Enabling Large-Scale Privacy-Preserving Recurrent Neural Networks with Fully Homomorphic Encryption

by

Vele Tosevski

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto

Enabling Large-Scale Privacy-Preserving Recurrent Neural Networks with Fully Homomorphic Encryption

Vele Tosevski
Master of Applied Science
Department of Electrical and Computer Engineering
University of Toronto
2023

Abstract

Fully homomorphic encryption (FHE) can be used to secure a variety of neural networks. We show that FHE can be applied to recurrent neural networks (RNNs) and used for image classification and speaker identification. Using the CGGI [25] mathematical foundation of FHE encryption, we develop several methods to aid in converting plaintext RNNs into CGGI-compatible RNNs. A requirement to utilize CGGI is to quantize RNNs into integers, which is a difficult task. One of the contributions in this thesis is a novel quantization procedure for RNNs that successfully quantizes large-scale RNNs greater than 1M parameters into ternarized parameters and binarized activations. In order to mitigate numeric overflow, this thesis proposes a novel regularization method called the Overflow-Aware Activity Regularizer (OAR) that teaches RNN networks to output correct values, regardless of the existence of overflow. Another contribution is the addition of an attention layer to encrypted RNNs. To the best of our knowledge, this is the first work to successfully incorporate attention by proposing a novel ciphertext-ciphertext multiplication method that simultaneously reduces the amount of computation required by half.

We develop two ground-breaking, large-scale, privacy-preserving RNNs with multiple RNN layers. The first, with around 2M parameters, achieves 2s latency over the MNIST dataset using two Nvidia A100 GPUs, which is three orders of magnitude better than the state-of-the-art, SHE [78]. By utilizing OAR, nearly all activations between plaintext and encrypted runs are the same. The second, with around 13M parameters and an attention layer, correctly classifies encrypted speech sequences up to 426 timesteps in length, one order of magnitude longer than SHE.

To those that believe in the value of privacy.

Acknowledgements

First and foremost, I would like to thank my supervisor, Glenn Gulak, for always believing in my work and for his enduring guidance and care, making himself available for any questions and concerns at all times. I would also like to thank the team at Lorica for their continuous support and for giving me the time to complete this thesis. I am sure that this work will advance our mission of protecting what is increasingly becoming our most valuable resource—data. Lastly and most dearly, I would like to thank my family for giving me the courage and strength to do anything I set my mind to, which includes this most challenging and rewarding experience of my life thus far. This thesis only affirms what I know to be true, and that is: with hard work (and a bit of music), you can accomplish anything.

Contents

List of Tables

List of Figures

List of Acronyms

(D)NN	(Deep) Neural Network
(F)(L)HE	(Fully) (Levelled) Homomorphic Encryption
(R)LWE	(Ring) Learning With Errors
BNN	Binarized Neural Networks [29]
CGGI	Chillotti-Gama-Georgieva-Izabachène [25]
CNN	Convolutional Neural Network
EMA	Exponential Moving Average
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
KS	Key-Switch
LAB	Loss-Aware Binarization [57]
LSTM	Long Short-Term Memory
LUT	Lookup Table
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology [66]
OAR	Overflow-Aware Activity Regularization
PBS	Programmable Bootstrap
PPML	Privacy-Preserving Machine Learning
PTQ	Post-Training Quantization

QAT	Quantization-Aware Training
RNN	Recurrent Neural Network
SMC	Secure Multi-Party Computation
SOTA	State-Of-The-Art
STE	Straight Through Estimator [9]
TFHE	Fully Homomorphic Encryption over the Torus (see CGGI)
TWN	Ternary Weight Networks [72]
WRPN	Wide Reduced-Precision Networks [87]

Notation

$[\cdot]_{\text{mod } k}$	Modulo operator for any modulus k
$\cdot \bmod k$	Modulo operator for any modulus k
λ	FHE security parameter
$\langle \cdot \rangle$	Dot product operation
$\lceil \cdot \rfloor$	Rounding operation to the nearest integer
$\mathbb{B}$	Subset of integers equal to $\{0, 1\}$
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution with mean μ and standard deviation σ
$\odot$	External multiplication between two ciphertexts
$\oplus$	Addition between two ciphertexts
$\otimes$	Scalar multiplication between any integer and ciphertext
$\phi(x) = x^N + 1$	M-th cyclotomic polynomial with degree N
$\chi(S)$	Represents a distribution of χ over set S
p	Refers to the plaintext modulus (unless stated otherwise)
$R_q = \mathbb{Z}_q(x)/(x^N + 1)$	Algebraic ring over the polynomials with coefficients from the group of integers modulo q, where N is a positive, integer power-of-two
$S(x)$	Polynomial with coefficients in set S
$U(S)$	Uniform distribution over set S
$x \sim \mathcal{P}$	Element x is sampled from probability distribution $\mathcal{P}$
$\mathbf{W}$	Uppercase variables in bold are matrices
$\mathbf{w}$	Lowercase variables in bold are vectors

w	Lowercase variables are scalars
LSB	Least Significant Bits
MSB	Most Significant Bits
Multi-CT	Encryption of every bit of the bit-decomposition of an integer
Single-CT	Encryption of an integer using one ciphertext

Chapter 1

Introduction

1.1 Motivation

Machine learning as a service has become a large, global industry in the past 20 years. As computer hardware becomes more advanced and data becomes richer and more available, the demand for advanced tools such as online chatbots and personal assistants that are powered by artificial intelligence (AI), becomes larger. As a result, the need to protect the privacy of data at rest, in transit, and during computation becomes even more important. Fully homomorphic encryption (FHE) is a privacy enhancing technology that can be used to protect data during computation, allowing the same services mentioned above to be performed over encrypted data. For instance, FHE enables the ability to send encrypted samples of speech to third-party voice assistants, with the assurance that the data can never be observed in the clear, and receive an encrypted answer that only the initial sender can decrypt.

While research predominantly surrounds the execution of prominent neural networks over encrypted data, such as convolutional neural networks and simple logistic regression [99], there is limited research in the area of recurrent neural networks (RNNs) [78, 101]. The characteristics of RNNs that make them powerful over sequential data serve as limitations in most FHE schemes. Primarily, RNNs are some of the deepest types of neural networks, since they are executed over sequential data of variable length. In FHE, the underlying security guarantees rely on the presence of error in each ciphertext [109], which grows with each consecutive mathematical operation, and can grow to a point where the ciphertext can no longer be decrypted properly. Consequently, even short RNN architectures are difficult to implement. Secondly, FHE operations continue to be orders of magnitude slower than their plaintext counterparts, making RNNs, which are usually time-sensitive, simply too inefficient to be useful. In response, this thesis strives to break down these barriers and enable the execution of large-scale RNNs over encrypted data. Specifically, we investigate RNN inference using FHE, which is the computational circuit that outputs encrypted predictions when given encrypted data.

1.2 Objectives

The core objective of this thesis is to provide techniques that overcome some of the limitations that have hindered the progress of achieving practical RNN inference over encrypted data. To tackle the problem of large execution latency, one of the requirements of this thesis is to utilize parallelization methods, specifically GPU acceleration, that can increase the efficiency of each FHE operation. Another requirement is to also investigate non-interactive execution of RNNs over encrypted data. Since RNN sequences can be very large, interactive solutions can dramatically increase communicational overhead to undesirable levels.

We focus on the CGGI FHE scheme [25] since it possesses the most efficient bootstrapping method (a way to refresh the error in ciphertexts). This allows us to theoretically unlock unlimited depth, a necessary characteristic of RNNs, while minimizing the latency. As a by-product of using CGGI, we must investigate ways to quantize RNNs, which is a difficult task in and of itself [73]. As a result, the objectives of this thesis include (1) proposing a way to successfully quantize RNNs into low-bit precision integers, (2) expanding the scale of RNNs to greater than 1,000,000 parameter models capable of processing sequences of greater than 25 timesteps in length, and preferably in the hundreds, and (3) incorporating attention for the first time over encrypted data to expand the types of RNNs that can be evaluated. In the process, we propose a novel method of RNN quantization, a novel multiplication method for ciphertext-ciphertext multiplication that is more efficient than the state-of-the-art, allowing us to perform attention, and a novel regularization method to reduce the necessary precision levels in RNNs, enabling the use of more efficient security parameters. Together, these techniques allow us to achieve a new state-of-the-art for evaluating large-scale RNNs over encrypted data.

1.3 Outline Of Thesis

Chapter 2 introduces important background information. We first discuss machine learning and continue to the specifics of recurrent neural networks. Then, we introduce fully homomorphic encryption followed by specifics of the CGGI scheme to understand available operations and their limitations. CGGI is limited to integer arithmetic, hence low-bit precision quantization methods are a topic of focus. In chapter 3, we introduce and evaluate the first contribution of this thesis, the Overflow-Aware Activity Regularizer (OAR), which is one of the key methods in enabling faster and more precise RNN inference over encrypted data. In chapter 4, we introduce our novel RNN quantization method, several techniques to convert quantized RNNS to CGGI-compatible RNNs, and two novel large-scale RNN architectures, one with two million parameters, and the other with twelve million parameters and using attention. We conclude chapter 4 by demonstrating our results and finally summarize contributions and topics for future investigation in chapter 5.

Chapter 2

Background

2.1 Machine Learning

Machine Learning (ML) has become a broad field over the past 50 years. Engineers, mathematicians, and scientists in a variety of disciplines such as computer hardware, software, and statistics have enabled giant leaps in the path towards general artificial intelligence (AI) through their contributions to the ML field. At its most fundamental level, ML encompasses a set of methods that utilize patterns in data to adjust a computers "knowledge" about the environment and a problem context. A computer can then use this "knowledge" to perform a task such as classifying a group of images [64] or predicting stock prices [124]. Generally, "knowledge" is in the form of a set of numbers called parameters, θ. Parameters and their interaction with input data define a model that is then used by the computer to perform tasks.

According to [123], an ML model is learned through a general, statistical process. First, the learner is provided with an arbitrary **domain set**, $\mathcal{X}$. This set contains the objects we want to label. For example, if we want to distinguish between cats and dogs, a possible domain set could contain images of cats and dogs. It could also contain recordings of cat and dog sounds. Elements in this set are normally vectors of data points commonly referred to as *features*. A learner is also provided with a **label set**, $\mathcal{Y}$. This set contains labels that can be assigned to elements of $\mathcal{X}$. For example, when distinguishing between cats and dogs, there could be two labels,

$$\mathcal{Y} = \{\text{“}cat\text{”} = 0, \text{“}dog\text{”} = 1\},$$

or three labels,

$$\mathcal{Y} = \{\text{“}cat\text{”} = 0, \text{“}dog\text{”} = 1, \text{“}unknown\text{”} = 2\},$$

or any other variation.

The possible combination of these two sets provides the learner with a **training set**, $\mathcal{S}$. A training set is a sequence of elements from $\mathcal{X}$ that may or may not be paired with

corresponding elements from $\mathcal{Y}$. For learning patterns without labels,

$$\mathcal{S} = \mathcal{X} = \{x_0, x_1, ..., x_{n-1}\},$$

and for learning patterns with labels,

$$\mathcal{S} = \mathcal{X} \times \mathcal{Y} = \{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}.$$

We refer to members of a training set as *training examples*. A training set is normally broken down into three subsets: *training*, *validation*, and *test*. The training set is used for updating model parameters during training, the validation set is used for guiding the training process, and the test set is used for evaluating the generalization power of the model. During the training process, the model accesses the training and validation sets but not the test set. This allows the learner to evaluate how well the model performs with new examples or in other words, how well it generalizes.

With every training example, a model performs a task. Therefore, the learner must provide an **output rule**, $\mathcal{A}(\mathcal{S})$. The output rule $\mathcal{A}$ is a function with an output dependent on a training set $\mathcal{S}$. The function could be a *classifier* where out of n classes, $\mathcal{A}$ outputs one of the n classes classifying the input training example from $\mathcal{S}$. As an illustration, $\mathcal{A}$ outputs numeral "1" in response to an input sequence of pixels depicting a handwritten numeral "1". The output rule $\mathcal{A}$ could also be a *regressor* where it outputs a continuous value. For example, given an input sequence of stock prices over one month, the model $\mathcal{A}$ outputs the predicted next-day stock price for a specific stock. $\mathcal{A}$ is also commonly referred to as a **forward step** since it moves an input *forward* through a series of transformations towards an output—the output rule.

Finally, a learner must provide at least one **measure of success**. This is a function that numerically measures how well a model performs a task it is being trained to perform. It is used to update the model during training and to track model performance throughout the training process. Normally, measures of success are placed at the end of the forward step to evaluate how different the output of the model is from the label or target. Theoretically, they could be placed at any point in the forward step to evaluate and augment certain parts of the model towards any definition of success. We refer to these as *local* measures of success. Activity regularization is an illustrative approach that leverages measures of success specific to the outputs of the layers in a model, thereby augmenting its capacity for generalization [85]. We refer to measures of success as **metrics**. An example of a basic metric for a classifier is **accuracy**. Accuracy is defined as the percentage of correct classifications.

During training, a **measure of error** is used to propagate updates to model parameters. A measure of error is a measure of success where a lower value is desired. We refer to a measure of error for a single example as a **loss function**, $\mathcal{L}(x; \theta)$, where $x \in \mathcal{S}$, and a

measure of error for multiple examples as a **cost function**, $\mathcal{C}(\{x_0, x_1, ..., x_{n-1}\}; \theta)$, where $\{x_0, x_1, ..., x_{n-1}\} \in \mathcal{S}$. An example of a loss function is the multi-class cross-entropy loss function,

$$\mathcal{L}_{CE} = -\sum_{c=0}^{n-1} t_c \log(y_c),$$

where c is one class out of n classes, t_c is the true label for the current example, and y_c is the probability produced by the model for class c. Since a classifier needs to output correct answers confidently, $\mathcal{L}_{CE}$ is designed to exponentially penalize highly-confident, incorrect predictions.

This summarizes a general learning process. There are many possible variations of the above methods since different tasks require different datasets, output rules and measures of success. Variations are generally grouped into three different learning paradigms: supervised, unsupervised, and reinforcement learning. A **supervised learning** process consists of a training dataset whose elements are matched with corresponding target labels. Classification and regression problems normally fall within the supervised learning paradigm since the goal of these problems is to output a correct target label. The algorithm teaching the learner in this case takes the role of an external supervisor with complete knowledge about the problem context. An **unsupervised learning** process consists of a training dataset without labels. The goal of unsupervised learning problems is to extract patterns from unlabelled data. Clustering, projection, and density estimation problems fall within the unsupervised learning paradigm [11]. For instance, the goal of clustering problems is to find commonalities between elements of a dataset by finding higher density clusters in vector space. A popular clustering algorithm is the global k-means algorithm that partitions an unlabelled dataset into k different clusters by learning k mathematical means of the unlabelled data [75]. **Reinforcement learning** (RL) processes train their models to maximise an expected reward signal gained from interacting with an environment over a period of time. An environment consists of a state that can be changed through a set of possible actions. State-action pairs produce a change in the environment transitioning the learner to a new state with a certain probability. A transition in state due to an action sends an immediate reward to the learner. RL algorithms usually begin by defining this process, which is commonly referred to as a Markov decision process (MDP) [127]. Algorithms such as Deep Q-Learning [88] have been successfully used to autonomously play Atari games, outperforming all previous methods (which include dynamic programming and linear solutions). While the learning processes may differ, the digital representation of models remains fairly constant. The most popular representation of a model is an artificial neural network, simply referred to as a **neural network** (NN).

2.1.1 Neural Networks

Neural networks are modelled after the human brain. At a very abstract level, neuronal cells accept inputs from other cells, aggregate the inputs, and fire based on the aggregated signals they receive. The signals produced by neurons are transmitted to other neurons forming a network of nodes (neurons) and edges (signals), similar to a graph structure. McCulloch and Pitts were the first to model electric circuits after this structure in 1943 [84]. Neurons in **McCulloch-Pitts (M-P) networks** consist of a weighted summation of binary input signals and a threshold activation function. The weighted summation passes through the activation function which outputs $+1$ for values greater than or equal to a defined threshold, or 0 for values less than the threshold. The values of the weights could be $+1$, denoting an excitatory signal, and -1, an inhibitory signal. The summation therefore determines whether the neuron is excitatory or inhibitory. M-P networks have been used to create simple circuits of common logic gates such as AND and OR gates [84]. However, the M-P model suffers from (1) an overly-simplified binarization of inputs and outputs causing it to be less expressive than modern NNs, (2) linear inseparability of non-linear functions such as the XOR function (common to all single layered networks), and (3) presetting of weights/threshold values making it a network that does not *learn*.

One of the first proposals of a learning rule for artificial neural networks was by Hebb in 1949 [51]. Hebb proposed that each connection between two neurons has a weight directly proportional to the product of the outputs of the neurons. In other words, if both neurons fire, then the strength of their connection should increase. Nearly a decade later, Rosenblatt was motivated by these ideas and developed the **perceptron**, which has since formed the basis of contemporary neural networks [116]. The perceptron model formulated by Rosenblatt is an expanded adaptation of the M-P model, incorporating a learning rule. It incorporates real-valued inputs, outputs, and weights and was the first to utilize an error signal derived from the predicted class output to modify weights during training. In a book published in 1962, Rosenblatt expanded on the perceptron and its applications in the fields of character and speech recognition [115]. Around the same time, Widrow and Hoff proposed *ADALINE* [132], a similar network to the perceptron whose training algorithm is a special case of the foundational **stochastic gradient descent** algorithm. These major contributions helped ignite the field of neural networks in the late 1950s and kept it alive until the late 1960s when Minsky and Papert showed that the perceptron suffered from linear inseparability, a result mostly contributed to by its shallow, one-layer structure [86]. They provided evidence that the perceptron cannot model the XOR function, a fairly simple function in the field of electronics. This caused the ML field to enter a decade of stagnation, largely due to a lack of funding for further research.

The introduction of backpropagation as a method to train neural networks in the mid-1980s helped re-energize the machine learning field. Although several authors can be credited for advancing the discovery of backpropagation [62, 77, 131], *Rumelhart et al.* were the

first to present a concrete mathematical framework for applying backpropagation to several neural networks including multilayered perceptrons with supporting experimental results [117].

Backpropagation, although proven to be incredibly inefficient and an incorrect representation of how the brain learns [52], remains the standard way to train neural networks to this day. After the network completes a forward pass, it compares the output with the target output and calculates an error function. During the backward pass, the network calculates the gradients of the error function with respect to each parameter and using the gradients, updates the parameters through the stochastic gradient descent process. The network calculates the gradients through a technique called **automatic differentiation**. Since the forward pass is a composition of multiple functions, automatic differentiation uses the chain rule to compute the partial derivatives of each composition starting from the loss function and moving backwards towards each parameter, while storing computed partials for efficient calculations of future partials.

2.1.2 Deep Neural Networks

The successful training of multilayered perceptrons using backpropagation gave rise to the era of **deep learning**. Multilayered perceptrons combine multiple layers of perceptrons by feeding the outputs of one layer into the inputs of the next. This creates additional internal representations between input and output layers in neural networks called hidden layers. As a result, hidden layers enable neural networks to model non-linear relationships between data inputs and outputs. A neural network is considered a deep neural network when it contains at least one hidden layer. Hidden layers caused neural networks to become more powerful. With a concrete learning procedure, researchers were enabled to start thinking about neural networks as hierarchical systems where different types of layers, designed to detect different patterns (i.e. visual, temporal), could be chained to better model more complex statistical relationships. This led to the development of modern DNN architectures in the late 1980's to late 1990's. For example, **convolutional neural networks** (CNNs) were created to mimic the way the human visual cortex analyzes images to learn patterns in images. At their core, CNNs utilize the mathematical convolution function between images and learned filters to identify patterns in an image. Filters, or kernels, are sets of shared parameters, usually structured as square matrices. The filtered outputs are sent forward through a non-linear activation function to a possible pooling layer. Pooling layers aggregate groups of values using functions like the average or maximum to achieve better generalization. Additional convolutional and pooling layers are chained and fed into a final layer. The structure of the final layer depends on the task. For example, if the task is to classify images then the final layer could be a simple linear classifier. Fukushima was the first to introduce convolution and pooling in neural networks with the *Neocognitron* in 1980 [35]. Inspired by Fukushima's work, LeCun trained the first convolutional networks

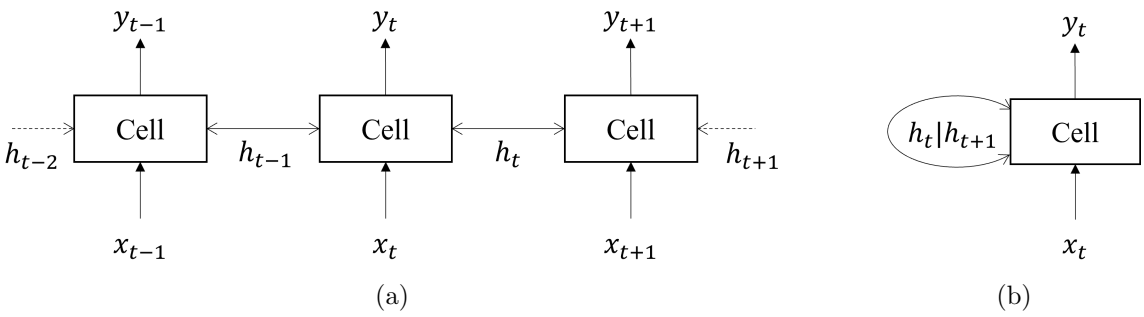


Figure 2.1: **The modern recurrent neural network.** Unrolled and feedback versions of sequence-to-sequence recurrent neural network layers are depicted in (a) and (b). Double-headed arrows indicate that RNN layers could be bidirectional.

with backpropagation to successfully recognize handwritten zip codes in 1989 [69, 67]. Development of CNNs continued in the 1990s, peaking in 1998 when *LeCun et al.* proposed *LeNet-5* [68], the foundation for modern convolutional networks. LeNet-5 was trained on the Modified-NIST (MNIST) dataset [66] which has since become the standard test dataset for performance of any DNN, also heavily used in this thesis. MNIST is a set of 60,000 training images and 10,000 test images of greyscaled, 28x28 pixel handwritten digits from 0 to 9 with the associated labels.

Recurrent neural networks (RNNs) are another popular type of deep neural network developed in the 1990s. RNNs are used primarily for modelling functions with sequential behaviour. The inputs fed to RNNs typically consist of temporal or sequential data, such as time series, speech, or audio data. The outputs of RNNs could be sequential or singular depending on the task. For example, speech recognition has a sequential input and output whereas sentiment analysis of textual reviews defines a stream of text as its input and a number as its output indicating the sentiment of the input text. To illustrate, figure 2.1a depicts an RNN with sequential inputs x_t and corresponding outputs y_t. The index t serves as the index for each element in the sequence, starting from $t = 0$ for the first element and ending at $t = \tau - 1$ for the τ-th element. The RNN cell is where learning occurs. During the forward step, the RNN cell computes a function of the current input x_t, current state h_t, and neural network parameters θ. The function outputs y_t and the state for the next timestep h_{t+1}.

RNNs have two major strengths. The first strength lies with the passing of state vectors between timesteps. This allows the RNN to form a sort of *memory* across long ranges. For example, x_t could strongly influence the final output of the RNN but may not be the final input of the RNN. In fact, it could be as far back as the beginning of the sequence ($t = 0$). To remember the contribution of x_t, the RNN updates the next state h_{t+1} with the output of the cell's forward function. It passes h_{t+1} to the next RNN cell in the sequence and the process repeats till the end of the sequence. While it is easier to understand RNNs

as cells chained together by their states (as in figure 2.1a), in reality, they are executed using a feedback loop. This allows RNNs to (1) operate in real-time without having to store every memory of the past and (2) process sequences without a known end. In figure 2.1b, we see that at any point in the sequence, there exists only the current input, current state, next state, and output; there is no knowledge of finality in the sequence nor past inputs. This behaviour is enabled by maintaining a state but also by sharing parameters across the sequence, the second major strength of RNNs. Like convolutional networks, RNNs utilize and update the same parameters every timestep. This is important for tasks such as real-time speech translation where after every input or fixed subset of inputs, the RNN-based translation model is expected to output the translation of what is being said. RNNs can operate as feedforward neural networks if they are unrolled (as in figure 2.1a). Unrolling requires knowing the full sequence ahead of time which has it's benefits such as faster runtime through parallel computation and drawbacks such as inability to perform in real-time. RNNs are a core focus of this thesis and are discussed in more detail in section 2.2.

One of the major reasons for the advancements of the machine learning field in the 1990s was the increase in availability of larger, more detailed datasets. Up until the 1950s, statisticians relied on small, manually typed datasets to model various processes. For example, anthropometric data was used to identify criminals from the early 1900s for decades until it was replaced with fingerprinting technology [37, 43]. From 1950 to 1990, the first researchers to study neural networks made use of small, synthetic datasets to prove the functionality of their methods [132, 43]. In the 1990s, several large datasets where compiled and used by researchers to develop new breakthroughs. Datasets such as MNIST [66] increased standard dataset sizes to the tens of thousands while datasets such as the PennTreebank dataset [82] contained millions of samples. The development of the internet in the early 1990s made it much easier in the coming decades for researchers to build larger datasets than ever before.

The increased availability of computers and highly parallel compute hardware was arguably the greatest reason for the advancement of the field of machine learning throughout the 1990s and into the next two decades. NVIDIA played a pivotal role with the creation of the modern GPU in 1999. Building upon this milestone, NVIDIA further expanded the capabilities of GPUs in 2006 with the introduction of the CUDA framework [79]. This framework enabled programmability of massively parallel computations in GPUs, revolutionizing many scientific fields including deep learning. The field of machine learning entered a second winter from the late 1990s to around 2006, possibly due to computational inefficiencies [43]. While datasets increased in size and deep neural networks became deeper and more complex, computers had just begun to increase in speed. Until 2006, it was difficult to train large models, end-to-end, by using backpropagation only. To solve this problem, *Hinton et al.* suggested to pretrain hidden layers in a neural network using a form of unsupervised learning and then fine-tune the whole model using supervised learning with backpropaga-

tion which showed remarkable results [53]. Researchers began to use unsupervised layer pretraining as a way to mitigate the impracticality of large model, end-to-end supervised training. For the next couple of years, many scientists experimented with applying GPU acceleration to deep learning [18, 106].

A breakthrough came in 2012 when *Krizhevsky et al.* utilized multiple GPUs to train a revolutionary deep convolutional neural network called *AlexNet* [64] on the ImageNet dataset, outperforming other networks by a 10% error margin in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [118]. AlexNet showed that by using GPUs, it was finally possible to train incredibly accurate models on the order of about 62M parameters, over datasets with millions of training examples, in an end-to-end configuration (raw pixels inputs to categorical outputs) using only backpropagation. Open-source machine learning frameworks such as Scikit-Learn, Tensorflow, and PyTorch [98, 1, 97] built on the success of AlexNet, enabling anybody with a computer and GPU to complete training or inference. After AlexNet, parameter sizes increased by orders of magnitude every couple of years [43]. New state-of-the-art models that utilized better compute hardware and new parallelism techniques were published for the ILSVRC challenge such as GoogLeNet [128] and VGGNet [126] which were deeper than AlexNet and solved other problems. For example, *Krizhevsky et al.* experimentally determined the size of convolutional filters and depth of filter maps for AlexNet [64] which prompted the question of whether this choice could be standardized. In response, *Simonyan et al.* introduced convolutional blocks in VGGNet which are standardized groups of convolutional and pooling layers that can be chained. They also suggested a specific way to augment convolutional filter sizes and depths of filter maps as a function of the depth of the network [126]. While VGGNet offered a type of standardization for CNNs, it dramatically increased the number of parameters by more than doubling the number of parameters in AlexNet. Alternatively, GoogLeNet introduced inception modules, another type of convolutional block where different filter sizes and pooling layers are applied in parallel to the same previous layer allowing the network itself to *choose* which filter size is necessary and when [126]. Due to their vast sizes and long training times, these networks have become backbone neural networks for several new architectures in recent years through the application of transfer learning [43]. In transfer learning, existing networks that have been pretrained on large datasets are reused and modified slightly to perform a task different to the one they were originally trained to perform. ResNet is another type of CNN that became state-of-the-art in the ILSVRC challenge in 2016 that is being used as a backbone network for several other tasks [48]. ResNet solved the problem of vanishing or exploding gradients as networks became deeper by introducing skip connections or residuals between CNN layers. Residuals are outputs from previous layers that are added to deeper layers with the intuition that information tends not to vanish or explode if it is available down the chain of layers.

