Ecole Polytechnique Fédérale de Lausanne (EPFL)

Privacy-preserving model aggregation in decentralized machine learning with hardware enclaves

Scalable Computing Systems Laboratory (SaCS)

Paul Nadal (300843)

Project Advisor: Professor Anne-Marie Kermarrec

Project Supervisors: Rafael Pires, Rishi Sharma

Contents

Chapter 1

Introduction

In recent years, the development of cloud services and neural networks has continued to grow. This has allowed machine learning to evolve and not longer be limited to a single computer. Decentralized machine learning now makes it possible to combine the power of several computers connected around the world and share data from several different users to best train a neural network.
This has a significant advantage in the rise of artificial intelligence and deep learning, but there are a few things to watch out for in this evolution.
Besides the issues of speed of data sharing, access to power sources for all... the most important aspect to consider for users is data privacy. Indeed, the protection of the sharing and use of private data through the internet is a point to consider when developing decentralized machine learning.
The topic of this project is to find a way to protect the data sent and received by each computer when sharing models between connected nodes when training a machine learning model.
To address this need for data protection, I used hardware enclaves in the code that averages the models. This report explains why and how SGX enclaves were able to address this data privacy need in the `decentralizepy` framework.

Chapter 2

Background

2.1 Decentralized machine learning

Training a machine learning model with a huge amount of data using a deep neural network is not suitable for a single machine. Sharing the resources of multiple linked computers optimizes and accelerates the learning of the model and provides a faster result. This is why distributed and decentralized machine learning is more and more used in deep learning. More specifically, each computer is considered as a node and is connected to one or more neighbors to which it shares its data or model once the learning is complete. Once all the data or models of its neighbors have been retrieved, the machine averages what is has just received with its current model and so on at each step or epoch until it reaches the highest possible accuracy.
This is exactly what the `decentralizepy` framework I worked on this semester does.

2.2 Enclaves and privacy

2.2.1 Intel SGX technology

Intel Software Guard Extensions (Intel SGX) is an Intel technology for application developers who seek to protect selected code and data from disclosure or modification.

An SGX enclave is a trusted execution environment embedded in a process. The central idea of SGX is the creation of a software "enclave" that is essentially a separate, encrypted region for code and data. The enclave is only decrypted inside the processor, which protects it even from being read directly from RAM.
SGX enclaves protect your code and data from both software and hardware attacks.
SGX is used to protect against many known and active cybersecurity threats, such as malware attacks, by reducing the attack surface of servers and workstations through the use of secure enclaves, which protect information from processes running at higher privilege levels.

At runtime, your application is divided into two parts: a secure part and an insecure part. When the application is launched, the enclave is created, and this enclave is placed in the secure part.

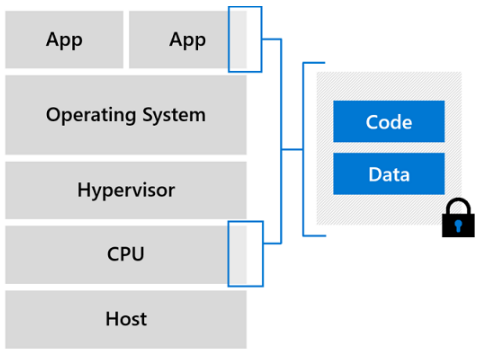

Figure 2.1: Schema of an SGX enclave

When an enclave function is called, only the code inside the enclave can see its data. External accesses are always denied. On return, the data in the enclave remains in the protected memory.
At the end of the execution of the process using the enclave, the enclave is destroyed to free the memory it occupied.
For example, if someone tries to attack the OS, BIOS, VMM or SMM layers, Intel SGX is there to provide an extra layer of protection by placing your sensitive data in an isolated, encrypted portion of memory. This means that these layers can be compromised, but your data remains protected because the application data stored in the enclave itself is inaccessible to unverified external parties and is therefore safe from destruction, manipulation or modification by unauthorized users.

