# Intellectual Property Protection for Distributed Neural Networks
## *Towards Confidentiality of Data, Model, and Inference*

Laurent Gomez[1], Alberto Ibarrondo[1], José Márquez[1] and Patrick Duverger[2]
[1]*SAP Security Research, 805, Avenue Dr. Maurice Donat, 06250 Sophia-Antipolis, France*
[2]*City of Antibes Juan-les-Pins, France*

Keywords: Intellectual Property Protection, Fully Homomorphic Encryption, Neural Networks, Distributed Landscapes, Smart Cities.

Abstract: Capitalizing on recent advances on HPC, GPUs, GPGPUs along with the rising amounts of publicly available labeled data; (Deep) Neural Networks (NN) have and will revolutionize virtually every current application domain as well as enable novel ones such as those on recognition, autonomous, predictive, resilient, self-managed, adaptive, and evolving applications. Nevertheless, it is to point out that NN training is rather resource intensive in data, time and energy; turning the resulting trained models into valuable assets representing an Intellectual Property (IP) imperatively worth of being protected. Furthermore, in the wake of Edge computing, NNs are being progressively deployed across decentralized landscapes; as a consequence, IP owners take very seriously the protection of their NN based software products. In this paper we propose to leverage Fully Homomorphic Encryption (FHE) to protect simultaneously the IP of trained NN based software, as well as the input data and inferences. Within the context of a smart city scenario, we outline our NN model-agnostic approach, approximating and decomposing the NN operations into linearized transformations while employing a Single Instruction Multiple Data (SIMD) for vectorizing operations.

## NOMENCLATURE

| | |
|---|---|
| $v$,$\mathbf{v}$,$\mathbf{V}$ | Scalar, Vector, Matrix/Tensor |
| $\langle t\rangle_{\mathbf{pub}}$ | Tensor $t$ encrypted with key *pub* |

## 1 INTRODUCTION

### 1.1 Motivation

Mimicking human's cortex, Neural Networks (NN) enable computers to learn through training. With the recent progress on GPU based computing capabilities, NN have received major improvements such as Convolutional Layers (Krizhevsky et al., 2012), Batch Normalization (Ioffe and Szegedy, 2015) or Residual Blocks (He et al., 2016). As part of the Deep Learning (DL) (Goodfellow et al., 2016) field, DNN have revolutionized the creation of software based applications for problems with a non-deterministic solution space (e.g. object detection, facial recognition, autonomous driving, video processing, among others).

But GPUs hardware, and labeled data sets come at a cost. In addition, NN training is data, time and energy-intensive. This makes the outcome of DL training very valuable: the topology, the number and type of hidden layers including design characteristics (defined before training); and specially the model, the values of all the parameters in the trained network.

Furthermore, with the rise of edge computing and Internet of Things (IoT), NN are meant to be deployed outside of corporate boundaries, closer to customer business and in potentially insecure environments. Industrial actors take very seriously the Intellectual Property (IP) protection of trained DNN. This new paradigm calls for solutions to protect IP of distributed DL inference processing systems, with DNN deployment and execution on decentralized systems.

The lack of solutions for IP protection exposes trained NN owners to reverse engineering on their DL models (Tramèr et al., 2016). As outlined in (Augasta and Kathirvalavakumar, 2012) (Floares, 2008), attackers can steal trained NN models. In such new coding paradigm, where design patterns are enforced in known and legacy implementations, the question of IP is at stake. The question is not so much how to protect the DNN architecture (since most architectures are grounded on well known research), but rather how to protect the trained DNN model.

Gomez, L., Ibarrondo, A., Márquez, J. and Duverger, P.
Intellectual Property Protection for Distributed Neural Networks - Towards Confidentiality of Data, Model, and Inference.
DOI: 10.5220/0006854701470154
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 147-154
ISBN: 978-989-758-319-3

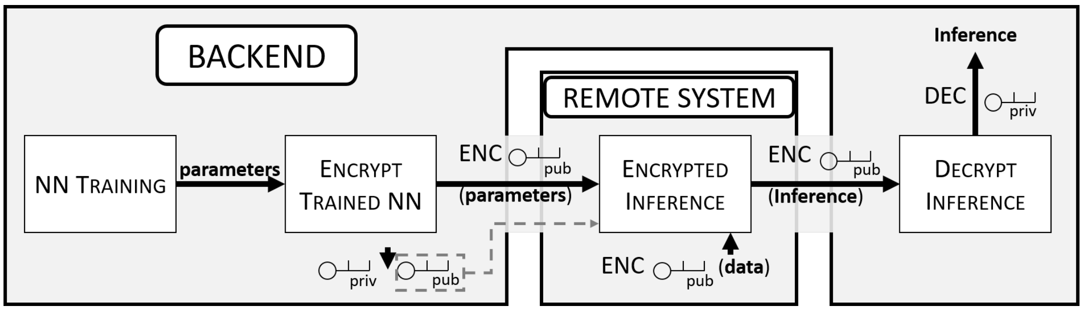

Figure 1: Diagram of IP protection solution.

