SAPIENZA

NEURAL NETWORK FINAL PROJECT

A Python implementation of CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy

---

Author:
Marzio MONTICELLI
(1459333)

Examiner:
Aurelio UNCINI

Supervisor:
Simone SCARDAPANE

A final project submitted in fulfillment of the requirements for the degree of Engineering in Computer Science

in the

Neural Networks for Data Science Applications
Ingengeria informatica, automatica e gestionale "Antonio Ruberti"

February 26, 2020

Declaration of Authorship

I, Marzio MONTICELLI (1459333), declare that this final project titled, "A Python implementation of CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy" and the work presented in it are my own. I confirm that:

- Where any part of this project has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project is entirely my own work.
- I have acknowledged all main sources of help.
- Where the project is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:
Marzio MONTICELLI (1459333)

Date:
February 26, 2020

"Success in creating AI would be the biggest event in human history. Unfortunately, it might also be the last, unless we learn how to avoid the risks."

Stephen Hawking

SAPIENZA

Abstract

Ingegneria dell'informazione, informatica e statistica
Ingengeria informatica, automatica e gestionale "Antonio Ruberti"

Engineering in Computer Science

A Python implementation of CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy

by Marzio MONTICELLI (1459333)

This project is about an implementation of the paper by Microsoft Research group in collaboration with Princeton University called **"CryptoNets: Applying Neural Network to Encrypted Data with High Throughput and Accuracy"** by Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Micheal Naehrig and John Wernsing published on February 24, 2016. The paper is about applying machine learning to a problem involves medical, financial, or other types of sensitive data requiring accurate predictions and careful attention to maintaining data privacy and security. In their work a method to convert learned neural networks to CryptoNets is presented. This project is about an implementation written in Python of the functionalities presented in CryptoNets paper. Results presented in this work are elaborated on the **MNIST optical character recognition tasks** and are compared to the results from the original paper so that to demonstrate the quality of both throughput and accuracy with respect to the results from the original work by Microsoft Research group…

Acknowledgements

I would like to express my sincere gratitude to my supervisor Simone Scardapane for his patience, motivation, enthusiasm, and knowledge.

Contents

List of Figures

List of Tables

List of Abbreviations

AES	**A**dvanced **E**ncryption **S**tandard
BVG	**B**rakerski-**G**entry-**V**aikuntanathan scheme
CN	**C**rypto**N**ets
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CRT	**C**hinese **R**emainder **T**heorem
DL	**D**eep **L**earning
FHE	**F**ully **H**omomorphic **E**ncryption
GPU	**G**raphics **P**rocessing **U**nit
HE	**H**omomorphic **E**ncryption
HElib	**H**omomorphic **E**ncryption **lib**rary
LWE	**L**earning **W**ith **E**rrors
ML	**M**achine **L**earning
MNIST	**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology
NN	**N**eural **N**etwork
OS	**O**perating **S**ystem
PYFHEL	**PY**thon **F**or **H**momorphic **E**encryption **L**ibrary
RLWE	**R**ing-**L**earning **W**ith **E**rrors
SEAL	**S**imple **E**ncrypted **A**rithmetic **L**ibrary
SIMD	**S**ingle **I**instruction **M**ultiple **D**ata
VM	**V**irtual **M**achine

Chapter 1

Environment And Installation

1.1 The Environment

This project was developed and built on **Linux** but you can use any operating system compatible with the specifications showed in next sections. If your system is not compatible with the specifications that follows than the code provided with this document should be unstable and you will be unable to run and test it on your local machine. Before starting the execution of the program please refer to the following sections and check your environment is compatible with the provided one. In sections that follows you will find a complete but synthetic explanation on how to setup and prepare your environment so that to run provided code successfully, in particular I suggest you to use **Anaconda Distribution** and **conda package manager** to install all the dependencies are required. Also pip package manager is required since part of the libraries are used in this work are not available in Anaconda Forge Cloud.

1.1.1 OS Environment

The code provided with this document was built and tested on a **Linux Mint 19.2 (Tina)- Ubuntu bionic** machine with **64-pc-linux-gnu (x86)** architecture. In particular the following libraries (or compatible) are required:

- **gcc (Ubuntu 7.4.0-1ubuntu1 18.04.1)** >= 7.4.0
- **g++ (Ubuntu 7.4.0-1ubuntu1 18.04.1)** >= 7.4.0
- **clang** >= 6.0.0-ubuntu2
- **anaconda (64-bit x86)** >= 3.7

If you use Linux than gcc and g++ languages must be supported by the GNU compiler and they must follow the C++17 specifications since this project use a bridge library for Microsoft SEAL.

"SEAL provides a set of encryption libraries that allow computations to be performed directly on encrypted data. SEAL enables software engineers to build end-to-end encrypted data storage and computation services where the customer never needs to share their key with the service."

Microsoft.com

FIGURE 1.1: The Linux Mint Logotype.

Also your **Python need to be compiled with a compiler following the C++17 specification**. Be sure your python version is compatibile with this specification since Pyfhel, the bridge library exposing SEAL functionalities, need to be successfully compiled within yout python envirnoment.

1.1.2 Using a Virtual Machine

If your system does not match the given specifications a good choice could be use a Virtual Machine (VM).
To have more information on what a VM is and how to install it on your computer you can refer to this awesome guide provided by IBM (https://www.ibm.com/cloud/learn/virtual-machines) in which you will find essential information on virtualization.

1.1.3 Python Environment

Anancoda3 is used to manage libraries, dependencies and Python environments. Through Anaconda Distribution you can easily create a custom environment containing all the main libraries used in this work. The following list contains all the libraries should be installed in your environment before executing the provided Cryptonet. To make sure your environment match the given requirements please go to next subsection which contains the list of libraries need to be manually installed before executing the code provided with this document. You will find all required step to setup your environment in the next section.

"Supported by a vibrant community of open-source contributors, Anaconda Distribution is the tool of choice for solo data scientists who want to use Python or R for scientific computing projects. Anaconda Distribution provides over 15 million users worldwide with access to thousands of the most popular data science and machine learning packages developed by the open-source community while we manage libraries, dependencies, and environments. "

Anaconda.com

FIGURE 1.2: The Anaconda Logotype

*) This list contains libraries are not strictly required but are usually installed in the base anaconda environment (as jupyter, notebook and spyder) and that have been used to test and compile project components.

- **libgccmutex** 0.1
- **tflowselect** 2.1.0
- **absl-py** 0.8.1
- **alabaster** 0.7.12
- **asn1crypto** 1.2.0
- **astor** 0.8.0
- **astroid** 2.3.2
- **attrs** 19.3.0
- **babel** 2.7.0
- **backcall** 0.1.0
- **blas** 1.0
- **bleach** 3.1.0
- **c-ares** 1.15.0
- **ca-certificates** 2019.10.16
- **cairo** 1.14.12
- **certifi** 2019.9.11
- **cffi** 1.13.1
- **chardet** 3.0.4
- **cloudpickle** 1.2.2
- **cryptography** 2.8
- **cudatoolkit** 10.0.130
- **cudnn** 7.6.4
- **cupti** 10.0.130
- **cycler** 0.10.0
- **cython** 0.29.13
- **dbus** 1.13.12
- **decorator** 4.4.1
- **defusedxml** 0.6.0
- **docutils** 0.15.2
- **entrypoints** 0.3
- **expat** 2.2.6
- **fontconfig** 2.13.0
- **freetype**2.9.1
- **fribidi** 1.0.5
- **gast** 0.2.2
- **glib** 2.63.1
- **gmp** 6.1.2
- **google-pasta** 0.1.7
- **graphite2** 1.3.13
- **graphviz** 2.40.1
- **grpcio** 1.16.1
- **gst-plugins-base** 1.14.0
- **gstreamer** 1.14.0
- **h5py**2.9.0
- **harfbuzz** 1.8.8
- **hdf5**1.10.4
- **icu** 58.2

- **idna** 2.8
- **imagesize** 1.1.0
- **importlibmetadata** 0.23
- **intel-openmp** 2019.4
- **ipykernel** 5.1.3
- **iPython** 7.9.0
- **iPythongenutils** 0.2.0
- **isort** 4.3.21
- **jedi**0.15.1
- **jeepney** 0.4.1
- **jinja2** 2.10.3
- **joblib** 0.14.0
- **jpeg** 9b
- **jsonschema** 3.1.1
- **jupyte-rclient** 5.3.4
- **jupyter-core** 4.6.1
- **keras-applications**1.0.8
- **keras-preprocessing** 1.1.0
- **keyring**18.0.0
- **kiwisolver** 1.1.0
- **lazy-object-proxy** 1.4.3
- **libedit** 3.1.20181209
- **libffi**3.2.1
- **libgcc-ng** 9.1.0
- **libgfortran-ng** 7.3.0
- **libpng** 1.6.37
- **libprotobuf**3.9.2
- **libsodium** 1.0.16
- **libstdcxx-ng**9.1.0
- **libtiff**4.1.0
- **libuuid** 1.0.3
- **libxcb** 1.13
- **libxml2** 2.9.9
- **markdown** 3.1.1
- **markupsafe** 1.1.1
- **matplotlib** 3.1.1
- **mccabe** 0.6.1
- **mistune** 0.8.4
- **mkl** 2019.4
- **mkl-service**2.3.0
- **mklfft** 1.0.15
- **mklrandom** 1.1.0
- **more-itertools** 7.2.0
- **nbconvert** 5.6.1
- **nbformat** 4.4.0
- **ncurses** 6.1
- **notebook** 6.0.2
- **numpy** 1.17.3
- **numpy-base** 1.17.3
- **numpydoc** 0.9.1
- **olefile** 0.46
- **openssl** 1.1.1d
- **opt-einsum** 3.1.0
- **packaging** 19.2
- **pandoc** 2.2.3.2
- **pandocfilters** 1.4.2
- **pango** 1.42.4
- **parso** 0.5.1
- **pcre**8.43
- **pexpect** 4.7.0
- **pickleshare** 0.7.5
- **pillow** 6.2.1
- **pip**19.3.1
- **pixman** 0.38.0
- **prometheus-client** 0.7.1
- **prompt-toolkit** 2.0.10
- **protobuf**3.9.2
- **psutil** 5.6.5
- **ptyprocess** 0.6.0
- **pycodestyle** 2.5.0
- **pycparser** 2.19
- **pydot** 1.4.1
- **pyfhel** 2.0.1
- **pyflakes** 2.1.1
- **pygments** 2.4.2
- **pylint** 2.4.3
- **pyopenssl** 19.0.0
- **pyparsing** 2.4.4
- **pyqt** 5.9.2
- **pyrsistent** 0.15.4
- **pysocks** 1.7.1
- **Python** 3.7.5
- **Python-dateutil** 2.8.1
- **pytz**2019.3
- **pyzmq** 18.1.0
- **qt** 5.9.7
- **qtawesome** 0.6.0
- **qtconsole** 4.5.5
- **qtpy** 1.9.0
- **readline** 7.0
- **requests** 2.22.0