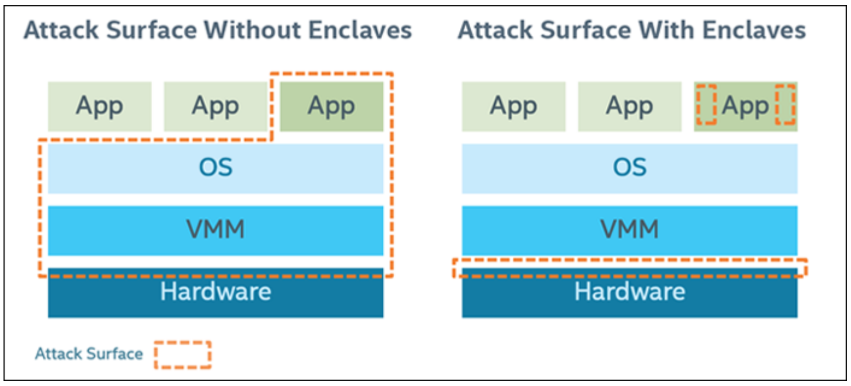

Figure 2.2: Diagram of vulnerable areas with and without enclave

Chapter 3

Enclaves in decentralized machine learning and implementation

Data sharing protection in the `decentralizepy` decentralized machine learning framework is a key point to implement so that each node can share its model without the others knowing who the model comes from.
The main goal of my project was to implement SGX HW enclaves in the `decentralizepy` framework in order to protect the privacy of the model received from each machine's neighbors when averaging the models.
To achieve this, I had to divide the work into 2 tasks: convert the Python code that calculates the average into C++, then use the SGX SDK to create the enclave.

3.1 Average part conversion in C++

3.1.1 Ctypes

The first thing to do to convert the computing part of the average from Python to C++ was to be able to call the C++ from the already existing Python code in order not to re-code the whole framework.
For that, I decided to use the Python library `ctypes` which seemed to me the most complete and the easiest to use.
In order to launch a program in C or C++ from Python you just have to give the path of the `.so` compiled file to the `CDLL` function, and define the types of the arguments and the return of the functions of the C/C++ program to finally call the function.

```
from ctypes import *
c_program = CDLL(path_to_c_file)
c_program.average.argtype = c_char_p
c_program.average.restype = c_char_p
res = c_program.average(data)
```

3.1.2 Reading data in C++

The second part of the conversion was to transform Python data into into C++ acceptable types (`int`, `char *`...). This can be done in two different ways, one being much more efficient than the other.

3.1.2.1 Using structs

The first option, less efficient, was to reuse the `ctypes` library and to build `structs` in the Python code and also in the C/C++ code, and convert the data we had into C/C++ types (`c_char_p`, `float`, `int`, `c_int_p`...) by iterating on them one by one.
However, this option was very expensive because we had to iterate over all the data in Python, transform them before passing them as arguments to the function in C++ and finally convert the data back from C++ to Python. Moreover, the purpose of this conversion being to use enclaves to keep the recovered data confidential, iterating on them in the Python code was not the best thing to do.

3.1.2.2 Using json

The second option, simpler and less expensive, is to keep the data received by the neighbors of each machine as it is (i.e. serialized in the form of a dictionary), and to pass it directly to the C++ program in order to parse it thanks to a json trusted library (we will see this in the next sub-chapter). Then, it is enough to calculate the average of the models in C++ and to return to the Python code a list of bytes (always serialized). Finally, once the averaged model is retrieved in the Python code, a simple function allows to deserialize this data and to retrieve the averaged model in a tensor form, the form in which the framework works. The only point of vigilance of this method is to carefully encode the string given to C++ in `UTF-8` code and carefully decode the returned data.

