Universidade Estadual de Campinas
Instituto de Computação

Leonardo Henrique Neumann

Automatic Generation of Secure Neural Network Models with Homomorphic Encryption

Geração Automática de Modelos de Redes Neurais Seguros com Criptografia Homomórfica

CAMPINAS
2021

Leonardo Henrique Neumann

Automatic Generation of Secure Neural Network Models with Homomorphic Encryption

Geração Automática de Modelos de Redes Neurais Seguros com Criptografia Homomórfica

Supervisor/Orientador: Prof. Dr. Edson Borin
Co-supervisor/Coorientador: Prof. Dr. Diego de Freitas Aranha

CAMPINAS
2021

Na versão final esta página será substituída pela ficha catalográfica.

De acordo com o padrão da CCPG: “Quando se tratar de Teses e Dissertações financiadas por agências de fomento, os beneficiados deverão fazer referência ao apoio recebido e inserir esta informação na ficha catalográfica, além do nome da agência, o número do processo pelo qual recebeu o auxílio.”
e
“caso a tese de doutorado seja feita em Cotutela, será necessário informar na ficha catalográfica o fato, a Universidade convenente, o país e o nome do orientador.”

Resumo

Criptografia homomórfica é uma solução criptográfica que permite que aplicações realizem computações sobre dados criptografados. O conceito vêm sendo estudado há vários anos por conta de suas potenciais aplicações, especialmente em relação à manipulação de dados sensíveis sem o risco de exposição do conteúdo. Com o crescimento significativo de aprendizado de máquina na última década, diversos provedores de serviços na nuvem começaram a oferecer soluções de hospedagem de modelos de aprendizado de máquina, o que levanta a preocupação com a privacidade especialmente em aplicações onde a privacidade é um requisito. Em resposta, redes neurais com foco em privacidade baseadas em criptografia homomórfica vêm sendo desenvolvidas, em grande parte utilizando modelos de criptografia homomórfica nivelada (LHE), que possibilita apenas computações em cadeias limitadas de operações. Esquemas de criptografia homomórfica total (FHE), por outro lado, podem realizar computações em cadeias com um número arbitrário de operações, tornando-as viáveis para a implementação de modelos de aprendizado profundo. No entanto, esquemas FHE foram considerados impraticáveis por muitos anos devido ao overhead computacional, porém, avanços recentes em esquemas FHE podem potencialmente melhorar o desempenho. Este trabalho propõe o desenvolvimento de um backend de compilador de redes neurais para gerar modelos homomorficamente criptografados usando o esquema TFHE, juntamente com a implementação de técnicas de otimização para melhorar o desempenho de modelos gerados.

Abstract

Homomorphic encryption is a cryptographic solution that allows applications to compute over encrypted data. It has long been studied due to the potential applications it enables, especially on the manipulation of sensitive data without the risk of exposing their content. With the significant growth of machine learning in the last decade, many cloud providers started to offer practical solutions for hosting machine learning models on the cloud, which raises privacy concerns especially on applications where privacy is a must. In response to that, privacy-preserving neural networks based on homomorphic encryption have been developed, most of them using leveled homomorphic encryption schemes (LHE), which only allows computations up to a certain number of operations on a single chain. Fully homomorphic encryption (FHE) schemes, on the other hand, can perform computations over chains of any arbitrary length, which makes it suitable for deep learning models. However, FHE schemes have long been considered impractical due to the computational overhead, but recent advances on FHE schemes can potentially improve the performance. This work proposes the development of a neural network compiler backend to generate fully homomorphic encrypted models using TFHE scheme, along with the implementation of optimization techniques to improve the performance of the generated models.

Contents

1 Introduction **8**

2 Neural Networks **10**
2.1 Neuron Model . . . 10
2.2 Network Training . . . 11
2.3 Network Architectures . . . 12
2.3.1 Multilayer Perceptron . . . 12
2.3.2 Recurrent Neural Networks . . . 13
2.3.3 Convolutional Neural Networks . . . 14

3 Tensor Compilers **15**
3.1 Tensors . . . 15
3.2 Code Generation . . . 16

4 Homomorphic Encryption **18**
4.1 Fundamental Concepts . . . 18
4.2 Learning With Errors . . . 19
4.3 TFHE Scheme . . . 20

5 Related Works **21**
5.1 TensorFlow XLA . . . 21
5.2 nGraph-HE . . . 22
5.3 CHET . . . 23

6 Objectives and Methodology **25**
6.1 Methodology . . . 25
6.2 Evaluation . . . 26
6.3 Schedule . . . 26

Bibliography **28**

Chapter 1

Introduction

The demand for machine learning has grown significantly over the past decade, being fueled by the development of new learning techniques and the ability to leverage hardware acceleration. This is allowing the implementation of deeper architectures that can reason about increasingly complex problems. Examples include, but are not limited to, computer vision [27], natural language processing [21], and recommendation systems [31].

In response to this demand, cloud services offer solutions for training and inferencing machine learning models, making it easier and sometimes cheaper to allocate the resources required for the task when compared to hosting and maintaining hardware on-premises, allowing researchers and developers to focus on the problems they want to solve. While being a practical solution for many applications, it raises privacy concerns both about the models and the data, especially when handling sensitive information.

Homomorphic Encryption (HE) is a cryptographic solution that allows computations to be done while the data is still encrypted, without the need of decrypting or knowing the secret keys [2]. It provides a model where the owner of the data does not need to trust the environment where the computation is performed. In other words, the data owner can upload the data into an untrusted environment, do computations over it, download the data, and decrypt the results, without the data ever being exposed [9].

There are different classes of HE schemes that are suitable for machine learning. Leveled HE, also known as Somewhat HE (SHE), supports a limited number of operations before the inherent accumulated noise becomes too large and the ciphertext is no longer decryptable [2]. Fully HE (FHE), however, introduces an additional noise management technique known as bootstrapping, which enables a homomorphic computation to have an unlimited number of operations by periodically reducing the noise [7].

Notwithstanding, FHE has been considered impractical for general use due to its performance overhead [9]. For this reason, most of the libraries like SEAL [26] and HElib [14] implement SHE schemes such as BFV [10] and CKKS [5] in the leveled setting, which limits the multiplicative depth (multiplications on a single operation chain) on which the computations can be performed, as they are the main source of noise.

However, recent advances on FHE schemes can potentially improve performance, making it practical for many sorts of applications where privacy is a need. TFHE [6] is an FHE scheme that can do fast bootstrap operations and can implement logic circuits and lookup tables, which can compute any operation required by neural networks. TFHE can also work on the leveled setting, but using bootstraps allows computation chains on any arbitrary length, making it suitable for deep neural networks.

Implementing homomorphically encrypted neural networks manually is difficult due to noise management and performance-related decisions like encryption parameter adjustments and batching techniques. However, the recent development of tensor compilers in the context of neural networks can not only aid the translation to homomorphically encrypted models but also explore information within the models to improve performance.

Having presented the motivations, our work proposes to implement a neural network compiler backend that can emit optimized code to work with encrypted data using the TFHE scheme. We plan to explore recent developments both in tensor compilers and homomorphic encryption schemes to provide a novel implementation that can offer a practical solution for privacy-preserving deep neural networks.

The following chapters are organized as follows. Chapter 2 explain relevant concepts about neural networks. Chapter 3 presents tensor compilers and fundamentals about tensors. Chapter 4 explores concepts about homomorphic encryption and the TFHE scheme. Chapter 5 discusses what has been already developed on tensor compilers and homomorphically encrypted models. Chapter 6 gives additional information about the implementation and how we expect to meet our goals.