- **rope** 0.14.0
- **scikit-learn** 0.21.3
- **scipy** 1.3.1
- **secretstorage** 3.1.1
- **send2trash** 1.5.0
- **setuptools** 41.6.0
- **sip**4.19.8
- **six**1.13.0
- **snowballstemmer** 2.0.0
- **sphinx** 2.2.1
- **sphinxcontrib-applehelp** 1.0.1
- **sphinxcontrib-devhelp** 1.0.1
- **sphinxcontrib-htmlhelp** 1.0.2
- **sphinxcontrib-jsmath**1.0.1
- **sphinxcontrib-qthelp**1.0.2
- **sphinxcontrib-serializinghtml** 1.1.3
- **spyder** 3.3.6
- **spyder-kernels** 0.5.2
- **sqlite**3.30.1
- **tensorboard**2.0.0
- **tensorflow** 2.0.0
- **tensorflow-base** 2.0.0
- **tensorflow-estimator** 2.0.0
- **tensorflow-gpu** 2.0.0
- **termcolor** 1.1.0
- **terminado** 0.8.2
- **testpath** 0.4.2
- **tk** 8.6.8
- **tornado** 6.0.3
- **traitlets** 4.3.3
- **urllib3** 1.24.2
- **wcwidth** 0.1.7
- **webencodings** 0.5.1
- **werkzeug** 0.16.0
- **wheel** 0.33.6
- **wrapt** 1.11.2
- **wurlitzer** 1.0.3
- **xz** 5.2.4
- **zeromq** 4.3.1
- **zipp** 0.6.0
- **zlib**1.2.11
- **zstd** 1.3.7

1.1.4 Minimum Requirements

This is the list of packages need to be manually installed in your Python3 environment so that to be able to execute the provided NN. Be sure to have at least this set of libraries before completing the installation and compile the provided code.

- **keras-applications** >= 1.0.8 » keras
- **keras-preprocessing** >= 1.1.0 » keras
- **matplotlib** >= 3.1.1
- **numpy** >= 1.17.3
- **pillow** >= 6.2.1
- **pyfhel** >= 2.0.1
- **Python** >= 3.7.5
- **scikit-learn** >= 0.21.3
- **tensorflow-gpu** >= 2.0.0 » tensorflow-gpu
- **tensorboard** >= 2.0.0 » tensorflow-gpu

1.2 Installation

This section contains a little guide on how to setup your environment so that to run the provided CN as well as to print your own results on a different data-set. As previously stated I suggest you to use the Anaconda Distribution since it can be helpful to easily setup a customized environment containing all requirements stated in the previous section.

1.2.1 Anaconda

Once you are sure your system match the requirements contained in **OS Environment** section you are ready to install Anaconda.
You can download Anaconda from the official site (https://www.anaconda.com) : be sure to download Anaconda3 (version 3.7 or grater) compatible with your system. If you need a complete guide to install and use Anaconda on your local machine, you can refer to the Anaconda official documentation (https://docs.anaconda.com); inside the official documentation you will find all the information you need to complete this task.

1.2.2 Preparing The Environment

Anaconda makes it easy to install TensorFlow, enabling your data science, machine learning, and artificial intelligence workflows.
This page (https://docs.anaconda.com/anaconda/user-guide/tasks/tensorflow) shows how to install TensorFlow with the conda package manager included in Anaconda and Miniconda. TensorFlow with conda is supported on 64-bit Windows 7 or later, 64-bit Ubuntu Linux 14.04 or later, 64-bit CentOS Linux 6 or later, and macOS 10.10 or later. The instructions are the same for all operating systems. No apt install or yum install commands are required.

First of all you need to create a new environment containing the current release of TensorFlow GPU (if supported). You can do it opening the Anaconda command prompt or PowerShell and typing the following two commands:

```
conda create -n cryptonets tensorflow-gpu
conda activate cryptonets
```

Your environment with TensorFlow GPU is now installed and ready to use.
If your machine does not support the TensorFlow GPU version, you can install the default version of the library since GPU functionalities are not strictly required.

1.2.3 Install Required Libraries

Once your environment contains TensorFlow (GPU) and is it active in the console you can easily install all the libraries are required by this project you can find in the previous section (**Minimum requirements**). You can install all but pyfhel library with the conda command as follow (be sure your cryptonets environment is active in your console) :

```
conda install keras
conda install matplotlib
...
conda install scikit-learn
```

If you need to force the installation of a specific library (matplotlib in this example) you can run the following command:

```
conda install matplotlib --force
```

Finally you need to install the latest release of **Pyfhel** (vrs 2.0.1 or grater) through pip package manager since it is not available in the Anaconda Cloud.

```
pip install Pyfhel
```

You can see the documentation related to Pyfhel in the official Github page.

1.2.4 Import Project Files

To complete the installation process you have to extract the provided folder in the directory created for your environment. In the following list you can find the documented structure of the uncompressed folder.

```
Cryptonets (the main directory)
├── main.py (run the project)
├── modules (all modules)
│   ├── cryptonet.py (the main cryptonet)
│   ├── dataset.py (data-sets elaboration)
│   ├── debug.py (debug networks)
│   ├── encodenet.py (NN encoded version)
│   ├── encryptednet.py (NN ecrypted version)
│   ├── exporter.py (export layers)
│   ├── model.py (the network)
│   ├── plainnet.py (NN plain version)
│   ├── util.py (utilities)
│   └── deprecated (deprecated files)
│       ├── encryption (deprecated - encrypted NN)
│       └── experiment (deprecated - experimental phase)
└── storage (processed files)
    ├── contexts (contexts with relative keys layers)
    ├── datasets (stored data-sets)
    ├── experiments (stored experiment results)
    ├── layers (exported layers)
    │   └── preprocessed (processed layers)
    └── models (exported models)
```

1.3 Note

If you prefer to use a different solution from Anconda you can follow the previous indications to install all the required libraries through the pip package manager. If you need to check if all the required libraries are installed in your environment you can do it through the following commands:

Using conda packages manager

```
conda list
```

Using pip packages manager

```
pip list
```

In both cases a full list of all the packages installed in your environment is printed.

Chapter 2

Neural Network

2.1 Background and Preliminaries

Emerging DL (deep learning) architectures and recurrent neural networks show a huge promise for artificial intelligence based application in fact have succeeded in many field including computer vision, speech and audio recognition, etc. . Commercial AI service providers as Google, Microsoft, and IBM have devoted a lot of effort to build deep learning models for various intelligence application bring convenience to daily life. This raises serious privacy concerns due to the risk of leakage of highly privacy-sensitive data. Such privacy concerns are hindering the applicability of neural network models for real world applications.

2.1.1 Neural Networks

Neural networks can be thought of as leveled circuits in which each level (also called layer) can be visualized as being stacked so that the bottom most layer is the input layer and the top most layer is the output of the network or the prediction. Neural networks commonly use several functions can be applied and computed at the nodes like **Weighted sum**, different kinds of **Pooling**, **Sigmoid**, **Rectified Linear**, etc. [0].
In a neural network each layer receives the data generated by its previous layer and outputs the processed data for the next layer. In particular, the raw data is encoded properly and fed into the first layer of a neural network, also known as the input layer. Then, these features from the raw data are gradually mapped to higher-level abstractions via the iterative update (a.k.a, feed-forward and back-propagation) in the intermediate (hidden) layers of the neural network until the convergence condition is achieved (e.g., the specified number of iteration). These mapping abstractions known as learned neural network model then can be used to predict the label in the last layer (i.e., the output layer)[1]. The hidden layers of a convolutional neural network typically consist of a series of convolutional layers that convolve with a multiplication or other dot product. Though the layers are colloquially referred to as convolutions, this is only by convention. Mathematically, it is technically a sliding dot product or cross-correlation. This has significance for the indices in the matrix, in that it affects how weight is determined at a specific index point... [2].

2.1.2 Application Scenario

Consider the scenario in which a federal clinic wants to train an AI model to help diagnose and it has to rely on a third party service. In this scenario there is the need to ensure privacy as well as to compute predictions based on the elaboration of

sensitive user data in speed and accurate way. The application of cryptography to AI models represent a potential solution to this and a variety of problems concerning data privacy and information security. The focal point is that all data elaborated in the cloud, owned by the service provider and related to clients sensible information, must be encrypted prior to its elaboration. This is possible only if input data in encrypted so that there is the persistence of well known mathematical properties without which the prediction of required results will be impossible. This project puts its roots in this idea and it is based on the application of feed-forward neural networks to encrypted data; To be more specific the main ingredients are neural networks and homomorphic encryption (HE): a **convolutional neural network** is applied to the encrypted input data it was made secure and private thanks to an **homomorphic encryption scheme**.

Fully homomorphic encryption, or simply homomorphic encryption, refers to a class of encryption methods envisioned by Rivest, Adleman, and Dertouzos already in 1978, and first constructed by Craig Gentry in 2009. Homomorphic encryption differs from typical encryption methods in that it allows computation to be performed directly on encrypted data without requiring access to a secret key. The result of such a computation remains in encrypted form, and can at a later point be revealed by the owner of the secret key. The security of the most practical homomorphic encryption schemes is based on the Ring-Learning With Errors (RLWE) problem, which is a hard mathematical problem related to high-dimensional lattices. Namely, the security assumption of these encryption schemes states that if the scheme can be broken efficiently, then the RLWE problem can be solved efficiently. A long line of peer-reviewed research confirming the hardness of the RLWE problem gives us confidence that these schemes are indeed at least as secure as any standardized encryption scheme.[3]

2.2 Network Architecture

In the original paper two networks are used: the first architecture is composed by 9 layers and it is used during the network training phase, while the simplified one is used only during the test phase; in particular it is composed by 5 layers are created starting from the first network by the integration of adjacent linear layers through their matrix multiplication.

In this project, differently from the original paper, the simplified architecture is used both in training and testing phases. The use of the simplified version helps both to reduce the time required to train the network and to speed up the computation of final results. Another important aspect to take into consideration is that the simplified architecture was restyled to match the constraints given by used technologies.