While convolutional networks such as AlexNet, GoogLeNet, VGGNet, and ResNet

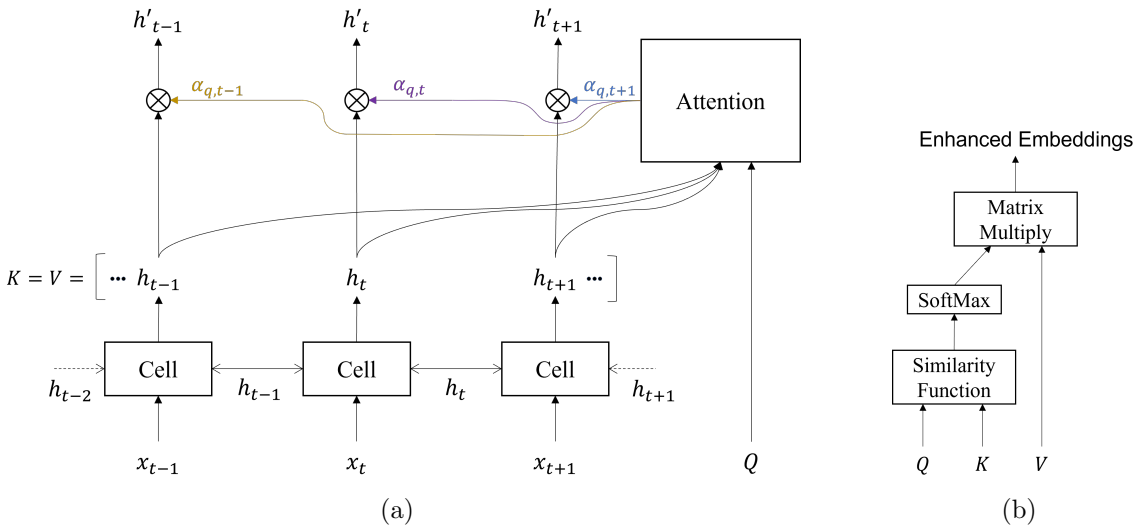


Figure 2.2: **The attention mechanism.** (a) Attention applied to the outputs of an RNN encoder, scaling the output states h_t with corresponding learned weights $\alpha_{q,t}$. (b) A generalized attention mechanism that learns and computes the similarity between queries and keys and scales the values accordingly.

showed remarkable results in image analysis, the **attention** model revolutionized natural language processing. In 2014, *Bahdanau et al.* introduced the attention mechanism. Combining it with recurrent neural networks, they achieved state-of-the-art results in machine translation [7]. When extracting distributed representations from RNNs, it is possible to use intermediate states (figure 2.5c) or the final state (figure 2.5a) as embeddings for classification layers or decoders. The problem with extracting the final state or even a relatively distant state from the beginning is the difficulty RNNs face with remembering long-term dependencies. While the LSTM [54] architecture was created to solve this problem, it still fails as sequences become longer [7]. Unfortunately, some parts of the sequence that may be more important than others for the end result become smoothed-out in the state across many timesteps. The function of attention is to enhance embeddings with a sense of context ensuring that states important to the end result remain emphasized.

Since the attention mechanism has access to many states and possibly previous output predictions, it learns to "attend" to the most important parts of the sequence. It does this by learning a set of weights for each value in the sequence. The magnitudes of the weights determine how important a value is in the sequence. Figure 2.2b illustrates how these weights are calculated. The query matrix $\boldsymbol{Q}$ acts as a representation of what the model is looking for to complete its task. For example, if we are translating a sentence from English to French word-by-word, we can use the previously predicted word from the network to *query* the network for the next most probable word (as in [7]). The attention mechanism accomplishes this by comparing the query matrix with the key matrix $\boldsymbol{K}$ to determine which value is most important to the query. The key matrix is a map to the value matrix $\boldsymbol{V}$ where

each vector in $\boldsymbol{K}$ represents a vector in $\boldsymbol{V}$. These vectors can be learned, which increases the representational power of the network, but are usually equivalent to their corresponding values for simplicity ($\boldsymbol{K} = \boldsymbol{V}$). To compare the query and key matrices, they are fed into a similarity function. This function varies for different types of attention mechanisms but is generally one of two variations: a simple feed-forward neural network which *learns* the similarity (used by *Bahdanau et al.* [7]), or a scaled matrix multiply which is a variation on cosine similarity (formally known as scaled dot-product attention [129], represented by equation 2.2 where the dot-product is between $\boldsymbol{Q}$ and $\boldsymbol{K}^T$ and the scale d_k is defined by the dimensionality of the query and key matrices). After the comparison, the results are fed into a `softmax` function (equation 2.1) which outputs the attention weights $\alpha_{q,t}$ for each query and key vector. The `softmax` function is used to ensure we are calculating relative importance and allowing smooth gradients to flow through the attention layer. The weights scale each value in $\boldsymbol{V}$, thus enhancing the original value matrix with contextual information.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.1}$$

$$\text{Attention}\left(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}\right) = \text{softmax}\left(\frac{1}{\sqrt{d_k}}\boldsymbol{Q}\boldsymbol{K}^T\right) \cdot \boldsymbol{V} \tag{2.2}$$

for $x_i \in \boldsymbol{x} \in \mathbb{R}^K$.

Attention can be applied to any sequence or group of values in neural networks. For example, figure 2.2a shows an RNN encoder parsing an input sequence $\{\boldsymbol{x}_t\}$, similar to figure 2.1a. The encoder outputs each of it states $\boldsymbol{h}_t$ which act as encoded representations of the input sequence at time t. In this example, the output states form both the key and value matrices. Assuming we have a query matrix $\boldsymbol{Q}$ for some task in the network, $\boldsymbol{Q}$ and $\boldsymbol{K}$ are fed into the attention mechanism which calculates the attention weights $\alpha_{q,t}$ for each value vector $\boldsymbol{h}_t$. The attention weights scale the output states and generate embeddings with contextual information $\boldsymbol{h}'_t$.

The attention model serves as the backbone of the **transformer** [129]. The transformer was introduced in 2017 as a potential replacement to recurrent neural networks. While attention enabled RNNs to preserve memory across longer distances, it did not change the sequential nature of RNNs which, depending on the scenario, is a major limitation of RNNs. *Vaswani et al.* [129] suggested that it is difficult to train RNNs on large amounts of data due to their sequential nature. Other networks such as CNNs and feed-forward networks can digest whole data inputs in parallel while RNNs digest inputs at every timestep. In the transformer model, whole sequences are processed in parallel. Each member of the sequence maintains information about their position in the sequence through the addition of a positional encoding, both for the inputs and outputs. The transformer replaces RNN encoders and decoders with stacked attention models.

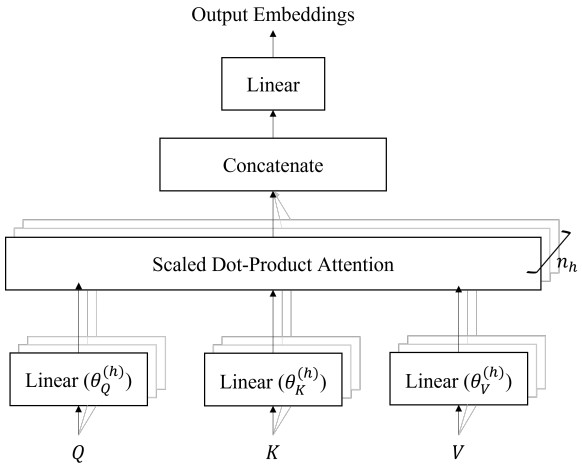


Figure 2.3: **Multi-headed scaled dot-product attention.**

While introducing scaled dot-product attention, *Vaswani et al.* [129] also introduced different variations such as **multi-headed attention**, **masked attention** (to maintain direction in auto-regressive tasks), and **self-attention**. Multi-headed attention instantiates several scaled dot-product attention modules and executes them in parallel with the intuition that multiple heads can extract better information than only one. The key, value, and query matrices are independently linearly transformed by n_h different sets of learned parameters before they enter the attention modules. The outputs from each head are concatenated and fed through a linear layer where they are linearly transformed by learned parameters once more to gain the final output embeddings. This structure is illustrated in figure 2.3. Since the transformer expects the whole output sequence at once, to maintain the auto-regressive nature of certain tasks such as machine translation where the output sequence is not known ahead of time, the output sequence is masked with $-\infty$ for timesteps that have not been output yet. This is called masked-attention. To build a general understanding of language, transformers can also be trained on unlabelled data (e.g. large corpora of sentences), enabled by self-attention. Self-attention sets the query matrix equal to the key matrix, therefore enabling the attention mechanism to extract importance relationships between each member of the sequence and each other member.

Transformers, and specifically self-attention, are central to the successful large language models from 2018 till today such as BERT, GPT-1, GPT-2, GPT-3, and most recently, GPT-4 [30, 104, 105, 16, 94]. While different with respect to training objectives, these models all pretrain stacks of transformers on vast amounts of unsupervised data to gain a general understanding of language. They have enabled applications such as advanced chatbots to revolutionize human-computer interaction. As a result, they have dramatically shortened the path towards general artificial intelligence.

While disadvantaged in parallelism, recurrent neural networks excel in real-time prediction due to their sequential execution. State-of-the-art speech recognition systems are

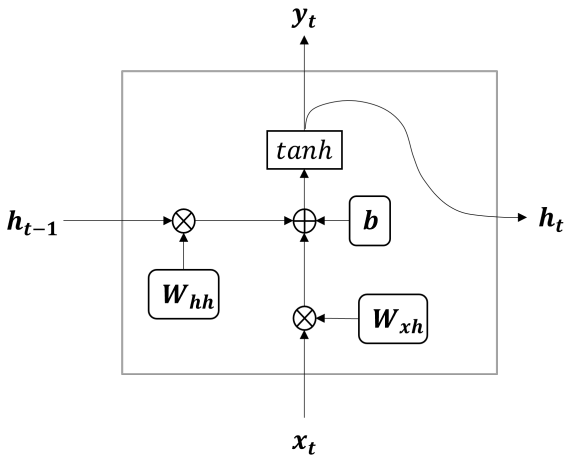


Figure 2.4: **The Vanilla RNN Cell.** The simplest form of RNN cell. In this figure, we use `tanh` as the activation function.

based on RNNs and special decoding procedures, enabling sub-second recognition of speech as it occurs [50]. The literature surrounding privacy-preserving RNNs is limited and thus applications such as privacy-preserving speaker and speech recognition remain fairly difficult to implement over encrypted data. In response, this thesis focuses on researching the applicability of fully homomorphic encryption techniques to RNNs with the vision to enable applications such as speaker and speech recognition over encrypted data.

2.2 Recurrent Neural Networks

The earliest form of recurrent neural networks was proposed by Hopfield in 1982 [55]. Hopfield modelled how the brain might remember and recall memories using a form of binary RNNs (where activations are Signum functions). These were subsequently called "Hopfield networks". Hopfield networks were considered a type of RNN only after 1986 when *Rumelhart et al.* [117] coined the term "recurrent neural network". Early Hopfield networks were trained using a form of the Hebbian rule [51] which is not as effective as backpropagation. To address this, the backpropagation-through-time algorithm was constructed by Werbos in 1990 [96] which helped form the modern RNN, both in structure and training process.

2.2.1 The *Vanilla* RNN

We introduced the general structure of recurrent neural networks in section 2.1.2 but did not discuss the composition of RNN cells. In general, the structure of different types of RNNs, which includes directionality and input-output relationships (i.e. sequence to one, sequence to sequence, or one to sequence), remains the same. The main variable characteristic between different types of RNNs lies within the mathematical structure of their cells. The most basic structure is the **vanilla RNN cell**, illustrated in figure 2.4.

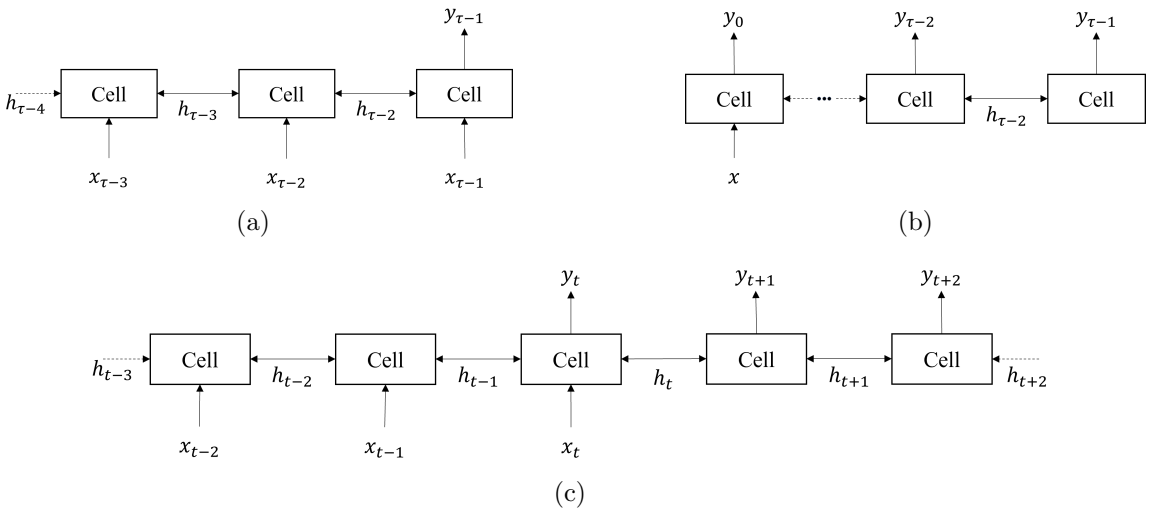


Figure 2.5: **Input-output relationships of RNNs.** (a) Many-to-one relationship where the last three states are shown. (b) One-to-many relationship showing the first and last two states. (c) Truncated many-to-many relationship where there are many inputs to many outputs but not necessarily in a one-to-one mapping. This type of relationship is commonly referred to as an encoder-decoder RNN style. All of these RNNs are displayed in an unrolled setting for better comprehension.

The vanilla cell has three learned parameters: $\boldsymbol{W_{xh}}$, $\boldsymbol{W_{hh}}$, and $\boldsymbol{b}$. The weight matrix $\boldsymbol{W_{xh}}$ linearly transforms the input vectors $\boldsymbol{x_t}$ for any $t \in [0, \tau)$. In figure 2.4, $\boldsymbol{x_t}$ is a vector but could also be a matrix if the inputs are batched. Similarly, $\boldsymbol{W_{hh}}$ denotes the matrix that linearly transforms the state vectors $\boldsymbol{h_t}$ (which could also be matrices when batching is enabled). The two linearly transformed vectors are summed with the third parameter $\boldsymbol{b}$ which denotes the bias vector for the layer. The non-linearity commonly applied to RNN neurons is the `tanh` function (equation A.2 in appendix A), as shown in figure 2.4, but can be replaced with any other non-linearity such as `sigmoid` (equation A.1) or `relu` [65]. The following set of equations represents the vanilla cell mathematically:

$$\boldsymbol{y_t} = \tanh(\boldsymbol{W_{xh}^T x_t} + \boldsymbol{W_{hh}^T h_{t-1}} + \boldsymbol{b}) \tag{2.3}$$

$$\boldsymbol{h_t} = \boldsymbol{y_t}, \tag{2.4}$$

where $\boldsymbol{x_t} \in \mathbb{R}^m$, $\boldsymbol{h_t}, \boldsymbol{h_{t-1}} \in \mathbb{R}^n$, $\boldsymbol{W_{xh}} \in \mathbb{R}^{m \times n}$, $\boldsymbol{W_{hh}} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b} \in \mathbb{R}^n$, m is the number of input features and n is the number of hidden units in the cell.

It is more intuitive to think about the cell as a composition of two different signals. One signal is the memory retained by the cell from previous timesteps while the other is the current input which introduces new information. As a result of the addition operation, both signals are weighted equally implying that they influence the cell's memory equally. This can be problematic with respect to backpropagation. Backpropagation-through-time,

the primary training algorithm of recurrent neural networks [96], can be understood as the application of the standard backpropagation algorithm on unrolled RNNs, effectively converting them into feed-forward neural networks. Each output of the RNN serves as a route through which loss gradients can flow towards the RNN parameters. As shown in figure 2.5, RNNs can have various input-output relationships. The many-to-one RNN (figure 2.5a) has one output and therefore one direction through which the loss function gradients can flow. The one-to-many and many-to-many RNNs (figures 2.5b and 2.5c) have several outputs and as a result, several gradient routes. The flow of gradients through vanilla RNNs introduces a couple of problems. Firstly, if the number of outputs and the length of sequences is too large, the requirement to store gradients across timesteps can lead to an unfavourable increase in memory usage. Secondly, gradients flowing through the state vectors can suffer from the vanishing or exploding gradient problem. Since the deepest connection in an RNN spans across the time dimension, the product of gradients can become very large or very small depending on whether the gradients are greater than zero or less than zero, leading to challenges in learning long-term dependencies. The product that causes this problem is the product of partial derivatives for each hidden state up to timestep t, as shown in equation 2.5.

$$\prod_{i=1}^{t} \frac{\partial \boldsymbol{h_i}}{\partial \boldsymbol{h_{i-1}}} = \frac{\partial \boldsymbol{h_t}}{\partial \boldsymbol{h_{t-1}}} \frac{\partial \boldsymbol{h_{t-1}}}{\partial \boldsymbol{h_{t-2}}} \frac{\partial \boldsymbol{h_{t-2}}}{\partial \boldsymbol{h_{t-3}}} \cdots \frac{\partial \boldsymbol{h_2}}{\partial \boldsymbol{h_1}} \frac{\partial \boldsymbol{h_1}}{\partial \boldsymbol{h_0}} \tag{2.5}$$

There are several techniques to address these problems. One technique involves clipping each gradient that is too large to a pre-defined value to limit the possibility of exploding gradients, a technique referred to as **gradient clipping**. Another technique is to use **truncated-backpropagation-through-time**, a form of the backpropagation-through-time algorithm where backpropagation is applied to only a fixed number of timesteps. This shortens the product in equation 2.5, reducing the total memory usage and the probability of it diverging to a large number or converging to a very small number. It is also possible to completely alter the cell to handle the vanishing and exploding gradient problem. Rather than clipping gradients, by introducing more parameters and **gating functions** such as the `sigmoid` function, we can teach the network to *select* the information it wants to consider in its output decision and manage potential gradient problems.

The **long short-term memory (LSTM)** cell was proposed by Hochreiter and Schmidhuber [54] in 1997 as an alternative cell structure to vanilla RNN cells to mitigate the exploding and vanishing gradients problem. They introduced a new subset of RNN cells that incorporate gates to control information flow within the cell, commonly referred to as gated RNN cells. Specifically, they introduced forget, input, and output gates. They also added a state vector to the cell, separate from the output and immediate state vectors, that is controlled by the gates and acts as a form of long-term memory. With the cell being

able to determine how much of the long-term state and how much of the immediate state influences the output at a time t as opposed to simply adding the vectors, information flow tends to remain stable both in the forward and backward directions. For more information regarding the LSTM cell, please refer to appendix A.

The LSTM cell is a computationally heavy cell. Within one timestep, a four-layer deep neural network is traversed, increasing the amount of computation when compared to regular RNN cells for the benefit of better performance. Other successful implementations of gated RNN cells have been proposed such as the **gated recurrent unit (GRU)** cell [26]. These types of cells are similar to LSTM cells and claim to achieve comparable performance with a fraction of the number of parameters and a simpler computational graph.

2.2.2 Applications

LSTMs have shown remarkable results in a variety of applications. They have excelled in modelling difficult tasks such as speech and handwriting recognition where dataset samples are not segmented into proper input-label sequences. For example, it is impractical to compile datasets with phoneme labels for each set of input audio frames. These types of tasks are referred to as *temporal classification* tasks, a term introduced by *Graves et al.* in 2006 [46]. By using a bi-directional LSTM to develop accurate encodings of raw data and a special decoding algorithm, the connectionist temporal classification (CTC) model [46] achieved state-of-the-art results in speech recognition over the TIMIT speech corpus. However, the most interesting application in [46] was online handwriting recognition. Using the continuous location and state of a pen, the CTC model can output the letters as they are written.

Graves later improved on the CTC model and introduced the recurrent neural network transducer (RNN-T) [45]. The main difference between CTC and RNN-T models is the addition of a prediction network. The CTC model only contains a bi-directional LSTM encoder network that outputs conditional probabilities with respect to the input sequence. The RNN-T model contains the same encoder network as well as a bi-directional LSTM prediction network that uses the output sequence generated by the encoder network as its input sequence. The addition of a prediction network is necessary to condition the output probabilities on the output as well as the input sequence of the network. The CTC network assumes that the output probabilities are all independent of each other which is not true in many real-world tasks such as machine translation. Graves showed that the RNN-T outperforms CTC in speech recognition. The experiments in [45] utilize a two-layer, stacked LSTM for the encoder network which suggests that stacked LSTMs (or deep LSTMs) outperform single-layer LSTMs for speech recognition. This was further supported by *Sak et al.* [119] who utilized the RNN-T model with several stacked LSTMs to model the acoustic frame labelling task. Current state-of-the-art speech recognition models such as in *He et al.* [50] continue to use the RNN-T as a base model with several stacked LSTMs

for both the encoding and prediction sub-models.

LSTMs have also successfully been integrated with other neural network layers. The LSTM cell can be combined with a convolutional network by replacing the matrix multiplication operations in the LSTM cell with convolution operations. *Shi et al.* [125] proposed the *ConvLSTM* architecture to handle sequences with spatial and temporal properties. As mentioned in section 2.1.2, attention was introduced through integration with general RNNs [7]. In this scenario, RNNs (and LSTMs) are used as great encoders of sequences that can learn the inherent causal relationships of a sequence. As a result, distributed representations (encodings) are produced that are fed to higher layers in the deep neural network which can aggregate or expand the RNN outputs to perform the desired task.

Since sequences have the potential to be very long, recurrent neural networks are some of the deepest types of neural networks. For example, state-of-the-art speech recognition papers such as [50] preprocess speech samples by segmenting them into 25ms frames *every 10ms*, yielding a 15ms overlap between frames. To illustrate, given a three second speech sample, they generate a sequence of about 300 audio frames (3000ms/10ms) that serves as the input sequence to their RNN model. Using a simple, one-layer RNN, this equates to a 300-layer deep neural network which is problematic for privacy-preserving recurrent neural networks operating end-to-end. The depth of a neural network is generally equal to the depth of its private circuit under leveled homomorphic encryption, assuming activation functions consume only one level. If we convert the RNN in [50] to a privacy-preserving RNN and execute it over speech samples that are three seconds long, we would need to select a leveled homomorphic encryption scheme that can handle depths of at least 300 levels. Since such a scheme does not exist, we need to select a scheme that includes an efficient bootstrapping method to refresh the ciphertexts after each activation function execution. A fully homomorphic encryption scheme officially referred to as the CGGI scheme and commonly known as the TFHE scheme [25], presents an efficient bootstrapping method. The next section introduces fully homomorphic encryption and is followed, in section 2.4, by a thorough explanation of the main scheme used in this thesis, CGGI.

2.3 Fully Homomorphic Encryption

Homomorphic encryption (HE), considered the "holy grail" of encryption, closes the data security attack surface during computation. Algorithms such as AES [89] can secure data at rest and transport protocols such as TLS [111] can secure data during transport. HE ensures that data remains secure during compute, which closes the remaining privacy vulnerability. Moreover, modern HE schemes base their security on a class of problems considered to be quantum resistant [109], meaning that upon the widespread use of quantum computing in the future, these algorithms are guaranteed to be safe against attacks according to the currently known capabilities of quantum algorithms.

2.3.1 Overview

Fully homomorphic encryption (FHE) is a type of encryption that enables an unbounded number of mathematical operations such as addition and multiplication over ciphertexts while preserving correctness in the underlying plaintexts. For example, for two plaintexts m_0 and m_1,

$$\text{Enc}(m_0) \oplus \text{Enc}(m_1) = \text{Enc}(m_0 + m_1) \tag{2.6}$$

$$\text{Enc}(m_0) \otimes \text{Enc}(m_1) = \text{Enc}(m_0 \times m_1), \tag{2.7}$$

where Enc represents a general FHE encryption function and $\bigoplus$ and $\bigotimes$ denote the FHE addition and multiplication operations over ciphertexts. FHE schemes also allow plaintext-ciphertext operations where ciphertexts can be multiplied or summed with plaintexts as shown in the following equations:

$$m_0 \oplus \text{Enc}(m_1) = \text{Enc}(m_0 + m_1) \tag{2.8}$$

$$m_0 \otimes \text{Enc}(m_1) = \text{Enc}(m_0 \times m_1) \tag{2.9}$$

To ensure security, ciphertexts under FHE incorporate a corresponding **noise** value that is introduced to the original plaintext during the encryption process. The noise value is sampled from a probability distribution defined by the FHE scheme χ_{err} such that it is large enough to guarantee security and small enough to guarantee accurate decryption. After decryption, the noise remains a part of the message and can be rounded away to return the correct message. Naturally, if the noise is large enough to mask the message, rounding procedures cannot return the correct message and the message becomes indistinguishable from the noise. Noise growth is directly proportional to the number and type of FHE operations undergone by the ciphertexts. FHE operations such as addition of ciphertexts and plaintext-ciphertext operations contribute less to the error while multiplication of ciphertexts contributes the most. However, chaining a large number of low-cost operations can also exceed the noise bound for accurate decryption. It is thus intuitive to envision a noise budget associated with each ciphertext—with every operation, the remaining budget decreases. If the budget is exceeded, then the scheme cannot guarantee correctness upon decryption.

Since it is impossible to measure the exact error in a ciphertext prior to decryption, an estimation is employed, enabling the design of computational circuits with mathematical assurances of correctness [10]. Thus, it is useful to define an FHE circuit as a directed, acyclic graph of FHE operations [10] with the number of consecutive operations considered the **depth** of the circuit. Specifically, since multiplication of ciphertexts is considered the most expensive operation, the number of consecutive multiplications is generally considered the depth of the FHE circuit. Depth is an important parameter considered in HE

circuit design not only for accuracy but also for computational complexity. Deeper circuits require more computationally expensive encryption parameters, entailing larger computational complexity and necessitating a trade-off between depth and efficiency. Another term used in conjunction with depth is the concept of **levels** of a ciphertext. For example, if a depth of ten is only possible for a given set of FHE parameters, then each "fresh" ciphertext, which denotes a ciphertext that has not undergone any operations, has ten levels before its noise budget is depleted.

In order to evaluate unbounded circuits with theoretically infinite depth, ciphertext noise must be decreased and thus, the noise budget increased. Therefore, the most important operation enabling FHE schemes is the bootstrapping method. Introduced by Gentry in 2009 [38], **bootstrapping** is an algorithm that performs the decryption function in the encrypted domain, effectively *re-encrypting* the ciphertext and reducing the noise in the process. While the implementation differs between schemes, the general idea of re-encrypting a ciphertext to "refresh" the underlying plaintext remains the same. By applying bootstrapping after a fixed amount operations, the ciphertext can continue to undergo further operations and can thus repeat this cycle a theoretically unlimited number of times. Gentry's proposed bootstrapping method transformed homomorphic encryption into fully homomorphic encryption, crediting him as the father of FHE.

2.3.2 Developmental Years

Prior to 2009, several homomorphic encryption schemes existed [83]. The idea of homomorphic computation on encrypted data was hypothesized by *Rivest et al.* [112] in 1978 where the authors referred to HE operations as "privacy homomorphisms". After its introduction, HE became a feature of many public cryptography schemes in a partial setting. **Partial homomorphic encryption (PHE)** schemes can perform only one operation, either addition or multiplication of ciphertexts. For example, the widely used RSA encryption scheme [113] is partially homomorphic over multiplication, exhibiting the behaviour described in equation 2.7. In contrast, Galbraith's encryption scheme over elliptic curves [36] is partially homomorphic over addition (as in equation 2.6). According to [2], one of the most significant steps towards FHE was the BGN [12] scheme in 2005. The BGN [12] scheme enabled an arbitrary amount of additions of ciphertexts and at most, one multiplication over ciphertexts. This classified the BGN scheme as part of a new class of HE schemes called **somewhat homomorphic encryption (SHE)** schemes. SHE schemes enable both additive and multiplicative homomorphisms over encrypted data. However, they are not considered FHE schemes because they are limited in the depth of their circuits. Moreover, depth is not considered an input parameter for SHE schemes; instead, it is a consequence of the security parameters [81]. *Compactness* is a term that refers to a scheme's independence relationship between depth and the bound on ciphertext bit growth—as the depth increases, if there is no bound on ciphertext growth, then the scheme is not *compact*, whereas if the size is

bounded, then the scheme is *compact* [5]. *Compactness* is not guaranteed for SHE schemes, but is a required property of **levelled homomorphic encryption (LHE)** schemes, consequently enabling the incorporation of depth as a setup parameter [5]. Similar to SHE schemes, LHE schemes allow both addition and multiplication of ciphertexts for circuits of limited depth. A popular example of an LHE scheme is BGV [15] which does not present a bootstrapping method, yet is still widely used in the community. One of the motivational factors of using an LHE scheme even after the invention of FHE schemes in 2009 is reduced computational complexity. The bootstrapping operation, excluding FHEW-based schemes [32], is generally the most computationally expensive operation and is impractical to perform in most cases (except CGGI [25]). For instance, the bootstrapping method introduced by Gentry in [38] took a half hour to refresh a single bit of the ciphertext [39].