Here is the algorithm used to average the models in the C/C++ program:

Algorithm 1: AVERAGE(DATA, STATE_DICT, n_neighbors)

```
Create an empty float pointer *total_params*
weight_total = 0
length = DATA.size()

for $i$ = 0, 1, ..., length - 1 do
    degree = DATA[i]["degree"]
    indices = DATA[i]["indices"]
    params = DATA[i]["params"]
    weight = $\frac{1}{(MAX(n\_neighbors, degree) + 1)}$
    weight_total += weight
    for $j$ = 0, 1, ..., indices.size() - 1 do
        total_params[indices[j]] += weight * params[i]

indices = STATE_DICT["indices"]
params = STATE_DICT["params"]

for $i$ = 0, 1, ..., indices.size() - 1 do
    total_params[indices[i]] += (1 - weight_total) * params[i]
return total_params
```

3.2 SGX enclaves

As a reminder, SGX allows developers to divide a computer's memory into enclaves, i.e. private, predefined memory areas that can better protect users' sensitive information and thus preserve privacy.

3.2.1 ECALLS and OCALLS

Before we define the terms ECALL and OCALL, let's first define the terms trusted and untrusted components. A trusted component includes the code that is executed in a protected area of the processor. This component is also called an enclave. An untrusted component is the rest of the application, including all its modules.

The application can invoke a pre-defined function inside the enclave, passing input parameters and pointers to shared memory within the application. These invocations from the application to the enclave are called ECALL (enclave calls).

When an enclave executes, it can perform an OCALL (outside call) to a pre-defined function in the application. Unlike an ECALL, an OCALL can not share the enclave's memory with the application, so it must copy the parameters into the application's memory before the OCALL.

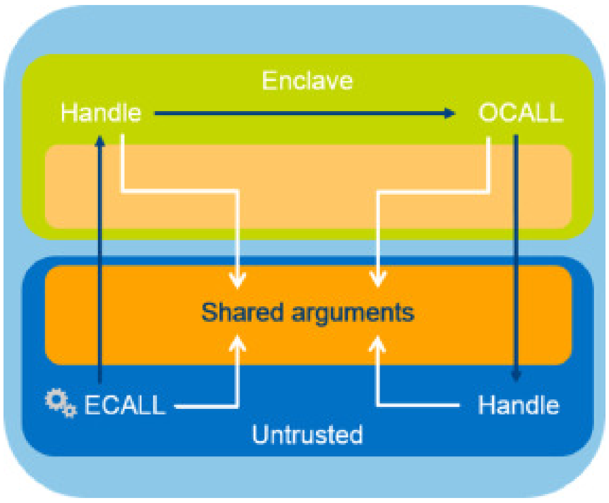

Figure 3.1: Schema of ECALLS and OCALLS

3.2.2 Intel SGX SDK

Intel provides the Intel Software Guard Extensions (Intel SGX) SDK for software developers who want to strengthen the security of their applications, by using Intel Software Guard Extensions technology.
The Intel Software Guard Extensions SDK is a set of APIs, source code samples, libraries and tools that allow the software developer to write and debug Intel Software Guard Extensions applications in C/C++.
Proper installation of the Intel SGX SDK on all machines is required in order to compile the code properly on all nodes.

3.2.3 MakeFile and dependencies

The creation of the MakeFile to compile the code of an enclave is done in 2 steps: first the compilation of the main program which creates the enclave, makes calls to the enclave (ECALLS) and destroys the enclave once the program is finished.
In a second step we have to compile the code inside the enclave which requires several specific libraries. Indeed, the enclave is not able to share its memory with untrusted components during its execution and the program containing the enclave must therefore call alternative functions.
The most important library that contains all the main SGX functions is the `sgx_urts` library.

3.2.4 Enclave creation

The creation and compilation of an SGX enclave is done in several parts. First you need to create the enclave in the main program and execute it with the simple function `sgx_create_enclave`.
Then calls to the functions contained in the enclave (ECALLS) are simply made like calls to regular external functions, only giving in addition the `eid` (or more simply the id of the enclave) to the function.
Finally when the use of the enclave is over, the main program must destroy the enclave in order to recover the memory occupied by it. This step is simply done by calling the `sgx_destroy_enclave` function which only needs the `eid`.
Each time the enclave is created or destroyed, it is important to ensure that the code returned by the function is `SGX_SUCCESS` in order to ensure that the enclave works properly in the program.