Chapter 2

Neural Networks

Artificial Neural Networks, or simply Neural Networks (NNs), are computing models inspired by the nervous system that forms animal brains. A neural network is composed of the connection of artificial neurons, also known as perceptrons, that vaguely resembles biological neurons. Their development is primarily driven by the desire of understanding how the brain works and the demand for systems that can work with abstract and loosely defined problems. The artificial neuron model was introduced by Rosenblatt (1958) [24], inspired by the earlier formalization of McCulloch and Pitts (1943) [19].

This chapter is structured as follows. Section 2.1 introduces the neuron model and the neural network structure. Section 2.2 shows how a neural network is trained. Section 2.3 presents architectural decisions that should be considered when modeling a neural network.

2.1 Neuron Model

Biological neurons receive input signals on their synapses, located on nervous terminations called dendrites, and then emits output signals over their axons which are dependent on the strength of the inputs. The strength of the output signal depends on several factors, like the distance of the synaptic cleft and the neurotransmitter configuration in that region. Likewise, the output signal y of an artificial neuron is modeled as the linear combination of an input signal vector x with the associated weight vector w that mediates a signal strength, added to a bias term b and then applied into a non-linear activation function $f(x)$ for threshold, as represented by the equation $y = f(b + \sum_i w_i x_i)$. The analogy between a biological neuron and its mathematical model is depicted in Figure 2.1.

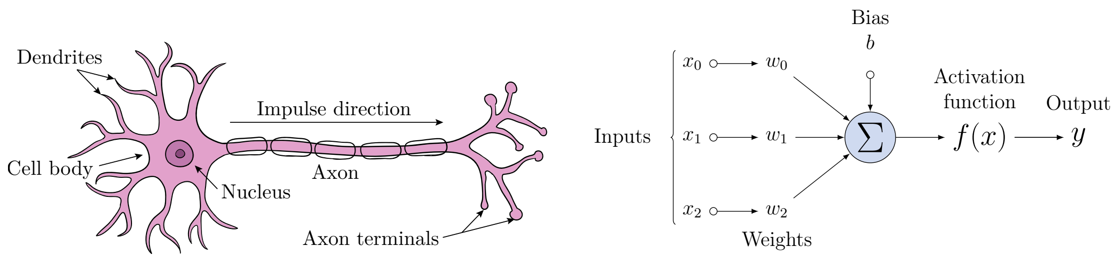


Figure 2.1: Representation of a neuron (left) and its mathematical model (right)

A neural network is modeled as a directed weighted graph with nodes representing artificial neurons and edges representing connections that resembles biological synapses. The weights can be positive or negative and represent the influence that input has over a neuron output. Inferencing is the process of feeding the network with data and allowing it to emit an output based on its internal structure and weights, whereas learning is the process of adjusting these weights so the network can approximate a desired behavior.

Most neural networks are organized layer-wise and can be classified as feed-forward or recurrent networks. The former only allows signals to flow from the input to the output, whereas the latter can contain feedback connections that allow signals to flow in both directions. Another distinction is related to the depth of the network. Deep neural networks (DNNs) can contain a large number of hidden layers, in contrast with shallow networks that are composed only of a few layers.

2.2 Network Training

Neural networks are generally used as function approximators and the training is done in a supervised learning fashion, where a limited number of labeled samples are given in the form of a pair (X, y), being X the observation or input and y the label or expected output. The sample set S contains input-output pairs that represent samples of a loosely defined target function $f(X)$, so that $y := f(X), \forall(X, y) \in S$.

A neural network that has no internal state can be modeled as a parametric approximator function $\hat{f}_\theta(X)$, where θ represents the free parameters that control the network behavior, such as the neuron's weights and biases [12]. In this model, finding good values for θ in the search space can lead to $\hat{f}_\theta$ to be a good approximation for f.

There are analytic and stochastic methods to find values for θ, which are usually defined in terms of a loss function and an optimizer. A loss function $\mathcal{L}(y, \hat{y})$ expresses the error between a prediction $\hat{y} := \hat{f}_\theta(X)$ and its correspondent label y, whereas an optimizer $\mathcal{O}(\hat{f}_\theta, S, \mathcal{L})$ is an iterative method for finding values for θ that can make $\hat{f}_\theta$ converge into the target function f. Figure 2.2 represents, in a simplified way, how training works.

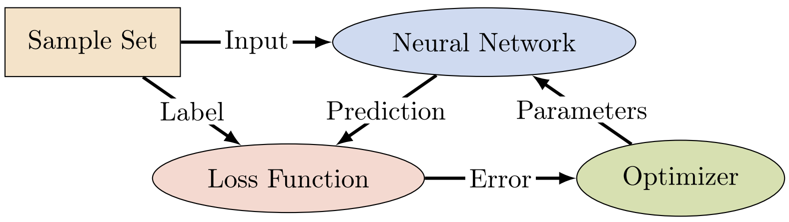


Figure 2.2: Neural network training

If we assume that $\hat{f}_\theta$ and the loss function $\mathcal{L}$ are both differentiable in respect to θ, then the training process can be treated as a general non-convex optimization problem, which is a requirement for all the gradient-based optimization methods. However, expressing the loss function gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ analytically becomes infeasible in a complex network. To solve this problem, it is used a concept named backpropagation [25], that obtains the gradient of the network computing the gradient of every single artificial neuron in terms of their inputs by applying the chain rule recursively, as depicted in Figure 2.3.

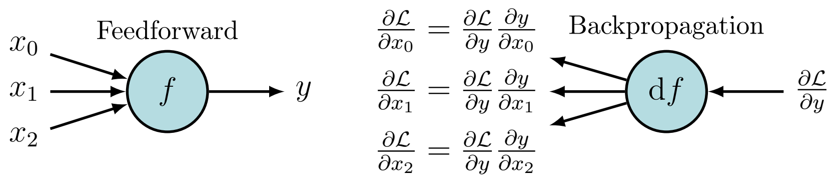

Feedforward
x_0
x_1
x_2
f
y
$\frac{\partial \mathcal{L}}{\partial x_0} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x_0}$
$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x_1}$
$\frac{\partial \mathcal{L}}{\partial x_2} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x_2}$
Backpropagation
$\mathrm{d}f$
$\frac{\partial \mathcal{L}}{\partial y}$

Figure 2.3: Feedforward and backpropagation over a single neuron

In the field of machine learning, the most commonly used optimization methods are based on gradient descent, such as Stochastic Gradient Descent (SGD), Adam, Momentum, AdaGrad, AdaDelta, and RMSProp [28]. For loss functions, the most commonly used in regression problems are Mean Square Error (MSE) and Mean Average Error (MAE), whereas the most common for classification are Hinge Loss and Cross-Entropy Loss [29].

Besides the parameters that are adjusted during the network training phase, there are other configurations related to the model and algorithms. Model hyperparameters are related to the model choice, such as the architecture, the number of hidden layers and nodes per layer, whereas algorithm hyperparameters are responsible for modifying the behavior of the training algorithms to improve the training quality and speed. Some hyperparameters that are found on training algorithms are described in Table 2.1.