It is possible to begin by defining a somewhat (or levelled) homomorphic encryption scheme and then introduce a bootstrapping procedure to achieve a fully homomorphic encryption scheme [81]. This was first presented in Gentry's breakthrough scheme in 2009 [38]. Many popular FHE schemes introduced after [38] such as BFV [14] and CKKS [19] similarly first develop an S(L)HE scheme and extend it to an FHE scheme. *Marcolla et al.* [81] give an excellent summary of the conditions for a scheme to be considered fully homomorphic. Specifically, a scheme must satisfy two essential criteria: (1) the ability to evaluate any (efficient) circuit of arbitrary depth, and (2) the property of *compactness*. During its fourteen year history, fully homomorphic encryption has undergone three previous generations of evolution, with the current generation being the fourth [81]. The third generation included schemes such as CGGI [22], the main scheme used in this thesis. As with most of the third generation schemes, CGGI achieved significant improvements in computational efficiency, specifically in bootstrapping. However, it was limited to operations over ciphertexts that encrypt a single bit. Using multiple ciphertexts, circuits could be developed to evaluate any function over binary inputs. CGGI introduced the term "gate bootstrapping" where the evaluation of every binary gate (i.e. AND, XOR, etc.) also results in a bootstrap of the ciphertext, allowing an unbounded depth to the binary circuit being evaluated. Performing a bootstrap for every bit operation becomes inefficient, despite the unparalleled speed of the bootstrap operation in CGGI. Furthermore, modelling integers over several bits/ciphertexts and stacking integers to perform vector operations greatly increases the cost in memory space and efficiency. As a result, CGGI continues to evolve in the fourth generation with the introduction of batched operations and the **programmable bootstrap (PBS)** [25, 20, 23, 10]. This type of bootstrapping operation can evaluate a lookup table (LUT) over integer inputs and refresh the ciphertext simultaneously. It can be seen as a form of gate bootstrapping for integers, thus decreasing the memory footprint of otherwise multiple single-bit ciphertexts and the number of bootstraps necessary for a circuit. The fourth generation also introduces CKKS [19], an FHE scheme encoding fixed-point precision values. Although the ability to perform operations over fixed-point values is helpful in certain con-

texts, such as machine learning where values are generally encoded in floating-point, CKKS is typically operated in a levelled setting due to its inefficient bootstrapping operation. In fact, the bootstrapping operation in CKKS involves a greater number of operations and hence is computationally much slower than the PBS operation in CGGI. It also lacks the simultaneous LUT evaluation capability found in CGGI's PBS operation. Thus, the types of computational graphs that can be evaluated practically by CKKS may limit the types and performances of various machine learning models. Additionally, CKKS cannot evaluate non-linear functions using the operations at its disposal whereas schemes such as CGGI can evaluate *any* function that can be encoded in a LUT through the PBS operation.

In the following section, we discuss the main fully homomorphic encryption scheme used in this thesis, CGGI. By detailing the construction and various operations provided by CGGI, we can better understand the construction of two privacy-preserving recurrent neural networks proposed later in this thesis. For a discussion surrounding the security of FHE in general, refer to appendix B.1.

2.4 Fully Homomorphic Encryption Over The Torus

The CGGI scheme, or more commonly known as the TFHE scheme, was first proposed by *Chilloti et al.* in 2016 [22]. There have been several iterations of the core scheme which mainly consist of optimizations, however its construction remains the same.

2.4.1 Learning With Errors

The security of CGGI comes from the hardness of the **(Ring) Learning With Errors** (R)LWE problem [109, 80]. The objective of the LWE problem is to recover a secret $\boldsymbol{s} \in \mathbb{Z}_q^n$ where for some size $n \geq 1$ and modulus $q \geq 2$, an adversary is given n linear equations of the secret, each with randomly added noise $e \in \mathbb{Z}_q$. Without e, the secret can be recovered using Gaussian elimination in polynomial time. With noise, the problem becomes significantly harder and can be reduced to being as hard as worst-case, hard lattice problems defined in lattice cryptography [109]. The following is a formalization of the LWE problem. Let χ be a probability distribution over $\mathbb{Z}_q$. Let $\mathcal{A}_{\boldsymbol{s},n,q,\chi}$ be an LWE distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ such that for a secret $\boldsymbol{s} \in \mathbb{Z}_q^n$, a sample $(\boldsymbol{a}, b)$ is defined by the following: $\boldsymbol{a} \sim U(\mathbb{Z}_q^n)$, $e \sim \chi(\mathbb{Z}_q)$, and $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e$. The adversary must thus derive $\boldsymbol{s}$ from $(\boldsymbol{a}, b)$, or correctly distinguish $(\boldsymbol{a}, b)$ from $(\boldsymbol{a}_r, b_r) \sim U(\mathbb{Z}_q^n \times \mathbb{Z}_q)$ to solve the *search*-LWE or *decisional*-LWE problems respectively. RLWE, or *Ring*-LWE, solves the LWE problem over the ring $\mathcal{R}_q$ and can be seen as the polynomial version of LWE where operations occur within $\mathcal{R}_q$ [110]. Specifically, let $\chi(x)$ be a probability distribution over $\mathcal{R}_q$, and let $N \geq 1$ be the order of the polynomials in $\mathcal{R}_q$, then the RLWE setting defines a distribution $\mathcal{A}_{s(x),N,q,\chi(x)}(x)$ from which we can sample $(a(x), b(x) = a(x) \cdot s(x) + e(x))$, where $a(x), s(x), e(x) \in \mathcal{R}_q$ are polynomials of order-N. As a result, RLWE enables packing and fast multiplicative operations using the

Fast Fourier Transform, all desired properties of a good encryption scheme.

2.4.2 Construction

CGGI is based on both LWE and RLWE with an extension to the torus. The real torus $\mathbb{T}$ is defined as $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ and is essentially the set of real numbers between $[0, 1)$ modulo 1. The torus set forms an Abelian group closed under addition $(\mathbb{T}, +)$, but it is not a ring due to the absence of closed multiplication. Furthermore, an external product exists for any scalar $s \in \mathbb{Z}$ where, for $y \in \mathbb{T}$, $s \cdot y \in \mathbb{T}$. The torus, as a group, can be extended to a ring by forming the set of polynomials with coefficients over the torus, and taking the quotient of this ring by the polynomial $x^N + 1$ for any N a power-of-two: $T = \mathbb{T}(x)/(x^N + 1)$. The N parameter is called the "ring dimension" of the scheme and is derived from the RLWE setting. In CGGI, we define $\phi(x) = x^N + 1$ as the M-th cyclotomic polynomial and N as its degree. For practical reasons, CGGI is built from a discretized version of the torus $\mathbb{T}_q$, for modulus $q = 2^\Omega$, where $\mathbb{T}_q = \mathbb{Z}/q\mathbb{Z} = \{\frac{i}{q} \mod 1 \mid i \in \{0, 1, ..., q-1\}\}$ [61]. As an illustration, we can imagine $\mathbb{T}_q$ as a circle divided into q equal portions where each portion denotes an element $y_i \in \mathbb{T}_q$. Similar to $\mathbb{Z}_q$, $\mathbb{T}_q$ can be extended to a ring over the polynomials divided by $\phi(x)$, $\mathcal{T}_q = \mathbb{T}_q(X)/(x^N + 1)$. Therefore, the elements of the ring exhibit a **negacyclic** property. Since $x^N = -1$, there are effectively $M = 2N$ elements in $\mathcal{T}_q$ where the second half are the negation of the first half. This becomes important when discussing programmable bootstrapping and the evaluation of lookup tables in section 2.4.4. Addition and multiplication can be defined on $\mathcal{T}_q$. Moreover, LWE and RLWE are problems well defined and difficult to solve over the discretized torus. Given all of these properties, we can create a fully homomorphic encryption scheme over the discretized torus. The following sections overview the main operations in CGGI: key generation, encryption, decryption, and evaluation operations. For more detailed information, refer to appendix C.

Key Generation

Given a security parameter λ (defined in appendix B.1), the key generation process outputs public information $(k, n, N, \chi_{\text{LWE}}, \chi_{\text{RLWE}}, p, q)$, and private information $(\boldsymbol{s}, s(x))$. χ_{LWE} and χ_{RLWE} are the LWE and RLWE error distributions respectively. The plaintext ring is defined by $\mathcal{P}_q = \frac{q}{p}\mathcal{T}_q$ and the plaintext group is defined by $\frac{q}{p}\mathbb{Z}_q$, where $q = 2^\Omega$. In this thesis, unless specified otherwise, we set $\Omega = 64$. The parameter p represents the precision of the plaintext message space and is defined as $p = 2^\omega$. We utilize $\lambda = 128$ and $k = 1$ in this thesis only. The private information includes the LWE and RLWE secret keys.

Encryption

We sample $\boldsymbol{a} \sim U(\mathbb{Z}_q)$, $e \sim \chi_{\text{LWE}}$, $a(x) \sim U(\mathcal{T}_q)$, and $e(x) \sim \chi_{\text{RLWE}}$. Given two plaintexts, $\mu \in \frac{q}{p}\mathbb{Z}_q$ and $\mu(x) \in \mathcal{P}_q$, we produce the following ciphertexts:

$$\text{LWE}_{n,q,\chi_{\text{LWE}}}(\mu, \boldsymbol{s}) = (\boldsymbol{a}, b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + \mu + e_q) \pmod{q} \tag{2.10}$$

$$\text{RLWE}_{k,N,q,\chi_{\text{RLWE}}}(\mu(x), s(x)) = (a(x), b(x) = a(x) \cdot s(x) + \mu(x) + e_q(x)) \pmod{q \cdot \phi(x)} \tag{2.11}$$

where $e_q = \lceil q \cdot e \rfloor$, $e_q(x) = \lceil q \cdot e(x) \rfloor$. The encoding function that maps messages to plaintexts is defined in appendix C.2.

Decryption

Given an LWE secret key $\boldsymbol{s}$ and ciphertext $(\boldsymbol{a}, b)$, or RLWE secret key $s(x)$ and ciphertext $(a(x), b(x))$, the following equations define the decryption operation in CGGI:

$$\mu + e_q = b - \langle \boldsymbol{a}, \boldsymbol{s} \rangle \pmod{q} \tag{2.12}$$

$$\mu(x) + e_q(x) = b(x) - a(x) \cdot s(x) \pmod{q \cdot \phi(x)} \tag{2.13}$$

This operation is also referred to as *unmasking* the ciphertext. The decryption operation leaves a noisy plaintext that needs to be decoded to retrieve the message. The decoding operation is defined in appendix C.3.

2.4.3 Levelled Operations

With a general understanding of the construction of CGGI, we can begin to define basic arithmetic operations in the encrypted domain. The operations that are defined in this section are dependencies of the programmable bootstrap operation and introduced in a way such that each previous operation is necessary for the next operation. We define certain operations rigorously while others abstractly. To be defined rigorously, the abstractly defined operations require definitions of certain sub-operations that are not important to this thesis and are therefore paraphrased. Wherever a definition is abstract, we refer the reader to its exact definition within the paper citation from which it originates.

LWE Addition and Scalar Multiplication

By the LWE and RLWE construction, we are already given closure under addition and scalar multiplication. Let the ciphertexts $\text{ct}_0 = (\boldsymbol{a}_0, b_0)$ and $\text{ct}_1 = (\boldsymbol{a}_1, b_1)$ be encryptions of plaintexts μ_0 and μ_1 under the same secret key $\boldsymbol{s}$. Given a scalar $u \in \mathbb{Z}$, the following

equations detail both operations:

$$\mathrm{ct}_0 \oplus \mathrm{ct}_1 = \left(\boldsymbol{a}_0 + \boldsymbol{a}_1, b_0 + b_1 = \langle(\boldsymbol{a}_0 + \boldsymbol{a}_1), \boldsymbol{s}\rangle + (\mu_0 + \mu_1) + e'_q\right) \tag{2.14}$$

$$u \otimes \mathrm{ct}_0 = \left(u \cdot \boldsymbol{a}_0, u \cdot b_0 = \langle u \cdot \boldsymbol{a}_0, \boldsymbol{s}\rangle + (u \cdot \mu_0) + e''_q\right) \tag{2.15}$$

where $e'_q = e_{q,0} + e_{q,1}$ and $e''_q = u \cdot e_{q,0}$. The equations above show that addition between ciphertexts and scalar multiplication by an integer are well-defined operations in CGGI that produce the correct output ciphertext. However, we also see that the error in both operations increases and must be monitored.

Key Switching

Sometimes it is necessary to change the underlying secret key of a ciphertext. Consider two secret keys $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$. The objective is to switch the secret key that encrypts ciphertext $\mathrm{ct} = \mathrm{LWE}_{n,q,\chi_{\mathrm{LWE}}}(\mu, \boldsymbol{s}_0) = (\boldsymbol{a}, b)$ from $\boldsymbol{s}_0$ to $\boldsymbol{s}_1$. We generate a keyswitch key $\mathrm{ksk}_{\boldsymbol{s}_1}(\boldsymbol{s}_0)$, which is a special encryption of the original key under the new key (for details about the type of encryption, we refer the reader to [25]). Using ksk, we perform the key switch operation, as further detailed in appendix D.1.

Multiplication - External Product

The external product operation occurs between two ciphertexts. It is well-defined where one ciphertext is an LWE ciphertext while the other is a GGSW ciphertext. The CGGI scheme introduces a type of ciphertext it calls a *General*-GSW ciphertext which stems from the GSW FHE scheme [40]. Given two plaintexts $\mu_0, \mu_1 \in \mathbb{Z}_q$, we obtain two ciphertexts $\mathrm{ct}_0 = \mathrm{LWE}_{n,q,\chi_{\mathrm{LWE}}}(\mu_0, \boldsymbol{s})$ and $\mathrm{ct}_1 = \mathrm{GGSW}(\mu_1, \boldsymbol{s})$, encrypted under the same secret key $\boldsymbol{s}$. The following equation represents the external product:

$$\mathrm{ct}_0 \odot \mathrm{ct}_1 = \mathrm{LWE}_{n,q,\chi_{\mathrm{ExtP}}}(\mu_0 \cdot \mu_1, \boldsymbol{s}) \tag{2.16}$$

where χ_{ExtP} represents a new, larger error distribution as a result of the external product operation.

Ciphertext Multiplexer

The external product enables an operation called the Ciphertext Multiplexer (CMux), unique to CGGI (where it is referred to as the *controlled* multiplexer [25]). At a very abstract level, the CMux operation selects between two ciphertexts using an encrypted selector bit and outputs the desired ciphertext. CMux is further detailed in appendix D.2.

Blind Rotation

The core enabler of programmable bootstrapping is blind rotation. To understand what rotation performs in general, we begin by visualizing the encrypted part of an RLWE ciphertext in its polynomial form:

$$b(x) = b_0 + b_1 x + b_2 x^2 + ... + b_t x^t + ... + b_{N-1} x^{N-1} \tag{2.17}$$

Assume we would like to rotate $b(x)$ by t coefficients to the left so that b_t becomes the first coefficient in the new, rotated polynomial $b^{(rot)}(x)$. We perform the following operation:

$$b^{(rot)}(x) = x^{-t} b(x) = b_t + b_{t+1} x + b_{t+2} x^2 + ... + b_{N-1} x^{N-t-1} - b_0 x^{N-t} - ... - b_{t-1} x^{N-1} \tag{2.18}$$

After a rotation of $t < N$, we notice that the first set of coefficients $\{b_0, b_1, ..., b_{t-1}\}$ rotate towards the left to their correct spots within a polynomial of order-N by wrapping around the end of the polynomial. However, they transform into negative values, a consequence of the negacyclic nature of $\mathcal{T}_q$. As discussed in section 2.4.2, polynomials in $\mathcal{T}_q$ are of order $M = 2N$ and therefore, monomials in $\{x^N, x^{N-1}, ..., x^{2N-1}\}$ (mod $\phi(x)$) become negative. Rotating by a positive value t rotates the polynomial t coefficients to the right, with the coefficients exhibiting the same negacyclic property.

In the cases mentioned above, the t value is public. In certain schemes, such as CKKS [19], the t value is always public. In CGGI, it is possible to rotate based on an encryption of t, and thus, **blind rotation** is a rotation of an RLWE ciphertext by an *encrypted* number of coefficients. For a more detailed explanation of this process, we refer the reader to appendix D.3. Intuitively, blind rotation is akin to unlocking a rotary combination lock. Depending on a secret t_{enc}, we rotate a certain amount of coefficients to the left, then to the right, and we continue until we reach the final encrypted bit where we rotate once more until the ciphertext is in the correct position. An observation follows from this type of operation: if we encode a table of values $f(x)$ in the coefficients of an RLWE ciphertext, or even a trivial RLWE ciphertext, then we can decompose x into its binary form and evaluate $f(x)$ using blind rotation. The resulting ciphertext will then have an encrypted $f(x)$ in the first coefficient of the RLWE polynomial. Using sample extraction, we can transform the coefficient to an LWE ciphertext. This is the essence of the programmable bootstrap which we discuss in section 2.4.4.

2.4.4 Programmable Bootstrapping

With the introduction of blind rotation in section 2.4.3, we can define the programmable bootstrap, the flagship operation in CGGI. In section 2.3.1, we mentioned that the general approach for bootstrapping a ciphertext is to perform the decryption circuit in the en-

crypted domain. In section 2.4.3, we showed how to evaluate an arbitrary function through blind rotation. In this section, we show how blind rotation is leveraged to perform an evaluation of an arbitrary function *and* bootstrap concurrently. We begin by discussing the general bootstrapping procedure. A subscript $2N$ denotes the modulus of the variable, and $\text{SampleExtract}(\text{ct}_{\text{RLWE}}, i)$ denotes the sample extraction operation for a coefficient indexed by i in an arbitrary RLWE ciphertext ct_{RLWE}. Consider the identity function $f_I(x) = x$ for $x, f_I(x) \in \mathbb{Z}$. By encoding the identity function into a trivial RLWE ciphertext V_I, a rotation by $\mu_{2N} + e_{2N}$ to the left places $\mu_{2N} + e_{2N}$ in the first index. A subsequent sample extraction produces a fresh LWE encryption of μ_p. The bootstrap process for an LWE ciphertext $\text{ct} = (\boldsymbol{a}, b)$ is defined by the following function:

$$\text{ct}_{\text{fresh}} = \text{SampleExtract}(x^{-\left(b_{2N} - \sum_{i=0}^{n} a_{i,2N} \cdot s_i\right)} \cdot V_I,\ 0) \tag{2.19}$$

This definition is abstract since there are several enabling operations that are not discussed here, such as modulus switching and external products with a bootstrapping key. We briefly discuss these operations in appendix D.4. For even more information, we refer the reader to [25].

As mentioned in section 2.4.2, the decryption process consists of decryption followed by decoding. When the rotation completes, the output ciphertext contains the result of $f_I(\mu_{2N} + e_{2N})$. It is possible for the error e_{2N} to add more rotation steps than necessary, moving the correct coefficient after rotation to an incorrect spot and outputting an incorrect value. For this reason, the RLWE polynomial must include *repetitions* of each element. If the first coefficient lands in a pack of repetitions of the correct value after rotation, then as long as $|e_{2N}| < N/p$, the error will not push it into an incorrect region. This is the final decoding step of the decryption process, which bootstrapping performs in the encrypted domain. Selecting parameters in the key generation process is thus very important since the noise distribution affects the final noise growth, which could impact the rounding procedure in the bootstrap process.

Effectively, bootstrapping enables the evaluation of any discretized, arbitrary function. We can define a function $f(x) \in \mathbb{Z}$ over domain $x \in [-N, N-1] \subset \mathbb{Z}$ which will be rotated according to $\mu_{2N} + e_{2N}$. However, due to the negacyclic property of RLWE ciphertexts, any functions that are not negacyclic (where $f(x + N) \neq -f(x)$) require a bit of padding, reducing the domain by half. Functions that are negacyclic such as the Signum function, do not require any padding and thus, the whole domain is available. For example, the identity function for $p = 16$ requires a lookup table with the following packs of repeated values in order: $[0, 1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1]$. However, for Signum, the value 1 is the only value required to be assigned to the coefficients across the RLWE polynomial. When querying the identity lookup table, the maximum value we can query is $f(7)$ whereas for the Signum function, we can correctly query $f(15)$. When we design lookup tables, we must understand the negacyclic property and select the correct parameters to define how

many repetitions there are in each pack of repeated values. This is very important for the evaluation of neural networks since different neural network weight matrices transform the domains of input distributions into different domains that could possibly lead to overflow in the ciphertext, producing incorrect values. The programmable bootstrap earns its name from this unique evaluation of lookup tables and a concurrent bootstrapping operation.

2.4.5 The Concrete Library

There are several libraries that implement the CGGI scheme. The original library was TFHE [24], written in C++. Several other libraries have been built as improvements to TFHE such as cuFHE [17] and NuFHE [93], which both introduced hardware acceleration to CGGI for the first time. These libraries implement older versions of CGGI and do not incorporate many of the new advancements after 2019. They also evaluate bit-level arithmetic rather than integer arithmetic. The leading library implementing CGGI is Concrete [21]. Concrete is a library written in the Rust programming language and is the main library for CGGI operations used in this thesis. It implements the construction and operations detailed above in full. Concrete also contains a hardware acceleration backend powered by Cuda. A successor to Concrete by the same engineering team, TFHE-rs [133], has been released but as of time of writing, does not provide a Cuda-accelerated backend.

There are several reasons why we are utilizing Concrete. The first reason is its support of integer arithmetic. In Concrete, it is possible to define parameters such as p and q which allow each ciphertext to encrypt larger values. In libraries such as cuFHE or NuFHE, integers are represented as $\log p$-sized arrays of ciphertexts, encrypting each bit and expanding the number of operations necessary for simple operations from one to $\log p$ operations. The second reason is the provision of a Cuda backend. Specifically, Concrete provides two Cuda implementations of the PBS operation. This thesis utilizes the **amortized programmable bootstrap** algorithm. This algorithm performs naive parallelism of the bootstrapping procedure by running the regular PBS algorithm on n ciphertexts at a time over several GPUs. Each Cuda block is assigned an LWE ciphertext to bootstrap. Therefore, we are limited by the number of blocks in a GPU. However, scalability is linear—the larger the number of GPUs used to run PBS operations, the faster it is to run them.

Section 2.4 detailed the CGGI scheme. Now that we understand the types of ciphertexts we can create, the types of messages we can encode, and the types of operations we can perform, we can discuss how to process deep neural networks using CGGI operations. One of the limiting factors of processing DNNs with CGGI is its native operation over integers. In contrast, DNNs are generally trained and saved in floating-point format. It is thus necessary to discuss another field of machine learning, which although is primarily focused on conserving energy and inference time, has natural applications to performing privacy-preserving machine learning with CGGI. Quantization of deep neural networks is an important part of this thesis and will be discussed in detail in section 2.5.

2.5 Quantization

In section 2.1.1, we mentioned that one of the greatest leaps in neural network research was the introduction of real-valued weights and signals. Modern deep neural networks continue to employ extended precision using 32/64-bit floating-point encoding of real-values. As DNNs become more widespread and available to consumers, importing and running inference on edge devices with limited computational resources becomes more necessary [60]. With better techniques to simultaneously handle vanishingly small and exploding gradients, model sizes are rising at an exponential rate, increasing the computational resources required at the server level to perform inference [94]. In response, the field of **quantization** has become a very important and well researched area in recent years.

Quantization strives to simplify DNNs in order to dramatically reduce the memory footprint, cost, and computational resources required to run inference. Rather than representing real-values as high-precision 32-bit floating-points, the real domain is discretized into a smaller p number of parts, where if $p = 2^{\omega}$, ω denotes the bit-precision of the values. Therefore, we can reduce 32-bit floating-point to 8-bit fixed-point or integer encoding, which effectively reduces the memory requirement to store values by four times. We also reduce the computational complexity by switching to integer arithmetic, which on a large scale, has the potential to dramatically reduce the energy and time required to process large applications. Quantization research is therefore targeted towards reducing model complexity while maintaining model performance and generalization power.

Our use-case for quantization is a consequence of necessity. CGGI can only process integers. Furthermore, we are strictly limited to integer-only arithmetic. Unless we represent an 8-bit fixed-point integer by 8 ciphertexts—one for each bit—we cannot perform logical right-shift operations in CGGI, a necessary operation for fixed-point integer arithmetic. While it is true that we can fill a lookup table with right-shifted values and evaluate a right-shift through a programmable bootstrap, being the most expensive operation, it can quickly become impractical to bootstrap a large number of values. Therefore, in this thesis, we only consider quantization techniques that produce regular integers with fixed precision, and are compatible with CGGI operations.

In the following sections, we discuss the fundamentals of quantization, which are necessary to understand before discussing the more complex types of quantization used in this thesis. We describe forms of less than three-bit quantization as well as the different ways to apply quantization to a model successfully. We conclude by reviewing the literature surrounding integer quantization of recurrent neural networks.

2.5.1 Fundamentals

Continuous values can be quantized in two different ways: uniformly and non-uniformly. **Uniform quantization** divides a distribution of numbers $\mathcal{D}$ into a set of p equidistant

numbers. Given $\mathcal{D}(\mathbb{R}) \in [\alpha, \beta]$ for $\alpha, \beta \in \mathbb{R}$,

$$d_Q(\alpha, \beta, p) = \frac{\beta - \alpha}{p} \tag{2.20}$$

where d_Q represents the distance between elements after a uniform quantization of $\mathcal{D}(\mathbb{R})$. **Non-uniform quantization** divides a distribution of numbers $\mathcal{D}$ into p numbers of variable distance apart. For example, in logarithmic quantization, distances between values are logarithmic with respect to the bounds of the distribution [70].

Uniform quantization is generally parameterized by a scale $s \in \mathbb{R}$ and zero point $z_Q \in \mathbb{R}$. If the domain of the quantized set of values is symmetrically centered around z_Q, that is, for a domain of $[\alpha, \beta]$, $|\alpha - z_Q| = |\beta - z_Q|$, then this type of quantization is considered to be **symmetric**. Otherwise, it is **asymmetric**. For example, a symmetric quantization of the bound $[\alpha, \beta]$, generally transforms the bound to a quantization bound $[-\gamma, \gamma]$, where $\gamma = \max(|\alpha|, |\beta|)$, with scale $s = d_Q(-\gamma, \gamma, p)$, and zero point $z_Q = 0$. An asymmetric quantization generally transforms the bound to $[\alpha', \beta']$, where $|\alpha'| \neq |\beta'|$, with scale $s = d_Q(\alpha', \beta', p)$, and an arbitrary zero point $z_Q \neq (\beta' - \alpha')/2$.

The general formula for quantization and dequantization is shown below:

$$x_Q = \left\lceil \frac{1}{s} \cdot x + z_Q \right\rfloor \tag{2.21}$$

$$x = s \cdot (x_Q - z_Q) \tag{2.22}$$

where the subscript Q denotes the quantized value. From equation 2.22, we observe that $x \cdot x \neq x_Q \cdot x_Q$ and $x + x \neq x_Q + x_Q$. Therefore, operations must be defined in the quantization space to preserve the correctness of the map. Since these operations are not important to this thesis, we will not define them. Instead, we refer the reader to [74].

Quantization can be applied at different levels within a DNN layer. A layer contains consists of activations and a set of parameters. When quantizing across a layer, the same quantization is used for all parameters and activations. However, activations and parameters can be quantized separately. This means that different quantization can be applied to different channels, combinations of channels, or the entire layer. The level at which quantization is applied, whether at the layer level, channel level, or model level, determines the **quantization granularity** of the DNN. For instance, if quantization parameters depend on the expected value of each input channel separately, each channel will have a separate quantization based on its expected value. Conversely, if quantization is applied to the entire layer, the expected value of all input channels will influence the quantization of the entire layer.