Table 2.1 illustrates the final network architecture: main differences between the paper network and the one in the table that follows, can be found in the use of the **softmax** function in place of the **sigmoid**. Softmax was preferred because of a strange behavior of the network even during the train step. Using the sigmoid function the network seems not to converge properly, maybe because of an internal multiplication by zero connected to the representation used by keras to compute the result (the approximation of some floating point numbers). This guide me to

TABLE 2.1: Network architecture

Layer	Input	Output	Description
0. (pre.) Average Max Pool	28x28	15x15	This layer reduces the image dimension by half and applies a padding on the left and the upper part of each images.
1. Convolution Layer	15x15	7x7x5	The convolution has windows, or kernels, of size 3x3, a stride of (2,2) and a mapcout of 5 (5 filters are used).
2. Flatten Layer	7x7x5	1x245	The input volume is flattened in a one-dimensional vector
3. Dense Layer (Squared)	1x245	1x100	Computes the dot product between the input and a weights matrix of dimension 245x100, squaring the value at each input node and summing the biases to the result.
4. Dense Layer (Squared)	1x100	1x10	Computes the dot product between the input and a weights matrix of dimension 100x10, squaring the value at each input node and summing the biases to the result.
5. Dense Layer (Softmax)	1x10	1x10	Computes the dot product between the input and a weights matrix of dimension 10x10, applying the softmax activation function at each input node and summing the biases to the result.

implement a more feasible solution using the well known softmax function with only a little decay in terms of output accuracy with respect to the original paper.

Another important change is the down-sampling of input data: in the original paper the MNIST data-set is used without manipulation; differently in this project input images are down-sampled thanks to the application of another layer before the input one. In particular, before to feed the network with the MNIST images, they are processed through an **Average Pool Layer** with kernel size and strides equal to 2. The resulting images were 2 times smaller and to handle this new input dimension the whole architecture was restyled. The down-sampling process of the input images and the final restyling of the network, have caused a 3% deterioration in terms of accuracy during the train step, price it was paid in hindsight, since this work is focused on the inference phase of the original paper.

Image 2.1 shows a graphical elaboration of the network in which each layer is represented as its input volume, output volume and operations are performed by each layer. As you can see the network is simple and, despite this, model accuracy is considered enough for the propose of this work.

Some experiments on the original model are conducted with the aim to increase its accuracy; Variations operated on the model will be taken into consideration in the section that follow. To anticipate such consideration, variations on the model consist partially in applying dropout technique as regularization method: during training, some number of layer outputs are randomly ignored or "dropped out." This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. With additional dropout layers

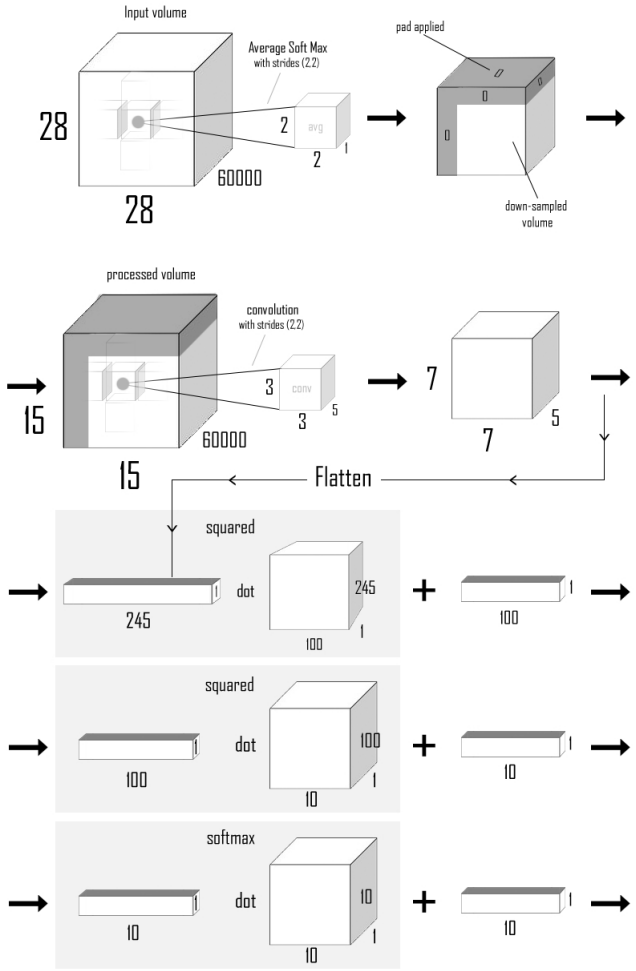


FIGURE 2.1: Volumes and operations characterizing the network.

the model acquire an accuracy close to the 99% in performing the optical character recognition task. On the other hand, without dropout regularization, model has achieved a maximum accuracy of 98%. The results will be presented in the next section are present in this work only as a complement to the main discussion.

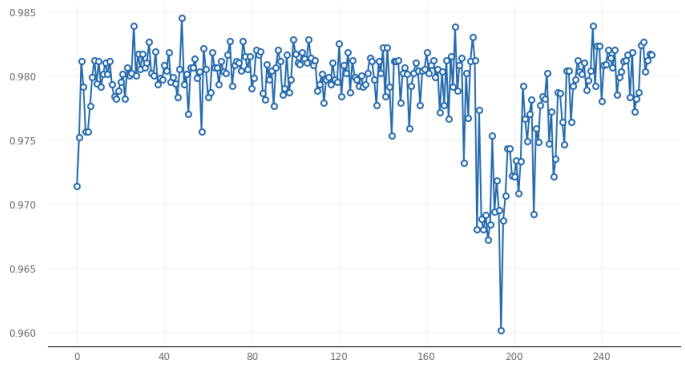
2.3 Network Optimization

In this section the optimization procedure and relative results are shortly presented since it is out the scope of this project. The first task performed was to understand data and to make some experiment on the model presented in the original paper. As explained before this elaboration was fundamental to improve the project core making it more feasible to the exempt computing power of the machine on which the network was developed.

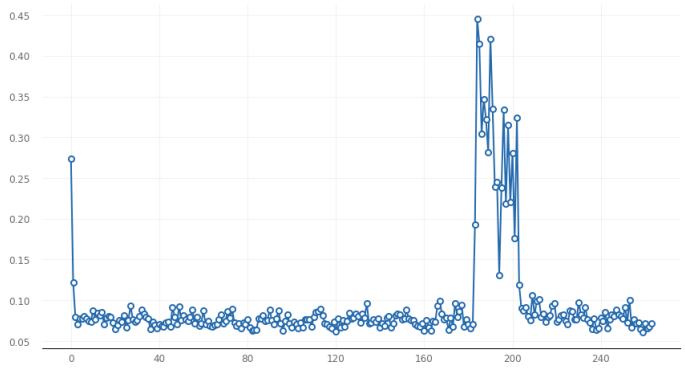
2.3.1 Hyperparameters Optimization

Since network architecture was very simple it was considered important to spend some times to make model accuracy better with respect to the machine on which

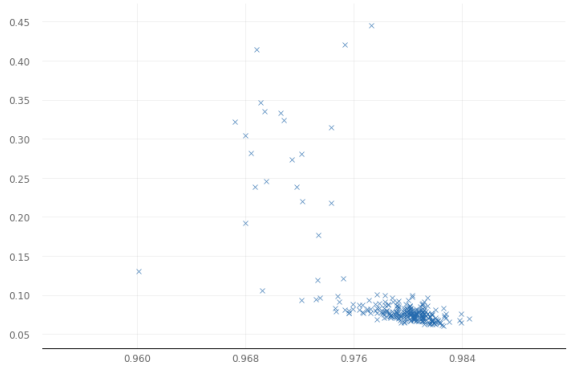
this project was developed: this means find best parameters for the model so that to reduce loss value and increase its accuracy. This process is done thanks to a library called **Talos** used to fully automate hyperparameter tuning and model evaluation. Talos library seems to be the best choice for this project because of developed model was implemented using the **Keras** library; Talos also simplifies the workflow making the optimization process simpler by decreasing the code complexity. To reflect this choice, developed model was changed both to make the hyperparametes selection easier, speeder and to generate plots was useful to analyze results with which final network was tuned.



(A) Accuracy



(B) Loss

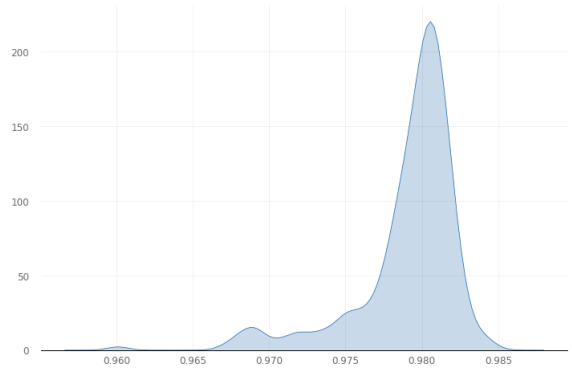


(C) Regression

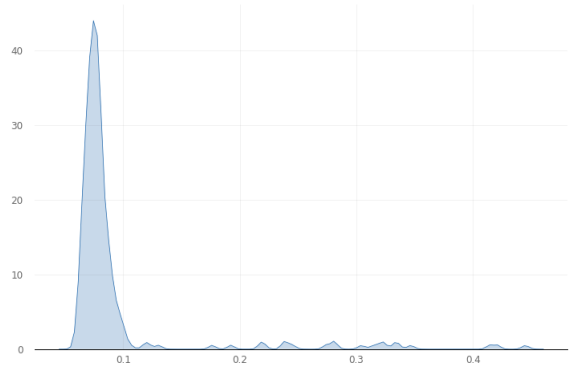
FIGURE 2.2: Estimated accuracy and loss of 264 experiments at the variation of 10 model hyperparameters

Plots 2.2a and 2.2b show the results after 264 experiments at the variation of 10 parameters among which: last activation function, optimizer, epochs number, batch

size, the dimension of the convolution kernel and strides, the probability of dropout. Best results are experienced using softmax as the last activation function, a batch size of 200, 50 training epochs and using a dropout in the visible layer with probability 0.2. The Stochastic Gradient Descent method, with a learning rate of 0.01, seems to be the best optimization choice. Figure 2.2c puts in comparison all accuracy values and relative loss: as you can see a lot of results are concentrated between 0.976 and 0.984 with a loss not grater than 0.1. This because at each experiment iteration, parameter variations are limited over the best results experienced in all previous runs so that to converge to the best configuration possible.