Hyperparameter	**Description**
Learning Rate	The value that determines the step size at each iteration while moving towards the minimum loss function.
Initialization	The criterion to determine the weights that the neural networks should be initialized in the training phase.
Stop Criterion	The criterion to determine when the neural network should stop training based on its current training state.
Batch Size	Determines how many samples should be used together when training the network on each step.

Table 2.1: Common algorithm hyperparameters

2.3 Network Architectures

During the development of neural networks, many architectures were proposed, each one developing new concepts and ideas that can offer better results for specific kinds of problems. This section presents the most important architectural decisions that can generalize to the common use cases of neural networks.

2.3.1 Multilayer Perceptron

The simplest feed-forward neural network is known as Multilayer Perceptron (MLP) and is represented in the Figure 2.4. It contains an input layer, one or more hidden layers, and an output layer, being all layers fully-connected to their neighbors. MLPs are very flexible and suitable for both classification and regression problems.

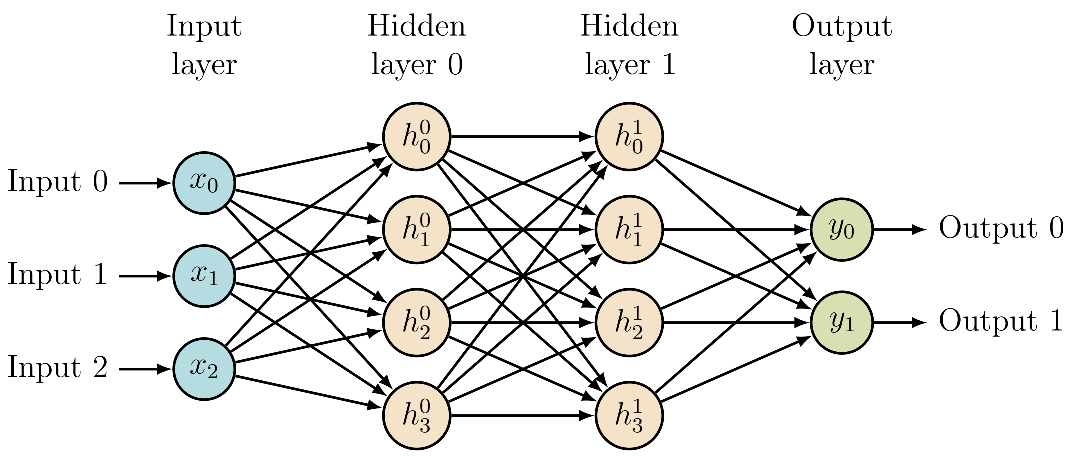


Figure 2.4: Multilayer Perceptron with two hidden layers

2.3.2 Recurrent Neural Networks

Feed-forward architectures cannot persist information about earlier inferences, hence are not able to handle sequences of temporal information such as text, audio, or video, which are common sources of data for many problems. Recurrent Neural Networks (RNNs), firstly described by Rumelhart *et al.* [25], is a special network architecture that solves this problem by adding recurrent connections between units to keep temporal information, allowing the network to reason about temporal sequences.

As learning through backpropagation requires the network to not contain cycles, a common technique is to unfold the network recurrence as if it were multiple feed-forward networks being trained in parallel. This concept of unfolding is named Backpropagation Through Time (BPTT) and is illustrated in Figure 2.5 with a small example containing a single recurrent connection. Unfolding the neural network many times can demand a lot of processing power, so it is also common to limit the time steps.

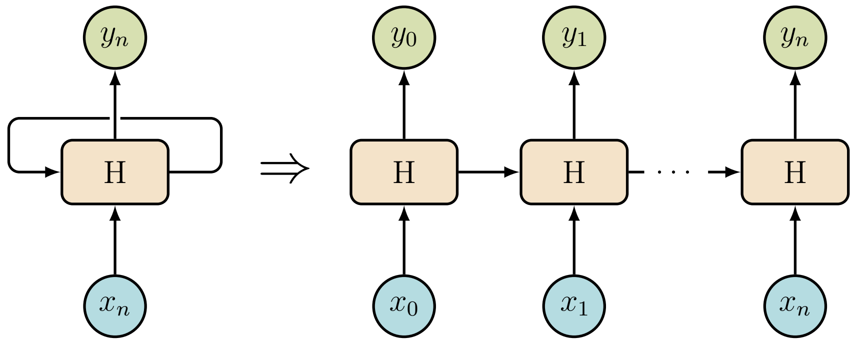


Figure 2.5: Unfolding of a Recurrent Neural Network

Although RNNs offer a solution that can work with temporal sequences of data, they are not good at dealing with long-term sequences, as the gradients are subject to vanish or explode during the training phase [15]. For this purpose, some RNN variants have been proposed, such as Long Term Short Memory (LSTM) and Gated Recurrent Unit (GRU), which can decide whether specific information should be remembered or not.

2.3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are feed-forward networks commonly used for image recognition. CNNs are inspired by how neurons are structured on the mammal's visual cortex. The main structures on CNNs are the convolutional and pooling layers.

Convolution is a linear operator commonly used in signal processing that can be understood as a measure of similarity between two functions or signals. In the context of image processing, the convolution operator is defined as follows:

$$(I * K)_{ij} = S_{ij} = \sum_m \sum_n I_{mn} \cdot K_{(i-m)(j-n)} \tag{2.1}$$

Where I is an input matrix with size $m \times n$ that represents an image and K is called a kernel or filter. The convolution of an image by a kernel will result in a third matrix which is the result of sliding the kernel over the image and then calculating the sum of the element-wise products between the image and the kernel elements. Figure 2.6 represents one step of the convolution between a 4×4 image and a 3×3 kernel, where each element in orange from the input matrix is multiplied by each element in blue from the kernel, whose sum results in the element in green at the resulting matrix.

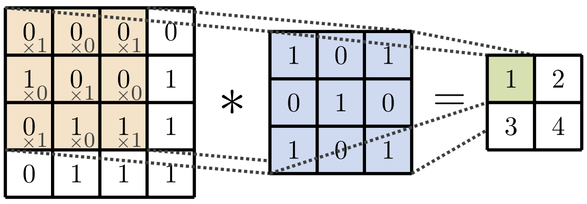


Figure 2.6: Convolution of a 3×3 kernel over a 4×4 matrix

The goal of convolutional layers is to learn how to extract features like patterns and shapes from images by adjusting the weights of their kernels, so the network can learn what features from an image are relevant when solving specific parts of a problem.

Pooling layers perform downsampling and are usually applied between convolutional layers to balance the network sensitivity concerning its parameters. Common pooling operations are the maximum and average pooling, that obtains the maximum and the average value of a sample region, respectively, as depicted in Figure 2.7.

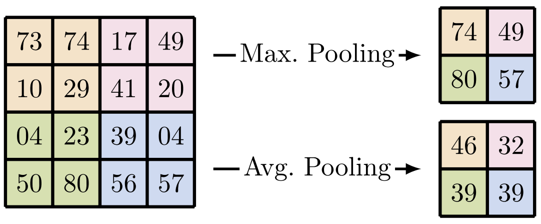


Figure 2.7: Maximum and average pooling

Traditional CNN architectures are usually composed of multiple groups of layers, each group containing a few convolutional layers followed by a pooling layer, with the last group composed of multiple fully-connected layers that are responsible for classification.

Chapter 3

Tensor Compilers

Tensor compilers are computer programs that can translate high level representations of tensor algebra operations into programs that can execute those operations over a given architecture. The concept was first implemented by Kjolstad *et al.* [16] in 2017 for general tensor algebra expressions, and later adopted by many deep learning frameworks.