2.5.2 Quantization-Aware Training

Another important concept in quantization is fine-tuning. When quantizing DNNs, statistics are gathered for each distribution that needs to be quantized and are used to set the quantization parameters and quantize the distributions. When this process occurs after training and during inference, it is referred to as **Post-Training Quantization (PTQ)**. There are several PTQ methods that find the optimal quantization parameters for each distribution mentioned in [41]. Quantization can be thought of as adding noise to a DNN. For each quantized value, $x_Q = x + e_Q$, where e_Q is the quantization error. After PTQ, the model does not learn how to mitigate quantization-induced error and suffers accuracy loss as a result. Using **Quantization-Aware Training (QAT)**, the model is retrained quantization noise added [60]. In short, QAT introduces fake quantization nodes. The nodes are added to each member of the DNN layer it is quantizing and the model undergoes training as usual. A fake quantization node quantizes an input and immediately dequantizes it after. This moves the input back into its original distribution with some added rounding noise. Since weights need to be quantized before they are multiplied by inputs, a fake quantization node is placed before the multiplication occurs. Similarly, a fake node is placed before the previous activations are multiplied. The network adjusts its parameters to mitigate the addition of quantization noise and is thus *aware* of the effect of the added noise on the loss objective. Because of its effectiveness in quantizing DNNs [41], this thesis experiments with QAT to perform quantization.

Ideally, the distance between a distribution and a quantized distribution should be minimized. The optimization objective of many papers is to find the closest quantized distribution to the original distribution [41]. That is, out of a set of possible quantizations $\mathcal{Q}$, find $Q \in \mathcal{Q}$ such that,

$$\min_{Q} \ \|\mathcal{D} - Q(\mathcal{D})\|^2 \tag{2.23}$$

where $Q(\mathcal{D})$ is the quantized distribution of $\mathcal{D}$. Different types of quantization impact the objective differently. For instance, fixed-point quantization, which quantizes a domain into fixed-point numbers, tends to create sets closest to the original distribution. One of the main reasons for this is the added precision after the decimal point, which allows for more fine-grained approximations to the original distribution. Unfortunately, unless each bit of a fixed-point value is encrypted, CGGI does not support fixed-point arithmetic so we do not consider these types of methods.

Other types of quantization, such as extreme quantization, have been studied and applied to DNNs executed with CGGI operations [102]. Extreme quantization refers to quantizing with precision $p < 2^8$. These types of methods are very appealing for integration with CGGI since it is recommended to operate CGGI with a precision of less than eight bits [23]. The following sections detail two extreme quantization methods used in this thesis.

2.5.3 Binary Quantization

Binarization of a deep neural network maps real-valued parameters and activations into a domain of two elements. Given an input $w \in \mathbb{R}$, binarization maps $w \to \{-1, 1\}$ or $w \to \{0, 1\}$, depending on the use case. Through binarization, matrix-multiplies become additions and subtractions of the value 1, reducing the computational complexity of the model. The memory footprint reduces as well to at most two bits (if -1 is part of the domain) for each parameter and activation. In this thesis, we focus on the binarization scheme proposed in one of the most cited papers that investigates binarization of deep neural networks, BinaryConnect [28]. In BinaryConnect, two binarization functions are proposed; one deterministic, and the other statistical. Furthermore, the network is trained while binarizing the weights, running a form of quantization-aware training. The binarization function we utilize in this thesis is the deterministic function. Given a parameter $w \in \mathbb{R}$, the deterministic binarization function defined in BinaryConnect is as follows:

$$w_b = \text{Signum}(w) = \begin{cases} +1 & \text{if } w \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.24}$$

BinaryConnect does not binarize activations, rather, it utilizes the ReLU function as the main activation function in all of its experiments which outputs the accumulation of the previous layer's activations (or zero if the accumulated value is less than zero). The activations are therefore in full, floating-point precision. Luckily, the accumulated values are integers. However, the authors of BinaryConnect also use a batch normalization operation that converts activations from integers to real-valued numbers. Thus, we cannot use their exact method for integration with CGGI. One of the most interesting contributions of the paper is the setup for training a binarized neural network. During the forward step, the weights undergo binarization, and during the backward step, gradients are computed using the binarized weights. However, in order to to maintain a continuous loss landscape, the weights are stored in floating-point precision and the updates are performed using floating-point precision. During inference, the floating-point weights are binarized. The updates to the weights push the weights past the threshold of binarization, which in the case of the Signum function is equal to 0, which allows the weights to change sign and move towards the loss minimization objective. A variant of this type of training is investigated in this thesis.

There are other renowned papers that propose other binarization methods. Binarized Neural Networks (BNN) [29], which is a follow-up paper to BinaryConnect by the same authors, binarizes weights as well as activations using equation 2.24. In BinaryConnect, the authors use the ReLU function for activations, which has a well-defined derivative. Conversely, the Signum function has a derivative equal to 0 everywhere. In order to enable effective backpropagation, BNN utilizes the **Straight Through Estimator (STE)** [9] as

the derivative of the Signum function. During backpropagation, the STE forwards the gradient flowing backwards g_Q while clipping it to be between $[-1, 1]$. To aid in our definition of the STE, let us define the clipping function. Given an input $x \in \mathbb{R}$ and clipping range $[\alpha, \beta]$, the clipping function is defined as follows:

$$\text{Clip}(x, \alpha, \beta) = \begin{cases} \alpha, & x < \alpha \\ x, & \alpha \leq x \leq \beta \\ \beta, & x > \beta \end{cases} \tag{2.25}$$

Therefore, given an input x and activation $a = \text{Signum}(x)$, the STE transforms the derivative of the cost function with respect to x as follows:

$$\frac{dC}{dx} = \frac{dC}{da} \cdot \frac{da}{dx} = g_Q \cdot \frac{da}{dx} \approx \text{Clip}\left(g_Q, -1, 1\right) \tag{2.26}$$

Interestingly, this derivative is the `hard-tanh` function. This means that while the forward pass performs quantization, the backward pass can set its own derivative, separate from the forward pass, while allowing the model to converge. The derivative can be set according to the dynamics the practitioner wants to model. [114] provides an excellent summary of the forward and backward functions used in a variety of important papers in table 3. [103] mentions that the STE is too rough of an estimate for the derivative of the Signum function. They suggest the use of the `tanh` function as opposed to the STE.

BNN, similar to BinaryConnect, stores its activations in fixed-point format. Despite activations being binary integers, BNN incorporates batch normalization with a modified algorithm. The authors introduce a technique to apply batch normalization to fixed-point encoded values, employing logical bit-shift operations. However, performing CGGI operations over fixed-point integers remains infeasible. Batch normalization, recommended by BinaryConnect and BNN, is a commonly employed operation in quantization. We explore the inclusion of batch normalization in quantized neural networks, and specifically recurrent neural networks, in section 2.5.5.

BinaryConnect and BNN achieve remarkable results on small datasets considering their extreme impact on the precision of existing networks. According to [103], using a variant of VGGNet [126], the VGG-Small network [134], the full precision evaluation achieves 93.8% accuracy on the CIFAR-10 image dataset [63]. BinaryConnect and BNN achieve 91.7% and 89.9% accuracy respectively. However, for an implementation of AlexNet [64] over the ImageNet dataset, BinaryConnect and BNN achieve 35.4% and 27.9% accuracy, much lower scores than a full precision score of 57.1% accuracy. Clearly, binarization does not work as well for larger models and datasets. To mitigate the accuracy loss, researchers shifted to utilizing scaled binary values (from $\{-\alpha, \alpha\}$ where $\alpha \in \mathbb{R}$) represented by fixed-point integers instead of ± 1 [107, 135, 57]. The intuition is that scaled binary values are closer to the actual distribution. For a parameter matrix $\boldsymbol{w} \in \mathbb{R}^{m \times n}$ where $m, n \in \mathbb{Z}_{\geq 1}$, the

optimization objective for scaled-binarization becomes:

$$\min_{\alpha, \boldsymbol{w}_b} \|\boldsymbol{w} - \alpha \boldsymbol{w}_b)\|^2 \tag{2.27}$$

where the objective is to find $\alpha \geq 0$ and a binary parameter matrix $\boldsymbol{w}_b \in \{-1, 1\}^{m \times n}$ that satisfy the minimization objective. Although this approach yields better model performance, it introduces fixed-point values that are incompatible with the CGGI scheme. However, we gain some other interesting ideas from these methods. For example, Wide Reduced-Precision Networks (WRPN) [87] propose that increasing the model size by 3x its original size dramatically increases the binarized model accuracy. The authors suggest that adding more degrees of freedom to each layer increases the representational power of the model, which can regain the accuracy lost as a result of quantization. For binarized parameters and activations, the accuracy of running ResNet-34 [48] on the ImageNet dataset [64] increases from 60.54% for 1x the original model size, to 69.85% for 2x, and 72.38% for 3x. Despite their use of scaled-binarization, we investigate the results of tripling the model size with ternarization in chapter 4 and find that it helps with our novel quantization process.

A theme that exists amongst many papers is leaving the first and last layer in full precision, or non-extreme quantization, when binarizing the rest of the model [103]. It is suggested that much of the information in the input layer to the model is important and binarization, or even ternarization, causes a overwhelming loss of information. Similarly, the output layer is important to define and complete the machine learning task being modelled by the deep neural network. Reducing the amount of information in the last layer negatively impacts the accuracy more so than any other layer (except the input layer). In chapter 4, we observe that it is important to keep the output layer in higher precision and thus implement our quantization process accordingly.

2.5.4 Ternary Quantization

Ternary quantization maps real-valued distributions to a distribution over the set of three numbers $\{-1, 0, 1\}$. One of the advantages of ternarization is its natural pruning capabilities. When multiplying a ternarized matrix $\boldsymbol{w}_T \in \{-1, 0, 1\}^{m \times n}$ with a vector $\boldsymbol{v} \in \mathbb{R}^n$ for $m, n \in \mathbb{Z}_{\geq 1}$, the elements in $\boldsymbol{w}_T$ equal to 0 can be skipped in the matrix multiply operation, effectively reducing the size of the matrix and computational complexity of the model. Ternarization also replaces multiplication with addition. Since we are multiplying by values in $\{-1, 0, 1\}$, effectively, we are adding, subtracting, or skipping, which reduces the complexity attached to multiplication operations. A ternarized distribution tends to match a normal distribution, which is a typical distribution for weight matrices in deep neural networks, better than a binarized distribution. This is due to the clustering of the majority of values in a normal distribution around 0. Since they are closer to 0 than to ± 1, including an element equal to 0 in the quantization domain decreases the distance be-

tween the normal distribution and the quantized distribution. Similar to BinaryConnect, TernaryConnect [76] contains a stochastic ternary quantization functions and extends the results in BinaryConnect to the ternary domain. This thesis utilizes the deterministic function proposed in Ternary Weight Networks (TWN) [72]. Given a parameter $w \in \mathbb{R}$, TWN defines the following deterministic ternarization function:

$$w_T = \text{Ternary}(w, \delta) = \begin{cases} -1, & w < -\delta \\ 0, & |w| \leq \delta \\ +1, & w > \delta \end{cases} \tag{2.28}$$

where $\delta \in \mathbb{R}$ is a threshold value, and $w_T \in \{-1, 0, 1\}$. TWN also seeks to minimize the objective in equation 2.27, where instead of finding $\boldsymbol{w}_b$ and α, the objective is to find the optimal $\boldsymbol{w}_T$ and α parameters that satisfy the minimization objective. Since we cannot use real-valued numbers with CGGI, we disregard the α parameter and focus only on the calculation and use of $\boldsymbol{w}_T$.

Unlike the Signum function, the Ternary function is parameterized by δ. If the Ternary function is used to ternarize only the parameters of a model (similar to TWN), then the threshold becomes a hyperparameter of the deep neural network. TWN offers some guidance on the selection of δ. By solving the modified minimization objective in 2.27, the authors of TWN suggest that,

$$\delta \approx 0.7 \cdot \mathbb{E}\left(|\boldsymbol{W}|\right) \tag{2.29}$$

where $\mathbb{E}(\boldsymbol{W})$ is the expected value across the elements of any parameter matrix $\boldsymbol{W}$. To generalize, the expectation in equation 2.29 should be taken over the parameter matrices that are considered part of the quantization granularity. For instance, if the quantization granularity is defined such that each layer requires different quantization, then the expectation should be applied over every element in every parameter matrix per layer. Trained Ternary Quantization (TTQ) [136] proposes an alternative way to evaluate δ. In TTQ, a constant factor t is maintained for all layers while $\delta_l = t \cdot \max(|\boldsymbol{W}|)$ is different for each layer.

2.5.5 Quantization of Recurrent Neural Networks

The main focus of this thesis is to enable processing of large-scale recurrent neural networks in the fully homomorphic encryption domain. Since the main FHE scheme used in this thesis is CGGI, this section details the current state-of-the-art in integer quantization of recurrent neural networks. *Ott et al.* [95] provide one of the first investigations of the ability to quantize recurrent neural networks. In their work, they experiment with weight quantization only and leave activations and matrix-multiply accumulation steps in floating-point. With respect to weight quantization, they test four different methods: (1) binarization, (2)

ternarization, (3) power-of-two ternarization, and (4) exponential quantization. Power-of-two ternarization and exponential quantization quantize values to power-of-two representations. This allows the practitioner to replace all multiplications between quantized values with logical bit-shift operations. However, this is only possible because of the fixed-point representation of the quantized values. Binarization and ternarization are implemented as defined in equations 2.24 and 2.28 respectively. All types of quantization are tested on three different types of RNNs: (1) Vanilla, (2) GRU, and (3) LSTM. For a detailed construction of each type of RNN, refer to section 2.2. The authors present remarkable results, some of which perform better than their baseline, state-of-the-art, real-valued results. Ternarization proves to be very effective at quantizing different types of RNNs. However, since activations and matrix-multiply accumulations are not quantized, this work serves solely as a reference for parameter quantization.

A major observation in [95] is the failure of binarization across all RNNs. The authors observe that binarization causes instability in the hidden state dynamics of the RNN. Specifically, the gradients of the hidden states explode as the timesteps increase, a phenomenon they do not observe with any other quantization method. They attribute this failure to the wide variance caused by mapping values close to 0 to ± 1. It is clear that this wide quantization error can explode when we consider its input to the large multiplication chain in equation 2.5, which occurs in every type of RNN. Figure 1b in [95] shows that binarization explodes the norm of the hidden states as the timesteps advance. *Ott et al.* [95] provide an empirical observation of the cause of this problem while *Hou et al.* [57] provide a theoretical proof that binarization, in its purest form, cannot quantize the weights in an RNN. Loss-Aware Binarization (LAB) [57] is a paper that investigates the impact of binarization of both the weights and activations in a DNN on the loss function. The authors suggest a new training algorithm that harnesses the curvature of the network, impacted by binarization, to learn a scale parameter α and binarized $\boldsymbol{w}_b$ for each parameter. This allows them to perform state-of-the-art scaled-binarization of the weights in the network and use the Signum function for activations successfully. They prove that scaled-binarization can be applied to RNNs since the scale stabilizes the hidden dynamics of the RNN cell. In Loss-Aware Weight Quantization (LAWQ) [56], the same authors generalize the training algorithm proposed in LAB to be able to quantize in any bit precision, including ternary. They propose a scaled-ternarization method that achieves state-of-the-art in their RNN experiments, while it is uncertain if activations are quantized.

As mentioned in the previous section, normalization is a common theme surrounding the successful quantization of recurrent neural networks. *Ardakani et al.* [4] were the first to apply batch normalization to quantized LSTMs. In their paper, they experiment with binarization and ternarization. Following LAB, they perform an analysis of the hidden states and input/output gates of an LSTM network. Their findings indicate that the probability distributions of the input and output gates (equations A.4 and A.7 in appendix A) center

around 1 rather than symmetrically around 0. This reduces the intended effect of adding gates to RNNs since by allowing every signal to pass through (result of the gates being equal to 1), the gates do not regulate the gradients at all. In other words, binarization causes the distributions to degenerate. To mitigate this problem, they suggest to perform batch normalization after every matrix-multiply in the LSTM network. However, although they binarize/ternarize the weights prior to matrix-multiplying with the input activations, the batch normalization operation is executed *after the quantized matrix-multiply*, changing the precision of the outputs back to floating-point. Similarly, the authors of LAB study the general application of different types of normalization in a subsequent paper [58]. In this paper, they provide a theoretical proof of the statement they make in LAB that quantizing LSTMs is difficult due to the exploding gradient problem. In response, they analyze the effects of weight [120], layer [6], and batch normalization [59] on the hidden dynamics of quantized LSTM networks. They show that any type of normalization decouples the weight gradients from any type of quantization, therefore cancelling out the negative effect of quantization on the gradients. Using normalization allows them to achieve state-of-the-art results on quantized LSTM networks that are almost identical to their full precision siblings. As in [4], normalization is applied after every matrix-multiply between inputs and quantized weight matrices.

Although scaled-quantization and normalization do not provide a path towards CGGI-compatible RNNs (due to CGGI not supporting fixed-point arithmetic), they are effective in reducing the computational complexity and memory footprint of deep neural networks. By quantizing weights and activations, although some operations occur in floating-point, the vast majority of operations which mainly consist of matrix-multiplies, become simple bit-shifts or additions and subtractions. Moreover, low-bit integers require much less memory allocation. In summary, other important works such as [49, 122] as well as those that have been discussed in this section quantize RNNs using at least one of the following methods: (1) quantize in fixed-point format, (2) binarize, ternarize, or quantize to other precisions over the integers while keeping matrix-multiply accumulations in high precision, (3) quantize weights and not activations, or (4) multiply a real-valued scalar by the quantized weights. Unfortunately, none of these methods are compatible with CGGI. However, some of these methods can be used and expanded upon to enable quantization of RNNs that are compatible with CGGI, as we discuss in later sections of this thesis. In the next section, we discuss privacy-preserving machine learning (PPML), which is the cross-section between deep neural networks and fully homomorphic encryption research. We begin by defining inference over encrypted data. We then discuss the state-of-the-art methods involved in performing inference using CGGI. In the last subsection, we conclude by discussing the limited area of research surrounding the execution of recurrent neural networks over the encrypted domain.

2.6 Privacy-Preserving Machine Learning

As machine learning continues to become more commercialized, the threat of compromised machine learning models and the data that powers them increases. Oftentimes, datasets are compiled by commercial entities using data that belongs to them or their customers in order to benefit their business. As new regulations for data privacy such as GDPR [27] enter into law, companies are required to find ways to enhance the security of their models and datasets. Privacy enhancing technologies such as Secure Multi-Party Computation (SMC) and Homomorphic Encryption (HE) can perform secure computation on data, thus satisfying legal obligations. These technologies can enhance the security of machine learning services by either protecting the datasets, the models, or both. By performing secure operations over obfuscated or encrypted data, they ensure important data is never leaked in the machine learning process. In this thesis, we focus on protecting input datasets only; that is, running inference of a model with plaintext weights (a *plaintext model*) over encrypted input data.

Privacy-preserving machine learning (PPML) is generally characterized by the two technologies mentioned above. SMC is a set of techniques that enable several mutually distrusting parties to compute public functions over a combined input [108]. The security objective is to guarantee no party can learn any information about another party's input data, aside from what it receives while interacting with the protocol. For example, with respect to machine learning, consider a public function being the evaluation of an inference. SMC allows multiple parties to combine their information privately and perform an inference over their combined data. A characteristic of the SMC set of protocols is that they are **interactive**, meaning they require large amounts of data to be transferred between parties at many points during the evaluation of the public function. This causes increased communication overhead that could otherwise be avoided by using homomorphic encryption. As mentioned previously, HE enables the evaluation of neural network processes over encrypted data. For example, consider a machine-learning-as-a-service type of interaction between a client and server. The client encrypts data and sends it to the server for encrypted inference. The server, being the owner of the model, performs inference over the encrypted data using HE operations. The server returns the encrypted result of the inference to the client and the client decrypts the result. In this scenario, the communication between the client and server is considered **non-interactive** since the client does not take part in the inference calculation. This type of interaction guarantees protection of the input data. In contrast, there are interactive protocols in PPML that require the client to decrypt, compute intermediate results, and re-encrypt. Similar to SMC protocols, these types of interaction suffer from large data overhead in communication [102], and possibly more severely than SMC. Thus, the neural networks evaluated in this thesis are all operated non-interactively and utilize FHE. For a discussion on the security and threat model of privacy-preserving neural networks we assume in this thesis, refer to appendix B.2.

2.6.1 General Inference Over Encrypted Data

An inference is composed of several operations that occur sequentially in the forward step of a model. Operations in a plaintext machine learning model are executed over parameters and inputs in plaintext. These include operations such as matrix multiplication, activation function execution, normalization, dot-products, etc. The objective of PPML is to reconstruct these operations using FHE operations over encrypted data. Reconstruction of these operations can vary across different schemes according to the types of encoding and FHE operations available. For instance, CKKS [19] can encrypt fixed-point numbers and perform any operation over batched values, while CGGI [22] can encrypt integers/bits and cannot perform all operations in batched form. Also, CKKS can only execute inferences of a certain depth while CGGI can execute unbounded inferences. Thus, there are advantages and disadvantages of using each scheme. As a result, the literature is generally divided between the use of either B/FV [33], BGV [15], and CKKS, versus the use of CGGI to perform inference over encrypted data—or in other words, the use of levelled HE versus FHE to perform PPML [71].

The LHE schemes mentioned above all perform Single Instruction, Multiple Data (SIMD) operations. While they may lack the small latency of CGGI, they excel in amortized latency over multiple ciphertexts. The first PPML implementations utilized LHE schemes [102]. Implementations such as CryptoNets [42] were successful in applying a small convolutional neural network over encrypted data, for an amortized time of 60ms per ciphertext, but with a total latency of 250s for each inference—a result of the most computationally intensive operation in LHE schemes: ciphertext-ciphertext multiplication. Since LHE schemes cannot perform non-linear operations, they approximate them using polynomials [102]. For instance, it is well known that Taylor series approximations of non-linear functions are excellent approximations within a certain bound, and around a certain value. However, evaluation of polynomials involves ciphertext-ciphertext multiplications, limiting the depth and increasing the latency of circuits that involve these operations. Activation functions in neural networks are non-linear functions and are thus approximated by polynomials by LHE schemes (explaining the large latency in CryptoNets [102]). In contrast, the CGGI scheme is able to perform an unbounded number of activation functions using the programmable bootstrap [13] (refer to section 2.4.4 for more details). It is true that the inability to batch bootstraps is a limiting factor of CGGI. However, standard parallelization can help to circumvent this challenge (as shown in chapter 4). LHE schemes such as CKKS also have the unique ability to encode fixed-point values of large precision. Therefore, many operations, aside from activation function evaluation, are simple to reconstruct in the FHE domain using CKKS. For example, batch normalization, which involves scaling and translating neurons, can be performed natively [102]. CGGI is restricted to integer arithmetic, which requires the additional quantization step (section 2.5). Scaling operations are therefore not possible without performing programmable bootstrapping. Thus, many operations

that are simple with CKKS are more difficult to reconstruct using CGGI. Nevertheless, as this thesis aims to conduct RNN inference on encrypted data, and given the deep nature of RNN architectures, we are compelled to employ the CGGI scheme. Therefore, in the next section, we discuss how to construct DNNs that are compatible with CGGI.

2.6.2 Inference Over Encrypted Data Using CGGI

In this section, we summarize the current methods of converting DNNs to CGGI-compatible DNNs. We draw inspiration from five papers that perform conversions differently. In tables 2.1 and 2.2, we summarize the quantization strategy, input representation, matrix multiplication operation, dot-product accumulation strategy, activation functions, types of models evaluated, performance, FHE security levels, and acceleration capabilities described within each work. For the quantization attributes (input, weight, and activation), *integer* quantization refers to any quantization to integers of precision greater than two bits. Qualifiers in front of *integer* give a more specific description of the quantization, but still refer to integers of precision greater than two bits. Binary and ternary types refer to integers of precision one and two bits respectively. *Multi-CT* and *Single-CT* refer to whether multiple ciphertexts or a single ciphertext represent a single entity that is operated upon. For simplicity, we refer to a model that runs encrypted inference using CGGI as a CGGI model. For more detailed information regarding each paper and attribute summarized by the tables, refer to appendix E.

From tables 2.1 and 2.2, we make a few general observations. Firstly, models can be categorized by the type of input representation: (1) arrays of ciphertexts (Multi-CT), or (2) a single ciphertext (Single-CT). Multi-CT models are generally required to operate circuits of multiple bootstraps to perform operations that are otherwise simpler with Single-CT models. As a result, Multi-CT models are generally much less efficient than Single-CT models. In contrast, they achieve better accuracies due to their (1) bit-wise representation, which allows execution of models with larger precision, (2) utilization of division operations through right-shifting of ciphertexts (important for fixed-point quantization), and (3) the application of better quantization techniques (for instance, logarithmic). In chapter 3, we introduce a method that allows us to use the more efficient Single-CT representation *and maintain accuracy* in quantized RNNs. Secondly, CGGI models that are executed using hardware acceleration, especially with GPUs, perform much more efficiently than models executed on a single CPU. Thirdly, we also observe that the Signum function is a popular and effective replacement for other activation functions. Fourthly, lower security levels provide more efficient executions than larger security levels, at the cost of less security. Through the techniques proposed in this thesis, we reduce the required precision allowing the use of more efficient, larger security levels. Lastly, binarization or ternarization can obtain good accuracy close to that of full precision models. In section 2.6.3, we discuss the state-of-the-art in converting RNNs to FHE-compatible RNNs.

	Prior Work				
Attributes	REDSec [34]	Concrete-DNN [20]	SHE [78]	TAPAS [121]	FHE-DiNN [13]
Input Representation	Multi-CT	Single-CT	Multi-CT	Multi-CT	Single-CT
Input Quantization	Symmetric Integer	Integer	Logarithmic Integer	Integer	Symmetric Integer
Weight Quantization	Ternary	Integer	Logarithmic Integer	Binary or Ternary	Symmetric Integer
Activation Quantization	Binary	Integer	Integer	Binary	Binary
Activation Function	Signum	Any w/ PBS	ReLU	Signum	Signum
Quantization Method	QAT	PTQ	PTQ	QAT	PTQ
Matrix Multiplication Operations	Mult. Circuit + Bridging + Addition	Scalar Mult. + Addition	Addition Circuit	Addition Circuit	Scalar Mult. + Addition
Accumulation	Single-CT	Single-CT	Multi-CT	Multi-CT	Single-CT

Table 2.1: **Summary of CGGI-Model Constructions in Various PPML Papers**. This table summarizes the various methods each PPML paper adopts for converting models to CGGI-compatible models. *Single-CT* and *Multi-CT* refer to the representation of integer data through one ciphertext or multiple ciphertexts, respectively.

Prior Work	**Experiment Descriptors and Results**						
	Hardware	Security Level	Model Name	Estimated No. Parameters	No. Hidden Layers	Runtime (ms)	Accuracy
FHE-DiNN [13]	CPU (4 Cores)	80-bit	FHE-DiNN30	24,000	1	515	93.71%
			FHE-DiNN100	80,000	1	1679	96.35%
TAPAS [121]	CPU (16 Cores)	Unknown	torch7 [29]	10M	3	234.36M	97.3%
SHE [78]	CPU (10 Cores)	152-bit	CNN	130,000	4	9000	99.54%
Concrete-DNN [20]	GPU (8 A100s)	128-bit	NN-20	230,000	20	7530	97.1%
			NN-50	480,000	50	18,890	94.7%
			NN-100	910,000	100	37,650	83%
REDsec [34]	GPU (8 T4s)	110-bit	FF-96	86,000	2	780	93.1%

Table 2.2: **Summary of CGGI-Model Experiments in Various PPML Papers**. The experiments in this table are conducted over the MNIST dataset [66]. Each model is trained on the MNIST dataset. All metrics are associated with evaluations over encrypted data. For CPU implementations, it is unclear whether parallelization was involved (except TAPAS where there is parallelization using 16 cores over the adder circuits). For GPU implementations, parallelization was involved.