## 1.2 State of the Art

Applying security to (Deep) Neural Networks is a current research topic sought using mainly two different techniques: variants of Fully Homomorphic Encryption/**FHE** (Gentry, 2009) and Secure Mupti-party Computation/**SMC** (Cramer et al., 2015). While FHE techniques allow encrypted addition and multiplication in a single machine, SMC employs gated circuits to perform arithmetic operations on shared data across several communicating machines. With these techniques at hand, NN protection is pursued for two main phases: **training** and **classification/inference**.

Secure NN training has been tackled using FHE (Graepel et al., 2012) and SMC (Shokri and Shmatikov, 2015), disregarding protection once the trained model is to be deployed and used. Other Machine Learning models such as linear and logistic regressions have also been trained in a secure way in (Mohassel and Zhang, 2017).

Regarding classification, SMC has led to cooperative solutions where several devices work together to obtain federated inferences (Liu et al., 2017), not supporting deployment of the trained NN to trusted decentralized systems. Inference using FHE encrypted data was covered in *cryptonets* (Gilad-Bachrach et al., 2016), improved in (Chabanne et al., 2017) and (Hesamifard et al., 2017). While preventing disclosure of data at inference phase, the security of the model itself is out of their scope.

So far, the only research addressing IP protection of NNs used watermarking (Uchida et al., 2017). Even though this technique can detect infringement, it cannot be prevented, thus failing to preserve confidentiality neither on the input data, inference nor on the NN model.

Regarding IP protection of the NN, the problem has been only addressed using watermarking (Uchida et al., 2017). In this case, even though infringement can be detected, without preventing it, no confidentiality preserving solution is elaborated neither on the input data, inference nor on the NN model.

To the best of our knowledge, no other publication has tackled protection of both trained NN models and data on decentralized and untrusted systems.

## 1.3 Proposed Solution

In this paper we propose a solution to **protect** both the **IP of trained NN**, **input data** and **output inference**, leveraging on FHE. Once trained, the parameters of the trained NN model are encrypted homomorphically. The resulting encrypted NN can be deployed on potentially insecure decentralized systems, while preserving the trained NN model and mitigating risk of reverse engineering. Inference can still be carried out over the homomorphically encrypted DNN, inserting homomorphically encrypted data and producing homomorphically encrypted predictions. Confidentiality of both trained NN, input data and inference results are therefore guaranteed.

This paper is organized as follows: section 2 provides an overview of our solution and the use case. Section 3 details the fundamentals of our approach. In section 4, we present the architecture and processes, concluding with future steps in section 5.

## 2 NEURAL NETWORK INTELLECTUAL PROPERTY PROTECTION SYSTEM

### 2.1 Overview

Our system is structured in 4 blocks (Figure 1):

1. **NN Training:** during this phase, unencrypted data is used to train the NN. Alternatively, we can import an already trained NN.
2. **Encryption of Trained NN:** once trained, the NN is protected, encrypting all parameters comprised in the model.

3. **Inference on Decentralized Systems:** the encrypted NN can be deployed on decentralized systems for DL inference, protecting its IP .
4. **Inference Decryption:** an encrypted NN produces encrypted inference, to be decrypted only by the owner of the trained NN.

### 2.2 Use Case

In this paper, we illustrate our approach with **video surveillance** use case for **risk prevention in public spaces**. Nowadays, cities are equipped with video surveillance infrastructure, where video stream is manually monitored and analyzed by police officers. This is time-consuming, costly and with questionable efficiency, thus cameras end up being used a posteriori to review incident. Indeed, smart cities rely on video-protection infrastructure to improve secure early detection of incidents in public spaces (e.g., early detection of terrorist attacks, abnormal crowd movement). By empowering cameras with deep learning capabilities on the edge, cameras evolve into multi-function sensors. Pushing the computation to where the data is being obtained substantially reduces communication overhead. This way, cameras can provide analytics and feedback, shifting towards a smart city cockpit.