This chapter is organized as follows. Section 3.1 introduces the concept of tensors and Section 3.2 gives a basic understanding on how tensor operations are translated into code.

3.1 Tensors

Tensors are algebraic objects that describe multi-linear relationships between sets of objects related to a vector space. They generalize scalars, vectors, and matrices under a framework that describes operations to manipulate objects of equal or different ranks.

The rank (or order) of a tensor is defined by the dimensions required to describe the object. Starting from a scalar (rank 0), one can inductively define vectors (rank 1) that contains one or more scalars, which can be further generalized into matrices (rank 2) that contains one or more vectors, up to any rank-N tensor, as depicted on Figure 3.1.

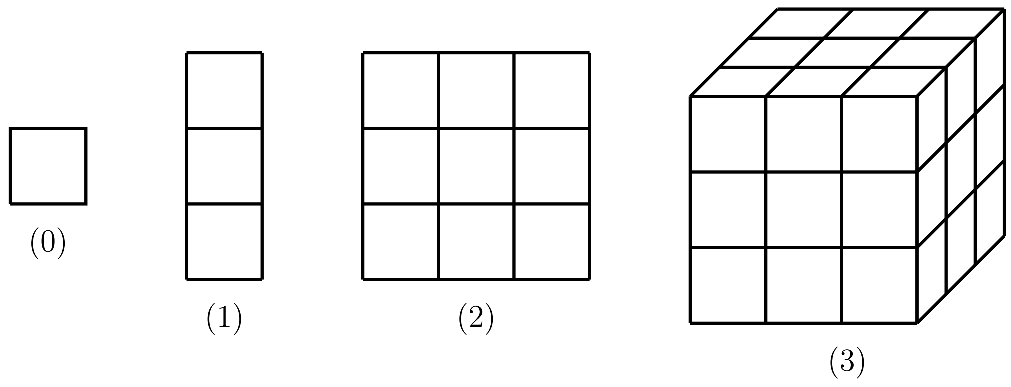


Figure 3.1: Representation of tensor dimensions with increasing ranks

The shape of a tensor is defined by a tuple $(s_0, s_1, \ldots, s_n)$ where each component s_i describes the size of the dimension i. For instance, a tensor with shape $(3, 1)$ is a rank-2 tensor (or matrix) with the first dimension with size 3 and a second dimension with size 1.

Tensor operations are usually represented through index notation, which is a compact way of representing how the elements of different tensors are operated with one another. Indices are represented as free variables associated with tensors within the expressions, describing the relation between different tensor dimensions. Also, rank-0 and rank-1 tensors are conventionally represented with lowercase letters, whereas higher-order tensors use uppercase letters. Some common operations from linear algebra that are useful on neural networks can be described using index notation, as represented in Table 3.1.

Operation	**Expression**
Vector-Scalar Addition	$a_i = b_i + c$
Element-wise Addition	$a_i = b_i + c_i$
Element-wise Multiplication	$a_i = b_i c_i$
Matrix-Vector Multiplication	$a_i = \sum_j B_{ij} c_j$
Matrix Multiplication	$A_{ij} = \sum_k B_{ik} C_{kj}$
Matrix Convolution	$A_{ij} = \sum_m \sum_n B_{mn} C_{(i-m)(j-n)}$

Table 3.1: Common tensor operations

3.2 Code Generation

To generate code that can compute a tensor expression for a given input, one must first identify how the indices are related to tensor dimensions, so then appropriate loop sequences can be generated to traverse the input data. The solution proposed by Kjolstad *et al.* [16] uses iteration graphs, which are directed graphs whose nodes represent indices and the paths formed by edges represent what indices relate to what dimensions in a tensor. Iteration graphs provide locality information for the code generation algorithm, so operations can be efficiently inserted within the nested loops that represent each associated index. Some examples of iteration graphs are shown in Figure 3.2.

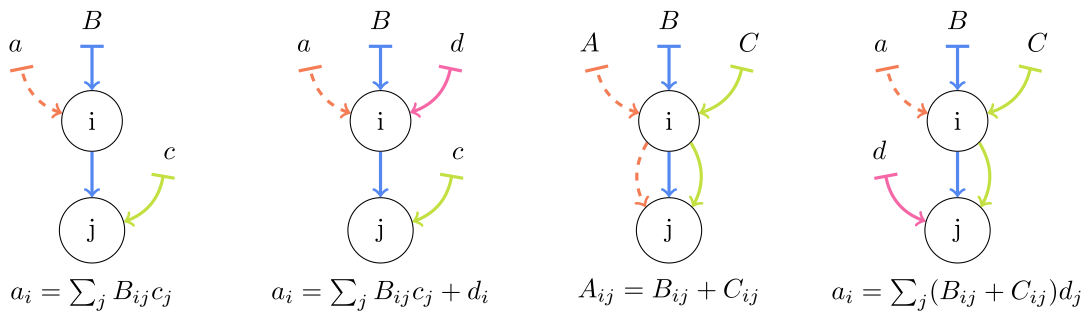
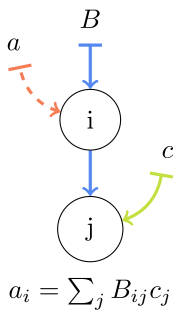

a
B
i
c
j
$a_i = \sum_j B_{ij} c_j$
a
B
d
i
c
j
$a_i = \sum_j B_{ij} c_j + d_i$
A
B
C
i
j
$A_{ij} = B_{ij} + C_{ij}$
a
B
C
i
d
j
$a_i = \sum_j (B_{ij} + C_{ij}) d_j$

Figure 3.2: Iteration graphs with corresponding expressions

Iteration graphs are constructed from a tensor expression by creating one path for every tensor indexing expression and adding new nodes for each new index found. For instance, an iteration graph for the expression B_{ij} should contain two nodes i and j, and a path B visiting i and then j. The tensor on the left side of the equation, which represents the main path, will determine the order of the generation of the loop sequences.

The code generation algorithm recursively visits each accessible node in the iteration graph starting with the first dimension of the main path. Then, a loop is emitted for each node visited in topological order, and also a summation variable if that index node is not part of the main path. Every operation for which all operands have already been indexed is then emitted in the node where the last operand was indexed in its last dimension. A representation of how code would be generated for an expression is shown in Figure 3.3.



$\Rightarrow$

```
for (int i = 0; i < size_i; ++i) {
  double sum_j = 0.0;
  for (int j = 0; j < size_j; ++j) {
    sum_j += B[i][j] * c[j];
  }
  a[i] = sum_j;
}
```

Figure 3.3: Code generation for a tensor expression using an iteration graph

Iteration graphs not only provide an intermediate representation that can be directly translated into loop sequences, but also implicitly perform code optimizations such as dead code elimination, common sub-expression elimination, operator fusion, and loop fusion. Due to the nature of the generated code, some aggressive optimizations can be applied to further improve the performance and resource usage on CPUs, GPUs, dedicated hardware accelerators, and embedded devices, some of which are described in Table 3.2.