2.6.3 Privacy-Preserving Recurrent Neural Networks

As discussed and analyzed in sections 2.2 and 2.5.5, recurrent neural networks are difficult to convert to FHE-compatible networks due to their large depth and sensitivity to quantization. However, there are some papers that propose techniques to evaluate RNNs in the FHE domain. For instance, CryptoRNN by *Bakshi et al.* [8] and a paper by *Podschwadt et al.* [100] utilize the CKKS [19] scheme to execute RNNs over encrypted data. However, since CKKS is a levelled FHE scheme, functions that use CKKS operations are bounded in depth. To solve this problem, both papers suggest an interactive approach, which is briefly discussed at the beginning of section 2.6. They propose that operations that either increase the depth or require approximation, such as calculating activation functions, should be evaluated in the clear by the owner of the data whenever necessary. This is an intriguing idea since quantization would not be required, solving one problem, and depth would become unbounded, removing the second obstacle. Since this thesis focuses on executing RNNs over encrypted data in an end-to-end setting, the methods in these papers are inapplicable.

Podschwadt et al. released a follow-up paper [101] that proposes a non-interactive approach. Their approach involves changing the structure of the RNN being converted. Given an RNN with τ timesteps, [101] proposes a division of the RNN across the time dimension such that one RNN becomes n sub-RNNs, and τ/n timesteps are executed in each. By dividing the RNN into several RNNs of lower depth, CKKS becomes a scheme that can be applied to any RNN in a non-interactive setting. The goal of RNNs is to model the evolution of a sequence across time. While short-term dependencies can benefit certain applications where contextual information is not necessarily required, long-term dependencies are important to model more complex sequences such as language sequences. By dividing an RNN input sequence into several independent RNNs, long-term dependencies are not guaranteed to be modeled. In [101], table 1 shows a drop in accuracy of around 5% in each metric, for each experiment between the full precision and divided models.

In terms of execution time, RNNs tend to be tasks that are time sensitive, meaning model execution latency is more important than throughput (for instance, speech recognition). In [101], the fastest implementation runs in 19.5 minutes. The throughput, as a result of CKKS batching, yields 14 inferences per second. However, an end-user would still need to wait 19.5 minutes for an inference regardless. An interesting observation is that the authors retrain their baseline, full precision model with polynomial approximations to recover the performance lost as a result of the switch. Similar to binarization, switching the activation function can cause exploding or vanishing gradients in recurrent neural networks. To mitigate this problem, the authors increase the effect of L2 regularization on the network so as to dampen the gradients, bettering performance.

As mentioned in the previous section, another paper, SHE [78], researches FHE-compatible RNNs. The authors claim to have proposed the first method to successfully convert LSTM models to FHE-compatible LSTM models. To convert an LSTM to an FHE-compatible

LSTM, the authors draw inspiration from the IRNN paper [65]. The IRNN paper shows that LSTMs can be successfully converted to vanilla RNNs that use the ReLU activation function rather than `tanh`. Inspired by these results, the authors train a single-layer RNN with 300 units, over 25 timesteps, to predict the next word in a sentence. They evaluate their methods using the Penn Treebank dataset [82]. They convert the inputs and parameters to the network using the same methods discussed in section 2.6.2. They utilize their ReLU evaluation circuit to perform the ReLU activation function in the FHE domain. The converted CGGI model achieves a runtime of 576s for 25 timesteps and degrades the plaintext performance by 2.1%. While the authors are successful in producing an RNN evaluation over encrypted data, it is incorrect to claim that they produced an LSTM evaluation since an LSTM is very different from a vanilla RNN. SHE also suffers from the same problem as [101] with respect to long latency by taking almost 600 seconds to predict the next word.

Summary

This section shows that the literature surrounding privacy-preserving recurrent neural networks is limited. Some papers use levelled schemes like CKKS which as a result, achieve good accuracy and throughput, while completely changing the structure of traditional RNNs and increasing the latency beyond acceptable levels [101]. Other papers use FHE schemes such as CGGI and as a result, achieve good accuracy without having to change the traditional structure of RNNs, but also suffer from large latency due to multi-ciphertext input representations and expensive gate-level operations [78]. It is clear that unless CKKS becomes unbounded, it is impossible to evaluate large-scale RNNs without having to completely alter the structure of the RNN. With respect to CGGI, gate-level operations of homomorphic circuits are too expensive to be practical for long sequences. From the literature (specifically FHE-DiNN [13], Concrete-DNN [20], and REDsec [34]), it is clear that integer operations are much more efficient and therefore, more desirable. Thus, this thesis proposes a method to evaluate large-scale RNNs using CGGI and integer arithmetic. Large-scale RNNs can be long in the number of timesteps and wide in the number of parameters. To convert these types of models to FHE-compatible models, both cases need to be considered. CGGI and our quantization strategy, discussed in chapter 4, are successful in addressing the length of these networks. The following section addresses the width of these networks. Large widths result in large matrix multiplies where accumulations in dot products can overflow the plaintext modulus p. The next section proposes a novel regularizer that teaches the network to produce the correct results after accumulation, independent of the choice of plaintext modulus.

Chapter 3

Overflow-Aware Activity Regularization

The first contribution of the thesis is presented in this chapter. As mentioned in earlier sections, when dealing with large models, integer accumulation can lead to the wrapping of integers within the plaintext modulus of an LWE ciphertext. This modulo wrapping effect can result in incorrect outputs for activation functions. This problem is discussed in detail in section 3.1. In section 3.2, we introduce Overflow-Aware Activity Regularization (OAR), a novel regularization method that teaches a model to adjust its pre-activations to fall within favourable regions of the domain, facilitating correct outputs of activation functions. Effectively, this diminishes the problem of overflow in discretized neural networks. In the final sections of this chapter, we discuss implementation details and experimental results. According to our results, this contribution is necessary to enable successful inference of recurrent neural networks using CGGI and is thus, an integral part of the methods discussed in chapter 4.

3.1 The Problem With Accumulation

Within every neural network, the multiplication and accumulation between parameters and inputs is represented by $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$, which we refer to as "pre-activations". Pre-activations are fed into activation functions to evaluate the non-linearity necessary for neural networks to learn. The combination of pre-activations and activations realizes a perceptron, the building block of deep neural networks (as discussed in section 2.1.1). Since the calculation of pre-activations is a linear transformation of some input vector $\boldsymbol{x} \in \mathbb{R}^n$ by parameters $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$, the set of m values that result from the calculation form a distribution with a domain $[\alpha, \beta]$. The pre-activation distribution can be visualized using a histogram. In figure 3.1, the pre-activation distribution of a dense layer is shown. The distribution is an exponential moving average (EMA) of the pre-activation at each forward

step, for each epoch, with smoothing value equal to 0.9 . We experimented with a dense layer with ternarized weights ($w_{i,j} \in \{-1, 0, 1\}$) and binarized inputs ($x_j \in \{-1, 1\}$), resulting in pre-activations that are integers. In the figure, the dark orange curve represents the EMA for the most recent epoch and the lighter shades represent the EMA for older epochs, with a lighter orange colour denoting an older epoch. The bounds of the distribution stabilize around $B = [-98, 113]$ and therefore, $z_i \in [-98, 113] \subset \mathbb{Z}$ for $\boldsymbol{z}_i \in \mathbb{Z}^m$.

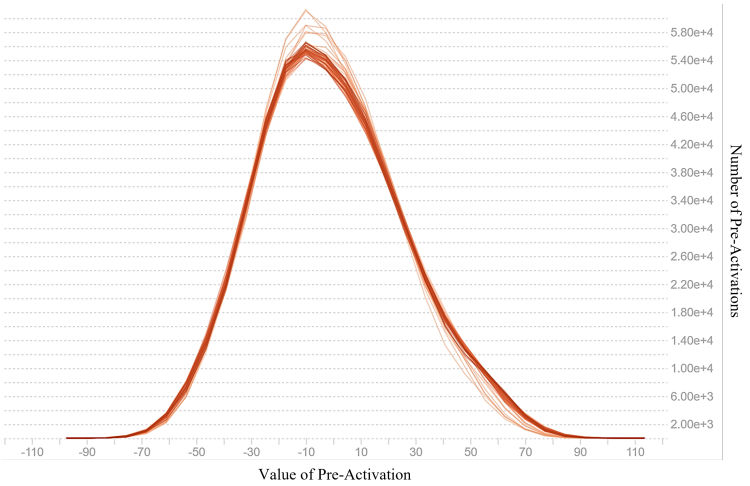


Figure 3.1: **Large Pre-Activation Distribution**. This figure shows the distribution of pre-activation values from a dense layer with 1792 input units and 1024 output units. The distribution contains values from a batch of 512 independent input vectors. The parameters are ternarized and the inputs are binarized. The shade of the orange colour depicting the distribution is lighter for older epochs. The vertical axis represents the number of values.

By using the Signum function for the activation function of this layer, it is simple to see that all pre-activations greater than or equal to zero would be classified as positive (the opposite is true for negative values). This statement is true for the set of integers with minimum bound equal to B and maximum bound equal to $[-2^{\omega-1}, 2^{\omega-1} - 1]$, where ω is the number of bits representing a word in the host operating system ($\omega = 32$ in our case since we are operating with floating-point precision on GPUs). In signed integer arithmetic, which is the set of operations supported by CGGI, integers that surpass the lower or upper bounds of the signed domain, flip the sign bit and become different values than expected. The term for this is **overflow**. For example, an 8-bit integer can represent any integer in $[-128, 127]$. If the integer accumulates to 128, then it automatically becomes -128. Similarly, if a dot product between binary and ternary values accumulates to -129, the integer automatically becomes 127. In both cases, the accumulated value has a different sign than the 8-bit represented value. Mathematically, this is due to modular arithmetic, where in this case, the modulus is $2^8 = 256$. We can define the signed function as the

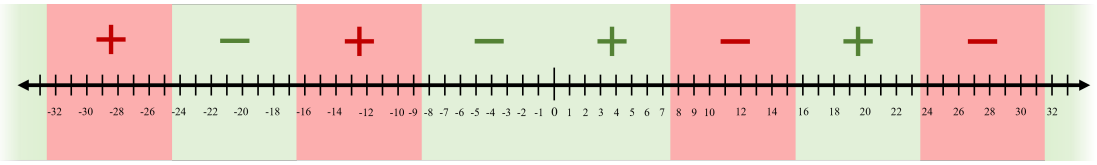


Figure 3.2: **Accumulation Regions**. This figure illustrates the regions where the sign function correctly (green shading) and incorrectly (red shading) outputs the sign for values within a modulus of 16. The "+" or "-" signs above the number line represent the output of the sign function for the values in the shaded region to which they belong. In each red region, they represent the erroneous sign produced by the sign function for the values in that region. Conversely, in the green regions, the signs displayed are correct and match the sign function's output for the values within those areas.

following mapping from an integer $s \in \mathbb{Z}$ to an integer $q \in \mathbb{Z}_p$:

$$q = \text{signed}(s, p) = \begin{cases} s \bmod p, & s \bmod p < \frac{p}{2} \\ -(p - (s \bmod p)), & s \bmod p \geq \frac{p}{2} \end{cases} \tag{3.1}$$

where $p = 2^\omega$ designates the domain of s to be $[\alpha, \beta] = [-\frac{p}{2}, \frac{p}{2} - 1]$. With respect to figure 3.1, if $p = 2^8 = 256$, then $B \in \mathbb{Z}_{256} = [-128, 127]$ and the Signum function outputs the correct signs. However, if $p = 2^7 = 128$, then $B \notin \mathbb{Z}_{128} = [-64, 63]$ and values that surpass either the lower or upper bound are overflowed, causing the Signum function to output the incorrect sign.

Since CGGI can practically support only 8-bit signed integers, accumulation can quickly lead to overflow. In fact, FHE-DiNN [13] performs a study on correct and incorrect signs after each layer, thereby recognizing that it is a problem. The vast majority of ciphertexts that are classified incorrectly in their study contain incorrect evaluations of the sign function. This is a result of their large division of the torus, causing the error in the ciphertext to influence the final decryption. Despite achieving good accuracy in their models, larger and deeper networks would cause an error in the sign to grow exponentially with each layer. To put the severity of this problem into perspective, an incorrect sign output is a very large distance away from a correct sign output (since $1 - (-1) = 2$). If we represent an activation $a_{noisy} = a_{original} + e$ where e is the error, then if $e > 1$ and since $w \cdot a_{noisy} = w \cdot a_{original} + w \cdot e$, $w \cdot e$ grows. After many layers, it can severally degrade the results.

Figure 3.2 helps visualize the effect of overflow on the output of the Signum function. In this example, consider a 4-bit integer and modulus of $p = 2^4 = 16$. The figure displays a number line with feasible values for a dot-product accumulation in $\mathbb{Z}$. The signs above the number line are the outputs of the Signum function for the values in the shaded region, modulo p. In a different setting, they could be the sign of an LWE ciphertext encrypting the accumulated value in $\mathbb{Z}_p$. The figure shows that certain regions, such as those shaded in red, cause the accumulated values to overflow *incorrectly* in $\mathbb{Z}_p$. The term *incorrectly* refers

to the sign of the accumulated value in $\mathbb{Z}$ being different from the sign of the same value in $\mathbb{Z}_p$ due to overflow. For example, in the region $[-16, -9]$, the sign of all accumulated values is "-1"; however, due to overflow, the sign in $\mathbb{Z}_p$ is "+1". In this case, the activation function would output the incorrect sign.

Figure 3.2 also shows that there are regions, such as those shaded in green, that overflow *correctly*, meaning that the signs of the accumulated values in those regions are the same as those in $\mathbb{Z}_p$. One example is an accumulated value of "18". The sign of this value in both $\mathbb{Z}$ and $\mathbb{Z}_p$ is "+1". In this case, the activation function outputs the correct sign, regardless of the fact that the number has overflowed. **This observation is crucial—there are regions in the domain where overflow is inconsequential**. In fact, 50% of the values in $\mathbb{Z}$ already yield a correct result from the Signum function in $\mathbb{Z}_p$. This could be a reason why FHE-DiNN [13], which uses the Signum function for the activation functions in its networks, and signed integer arithmetic for accumulation, achieves good accuracy, despite an abundance of incorrect outputs. However, this is only speculation and is left to be investigated in future work.

This observation is a double-edged sword. In one case, 50% of the accumulated values fall within the correct regions that output the correct sign in $\mathbb{Z}_p$. In another, 50% of the values also fall within the incorrect regions that output the incorrect sign. **This is where we can make a second observation—if an accumulated value falls within an incorrect region, overflowing the value once more, which flips the sign bit, places it in a correct region**. For example, considering figure 3.2 once more, if the accumulated value of "12" falling in incorrect region $[8, 15]$ can be moved to correct regions $[0, 7]$ or $[16, 23]$, then the Signum function outputs "+1" in $\mathbb{Z}_p$, the correct value for "Signum(12)". Since the accumulated values discussed here are referring to pre-activations, it is necessary to find a way to map all pre-activations to the correct regions in $\mathbb{Z}$. The following section details a novel method using regularization that *teaches* the parameters of the network to map pre-activations to correct regions of the domain.

It is important to clarify that the observations made above are only accurate for when the pre-activations are fed into a Signum activation function. Functions that assign different values to each input such as `ReLU` and `Sigmoid` might not have the same attributes. For instance, given a modulus of $p = 2^4 = 16$, $\text{ReLU}(20) = 20 \neq 4 = \text{ReLU}(\text{signed}(20, p))$. Clearly, correct regions in the domain of the `ReLU` function yield the same sign but different values, which could cause large values in $\mathbb{Z}$, such as "20", to be smaller in $\mathbb{Z}_p$ than other values in $\mathbb{Z}$, such as "7" (also "7" in $\mathbb{Z}_p$). This could cause the parameters of the model to weigh certain values incorrectly and degrade the accuracy of the model.

3.2 Novel Regularization Technique

Regularization in deep learning involves adjusting the values of the parameters to minimize an additional objective [43]. There are various types of regularization. For instance, L1/L2 regularization were introduced to reduce the level of overfitting during the training process. It has been observed that noisier parameters tend to generalize more poorly than parameters that are numerically closer together [43]. For this reason, L1/L2 regularization methods seek to minimize the magnitude of the parameters by introducing an additional loss to the cost function (equations F.1 and F.2 in appendix F). If $\mathcal{L}_{CE}$ refers to cross-entropy loss, the primary minimization objective of a classification algorithm, then adding regularization loss $\mathcal{L}_{REG}$ to the objective yields a total loss of $\mathcal{L} = \mathcal{L}_{CE} + \mathcal{L}_{REG}$. Within $\mathcal{L}_{REG}$, multiple regularization losses can be added together, including L1 and L2 losses of each parameter matrix.

Another type of regularization that can be added is activity regularization, as touched upon in section 2.1. Activity regularization is applied to the activations or pre-activations of a layer. For instance, L1 and L2 losses can be applied to activations to reduce the size of the activation distribution and center it around zero. A great example of another application of activity regularization is presented by *Ding et al.* [31]. They define three different problems with the distributions of pre-activations when training binarized neural networks which they solve using activity regularization over pre-activations. Inspired by this work, this thesis proposes a novel regularizer over pre-activations that solves the accumulation problem of low precision neural networks defined in the previous section.

The following set of equations define the novel **Overflow-Aware Regularizer (OAR)** for one pre-activation input $x \in \mathbb{Z}$ and a modulus $k = 2^\omega$,

$$\mathrm{OAR}_1(x, k) = \mathrm{ReLU}\left(1 - \frac{4}{k}\left|\left[|x| - \frac{k-2}{4}\right]_{\mathrm{mod}\ k} - \frac{k}{2}\right|\right) \tag{3.2}$$

$$\mathrm{OAR}_2(x, k) = \mathrm{OAR}_1^2(x, k) \tag{3.3}$$

To extend this to a loss function over pre-activation vector $\boldsymbol{x} \in \mathbb{Z}^n$, the minimization objective for a loss function from the OAR regularizer can be defined as,

$$\min_{\vartheta}\left[\mathcal{L}_{OAR}(\boldsymbol{x}, k) = \sum_{i=0}^{n-1} \mathrm{OAR}(x_i, k)\right] \tag{3.4}$$

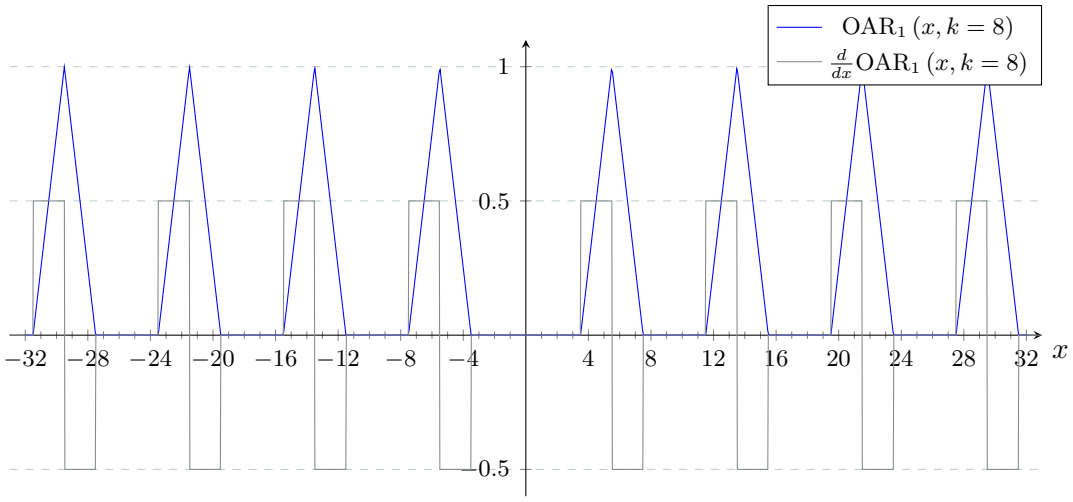
where ϑ is the set of all parameters in the model, and OAR can be either equation 3.2 or 3.3.

As discussed in section 3.1, the task of this regularizer is to push pre-activations in the incorrect regions to the correct regions in $\mathbb{Z}$. To achieve this, the OAR regularizer penalizes any value in the incorrect regions by assigning a number, acting as a penalty, to each value. OAR assigns "0" to each value in the correct regions so that they are not penalized. Since

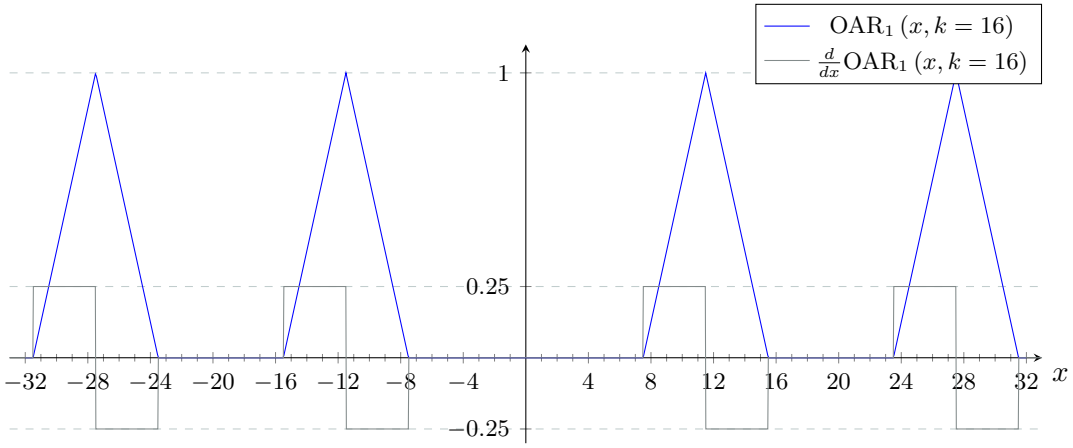
every value in an incorrect region needs to overflow once in either the region to its left or its right, the OAR regularizer has to push weights to the right as well as push them to the left. This means that the regularizer must also inhibit negative gradients, a result of the subtraction of scaled gradients in the update rule in stochastic gradient descent. Observe that the number of values in each incorrect region is always even since the modulus defined is always even. Thus, it is practical to push the first half of the values in each incorrect region towards the correct region immediately before, and the second half of the values to the correct region immediately after. Drawing inspiration from the effectiveness of L1 and L2 regularization, which has been well-established, the penalty should be proportional to the absolute value of the distance between the signed incorrect value and its closest correct value. This translation can be seen inside the outer absolute value operation in equation 3.2. The derivative would therefore be positive for half of the values in the incorrect region and negative for the other half. As a result, the graph of the regularizer, shown in figure 3.3, is a series of *hat* functions where in each incorrect region, half of the values are on the upward line, and the second half of the values are on the downward line.

In figure 3.3, two graphs are illustrated. The first (figure 3.3a), shows the OAR_1 regularizer for a modulus $k = 2^3 = 8$. Each incorrect region contains a *hat* function with a maximum value of "1" at its tip. The derivative function is also shown, where it is "+0.5" for values that should be pushed to the correct region immediately left, and "-0.5" for values that should be pushed right. The values in the correct regions have a penalty of "0" and thus, a derivative of "0", stopping any unnecessary parameter updates. The second graph (figure 3.3b), shows the same regularizer for a modulus of $k = 2^4 = 16$. It is important to notice that the magnitude of the gradient in the second figure is less than in the first figure. This is due to the scaling by $\frac{1}{k}$ in equation 3.2. Ensuring that magnitudes of the penalties remain bounded lessens the probability that the loss function has a large value. If every value within the incorrect regions is assigned a penalty greater than or equal to "1", then for a large modulus, a very large number of values would increase the loss substantially. Also, applying a learning rate to the regularizer for one modulus can be similarly applied to another modulus, due to the consistency in magnitudes of OAR regularizers with different moduli. However, the impact of the scaling factor can be adjusted for each model according to its needs using the learning rate applied to the regularizer.

The OAR regularizer is applicable to any modulus $k \geq 2$ (since 1-bit is the minimum quantization level), meaning it can generalize to any precision and thus, any integer quantization level. Another important feature is the relatively low amount of strain applied to the model as a result of the low amount of movement each value in the incorrect regions needs to undergo to fall within the correct regions. In this context, *strain* is defined as the effort a model exerts when moving parameters towards an objective. The regularizer ensures that each incorrect value must move a maximum of $\frac{k}{2}$ spots to the left or the right. Alternatively, applying L1 or L2 regularization effectively makes the values move an unbounded number



(a) $k = 8$



(b) $k = 16$

Figure 3.3: **Overflow-Aware Regularizer**. An illustration of the L1 Overflow-Aware Regularizer and its derivative for two different moduli ($k = 8$ and $k = 16$).

of spaces to the left or right, since the learning objective compels values to move as close to zero as possible. The strain on the model to balance accuracy with moving pre-activations is thus much less when using OAR than it is when using L1/L2 regularization. To retain the model's optimal convergence, the model parameters should be changed as minimally as possible, making this a very important feature. The graph of OAR_2 is very similar to the graph of OAR_1. The *hat* functions become parabolic and as a result, the derivatives become linear but with the same sign as in OAR_1. This adds a higher, relative penalty to values closer to the middle of the incorrect regions. This teaches the values in the center to move closer to the correct regions *faster*, since the derivative is not constant anymore. In the following region, we experiment with both OAR_1 and OAR_2, as well as the comparable effectiveness of basic L2 activity regularization.

In order to measure the effectiveness of the OAR regularizer with respect to the number

of values in correct regions, we define the OAR metric. The main part of the OAR metric is a step function with period k and 50% duty cycle, translated by "0.5" to the left, and mirrored across the y-axis. This function maps values in incorrect regions to "1", while values in the correct regions to "0". By summing across the output, the number of values in incorrect regions is calculated. As a result, the percentage of values in correct regions can be extracted with ease, which is the definition of the OAR metric. The modified step function described in this paragraph is illustrated in figure F.2 for a modulus of $k = 16$.

In the following two sections, we discuss our implementation of the OAR regularizer and some experiments surrounding its effectiveness for our task. Our experiments reveal some optimizations and settings that are necessary for the OAR regularizer to be effective. In general, our results appear to be positive. To the best of our knowledge, this type of regularizer also appears to be the first in the published literature in the area of quantization.

3.3 Implementation

3.3.1 MNIST RNN Architecture

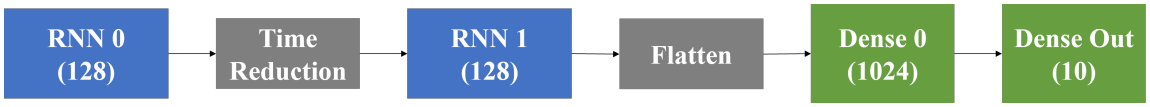


Figure 3.4: **The MNIST RNN**. This diagram shows the architecture of the MNIST RNN. The numbers in the parenthesis of certain layers indicate the number of units in the layer. The layers shaded in grey reshape the intermediate activation matrices and do not include any parameters. The numbers after the names of the layers indicate the index of the specific layer. For example, RNN 0 is the first of two RNNs.

We evaluate the OAR regularizer over the MNIST dataset [66]. For more information on the MNIST dataset, refer to section 2.1.2. We utilize a toy RNN model to stay within the scope of this thesis. To the best of our knowledge, for performing digit classification over MNIST, the model we use is novel. In the MNIST dataset, each image is 28×28 pixels large. For each image, we extract the 28 rows of the image, consisting of 28 pixels each, and use each row as an input to the RNN at each timestep. Figure 3.4 shows our model architecture. We refer to the model as the *MNIST RNN* from this point forward. The first layer is a basic, vanilla RNN with 128 hidden units. The time reduction layer downsamples the outputs of the first RNN by a factor of two. The layer stacks every second output vector on the previous output vector, thus reducing the number of timesteps from 28 to 14. We draw inspiration for this layer and its position between the first stacked RNNs in the network from [50], where this technique works well for speech recognition and more complex LSTM networks. The third layer is another vanilla RNN layer with 128 units. The purpose of stacking two RNNs is to generate internal representations of the sequence at both a recurrent and feed-forward level. From section 2.2.1, we know that each RNN contains two parameter matrices, one

that learns from the input features, and another that learns from the state features. Since handwriting is a sequential process, modeling this problem with RNNs has the potential to produce great results [44]. The next layer flattens the outputs (which is a 14×128 matrix) into a 1792-unit vector. The last two layers are dense layers, or linear layers, which are simple, feed-forward networks. The activation function for each layer is the `tanh` function, a choice made to support a change to the *Signum* function during quantization. In total, this model contains 1,914,368 parameters. However, there are no bias parameters in the model. The decision to exclude bias vectors comes from the ineffectiveness of quantizing them using ternarization or binarization. Ternarizing a bias vector sets it to an element in $\{-1, 0, 1\}$. The goal of a bias vector is to translate the pre-activation distribution of neural network layers. Pre-activation distributions with integer values are much larger than "-1", "0", or "1" and thus, a small bias vector would not translate the pre-activation distribution nearly enough as it would in the original distribution. At the output stage of the model, the output of the MNIST RNN is fed into a `softmax` for classification.