With such approach, video management shifts from sole protection to versatile monitoring. These cameras has not only a simple - but essential - security role. It can also measure in real time the pulse of the agglomeration throughout vehicle flows and people who use them to redefine mobility, reduce public lighting costs, smooth traffic flow, etc.

## 3 FUNDAMENTALS OF IP PROTECTION

### 3.1 Homomorphic Encryption

While preserving data privacy, Homomorphic Encryption (HE) schemes allow certain computations on encrypted data without revealing neither its inputs nor its internal states. (Gentry, 2009) first proposed a Fully Homomorphic Encryption (FHE) scheme, which theoretically could compute any kind of function, but it was computationally intractable.

FHE evolved into more efficient techniques like Somewhat/Leveled Homomorphic Encryption SHE/LHE, which preserve both addition and multiplication over encrypted data. Similar to asymmetric encryption, during *KeyGen* a public key (*pub*) is generated for encryption, and a private key (*priv*), for decryption. Encrypted operations hold:

$$\begin{aligned} Enc_{\mathbf{pub}}(a*x+b) \equiv \langle a*x+b\rangle_{\mathbf{pub}} = \\ \langle a*x\rangle_{\mathbf{pub}} + \langle b\rangle_{\mathbf{pub}} = \langle a\rangle_{\mathbf{pub}} * \langle x\rangle_{\mathbf{pub}} + \langle b\rangle_{\mathbf{pub}} \end{aligned} \tag{1}$$

Modern implementations such as **HELib** (Halevi and Shoup, 2014) or **SEAL** (Laine and Player, 2016) include Single Instruction Multiple Data (SIMD), allowing multiple data to be stored in a single ciphertext and vectorizing operations. Hence, FHE protection implies vectorized additions and multiplications.

### 3.2 Data Encryption

The data encryption mechanism depends on the chosen scheme, the most efficient being BGV (Brakerski et al., 2011) and FV (Fan and Vercauteren, 2012). The encryption process is computationally slow, hence it can generate a bottleneck for the whole system, having a negative impact on overall performance.

$$\mathbf{X} \xrightarrow{encryption} ENC_{\mathbf{pub}}(X) = \langle X\rangle_{\mathbf{pub}} \tag{2}$$

### 3.3 Protecting Deep Neural Networks

Multiple architectures of deep neural networks have been designed addressing various domains. Our approach for IP protection is **agnostic about the architecture** of the Deep Neural Network. Nonetheless, in this paper we employ Deep Convolutional Neural Networks (DCNN), appropriate for video processing.

A DNN with $L$ layers is composed of:

1. An **input layer**, the tensor of input data $\mathbf{X}$
2. $L-1$ **hidden layers**, mathematical computations transforming $\mathbf{X}$ somewhat sequentially.
3. An **output layer**, the tensor of output data $\mathbf{Y}$.

We denote the output of layer $i$ as a tensor $\mathbf{A^{[i]}}$, with $\mathbf{A^{[0]}} = X$,and $\mathbf{A^{[L]}} = Y$. Tensors can have different sizes and even different number of dimensions. Layers inside a NN can be categorized as:

- *Linear*: they only involve polynomial operations, and can be seamlessly protected using FHE, such as Fully Connected layer (FC), Convolutional layer (Conv), residual blocks, and mean pooling.
- *Non-linear*, they include other operations (max, exp, division), and must be converted into sums and multiplications. E.g.: Activation Functions, Batch Normalization, max pooling...

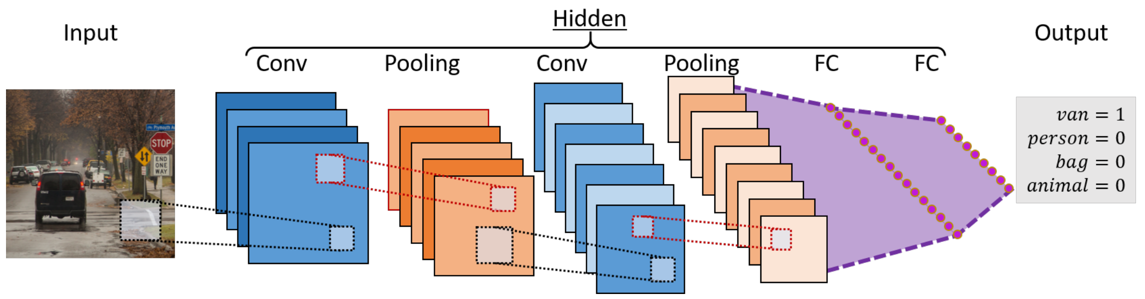

Figure 2: Example of architecture in a Deep Convolutional Neural Network.

