In this latter case a statistical database can be used in the training phase, as is discussed in the differential privacy literature [Dwo06] .

Prior works. The Cryptonets paper [DGBL+16] was the first initiative to address the challenge of achieving homomorphic classification. The main idea consists in applying a leveled Somewhat Homomorphic Encryption (SHE) scheme such as BGV [BGV12] to the network inputs and propagating the signals across the network homomorphically, thereby consuming levels of homomorphic evaluation whenever non-linearities are met. In NNs, non-linearities come from activation functions which are usually picked from a small set of non-linear functions of reference (logistic sigmoid, hyperbolic tangent, etc) chosen for their mathematical convenience. To optimally accommodate the underlying SHE scheme, Cryptonets replace their standard activation by the (depth 1) square function, which only consumes one level. A

number of subsequent works have followed the same approach and improved it, typically by adopting higher degree polynomials as activation functions for more training stability [ZYC16], or by renormalizing weighted sums prior to applying the approximate function so that its degree can be kept as low as possible [CdWM+17]. Practical experiments have shown that training can accommodate approximated activations and generate NNs with very good accuracy.
However, this approach suffers from an inherent limitation: the homomorphic computation, local to a single neuron, depends on the total number of levels required to implement the network, which is itself roughly proportional to the number of its activated layers. Therefore the overall performance of the homomorphic classification heavily depends on the total multiplicative depth of the circuit and rapidly becomes prohibitive as the number of layers increases. This approach does not "scale well" and is not adapted to deep learning, where neural networks can contain tens, hundreds or sometimes thousands of layers [HZRS15, ZK16].

Our contributions. We adopt a scale-invariant approach to the problem. In our framework, each neuron's output is refreshed through bootstrapping, resulting in that arbitrarily deep networks can be homomorphically evaluated. Of course, the entire homomorphic evaluation of the network will take time proportional to the number of its neurons or, if parallelism is involved, to the number of its layers. But operating one neuron is now essentially independent of the dimensions of the network: it just relies on system-wide parameters.
In order to optimize the overall efficiency of the evaluation, we adapt to our use case the bootstrapping procedure of the recent construction by Chillotti *et al.*, known as Torus-FHE [CGGI16b]. TFHE provides a toolkit of schemes and operations based on variants of Ring-LWE and GSW [LPR10, GSW13] that handle real values on the Torus $[0, 1)$ instead of modular integers. A single neuron computation is made of a homomorphic multi-addition followed by a fast bootstrapping procedure which, on top of regularizing the noise level of the ciphertext, also applies the activation function for free.
In our framework, unlike in standard neural networks, the neuron inputs and outputs, the weights and biases, as well as the domain and range of the activation function cannot be real-valued and must be discretized. We found it optimal to represent signals as ± 1 values, weights and biases as signed integers and to use the sign function as activation function. We call such networks *Discretized Neural Networks* or DiNNs. This particular form of neural networks is somehow inspired by a more restrictive one, referred to in the literature as *Binarized Neural Networks* (BNNs) [CB16] where signals and weights are restricted to the set $\{-1, 1\}$ instead of $\{-W, \ldots, W\}$ in the case of DiNNs. Interestingly, it has been empirically observed by [CB16] that BNNs can achieve accuracies close to the ones obtained with state-of-the-art classical NNs, at the price of an overhead in the total network size, which is largely compensated by the obtained performance gains. Since our DiNNs are an extension of BNNs and stand in between BNNs and classical NNs, we expect the overhead in network size to be somewhat smaller.
The learning phase of DiNNs differs in several ways from the one of classical NNs because we need to compensate for the fact that using discrete values makes performing some calculations impossible. We suggest a dedicated backpropagation procedure that performs a gradient descent to learn the weights and biases of a trained DiNN. Although we did not investigate the learning phase much further, we suspect that, as for general NNs, it can be optimized and customized in many different ways.
Finally, we conducted experiments[5] by applying our generic DiNN framework to the MNIST dataset [LBBH98]. Since our backpropagation algorithm was implemented from scratch and purely CPU-based (contrarily to GPU-based software tools that developers may use to train classical NNs), building our DiNN took a significant amount of computational effort. Therefore, to ease our training phase, we cropped and simplified the format of the MNIST images, resulting in a *de facto* loss of accuracy. In the end, we built a DiNN with a single hidden layer of 30 neurons and featuring an accuracy of 93.17%. Although much better accuracies could certainly be obtained through various optimizations, this was not the goal of this work. Our aim was conducting experiments to measure the accuracy of the homomorphic classification and comparing it to that in the clear. We found that, for a security level of roughly 80 bits, our implementation takes about 0.88 seconds per classification (with no underlying parallelism whatsoever) and achieves 92.94% accuracy when evaluated homomorphically.

[5] Our code is available online at https://github.com/mminelli/dinn

Comparison with Cryptonets. The NN used by [DGBL+16] achieves an accuracy of 98.95 % when evaluated on the MNIST dataset. Propagated signals are reals properly encoded into compatible plaintexts and a single encrypted input (i.e., an image pixel) takes $382 \cdot 8192$ bits. The complete homomorphic evaluation of the network takes 570 seconds. In our case, a loss of accuracy occurs due to the preliminary simplification of the MNIST images, and also from the binarization of the network. We stress however that our prime goal was not accuracy but to achieve a qualitatively better homomorphic evaluation at the neuron level. In comparison, each of our input ciphertext is 162.7 times smaller and our overall evaluation is 647 times faster.We also achieve scale-invariance, meaning that we can keep on computing over the encrypted outputs of our network, whereas Cryptonets are bounded by the initial choice of parameters.

Outline of the paper. The paper is organized as follows: in section 2 we define our notation and we introduce notions about fully homomorphic encryption, its formulation over the torus, and artificial neural networks; in section 3 we present our Discretized Neural Networks and show how to train and evaluate them on data in the clear; in section 4 we move to the homomorphic evaluation of a DiNN and present how we build a model that satisfies our needs; in section 5 we give experimental results on data in the clear and on encrypted inputs, draw some conclusions and identify several open directions for future research.

2 Preliminaries

In this section we clarify our notation and recall some definitions and constructions that are going to be useful in the rest of the paper.