Optimization	**Description**
Hardware Intrinsic Mapping	Replaces instructions by intrinsics of the target hardware.
Automatic Parallelization	Executes data-independent loops on multiple threads.
Memory Latency Hiding	Reorders instructions to hide memory access latencies.
Algebraic Simplification	Simplifies tensor expressions using algebraic properties like commutativity, associativity, and distributivity.
Strength Reduction	Replaces operations with cheaper ones whenever possible.
Constant Folding	Replaces constant variables and arrays by literal values.
Loop Reordering	Reorders nested loops to optimize spatial locality.
Loop Unrolling	Unrolls loop bodies into multiple copies to improve instruction-level parallelism and reduce loop branches.
Loop Tiling	Splits loops into several tiles with better data locality that fit into cache lines to reduce cache miss rates.

Table 3.2: Code optimization techniques

Chapter 4

Homomorphic Encryption

Homomorphic Encryption (HE) is a cryptographic solution that allows computations to be done while the data is still encrypted. Its theoretical foundation dates back to 1978 in a paper written by Rivest, Adleman, and Dertouzos [23]. However, the first practical fully homomorphic scheme was only proposed in 2009 by Gentry [11]. Since then, many HE schemes have been studied to provide solutions for privacy-preserving computation.

In this chapter, we introduce the fundamental concepts related to homomorphic encryption (Section 4.1); explain both the LWE and the RLWE problems (Section 4.2); and present the TFHE homomorphic encryption scheme (Section 4.3).

4.1 Fundamental Concepts

A homomorphism is defined as a map that preserves algebraic structures between the domain and the range of an algebraic set. In the context of homomorphic encryption, this map is a correspondence between operations on the plaintext and ciphertext spaces, in such a way that operations performed over the ciphertext space are somehow reflected as associated operations performed over the plaintext messages they encrypt.

An encryption scheme is homomorphic over an operation $\diamond \in \mathbb{F}_P$ in the plaintext space if there is an operation $\circ \in \mathbb{F}_C$ in the ciphertext space that satisfies the following equation:

$$E(m_1) \circ E(m_2) = E(m_1 \diamond m_2), \forall m_1, m_2 \in M \tag{4.1}$$

Where M is the set of all possible messages, and E is the encryption function.

Any homomorphic encryption scheme will, by design, contain an algorithm for encryption, decryption, key generation, and a map between operations over the plaintext and ciphertext spaces $\phi : \mathbb{F}_P \leftrightarrow \mathbb{F}_C$. An example of homomorphism over the multiplication operator is the RSA scheme. Given two messages $m_1, m_2 \in M$, the encryption function E and a public key $pk = (n, e)$, the following property holds:

$$E_{pk}(m_1) \times E_{pk}(m_2) = (m_1 \times m_2)^e \bmod n = E_{pk}(m_1 \times m_2) \tag{4.2}$$

Although RSA is homomorphic over multiplication, it would not be secure as an HE scheme, because an encrypted message would lead to the same ciphertext every time, making it vulnerable to the Chosen-Plaintext Attack (CPA).

To protect against CPA, most HE schemes introduce a controlled amount of random noise, so a plaintext message encrypted twice have a high probability of becoming different ciphertexts. However, the drawback is that the noise may accumulate with every operation, up to the point that the message is corrupted, therefore limiting the number of operations.

To have an encryption scheme that can evaluate arbitrary functions, it is sufficient for it to be homomorphic over addition and multiplication, as long as they can be performed an arbitrary number of times. This is because addition and multiplication forms a functionally complete set of operations, considering that any smooth function can be approximated by using polynomials and the only operations required to compute polynomials are addition and multiplication. Similarly, bitwise AND and XOR also forms a functionally complete set, as they can be thought of as multiplication and addition over $GF(2)$, respectively.

Schemes that are homomorphic over functionally incomplete operation sets are Partially HE (PHE) schemes; if they can evaluate arbitrary functions with limited depth, then they are Somewhat HE (SHE) or Leveled HE (LHE) schemes; if they implement functionally complete operation sets with unlimited depth, then they are Fully HE (FHE) schemes.

The first practical FHE scheme was achieved by Gentry [11] in 2009 with the concept of bootstrap. The general idea of bootstrapping is to build a SHE scheme with a simple enough decryption function, and then evaluate the decryption homomorphically to obtain a clean ciphertext, allowing the noise to be periodically reduced.

Recent homomorphic encryption schemes are mostly based on the Learning With Errors (LWE) problem or the Ring Learning With Errors (RLWE) variant, which are considered very hard problems in modern cryptography, being resistant even against quantum attacks.

4.2 Learning With Errors

Learning With Errors (LWE) is a computational problem about distinguishing random linear equations that were perturbed with a small amount of noise from truly uniform equations. LWE was introduced by Regev [22] in 2005, who was awarded with the Gödel Prize in 2018 for the same work. Regev showed that the LWE problem is as hard to solve as several worst-case lattice problems, being used as the foundation for many public-key cryptosystems, such as homomorphic encryption schemes and key exchange algorithms.

The notation used on the next definitions are given as follows. $\mathbb{B}$ denotes the set $\{0, 1\}$. $\mathbb{T}$ denotes the ring $\mathbb{R}/\mathbb{Z}$, or $\mathbb{R}$ mod 1. S_q denotes the set $\{s : s \in S, -\frac{q}{2} < s \leq \frac{q}{2}\}$. S^n denotes the set of n-vectors over a set S. $\Phi_{2^d}(x)$ denotes the 2^d-th cyclotomic polynomial. χ_S denotes a fixed probability distribution over samples in a set S. $S[x]$ denotes polynomials in x with coefficients belonging to a set S. $\mathcal{B}$, $\mathcal{R}$, $\mathcal{T}$, $\mathcal{Z}$, $\mathcal{Z}_q$ denotes the polynomial rings $\mathbb{B}[x]/\Phi_{2^d}(x)$, $\mathbb{R}[x]/\Phi_{2^d}(x)$, $\mathbb{T}[x]/\Phi_{2^d}(x)$, $\mathbb{Z}[x]/\Phi_{2^d}(x)$, $\mathbb{Z}_q[x]/\Phi_{2^d}(x)$, respectively.

The LWE problem is defined as follows. There exists an unknown linear function $f : \mathbb{Z}_q^n \to \mathbb{Z}_q$ in the form $f(x_i) = (s \cdot x_i) + e_i$, with $s \in \mathbb{Z}_q^n$ being the secret and $e_i \sim \chi_{\mathbb{Z}}$ a small amount of noise. Given a limited amount of sample pairs in the form (x_i, y_i) where $x_i \in \mathbb{Z}_q^n, y_i \in \mathbb{Z}_q$, with a high probability of $y_i = f(x_i)$, the LWE problem is described as:

- **Search LWE**: Find the value of s.
- **Decision LWE**: Distinguish between pairs from f and uniformly random pairs.

Ring Learning With Errors (RLWE) is an algebraic variant of LWE introduced by Lyubashevsky *et al.* [18]. The RLWE problem is defined as follows. There exists an unknown linear function $f : \mathcal{Z}_q \rightarrow \mathcal{Z}_q$ in the form $f(x_i) = (s \cdot x_i) + e_i$, with $s \in \mathcal{Z}_q$ being the secret and $e_i \sim \chi_{\mathcal{Z}}$ a small amount of noise. Given a limited amount of sample pairs in the form (x_i, y_i) where $x_i, y_i \in \mathcal{Z}_q$, with a high probability of $y_i = f(x_i)$, the RLWE problem can be described in both search and decision variants similarly to the LWE variants above.