Selecting a DNN **architecture** involves choosing: the number, types, order and size of the layers. An example of DCNN architecture is shown in Figure 2:

$$[Conv \rightarrow Pool]^n \rightarrow [FC]^m$$

Generally, DNN are designed mimicking well known architectures such as LeNet (LeCun et al., 2015), VGGNet(Simonyan and Zisserman, 2014) or ResNet (He et al., 2016), de-facto standards for object recognition and image classification.

In pursuance of full protection fors any given DNN, each layer needs to protect its underlying operations.

### 3.3.1 Fully Connected Layer (FC)

Also known as Dense Layer, it is composed of N parallel *neurons*, performing a $\mathbb{R}^1 \rightarrow \mathbb{R}^1$ transformation (Figure 3). We will define:

$\mathbf{a^{[i]}} = \left[a_0^{[i]} \dots a_k^{[i]} \dots a_N^{[i]}\right]^T$ as the output of layer $i$;

$\mathbf{z^{[i]}} = \left[z_0^{[i]} \dots z_k^{[i]} \dots z_N^{[i]}\right]^T$ as the linear output of layer $i$;

($\mathbf{z^{[i]}} = \mathbf{a^{[i]}}$ if there is no activation function)

$\mathbf{b^{[i]}} = \left[b_0^{[i]} \dots b_k^{[i]} \dots b_N^{[i]}\right]^T$ as the bias of layer $i$;

$\mathbf{W^{[i]}} = \left[\mathbf{w_0^{[i]}} \dots \mathbf{w_k^{[i]}} \dots \mathbf{w_N^{[i]}}\right]^T$ as the weights of layer $i$.

Neuron $k$ performs a linear combination of the output of the previous layer $\mathbf{a^{[i-1]}}$ multiplied by the weight vector $\mathbf{w_k^{[i]}}$ and shifted with a bias scalar $b_k^{[i]}$, obtaining the linear combination $z_k^{[i]}$:

$$z_k^{[i]} = \left(\sum_{l=0}^{M} w_k^{[i]}[l] * a_l^{[i-1]}\right) + b_k^{[i]} = \mathbf{w_k^{[i]}} * \mathbf{a^{[i-1]}} + b_k^{[i]} \tag{3}$$

Vectorizing the operations for all the neurons in layer $i$ we obtain the dense layer transformation:

$$\mathbf{z^{[i]}} = \mathbf{W^{[i]}} * \mathbf{a^{[i-1]}} + \mathbf{b^{[i]}} \tag{4}$$

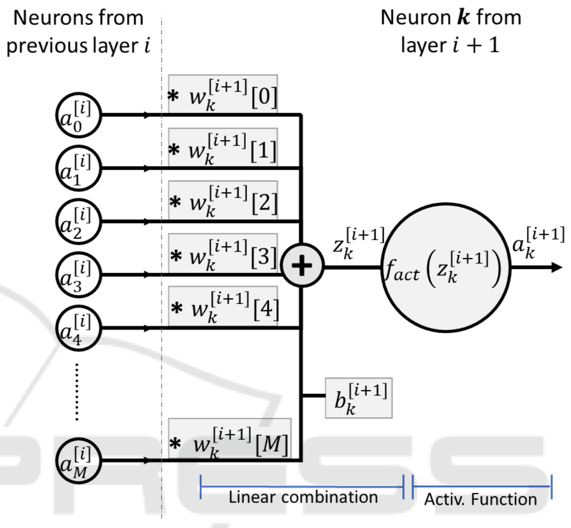

Figure 3: FC with activation for neuron $k$.

**Protecting FC Layer.** Since FC is a linear layer, it can be directly computed in the encrypted domain using additions and multiplications. Vectorization is achieved straightforwardly:

$$\left\langle \mathbf{z^{[i]}} \right\rangle_{\mathbf{pub}} \equiv \left\langle \mathbf{W^{[i]}} * \mathbf{a^{[i-1]}} + \mathbf{b^{[i]}} \right\rangle_{\mathbf{pub}}$$
$$= \left\langle \mathbf{W^{[i]}} \right\rangle_{\mathbf{pub}} * \left\langle \mathbf{a^{[i-1]}} \right\rangle_{\mathbf{pub}} + \left\langle \mathbf{b^{[i]}} \right\rangle_{\mathbf{pub}} \tag{5}$$

### 3.3.2 Activation Function

Activation functions are the major source of non-linearity in DNNs. They are performed element-wise ($\mathbb{R}^0 \rightarrow \mathbb{R}^0$, thus easily vectorized), and generally located after linear transformations (FC, Conv). All activation functions are positive monotonic.