2.1 Notation

We denote the real numbers by $\mathbb{R}$, the integers by $\mathbb{Z}$ and use $\mathbb{T}$ to indicate $\mathbb{R}/\mathbb{Z}$, i.e., the torus of real numbers modulo 1. We use $\mathbb{B}$ to denote the set $\{0,1\}$, and we use $\mathcal{R}[X]$ for polynomials in the variable X with coefficients in $\mathcal{R}$, for any ring $\mathcal{R}$. We use $\mathbb{R}_N[X]$ to denote $\mathbb{R}[X]/\left(X^N+1\right)$ and $\mathbb{Z}_N[X]$ to denote $\mathbb{Z}[X]/\left(X^N+1\right)$ and we write their quotient as $\mathbb{T}_N[X] = \mathbb{R}_N[X]/\mathbb{Z}_N[X]$, i.e., the ring of polynomials in X quotiented by $\left(X^N+1\right)$, with real coefficients modulo 1. Vectors are denoted by lower-case bold letters, while matrices are indicated by upper-case bold letters. We use $\|\cdot\|_1$ and $\|\cdot\|_2$ to denote the L_1 and the L_2 norm of a vector, respectively. We use $\langle \mathbf{a}, \mathbf{b} \rangle$ to denote the inner product between vectors $\mathbf{a}$ and $\mathbf{b}$.
Given a set A, we write $a \xleftarrow{\$} A$ to indicate that a is sampled uniformly at random from A. If $\mathcal{D}$ is a probability distribution, we will write $d \leftarrow \mathcal{D}$ to denote that d is sampled according to $\mathcal{D}$.

2.2 Fully homomorphic encryption over the torus

Learning with errors. The Learning with Errors (LWE) problem was introduced by Regev in [Reg05]. We now propose a slightly different formulation based on a recent work by Chillotti *et al.* [CGGI16b]. Let n be a positive integer and χ be a probability distribution over $\mathbb{R}$ for the noise. For any vector $\mathbf{s} \in \{0,1\}^n$, we define the LWE distribution $\mathsf{lwe}_{n,\mathbf{s},\chi}$ as $(\mathbf{a}, b)$, where $\mathbf{a} \xleftarrow{\$} \mathbb{T}^n$ and $b = \langle \mathbf{s}, \mathbf{a} \rangle + e \in \mathbb{T}$, with $e \leftarrow \chi$.
Then the LWE assumption states that, for $\mathbf{s} \xleftarrow{\$} \{0,1\}^n$, it is hard to distinguish between $(\mathbf{a}, b)$ and $(\mathbf{u}, v)$, for $(\mathbf{a}, b) \leftarrow \mathsf{lwe}_{n,\mathbf{s},\chi}$ and $(\mathbf{u}, v) \xleftarrow{\$} \mathbb{T}^{n+1}$.

Sub-Gaussians. Let $\sigma > 0$ be a real Gaussian parameter. We define the Gaussian function with parameter σ as $\rho_\sigma(x) = \exp\left(-\pi |x|^2 / \sigma^2\right)$ for any $x \in \mathbb{R}$. Then we say that a distribution $\mathcal{D}$ is sub-Gaussian with parameter σ if there exists $M > 0$ such that for all $x \in \mathbb{R}$,

$$\mathcal{D}(x) \leq M \cdot \rho_\sigma(x).$$

All the error distributions that we consider in this paper are actually sub-Gaussian. This helps us bounding the probability of having decryption errors with the following lemma.

Lemma 2.1 (Pythagorean additivity of sub-Gaussians). *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be sub-Gaussian distributions with parameters σ_1 and σ_2, respectively. Then $\mathcal{D}^+$, obtained by sampling $\mathcal{D}_1$ and $\mathcal{D}_2$ and summing the results, is a sub-Gaussian with parameter $\sqrt{\sigma_1^2 + \sigma_2^2}$.*

LWE-based encryption scheme. We recall the Regev encryption scheme from [Reg05]. Let $\mu \in \{0,1\}$ be a message and λ the security parameter; we encrypt and decrypt as follows:

$\mathsf{Setup}(\lambda)$**:** choose $n = n(\lambda)$ and return $\mathbf{s} \xleftarrow{\$} \{0,1\}^n$
$\mathsf{Enc}(\mathbf{s}, \mu)$**:** return $(\mathbf{a}, b)$, with $\mathbf{a} \xleftarrow{\$} \mathbb{T}^n$ and $b = \langle \mathbf{s}, \mathbf{a} \rangle + e + \frac{\mu}{2}$, where $e \leftarrow \chi$
$\mathsf{Dec}(\mathbf{s}, (\mathbf{a}, b))$**:** return $\lfloor 2(b - \langle \mathbf{s}, \mathbf{a} \rangle) \rceil$

We usually refer to e as the *noise* of the ciphertext, and say that a ciphertext is a valid encryption of μ if it decrypts to μ with overwhelming probability.

We now give some notions on the formulation of FHE over the torus and the bootstrapping procedure. The following part is based on [CGGI16b].

TLWE. TLWE is a generalization of LWE and Ring-LWE [LPR10]. Let $k \geq 1$ be an integer, N be a power of 2 and χ be an error distribution over $\mathbb{R}_N[X]$. A TLWE secret key $\bar{\mathbf{s}} \in \mathbb{B}_N[X]^k$ is a vector of k polynomials over $\mathbb{Z}_N[X]$ with binary coefficients. Given a polynomial $\mu \in \mathbb{T}_N[X]$, a fresh TLWE sample of μ under the key $\bar{\mathbf{s}}$ is $(\mathbf{a}, b) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$, with $\mathbf{a} \xleftarrow{\$} \mathbb{T}_N[X]^k$ and $b = \bar{\mathbf{s}} \cdot \mathbf{a} + \mu + e$, where $e \leftarrow \chi$. From a TLWE encryption $\bar{c}$ of a polynomial $\mu \in \mathbb{T}_N[X]$ under a TLWE key $\bar{\mathbf{s}}$ we can extract a LWE encryption $c' = \mathsf{ExtractSample}(\bar{c})$ of the constant term of μ under an "extracted key" $\mathbf{s}' = \mathsf{ExtractKey}(\bar{\mathbf{s}})$. For the details of the algorithms $\mathsf{ExtractSample}$ and $\mathsf{ExtractKey}$, we refer the reader to [CGGI16b, Definition 4.1].

TGSW. TGSW is a generalized scale invariant version of the GSW FHE scheme [GSW13]. The key concept here is that TGSW can be seen as the "matrix equivalent" of TLWE, just like GSW can be seen as the "matrix equivalent" of LWE. More details can be found in [CGGI16b].

Overview of the bootstrapping procedure. The core idea for the efficiency of the new bootstrapping procedure is the so-called external product $\boxdot$, that performs the following mapping

$$\boxdot : TGSW \times TLWE \to TLWE$$

This allows for a further speed-up of a fast bootstrapping technique introduced by Ducas and Micciancio in [DM15], which in turn follows a sequence of works whose goal is to improve the efficiency of this costly operation (e.g. [AP13, AP14]).