4.3 TFHE Scheme

TFHE is a homomorphic encryption scheme proposed by Chillotti *et al.* [6] in 2016. It introduces a fast bootstrap operation compared to the previous state of art, as well as a generalized representation for both LWE and RLWE problems over the torus $\mathbb{T}$.

The TFHE scheme uses a generalized, scale-invariant definition of the LWE problem over the torus, which is defined as follows. There exists an unknown linear function $f : \mathbb{T}^n \rightarrow \mathbb{T}$ in the form $f(x_i) = (s \cdot x_i) + e_i$, with $s \in \mathbb{B}^n$ being the secret and $e_i \sim \chi_{\mathbb{R}}$ a small amount of noise. Given a limited amount of sample pairs in the form (x_i, y_i) where $x_i \in \mathbb{T}^n, y_i \in \mathbb{T}$, with a high probability of $y_i = f(x_i)$, the scale-invariant LWE problem can be described in both search and decision variants like LWE in the previous section.

Just like the original LWE definition, the TFHE scheme also presents a ring variant, named TLWE, which is defined as follows. There exists an unknown linear function $f : \mathcal{T}^n \rightarrow \mathcal{T}$ in the form $f(x_i) = (s \cdot x_i) + e_i$, with $s \in \mathcal{B}^n$ being the secret and $e_i \sim \chi_{\mathcal{R}}$ a small amount of noise. Given a limited amount of sample pairs in the form (x_i, y_i) where $x_i \in \mathcal{T}^n$, $y_i \in \mathcal{T}$, with a high probability of $y_i = f(x_i)$, the TLWE problem can be described in both search and decision variants similar to the previous definitions.

The basic symmetric encryption scheme using TLWE can be defined as:

- **Encryption**: Given a message encoded as a polynomial $\mu \in \mathcal{T}$, the encryption of μ under a secret s is obtained by $enc_s(\mu) = (x, y + \mu)$, where (x, y) is a sample pair generated from the function f.

- **Decryption**: Given a ciphertext in the form $c = (x, y)$, the approximate decryption of c using a secret s is obtained by $dec_s(x, y) = y - (s \cdot x)$. Note that a small amount of noise is still present in the decrypted message.

For applications that are not tolerant to the small amount of noise, one solution is to restrict the message space to a discrete subset, whose packing radius is larger than the amplitude of the error, and then rounding the values to retrieve the exact message.

Chapter 5

Related Works

Tensor compilers and homomorphically encrypted neural networks are recent topics in the literature. This chapter explores implementations of tensor compilers for machine learning, including compilers that can generate homomorphically encrypted models.

5.1 TensorFlow XLA

With the Google Brain project and the several breakthroughs in deep learning that were developed, multiple machine learning frameworks began to emerge with the goal of offering a general-purpose, high-performance architecture that could make use of different processing devices like CPUs and GPUs to train very large-scale models. TensorFlow [1] is an open-source machine learning framework developed by Google to meet this demand.

A TensorFlow computation is described by a directed acyclic graph that represents a dataflow computation, with extensions for allowing some kinds of nodes to maintain and update persistent state. In this graph, each node represents a tensor operation that can have any number of inputs and outputs, and the edges describe how the tensors flow through the operation graph. An example of a computation graph is given in Figure 5.1.

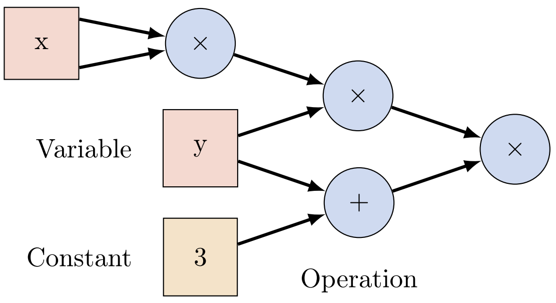


Figure 5.1: Simple computation graph

With the increasing popularity of TensorFlow as a general framework for machine learning, it has been observed that the execution model based on tensor graph interpretation is inefficient, so alternative backends based on just-in-time and ahead-of-time compilation has been proposed and developed. XLA [30], which stands for Accelerated Linear Algebra, is a tensor compiler backend developed to address such inefficiencies.

When a TensorFlow graph is run under the default execution model, every tensor operation is executed individually by dispatching a precompiled kernel to the target hardware. On XLA, the whole graph is translated into a single kernel that is dispatched once, not only reducing the bandwidth used for streaming intermediate computation, but also allowing aggressive and target-specific optimizations to be applied to the code.

XLA translates the tensor graph into High-Level Operations (HLO), which is an intermediate representation (IR) that describes tensor operations. After the translation, some optimization passes like constant folding, operation fusion, and dead code elimination are applied, resulting in a smaller representation which is then compiled into target code using an XLA backend. Figure 5.2 illustrates this process.

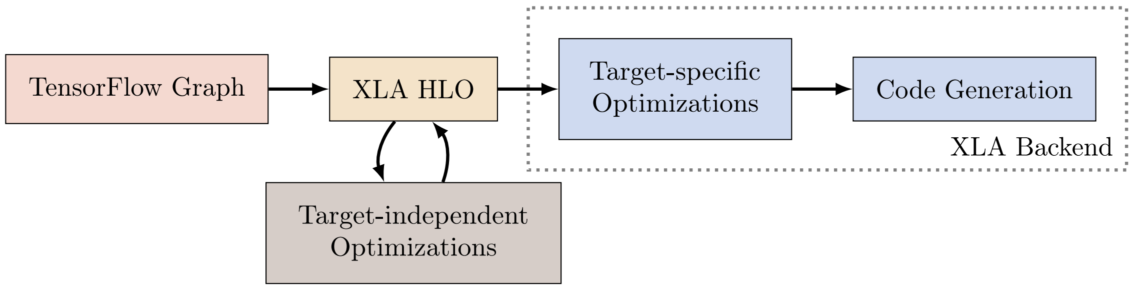

TensorFlow Graph
XLA HLO
Target-specific Optimizations
Code Generation
XLA Backend
Target-independent Optimizations

Figure 5.2: TensorFlow XLA pipeline

5.2 nGraph-HE

nGraph [8] is an open-source tensor compiler developed by Intel, based on their previous experience implementing machine learning libraries, that provides a framework-agnostic and platform-agnostic compiler, execution engine, and intermediate representation. Unlike TensorFlow XLA [30], nGraph supports multiple machine learning frameworks, intending to reduce the M-times-N complexity problem, on which M frameworks implement support for N platforms. An overview of nGraph pipeline is depicted in Figure 5.3.

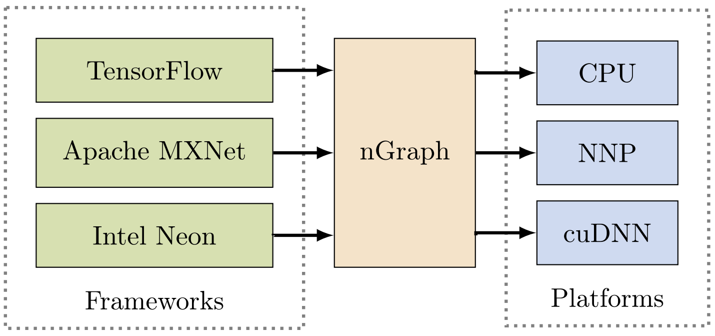


Figure 5.3: Intel nGraph pipeline

nGraph is divided into three components: the framework bridges, which translate computational graphs from other frameworks, the intermediate representation, and the transformers, which manages the code generation and execution over the platforms.