(A) Accuracy



(B) Loss

FIGURE 2.3: Kernel Density Estimation of 264 results at the variation of 10 model hyperparameters

This result it is also clear in Figures 2.3a and 2.3b in which the kernel density estimation are showed for both the accuracy and the loss value after 264 runs: the probability density function of both accuracy and loss guides to select parameters over specific intervals that are finally tuned and fixed so that to export the best model possible. Further experiments on the model was done, and in particular trying to minimize the loss without the additional dropout layer. The dropout layer creates complexities can't be easily handled with the encryption scheme on which this project is concentrated.

Figure 2.4 shows the complete histogram of the accuracy experienced for all the experiments conducted on this particular problem: on the y-axes the number of results per accuracy value is plotted and, as you can see, 98% of accuracy seems to be the best we can do on this machine (in mean) without further analysis. Despite this, collected results seems to be enough for the propose of this work. To complete the discussion about hyperparameters optimization and justify the choice to use a

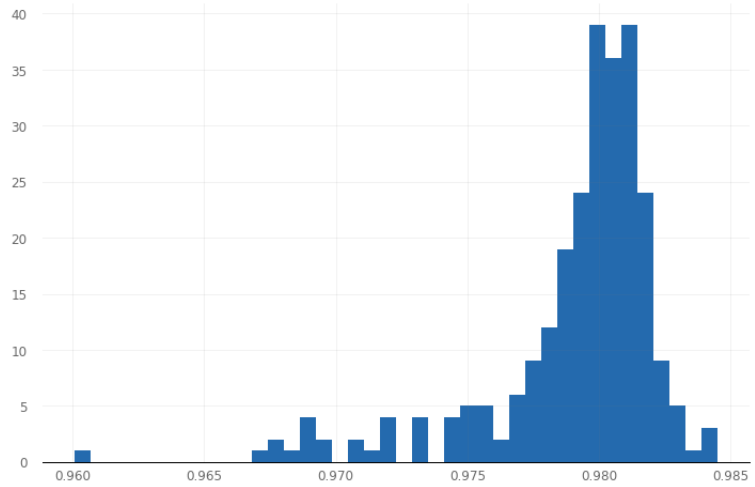
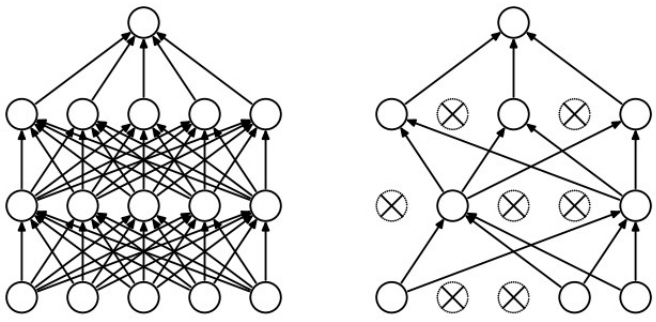


FIGURE 2.4: The complete accuracy histogram

dropout regularization method, it will be shortly taken into consideration in the next subsection. As explained before all the dropout layers are removed at the end of this process so that to make easier to implement the encrypted version of the network, causing obviously a slight reduction in the model accuracy. The optimal model experienced after the removal of dropout it is finally used as a good point to evaluate the encryption process it will be detailed presented in the next chapter.

2.3.2 The Dropout Regularization Method

Dropout prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5 [4].



(a) Standard Neural Net (b) After applying dropout.

FIGURE 2.5: Example of the dropout regularization.

During training, some number of layer outputs are randomly ignored or "dropped out." This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "view" of the configured layer. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust[4].

2.3.3 The Keras Model

Just below the class containing model implemented with Keras is showed. As you can see no dropout layers are used and main parameters are passed in a variable containing values resulting from the hyperparameters optimization phase taken into consideration in this section. Fit function is used by the class to actually fit the model and elaborate its history. Returned model contains both weights and biases elaborated during the training phase over MNIST training data containing 60000 samples. On the other hand model's accuracy is evaluated through the two sets xval and yval, test set and relative labels respectively.

```
from tensorflow import keras
from tensorflow.keras.layers import Activation
from tensorflow.keras import utils

def square_activation(x):
    return x**2

utils.get_custom_objects().update(
{'square_activation': Activation(square_activation)})

class Model:

    def fit(self, x_train, y_train, xval, yval,
            params = {
                'kernel_size': (3,3),
                'strides': (2,2),
                'last_activation': 'softmax',
                'optimizer': 'SGD',
                'loss': 'categorical_crossentropy',
                'batch_size': 200,
                'epochs': 50,
                'learning_rate': 0.01,
                'momentum': 0.9,
                'nesterov': False
            }):

        model = keras.models.Sequential()

        model.add(keras.layers.Conv2D(5, kernel_size=params['kernel_size'],
strides=params['strides'], input_shape=x_train[0].shape))
        model.add(keras.layers.Flatten())
        model.add(keras.layers.Dense(100, activation=square_activation))
        model.add(keras.layers.Dense(10, activation=square_activation))
```

```
        model.add(keras.layers.Dense(10, activation=params['last_activation']))

        sgd = keras.optimizers.SGD(lr=params['learning_rate'],
momentum=params['momentum'], nesterov=params['nesterov'])

        model.compile(optimizer=sgd,
                      loss=params['loss'],
                      metrics=['accuracy'])

    #    model.summary()

        history = model.fit(
            x_train, y_train,
            validation_data=[xval, yval],
            epochs=params['epochs'],
            batch_size=params['batch_size']
        )

        return history, model
```

Default parameters passed to the fit function also contain records connected to the SGD Nesterov momentum that seems no to increase the convergence to the optimal for this particular data even at the variation of the momentum. As for other parameters taken into consideration, different intervals are tested to optimize nesterov update but without a relevant increase of the model accuracy that gravitates around the 98% as showed in previous plots.

This preliminary model in used so that to test the final model implementation will be presented in next chapters after a little introduction of the used cryptographic scheme. In particular weights and biases computed by each layer was compared with relative outputs belonging to the plain version of the network on which the HE scheme is finally applied. Such new version was implemented to handle particularities of the cryptographic scheme is used

.

Chapter 3

Homomorphic Encryption

3.1 The Cryptographic scheme

Public-key cryptography, or asymmetric cryptography, is a cryptographic system that uses pairs of keys: public keys which may be disseminated widely, and private keys which are known only to the owner. The generation of such keys depends on cryptographic algorithms based on mathematical problems to produce one-way functions. Effective security only requires keeping the private key private; the public key can be openly distributed without compromising security [5].

Homorphic encryption (HE) is an asymmetric scheme based on an invertible function $\Phi(\bullet)$; such function can be inverted only with a specific private key generated when the cryptographic context is created.

(1) Given a value $x \in \mathbb{Z}$ on which the function $\Phi(\bullet)$ is defined the relation $\Phi^{-1}(\Phi(x)) = x$ must hold for each value $x \in \mathbb{Z}$ on which $\Phi(\bullet)$ is defined.

Moreover, differently from traditional cryptography, in this map both additive and multiplicative structure of the **rings** must be preserved so that to have the possibility to perform such operations over encrypted data.

Lets consider $x_1, x_2 \in \mathbb{Z}$, we have that $\Phi(x_1 + x_2) = \Phi(x_1) + \Phi(x_2)$ and $\Phi(x_1 \cdot x_2) = \Phi(x_1) \cdot \Phi(x_2)$, from (1) following relations must hold:

$$\Phi^{-1}(\Phi(x_1 + x_2)) = \Phi^{-1}((\Phi(x_1) + \Phi(x_2)) = x_1 + x_2$$

$$\Phi^{-1}(\Phi(x_1 \cdot x_2)) = \Phi^{-1}((\Phi(x_1) \cdot \Phi(x_2)) = x_1 \cdot x_2$$

For example, consider the map $\Phi(\bullet) : \mathbb{Z} \Rightarrow \mathbb{Z}_{13}$ such that $\Phi(x) := x(mod 13)$, $\Phi(\bullet)$ is a ring homomorphism between the ring $\mathbb{Z}$ and $\mathbb{Z}_{13}$.

3.1.1 Encryption Scheme

As explained in the original paper the exryption scheme maps plaintext messages from the ring $\mathbb{R}_t^n := \mathbb{Z}_t[x]/(x^n + 1)$ to the ring $\mathbb{R}_q^n := \mathbb{Z}_q[x]/(x^n + 1)$ and it is based on the following assumpions:

Given two random polynomials $f', g \in \mathbb{R}_q^n$ where $\mathbb{R}_q^n$ elements can be thought of a sets of the form $\{ p(x) + q(x)(x^n - 1) : q(x) \in \mathbb{Z}_q[x], p(x) \in \mathbb{Z}_q[x] \}$ and where the general $\mathbb{Z}_m$ is composed by integers modulo m can be thought of as sets of the form $\{ i + am : a \in \mathbb{Z} \}$ where i is an integer and where the set $\mathbb{Z}[x]$ is composed by polynomials with coefficients in a commutative ring $\mathbb{Z}$ that is itself a ring (a set on which both addition and multiplication are defined):

$$h := (tgf^{-1}) \text{ is the public key}$$

$$f := (tf' + 1) \text{ is the secret key}$$

Since not every element in $f', g \in \mathbb{R}_q^n$ is invertible, these steps are iterated until the corresponding f has an inverse and h can be computed. Given the public key h a message $m \in \mathbb{R}_t^n$ is encrypted by computing

$$c := [[q/t]m + e + hs]_q$$

where e and s are random noise polynomials in $\mathbb{R}_q^n$ with coefficients of small absolute value and with $[\bullet]_q$ is denoted the reduction of the coefficients of a modulo q to the symmetric interval of length q around 0.
While the decryption is done by computing:

$$m := [[\frac{t}{q} fc]]_t$$

where the product fc is computed in $\mathbb{R}_q^n$, coefficients are interpreted as integers, scaled by $\frac{t}{q}$, rounded to the nearest integers and finally interpreted modulo t.

