UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

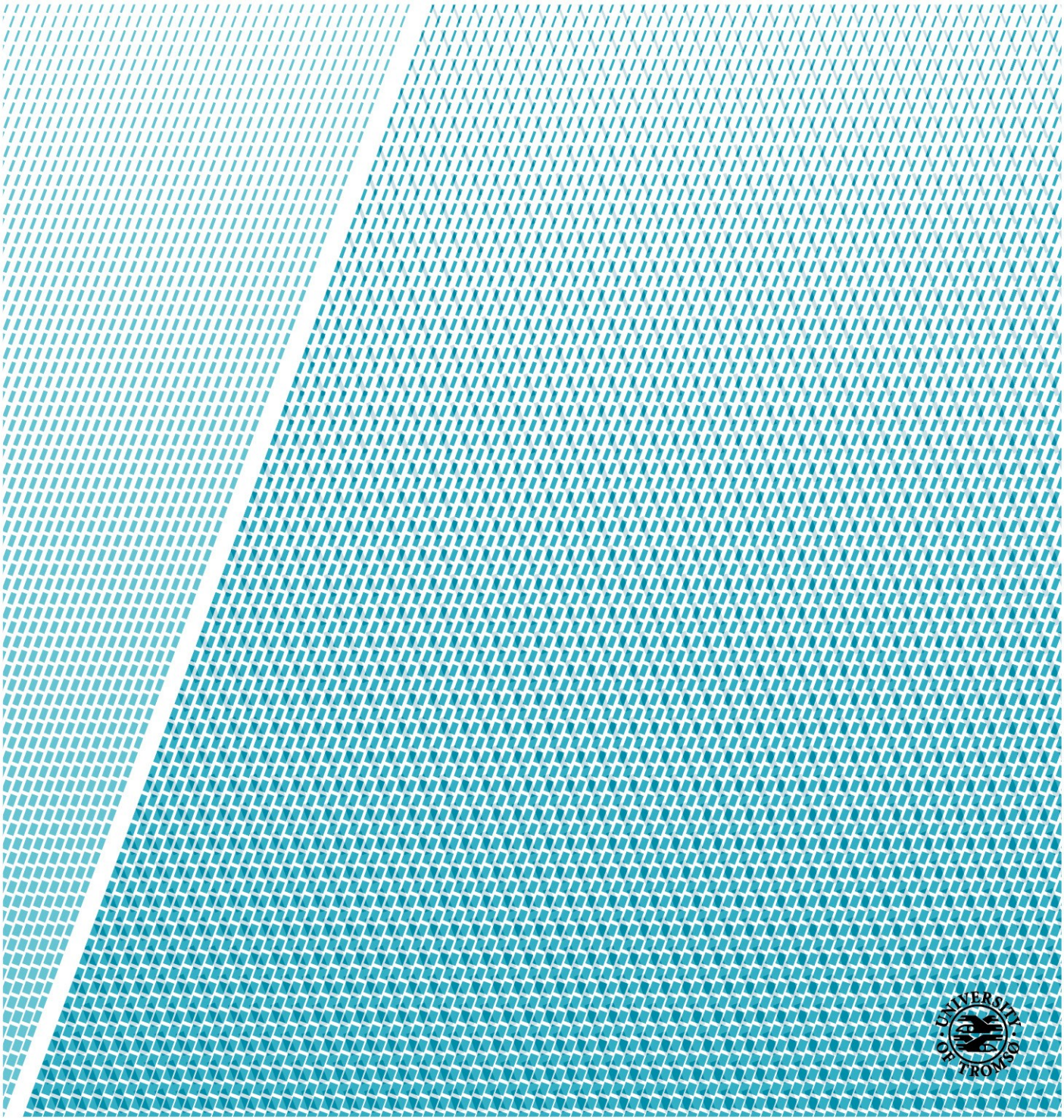
Faculty of Science and Technology, Department of Computer Science

Detection of Wheezes and Breathing Phases using Deep Convolutional Neural Networks

—

Johan Fredrik Eggen Ravn

INF-3981, Master's thesis in computer science, June 2017



Abstract

A Fully Homomorphic Encryption (FHE) scheme allows you to perform an arbitrary number of computation on encrypted data without decrypting it first. This is a very useful feature as it allows us to perform computations on data without knowing the data itself. Computations on sensitive data can be done in an untrusted environment without compromising privacy. Even though this may sound like the holy grail of cryptography which could solve the worlds IT problems when it comes to security and trust, it have some issues. Unfortunately Fully Homomorphic Encryption(FHE) libraries are very limited in Python. In this paper we have implemented a Python API based of a FHE C++ library called HElib and we will take a look at potential usages and limitations to the library.