The MNIST dataset is split into 60,000 training images and 10,000 test images. We adopt the categorical cross-entropy loss function for the main minimization objective and utilize the Adam optimizer. We perform simple input processing for each image. First, we normalize each image by dividing each 8-bit pixel value by 255, the maximum value of 8-bit unsigned integers. To improve generalization, we randomly flip some images horizontally and add random brightness over random images in the dataset. The training and validation sets are shuffled before each epoch. We use TensorFlow [1] for all ML training and evaluation.

The model is trained in 32-bit floating-point precision. We set the batch size to 512, the learning rate to $1e^{-4}$, and apply L2 regularization to all parameters, including the recurrent parameters, with a regularization rate of $1e^{-4}$. We obtain a test accuracy of 98.13% using the MNIST RNN. The training and validation curves are presented in figure F.5. The validation curve achieves 99% accuracy while the test accuracy is around 1% lower. Nevertheless, it serves as a great toy model for quantization and CGGI conversion experimentation.

The OAR regularizer is applied to the model, along with quantization, concurrently. The method for quantizing is presented in chapter 4 as it is more relevant to the CGGI conversion process. In this chapter, we detail the OAR regularizer and present our results regarding its effectiveness only. As such, we assume that our model has gone through a few steps of quantization first and reached a point where we can apply OAR regularization. To summarize these steps, we first train the full precision model. The second step involves changing every activation function to the Signum function and retraining. In the third step, we ternarize the inputs of the model and retrain once more. Finally, we ternarize the parameters of the model, switch the activations to the ModSignum activation function defined in the next section, apply OAR regularization, and retrain the model, all in the

final step. This novel, 4-step quantization process is detailed in chapter 4.

3.3.2 Signum Activation Function Over A Modulus

As detailed in chapter 4, the MNIST RNN model is quantized with ternary weights and binary activations. The activations are binarized by applying the Signum function with a `tanh` derivative during backpropagation. Thus, the pre-activations in each layer are integers in $\mathbb{Z}$ and are ready to be moved to the correct accumulation regions for integer arithmetic in $\mathbb{Z}_p$ (for $p = 2^\omega$ where ω is the bit-width of the accumulated integers). We create a custom regularization function in TensorFlow that computes the loss function defined in equation 3.4. In order to mimic the CGGI evaluation environment, we change the evaluation of the Signum function in TensorFlow to first execute the signed modulo operation of the pre-activations with a set modulus p, and then apply the original Signum function. This ensures that the pre-activations are in $\mathbb{Z}_p$ rather than $\mathbb{Z}$, and the Signum function is executed in $\mathbb{Z}_p$. This allows us to mimic the arithmetic of the encrypted model in the plaintext model and calculate relevant metrics. It also allows the network to adjust to possible incorrect values resulting from signed, low-bit integer arithmetic, which is the goal of quantization-aware training. Equation 3.5 shows the modified Signum function for pre-activation $x \in \mathbb{Z}$ and a modulus of $k = 2^\omega$.

$$\text{ModSignum}(x, k) = \text{Signum}\left(\text{signed}(x, k)\right) \tag{3.5}$$

To apply the OAR regularizer, we follow the same method used to apply general activity regularization—in each layer, we calculate the OAR loss from the OAR regularizer with respect to the pre-activations and add it to the total loss of the model. We change every Signum activation function to the ModSignum activation function, defined in equation 3.5. In the following section, we present our results with respect to various experiments.

3.4 Results and Discussion

For all of the experiments in this section, we train the network using TensorFlow and one Nvidia RTX2080Ti GPU. We use Adam as the optimizer with its default settings. The learning rate for all the experiments is the output of a Cosine learning rate scheduler, starting at $1e^{-4}$, and decreasing towards $1e^{-5}$ according to the Cosine function for 100 epochs. The batch-size used is 512, the *temperature scale* for each RNN is 4, and the *threshold scale* for each layer is 1.5. With respect to the meaning of *temperature scale* and *threshold scale*, please refer to sections 4.1.2 and 4.1.1 respectively. Since these parameters are not important to this chapter for reasons other than experimental reproducibility, and since the experiments cannot be reproduced without the quantization method discussed in chapter 4, they are not discussed here. There is no regularization other than OAR applied

in the model, unless stated otherwise. The regularization methods tested in this section are applied to all layers of the model except for the last layer. In section 4.2, we discuss a method to generate multiple ciphertexts from the last layer, allowing us to forgo the use of regularization.

3.4.1 MNIST RNN Accuracy With And Without OAR Regularization

In this experiment, we analyze the general effectiveness of the OAR regularizer on the MNIST RNN in figure 3.4. In table 3.1, the results of both applying the OAR regularizer and not applying it on the accuracy of the RNN network is shown for different bit-widths (ω). We test bit-widths up to 8-bit, since it is the practical limit for CGGI operations. For the results in the table, we used the ModSignum activation function, defined in section 3.3 to mimic the results obtained from running the MNIST RNN over encrypted data with CGGI. For each run, we use $1e^{-3}$ for the regularization rate. We train each model for 1000 epochs and report the final results. The OAR regularizer we use is the OAR_2 regularizer, since in appendix F.1, we see that it outperforms OAR_1.

Bit-Width	Accuracy Without OAR_2	Accuracy With OAR_2	Percent Difference
8	95.30%	95.19%	-0.11%
7	95.15%	93.97%	-1.18%
6	46.82%	89.35%	**+42.53%**
5	9.88%	80.73%	**+70.85%**
4	11.13%	11.04%	-0.09%
3	11.14%	11.16%	+0.02%

Table 3.1: **MNIST RNN accuracy with and without OAR_2 for different bit-widths**.

The far-right column in the table displays the difference in test accuracy between runs with OAR and runs without. From the table, we can see that for bit-widths of "7" and "8", there is little difference between runs with OAR and without. From analyzing the OAR metric values for these runs, the runs without OAR have good OAR accuracy, suggesting that 8-bit or 7-bit integers already cover the domain of the pre-activations. However, for 5-bit and 6-bit integers, the runs without OAR display an OAR metric accuracy of around 50% and 70% respectively. 50% is the lower bound since from section 3.1, we know that half of the domain already lies within the correct regions of $\mathbb{Z}_p$ without any correction. This shows that the domain extends past 5-bit integers for sure. For 6-bit integers, the accuracy increases to around 47% without the OAR regularizer which makes sense since 70% of the values are already in the correct regions. For both 5-bit and 6-bit, the OAR regularizer is crucial to obtaining high accuracy, with a +71% and +43% difference in accuracy when it

is used. This clearly shows that the OAR regularizer works and should be used to obtain high accuracy in lower bit-widths. For bit-widths lower than 4-bit, both runs with and without the OAR regularizer fail, achieving accuracy no better than random. This shows us that the OAR regularizer, while it works well for certain bit-widths, has its limits. A reason for its failure in lower bit-widths could be that the intended domain becomes too granular. DNNs are stochastic processes and thus, allowing larger regions for pre-activation distributions gives the network some leniency in generating pre-activations with noise. For these tests, we used a regularization rate of $1e^{-3}$. In the next section, we test how accuracy is impacted by different rates.

3.4.2 MNIST RNN Accuracy For Different OAR Regularization Rates

In this experiment, we test several different rates of regularization for the OAR_2 regularizer, for the two bit-widths that were found to work well in the previous section. Table 3.2 displays the accuracy of the MNIST RNN as it responds to changes in the OAR regularization rate.

	Test Accuracy	
Regularization Rate	5-bit	6-bit
0	9.88%	46.82%
$1e^{-6}$	12.30%	70.33%
$5e^{-6}$	11.84%	86.34%
$1e^{-5}$	19.77%	87.58%
$5e^{-5}$	75.28%	90.73%
$\mathbf{1e^{-4}}$	**81.11%**	**92.11%**
$5e^{-4}$	79.44%	91.76%
$1e^{-3}$	80.73%	89.35%
$5e^{-3}$	76.21%	86.01%
$1e^{-2}$	76.79%	83.92%

Table 3.2: **MNIST RNN accuracy using OAR_2 for different regularization rates and bit-widths**.

Acting as a control, the first row shows the accuracy of the model without OAR regularization. According to the chart, the general observation is that $1e^{-4}$ provides the best accuracy in both bit-widths. The 5-bit setting appears to be more sensitive to changes in the rate. For instance, between the $1e^{-5}$ and $5e^{-5}$ rates, the change in accuracy is over 50%, which is more than twice the largest change in accuracy in the 6-bit setting. This is expected since the more granular the intended domain, the likelier it is that noisy values could change regions from correct to incorrect and vice-versa, displaying a higher sensitivity

to regularization. This intuition stems from the fact that a more granular domain, with respect to the frequency of correct and incorrect regions, contains shorter distances between regions than a less granular region. Another symptom of this phenomenon is the relatively large accuracy difference between 6-bit and 5-bit settings for a regularization rate of $1e^{-4}$. The 6-bit setting obtains an accuracy of around 11% higher than the 5-bit setting. In fact, the 6-bit setting is consistently higher than the 5-bit in terms of accuracy for each rate. Another important observation is the change in accuracy for both bit-widths as the rate changes. As the rate increases, the accuracy increases. However, once the rate reaches a specific value, in this case $1e^{-4}$, further increases to the rate cause a linear decrease in accuracy. Thus, the rate is an important hyperparameter, as well as the bit-width, when applying OAR regularization.

3.4.3 A Visual Confirmation Of Effectiveness

For each of the experiments in this section, we logged the value of the pre-activations in each layer. While each layer exhibited the effects of OAR regularization, the Dense 0 layer, since it contains the largest number of parameters and accumulations, was the best layer to observe for signs of the effectiveness of OAR regularization. For each experiment, we observed the pre-activation distribution of Dense 0 and expected to see a distribution with several pre-activations in the correct regions and little in the incorrect regions. Pictorially, this can be imagined as a distribution with holes in the incorrect regions and spikes in the correct regions. In figure F.4, located in appendix F, it can be seen that pre-activations indeed do move to the correct regions.

Figure F.4a shows the distribution of the 5-bit setting and figure F.4b, the distribution of the 6-bit setting. These graphs were recorded for the experiments with regularization rate equal to $1e^{-4}$ in table 3.2. The graphs are annotated with grey lines over-top of the distribution, designating the regions of accumulation between the lines. A green checkmark indicates that the region is a correct accumulation region. The red "x" symbol indicates that the region is an incorrect region. It is evident that in both graphs, the distributions are divided as intended. Large amounts of pre-activations are present in correct regions while small amounts of values are present in incorrect regions. It is also evident that the regions are much more granular in the 5-bit setting. The distance between the same types of regions (between correct regions, for example) is smaller in the 5-bit setting than in the 6-bit setting. As a result, the incorrect regions show some spikes, or the rising and falling edges of some spikes. However, the regions are still clearly distinguished and the spikes in the correct regions are relatively much larger. This is confirmation of the phenomenon discussed in section 3.4.2. Since the 6-bit graph shows clearer, more well-defined flat regions and spikes, it is safe to assume that noisy regions could contribute to the 11% loss of accuracy between the 6-bit and 5-bit experiments.

The OAR metric confirms the results displayed in figures F.4a and F.4b in appendix F.

The OAR metric for these experiments is shown in figure F.3. The curves for the Dense 0 and RNN 1 layers are shown for both 5-bit and 6-bit experiments. The graph shows the metric starting from around 65% and increasing towards 100% for the Dense 0 layer, while the RNN 1 layer starts at a higher accuracy, also increasing towards 100%. In contrast, figure F.3 also shows the OAR metric for the 5-bit and 6-bit runs without OAR regularization. In these settings, the ModSignum function is used without OAR regularization. The curves without OAR do not increase and hover around the same values. This difference in OAR metric values shows that OAR regularization is necessary to move the pre-activations to the correct regions. The visual representations of pre-activation distributions along with the OAR metric graphs clearly show the positive impact of OAR regularization on running RNNs in low-bit precision settings. For a detailed analysis on the effectiveness of the different types of activity regularization techniques mentioned in this thesis, including OAR_1 versus OAR_2, refer to appendix F.1.

3.4.4 OAR For Longer RNNs

In these experiments, we resized the input images from 28×28 to 128×128 pixels using `tf.image.resize` from the TensorFlow images library. In return, this increased the number of parameters to 8,480,768 and the number of timesteps from 28 to 128. The purpose of these experiments is to evaluate how OAR regularization performs with longer sequences. Given the structure of the MNIST RNN, a longer sequence creates a larger input for the Dense 0 layer. Rather than 14×128, the number of inputs increases to 64×128, a nearly 5x increase to an already large amount of parameters. This is important since it increases the size of the domain of the pre-activation distribution in Dense 0. In conjunction, this experiment also evaluates the performance of OAR regularization for larger networks.

Using the same quantization method from section 4.1, a Cosine decay learning rate of $5e^{-7}$, and OAR_2 regularization with rate $1e^{-4}$, for a bit-width of 6-bits we achieve a test accuracy of 92.69%. This shows that longer networks can be retrained successfully using OAR regularization. An interesting observation is that the OAR metric for the Dense 0 layer achieves only 70.57%. This suggests that larger networks might not need to move every value to the correct regions in order to obtain good accuracy. It is possible that 30% of the pre-activations, in this case, are not as important to the learning objective as the other 70%. Larger networks with more parameters have a larger capacity to learn, and as a result, a higher chance of overfitting. Overfitting is mitigated in our case due to the inherent regularization caused by quantization itself. The quantization strategy of ternarization can be seen as a form of dropout which has proven to be effective against overfitting. However, ternarization limits the expression of each neuron since the precision is decreased significantly. Introducing more neurons increases the number of degrees of freedom in the layer and thus, regains some of the representational power lost through low-precision quantization [87]. Thus, with an abundance of parameters, it is possible that

all of them are not equally as effective with respect to the learning objective. The key observation here is that high accuracy can be achieved with levels of the OAR metric less than 100%, which shows the importance of training with the ModSignum function as well as OAR regularization.

3.5 Summary

This chapter introduced a novel regularization method to mitigate the overflow accumulation problem. When accumulating integers in $\mathbb{Z}_p$, for a power-of-two modulus $p = 2^\omega$, it is likely that values will overflow. Once they overflow, their sign could flip, causing the Signum function to output the incorrect value. Harnessing the power of overflow, section 3.1 identified regions of overflow in $\mathbb{Z}$ that output the correct sign, and thus, section 3.2 proposed a way to overflow values in incorrect regions into correct regions to output the correct sign. It is important to output the correct sign in discretized neural networks since they rely on the Signum function as the main activation function for each layer. Since accumulation occurs in deep neural network layers when calculating pre-activations, section 3.2 proposed the Overflow-Aware Regularizer (OAR), a form of activity regularization applied to pre-activation distributions. OAR assigns a penalty to values in incorrect regions, which is used by the learning algorithm to move the values to correct regions in response, while maintaining accuracy.

In section 3.3, we introduced the MNIST RNN architecture, a novel deep neural network with around two million parameters, to be evaluated over the MNIST dataset. To mimic the modular arithmetic used in CGGI-compatible DNNs, and allow the network to observe changes in distributions as a result, we also introduced the ModSignum activation function. Combining OAR regularization and the ModSignum function generated promising results.

In section 3.4.1, our experiments revealed that the OAR regularizer does move pre-activations into correct regions for bit-widths of 5-bit and up, with acceptable accuracy. For instance, we achieve around 93% accuracy using 6-bit OAR regularization and an OAR metric of around 100%, showing that the values are in the correct regions. These results show an accuracy degradation of only 2% with respect to a the same model utilizing the Signum activation function without modular arithmetic. In section 3.4.2, we discussed the different settings that can be chosen in OAR regularization and their impact on the final test accuracy. The results demonstrated that certain rates of regularization are better than others, making it a suitable hyperparameter for training. Section 3.4.3 confirmed the effectiveness of OAR in moving values to correct regions visually. We showed several distributions as histograms and observed that the number of values spikes in the correct regions and declines to almost zero in the incorrect regions for different bit-widths. We confirmed that OAR causes this phenomenon by observing the OAR metric results and obtaining almost 100% OAR accuracy in each experiment that utilized OAR. In the exper-

iments without OAR, both MNIST test accuracy and OAR accuracy did not improve, with OAR accuracy hovering close to 50%. In section 3.4.4, we concluded the results section by observing that OAR regularization also works for longer, and larger, RNNs, and that a good OAR accuracy does not necessarily always result in a good test accuracy. In fact, training with ModSignum is equally as important as training with OAR regularization.

In the next chapter, we build and evaluate large-scale, privacy-preserving recurrent neural networks with CGGI. We propose a unique quantization strategy to quantize RNNs and define new methods to convert RNNs to CGGI-compatible RNNs. OAR regularization plays an important role in the next chapter, allowing us to use smaller CGGI parameters and create both faster and more accurate RNNs as a result.

Chapter 4

Recurrent Neural Networks Over Encrypted Data

In this chapter, we make several contributions to the field of executing RNNs, including RNNs with attention, over encrypted data and demonstrate the effectiveness of our methods. Evaluating recurrent neural networks over encrypted data is a difficult task since RNNs can be of variable depth and cannot be easily quantized to operate under modular, signed integer arithmetic. To mitigate the problem of depth, this thesis utilizes the CGGI [25] scheme, a fully homomorphic encryption scheme with very efficient bootstrapping (detailed in section 2.4) and the ability to evaluate non-linear functions without approximation through the use of lookup tables. While CGGI offers these resourceful abilities, it is restricted to operations under modular, signed integer arithmetic, making quantization of RNNs necessary. Quantization of RNNs resulting in operation under this type of arithmetic is difficult and has not been previously published, as detailed in section 2.5.5. As a result, the literature surrounding RNN evaluation over encrypted data using CGGI is limited. The research that exists proposes inefficient methods, as discussed in section 2.6.3.

In response, section 4.1 proposes a novel, low-precision quantization method that quantizes RNNs through activation binarization and parameter ternarization. The resulting operations occur under modular, signed integer arithmetic, enabling efficient evaluation of large-scale RNNs over encrypted data. Section 4.2 proposes new methods compatible with CGGI that are integral to the conversion of quantized RNNs to CGGI-compatible RNNs, including those that implement attention. Section 4.3 introduces a novel, 12 million parameter RNN architecture with attention for speaker identification and discusses our implementation of both the MNIST RNN and this RNN using CGGI operations over encrypted data. The final sections of this chapter demonstrate and analyze our results.

4.1 A Method To Quantize Recurrent Neural Networks

As detailed in section 2.5.5, a successful low-precision quantization method that results in modular, signed integer arithmetic without bit-shifting and full-precision accumulation has not been proposed in the literature. In this section, we discuss our novel quantization method which is structured as a four-step process. The process is briefly mentioned in section 3.4 and detailed in section 4.1.3. Sections 4.1.1 and 4.1.2 introduce methods that are crucial to the performance of the four-step quantization process.

We strive to extend and refine the methods in FHE-DiNN [13] since their methods work well for evaluating regular, feed-forward neural networks over encrypted data, both efficiently and accurately. As a result, we define a new way of quantizing vanilla RNNs. In general, our process results in an RNN that does not contain any bit-shifting/division operations. The RNN performs matrix multiplication accumulations into low-bit, integer values. The parameters of the RNN are ternarized, with values in $\{-1, 0, +1\}$, and the activations are binarized, with values in $\{-1, +1\}$. The inputs to the RNN are also ternarized to keep the distribution similar to the binarized recurrent state vectors. Adding OAR regularization, as defined in chapter 3, allows the quantized RNN to safely overflow into correct accumulation regions, and choose a lower bit-width for accumulation as a result.

4.1.1 Generalization Of The Ternarization Threshold

We begin to define our quantization process by defining the quantization of the parameters of the network. In section 2.5.5, we reviewed the literature surrounding RNN quantization. We discovered that some works prove binarization is impossible with respect to quantizing RNN parameters [57]. We also observed that ternarization has been successful for quantizing parameters of RNNs [56]. For this reason, we decide to use ternarization for quantization of parameters in our RNN quantization method. As a consequence of ternarization, all scalar multiplications in the matrix multiplication operation between parameters and activations to can be changed to simple additions and subtractions. This is very beneficial since plaintext-ciphertext addition in CGGI is less expensive than scalar multiplication in regard to both depleted noise budget and efficiency. The application of ternarization is where we digress from FHE-DiNN. In FHE-DiNN, the parameters are quantized to larger precisions when in fact, for RNNs, we find that it is not necessary and that ternarization suffices.

The ternarization method we adopt is the deterministic method from TernaryConnect [76], as shown in equation 2.28. For the threshold value, we use a variant of the setting in TWN [72], the original of which is shown in equation 2.29. If we were to use the threshold defined in TWN, we would be using their exact method of quantization, with the exception of scaling the ternary domain by a fixed-point scalar α. Instead, through experimentation, we find that equation 2.29 should include a hyperparameter, which we notate as g. The hyperparameter $g \in \mathbb{R}_{>0}$, which we refer to as the **threshold scale**, controls the size of the

threshold linearly. We can see this in the following equation, representing our calculation of the new threshold:

$$\delta = g \cdot \mathbb{E}\left(|\vartheta_l|\right) \tag{4.1}$$

where ϑ_l is the set of all parameter matrices in the layer l. The hyperparameter g controls the size of the threshold by scaling the expected value of the absolute value of all the parameters in a layer. It is also very important to observe that the quantization granularity set by our ternarization method is at the layer level. Through experimentation, we find that this type of granularity is more effective in quantizing the network than calculating a separate threshold for each parameter matrix. Effectively, our method generalizes the threshold set in TWN, where a value of $g = 0.7$ reduces to the original threshold proposed in TWN.

4.1.2 Gradient Scaling

Ternarizing all of the parameters at once, and thus changing the pre-activations to integers, causes exploding gradients in vanilla RNNs, a phenomenon we observed during experimentation. In many scenarios, the model would begin to train and after a few training steps, would display `NaN` values for every parameter and metric, indicative of exploding or vanishing gradients. Upon investigation of the gradient norms, we noticed that they are much larger than the norms in floating-point runs, compelling us to conclude that the gradients are extremely large. Other researchers have also observed this problem, for example [57], during RNN quantization. The quantized RNN does not train as a result.

Exploding gradients are generally mitigated by clipping the norms of the gradients, using normalization, or switching to gated RNNs, as discussed in section 2.2.1. Section 2.5.5 also highlights several papers that use normalization on top of quantized parameters to address this problem, some even claiming that mathematically, RNNs cannot be quantized into low-bit precision without normalization [122]. Analyzing the mathematics of normalization, the equation reduces to a scaling and translation operation over the activations in the layer it is being applied—the activations are translated by a scaled version of the mean specific to the type of normalization (e.g. batch [59] or layer [6] normalization), plus a learned bias parameter, and they are also scaled by a learned parameter as well as the reciprocal of the standard deviation, specific to the type of normalization. With respect to gradients, scaling the activations in the forward step also scales the gradients in the backward step during the training process, due to the rules of derivation in Calculus. Scaling exploding gradients could explain why normalization during quantization helps to stabilize the training of quantized RNNs, however, this is only a hypothesis and a rigorous analysis is left for future work. Gradient clipping scales gradients down as well but loses potentially valuable information by mapping values greater than a specific clipping level to the same number. It would be

ideal to scale the gradients without mapping them to a single value, and without scaling the forward distribution since fixed-point is not supported by CGGI.

In response, we introduce a new hyperparameter for each layer called the **temperature scale**. We notate the temperature scale by the variable s. To scale the gradients flowing through a DNN layer and leave the forward distribution untouched, we divide the pre-activation gradients by s during training while not changing the pre-activations themselves. This is similar to the method used by binarization papers when binarizing activations (as discussed with references in section 2.5.3). When quantizing activations, it is common practice to use one function to calculate the activation in the forward step, and replace the function with another in the backward step [114]. To be more specific, frameworks such as TensorFlow [1] build a computational graph of operations in the forward step. When performing backpropagation, the graph is traversed in the reverse direction. By replacing a part of the graph with another function, while you are not calculating that function in the forward step, the backward step traverses the reverse of the newly replaced function. Therefore, using each s_l value, where l is the index of the layer, we multiply the gradient flowing backwards through the pre-activations of layer l by $\frac{1}{s_l}$.

The temperature scale can technically be any floating-point, real value, since the parameters, gradients, and gradient updates are in floating-point during training ($s \in \mathbb{R}$). Thus, for $s \in (0, 1)$, the gradients are scaled to be larger due to the fraction $\frac{1}{s}$. This could help in vanishing gradient scenarios where the gradients should be increased to maintain stability. For $s = 1$, there is no scaling of the gradients. This is the standard setting for training DNNs. For $s > 1$, the gradients are scaled to be smaller. This is the setting that should be used to mitigate exploding gradients. It is important to note that as s becomes much larger than 1 or very close to 0, exploding gradients could become vanishing gradients, and vanishing gradients could become exploding gradients respectively, causing instability in the training process. Thus, similar to other hyperparameters, a search for the best values should be undertaken and the gradient norms analyzed in the process to determine when and if these problems are being mitigated.

4.1.3 The Four-Step Quantization Process

In this section, we introduce the four-step quantization process of successfully quantizing vanilla RNNs to low-bit precision. We use the methods described in the previous section as well as other methods from previous chapters to enable this process.

Step One: Train (Wider) Full-Precision RNN

The first step of the process is to train a full-precision, vanilla RNN. Our method quantizes on top of an already trained RNN, that is comprised of parameters, gradients, and activations in floating-point precision. RNNs could already be pretrained, in which case we may need to perform some retraining, or they could be trained from scratch. The important

things to note are that the activation functions in the RNN should be set to the `tanh` function. If the model is already pretrained, then the activations should be changed to `tanh` and the model, retrained. We recommend that the model is trained from scratch with `tanh`. The hyperbolic tangent function is one of the closest continuous functions to the Signum function. Since our quantization method requires use of the Signum function, training using `tanh` helps structure the internal representations around this particular non-linearity and thus, once the Signum function is introduced, the model adjusts more easily.

If the model is considerably deep and combines other layers, it may be necessary to widen the layers. From experimentation, we noticed that smaller networks that performed well in floating-point were not quantizing effectively without widening the layers. This intuition stems from WPRN [87]. When we discussed WPRN in section 2.5.3, we highlighted a 12% increase in accuracy when tripling the number of parameters in each layer. Thus, if a model fails to quantize at any of the next steps, it is suggested to try and widen the layers if this has not been done already.

To summarize, in this step, if a model is provided, change all the activations to `tanh` and retrain. If a model is not provided, make sure to use `tanh` when training from scratch. If the network does not quantize in the steps that follow, triple the number of parameters in each layer and retrain from scratch.

Step Two: Change Activations To Signum Function With Tanh Derivative

In this step, a model, trained according to step one, already exists. However, all of the parameters and activations are in floating-point. Also, the activation function used for each layer in the model is the `tanh` function. To begin the actual quantization process, we binarize the activations first. This intuition stems from [103] where the authors suggest that quantization of activations first may lead to better results when quantizing deep neural networks. We change the activation functions in the model to the Signum activation function and change the derivative of the Signum function to the derivative of the `tanh` function. Gradients that pass through each activation thus experience the `tanh` function rather than the Signum function, causing a smoother loss landscape. After replacing the activations, we retrain the model after initializing all of the parameters to the parameters in the trained model from step one. Effectively, we fine-tune the model, recovering much of the accuracy lost through the switch to binary activations. At this point, the activations in the model are all either "+1" or "-1", although the parameters and inputs to the model are still in floating-point.

It is possible that at this point, some of the gradients might begin to explode in the RNN layers. If this occurs, it is recommended to first increase the batch size. Increasing the batch size causes smaller and smoother gradient updates since the number of gradients being averaged in the stochastic gradient descent weight update process increases and thus, the scale 1/B multiplied to each gradient multi-sum decreases ("B" denotes batch size).

If increasing the batch size does not solve the problem, it is recommended to apply a temperature scale to the RNN layers, which is discussed in section 4.1.2.

Step Three: Ternarize The Inputs

In this step, we initialize the model from step two with the updated parameters from step two, and ternarize the inputs according to the process detailed in section 4.1.1. In short, calculate the mean of the input matrix, select a threshold scale (for inputs, we have used $g = 0.7$ mainly), apply the ternarization method, and retrain the network once more. This differs from common practice which is to keep the input layer in full precision or larger precision. We assume that the input and recurrent state distributions in the RNN are closer together with ternarized inputs since the recurrent states are binarized. We note that any type of quantization, as long as it is integer and not fixed-point, should be fine (e.g. symmetric 8-bit quantization as in FHE-DiNN [13]). We use ternarization since it matches the distributions in the RNN cell and offers lower, symmetric precision, which is better for CGGI evaluation (lower precision enable faster computation). We also find in our experimentation that ternarization demonstrates good results.