3.1.2 Operations Over Encrypted Data

Since the original maps are relative to commutative rings it is possible to perform addition and multiplication over data encrypted according to the encryption scheme showed before. In particular, given two cipher text:

$$c_1 := [[q/t]m_2 + e_1 + hs_1]_q$$

$$c_2 := [[q/t]m_2 + e_2 + hs_2]_q$$

where m_1 and m_2 are two messages, it it possible to add them together in $\mathbb{R}_q^n$ to yield the encryption of $m_1 + m_2$ since:

$$c_1 + c_1 := ([q/t]m_1 + e_1 + hs_1) + ([q/t]m_2 + e_2 + hs_2) := [\frac{q}{t}](m_1 + m_2) + (e_1 + e_2) + h(s_1 + s_2)$$

with decrypts to $m_1 + m_2 \in \mathbb{R}_t^n$.

As addition, the multiplication between two cyphertext is possible by computing:

$$[\frac{t}{q} c_1 c_2] = [\frac{q}{t}](m_1 m_2) + e' + h^2 s_1 s_2$$

where e' is small noise term. It decrypts to $m_1 \cdot m_2$ but under the secret key $f^2 \in \mathbb{R}_q^n$. To decrypt the resulting cyphertext with the original secret key it necessary

to use a method called **relinerarization**:

since the multiplication of the two polynomials c_1 and c_2 scaled by $\frac{t}{q}$ ends up with a cyphertext consisting of 3 ring elements instead of 2 we need to rectify this phenomenon by reducing the dimensionality of the cyphertext up to two.

Let $c = [c_0, c_1, c_2]$ denote a degree 2 cyphertext, than we need to find $c' = [c'_0, c'_1]$ such that:

$$[c_0 + c_1 \cdot s + c_2 \cdot s^2]_q = [c'_0 + c'_1 \cdot s + r]_q$$

where r is still small term adding noise to the decryption process. The **relinearization key** is generated according to this concept trying to obtain a good approximation of the $c_2 \cdot s^2$ modulo q term with the price of adding a noise equal to r. It is clear that this additional noise has to be considered when relinearization is performed so that to be able to decrypt to the right value. For more details about homomorphic encryption and on the relinearization process it is possibile to consult the awesome paper from Junfeng Fan and Frederik Vercauteren [6].

3.1.3 Parameter Selection

The choice of n, q and t it is fundamental so that the amount of noise (added both in encryption and during cyphertexts relinearizations) is suitable by the cryptographic scheme. Since the cryptographic scheme is applied to a neural network computing its output through a series of additions, multiplications and other real functions on real values, we have to carefully consider the maximum amount of noise that a cyphertext can have and, as showed before, it depends directly from the selection of parameters q and t: in particular it is fundamental to select a q large enough to support the increased noise and a large t to prevent coefficients of the plaintext polynomials from reducing modulo t at any point during computation. Another important aspect is to choose a large n for security reason.

But consider this assumptions it is not enough: neural networks operate over real numbers while the atom construction in the HE schemes is on polynomials in $\mathbb{R}_t^n$. This means we need a method to map real numbers in polynomials in such a way to preserve addition and multiplication operations. In this project such mapping was done through the conversion of real numbers into fixed precision numbers which representation is used to convert them into polynomials with the coefficients given by their binary expansion. The encoding and relative decoding process is performed over tensors prior to layer encryption and after output decryption.

3.2 Pyfhel

As explained before the MNIST data-set is used to train and test the network but both train and test images are down-sampled to reduce the number of parameters used by each layer. In particular this reduction in size is operated so that to apply only one scheme in place of the two used in the original paper: this choice is also reflected by the fact we use a library called **Pyfhel** to create the cryptographic context with relative keys.

As we can read from its documentation, PYthon For Hmomorphic Encryption Libraries implements some functionalities of Homomorphic Encryption libraries such

as sum, multiplication, or scalar product in Python. Pyfhel uses a syntax similar to normal arithmetics (+,-,*). The library is useful both for simple Homomorphic Encryption Demos as well as for complex problems such as Machine Learning algorithms. It was developed by Alberto Ibarrondo and Laurent Gomez (SAP) in collaboration with EURECOM and it is built on top of Afhel, an Abstraction Hmomorphic Encryption Libraries in C++. Afhel serves as common API for SEAL, HElib and PALISADE backends; This project uses the v2.0.1 of Pyfhel library you can easily install as explained in the first chapter.

Pyfhel works with three main classes:

- **Pyfhel** : The main class expose all operations can be performed between cyphertexts and plaintexts are represented through dedicated classes. Pyfhel class is used both to actually generate the cryptographic context and keys can be easily saved and restored from the local environment.
- **PyPtxt** : It is used to represent plaintexts and to use them inside the cryptographic context created through the Pyfhel class
- **PyCtx** : It is used to represent cyphertexts so that to perform operations between them inside the cryptographic context.

3.2.1 Context Creation

The cryptographic context is created through the **contextGen** function and it is based on parameters, as well as integer, fractional and batch encoders. The HE context is required for any other function (encryption/decryption,encoding/decoding, operations): in particular this library is used only with batch encoding and it is possible only if selected t is prime and $t-1$ is multiple of $2*n$. Batch encoding process encodes a 1D tensor into a PyPtxt plaintext, it is based on the created context and it is possible to encode only 1D tensor with n elements at time.
For the same reason the batch encryption encrypt 1D tensor represented through the PyPtxt class previously encoded through the batch encoding process so that we can encrypt n elements at the same time within the same cyphertext. Batch encoding is operated thanks to the **Chinese Remainder Theorem** that states that:

An element $p \in \mathbb{R}$ is said to be prime if for every $f, g \in \mathbb{R}$ it is true that if p divides fg, than p divides at least one of f and g, moreover $\mathbb{R} \cong \prod_i \frac{\mathbb{R}}{p_i}$ when $p_1, \ldots, p_n$ are distinct primes.

In other words **CRT** states in particular that an element p can be uniquely represented by elements $p_1, \ldots, p_n$ such that $p_i \in \frac{\mathbb{R}}{p_i}$ and this allows breaking p into n smaller values $p_1, \ldots, p_n$ can be obviously used to reconstruct the original p. This consideration is important to understand how n elements can be encoded, encrypted and decrypted together. In particular this library is used, making an application of the **CRT**, to perform Single Instruction Multiple Data (**SIMD**) operations at no extra cost.
When Pyhel class is instantiated and contextGen function is called the library invokes the relative function inside the **Afseal** a C++ library that creates an abstraction over the basic functionalities of HElib as a Homomorphic Encryption library and SEAL.
HElib is a software library that implements homomorphic encryption (HE). Currently available is an implementation of the Brakerski-Gentry-Vaikuntanathan (BGV) scheme,

along with many optimizations to make homomorphic evaluation runs faster, focusing mostly on effective use of the Smart-Vercauteren ciphertext packing techniques and the Gentry-Halevi-Smart optimizations.

As we can read in [7] BGV is radically new approach to fully homomorphic encryption (FHE) that dramatically improves performance and bases security on weaker assumptions. A central conceptual contribution BVG is a new way of constructing leveled fully homomorphic encryption schemes (capable of evaluating arbitrary polynomial-size circuits), without Gentry's bootstrapping procedure. Specifically, it offers a choice of FHE schemes based on the learning with error (LWE) or ring-LWE(RLWE) problems that have 2^{λ} security against known attacks.

In HElib library the coefficient modulus is generated starting from both the security level is setted at initialization phase and t value. **Security Level** value is equivalent to **Advanced Encryption Standard** (AES) key length (AES is a subset of the Rijndael block cipher developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, who submitted a proposal to NIST during the AES selection process. Rijndael is a family of ciphers with different key and block sizes. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits [8]).

In particular to get a lower-bound on the parameter $N = \Phi(t)$ the following step are used:

- Each level in the modulus chain corresponds to $p = p/2$ bits (where we have one prime of this size, and all the others are of size p bits). When using DoubleCRT, we need $2m$ to divide $q - 1$ for every prime q.
- With L levels, the largest modulus for "fresh ciphertexts" has size $Q_0 \approx p^{L+1} \approx 2^{(L+1)*p}$
- We break each ciphertext into upto c digits, do each digit is as large as $D = 2^{(L+1)*p/c}$
- The added noise variance term from the key-switching operation is $c * N * \sigma^2 * D^2$, and this must be mod-switched down to $w * N$ (so it is on par with the added noise from modulus-switching). Hence the ratio P that is used for mod-switching must satisfy $c * N * \sigma^2 * D^2/P^2 < w * N$ or $P > \sqrt{c/w} * sigma * 2^{(L+1)*p/c}$.
- With this extra P factor, the key-switching matrices are defined relative to a modulus of size $Q_0 = q_0 * P \approx \sqrt{c/w} * \sigma * 2^{(L+1)*p*(1+1/c)}$
- To get k-bit security we need $N > \log(Q_0/\sigma)(k + 110)/7.2 \approx N > (L + 1) * p * (1 + 1/c)(k + 110)/7.2$.

You can read more about how HElib is implemented in the official github repository (https://github.com/shaih/helib/) and the associated scientific paper in this report (https://eprint.iacr.org/2014/106.pdf).

SEAL library it is also invoked when the cryptographic context is created: the plain modulus and the polynomial modulus are setted through the contextGen function creating the SEAL Context while the coefficient modulus is generated thanks to HElib starting from the security level passed. When the context is created also

fractional and integer encoders are initialized starting from the plain modulus, the polynomial modulus and two integers representing the number of digits fractional and integer numbers should be truncated. As explained before Pyfhel is used only in batch mode. It is initialized when context is created and in particular a specific class used to build CRT is used. PolyCRTBuilder class takes a reference to the created context and the allocated output class is used when multiple data need to be computed together according to the elaboration showed before.

At initialization phase it is also important to create public, secret and relinearization keys since they are used to actually make encryption, decryption and operations over encrypted data. Keys are setted in the local environment and can be easily stored as files so that to have the possibility to restore them without the need of re-generate them. Public and secret keys are created starting from context parameters with the function KeyGen that as before invokes the Afseal KeyGen function performing the creation of a keys generator is used to actually create such keys.
The relinearization key on the other hand is created with the function relinKeyGen by setting the number of bit and the size for the key we want to create. It is used to both evaluate polynomials and to perform the relinearization process after each exponentiation and multiplication. An higher number of bit means initialize a key that is faster but noiser while the key size must be defined taking into consideration that to relinearize an encrypted data of size $K + 1$, at lest a key of size $K - 1$ is required.