3.2.4.1 EDL file

The first step in compiling the enclave is the creation of an EDL file that allows you to simply define which functions are trusted and which are untrusted.
The format of an EDL file is very simple:

```
enclave {
    trusted {
        //trusted functions
    };
    untrusted {
        //untrusted functions
    };
};
```

The EDL file offers several options for passing pointers into an ECALL or an OCALL but the two main ones are `in` and `out`.
For an ECALL, with the `in` option, the buffer is copied from the application into the enclave and changes will only affect the buffer inside the enclave, whereas for an OCALL, with the `in` option, the buffer is copied from the enclave to the application and changes will only affect the buffer outside the enclave.

On the other hand, for an ECALL, with the `out` option, a buffer will be allocated inside the enclave and initialized with zeros, and it will be copied to the original buffer when the ECALL exits, while for an OCALL, with the `out` option, a buffer will be allocated outside the enclave and initialized with zeros. This non-secure buffer will be copied to the original buffer in the enclave when the OCALL exits.

3.2.4.2 Edger8r Tool

This tool generates edge routines by reading an EDL file provided by the user. These edge routines provide the interface between untrusted components and enclaves.
More specifically, given an EDL file `enclave.edl`, `Edger8r` generates 4 files :

- `enclave_t.h` : contains the prototype declarations for trusted proxies and bridges (included in the enclave program)
- `enclave_t.c` : contains the function definitions for trusted proxies and bridges
- `enclave_u.h` : contains the prototype declarations for untrusted proxies and bridges (included in the host program)
- `enclave_u.c` : contains the function definitions for untrusted proxies and bridges

These files can be generated by the following simple command in the MakeFile :
`sgx_edger8r enclave.edl --search-path /opt/intel/sgxsdk/include`

3.2.4.3 Enclave Signing Tool

It generates the enclave metadata, which includes the enclave signature, and adds this metadata to the enclave image. This tool is used to digitally sign an enclave library with the private key contained in the `private.pem` file that is created with the command `openssl genrsa -out private.pem -3 3072`.
A configuration file can also be added if needed.
Run `sgx_sign sign -key private.pem -enclave enclave_name -out signed_enclave` to sign your enclave.

Chapter 4

Evaluation

4.1 Machines configuration

The framework runs on machines running under `Ubuntu 20.04.3 LTS (Focal Fossa)` operating system. The CPU is an `Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz` with `64 bits` and the machines have `64 GiB of DDR4 at 2666MHz`.
All the following tests are performed with 4 machines, 4 nodes per machine and 16 iterations (test after each iteration) on the `celeba` dataset.

4.2 Testing accuracy

The first things to test after the implementation of the hardware enclave in the `decentralizepy` framework are accuracy, training and testing losses. Indeed, we want to check that this new implementation does not degrade the performance of the model learning.