Roughly speaking, the external product of a TGSW encryption of a polynomial $\mu_1 \in \mathbb{T}_N[X]$ and a TLWE encryption of a polynomial $\mu_2 \in \mathbb{T}_N[X]$ is a TLWE encryption of $(\mu_1 \cdot \mu_2) \in \mathbb{T}_N[X]$.
Intuitively, the bootstrapping procedure starts with a "pre-loaded" polynomial (called $\mathsf{testVector}$), then homomorphically rotates it by $b - \langle \mathbf{s}, \mathbf{a} \rangle$ and finally extracts a LWE ciphertext of its constant term. A nice feature of this approach is that the $\mathsf{testVector}$ can be initialized so that each possible value of $b - \langle \mathbf{s}, \mathbf{a} \rangle$ is mapped to a chosen output.
More specifically, the bootstrapping procedure takes as input the following arguments: a LWE sample under a key $\mathbf{s}$, a bootstrapping key and a keyswitching key. In particular, given a TLWE secret key $\bar{\mathbf{s}}$, the bootstrapping key bk is composed of TGSW encryptions of the n entries of $\mathbf{s}$ under $\bar{\mathbf{s}}$, and the keyswitching key ksk is composed of LWE encryptions of $\mathbf{s}' = \mathsf{ExtractKey}(\bar{\mathbf{s}})$ under $\mathbf{s}$. In order to bootstrap a LWE ciphertext $(\mathbf{a}, b)$,

1. Scale b and the entries of $\mathbf{a}$ by a factor $2N$ and round them to the closest integer;

2. Compute homomorphically the monomial $X^{b-\langle \mathbf{s},\mathbf{a} \rangle}$: we compute $X^{-s_i a_i}$ as $s_i \cdot X^{-a_i} + (1 - s_i)$ using the bootstrapping key bk in place of s_i and then multiply everything together with X^b by taking advantage of the external product operation. We note that X^b and all the X^{a_i} are easy to compute, since $\mathbf{a}$ and b are known;
3. Multiply $\mathsf{testVector}$ and the ciphertext calculated at step 2 to obtain a TLWE encryption and then extract a LWE ciphertext $(\mathbf{a}', b')$ of the desired output value;
4. Perform a keyswitching to obtain the final LWE ciphertext: this corresponds to homomorphically computing $b' - \langle \mathbf{s}', \mathbf{a}' \rangle$ by using ksk in place of $\mathbf{s}'$.

2.3 Artificial neural networks

An (artificial) neural network is a computing system inspired by biological brains. It is composed of a population of (artificial) neurons. Each neuron accepts multiple real-valued inputs $(x_1, \ldots, x_p)$ and performs two computations:

1. It computes $y = \sum_{j=1}^{p} w_j x_j + \beta$, which is a weighted sum of the inputs with some real values called *weights*: w_j is the weight associated to the input x_j and β is also real-valued and referred to as the *bias*.
2. It applies a non-linear function f, the *activation* function, and returns $f(y)$.

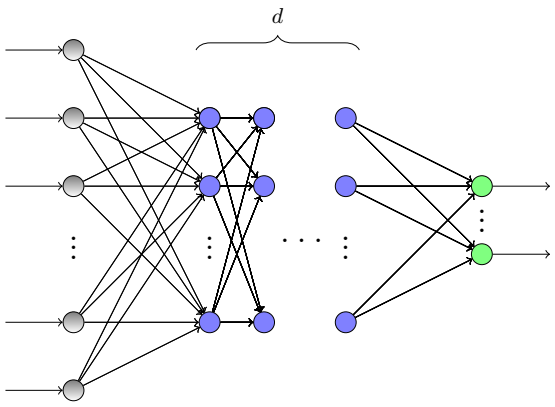

Fig. 2.1: A generic feed-forward neural network: one input layer, d hidden layers, and one output layer of varying sizes.

The neuron's output can be written vectorially as $f(\langle \mathbf{w}, \mathbf{x} \rangle)$ if one extends the input and weight vectors by appending 1 and β to them respectively. The neurons of a neural network are organized in successive layers, which are categorized according to their activation function. Neurons of one layer are connected to the neurons of the next layer by paths that are associated to weights. An input layer composed of the network's inputs as well as an output layer made of the network's output values are also added to the network. Internal layers are called *hidden*, since they are not directly accessible from the external world. NNs are usually composed of layers of various types: fully connected (every neuron of the layer takes all incoming signals as inputs), convolutional (it applies a convolution to its input), pooling, and so forth. Neural networks could in principle be recurrent systems, as opposed to the purely feed-forward ones, where each neuron is only evaluated once. The *universal approximation theorem* (see, e.g., [Hor91, Cyb89]) states that a neural network with a single hidden layer that contains a finite amount of neurons, can approximate any continuous function. The interest in evaluating deep neural networks is motivated by the fact that these systems have several layers of non-linearities, which allow to add levels of abstraction that cannot be obtained with shallow networks.
The FHE framework presented in this work is able to evaluate neural networks of arbitrary depth, that comprise possibly many hidden layers as depicted in Figure 2.1.

2.4 Backpropagation and stochastic gradient descent

Backpropagation is the standard algorithm for training a neural network, i.e., for finding a combination of weights that approximates well the correct mapping between inputs and outputs.
We now give a very high-level overview of the algorithm and we refer a curious reader to specialized literature (e.g., [LBOM98]) for more details. Given an input pattern, one calculates the output given by the network; then, according to the expected output that was associated to the input, one computes $\varepsilon(\text{expectedOutput} - \text{realOutput})$, where $\varepsilon(\cdot)$ is an error function.
Finally, one modifies the weights of the network in order to minimize the error, by following the direction given by the gradient of the error function with respect to the weights.
Calculating this gradient is the particularly tricky part: in theory, one should take into account all the pairs (input, expectedOutput) when doing so, but very large datasets can make this approach quickly impractical. A possible solution is then applying the *stochastic gradient descent* (SGD), where the gradient is calculated by taking into account only a subset of the total dataset. It has been verified that this approach is very effective in case of large-scale machine learning problems.
A typical issue when training a network is that of *overfitting*, where the network tends to "learn the training samples by heart", rather than generalizing and understanding the underlying structure and relationships. In order to mitigate this problem, a widely used technique is called *regularization*. Roughly speaking, an extra term is added to the error function, so that every time a weight is updated, a certain percentage of the weight is subtracted from itself. This helps keeping the absolute value of the weights small, which in turn helps preventing overfitting. For more details, we refer the reader to specialized literature, e.g., [KH91].
In the specific case of DiNNs, the details of our training procedure are presented in section 3.

2.5 The MNIST dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a dataset of images representing digits handwritten by more than 500 different writers, and is commonly used as a benchmark for training machine learning systems [LBBH98]. The MNIST database contains 60 000 training images and 10 000 testing images. The format of the images is 28×28 and the value of each pixel represents a level of gray. Moreover, each image is labeled with the digit it depicts.
A typical neural network for the MNIST dataset has $28 \cdot 28 = 784$ input nodes (hence one per pixel), an arbitrary number of hidden layers with an arbitrary number of neurons per layer, and finally 10 output nodes (one per possible digit). The output values can be interpreted as "scores" given by the NN: the classification is then given by the digit that achieves the highest score.
Over the years, the MNIST dataset has been a typical benchmark for classifiers, and many approaches have been applied: linear classifiers, principal component analysis, support vector machines, neural networks, convolutional neural networks, etc. For a more complete review on these approaches, we refer the reader to, e.g., [LBBH98]. Neural networks are known to perform well on this dataset, with accuracies that easily surpass 95%. For example, [LBBH98] proposes different architectures for neural networks and obtains more than 97% of correct classifications. More recent works even surpassed 99% of accuracy [CMS12]. For a nice overview on the results obtained on this dataset and on the techniques that were used, we refer the reader to [LCB98].