$$a_k^{[i]} = f_{act}\left(z_k^{[i]}\right) \tag{6}$$

- *Rectifier Linear Unit (ReLU)* is currently considered as the most efficient activation function for DL. Several variants have been proposed, such as Leaky ReLU(Maas et al., 2013), ELU(Clevert

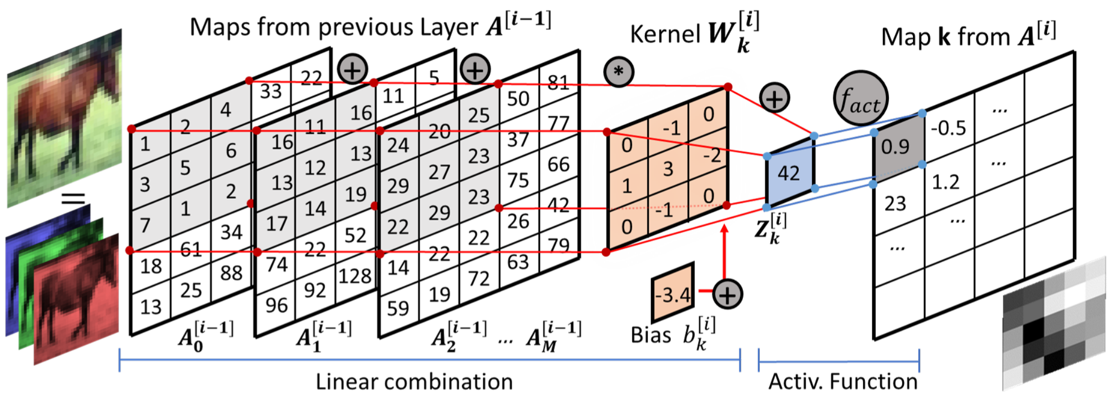

Figure 4: Conv layer with activation for map $k$.

et al., 2015) or its differentiable version *Softplus*.

$$\begin{aligned} ReLU(z) &= z^+ = max(0,z) \\ Softplus(z) &= log(e^z+1) \end{aligned} \tag{7}$$

- *Sigmoid* $\sigma$ The classical activation function. Its efficiency has been debated in the DL community.

$$Sigmoid(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{8}$$

- *Hyperbolic Tangent (tanh)* is currently being used in the industry because it is easier to train than ReLU: it avoids having any inactive neurons and it keeps the sign of the input.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{9}$$

**Protecting Activation Functions.** Due to its innate non-linearity, they need to be approximated with polynomials. (Gilad-Bachrach et al., 2016) proposed using only $\sigma(z)$ approximating it with a square function. (Chabanne et al., 2017) used Taylor polynomials around $x = 0$, studying performance based on the polynomial degree. (Hesamifard et al., 2017) approximate instead the derivative of the function and then integrate to obtain their approximation. One alternative would be to use Chebyshev polynomials.

### 3.3.3 Convolutional Layer (Conv)

Conv layers constitute a key improvement for image recognition and classification using NNs. The $\mathbb{R}^{2|3} \to \mathbb{R}^{2|3}$ linear transformation involved is **spatial convolution**, where a 2D $s*s$ filter (a.k.a. *kernel*) is multiplied to the 2D input image in subsets (*patches*) with size $s*s$ and in defined steps (**strides**), then added up and then shifted by a bias (see Figure 4). For input data with several channels or *maps* (e.g.: RGB counts as 3 channels), the filter is applied to the same patch of each map and then added up into a single value of the output image (cumulative sum across maps). A map in Conv layers is the equivalent of a neuron in FC layers. We define:

$\mathbf{A_k^{[i]}}$ as the map $k$ of layer $i$;
$\mathbf{Z_k^{[i]}}$ as the linear output of map $k$ of layer $i$;
($\mathbf{Z_k^{[i]}} = \mathbf{A_k^{[i]}}$ in absence of activation function)
$b_k^{[i]}$ as the bias value for map $k$ in layer $i$
$\mathbf{W_k^{[i]}}$ as the $s*s$ filter/kernel for map $k$.