Acknowledgements

Morbi sit amet diam condimentum, posuere tellus ut, fringilla massa. Nulla viverra dolor leo, vel cursus erat ornare non. Phasellus vehicula velit eget posuere rutrum. Integer at egestas enim, sed vulputate sapien. Nunc odio metus, mattis in erat at, rhoncus venenatis tortor. Vivamus porttitor molestie risus, vel gravida leo lacinia ac. Phasellus efficitur vehicula risus ut ornare. Pellentesque tincidunt libero ac massa finibus, sed hendrerit nisl pellentesque. Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

Contents

List of Figures

List of Tables

Introduction

When dealing with sensitive data we need to make sure that the data is secure. This means, if the data is leaked due to a security breach, error in the system or any other reasons, we need to make sure the data are unrecognizable for anyone who is not authorized. This is commonly solved by using an encryption algorithm to encrypt the data, making it unreadable to anyone who do not know the private key used during the encryption. As more and more businesses and organizations are moving away from having a their own dedicated server to renting server space in large server parks, like a cloud, security are becoming a bigger concern than ever before. Traditionally businesses and organizations would have their own private server which would handle computation and storing of the data. The only people having direct access to the server would be authorized personnel hired by the business or the organization. Meaning the only real threat from a potential attacker would come from outside the business/organization, as any attacks from inside the business/organization would be met with prosecution and lawsuit.

As the trend of cloud services have exploded in the last years this have changed. We no longer have control over the physical location of the server, and we now have to trust the cloud service to uphold access control agreed upon. When storing data on cloud servers we are faced with not only the same security threats as on the private server, but also additional threats specific for cloud computing, like side channel attacks[12], virtualization vulnerabilities and abuse of cloud services. Cloud administrators or unauthorized users might access the data and obtain sensitive information for various motivations. Because of the increased threat when storing data in a cloud environment the data need to be encrypted at all times with our own encryption key, as even the cloud service cannot be trusted.

If we use the cloud service to only store data, then this is pretty simple, but what if we want to do some computation on the data stored? One way to do this is to encrypt the sensitive data and store this on the cloud. To perform a computation the data would be downloaded to a private server and decrypted. The computation is performed and the data is encrypted and uploaded back to the cloud. The problem with this approach, it's inefficient, especially when a lot of computation is done and it also requires a private server to perform the computation. Another approach is to use a Homomorphic Encryption(HE) scheme which enables us to perform computation on encrypted data without decrypting it first. This means, we can perform computation on ciphertexts generating an encrypted result which, when decrypted, matches the result of operations performed on the plaintext. By using Homomorphic Encryption we can perform the computation without ever exposing the data which means we can ensure that the data is secure at all times during the computation.

Why do we want Homomorphic Encryption? As shown in the previous example it allows us to do computation on our data without ever exposing the data during the computation. This essentially means that we can now export the computation work to a third party like a cloud service. We no longer have to pay for expensive servers to perform heavy computations, we can export the job an external server which is much cheaper in term of power consumption, server maintenance and bandwidth cost. This also allows us to pay for exactly the amount of storage and computational power needed, which can dynamically increased when needed. But is also have other benefits like hospital record can be examined without compromising patient privacy and financial data can be analyzed without opening it up to theft.

If Homomorphic Encryption is so great why are not everyone using it already? Even though there are many benefits which comes with using Homomorphic Encryption, there are some issues. First of all, performance. The first Fully Homomorphic Encryption scheme by Craig Gentry[1] was taking 100 trillion times as long to perform computations on encrypted data than plaintext computation. In this paper we will take a look at a FHE library called HElib which is an open source library in C++. With Boost Python we have created a Python API of this library and will use this to see how far the FHE schemes have come and look at practical usages and/or which challenges still remain for FHE schemes.