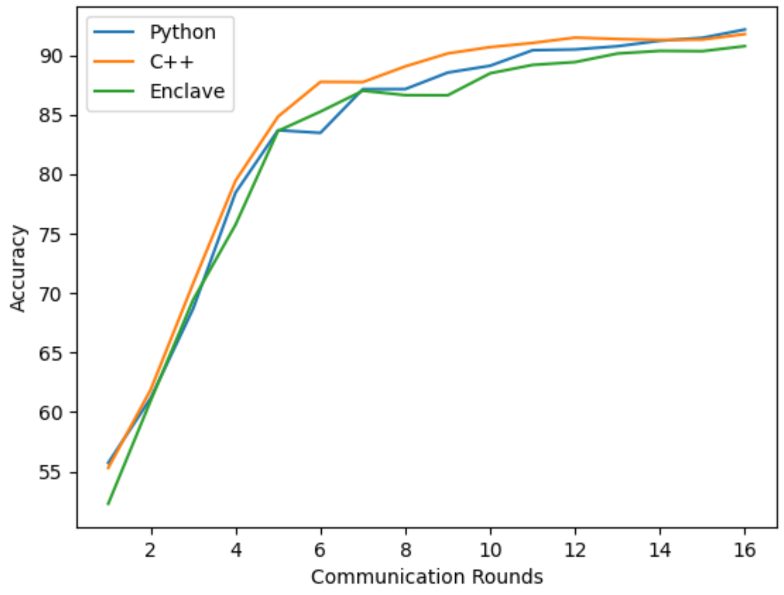

Figure 4.1: Accuracy comparison between Python, C++ and the SGX enclave

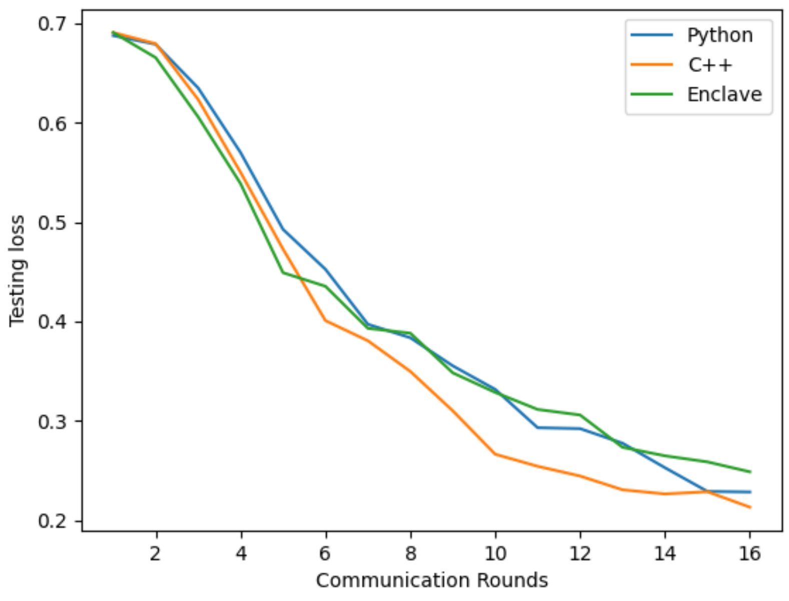

Figure 4.2: Training loss comparison between Python, C++ and the SGX enclave

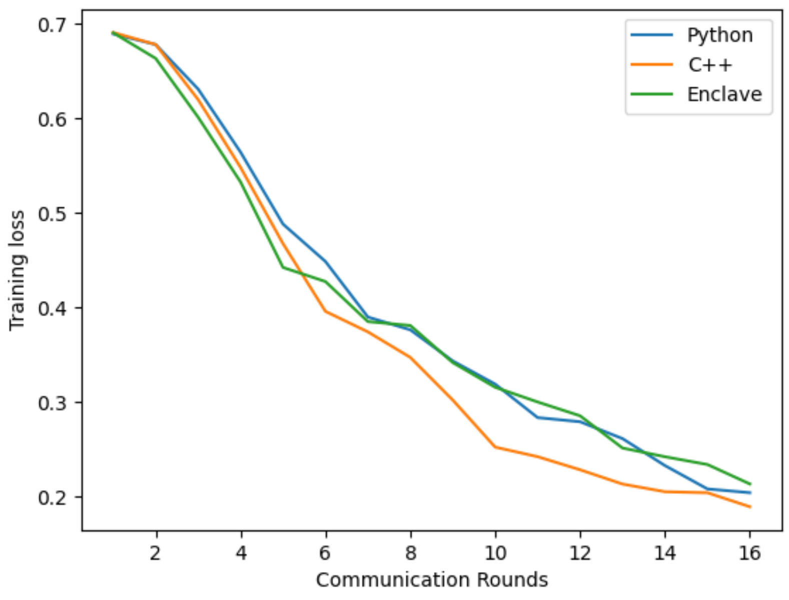

Figure 4.3: Testing loss comparison between Python, C++ and the SGX enclave

We can see here after comparing the curves of accuracy, training loss and testing loss for the 3 different configurations (Python, C++ and enclave) that they only slightly differ. We can thus conclude that the conversion of the average part from Python to C++ and then the implementation of the SGX enclaves in the framework does not impact the model learning performance.