This operation can be vectorized by smartly replicating data (Ren and Xu, 2015). The linear transformation can be expressed as:

$$\mathbf{Z_k^{[i]}} = \left( \sum_{m=0}^{M\ maps} \mathbf{A_m^{[i-1]}} \oplus \mathbf{W^{[i]}}_k \right) + b_k^{[i]} \tag{10}$$

**Protecting Convolutional Layers.** Convolution operation can be decomposed in a series of vectorized sums and multiplications over patches of size $s*s$. :

$$\begin{aligned} &\left\langle \mathbf{Z_k^{[i]}} \right\rangle_{\mathbf{pub}} = \left\langle \left( \sum_{m=0}^{M\ maps} \mathbf{A_m^{[i-1]}} \oplus \mathbf{W_k^{[i]}} \right) + b_k^{[i]} \right\rangle_{\mathbf{pub}} = \\ &\sum_{m=0}^{M\ maps} \left\langle \mathbf{A_m^{[i-1]}} \oplus \mathbf{W_k^{[i]}} \right\rangle_{\mathbf{pub}} + \left\langle b_k^{[i]} \right\rangle_{\mathbf{pub}} = \\ &\left\{ \sum_{m=0}^{M} \left\langle \mathbf{A_m^{[i-1]}}[j] \right\rangle_{\mathbf{pub}} * \left\langle \mathbf{W^{[i]}}_k \right\rangle_{\mathbf{pub}} \right\}_{[s*s]} + \left\langle b_k^{[i]} \right\rangle_{\mathbf{pub}} \end{aligned} \tag{11}$$

### 3.3.4 Pooling Layer

This layer reduces the input size by using a packing function. Most commonly used functions are **max** and **mean**. Similarly to convolutional layers, pooling layers apply their packing function to patches (subsets) of the image with size $s*s$ at strides(steps) of a defined number of pixels, as depicted in Figure 5.

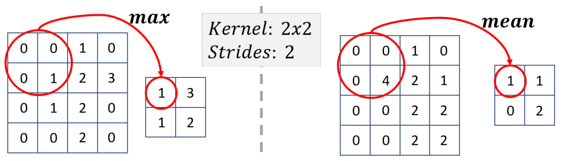

Figure 5: Max and Mean packing for Pooling layers.

**Protecting Pooling Layer.** *Max* can be approximated by the sum of all the values in each patch of size $s*s$, which is equivalent to scaled *mean* pooling. *Mean* pooling can be scaled (sum of values) or standard (multiplying by $1/N$). By employing a flattened input, pooling becomes easily vectorized.

### 3.3.5 Other Techniques

- **Batch Normalization (BN)** reduces of the range of input values by 'normalizing' across data batches: subtracting mean and dividing by standard deviation. BN also allows finer tuning using trained parameters $\beta$ and $\gamma$ ($\varepsilon$ is a small constant used for numerical stability).

$$a_k^{[i+1]} = BN_{\gamma,\beta}(a_k^{[i]}) = \gamma * \frac{a_k^{[i]} - E[a_k^{[i]}]}{\sqrt{Var[a_k^{[i]}] + \varepsilon}} + \beta \quad (12)$$

**Protection of BN** is achieved by treating division as the inverse of a multiplication.

$$\left\langle a_k^{[i+1]} \right\rangle_{\mathbf{pub}} = \langle\gamma\rangle_{\mathbf{pub}} * \left( \left\langle a_k^{[i]} \right\rangle_{\mathbf{pub}} - \left\langle E[a_k^{[i]}] \right\rangle_{\mathbf{pub}} \right) * \left\langle \frac{1}{\sqrt{Var[a_k^{[i]}] + \varepsilon}} \right\rangle_{\mathbf{pub}} + \langle\beta\rangle_{\mathbf{pub}} \quad (13)$$

- **Dropout and Data Augmentation** only affect training procedure. They don't require protection.
- **Residual Block** is an aggregation of layers where the input is added unaltered at the end of the block, thus allowing the layers to learn incremental ('residual') modifications (Figure 6).

$$\mathbf{A^{[i]}} = \mathbf{A^{[i-1]}} + ResBlock\left(\mathbf{A^{[i-1]}}\right) \quad (14)$$

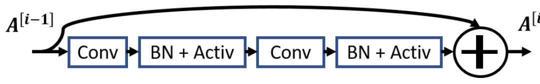

Figure 6: Example of a possible Residual Block.