3 Discretized neural networks: training and evaluation

In this section we formally define DiNNs and we explain how to train and evaluate them.

First of all, we recall that state-of-the-art fully homomorphic encryption schemes cannot support operations over real messages[6]. Traditional neural networks have real-valued weights, and this incompatibility motivates investigating alternative architectures.

[6] As all the computations are done over the torus (i.e., modulo 1), scaling a ciphertext by any integer factor preserves the relations that make the decryption correct. However, this does not hold for non-integer factors.

Definition 3.1. *A Discretized Neural Network (DiNN) is a feed-forward artificial neural network whose inputs are integer values in* $\{-I, \dots, I\}$ *and whose weights are integer values in* $\{-W, \dots, W\}$*, for some* $I, W \in \mathbb{N}$*. For every neuron of the network, the activation function maps the inner product between the incoming inputs vector and the corresponding weights to integer values in* $\{-I, \dots, I\}$*.*

In particular, for this paper we choose $\{-1, 1\}$ as the input space and $\mathsf{sign}(\cdot)$ as the activation function for the hidden layers:

$$\mathsf{sign}(x) = \begin{cases} -1, & x < 0, \\ +1, & x \geq 0. \end{cases} \tag{3.1}$$

These choices are inspired by the fact that we designed the model with the idea of performing homomorphic evaluations over encrypted input. As a consequence, we wanted the message space to be as small as possible, which, in turn, would allow us to increase the efficiency of the overall evaluation.
We also note that using an activation function whose output is in the same range as the network's input allows us to maintain the same semantics across different layers. In our case, what enters a neuron is always a weighted sum of values in $\{-1, 1\}$.

3.1 Evaluating a DiNN

In order to make the evaluation of the network compatible with FHE schemes, discretizing the input space is not sufficient: we also need to discretize the weights of the network. Our discretization of the weights then happens by rounding them to the closest multiple of a value τ (tick size), which can be treated as a parameter.

$$\mathsf{processWeight}(w, \tau) = \tau \cdot \left\lfloor \frac{w}{\tau} \right\rceil \tag{3.2}$$

In the following, we use daggers (†) to denote weights that have been discretized through the formula in Equation 3.2.
The parameter τ has to be chosen carefully, since it defines the message space that our encryption scheme must support. Thus, we want the bound on $\langle \mathbf{w}^\dagger, \mathbf{x} \rangle$ to be small for all neurons, where $\mathbf{w}^\dagger$ and $\mathbf{x}$ are the discretized weights and the inputs associated to the neuron, respectively. In the rest of the paper, we refer to $\langle \mathbf{w}^\dagger, \mathbf{x} \rangle$ as a *multisum*.

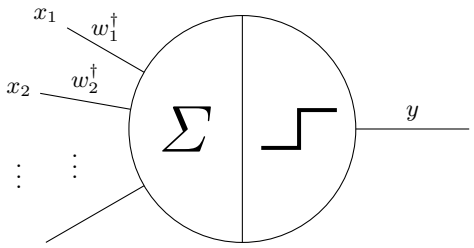

Fig. 3.1: Evaluation of a single neuron. The output value is $y = \mathsf{sign}\left(\langle \mathbf{w}^\dagger, \mathbf{x} \rangle\right)$, where $w_i^\dagger$ are the processed weights associated to the incoming wires and x_i are the corresponding input values.

Then, evaluating a DiNN essentially amounts to processing the weights obtained in the training phase and propagating the inputs through all the layers, according to the following procedure:

1. Initialize the input wires $\mathbf{x}$ to the input data. In the rest of the procedure, $\mathbf{x}$ represents the input of the current layer;
2. For each but the last layer, feed-forward the input to the next layer by computing $\mathbf{x} := \mathsf{sign}\left(\mathbf{W}_i^\dagger \cdot \mathbf{x}\right)$, where $\mathbf{W}_i^\dagger$ is the matrix that contains the processed weights associated to the i-th layer. For clarity, in Figure 3.1 we show the evaluation of a single neuron;

3. Feed-forward the result to the output layer by computing $\mathsf{output} := \mathbf{W}_o^\dagger \cdot \mathbf{x}$, where $\mathbf{W}_o^\dagger$ is the matrix of processed weights associated to the output layer[7].

3.2 How to train a DiNN

Choosing $\mathsf{sign}(\cdot)$ as activation function for the hidden layers reduces the message space that our scheme has to support, which positively impacts the efficiency of our FHE evaluation. However, this binarization step however causes severe problems in the backpropagation algorithm, because the gradient of this function is zero almost everywhere, making this function incompatible with the gradient descent method. As in [CHS+16], we go on by replacing the true gradient $\frac{d}{dx}\mathsf{sign}(x)$ with a rectangular function:

$$\Pi(x) = \begin{cases} 1, & |x| \leq 1, \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

In order to have a continuous activation function for the last layer, which allows us to make the training sensible even to small variations in the output, we substitute $\mathsf{sign}(\cdot)$ with a continuous function, which we denote as $\mathsf{act}(\cdot)$. For example, $\mathsf{sign}(\cdot)$ can be approximated with $\mathsf{arctan}(\cdot)$ or $\mathsf{tanh}(\cdot)$, and both these functions have already been successfully used in the machine learning literature. In Figure 3.2 we sketch the plots of these functions.

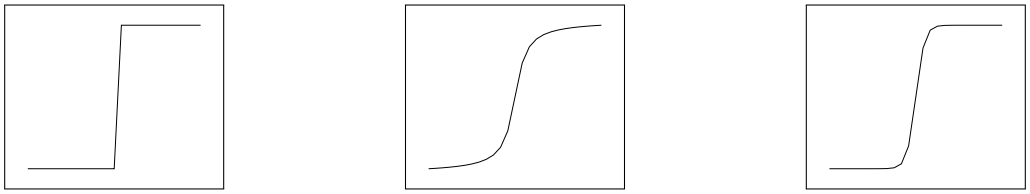

Fig. 3.2: Possible activation functions for the neurons. We use $\mathsf{sign}(\cdot)$ for the neurons in the hidden layers and a continuous approximation for the neurons in the output layer.