3.2.2 Operations

Pyfhel makes easy to compute operations over encrypted data since it implements a function for each operation is required to be performed between cyphertexts or plaintexts. As explained before we use native batch mode to perform SIMD operations. In this project only the following functions are used:

- **add(PyCtxt ctxt, PyCtxt ctxt_other, bool in_new_ctxt=False)** : Sums two ciphertexts. Encoding must be the same. Requires same context and encryption with same public key. If in_new_ctxt is false than the result is applied to the first ciphertext, a new PyCtxt object is returned otherwise.

  Args:
  ctxt (PyCtxt) : ciphertext whose values are added with ctxt_other.
  ctxt_other (PyCtxt) : ciphertext left untouched.
  in_new_ctxt (bool=False) : result in a newly created ciphertext

  Return:
  PyCtxt resulting ciphertext, the input transformed or a new one

- **add_plain(PyCtxt ctxt, PyPtxt ptxt, bool in_new_ctxt=False)** : Sums a ciphertext and a plaintext. Encoding must be the same. Requiressame context and encryption with same public key. The result is applied to the first ciphertext.

  Args:
  ctxt (PyCtxt) : ciphertext whose values are added with ctxt_other.
  ptxt (PyPtxt) : plaintext left untouched.
  in_new_ctxt (bool=False) : result in a newly created ciphertext

Return:
PyCtxt resulting ciphertext, the input transformed or a new one

- **multiply(PyCtxt ctxt, PyCtxt ctxt_other, bool in_new_ctxt=False)** : Multiplies two ciphertexts. Encoding must be the same. Requires same context and encryption with same public key. The result is applied to the first ciphertext.

  Args:
  ctxt (PyCtxt) : ciphertext multiplied with ctxt_other.
  ctxt_other (PyCtxt) : ciphertext left untouched.
  in_new_ctxt (bool=False) : result in a newly created ciphertext.

  Return:
  PyCtxt resulting ciphertext, the input transformed or a new one

- **multiply_plain(PyCtxt ctxt, PyPtxt ptxt, bool in_new_ctxt=False)**: Multiplies a ciphertext and a plaintext. Encoding must be the same. Requires same context and encryption with same public key. The result is applied to the first ciphertext.

  Args:
  ctxt (PyCtxt) : ciphertext whose values are multiplied with ptxt.
  ptxt (PyPtxt) : plaintext left untouched.
  in_new_ctxt (bool=False) : result in a newly created ciphertext.

  Return:
  PyCtxt resulting ciphertext, either the input transformed or a new one

- **square(PyCtxt ctxt, bool in_new_ctxt=False)** : Square PyCtxt ciphertext value/s..

  Args:
  ctxt (PyCtxt) : ciphertext whose values are squared.
  in_new_ctxt (bool=False) : result in a newly created ciphertext

  Return:
  PyCtxt resulting ciphertext, the input transformed or a new one

In the next chapter we will see the application of these functions to the presented neural network and the procedures are applied to makes them secure.

Chapter 4

The CryptoNet

In this chapter the application of the homomorphic scheme on the presented model will be showed with details. Each layer and relative computations will be described underlining how data is manipulated, what and how operations are performed by putting in comparison the provided solution with the one exposed on the original paper [0]. A little introduction about the used technologies is provided as well as how such technologies are used to successfully implement and evaluate the CryptoNet taking as assumption what previously stated in 3.2.

4.1 Input Layer

As partially described in previous chapters the MNIST input data was manipulated and in particular each image is down-sampled by the application of an Average Pool layer with kernel dimension and strides equal to $(2, 2)$. The application of the Average Pool layer involve a simple operation schematized in figure 4.1: a window of size 2 is scrolled over the image averaging the RGB value of the 4 pixels in the window. The final result is an image with half of the dimension of the original one and where each pixel is the average of 4 pixels in the original image.

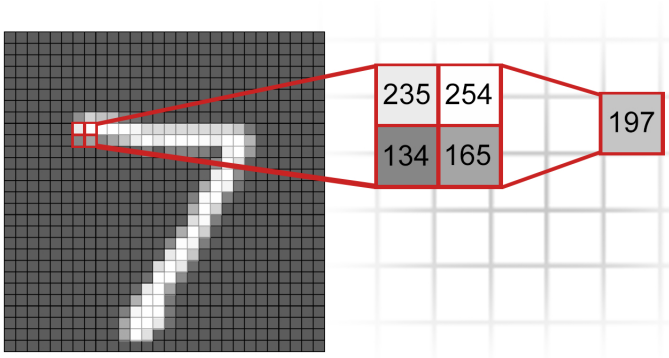


FIGURE 4.1: A sliding window of dimension 2 in the Average Pool layer applied to a MNIST sample image.

This process is iterated for each image in both train and test samples. Before saving final datasets pixel values are reshaped from the range $[0, 255]$ to $[0, 1]$ and a zero pad is applied to the upper and left side of each image. Taking input images of 28x28 pixels the described operation result in images of 15x15 pixels where the left and upper corner are all zero values. In particular the zero pad would be part of the convolution layer but I have preferred to add it at this time (restyling the

model to handle this preventive operation) so that to decrease the complexity of the convolutional layer making it easier to be handled during the encryption of the network.

4.1.1 Input Preprocessing

Down-sampled train dataset is used to train the network which weights and biases are finally exported and saved in the local storage. Before network is tested weights and biases, characterizing each layer after the train step, are pre-processed so that to handle the problem underlined in the previous chapter and connected to the atomic construction in the HE scheme. In particular they are encoded through a function that given a precision p_r and a floating point number n_i returns the integer representation of the floating point number multiplying its value by $(10)^{p_r}$ and rounding the result to the closer integer. This procedure it is necessary since in batch mode it is possible to encrypt only integer values thanks to the application of the SIMD computation.

Within this first encoding process also the problem connected to negative numbers is handled: when the encryption process is performed, neural network weights and biases are reduced modulo t and the CryptoNet is not able to compute the right result without handling negative numbers since a lot of information would be lost. To solve this problem the encoding function additionally maps the integer representation of n_i in the interval $[0, t]$ where $t/2$ represent the zero of the new representation.

Negative numbers are mapped to the interval $[(t/2)+1, t]$ while positive numbers are mapped to the interval $(0, t/2]$. As rebound of this procedure only $t/2$ positive numbers can be encoded or, to be more precise, all integers in the range $(0, t/2]$ while only $(\frac{t}{2})$ negative numbers can be encoded or more precisely all negative numbers in the interval $[-(t/2), 0)$.

Both test set, network weights and biases are first encoded and than saved in the local storage. This step is followed by the encryption process and the computation of the final result operated by subsequent layers.

4.1.2 Parallel Computation

After the encoding and other processes operated over train dataset it is encrypted such that to apply the HE scheme previously described: in particular a batch of n images is computed at the same time generating one encrypted file per image pixel. As explained in section 3.2 the cryptographic scheme permits us to encrypt n integer values in the same ciphertext thanks to both batch encoding and CRT techniques but we have to consider the fact we need to compute operations between single pixels and we can't do it easily by encrypting one or multiple images in the same cyphertext. One solution is to encrypt all pixels at the same position in the same ciphertext so that to be able to compute required operations without reinventing the wheel. This procedure can be easily understood in the following:

Let consider $(im_0, \ldots, im_n)$ images such that $im_i \equiv (px_{i0}, \ldots, px_{i224})$ where px_{ij} is the j-th encoded pixel of the i-th image. The j-th cyphertext c_j is generated through the application of the encryption function $enc(\bullet)$ to the set $P_j \equiv (px_{0j}, px_{1j}, \ldots, px_{nj})$ given the public key pk.

$$c_j = enc_{pk}(Pj), j \in (0, 224), P_j \equiv (px_{0j}, px_{1j}, \ldots, px_{nj})$$

Because of such encryption we can compute n images in parallel since we perform operations between exactly n elements at time in particular matrices are taken into consideration in the next section have to be considered as composed by elements of cardinality n or better each matrix element is not a single value yet but a vector of size n containing the j-th pixel of each of the n image. As result of such encryption process, the output of this computation is a folder containing 225 files each one containing the P_j-th set in encrypted form.

4.2 The Plain Network

As explained, the Keras model presented in chapter 2 is used to test the network architecture and to extract weights and biases are compared to the outputs of the final network. Before this step a "plain version" of the network is developed from scratch so that to be possible to handle the HE scheme and compute relative results. As explained before the implemented architecture was simple and the plain implementation reflect this simplicity. Differently from the Keras model the new version is built so that to successfully implement its encrypted form, in particular and in accordance with the previous section analysis the new version is coded such that each layer can handle n images in parallel. The encrypted version of the network is built starting from its plain form which inputs and outputs are showed in figure 4.2. As you can see the plain network takes as input 255 vectors (files) P_j on which convolution filters are used to compute the sliding dot product is required by the layer. In this section the plain version of the network will be presented through the analysis of each layer.

4.2.1 Convolution Layer

Convolution layer takes as input the 255 precomputed files containing the pixel values of exactly n images are computed together. As we know convolution it is technically a sliding dot product or cross-correlation between input and its filters. As showed in table 2.1 in the original architecture the convolution layer has 5 windows of size 3x3 and a stride of (2,2) but since our input is now a matrix composed by 255 vectors of n elements we have to manipulate such windows so that to compute the result in parallel. To accomplish this task and how showed in figure 4.2 b) we have to consider each element in each filter separately so that we can apply the plain multiplication and addition operations between vectors of dimension n. This choice it also mandatory since within our HE scheme we can encode, encrypt and make operations only between vectors of dimension n so that to exploit the batch computation properly.