Background

This chapter describes relevant technical background and relevant work. We will first define the field of machine learning, and then describe the methods that have been used in this thesis. We will explain feature extraction for audio signals using spectrograms. We will also describe recent relevant work that uses the same machine learning methods and feature extractions methods as in this thesis.

2.1 Machine learning

Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc. [1].Mauris vel interdum nunc. Curabitur mi lectus, rhoncus venenatis ullamcorper quis, mattis volutpat dui. Proin at lectus ac metus ornare lacinia et pretium risus [2].

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{P + N}$$

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

Recall

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

Precision

2.2 Deep learning

2.2.1 Convolutional neural network

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris. Figure 2.1 shows an illustration of the convolutional layer.

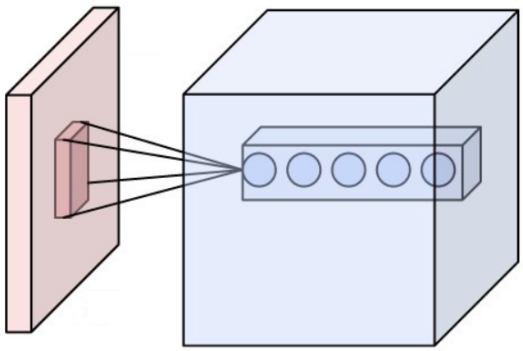
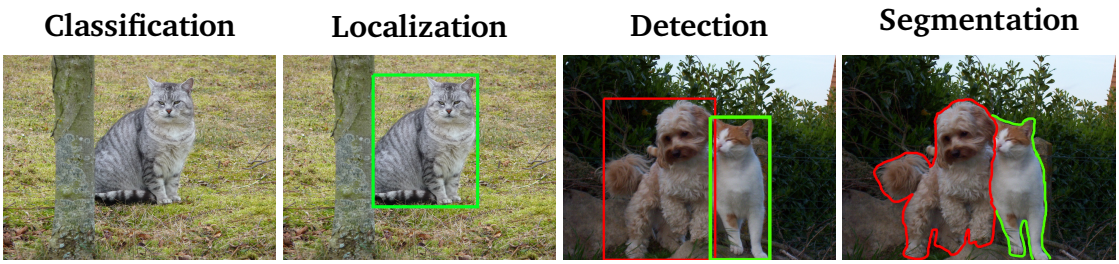


Figure 2.1: Illustration of the convolutional layer . The red box is the input, and blue box is the result. The width and height of the input is preserved, but since the convolutional filter has 32 filters, the depth of the output volume is 32 pixels.



(a) Cat **(b)** Cat **(c)** Cat, Dog **(d)** Cat, Dog

Figure 2.2: Vizualisation of four different image recognition tasks

2.3 Classification vs. detection

- **Classification:** Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc.
- **Object localization:** Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc. the size of the bounding box.
- **Object detection:**Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc.

Approach

This chapter outlines the methodology and actions which will be taken to create a Fully Homomorphic Encryption API in Python. We will also take a look at how we can test the Python API to measure the difference in performance between the C++ library and the Python library. We will explain the approach of choosing a Python wrapper for the C++ code. And we will look at how we can determine possible usages and limitations of the library.

3.1 Objectives

The main objective of this thesis is to create a Fully Homomorphic Encryption API in Python and measure the performance to get a sense of how far we have come in the development and implementation of FHE schemes. The Python API will be based on an already existing FHE library. The Python API should support a multitude of functions in order to classify it as a FHE API. Ideally the Python API should not be significantly slower than the original library in terms of compilation time and runtime. Lastly we want to highlight the benefits/drawbacks of having a FHE scheme in Python and find potential usages of the API.

The thesis work is structured into the following three stages

1. A design stage

2. An implementation stage

3. Measurement, analysis and comparison stage

3.2 Design Stage

In the design stage we will discuss the necessary steps to design a solution which satisfies the objectives mentioned above. For simplicity sake the design stage is divided into several steps needed to fulfill the requirements of the objectives, these are outlined below.

1. Research and choose a suiting C++ wrapper for python

2. Create an API which is easily understood

3. Find a suitable model for testing the API

3.2.1 C++ wrapper for Python

In this section we will examine different C++ wrappers to find one that is suitable to wrap the FHE scheme to python. Before choosing the wrapper the FHE scheme implementation should be examined to see if there are some special functionality the wrapper should support. If the FHE scheme have a lot of user defined objects or any other functionality, the wrapper chosen should be able to handle this in a somewhat simple matter. The main goal of this exercise is to find a wrapper to create a Python API from an already existing FHE scheme. The wrapper should ideally have a good performance i terms of runtime and the API created with the wrapper should be well structured and easily organizable.