The training of the network then follows the classical backpropagation algorithm with stochastic gradient descent (SGD), specifically adapted to work with our binarization step. In particular, using SGD means that the gradient on the training set is approximated by the gradient calculated on a subset of it, called *minibatch*. The dimension of each of these minibatches is a parameter of the learning phase, which can also be changed dynamically during the training.
Finally, we stress the following fact: since the weights used in the evaluation phase will be discrete values, we also discretize them during the forward pass of our training algorithm. The reason behind this choice is simple: this helps us obtain the set of weights that performs best (i.e., minimizes the classification error), *when discretized*.
We now sketch the steps of the algorithm. First, initialize the weights to random values and partition the training set in minibatches, then for each minibatch:

1. Initialize the input wires $\mathbf{x}$ to the input data. In the rest of the procedure, $\mathbf{x}$ represents the input of the current layer;
2. For each but the last layer, feed-forward the input to the next layer by computing $\mathbf{x} := \mathsf{sign}\left(\mathbf{W}_i^\dagger \cdot \mathbf{x}\right)$, where $\mathbf{W}_i^\dagger$ is the matrix that contains the processed weights associated to the i-th layer;

[7] Note that here we do not apply any activation function to the neurons in the output layer. Since we use the network for classifying inputs, we are only interested in sorting the classes according to the given score. A typical activation function is monotonic, so applying it to the neurons in the output layer would not change anything in the result of the classification.

3. Propagate the result to the output layer by computing $\mathsf{output} := \mathsf{act}\left(\mathbf{W}_o^\dagger \cdot \mathbf{x}\right)$, where $\mathbf{W}_o^\dagger$ is the matrix of processed weights associated to the output layer;
4. Calculate the error ε as a function of the difference between output and the expected output given by the labels[8];
5. Calculate $\nabla_i = \dfrac{\partial \varepsilon}{\partial \mathbf{W}_i}$ for $i \in \{1, 2\}$, i.e., the gradient of the error function with respect to the weights[9];
6. Update the weights according to the formula $\mathbf{W}_i := \mathbf{W}_i - \eta\left(\nabla_i + \xi \mathbf{W}_i\right)$, where η is a parameter called *learning rate* and ξ is a parameter called *regularization factor*.

It is fundamental to notice that the training algorithm is executed on real weights. As already noted in [CB16], this seems necessary for achieving a good result.

4 Homomorphic evaluation of a DiNN

We now give a high level description of our procedure to homomorphically evaluate a DiNN. We basically need two ingredients: we need to be able to compute the multisum between the encrypted inputs and the weights and we need to homomorphically extract the sign of the result. In order to maintain the scalability of our scheme across the layers of the DiNN, we perform a bootstrapping for every neuron in the hidden layers. This ensures that the ciphertext encrypting the sign of the result can be used for further computations, without a fixed limit on the number of layers that the DiNN can contain. This allows us to evaluate even deep neural networks and choose parameters that are independent of the number of layers.

4.1 Evaluating the multisum

In our framework, the weights of the network are available in clear, so we can evaluate the multisum by just using homomorphic additions. The only things that we have to care about are the message space of our encryption scheme – which has to be large enough to accommodate for the largest possible result of the operation – and the noise level, which might grow too much and perturb the result.

Extending the message space. In order for our FHE scheme to be able to correctly evaluate the multisum, we need all the possible values of the multisum to be inside our message space. To this end, we extend the TFHE scheme from [CGGI16b].

Construction 1 (Extended LWE-based encryption scheme) *Let B be a positive integer and let $m \in [-B, B]$ be a message. Then we split the torus into $2B + 2$ "slices", one for each possible message plus a "forbidden" one, and we encrypt and decrypt as follows:*

$\mathsf{Setup}(\lambda)$**:** *choose $n = n(\lambda)$ and return $\mathbf{s} \xleftarrow{\$} \mathbb{T}^n$*
$\mathsf{Enc}(\mathbf{s}, m)$**:** *return $(\mathbf{a}, b)$, with $\mathbf{a} \xleftarrow{\$} \mathbb{T}^n$ and $b = \langle \mathbf{s}, \mathbf{a} \rangle + e + \frac{m}{2B+2}$, where $e \leftarrow \chi$*
$\mathsf{Dec}(\mathbf{s}, (\mathbf{a}, b))$**:** *return $\lfloor (b - \langle \mathbf{s}, \mathbf{a} \rangle) \cdot (2B + 2) \rceil$*

Note that ciphertexts can be added and scaled by a known constant in the following way: for any secret key $\mathbf{s}$, any $c_1 = (\mathbf{a}_1, b_1) \leftarrow \mathsf{Enc}(\mathbf{s}, m_1)$, $c_2 = (\mathbf{a}_2, b_2) \leftarrow \mathsf{Enc}(\mathbf{s}, m_2)$, and constant $w \in \mathbb{Z}$, we have that $\mathsf{Dec}(\mathbf{s}, (\mathbf{a}_1 + w \cdot \mathbf{a}_2, b_1 + w \cdot b_2)) = m_1 + w \cdot m_2$ as long as $m_1 + w \cdot m_2 \in [-B, B]$ and the noise does not grow too much. The first condition is easily met by choosing $B = \max_{\mathbf{w}} \|\mathbf{w}\|_1$.
The purpose of the forbidden area is to prevent overflows that would cause sign changes from B to $-B$ and vice versa, thus allowing us to relax the constraints on the parameters. An attentive reader might notice that we have the same problem around the value 0. Although this is true, we conjecture that this is not a serious danger for the correctness of the classification. The intuitive reason is that the magnitude of a neuron's output is related to the confidence that the network has in the value. In different terms, a sign change between +1 and -1 is definitely less serious than one between B and $-B$.

[8] The expected output is transformed into an indicator vector, i.e., a number $d \in [0, 9]$ is transformed into a vector in $\mathbb{Z}^{10}$, whose entries are all 0s apart from the d-th, which is 1. This is the real target that the network is trying to approximate.
[9] Remember that the derivative of the sign function is computed as per Equation 3.3.

Fixing the noise. Increasing the message space has an impact on the choice of parameters. Evaluating the multisum with weights $\mathbf{w}$ means that, if the standard deviation of the initial noise is σ, then the standard deviation of the output noise is as high as $\|\mathbf{w}\|_2 \cdot \sigma$, which in turn means that our initial standard deviation must be smaller than the one in [CGGI16b] by a factor $\max_{\mathbf{w}} \|\mathbf{w}\|_2$. Moreover, for correctness to hold, we need the noise to remain smaller than half of a slice of the torus. As we are splitting the torus in $2B + 2$ slices rather than 2, we need to further decrease the noise by a factor B. Special attention must be paid to security: taking a smaller noise might in fact compromise the security of the scheme. In order to mitigate this problem, we can increase the dimension of the LWE problem n, but this in turn induces more rounding errors (in the first part of the bootstrapping procedure). In conclusion, the best approach seems to be choosing secure parameters and then experimenting to pick the set that guarantees the highest accuracy.