**Protection of ResBlock** is achieved by protecting the sum and the layers inside ResBlock:

$$\left\langle \mathbf{A^{[i]}} \right\rangle_{\mathbf{pub}} = \left\langle \mathbf{A^{[i-1]}} \right\rangle_{\mathbf{pub}} + \left\langle ResBlock\left(\mathbf{A^{[i-1]}}\right) \right\rangle_{\mathbf{pub}} \quad (15)$$

## 3.4 Model Training and Outcome

Training is data and computationally intensive, performed by means of a backpropagation algorithm to gradually optimize the network loss function. It is also possible to reuse a previously trained model and apply fine tuning. As a result you get a trained **model**:

- Weights **W** and biases **b** in FC and Conv layers.
- $\mathbf{E[A]}$, $\frac{\mathbf{1}}{\sqrt{\mathbf{Var[A]}}}$, $\beta$ and $\gamma$ parameters in BN.

Those constitute the secrets to be kept when deploying a NN to decentralized systems. We focus solely on protecting IP of the **model**, leaving protection of the **architecture** out of the scope of this paper.

## 3.5 Inference Decryption

The decryption of the last layer's output **Y** is simply performed utilizing the private encryption key, as in standard asymmetric encryption schemes:

$$\left\langle \mathbf{A^{[L]}} \right\rangle_{\mathbf{pub}} \xrightarrow{decryption} DEC_{\mathbf{priv}}\left( \left\langle \mathbf{A^{[L]}} \right\rangle_{\mathbf{pub}} \right) = \mathbf{Y} \quad (16)$$

## 4 ARCHITECTURE

In this section we outline the architecture and information flows in our IP protection system, whose decomposition can be seen in Figure 7.

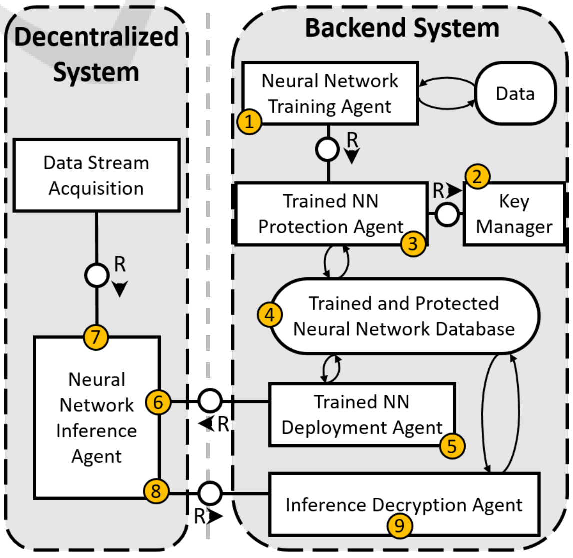
Decentralized System
Backend System
Neural Network Training Agent
Data
1
R
2
R
Trained NN Protection Agent
Key Manager
3
Data Stream Acquisition
R
Trained and Protected Neural Network Database
4
7
Neural Network Inference Agent
6
R
Trained NN Deployment Agent
5
8
R
Inference Decryption Agent
9

Figure 7: Activity Diagram in our solution.

**Encryption of Trained NN** ① ② ③ ④
In the backend, a NN is trained within a *NN Training Agent*. The outcome of the training (NN architecture and parameters) is pushed to the *Trained NN Protection Agent*. Alternatively, an already trained NN can be imported directly into the *Protection Agent*.

The *NN Protection Agent* generates a Fully Homomorphic key pair from the *Key Generator* component. The DNN is then encrypted and stored together with its homomorphic key pair in the *Trained and Protected NN Database*.

**Deployment of Trained and Protected NN** ⑤
At the deployment phase, the *Trained NN Deployment Agent* deploys the NN on decentralized systems, together with its public key.

**NN Inference Processing** ⑥ ⑦ ⑧ ⑨
On the decentralized system, data is collected by a *Data Stream Acquisition* component, and forwarded to the *NN Inference Agent*. Encrypted inferences are sent to the *Inference Decryption Agent* for their decryption using the private key associated to the NN.

IP of the NN, together with the computed inferences, is protected from any disclosure on the decentralized system throughout the entire process.

## 4.1 Sequential Processes

### 4.1.1 Encryption of Trained NN

Once a Neural Network is trained or imported, we encrypt all its parameters, using the Protected NN DataBase to store it and handle Homomorphic Keys (Figure 8).

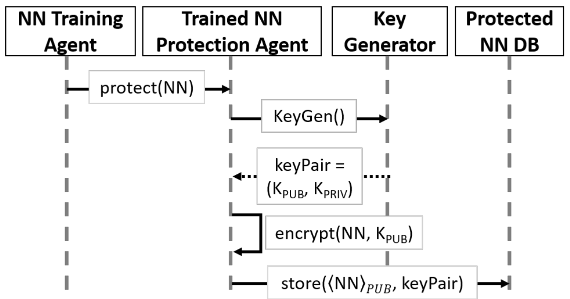

Figure 8: Sequence diagram of Trained NN Encryption.