4.3 Testing speed

The second point to test is the speed of execution of the program in order to determine if the new implementation speeds up the execution or slows it down.

From several researches, it seems that C++ is the fastest alternative compared to Python. C++ is considered to have a faster execution time, mainly because Python is written in C. C++ is often considered the fastest programming language in the world.

I measured the execution times of the whole program and only the average part for the 3 configurations (Python, C++ and enclave) and reported them in the following table:

Time	All program	Average part
Python	31 minutes 15 seconds	0.033 seconds
C++	31 minutes 34 seconds	0.741 seconds
Enclave	31 minutes 53 seconds	0.738 seconds

Table 4.1: Program execution time for each configuration

Contrary to the initial assumption that C++ would be faster than Python, these execution time readings say the opposite. Indeed, we can see here that the Python implementation is the fastest (20x faster than one with the enclave), followed by the C++ implementation and finally the implementation with the SGX enclave. This is probably due to the fact that the program with the enclave (or the C++) is itself called by Python and adding the time of the call to the code via the `ctypes` library can increase the total execution time of this average part.
However, this increase in execution time is minimal in the program and only makes it slow down by a few seconds over its entirety. We will therefore consider here that the difference obtained is negligible.
On the other hand, a framework totally in C++ which would make the call to the enclave would surely allow to reduce the execution time of the program.

4.4 Testing memory usage

The last point to test is of course the use of memory during the execution of the framework. To do this, I first noted the number of bytes and the percentage of RAM used by each node after each iteration.

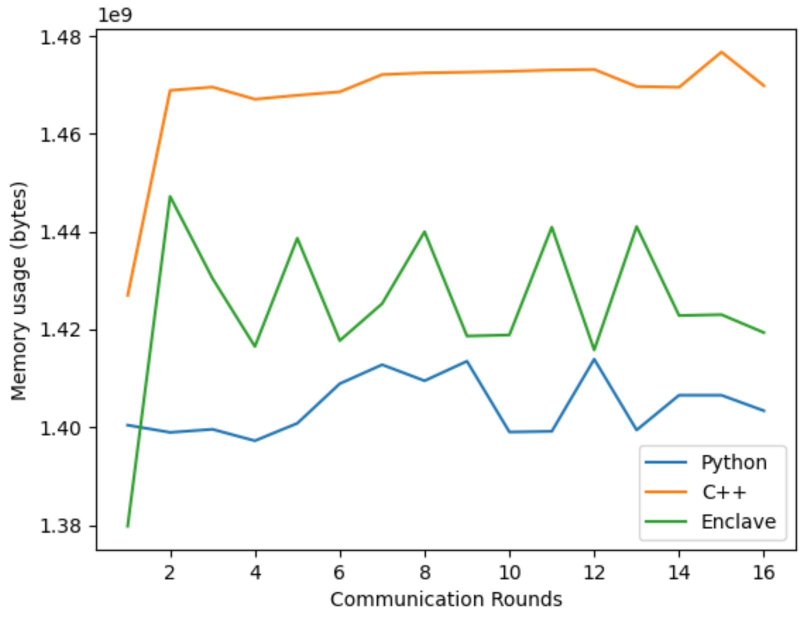

Figure 4.4: Comparison of memory usage (in bytes) between Python, C++ and the SGX enclave