4.2 Homomorphic computation of the sign function

We take advantage of the flexibility of the bootstrapping technique introduced by Chillotti *et al.* [CGGI16b] in order to perform the sign extraction and the bootstrapping at the same time. This only requires changing the testVector to program the values we want to recover after the bootstrapping. The first step of the bootstrapping basically consists in mapping the torus $\mathbb{T}$ to an object that we will refer to as the wheel. This wheel is split into $2N$ "ticks" that are associated to the possible values that are encrypted in the bootstrapped ciphertext. Programming the testVector means choosing N of these values; the remaining N are automatically set to the opposite values because of the anticyclicproperty of $\mathbb{T}_N[X]$. In order to extract the sign, we thus want the top part of the wheel to be associated to the value $+1$. The bottom part will then be automatically associated to -1. This idea is illustrated in Figure 4.1.

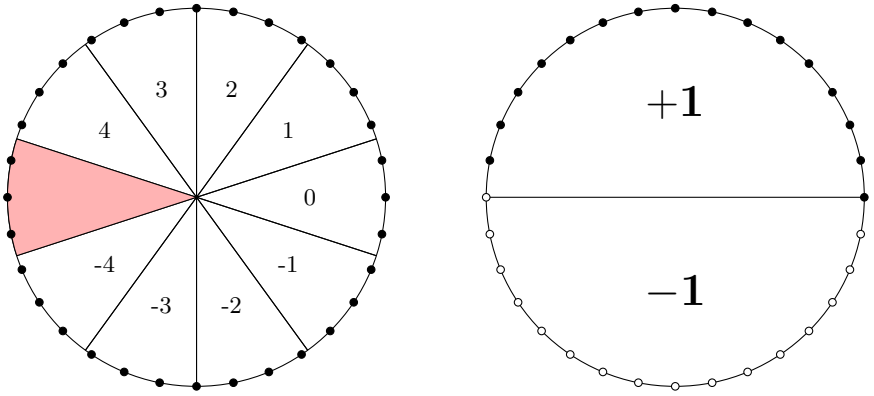

Fig. 4.1: On the left, we show the first step of the boostrapping, which consists in embedding the torus (the continuous circle) over the wheel (the $2N$ dots on it) by rounding to the closest dot. Each slice corresponds to one of the possible results of the multisum operation (the colored slice represents the forbidden zone). On the right we show the final result of the bootstrapping: each dot of the top part of the wheel is mapped to $+1$ and each dot of the bottom part to -1. This can be roughly seen as embedding the wheel back onto the torus.

Another interesting feature of the bootstrapping procedure of Chillotti *et al.* is that we can dynamically change the message space during the bootstrapping. This allows us to set smaller parameters and smartly use all the space in the torus to minimize the errors (by taking bigger slices).

From now on, we say that a bootstrapping is *correct* if, given a valid encryption of a message μ, its output is a valid encryption of $\mathsf{sign}(\mu)$ with overwhelming probability.

4.3 Scale-invariance

If the parameters are set correctly then, by using the two operations described above, we can homomorphically evaluate neural networks of any depth. In particular, the choice of parameters is independent of the depth of the neural network. This result cannot be achieved with previous techniques of evaluation relying on somewhat homomorphic evaluations of the network. In fact, they have to choose parameters that accommodate for the whole computation, whereas our method only requires the parameters to accommodate for the evaluation of a single neuron. The rest of the computation follows by induction. More precisely, our choice of parameters only depends on bounds on the norms ($\|\cdot\|_1$ and $\|\cdot\|_2$) of the input weights of a neuron. In the following, we denote these bounds by M_1 and M_2, respectively.
We say that the homomorphic evaluation of the neural network is *correct* if the decryptions of its output scores are equal to the scores given by its evaluation in the clear with overwhelming probability. Then, the scale-invariance is formally defined by the following theorem:

Theorem 4.1 (Scale-invariance of our homomorphic evaluation). *For any* DiNN *of any depth, let σ be a Gaussian parameter such that the noise of* $\mathsf{Bootstrap}(\mathsf{bk}, \mathsf{ksk}, \cdot)$ *is sub-Gaussian with parameter σ. Then if the bootstrapping is correct on input ciphertexts with sub-Gaussian noise of parameter $\frac{\sigma}{M_2}$ and message space larger than $2M_1 + 2$, the result of the homomorphic evaluation of the* DiNN *is correct.*

Proof. The proof is a simple induction on the structure of the neural network. First, the correctness of the evaluation of the first layer is implied by the choice of parameters for the encryption[10].
If the evaluation is correct for all neurons of the ℓ-th layer, then the correctness for all neurons of the $(\ell + 1)$-th layer follows from the two observations made in the previous subsections:

- The result of the homomorphic evaluation of the multisum is a valid encryption of the multisum;
- The result of the bootstrapping is a valid encryption of the sign of the multisum.

This first fact is implied by the choice of the message space, since the multisum value is contained in $[-M_1, M_1]$. The second one comes directly from the correctness of the bootstrapping, because the homomorphic computation of the multisum on ciphertexts with sub-Gaussian noise of parameter σ yields a ciphertext with sub-Gaussian noise of parameter at most σM_2 (cf. Lemma 2.1).
Then, the correctness of the encryption scheme ensures that the final ciphertexts are valid encryptions of the scores. □

We note that the distribution of the noise in the ciphertext output by the bootstrapping procedure is sub-Gaussian; a bound on its Gaussian parameter is provided in [CGGI16b], together with how to pinpoint the parameters in order to ensure the correctness of the bootstrapping.

5 Experimental results and conclusions

We implemented the proposed approach to test its accuracy and efficiency. This section is divided into two parts: the first one describes the training of the neural network over data in the clear and the second one details the results obtained when evaluating the network over encrypted inputs.

5.1 Pre-processing the MNIST database

First of all, we re-scaled every size-normalized and centered digit of the MNIST dataset from 28×28 to 16×16 pixels. This step actually decreases the number of inputs from 784 to $16 \cdot 16 = 256$, possibly sacrificing accuracy of our network. On the other hand, this modification allows us to facilitate the homomorphic evaluation of the network. Next, the 8-bit depth values of the given MNIST gray-scale images have been clamped to 1-bit of information content – black or white – for both the offline and online phase of working with our network. Specifically, we binarized the images with a threshold value equal to 128: any pixel whose value is smaller than this threshold is mapped to -1; the others are mapped to $+1$. The resulting input dataset hence comprises smaller, black-and-white images. There is obviously a trade-off between the size of the input, the "amount of information" it contains and the classification quality, but we think this experiment setting is a good compromise for serving our purpose.

[10] If it is not, we can bootstrap all input ciphertexts in order to ensure this holds.