Their intermediate language, named nGraph IR, represents directed acyclic graphs of stateless tensor operation nodes, which may contain additional attributes and metadata that affect their behavior. nGraph IR is strongly typed regarding element types and tensor shapes and does not require the types to be annotated outside boundaries. Another characteristic is that the operation nodes do not enforce a specific tensor layout, allowing the layouts to be dynamically inferred based on the inputs.

With the increasing concern about data privacy and recent improvements on homomorphic encryption algorithms, Boemer *et al.* implemented nGraph-HE [3], which is a nGraph extension that can generate homomorphically encrypted network models.

Behaving as a target platform for nGraph, nGraph-HE is originally based on SEAL [26] and supports both BFV [10] and CKKS [5] schemes in the leveled setting, which are used for integer and fixed-point arithmetic operations, respectively. Like nGraph, it is also designed to be scheme-agnostic, so other libraries and schemes can be supported.

nGraph-HE is not only able to use optimizations available on nGraph like operator fusion, constant folding, and data layout optimizations, but it also introduces several scheme-specific optimizations, which includes special plaintext value bypass, HE-SIMD packing, plaintext operations, and graph-level optimizations.

As a follow-up, Boemer *et al.* implemented nGraph-HE2 [2], which focuses on the CKKS scheme and introduces new optimization techniques that achieve *2.6x-4.2x* speedups when compared to nGraph-HE. It includes two graph-level optimizations that are depth-aware encoding and lazy rescaling, as well as scheme-specific optimizations on both ciphertext-plaintext addition and multiplication.

Another feature introduced in nGraph-HE2 is the client-aided model, which is a hybrid two-party computation approach that avoids the costly CKKS bootstrapping process by sending intermediate computations back and forth between the machine running the inference model and a trusted machine that possesses the secret key to remove the excessive noise by re-encrypting the data.

5.3 CHET

CHET [9] is a proprietary tensor compiler developed by Microsoft, designed to automate the implementation of homomorphically encrypted neural networks. Based on the SEAL [26] library, also developed by Microsoft, it supports the CKKS [5] encryption scheme, as well as a domain-specific language for specifying tensor circuits and an interface that can support multiple encryption libraries. An overview is shown in Figure 5.4.

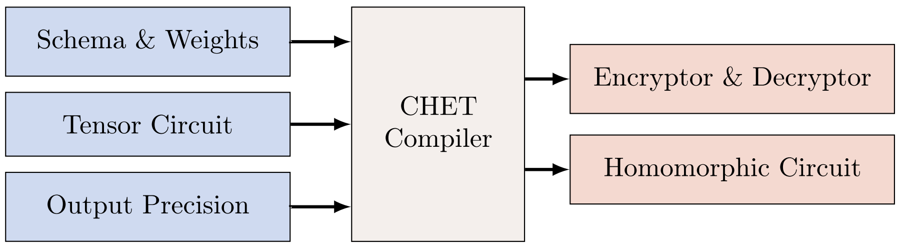


Figure 5.4: Overview of the CHET compiler

Given a high-level representation of the tensor circuit, CHET translates this representation into a Homomorphic Tensor Circuit (HTC), which includes characteristics relevant to the encryption scheme within the tensor layouts and operations. Then, it compiles the corresponding circuit to an interface named Homomorphic Instruction Set Architecture (HISA), which provides building blocks for multiple encryption schemes.

An important aspect of HISA is that it is not meant to provide a unified and homogeneous interface over multiple schemes as one would expect from an intermediate representation. Instead, HISA provides multiple instruction sets, also known as profiles, that account for common operation groups that are available on different schemes. Profiles may or may not be implemented for particular schemes, similarly as architectural extensions may only be available on particular CPU models. A brief description of those profiles is presented in Table 5.1.

Profile	**Description**
Encryption	Provides plaintext encryption and decryption, and instructions to copy ciphertext and free unused resources.
Integers	Provides integer vector encoding and decoding, and arithmetic operations for ciphertext, plaintext, and scalar values (except division).
Division	Provides arithmetic division by scalars and largest divisor operator.
Relin	Provides relinearization and multiplication without relinearization.
Bootstrap	Provides bootstrapping operation.

Table 5.1: HISA profiles

Further contributions of the CHET compiler are the optimization techniques applied over the generated models, which includes automatic tuning of the encryption parameters to balance between security and accuracy, determining efficient tensor layouts to minimize the number of operations and performing CKKS scheme-specific optimizations.

Chapter 6

Objectives and Methodology

This proposal aims to investigate the practical implications of homomorphic evaluation of tensor models using the TFHE encryption scheme alongside with recent optimizations that can potentially improve the performance by implementing a tensor compiler to automate the task. To meet this goal, we propose the following steps:

- Choose an existing tensor compiler framework that already compiles to homomorphic encryption schemes and then reproduce the results with an existing backend;
- Implement a tensor compiler backend using TFHE and evaluate the results against existing implementations to compare performance and resources usage;
- Implement optimizations on the novel backend to generate performatic code by experimenting with different parameters and techniques.

6.1 Methodology

To have a reference algorithm to compare with the novel implementation, we are going to choose a tensor compiler framework, taking as initial consideration TensorFlow XLA [30] with nGraph-HE [3] backend as it already implements both BFV [10] and CKKS [5] schemes, but still considering alternatives like nGraph-HE2 [2] and CHET [9]. The framework will be selected primarily considering the availability of HE schemes to compare against our work, but other aspects like performance and compatibility shall be considered as well.

The implementation will be divided into two steps (#1 and #2). In the first step (#1), we will implement a tensor compiler backend that can generate code containing general-purpose optimizations provided by the target compiler, which should be LLVM or GCC. In the second step (#2), we are going to implement domain-specific optimizations that can further explore characteristics of the TFHE scheme. We plan to generate the models using lookup tables (LUTs) to perform fast computation of homomorphic operations, based on the work being developed by Guimarães, Borin, and Aranha [13].

Among the domain-specific optimizations to be considered on the second step, we are going to evaluate changes to the numeric base and precision, bootstrap depth, lookup table radix [13], LogNet [17], multi-value bootstrapping [4], and homomorphic ring decryption [20], with that order of priority. The optimizations are meant to be parametric with heuristic based defaults, so users can override the compilers judgement whenever necessary.

6.2 Evaluation

To evaluate the implementation against existing homomorphic compiler backends, we are going to use small convolutional neural network architectures such as LeNet-5 and SqueezeNet with the MNIST and CIFAR10 datasets. Those architectures were chosen considering previous attempts in the literature to run HE models, so they are known to work on existing backends and should have no issues related to the multiplicative depth for schemes that do not perform bootstrap. The models should be compared in terms of:

- **Performance**: How much time does it take to run each model?
- **Energy Consumption**: How much energy is required to run each model?
- **Memory Consumption**: How much memory does it need to run each model?
- **Accuracy**: How well is the accuracy on both encrypted and unencrypted models?
- **Loss**: How loss values compare between encrypted and unencrypted models?

To answer these questions, we are going to run encrypted and unencrypted models for each architecture and dataset configuration multiple times (to be defined based on the running time of the slower configuration), and then calculate the average, median, and standard deviation for each criterion above. The same initialization weights and loss functions should be used for each configuration to understand how the random noise on encrypted models can affect the results compared to unencrypted models. The same procedure will be used to evaluate both implementations #1 and #2.