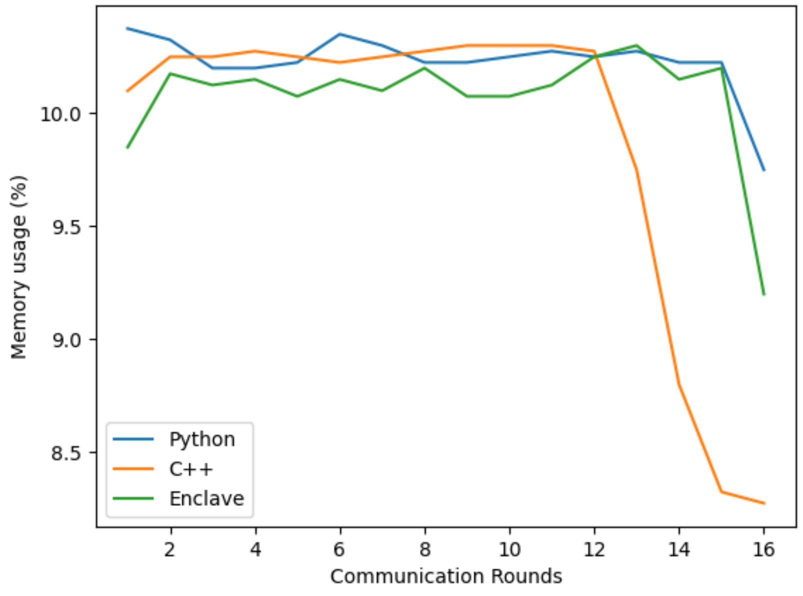

Figure 4.5: Comparison of memory usage (in %) between Python, C++ and the SGX enclave

With the help of these curves we can see that there is a slight difference in memory usage between the 3 different implementations even if on the graph illustrating the percentages, this difference is less visible. Indeed, we can see on the graph 4.4 that the

C++ and enclave implementations use slightly more memory, which is probably due to the sharing of the model with the C++ code which itself uses memory.
I tried to use another library to get a more detailed result on the memory usage for each line of code in order to see how much of it was used by the average function but I didn't get a conclusive result.
Moreover, it is difficult to know how much memory is used when combining Python and C++ code. For more accuracy, it would be interesting to measure the memory used directly in the C++ program but this would require additional research.
However, we can conclude once again from this test that the changes made for the implementation of SGX enclaves do not degrade the performance of the initial framework.

Chapter 5

Conclusion

Decentralized machine learning is becoming more and more popular as it opens new doors to deep learning and neural networks. The main goal of this project was to find a solution to guarantee the privacy of models shared between neighbors in the `decentralizepy` decentralized machine learning framework.
The sensitive part was when sharing models in the average part. In order to address this problem, I first converted the average part from Python to C++. In a second step, I implemented hardware enclaves on this part of the code so that data received from neighbors could only be read in a memory area inaccessible by other processes.

The code of this new implementation in C++ compiles and works correctly with the rest of the framework in Python.
The tests performed in the previous chapter allow us to conclude that the implementation of data protection via SGX enclaves does not degrade the performance of the framework. Indeed, whether it is a question of model accuracy, program execution time or memory used by the program, the results are equivalent or the slight differences are negligible on the overall scale of the framework.

Chapter 6

Potential next steps

This project may open doors to other related projects.
First, this project implements SGX enclaves that are hypothetically supposed to decrypt the models of the enclave's neighbors. However, the actual data sent and received is not encrypted when sent nor decrypted in the enclave. The implementation of this system would finalize the data privacy protection in the `decentralizepy` framework.
Another extension of this project could be to test the reliability of the enclaves by trying to break into them, although this is more of a cyber security issue than a machine learning issue.
Finally, a last project option in the continuity of this one would be to implement hardware enclaves not for model sharing as here but for data sharing (which is a project currently in progress).
In conclusion, I think that this project is only the beginning of the implementation of a privacy protection of shared data/models but that the framework should be improved to get something even more secure.

Bibliography

[1] Travis Addair *Decentralized and Distributed Machine Learning Model Training with Actors*, Stanford University

[2] Blaise Agüera y Arcas (2018) *Decentralized Machine Learning*, 2018 IEEE International Conference on Big Data (Big Data)

[3] `https://download.01.org/intel-sgx/linux-1.6/docs/Intel_SGX_SDK_Installation_Guide_Linux_1.6_Open_Source.pdf`

[4] `https://sgx101.gitbook.io/sgx101/`

[5] `https://github.com/openenclave/openenclave/blob/master/samples/helloworld/README.md`

[6] `https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf`

[7] `https://www.intel.com/content/dam/develop/external/us/en/documents/overview-of-intel-sgx-enclave-637284.pdf`