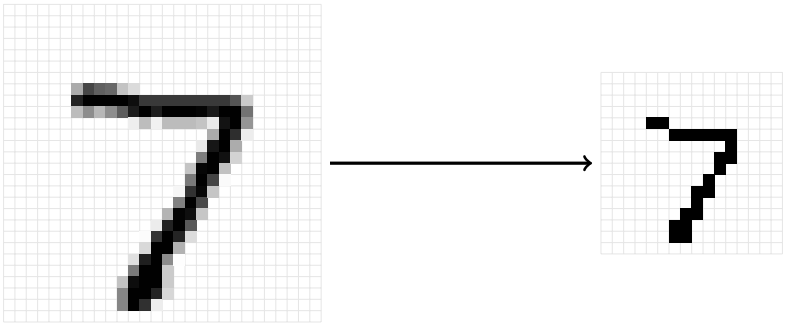

Fig. 5.1: An example of our preprocessing on one of the images from the test set of the MNIST database.

5.2 Training the DiNN over data in the clear

In order to train the neural network, we first chose its "topology", i.e., the number of hidden layers and neurons per hidden layer. We experimented with several values, always keeping in mind that a smaller number of neurons per layer is preferable: having more neurons means that the value of the multisum will be potentially higher, thus requiring a larger message space in the homomorphic evaluation, which in turn forces to choose bigger parameters for the scheme. After some tries, we settled for a neural network with a single hidden layer composed of 30 neurons. Although other values are certainly worth trying, we believe this offers a good compromise between classification accuracy and size of parameters.
In practice, in order to train the neural network, we wrote a Python script that allowed us to have the maximum flexibility in the choice of the structure and the parameters.
For training the network, we fixed the initial value of the learning rate to $\eta = 0.006$ and we used an approach called *learning rate decay*: η is multiplied by a constant factor (smaller than 1) every few epochs. In particular, we multiplied the learning rate by 0.95 every 5 epochs of training. The idea behind this procedure is that of progressively reducing the "width of the step" in the optimization process. This results in wider steps – which will hopefully set the cost minimization towards the right direction – at the beginning of the training, followed by narrower steps to "refine" the set of weights that has been found. The size of the minibatches has been fixed to 10 and kept constant throughout the entire training phase. The discretization of the weights happens according to Equation 3.2. For choosing the final value of tick size τ we experimented and found that $\tau = \frac{1}{10}$ is a good compromise between accuracy (more possible values for the weights give the network more expressiveness) and the message space the homomorphic scheme will have to support. We observed the following bounds on the norms of the weight vectors:

$$\begin{aligned} \max_i \left\| \mathbf{w}^\dagger_{1,i} \right\|_1 &= 5829, & \max_i \left\| \mathbf{w}^\dagger_{1,i} \right\|_2 &< 368, \\ \max_i \left\| \mathbf{w}^\dagger_{2,i} \right\|_1 &= 696, & \max_i \left\| \mathbf{w}^\dagger_{2,i} \right\|_2 &< 135; \end{aligned} \tag{5.1}$$

where $\mathbf{w}^\dagger_{j,i}$ is the vector of weights associated to the i-th neuron of the j-th layer.

Finally, a few words on the activation function: during the training phase we experimented both with $\mathsf{act}(\cdot) = \mathsf{arctan}(\cdot)$ and with $\mathsf{act}(\cdot) = \mathsf{tanh}(\cdot)$, finally settling for the latter.
Through this procedure, we were able to design a FHE-friendly model of neural network and find a set of weights that allows it to achieve 93.17% of correct classifications when evaluated in the clear on the test set.
In Figure 5.2 we show the architecture that we considered in this work: a fully connected neural network with one hidden layer composed of 30 neurons.

5.3 Classifying encrypted inputs

Implementing the homomorphic evaluation of the neural network over encrypted input was more than a mere coding exercise, but allowed us to discover several interesting properties of our DiNNs.

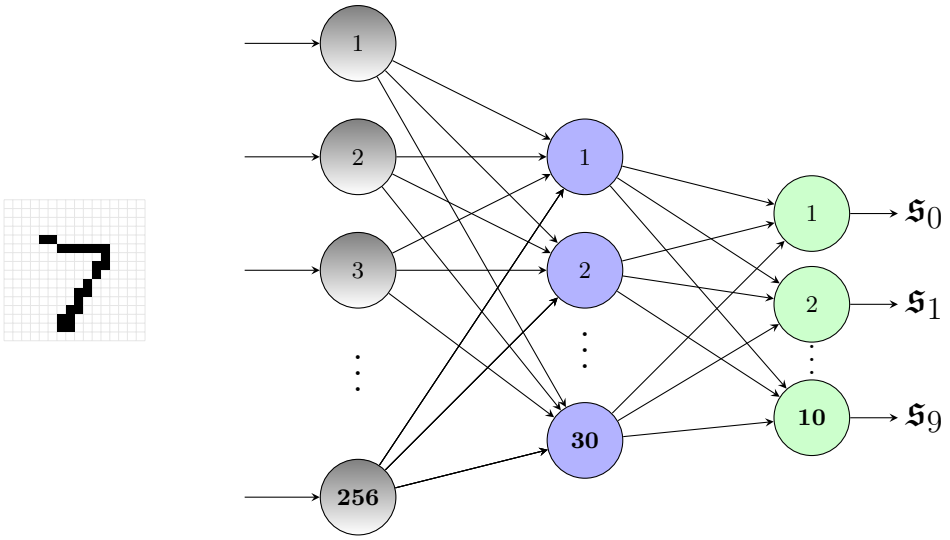

Fig. 5.2: Depiction of our neural network with 256:30:10–topology. It takes as input the pixels that compose the image and outputs the scores $\mathfrak{s}_i$ assigned to each digit.

The starting point was the TFHE library by Chillotti *et al.* , which is freely available on GitHub [CGGI16a] and which was used to efficiently perform the bootstrapping operation. The library takes advantage of FFT processors for fast polynomial multiplication and, although not parallelized, achieves excellent timing results. We extended the code to apply this fast bootstrapping procedure to our use case.

Parameters. We now present our setting of the parameters, following the notation of [CGGI16b], to which we refer the reader for further details.

- Dimension of the LWE problem: $n = 600$;
- Noise parameter of LWE ciphertexts: $\sigma = 2^{-20}$;
- Degree of the polynomials in the ring: $N = 1024$;
- Dimension of the $TLWE$ problem: $k = 1$;
- Basis for the decomposition of $TGSW$ ciphertexts: $Bg = 1024$;
- Length of the decomposition of $TGSW$ ciphertexts: $\ell = 3$;
- Basis for the decomposition during key switching: 2;
- Length of the decomposition during key switching: $t = 30$;

With this choice of parameters, we achieve roughly 80 bits of security and we are able to homomorphically classify an image in approximately 0.88 sec/classification[11] when running the code on an Intel Core i7-4720HQ CPU @ 2.60GHz.