6.3 Schedule

The tasks listed below will be carried out during the master's course in computer science at the Institute of Computing.

1. **Class Requirements**: Obtain 22 credits on disciplines in the master's course.
2. **Teaching Internship**: Carry out activities in the didactic internship program.
3. **Literature Review**: Studying and reviewing related publications.
4. **Qualification Exam**: Writing the proposal and taking the Qualification Exam.
5. **Results Reproduction**: Reproduce previous results in the literature to help understand the details and limitations of the existing compiler backends.
6. **Framework Selection**: Select the framework where the base compiler backend will be implemented based on the results obtained on task 5.

7. **Implementation #1**: Implement the base compiler backend without the domain-specific optimizations.
8. **Experiments #1**: Experiment with the implementation from task 7 to obtain preliminary results and identify improvements for the next iteration.
9. **Article #1**: Write an article with an analysis of the preliminary results.
10. **Implementation #2**: Implement the optimized compiler backend containing domain-specific optimizations and improvements identified on task 8.
11. **Experiments #2**: Experiment with the implementation from task 10 to obtain the final results and compare against results from task 8.
12. **Article #2**: Write an article with an analysis of the final results.
13. **Dissertation Synthesis**: Write the dissertation with results from earlier tasks.
14. **Revision and Defense**: Review and defend the dissertation.

The schedule containing the estimated time for each task is shown in Figure 6.1.

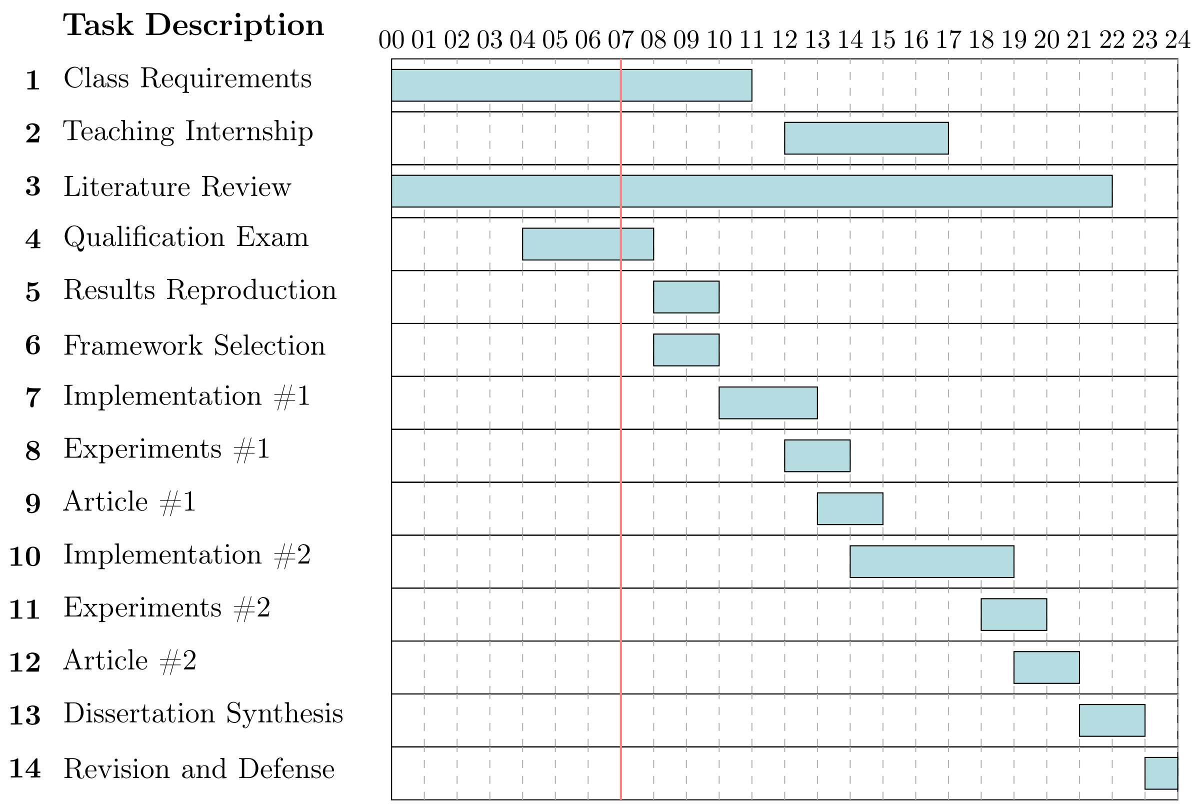


Figure 6.1: Research schedule

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Fabian Boemer, Anamaria Costache, Rosario Cammarota, et al. NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, pages 45—-56, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Fabian Boemer, Yixing Lao, Rosario Cammarota, et al. NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, pages 3—-13, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New Techniques for Multi-value Input Homomorphic Evaluation and Applications. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 106–126, Cham, 2019. Springer International Publishing.

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, et al. Homomorphic Encryption for Arithmetic of Approximate Numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, et al. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[7] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. *IACR Cryptology ePrint Archive*, 2021:91–109, January 2021.

[8] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, et al. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv*, 1801.08058, January 2018.

[9] Roshan Dathathri, Olli Saarikivi, Hao Chen, et al. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 142—156, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive*, 2012:144–163, March 2012.

[11] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169—-178, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):229–253, February 2021.

[14] HElib: An Implementation of Homomorphic Encryption. `https://github.com/homenc/HElib`, September 2014. Access on 01/03/2021.

[15] Fakultit Informatik, Y. Bengio, Paolo Frasconi, et al. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*, pages 237–243. Wiley-IEEE Press, 2003.

[16] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, et al. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages*, 1(1), October 2017.

[17] Edward H. Lee, Daisuke Miyashita, Elaina Chai, et al. LogNet: Energy-Efficient Neural Networks Using Logarithmic Computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904, March 2017.

[18] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. *Journal of the ACM*, 60(6), November 2013.

[19] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, December 1943.

[20] Daniele Micciancio and Jessica Sorrell. Ring Packing and Amortized FHEW Bootstrapping. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 100:1–100:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[21] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, April 2020.

[22] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84—93, New York, NY, USA, 2005. Association for Computing Machinery.

[23] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. *On Data Banks and Privacy Homomorphisms*, pages 169–179. Academia Press, 1978.

[24] Frank Rosenblatt. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408, November 1958.

[25] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, pages 696—-699. MIT Press, 1988.

[26] Microsoft SEAL. `https://github.com/Microsoft/SEAL`, November 2020. Access on 01/03/2021.

[27] Himanshu Shekhar, Sujoy Seal, Saket Kedia, et al. Survey on Applications of Machine Learning in the Field of Computer Vision. In Jyotsna Kumar Mandal and Debika Bhattacharya, editors, *Emerging Technology in Modelling and Graphics*, pages 667–678, Singapore, 2020. Springer Singapore.

[28] Shiliang Sun, Zehui Cao, Han Zhu, et al. A Survey of Optimization Methods From a Machine Learning Perspective. *IEEE Transactions on Cybernetics*, 50(8):3668–3681, November 2020.

[29] Qi Wang, Yue Ma, Kun Zhao, et al. A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, April 2020.

[30] XLA: Optimizing Compiler for Machine Learning. `https://www.tensorflow.org/xla`, March 2017. Access on 01/03/2021.

[31] Shuai Zhang, Lina Yao, Aixin Sun, et al. Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Computing Surveys*, 52(1), February 2019.