To perform convolution the function **get_conv_map** is used to retrieve pixel indices in the sliding window according to the flatten version of the input; the function in fact, taking the size of the input I, the kernel width/height dimension K, the stride S, and the padding $P = 0$ computes a matrix containing K^2 vectors each one populated with a number of indexes equal to:

$$(((I - K + (2P))/S) + 1) = (((I - K)/S) + 1)$$

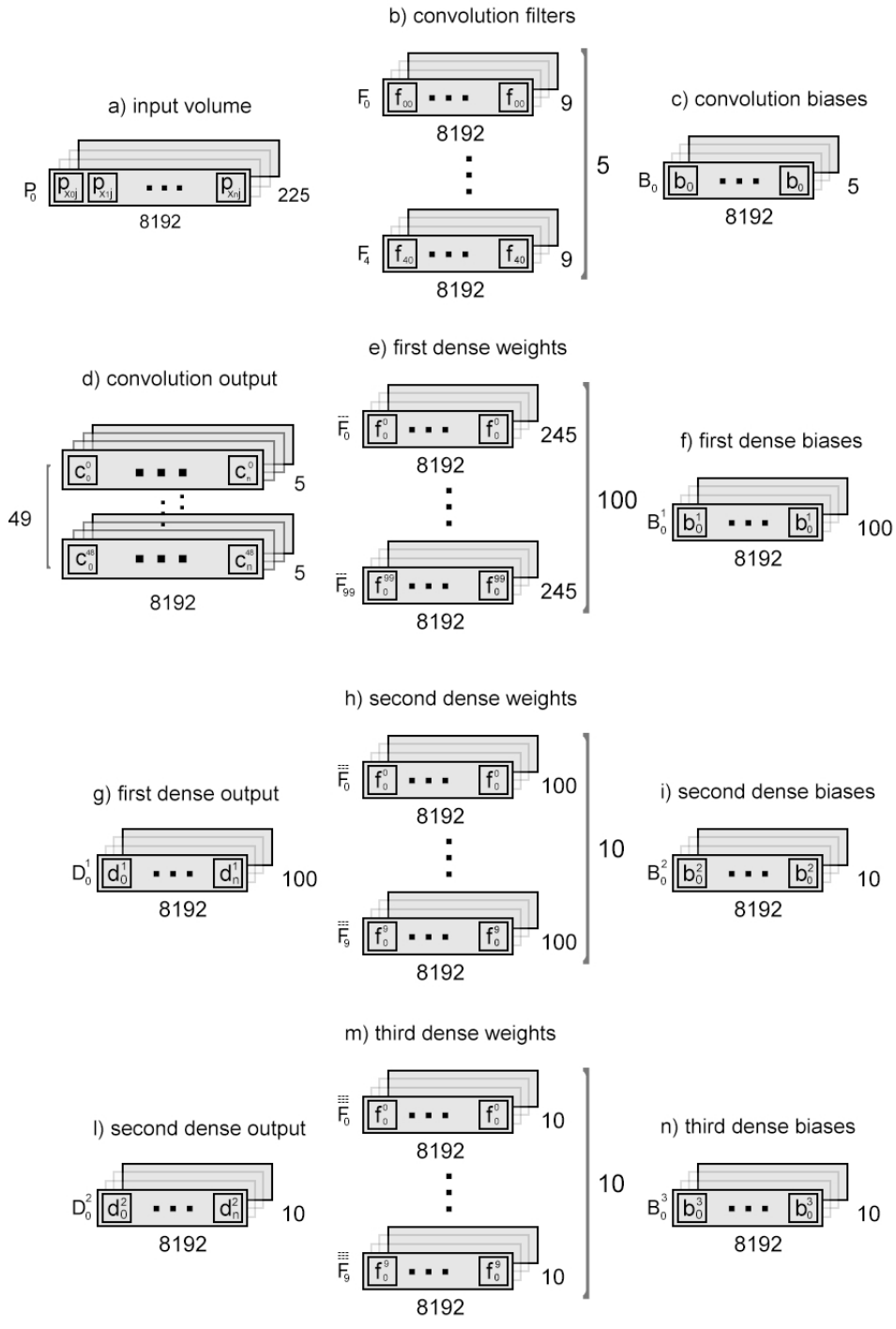


FIGURE 4.2: Plain network represented as the input and output vectors of each layer.

The resulting map contains all pixel indexes in the flatten input organized according to which filter element need to be multiplied with the specific pixel in the map. Such data is than used in **get_conv_windows** function that using such map return a matrix containing all the sliding windows populated with pixel indexes in the flatten input need to be multiplied with filter element at the same index. This matrix is than simply iterated computing in appropriate way the multiplication between the input vector and the vector of size n containing the weight need to be multiplied with. The 9 vectors of size n in the sliding window are summed and relative bias vector of size n is summed to the result. This process is iterated 5 times, once per filter need to be applied. This operation over input vectors ends with the flatten version of the convolution output volume organized into $49x5$ vectors each of dimension n.

4.2.2 First Dense Layer

The first dense layer is than computed. In the original architecture the first dense layer takes inputs of dimension 245 and produce an output of dimension 100. Since we have an input volume of $49x5$ vectors of dimension n we need to compute weights and biases as it was done in convolution layer. In particular each of the $245x100$ weight values in this layer need to be transformed into vectors of dimension n such that we can perform operations between n elements at time. In particular for each of the 100 matrices of dimension $245xn$ each vector of dimension n in the flatten input is multiplied with relative weight vector of dimension n, such vectors are summed together and the vector of dimension n containing the relative bias is than summed to the result that is finally squared element-wise. The resulting 100 vectors of dimension n are than used to compute the second dense layer.

4.2.3 Second Dense Layer

The second dense layer in the plain network architecture takes as input the output elaborated in the first dense layer and it computes same operations are computed by the previous layer. In particular the 100 vectors each of dimension n are multiplied with relative weight vectors and summed together according to the filter is applied so that to make possible to add the bias vectors. Such operation, which input and output vectors are showed in Figure 4.2 sections g, h, i and l, result in 10 vectors of dimension n that are squared element-wise. The result is a matrix of dimension 10xn that is used as input of the last dense layer.

4.2.4 Third Dense Layer

Contrary to the previous dense layer, in the third fully connected layer the linear activation function $f(a) = a$ is used in place of the square activation function. As for the previous dense layer the 10 vectors of dimension n are multiplied with relative weight vectors and bias is summed to the resulting sum operated over input vectors according to the filter need to be applied. The output is a matrix of ten vectors of dimension n containing the transpose of the result.

4.2.5 Output Evaluation

The output of the last dense layer is transposed so that to have a result organized in such a way to be easily evaluated. As you can notice the linear activation function is used in place of the softmax function. The normalized exponential function simply normalizes results into a probability distribution consisting of 10 probabilities proportional

to the exponentials of the input numbers so that each component will be in the interval $(0, 1)$ and the components will add up to 1. In particular the final evaluation consist in evaluate which label has the maximum among all the labels connected to an input image.

Since this classification is performed we can simply apply the **argmax** function to each row in our output so that to retrieve the index at which the maximum value is present: this data is finally compared to test labels so that to compute the accuracy and the loss of the model. The final accuracy of the plain network is than compared to the one given by the keras model so that to evaluate the differences between the two models and in particular evaluate the performances of the developed architecture. This final step is followed by the implementation of the encrypted version of the network in which the input images are previously encrypted and than elaborated by the network by the substitution of plain operations with their encrypted version.

4.3 The Encrypted Network

The encrypted version of the network is developed starting from two versions of the original architecture: the first is the plain network described above, while the second is the same network in which input data is manipulated considering the encoding process such that to exploit the CRT technique to handle the computation of big numbers. When the network values are multiplied with the given precision and rounded to the closer integer the numeric space increase relatively. Such increment in magnitude can only be handled by the usage of CRT explained at the end of chapter 3. Differently from the usage it was done by **Pyfhel** library during the parallel encoding so that to encrypt n values in the same polynomial, in this case the CRT is used so that to reconstruct a value starting from its representations of unique elements in the associated rings. This procedure is mandatory so that to compute a evaluable result since during the computation of each scheme some parameters exceed the threshold given by the selected t losing their effective values. In particular, so that to be possible to exploit the CRT in such a way to reconstruct the final result and to evaluate in successful way the network, values are computed during each step need not to exceed t. The encoded version of the network is used precisely to test such behaviour and checking no value during computation exceed the given threshold: different values for the network precision are used but all precisions grater than 2 cause a decrease up to 38% of the network accuracy.
An accuracy of 59.9975% seems to be the minimum can be achieved with the used numeric representation.

With the machine is used to develop and compile the network it is also hard to handle all data required to compute single layer computations, in particular to handle all data need to be maintained in memory during the computation of the second dense layer. This result it's not surprising but it limits the evaluation of the elaborated results.

The encrypted version of the network in particular use two parameters t exploiting the CRT technique as explained in the original paper. As well as the t parameter, the parameter n is selected to match the original scheme: in particular the network is evaluated separately for each parameter t and the final result given by each scheme is than composed and evaluated together so that to reconstruct the congruent result starting from its decomposition. This operation is performed by the class *Cryptonet*

in which the two scheme relative to t parameters are built and used to retrieve the partial results previously stored in the local storage. This makes possible to apply the CRT technique previously described. The result deriving from this elaboration is than compared with the one given by the Keras model underlining differences between original and new predictions that are finally printed in the terminal.

Differently from the figure 4.2 each vector of n elements is manipulated in a single $PyCtxt$ class instance containing n elements encoded and encrypted in a single polynomial: the resulting volumes can be represented as a compressed form of the ones showed in previous figure. Operations described in each layer definition of the plain network are obviously replaced with relative operations over encrypted data exposed by the **Pyfhel** library as showed at the and of chapter 3. When the $PyCtxt$ class representing the local sum of the layer vectors connected to a specific filter need to be computed, the *add* function is used. On the other hand when biases or weights are summed to the resulting $PyCtxt$ instance, the $PyPtxt$ class is used so that we can exploit the power of the plain addition. Biases and weights in fact are known by the service provider in plain form and not need to be encrypted. Plain addition and multiplication between polynomial are less expensive operations with respect to their encrypted form, also considering the computational power and time need to be spent to encrypt the required information so that to elaborate the result. Such operations between encrypted and plain data are easily implemented thanks to the API exposed by the used library. All the other operations between vectors are computed between encrypted data.

Chapter 5

Results and Conclusion

In this chapter the result from experimental phase will be presented putting them in connection with the original paper from Microsoft Research Group. In particular results are grouped in two main areas in which message sizes and time performances will be showed separately. Moreover in this chapter some conclusions about this project are delineated focusing on the application of such technologies in real world cases.

5.1 Results

The results are contained in this section are subdivided in three main subsections each one taking into consideration a different aspect of the final elaboration. The first is concentrated in message sizes it is important to understand how many data need to be exchanged between the client and the service provider and the amount of data need to be computed on service provider cloud so that to reconstruct the result. In the second subsection the time to perform operations and to compute the final result will be taken into consideration since it is a fundamental point connected to the usability of such technology in a real world case. Finally the last section contains a global vision about the scheme performances putting them in comparison with the one of the original paper. All the results are elaborated applying the network on a PC with a single Intel core i5 with 8 GB of RAM running the Linux Mint 19.2 (Tina)-Ubuntu bionic operative system.

5.1.1 Message sizes

In the previous chapter plain network and relative encrypted form are described with details by showing input and output vectors are used by each layer to compute the final result but no discussions about message dimensions are evaluated. To make more clear how many data is involved, following tables resumes the number of records per instance, the dimension of produced files and a description containing the final size per elaborated result.