Evaluation. Our homomorphic evaluation then proceeds as follows:

1. For each image in the test set, encrypt the image under the extended LWE encryption scheme with parameter $B = 349$ (cfr. Construction 1). The choice of this parameter was made after analyzing the weights obtained by training the neural network;
2. For every neuron in the hidden layer, calculate the multisum by homomorphically scaling each input by the corresponding weight and then adding them together;
3. Bootstrap each of the resulting ciphertexts to extract the sign. This corresponds to the following mapping:
$$\mathsf{Encryption}\left(\langle \mathbf{w}^\dagger, \mathbf{x}\rangle\right) \to \mathsf{Encryption}\left(\mathsf{sign}\left(\langle \mathbf{w}^\dagger, \mathbf{x}\rangle\right)\right)$$
During the bootstrapping we also reduce the message space from $2B + 2 = 700$ slices to 400 slices. This latter value was empirically chosen and verified to work well with our construction;

[11] Profiling the code showed that the bootstrapping procedure takes more than 97% of the execution time.

```
1 ### Classified samples: 10 000
2 Time per bootstrapping: 0.0256455 sec/bootstrapping
3 Errors: 706  / 10 000 (7.06%)
4 Disagreements: 205
5 (pro-clear/pro-hom: 97 / 74)
6 Wrong bootstrappings: 1709
7 Disagreements with wrong bootstrapping: 122
8 Avg. time for the evaluation of the network: 0.880569 sec/classification
```

Fig. 5.3: Results of homomorphic evaluation of the DiNN on the full test set. Timings refer to the execution on a single core of an Intel Core i7-4720HQ CPU @ 2.60GHz processor.

4. For every neuron in the output layer, calculate the multisum as in step 2;
5. Return the 10 ciphertexts corresponding to the 10 scores assigned by the neural network.

The homomorphic evaluation of the network on the entire test set was compared to its classification in the clear and we observed the following facts:

Observation 1 *The accuracy achieved when classifying encrypted images is close to that obtained when classifying images in the clear.*

Out of the 10 000 images in the test set, the accuracy of the classification was 93.17% in the case of inputs in the clear and 92.94% in the case of encrypted inputs. This gap is explained by the following observations.

Observation 2 *During the evaluation, some signs are flipped during the bootstrapping but this does not significantly harm the accuracy of the network.*

We set aggressive parameters, knowing that this could sometimes lead the bootstrapping procedure to return an incorrect result when extracting the sign of a message. In fact, we conjectured that the neural network would be resilient to perturbations and experimental results proved that this is indeed the case: when running our experiment over the full test set, we noticed that the number of wrong bootstrappings is 1709 (out of 300 000) but this did not change the outcome of the classification in more than 122 cases (cf. Figure 5.3, line 6).

Observation 3 *The classification of an encrypted image might disagree with the classification of the same image in the clear but this does not significantly worsen the overall accuracy.*

This is a property that we expected during the implementation phase and our intuition to explain this fact is the following: the network is assigning 10 scores to each image, one per digit, and when two scores are close (i.e., the network is "hesitating" between two classes), it can happen that the classification in the clear is correct and the one over the encrypted image is wrong. But the opposite can also be true, thus leading to classifying correctly a sample that was misclassified in the clear. We experimentally verified that disagreements between the evaluations do not automatically imply that the homomorphic classification is worse than the one in the clear: out of 205 disagreements, the classification in the clear was correct 97 times, against 74 times for the homomorphic one[12] (cf. Figure 5.3, line 4).

Observation 4 *Despite the bounds given in Equation 5.1, we were able to choose a considerably smaller message space without harming the accuracy of the classification.*

This is explained by the fact that, as shown by the experiments, the concrete values of the multisums are almost always far from the bound, thus allowing for restricting the message space.

[12] In the remaining cases, the classifications were different but they were both wrong.

Before stating some open problems, we conclude with the following note: using a bigger neural network can lead to a better classification accuracy, at the cost of performing more calculations and, above all, more bootstrapping operations. For example, we noticed that a network with topology 256:200:10 (i.e., one hidden layer with 200 neurons), the accuracy grows higher than 95%. Although it is true that evaluating a bigger network is computationally more expensive, we stress that the bootstrapping operations are independent of each other and can thus be performed in parallel.

Open problems. This work opens a number of possibilities and, thus, raises several interesting open problems. The first one is about training our DiNNs more efficiently, e.g. by running the training algorithm on GPUs or parallelizing it. More generally, the machine learning literature is full of hacks and improvements for the learning phase and it would be interesting to see if these carry over to our model and help improving accuracy. This would open the way to Deep DiNNs, i.e., discretized neural networks composed of a huge number of layers, that we can evaluate thanks to the scale-invariant property of our approach. Another natural question is whether we can batch several bootstrappings together, in order to improve the overall efficiency of the evaluation.
Most interestingly, our framework supports a larger variety of cognitive architectures (e.g. Recurrent Neural Networks [Elm91]) and we leave training and evaluating them as an interesting open problem. Finally, the methodology presented in this work is by no means limited to image recognition, but can be applied to other machine learning problems as well.

References

AP13. Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 1–20. Springer, Heidelberg, August 2013.

AP14. Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 297–314. Springer, Heidelberg, August 2014.

AS00. Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. *SIGMOD Rec.*, 29(2):439–450, May 2000.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

BV11a. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

BV11b. Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Heidelberg, August 2011.

BV14. Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based FHE as secure as PKE. In Moni Naor, editor, *ITCS 2014*, pages 1–12. ACM, January 2014.

CB16. Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

CdWM+17. Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.

CGGI16a. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast Fully Homomorphic Encryption Library over the Torus. https://github.com/tfhe/tfhe, 2016.

CGGI16b. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2016.

CHS+16. M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *ArXiv e-prints*, February 2016.

CMS12. D. Cireşan, U. Meier, and J. Schmidhuber. Multi-column Deep Neural Networks for Image Classification. *ArXiv e-prints*, February 2012.

Cyb89. G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

DGBL+16. Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. Technical report, February 2016.

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Heidelberg, April 2015.

Dwo06. Cynthia Dwork. Differential privacy (invited paper). In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP 2006, Part II*, volume 4052 of *LNCS*, pages 1–12. Springer, Heidelberg, July 2006.

Elm91. Jeffrey L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7(2):195–225, Sep 1991.

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

GHS12. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

GSW13. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Heidelberg, August 2013.

Hor91. Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991.

HZRS15. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

KH91. Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Proceedings of the 4th International Conference on Neural Information Processing Systems*, NIPS'91, pages 950–957, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

LBBH98. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

LBOM98. Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

LCB98. Y. LeCun, C. Cortes, and C.J.C. Burges. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May 2010.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

SS10. Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 377–394. Springer, Heidelberg, December 2010.

SV10. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 420–443. Springer, Heidelberg, May 2010.

vDGHV10. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 24–43. Springer, Heidelberg, May 2010.

ZK16. Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *CoRR*, abs/1605.07146, 2016.

ZYC16. Qingchen Zhang, Laurence T. Yang, and Zhikui Chen. Privacy preserving deep computation model on cloud for big data feature learning. *IEEE Transactions on Computers*, 65(5):1351–1362, 2016.