Step Four: Ternarize The Parameters

Similar to the other steps, we begin by initializing the model from the previous step, consisting of the updated parameters, input ternarization, and Signum activations. Next, for each layer, we calculate the mean of the parameters and select a ternarization threshold scale. Using these values, we calculate a ternarization threshold for each layer that remains fixed throughout the retraining process. It is important to note that the threshold is calculated based on the updated parameters from step three. Subsequently, for each RNN layer, we also set a temperature scale to mitigate possible exploding gradients. Finally, we retrain the network with the following addition: at the beginning of each forward step in the retraining process, the parameters are ternarized using the calculated thresholds.

We can better understand this process with an example. Using the MNIST RNN architecture from chapter 3, we complete step three and begin calculating the ternarization thresholds—for the RNN 0 layer, we calculate the mean of all the parameters and set the threshold scale to $g = 1.6$, and we perform the same calculation for each subsequent layer and set the associated threshold scales. At this point, we also add gradient scaling to each RNN layer in order prevent exploding gradients from occurring—for example, RNN 0 and RNN 1 both are set to include gradient scaling with a temperature scale $s = 4$. We begin to retrain the model, ternarizing the weight matrices at the beginning of every forward step. To visualize the ternarization and retraining process, since the thresholds remain the same for the entire training process, the weights are adjusted by floating-point updates to move past the threshold, becoming $+1$ or -1, or inside the threshold, becoming "0". Once

training completes, we can apply ternarization using the same thresholds and extract the quantized weights.

In the process described in the previous paragraph, the Signum function is used for the activation function. The accumulation of pre-activations is in floating-point but the values are all integers (since matrix multiplication occurs between ternarized parameters and binarized activations). To set the precision of the accumulation to $p = w^{\omega}$ where ω is the number of bits, this is where we can apply OAR regularization. In step four, after the thresholds are calculated and the temperature scales are set, OAR regularization is applied to each layer (except the last), the activations are switched to the ModSignum function, and retraining with ternarization of the parameters commences.

Summary

After the four-step process, a vanilla RNN becomes a quantized RNN with ternarized weights, ternarized inputs, and binarized activations. The accumulation can be unbounded or it can be set to a specific bit-width (using OAR regularization to maintain accuracy). In figure 4.1, we summarize the four-step process. The numbers at the corners of the dashed boxes indicate the step number. As shown in the figure, we begin with a vanilla RNN, trained or newly initialized. We feed the RNN into step one. Step one performs widening of the network, if necessary, changes the activations in the RNN to `tanh`, and trains the RNN from scratch if it is newly initialized, or retrains the pretrained RNN. We feed the updated model to the second step. In the second step, we change all the activations to Signum and the derivatives to the derivative of `tanh`. We add a temperature scale to the RNN layers if necessary. We then retrain the model with the new setup. We continue the process and feed the updated RNN to the third step. In this step, we only ternarize the inputs and then retrain the model, to recover the potential accuracy loss from the input ternarization. We conclude the process by feeding the updated RNN into the fourth step. In the fourth step, we set the temperature scale, if we have not already. We then calculate the thresholds for ternarization, which remain fixed throughout the training process. Finally, if we require OAR regularization, we update the model appropriately and retrain while ternarizing the parameters of the network at the beginning of each forward step.

In section 4.3, we revisit the quantization process and discuss it implementation in TensorFlow. In this section, we showed that the quantization process creates an RNN that can have bounded accumulation, low-precision parameters, inputs, and activations, and as a result, no multiplication operations. These are all characteristics that enable us to perform efficient evaluation of large-scale RNNs over encrypted data using CGGI. Before we discuss our implementation of CGGI-compatible RNNs, in the next section, we propose some important operations and techniques that are necessary to convert the types of RNNs we experiment with to CGGI-compatible RNNs.

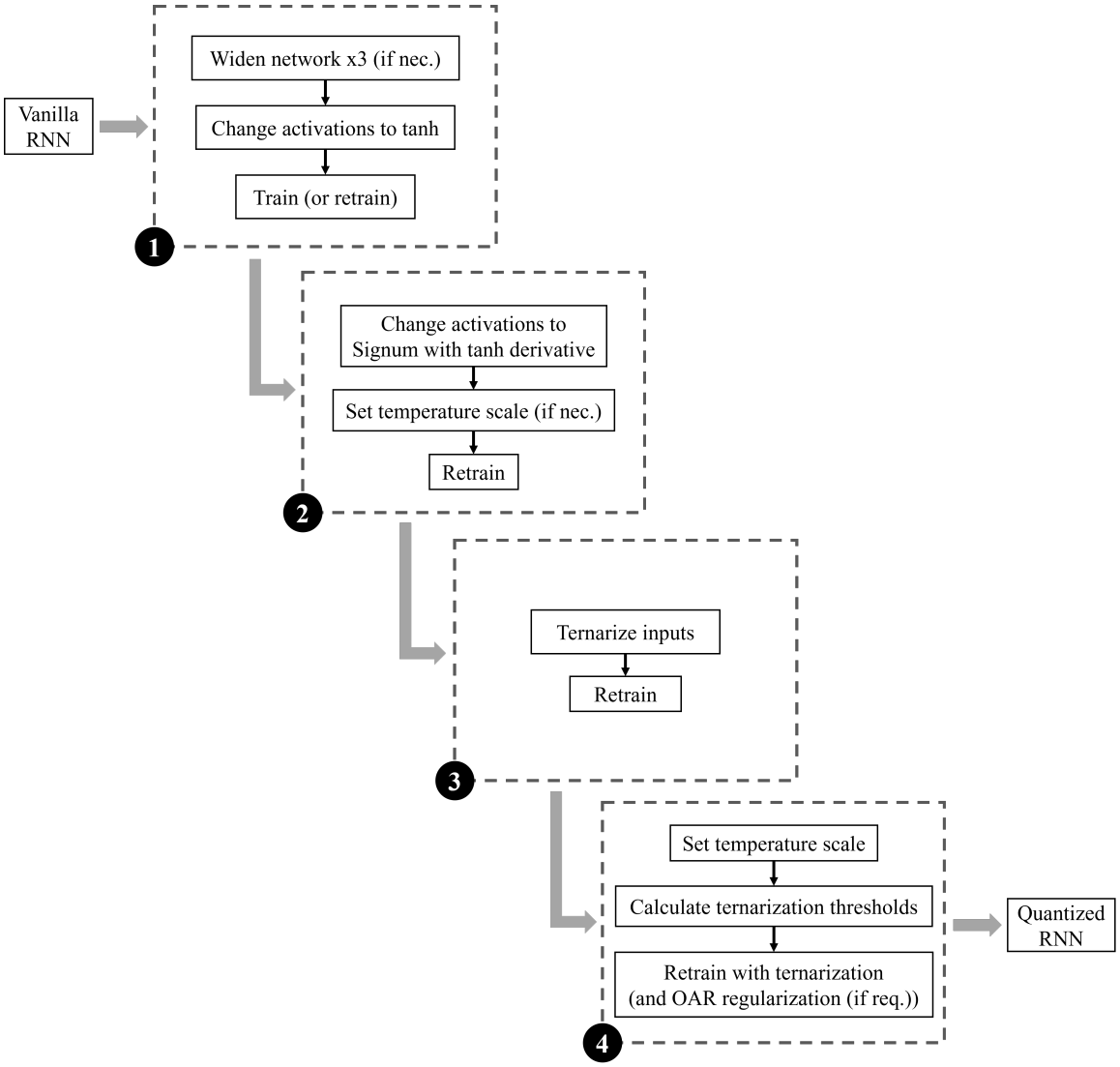


Figure 4.1: **The four-step vanilla RNN quantization process**.

4.2 Conversion From Plaintext To CGGI-Compatible RNNs

In section 4.1, we proposed a quantization-aware training method for quantizing RNNs with CGGI-friendly operations. The resulting quantized RNN performs additions and subtractions rather than scalar multiplications, which allows us to use less expensive CGGI operations. It quantizes parameters to 2-bit, ternary precision and activations to 1-bit, binary precision, leaving a lot of room for accumulation (up to 8-bits). It can also support larger precisions for accumulation when trained with OAR regularization. The activation functions are set to the Signum function, which is a negacyclic function well supported in CGGI through a single programmable bootstrap operation (for more information on programmable bootstrapping, we refer the reader to sections 2.4.3 and 2.4.4). This is all that is necessary to implement basic, large-scale RNNs with additional feed-forward layers using

CGGI operations over encrypted data.

The MNIST RNN architecture we propose in section 3.3 contains two RNN layers with two feed-forward layers for classification on top. The last RNN layer before the feed-forward layers outputs a feature vector for each timestep in the sequence. To feed each vector into the dense layer and output a prediction based on the entire sequence, the outputs of the last RNN are flattened into one large vector of size $T_l \times o_l$, where T_l and o_l are the size of the sequence and number of output units in the last RNN respectively. When this vector is fed into Dense 0, or in general a feed-forward network with number of output units o_{l+1}, the size of the parameter matrix in the feed-forward layer becomes dependent on the number of timesteps. Specifically, the size of the parameter matrix is $T_l \times o_l \times o_{l+1}$, due to the fully-connected property of feed-forward layers. This presents a potential problem for large sequences—the number of parameters increases with the number of timesteps. This also limits the types of recurrent networks we can evaluate to those that are evaluated over sequences of a fixed length. For some applications where the sequence is fixed, this is fine, but for others such as speaker identification, sequences could be of variable size and require only one output, not a sequence of outputs. Thus, it is necessary to find a method to aggregate the output sequence of RNN layers so that the sequence can be of variable length, while still being able to classify inputs based on every output in the sequence.

In section 2.1.2, we discussed an aggregation method, proven to work with RNNs, called attention. If we approximate the attention mechanism by a feed-forward network, which is the technique employed by *Bahdanau et al.* [7], we would need to multiply each member of the RNN's output sequence by the output of the feed-forward network. In this scenario, the query and key matrices are the same—they are the outputs of the last RNN. They are fed into a feed-forward network to determine the relative scales of each member of the output sequence (the values). This type of addition to the RNN network allows the RNN to determine which of its outputs is most important for the task, and the features that determine this importance, enabling the evaluation of RNNs with variable-length sequences.

While this helps us cover more types of RNNs, it forces us to perform ciphertext-ciphertext multiplication since the outputs of the attention mechanism are multiplied by the outputs of the RNN, both of which are encrypted. In section 4.2.1, we propose a new way of multiplying two binarized ciphertexts by taking advantage of their binary nature. As a result, the new method is more efficient than alternative methods. In section 4.2.2, we observe a negative effect of performing the Signum PBS over values close to zero and propose a way to train the network to handle this additional noise. In the last section, we discuss a way to extend the precision of the output layer. The combination of these techniques enable the evaluation of large-scale RNNs over encrypted data. We put all of these techniques together in section 4.3.

x	y	$x-y$	$m\cdot(x-y)$	$x\cdot y$
$+1$	$+1$	0	0	$+1$
$+1$	-1	$+2$	$+2m$	-1
-1	$+1$	-2	$-2m$	-1
-1	-1	0	0	$+1$

Table 4.1: **Novel Binary Multiplication**. A truth table for $x \cdot y$ where $x, y \in \{-1, +1\}$, extended to include mappings for a novel multiplication method using only addition/scalar multiplication.

4.2.1 Multiplication Of Binarized Ciphertexts Using One PBS

Although the external product (discussed in section 2.4.3) offers a way to multiply two ciphertexts, it restricts the types of circuits we can evaluate to those that multiply GGSW with LWE ciphertexts. In a privacy-preserving machine learning scenario, and specifically for attention layers, the multiplication operation can occur between two LWE ciphertexts. A way to apply the external product in this case is to promote one of the LWE ciphertexts to a GGSW ciphertext using the circuit bootstrapping operation defined in CGGI. However, the circuit bootstrap is a computationally expensive operation [47] and would dramatically increase the execution time of circuits that are abundant in multiplication operations. For this reason and the absence of a native LWE-LWE multiplication operation, the authors of CGGI suggest an inner product operation between LWE ciphertexts called BFV-like multiplication [23]. After testing the BFV-like multiplication, we observed that with any set of practical parameters, the noise increases past the noise budget after one multiply. The external product is great for operations such as programmable bootstrapping where a fresh GGSW encryption of values that are multiplied is always available. We refer the reader to [23] for a rigorous definition of the methods discussed above.

In response to the ineffectiveness of these methods, the creators of CGGI propose a programmable bootstrapping approach to multiplying two LWE ciphertexts. According to [23], the following mathematical relationship between two integers x and y can be calculated using two PBS operations evaluating $a \to \frac{a^2}{4}$:

$$x \cdot y = \frac{(x+y)^2}{4} - \frac{(x-y)^2}{4} \tag{4.2}$$

where the PBS operations occur over $a = x + y$ and $a = x - y$. This is the state-of-the-art for multiplying two LWE ciphertexts using CGGI. It is possible to evaluate both PBS operations at once, making the execution time equivalent to running only one PBS. Nevertheless, computational resources still need to be reserved for two PBS operations.

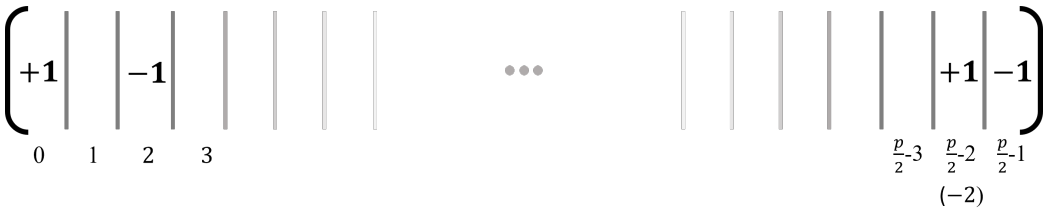
In our situation, both x and y are binary values (since we are multiplying two activations). If both x and y are binary, the multiplication of $x \cdot y$ becomes simpler. We observe that there are four possible combinations and outputs of the function $x \cdot y$. We summarize and extend them in table 4.1.

The key observation is that subtracting x and y yields three unique values: 0, +2, and −2. As a result, we can create a lookup table (LUT) to map $\text{LWE}(x) - \text{LWE}(y)$ to the correct encrypted $\{-1, +1\}$ values for $\text{LWE}(x) \cdot \text{LWE}(y)$ using a **single programmable bootstrap**. In the table, we see that both subtraction results of 0 map to the correct multiplication results of +1. Similarly, the subtraction results of +2 and −2 both map to the correct multiplication results of −1. Thus, we have to create an LUT with different values. Recall from section 2.4.4 that the LUT in CGGI is a trivial RLWE ciphertext with N coefficients (N being the ring dimension of the RLWE setting). Since the plaintext precision $p < N$, encoded LUTs must contain packs of repeated values (which are packs of repeated coefficients in the RLWE ciphertext) to perform correct blind rotation. We refer to these packs as mega-packs for the remainder of this discussion. The number of values we can encode is dependent on the negacyclic property of the function being encoded. If the function is negacyclic, then we can encode $p/2$ values, meaning there will be $p/2$ mega-packs. Otherwise, we must encode p values resulting in p mega-packs. For the Signum function, we do not have to worry about setting different mega-packs since the negacyclic property allows us to encode +1 in every coefficient (i.e. negative values are wrapped around N into the negative domain by the rotation and positive values are already in the positive domain). For our multiplication table, we need to encode three separate values, and we can use the negacyclic property.

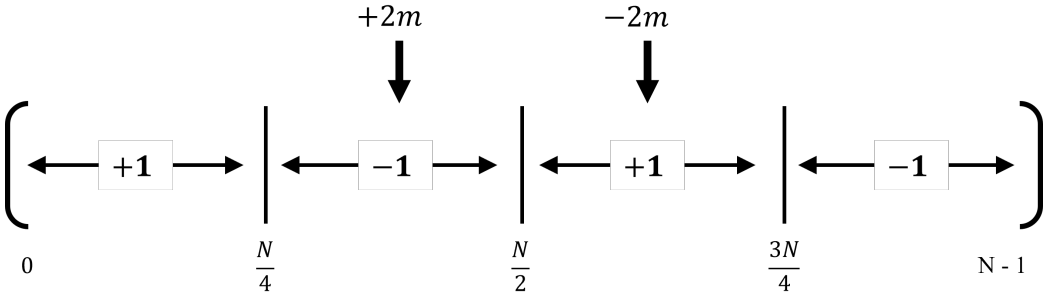
Let us refer to the output of the multiplication lookup table as $\text{Mult}(x-y)$. To aid in our explanation, figure 4.2a illustrates the LUT we are creating. In the figure, the mega-packs are indexed underneath. The indexed spaces between two lines indicates a mega-pack of repeated coefficients and the values at the center are the encoding applied to each coefficient in the mega-pack. To encode $\text{Mult}(0) = +1$, we can map the first mega-pack to +1. To encode $\text{Mult}(+2) = -1$, we can map the third mega-pack to −1. To encode $\text{Mult}(-2) = -1$, we can make use of the negacyclic property. Since −2 is negative, rotation causes the LUT to wrap around N by two mega-packs and output the negation of the second last mega-pack as a result (indexed by "(-2)" underneath in the figure). In response, we can encode the second last mega-pack to +1 which results in −1 when $\text{LUT}(-2)$ is evaluated (where LUT is the lookup operation in this case).

The question of what values to encode in the remaining mega-packs remains. In truth, it does not matter as long as the error in the ciphertext encrypting $x - y$ is not too large. If the error is large and negative, then it is possible that $\text{LUT}(x - y = 0)$ could wrap over N into the negative domain and output the value encoded in the last mega-pack. To mitigate this, we can set the last mega-pack to encode −1. Therefore, in any case, if the error is large enough and $\text{LUT}(x - y = 0)$ indeed does wrap around N, then the output will still be $-(-1) = +1$, the correct value for $\text{Mult}(x - y = 0)$. The rest of the values do not have to be encoded, as shown in the empty spaces in figure 4.2a.

Another observation is that the values −2, 0, and +2 are relatively close together.



(a) Original



(b) Optimized

Figure 4.2: **Novel Multiplication Lookup Table**. The original LUT design (a) can be optimized to allow for more redundancy and a larger rounding interval as a result. Therefore, the optimized version (b) can theoretically handle larger noise and more consecutive operations prior to bootstrapping.

When the modulus switching operation changes them from a modulus of q to $2N$, which is one of the first operations in the programmable bootstrap, it is possible that a large error could move values in one mega-pack to another, similar to the phenomenon discussed in the previous paragraph for LUT$(x - y = 0)$. For example, in figure 4.2a, the mega-packs indexed by LUT$(x - y = -2)$ and LUT$(x - y = -1)$ are adjacent. If the error is too large in either ciphertext encrypting those values, then the incorrect sign could be outputted. It is possible to optimize this and render this scenario very unlikely to occur. We show the optimized LUT in figure 4.2b. The indices underneath the LUT now refer to the indices of the coefficients in the RLWE trivial ciphertext. From figure 4.2a, we can see that the vast majority of mega-packs (and thus, coefficients) are empty. We can use the extra space for more redundancy and thus, better rounding during blind rotation. This means that if the error is too large for a certain ciphertext, say LWE(-2), separating it as far as possible from the other mega-packs would increase the amount of error tolerance in rounding. Given that we are encoding four values, we can maximize redundancy for each value by partitioning the LUT into four segments, resulting in four mega-packs of $N/4$ repeated coefficients, as shown in figure 4.2b. To map each value to the center of each segment, we multiply LWE$(x - y)$ by a scalar m. We define m in the following equation:

$$m = 3 \cdot 2^{\log_2(p)-5} \tag{4.3}$$

Type of Multiplication	**Data Type Support**	**Noise Growth**	**Types of Ciphertexts**	**Notes**
External Product	Integers	Medium	GGSW-LWE	One ciphertext must be GGSW, requires circuit bootstrapping
BFV-Like Multiplication	Integers	Severe	LWE-LWE	Severe increase in noise
PBS Multiplication	Integers	Very Low	LWE-LWE	Requires two PBS operations
This Work	**Binary Integers**	**Very Low**	**LWE-LWE**	**Requires one PBS operation**

Table 4.2: **Summary of CGGI LWE-LWE multiplication methods**.

for plaintext precision $p = 2^{\omega}$ and $\omega \geq 5$. The output of this operation is shown in the fourth column of table 4.1. Multiplying $m \cdot \text{LWE}(0)$ only multiplies the error. If the error is positive, the value should land in the first quarter during rotation. If it is negative, it should land in the last quarter. Similar to the original LUT, the coefficients in the last quarter are assigned a value of -1 to counteract the outer negative sign resulting from the negacyclic property. The coefficients in the first quarter are assigned a value of $+1$. Multiplying $m \cdot \text{LWE}(+2)$ should point to the second quarter, where we assign the coefficients to a value of -1. Similarly, multiplying Multiplying $m \cdot \text{LWE}(-2)$ should cause it to point to the third quarter, where we assign the coefficients to a value of $+1$ ("+" to counteract the negacyclic property). Thus, the amount of tolerable error is maximised.

In summary, to perform our multiplication operation between two ciphertexts $\text{LWE}(x)$ and $\text{LWE}(y)$ where $x, y \in \{-1, +1\}$, we perform the following set of operations:

$$\text{LWE}(x) \cdot \text{LWE}(y) = \text{PBS}\left(m \cdot (\text{LWE}(x) - \text{LWE}(y))\right) \tag{4.4}$$

where PBS denotes a single programmable bootstrap using the LUT from figure 4.2b, and m is calculated according to equation 4.3. Table 4.2 summarizes and contrasts the various multiplication methods for two binary ciphertexts. As a result of the simplification made possible by the binarization of ciphertexts, our method is the least expensive method with respect to both number of operations and noise growth. In section 4.3, we make use of this operation to perform attention.

4.2.2 Grey Signum Activation Function

In the previous section, we touched upon a problem associated with evaluating LUTs for an input of "0". This problem also arises when evaluating the Signum function. If the error of $\text{LWE}(0)$ is too large and negative, the Signum PBS operation could output $\text{LWE}(-1)$

rather than the correct value of LWE(+1). Given that pre-activations are accumulations of potentially large amounts of values, this type of scenario is possible. Similar to the other methods we have already proposed, we can introduce this error to the training loop, allowing the model to adjust accordingly. Specifically, we propose a novel activation function—the **grey signum activation function**. Since assigning the output of Signum(0) = 1 is a convention, we propose changing the output of Signum(0) to reflect what is actually happening over the encrypted domain. The grey signum activation function is a probabilistic function. The function is presented in equation 4.5.

$$\text{GreySignum}(x) = \begin{cases} -1 & \text{if } x < 0 \\ -1 & \text{if } x = 0, \text{ with probability } p \\ +1 & \text{if } x = 0, \text{ with probability } 1 - p \\ +1 & \text{if } x > 0 \end{cases} \tag{4.5}$$

For values less than zero, and values greater than zero, the output is the same as the Signum function. For values that are equal to zero, we assign them to -1 with a probability of p, or $+1$ with a probability of $1 - p$. The probability p is empirically determined from observing the percentage of values that output -1 when running the Signum PBS function on encrypted vectors of all zero elements in CGGI. That way, while training, the grey signum activation function is used and the model can adapt to the added error, which acts as simulated error. The grey signum function is similar to the stochastic binarization function in BinaryConnect [28], with differences in both the way the probability is calculated and the range of values stochastically assigned. For the grey signum activation function, probability is determined empirically based on outputs of CGGI circuits, and only the values equal to zero are stochastically assigned whereas in BinaryConnect [28], probability is determined by a variant of the `sigmoid` function and DNN parameters, and values other than zero, including positive values, can be assigned to -1.

4.2.3 Wider Precision For Output Logits

In the output layer, accumulation becomes more important than sign. As mentioned previously, we do not calculate a `softmax` or `sigmoid` function in the output layer. Instead, we return the encrypted output logits. For this reason, we cannot guarantee security of the model parameters. By returning the logits, the client can decrypt the logits and perform a `softmax` or `sigmoid` calculation in the clear. When calculating the `softmax`, the magnitude of the values is important in order to determine the maximum. Therefore, numeric overflow in the last layer could become a problem. In section 3, we observed there is a way to harness overflow and make it useful. However, this observation was a result of the network using the Signum function for every activation (when overflow occurs, it is possible to overflow multiple times and regain the sign changed by the initial overflow).

We can increase the size of the security parameters in CGGI to increase the available plaintext precision in the output ciphertexts, in order to support a larger accumulation space. In response, this would decrease the efficiency of the CGGI computation and not be desirable. Since the output layer is composed of a matrix multiplication operation only, we can divide the summation in each dot product into several smaller summations. Rather than summing each dot product into one ciphertext of large precision for each output unit, this procedure creates several ciphertexts representing the smaller summations, each requiring a smaller precision. Together, d ciphertexts represent each output unit. Instead of sending o_{L-1} ciphertexts to the client to decrypt, this would send $o_{L-1} \cdot d$ ciphertexts, for a division parameter d and number of output units o_{L-1} ($L-1$ is the index of the last layer out of L layers). We do not worry about the expansion in network bandwidth since it is not a focus in this thesis. On the client side, the ciphertexts are decrypted and recombined to complete the full summation for each output unit.

In our quantization method, the matrix multiplication operation in the output layer is composed of a series of dot products containing additions and subtractions of "-1" and "$+1$" (the input to the layer is a vector of binarized ciphertexts and the parameter matrix is ternarized). This allows us to set an upper bound on the values of the output of each dot product. If all the parameters are equal to one and the inputs are all equal to one as well, then the maximum absolute value of the dot product is the number of input units i_{L-1}. Taking this into consideration, if we set a value for the d parameter, then the ciphertexts containing the smaller summations are bounded accordingly—they would need to be able to handle accumulations up to i_{L-1}/d or down to $-i_{L-1}/d$, thus bounding the plaintext precision for each ciphertext by $p \geq 2 \cdot i_{L-1}/d$. By increasing d, we can guarantee that there will be no overflow.

The following equation summarizes our process. In the matrix multiplication between an input vector $\boldsymbol{x} \in \{-1, +1\}^{i_{L-1}}$ and parameter matrix $\boldsymbol{W} \in \{-1, 0, +1\}^{i_{L-1} \times o_{L-1}}$, let us observe the calculation of each output unit, which is the dot product between each row of $\boldsymbol{W}$, denoted by $\boldsymbol{W}_k$, and $\boldsymbol{x}$.

$$\langle \boldsymbol{W}_k, \boldsymbol{x} \rangle = \sum_{j=0}^{i_{L-1}} W_{k,j} \cdot x_j = \underbrace{\sum_{j=0}^{i_{L-1}/d-1} W_{k,j} \cdot x_j}_{\text{Summation } 0} + \underbrace{\sum_{j=i_{L-1}/d}^{2 \cdot i_{L-1}/d-1} W_{k,j} \cdot x_j}_{\text{Summation } 1} + \ldots + \underbrace{\sum_{j=(d-1) \cdot i_{L-1}/d}^{i_{L-1}-1} W_{k,j} \cdot x_j}_{\text{Summation } d-1} \tag{4.6}$$

Equation 4.6 shows how the dot product that calculates each output unit can be divided into several smaller dot products. It also shows how they can be recombined to evaluate the original dot product. In our RNN evaluation method with CGGI, each smaller dot product is calculated by the server through a series of additions or subtractions of binarized ciphertexts. The server sends every resulting ciphertext for each output unit to the client.

The client decrypts the ciphertexts and recombines them according to equation 4.6 in the clear for each output unit.

Summary

In this section, we discussed three techniques for evaluating RNNs using CGGI operations. The first technique allows us to perform more efficient multiplication of two binarized ciphertexts and in return, enables the evaluation of attention layers. Attention layers are very effective aggregation layers when used to aggregate RNN outputs. This allows us to evaluate RNNs over sequences of variable length. In the next section, we define a new DNN architecture, the *SpeakerID RNN*, for modelling the task of speaker identification, a sub-task of speaker recognition. We utilize multiple RNN layers and attention to evaluate audio waveforms of variable length. Using the techniques in this section along with our novel RNN quantization process detailed in section 4.1, we convert both the MNIST RNN and SpeakerID RNN architectures to CGGI-compatible models. Section 4.3 details our implementation and section 4.4 presents the evaluation results. We also utilize the other techniques introduced in this section, the grey signum activation function and expansion of output ciphertexts operation. The grey signum activation function mitigates the error introduced by the large amount of zeroes in pre-activation distributions while the expansion of output ciphertexts guarantees there is no overflow in the last layer, a problem that must be avoided.