Table 5.1 contains information about data is maintained by the client, in particular input data is pre-elaborated, encoded and than encrypted with client public key. At the end of this process encrypted data is sent to the service provider and elaborated in the cloud. The pre-computation process not only makes the elaboration of the result lighter but also decrease the dimension and the time is required to send data to the service provider. In this project data need to be exchanged between the client and the service provider is about half of the one in the original paper. It is obvious that with this new "compressed" representation of the test dataset we have lost a little bit of accuracy but on the other hand we have made data to be transmitted

TABLE 5.1: Client - message sizes of CryptoNet for MNIST

Dataset	Nr. records	Dimension	Description
Test	8192	(1x)28x28	Each of the 8192 images in the dataset has a size of 28x28 pixels and a dimension of 0.78 KB. The total dimension of the original batch is 6.34 MB.
Pre-computed	8192	(1x)15x15	Pre-computed dataset has images of 15x15 pixels each one of dimension 0.22 KB. The total dimension of the pre-computed dataset is 1.84 MB
Encoded	8192	(2x)15x15	The encoded dataset contains 8192 images which pixel values are float between 0 and 1. The total dimension of the encoded dataset is 29.4MB
Encrypted	225	(2x)1	Encrypted dataset contains 2x225 instances each one of size 534.4 KB. The total amount of data need to be sent to the service provider is 240.48 MB.
Result	10	(2x)1	The decrypted output is organized in 2 file with a total dimension of 1.3 MB.

lighter and the transmission process speeder. The encrypted version of the dataset is about 240.48 MB and it is 8 times bigger than its plain version that, after all the preliminary processes, is about 29.4 MB. However the same message contains 8192 images and therefore, the per image size is only 29.35 KB since two different *schemes* are applied. The final elaboration is a composition of results comes from the two different schemes are applied and which results are used to produce the final predictions. In tables that follow such division is not taken into consideration since data is considered as belonging to a single scheme operating on the test set on which we want to produce predictions. Moreover it is important to consider that the showed dimensions are connected to the computation of the test set considering a encoding precision of 2. This parameter is important because it influences the dimension of numbers are involved in the computation that it is directly connected to the representation is used for every integer value. No additional tables are showed for different values of the precision parameter since increasing it means trying to compute numbers of infeasible size causing the complete corruption of output data because of integer reduction operated by the cryptographic scheme and their inevitable approximation connected to the specific machine and the python environment is used. Such important point could be resolved incrementing the number of schemes used to compute the final result or technically using more t values for the computation: the application of multiple t linearly increase both the space, the total amount of data elaborated and the time is required to complete each computation.

Table 5.2 on the other hand contains informations received by the service provider and elaborated in the cloud. The table in particular is organized per layer and it shows the dimension of data exchanged between layers. In a distributed fashion different layers could be elaborated by different nodes in the cloud. Imagine a real case in which a distributed architecture is involved in the computation of the predictions: if we have s different schemes we could desire to distribute each scheme

TABLE 5.2: Service Provider - message sizes of CryptoNet for MNIST

Layer	Nr. records	Dimension	Description
Convolution	245	(2x)1	Contains 49x5 vectors organized in a 245 files each one of 534.4 KB. The output of this layer has a total dimension of 257 MB.
First dense	100	(2x)1	Contains 100 vectors organized in a 100 files each one of 786.6 KB. The output of this layer has a total dimension of 158 MB.
Second dense	10	(2x)1	Contains 10 vectors organized in a 10 files each one of 17 MB. The output of this layer has a total dimension of 546 MB.
Third dense	10	(2x)1	Contains 10 vectors organized in a 10 files each one of 134.5 MB. The output of this layer has a total dimension of 2.6 GB.

on a different cluster and compute partial results in parallel way on different nodes. Taking into consideration such real case it is important to understand how many data is exchanged between layers since could be possible that different layers elaborating data with a different scheme could be placed on different nodes of a distributed system. Convolutional layer takes as input the dataset encrypted by the client and produce 490 files (2x245) with a total dimension of 257 MB. Such data is than used by the first dense layer that produce 200 files with a total dimension of 158 MB. The second dense layer on the other hand takes 158 MB of data organized in 200 files and produces 10 files about 17MB each: the total dimension of second dense layer output is 0.5 GB. Because of the operations involved in dense layers the output of the last layer is about 2.6 GB which is divided in 10 files containing predictions in transposed form. Such encrypted data is than sent back to the client who can decrypt the result thanks to his/her private key. As you can notice keys dimension are not taken into consideration since the total amount of space required to store both public, private and relinearization keys is less than 35 MB. To maintain tables organization the final result dimension is showed in Table 5.1 since the decryption and the evaluation processes are operated on the client machine. The client in fact is the only one can actually decrypt the predictions through his/her private key.

Results connected to each layer are bigger than the ones presented in the original paper: as explained encrypted values are maintained within dedicated classes which also contains a lot of information useful to restore values from saved files. This fact is also connected to the approximation used to represent integer values and relative computations are operated by the layers using the batching technique explained in the previous chapters.

5.1.2 Time Analysis

Another important aspect is to consider network performances by computation time point of view. The first model is presented has higher performances with respect to the model was created from scratch. This fact is connected to how operations are implemented: in convolution and dense layers the networks was built so that

TABLE 5.3: Breakdown of the time it takes to apply CryptoNets to MNIST dataset

Layer	**Instances**	**Total experienced time**
Input (encoding)	2	55.64 s
Input (encryption)	2	22.96 s
Convolution	2	23.3 s
First dense	2	244.46 s
Second dense	2	184.28 s
Third dense	2	58.23 s
Output (decryption)	2	132.95 s
Output (decoding)	2	1.3 s
Total time	//	728.96 s = **12.15 m**

TABLE 5.4: The performances of CryptoNets for MNIST dataset

Stage	**Latency**	**Throughput**
Encoding + Encryption	81.8	55.64 s
Network application	2	55.64 s
Decryption + Decoding	2	55.64 s

it is possible to apply the cryptographic scheme and this fact makes the general behaviour slower than the model with which weights and biases are computed. In particular the time required to compute weights and biases is not taken into consideration here, since it is not directly connected to our elaboration. Moreover it is possible to use pre-trained networks and to apply the method presented in this work without the computation of such values.

Table 5.3 summarizes the time required by each layer to compute the output and to save it in the local disk. In a real world application the transmission time interleave between client and service provider communication should be considered but not here, since it is directly connected to the way and the transmission channel is used to exchange data between parties. On the other hand it is important to underline the time required to compute the output of each layer. In the layer computation the time to retrieve and encode layer's weights and biases and the time it is necessary to store the final result on disk are considered; this makes the resulting value higher than what it is actually needed to compute the result. Moreover it is important to consider how CryptoNet is implemented since all data elaborated by the encrypted network it is contained in classes are used to encode values and thanks to which it is possible to perform operations between encrypted data. Such overhead makes performances lower than it was expected.

In general, time appears to be better than the original paper and it is connected to test data representation; despite this the experienced performances are actually worst since they are relative to the computation of half of the data is used in the original work.

Table 5.4 contains informations about the perfomances of CryptoNets for compressed MNIST dataset. Three main stages identified during the process are considered separately and the final throughput is showed. Since apply the network allows making 8192 predictions simultaneously, this PC can sustain a throughput of $8192x3600/511 = 57712$ predictions per hour. Encoding and encryption processes take 78 s to be

computed, therefore if 8192 instances are encoded a throughput of 378092 instances per hour can be encoded and encrypted. Decrypting data takes 132.95 s and additional 1.3 seconds to decode predictions for the two instances. Therefore a throughput of 219673 decryptions and decoding per hour is achievable with our setup.

5.2 Conclusion

Technologies are used to implement the CryptoNets in general appears not to be the best choice possible this because libraries are used to create and apply the cryptographic scheme are not natively developed to be used with neural networks. A native implementation using SEAL library without other bridge libraries could make possible to control in better way the behaviour of the cryptographic scheme making the computation of the network more feasible to errors. On the other hand using Pyfhel makes possible to develop a working solution without coding and testing a per use implementation. Different methods can be used to made data safe for example using traditional cryptography that appears to be the best choice at the increasing of neural network complexity. The scheme is presented in this work in fact has mathematical limitations not permitting to handle in easy way layers with not linear activations. We have also to consider that such scheme is a quite new solution and it is still evolving. Major challenge is the computational complexity that, even for simple image recognition tasks, makes network test and evaluation very slow at the increase of the parameters are used. It is obvious that different methods or a more complex evaluation on a particular problem can improve the computation time and complexity even maintaining higher throughput and network accuracy. In our particular case a way to improve network performances could be using GPU in place of CPU and, as suggested in the original paper, FPGAs to accelerate the computation. Another important aspect is that to satisfy a certain level of security and to perform operations over big numbers we have to select high parameters and it is a limitation since network performances are directly connected to them. Moreover noises are generated by operations need to be maintained as low as possible so that to be sure the result not to be corrupted and predictions can be reconstructed starting from their encrypted form. We also need to consider that the paper on which this work is based it is an old version, more work it was done up today, in fact the CryptoNets project is still evolving and some of the problems are underlined in this section was resolved in the new version of the library.

In general the original work leaves much room for improvement but it delineates an interesting direction, a point will be fundamental for next generations and on which much work need to be done. As Stephen Howking says AI would be the biggest event in human history but it is important to evaluate the risks. Prevent sensible data to be stolen it is important but it is also important to create a new way of see ourself in which security is a way of thinking first, and than a protocol need to be followed.

Appendix A

References

- **[0]** CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy
- **[1]** CryptoNN: Training Neural Networks over Encrypted Data (arXiv:1904.07303)
- **[2]** CS231n Convolutional Neural Networks for Visual Recognition
- **[3]** Homomorphic Encryption Standardization
- **[4]** Dropout: A Simple Way to Prevent Neural Networks from Overfitting
- **[5]** Cryptography and Network Security: Principles and Practice. Prentice Hall. p. 165. ISBN 9780138690175. Stallings, William (3 May 1990)
- **[6]** Somewhat Practical Fully Homomorphic Encryption ICT-2007-216676.
- **[7]** Fully Homomorphic Encryption without Bootstrapping. Zvika Brakerski and Craig Gentry.
- **[8]** AES Proposal: Rijndael. Daemen, Joan; Rijmen, Vincent (March 9, 2003)