### 4.1.2 Deploy Trained and Protected NN

The newly trained and protected deep neural network is deployed on the decentralized systems, including:

1. Network architecture;
2. Network model: Encrypted parameters;
3. Public encryption key.

### 4.1.3 Encrypted Inference

On the decentralized system, data is collected and injected into the deployed NN. We must encrypt $\mathbf{A}^{[\mathbf{0}]} = \mathbf{X}$ with the public encryption key associated to the deployed NN (Figure 9).

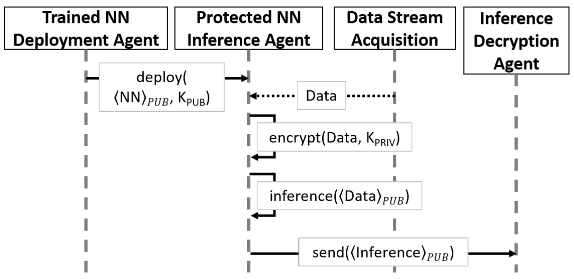

Figure 9: Sequence diagram of inference processing.

### 4.1.4 Inference Decryption

Encrypted inferences are sent to backend, together with an identifier of the NN used for the inference. The inference is homomorphically decrypted using the mapping private decryption key (Figure 10).

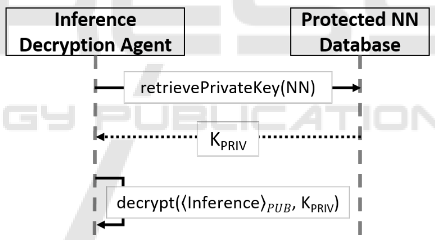

Figure 10: Sequence diagram of inference decryption.

## 5 CONCLUSION

This paper elaborates on a solution for the protection of Intellectual Property of decentralized Deep Neural Networks. Leveraging on Fully Homomorphic Encryption, we encrypt trained DNN, while preserving the confidentiality of input data and resulting inferences. This approach requires the modification of DNN to use linear approximation of activations functions, together with the decomposition of all operations into sums and multiplications, and encryption of input data at inference phase.

As future work, we will evaluate our approach on a real smart city use case. Overall performance of the system will be studied and compared with its unencrypted version. In that context, we consider the impact of all operations performed on the backend as negligible, including encryption of DNN, or decryption of inferences. Nevertheless, considering the resource restriction on decentralized systems, encryption of input data as well as encrypted computations are expected to have a major impact on the performance of the overall system.

## REFERENCES

Augasta, M. G. and Kathirvalavakumar, T. (2012). Reverse engineering the neural networks for rule extraction in classification problems. *Neural processing letters*, 35(2):131–150.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2011). Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277.

Chabanne, H., de Wargny, A., Milgram, J., Morel, C., and Prouff, E. (2017). Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Cramer, R., Damgård, I. B., et al. (2015). *Secure multiparty computation*. Cambridge University Press.

Fan, J. and Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144.

Floares, A. G. (2008). A reverse engineering algorithm for neural networks, applied to the subthalamopallidal network of basal ganglia. *Neural Networks*, 21(2-3):379–386.

Gentry, C. (2009). *A fully homomorphic encryption scheme*. Stanford University.

Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. (2016). Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Graepel, T., Lauter, K., and Naehrig, M. (2012). Ml confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*, pages 1–21. Springer.

Halevi, S. and Shoup, V. (2014). Algorithms in helib. In *International cryptology conference*, pages 554–571. Springer.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hesamifard, E., Takabi, H., and Ghasemi, M. (2017). Cryptodl: Deep neural networks over encrypted data. *CoRR*, abs/1711.05189.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Laine, K. and Player, R. (2016). Simple encrypted arithmetic library-seal (v2. 0). Technical report, Technical report, September.

LeCun, Y. et al. (2015). Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, page 20.

Liu, J., Juuti, M., Lu, Y., and Asokan, N. (2017). Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–631. ACM.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3.

Mohassel, P. and Zhang, Y. (2017). Secureml: A system for scalable privacy-preserving machine learning. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 19–38. IEEE.

Ren, J. S. and Xu, L. (2015). On vectorization of deep convolutional neural networks for vision tasks. In *AAAI*, pages 1840–1846.

Shokri, R. and Shmatikov, V. (2015). Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321. ACM.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. (2016). Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618.

Uchida, Y., Nagai, Y., Sakazawa, S., and Satoh, S. (2017). Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 269–277. ACM.