3.2.2 Create the API

There are currently few options when it comes to FHE libraries in Python. When the Python API is created it is important that the API is well structured, well documented and it should include examples for potential usages. Ideally the Python API should be as similar to the underlying library as possible, making it easier for the user to find valuable documentation on how to use the library. The library being wrapped to Python is most likely an extensive library which means the user should be able to use the documentation for the API combined with the underlying library documentation for a more thorough understanding on how to use the API to the

full extent. Any differences between the API and the underlying library should be highlighted making it easy for the user to see the parallels between the API and the library. As the underlying library is most likely an extensive library it will consist of many files. When creating the Python API this needs to be taken into consideration as the API should be presented in a lucid manner corresponding to the underlying library. As the underlying library will most likely be large it should also be taken into consideration how much of the library should be wrapped to Python to support the functionality required by an FHE scheme.

3.2.3 Find a suitable model for testing the API

Once the API is created the performance of the Python API should be measured with regards to the existing library in C++. A benchmark test should be created on the existing C++ library which can be used in comparison to the Python API. The tests created should include most of the functions wrapped to Python to see if there are some operations or functions which are slower than others in regards to the benchmarks. If a function or an operation is slower than others it should be investigated to provide an explanation why this is the case and possible solutions to the result. The tests implemented should not only serve as a measurement in performance but also as examples to potential usages of the library. The tests should show how the library can be used in different scenarios and also show the limitations to the library. It's important that the tests highlight the benefits and drawbacks of having the FHE scheme implemented in Python compared to the C++ implementation. If there are other implementations of a FHE scheme in Python this could be used in the testing phase to measure the difference in performance in terms of runtime and functionality compared to the API implemented.

During the research stage of the assignment an implementation of the FHE scheme in Python was found. This implementation does have some limited functionality compared to the existing C++ library, but it can be used as a comparison to our implementation. We should look at this library to compare the difference in performance and functionality to our implementation of the HElib library.

3.3 Implementation stage

The implementation stage is divided into a few objectives to satisfy the goals outlined in the design stage.

1. Choose an underlying FHE scheme implementation

2. Creating the Python API

3. Create tests to measure the performance

3.3.1 Choose a Fully Homomorphic Scheme implementation

In the first task of the implementation stage a FHE scheme should be chosen as the basis of the Python API. Research should be done to explore the different existing implementations of FHE which are available. The FHE scheme chosen should be an up-to date implementation which addresses some of the main problems of FHE like the increasing key-size and computation performance. This is important as this scheme will later be used to see the potential usages and see how far the development of an FHE scheme has come.

The FHE scheme should be open source making it easier to access additional information and examples on how the scheme is implemented.

Design

In this chapter we will display the results of the tasks outlined in the approach section.

4.1 C++ wrapper for Python

When it comes to wrapping C++ code to python there are many options to chose from, like Boost, SWIG, cTypes and manual wrapping. We wanted a wrapper which would satisfy all the requirements to wrap the code in terms of user defined objects and types. But satisfying these requirements alone is not enough, it needs to be fast in terms of compiling and execution time and it needs to be simple. The first option considered was SWIG. SWIG was considered as the first option because of it's simplicity. It is a software development tool for building scripting language interfaces largely automatically. When using SWIG we ran into a problem. Even though it did wrap a some parts of the code automatically, a converter for all the user defined objects and types had to be created. As the HElib library is big and we do not need the majority of the functions in conjunction with a lot of user defined types and objects made the automation aspect of the software rather useless.

Discussion

5.1 asdf

Conclusion

6.1 Future Work

Bibliography

[1] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, pp. 115–118, Feb 2017. Letter.

[2] "Homesite quote conversion." `https://www.kaggle.com/c/homesite-quote-conversion`. Accessed: 2017-30-05.

Appendices

tttt

Experiment	d	f	f	f
x	x	d	figure	f
x	x	d	f	f
x	x	d	f	f
x	x	d	f	f
x	x	d	f	f

Table A.1: asdf