4.3 Implementation

In this section, we detail our implementation of two RNNs and their conversion to CGGI-compatible RNNs. In the previous two sections, several methods were introduced that aid in this task. In section 4.1, we proposed a novel quantization process that quantizes RNNs into ternarized parameters and binarized activations. This is a necessary first step in converting RNNs to CGGI-compatible RNNs. In section 4.2, we proposed three helpful methods to convert quantized RNNs to CGGI-compatible RNNs—specifically, (1) a novel multiplication method to compute attention layers, (2) a new variant of the Signum function that adjusts the network to handle error introduced by CGGI, and (3) a way to divide the matrix multiplication in the last layer into several, smaller dot product operations in order to ensure numeric overflow does not occur. We utilize all of these methods as well as the OAR regularizer introduced in chapter 3 in our implementation.

We begin by introducing the SpeakerID RNN, a multi-layer, deep RNN with attention that models the speaker identification task in section 4.3.1. In section 4.3.2, we then discuss our application of the RNN quantization process introduced in section 4.1 to both the MNIST RNN and SpeakerID RNN architectures. Before we detail our CGGI conversion process for both RNNs in section 4.3.3, we perform benchmarking for various CGGI security

parameter sets with a minimum security level of $\lambda = 128$ in appendix G.2.1 to determine the ideal parameter set for use in our experiments. Finally, in section 4.4, we showcase our experimental results.

4.3.1 Novel Speaker Identification RNN

The speaker identification task is a difficult real-world problem that requires (1) a speaker to record his speech, and (2) a model to classify the correct speaker given a discrete-sampled analog waveform as input. Since input speech could be considered private information, there is a natural application for this type of technology over encrypted speech data. For instance, we can imagine a scenario where a government agency would like to determine from a waveform who is speaking while keeping the content of the speech private.

One of the most popular datasets containing speech data is VoxCeleb [90]. VoxCeleb is considered a large-scale dataset for various speaker tasks such as identification (a multi-class problem to classify speakers from raw audio), verification (comparison of one embedding to others to verify the identity of a speaker), and diarisation (to discover who is speaking and when). This thesis focuses on speaker identification. The dataset is compiled from various recordings of celebrities speaking on YouTube, all of which are recorded in-the-wild without any pre-processing. There are two datasets: VoxCeleb1 and VoxCeleb2 [91]. VoxCeleb2 contains many more samples than VoxCeleb1, however VoxCeleb1 is already fairly large enough to experiment with and thus, is the dataset used in this thesis. VoxCeleb1 contains 1251 speakers and 153,516 utterances in total. On average, each utterance is 8.2s long and there are 123 utterances per speaker. The dataset is fairly balanced in terms of sex with 55% of the speakers being male. The speakers are very diverse, with varying ages, ethnicities, accents, and professions. The recordings are taken in various settings such as concert halls, quiet rooms, and include conversations with multiple speakers in the background.

In our implementation, we continue to use TensorFlow for training and quantizing our networks. Using TensorFlow datasets, we can import VoxCeleb1 easily [1]. The dataset split is 134,000 for training, 6,670 for validation, and 7,972 for test. The utterances are arrays of 64-bit signed integers, sampled at a frequency of 16kHz. Every utterance is labelled by a speaker identification number from 1 to 1251. The original VoxCeleb1 paper [90] proposes a convolutional network to perform speaker identification, similar to VGGNet [126]. They obtain a top-1 test accuracy of 80.5% and top-5 test accuracy of 92.1%. Top-5 accuracy is the proportion of correct labels in the top-5 predictions after each inference. The accuracy they report is also over the full utterance and not segments. When training, they train over segments of three seconds.

Since the focus of this thesis is RNN evaluation, we propose a new RNN architecture that can be evaluated over speech samples of variable length. In figure 4.3, we present the **SpeakerID RNN**. It is similar to the MNIST RNN model, differing in the replacement of

[1] https://www.tensorflow.org/datasets/catalog/voxceleb

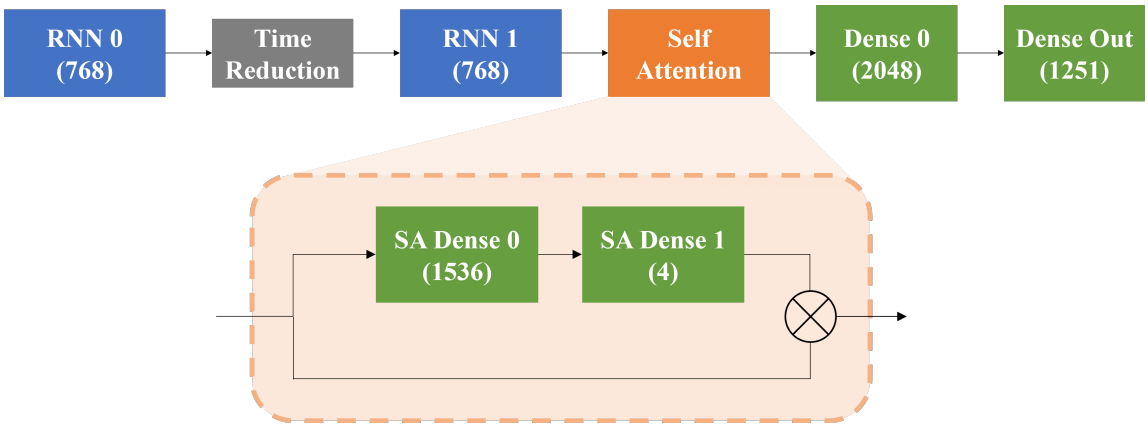


Figure 4.3: **The SpeakerID RNN**. A novel RNN architecture for modelling the speaker identification task.

the flattening layer, which flattens the RNN output sequence, with an **attention** layer to aggregate the outputs along the time dimension instead. The time reduction factor is two, similar to the MNIST RNN. The model contains 12,644,352 parameters, which is larger than the MNIST RNN by about an order of magnitude. The size of this model is already tripled from the original size to enable quantization, following step one of the four-step RNN quantization process detailed in section 4.1.3. By removing the flattening layer, the model size is no longer dependent on the number of timesteps in the RNN. For any number of timesteps, the model size remains the same, as a result of the addition of an attention layer.

The attention layer, shown in the bottom half of the figure, contains two dense layers, the first with 1536 units and the second with 4 units. Since the input to the attention layer is a sequence of feature vectors with each vector being an output from the final RNN layer for a particular timestep, the dense layers in the attention layer are applied to each feature vector in parallel. Effectively, the parameters are shared across the timesteps. The dense layers learn the importance of each timestep and output four values for each one. The four values act as four attention heads. By performing the final matrix multiply in the attention layer between the outputs of the final RNN and each of the attention heads, the attention layer outputs a vector that is four times the number of output units in the final RNN. Thus, it aggregates and scales each feature in the final RNN and presents four different views for the classification layers (the final dense layers) to expand upon. The final layer outputs a vector with a size equal to the number of classes (in this case, speakers).

Drawing inspiration from [50], we pre-process the raw audio of each sample using a set of operations to create input feature vectors with 320 elements for each timestep. The sampling rate for the audio samples is 16,000 samples/second. The following list represents the set of pre-processing steps, in order.

1. From the sample, extract a random 4 second segment of audio (roughly around 400

frames). Pad with zeros if necessary.

2. Perform a Short-time Fourier Transform to generate a spectrogram for a frame size of 25ms, frame step size of 10ms, and FFT size of 1024.

3. Transform spectrogram from linear scale to mel-frequency scale using the following settings: 80 mel-scale bins, 80Hz lower bound, and 7600Hz upper bound.

4. Apply the natural logarithm to obtain log-mel frequency spectrogram.

5. Normalize.

6. Stack the frames in the spectrogram, along the time dimension, in groups of four frames, while downsampling by a factor of three at the same time. In other words, follow this process: stack four frames, move a stride of three frames, and repeat, yielding an overlap of one frame. Pad if necessary.

Since we are using 80 mel-scale bins, each timestep of the pre-processed sequence should be represented by a vector of 32-bit floating-point numbers with $80 \cdot 4 = 320$ elements. Refer to the codebase for the functions used in this section. With respect to CGGI conversion, the client completes all pre-processing steps *before* sending the encrypted input sequence to the server. We leave the calculation of pre-processing steps in the encrypted domain for future work.

There are two ways to evaluate the SpeakerID RNN. The first is to run the model on the test or validation set using random, 4 second samples. During training, we use this method to track the validation accuracy. The second way is to run the model on the whole sample, which we call a *full evaluation*. Effectively, this is the same as removing step one from the pre-processing sequence of operations above.

4.3.2 Quantization

In this section, we discuss our quantization implementation for both the MNIST RNN and SpeakerID RNN models. For both RNNs, we follow the novel four-step quantization process outlined in section 4.1.3. We present our results in section 4.4.1.

Step One

The first step is to train both networks in full precision, 32-bit floating-point. We discussed training the MNIST RNN in full precision in section 3.3. The number of parameters are not tripled, as suggested by the four-step process, since we were able to quantize without problems. For the SpeakerID RNN, as mentioned previously, the parameters are already tripled since a lesser amount did not quantize well. The activations are all set to `tanh`. Since we are using attention, we utilize the learning rate scheduler proposed in [129]. We set the number of warmup steps (not epochs) to 5400, and the size of the model to the size

```
import tensorflow as tf
def sign_with_tanh_deriv(x):
    out = tf.keras.activations.tanh(x)
    q = tf.math.sign(x)
    q += (1.0 - tf.math.abs(q))
    return out + tf.stop_gradient(-out + q)
```

Figure 4.4: **Signum activation function with `tanh` derivative**.

of the first dense layer in the attention module, 1536. We use the Adam optimizer without gradient clipping, and a batch size of 512. We also use $\mathcal{L}_2$ regularization with a rate of $1e^{-4}$. After 261 epochs, **we achieve a full evaluation test accuracy of 82.0%**. For the full evaluation, we also achieve a top-5 test accuracy of 92.5%. Thus, the model achieves a better accuracy than the VGG-like model in VoxCeleb [90]. Since the purpose of this thesis is to evaluate RNNs over encrypted data, we do not pursue higher levels of accuracy.

Steps Two and Three

Step two involves changing the activation function of the model in step one and retraining. To change the activation function to the Signum function with a `tanh` derivative, we utilize the `tf.stop_gradient()` function in TensorFlow to stop the gradient through the correct forward function, which is Signum, while still outputting the correct Signum output. TensorFlow is tricked into believing the forward function is `tanh`, which is why it uses the `tanh` derivative in backpropagation, while outputting the Signum function with no gradients flowing through it. The code snippet in figure 4.4 shows our implementation.

We draw inspiration for this method of implementing one type of forward function and a different type of backward function from **QKeras** [2]. QKeras is a quantization extension of the Keras library in TensorFlow that includes quantized layers and activations. These layers can be used as drop-in replacements for the associated full precision layers (e.g. Dense to QDense, RNN to QRNN), to perform quantization-aware training. Each QKeras layer contains settings for quantizers, which quantize the parameters during the forward step. By assigning the activation functions to quantized activation functions, the model performs QAT. We utilize QKeras throughout our implementation.

We can apply a temperature scale in a similar way. The code snippet in figure 4.5 shows the division of the temperature scale, which TensorFlow applies to the backwards gradient, and the addition of the non-divided pre-activation, which TensorFlow outputs. We do not apply a temperature scale in step two in any of our implementations. We apply the temperature scale in step four. For both RNNs, we initialize all of the parameters with those in the converged model in step one. We change all of the activations to the

[2] https://github.com/google/qkeras

```
import tensorflow as tf
# Add two dot products (pre_act = W_x*x + W_h*h)
# Division is to control the gradient but output is W_x*x + W_h*h
output = pre_act/s + tf.stop_gradient(-pre_act/s + pre_act)
```

Figure 4.5: **Application of temperature scale to RNN gradients**. *pre_act* stands for pre-activation, which is the addition of both matrix multiplies in the Vanilla RNN (refer to section 2.2.1 for more information).

`sign_with_tanh_deriv` function from figure 4.4. For the SpeakerID RNN, we do not use the attention learning rate scheduler. Rather, we run Adam with a set learning rate of $1e^{-4}$. Aside from that, the training setup for both RNNs remains the same as in step one. After training both models for step two, we reinitialize the models, ternarize the inputs, and retrain using the same training setup.

Step Four

In step four, we reinitialize the parameters from step three for both models. After some experimentation, we find that a temperature scale of 4 and 5 for the RNN layers in the MNIST RNN and SpeakerID RNN networks respectively provide good results. We also calculate the expected values of the parameter sets for each layer according to equation 4.1. We set the ternary threshold scales to 1.5 and 1.6 for the MNIST RNN and SpeakerID RNN models respectively. To train the MNIST RNN, we use the same training setup as in section 3.4. For training the SpeakerID RNN, we also use a Cosine learning rate scheduler, starting from $2e^{-6}$, and decaying to $2e^{-7}$ after 100 epochs. We use a batch size of 512 and the Adam optimizer without gradient clipping. We stop using $\mathcal{L}_2$ regularization at this point for both models.

For OAR regularization and low-bit accumulation, the results for the MNIST RNN are presented in section 3.4. Unfortunately, OAR regularization did not work for the SpeakerID RNN architecture. It is possible that the addition of an attention layer could be a problem, being the major difference between both model architectures. It is also possible that the VoxCeleb dataset presents a difficult challenge for this type of regularization. We leave an investigation of this problem for future work. For evaluating our quantization process itself, we let the pre-activations in this step be represented by 32-bit integers. The arithmetic is still integer arithmetic, except that we do not use the ModSignum function. Thus, accumulation occurs in 32-bit integers.

4.3.3 CGGI Conversion

We implement the MNIST RNN and SpeakerID RNN using Concrete [21]. Specifically, we use a modified version of the Concrete-Core library[3]. To perform the PBS operation on an array of ciphertexts using multiple GPUs, the Concrete-Core library divides the array across each GPU and attempts to perform the PBS operation on each GPU simultaneously.

We provide a discussion of the parameter sets used in this thesis in appendix G.2.1. We also run some benchmarks to help in our selection of parameter sets. The three main parameter sets are defined in table G.2, where set 1 is the largest and least efficient set, and set 3 is the smallest and most efficient set.

We perform a dot-product, KS, and PBS (the output of a neuron in neural networks) in the same order as recommended in [10], specifically AP type 1. AP type 1 performs a dot-product first, then a key-switching operation, and finally a PBS, and is empirically shown to achieve the lowest noise propagation. Performing a key-switch is necessary since the bootstrapping key is an encryption of the original secret key, restricting the types of ciphertexts that can be bootstrapped to those that are encrypted using the original secret key. Furthermore, the output of the PBS is an LWE encryption using the coefficients of the RLWE secret key, making it necessary to switch the LWE ciphertext back to an encryption under the original secret key. Thus, we encrypt the inputs to each network using the extracted LWE key from the RLWE secret key.

We encrypt each input as an LWE ciphertext, creating arrays of input ciphertexts. We perform dot-products between ternarized parameters and encrypted inputs using addition and subtraction operations, and no scalar multiplications. For parameters that are equal to zero, we do not add or subtract them, we skip them to reduce the error (since error is still a part of an encryption of zero). Scalar multiplication only occurs for the multiplication of the m parameter prior to the bootstrap in the novel, single-PBS multiplication algorithm we introduced in section 4.2.1. Alongside our implementation over encrypted data, we perform an inference over plaintext data as well, so that we can evaluate the differences between the encrypted and plaintext outputs of each layer.

The implementation of the RNNs is sequential, meaning each timestep is processed after the other. We do not perform any pipelining between RNN layers. For example, if there are two RNN layers in the network (indexed by l and $l+1$), pipelining allows you to parallelize their evaluation by evaluating timestep t_i^l and t_{i-1}^{l+1} at the same time, for $i \in [0, T-1]$ where T is the size of the sequence. We execute each layer sequentially. For example, in the MNIST RNN, we execute RNN 0 first (28 timesteps), then execute RNN 1 ($28/2 = 14$ timesteps), then Dense 0, and finally Dense Out. With respect to inference latency, our implementation of executing both RNN layers sequentially is equivalent to executing one

[3] A bug causes the amortized PBS operation to be performed one GPU at a time instead of using multiple GPUs asynchronously in the original Concrete-Core library. Our modifications to the library, located in the following repository, fix this bug: https://github.com/irskid5/concrete-core/tree/3c9d247

RNN layer with $28 + 14 = 42$ timesteps. The only parallelization we experiment with is GPU acceleration for PBS and KS operations. Due to time constraints, we leave multi-core CPU experimentation for future work.

The models are implemented in an end-to-end, non-interactive setting. This means that there is no communication between the client and server during inference—the client encrypts the data, sends it to the server; the server executes the inference, and sends the final results back to the client for decryption. We simulate the client in the server, and thus do not participate in any data transfer, since it does not influence any of the algorithms discussed in this thesis.

For the MNIST RNN, we experiment with OAR regularization and thus, a lower precision level of 6 bits. For the SpeakerID RNN, since OAR regularization did not work, we experiment with a higher precision level of 11 bits, to accommodate the largest pre-activation distribution (the Dense 0 layer). We also perform step four of the four-step quantization process with the GreySignum function as well as the Signum function since a larger precision reduces the rounding interval for each query in the LUTs, increasing the probability of queries of zero outputting the incorrect value. For the MNIST RNN, we divide the last layer into 4 accumulations and for the SpeakerID RNN, into 64 accumulations, following the method introduced in section 4.2.3. Using only one accumulation results in a large decrease in accuracy.

We pre-process both MNIST and VoxCeleb1 datasets in Python using TensorFlow, and extract the datasets and quantized parameters. We import the extracted values into the Concrete-Core implementation and use them to evaluate both the MNIST and SpeakerID RNNs. Our Rust implementation uses only Concrete-Core for the RNN evaluation. We create the inference layers and computational graphs from scratch, using only Concrete-Core operations. For training and extraction of parameters and datasets, we use machine A. For our implementation over encrypted data as well as benchmarking, we use machine B. For more detailed information regarding our experimental setup, refer to table G.1. We open source both the training[4] and CGGI implementation[5] codes on GitHub.

4.4 Results and Discussion

We discuss our results regarding the implementation of large-scale RNNs over encrypted data, in an end-to-end, non-interactive setting. We begin by observing the effects of the four-step quantization process in section 4.4.1. We conclude by discussing the accuracy and latency results of executing the MNIST RNN and SpeakerID RNN architectures over encrypted data in section 4.4.2. Additionally, we provide a detailed analysis of the effect of temperature scaling with regard to successful quantization of RNNs in appendix G.4.1. For

[4] https://github.com/irskid5/tf_speaker_rec
[5] https://github.com/irskid5/tfhe_rnns

the experiments in this section, we use two experimental setups: **machine A** and **machine B**. Further details are available in table G.1 in appendix G.

4.4.1 Four-Step Quantization Results

In this section, we present our results for quantizing both the MNIST RNN and SpeakerID RNN using the four-step quantization process. In table 4.3, we present training/retraining results for each step of the four-step process. The results are obtained from following the implementation process detailed in section 4.3.2 and training using machine A. We display the top-1 and top-5 accuracies over the test sets of each RNN. For the SpeakerID RNN, we present the full evaluation accuracies, which are obtained from evaluations over the whole sequence in each sample.

	Accuracy		
	SpeakerID RNN		**MNIST RNN**
Step	**Top-1**	**Top-5**	**Top-1**
1	82.04%	92.52%	98.13%
2	77.36%	90.28%	97.29%
3	74.16%	88.64%	97.56%
4*	**57.00%**	**78.05%**	**94.42%**

Table 4.3: **Accuracy results of the four-step quantization process on MNIST and SpeakerID RNNs**. (*) No OAR regularization applied.

From the table, we see that both RNNs successfully quantize. The MNIST RNN achieves an accuracy of -3.71% with respect to the baseline RNN (step 1). According to our research, we believe that this is the first time RNNs have been successfully quantized with binary activations, ternary parameters, no scaling of weights or activations, and no normalization. The arithmetic is also purely integer arithmetic, not fixed-point, without use of any right-shift operations—all CGGI-friendly attributes. From section 2.5.5, we see that our implementation is different from other works in this way. Research has not progressed down this path because it has not been necessary to eliminate right-shift operations. These are very computationally cheap integer operations that can be executed easily, which is why there is a plethora of research surrounding fixed-point quantization. Since we cannot perform fixed-point integer arithmetic with CGGI operations without expanding the number of ciphertexts and causing a large increase in inference latency in the process, it has been necessary to research this area and possibly achieve unfavourable results. For the MNIST RNN, the results are impressive. Although not mentioned in the table, increasing the input dimensions to 128x128, which increases the number of parameters to 8,480,768 and the number of timesteps to 128, also provides a good accuracy of 94.91%. In contrast, the SpeakerID RNN results are a little less favourable.

From the table, we can see that there is a consistent decrease in accuracy from step one through step four. For the MNIST RNN, the decrease is not as substantial. For the SpeakerID RNN, the decrease is larger—for a full evaluation, we obtain a decrease in top-1 accuracy from the baseline (step one) of -25.04%. However, it is worth noting that 57% is much greater than an accuracy of 0.08%, the accuracy obtained from guessing randomly. There is also a decrease in top-5 accuracy of -14.47%, which is less than the top-1 accuracy—a more promising result. Nevertheless, the results indicate a promising first-step towards pure integer quantization of models like the SpeakerID RNN, being on the order of around 12 million parameters. In addition, the primary focus of this thesis is to *enable* evaluation of large-scale RNNs on encrypted data, so we do not consider the lower accuracy of the SpeakerID RNN a major problem.

By analyzing the start and end validation accuracies of each step in the four-step process, we realize the importance of dividing the quantization process into steps. For the MNIST RNN, in step two, the validation accuracy increases from 79.29% to 98.69% after 68 epochs, an increase of **19.4%**. Given that the model from step one achieves 99.12% accuracy, by introducing the Signum activation function with `tanh` derivative, there is an automatic decrease in accuracy of 19.83%, which retraining in step two regains. This also shows the power of quantization-aware training—accuracy can be regained from an initial loss resulting from quantization. For the third step, the accuracy loss is much less. The validation accuracy starts from 97.29% by applying the changes made in step three, and increases to 99.18% after 24 epochs, an increase of **1.89%**. Images in the MNIST dataset are black and white, and when quantized to ones and zeroes, are still easily recognizable by the human eye. This could explain why the accuracy is already quite high when switching to step three. Finally, for step four, the accuracy increases from 62.11% to 94.67% after 300 epochs, a very large increase of **32.56%**. To compare the results of the four-step process with applying full quantization right after the initial full-precision run, we can combine steps two to four and apply them all at once, right after step one. The quantization achieves around 54% validation accuracy after 155 epochs, which is much less than the 94.42% validation accuracy obtained through the four-step process (around 40%), showing the effectiveness of this strategy.

The step-by-step results are even more interesting for the SpeakerID RNN. Step two begins with a validation accuracy of 37.71% and achieves 69.32%, a difference of **31.67%**. Step three begins with 55.21% and ends with 62.98%, a difference of **7.77%**. Finally, step four begins with 25.59% and ends with 42.79%, increasing by **17.2%**. When we perform steps two to four in one step, similar to the MNIST RNN, the model obtains a validation accuracy of 12.94%, a difference of **29.85%** from the validation accuracy using all steps. The step two and four differences are large, similar to the MNIST RNN evaluation. The step three difference is larger relative to the MNIST RNN, which is likely a cause of the more complicated input distribution from VoxCeleb. Thus, all four steps are important

and significantly increase the accuracy relative to a process where quantization occurs in one step, which the majority of quantization papers employ. This also offers a method to optimize the overall performance. By studying each step individually and increasing the accuracy as much as possible for each step, the overall accuracy after quantization would benefit.

In appendix G.4.1, we provide an analysis of applying the gradient control procedure from section 4.1.2. In summary, gradient control is crucial to enabling training of the SpeakerID RNN. It also aids in increasing the accuracy of the MNIST RNN. Our results indicate that exploding gradients do occur and that gradient control mitigates this problem.

4.4.2 RNNs Over Encrypted Data

In the following sections, we present the results of our evaluation of both the MNIST and SpeakerID RNNs over encrypted data. The results are obtained from running the models on machine B, using two A100 GPUs unless stated otherwise. Models over encrypted data and their equivalent evaluations over plaintext data are referred to as *encrypted* and *plaintext* models. MNIST RNN plaintext runs utilize the ModSignum function whereas SpeakerID RNN plaintext runs utilize the Signum function. We refer to inferences that predict the correct label as *correct* inferences and similarly, inferences that predict the incorrect label as *incorrect* inferences. We utilize some common metrics to measure the effectiveness of inference runs. For the end result of inferences, we measure top-1 and top-5 accuracy where top-5 refers to the accuracy of the correct label being in the top five predictions of the inference. On a layer level, we measure the mean absolute error (MAE) between the encrypted and plaintext pre-activations. We also measure the percent difference between the encrypted and plaintext activations. To perform this measurement, at each layer, we decrypt the pre-activations and activations and calculate the metric with respect to the equivalent pre-activations and activations in the plaintext model, running alongside the encrypted model. We average the values for all of the runs at the end. The plaintext model is deterministic meaning that it serves as a stable comparison to what the encrypted model should be outputting.

4.4.2.1 MNIST RNN Over Encrypted Data

We present the accuracy and latency results for running two different types of MNIST RNN models in table 4.4 over the entire MNIST test dataset (10,000 samples). Both models are executed in 6-bit precision. The first MNIST RNN model is the regular model from section 3.3, trained with 6-bit OAR regularization, while the second is the enlarged model mentioned in section 3.4.4, capable of evaluating upscaled 128×128 pixel images. We experiment with parameter sets 2 and 3 since we find they produce satisfactory results, without needing to increase the size of the parameters to set 1. Parameter set 2 is an ideal choice for 6-bit

Type of Model	Set ID	Top-1 Accuracy		Top-5 Accuracy		Average Latency (s)
		Encrypted	Plaintext	Encrypted	Plaintext	
Regular	2	90.86%	90.99%	99.63%	99.68%	4.86
	3	90.82%		99.61%		2.10
Enlarged	2	89.42%	92.27%	99.43%	99.65%	21.72
	3	87.56%		98.99%		10.26

Table 4.4: **Accuracy and latency measurements for various MNIST RNN inferences over encrypted data**. The regular model processes input images with 28×28 pixels, whereas the enlarged model processes images with 128×128 pixels. The input images to the enlarged model are upscaled versions of the original MNIST test set input images.

Type of Model	Set ID	Percent Difference			MAE
		RNN 0	RNN 1	Dense 0	Dense Out
Regular	2	0.00%	0.10%	1.94%	2.02
	3	0.56%	0.46%	5.72%	3.96
Enlarged	2	7.85%	11.42%	34.92%	53.06
	3	19.52%	20.80%	35.82%	53.38

Table 4.5: **Error metrics for each layer in various MNIST RNN inferences over encrypted data**. MAE refers to mean absolute error. See beginning of section 4.4.2 for information on these metrics.

precision according to [10]. We also experiment with parameter set 3, which yields more efficient computation, but can handle less amounts of consecutive operations than set 2.

We begin with the regular model. For parameter set 2, the encrypted and plaintext runs achieve top-1 accuracies of 90.86% and 90.99%, a difference of only 0.13%. The top-5 accuracies near 100% with an even smaller difference of 0.05%. We can see a possible reason for these excellent results in table 4.5, where we show the percent difference and MAE in each layer between the plaintext and encrypted run distributions. There is a 0% difference in the RNN 0 layer as well as a 0.10% difference in the RNN 1 layer between activations. This shows that the encrypted RNN, across all of its timesteps, outputs almost exactly the same activations as the plaintext RNN. With respect to execution runtime, we achieve a latency of 4.86 seconds per sample. By changing the parameter set to 3, we gain a decrease in latency by more than half to 2.10 seconds per sample while keeping the same accuracy metrics within 0.04%. As expected, the percent difference and MAE metrics are slightly worse, with the greatest difference being +3.78% in the Dense 0 layer. However, the impact on accuracy is negligible. Without using OAR regularization, the distributions move past 8-bit levels, which is the practical maximum supported by CGGI. Thus, OAR regularization is the key to reducing the accuracy and size of the parameters as a result, enabling much

more efficient and correct computation.

For the enlarged model, there is a greater difference between encrypted and plaintext run accuracies, with the maximum being -4.71% using parameter set 3. Respectively, table 4.5 shows an order of magnitude increase in error in all layers, with the parameter set 3 run experiencing nearly 20 times more error in the RNN 0 layer than the regular model run using set 3. It is clear that an increase in error in the RNN layers is the main contributor of worse performance. The error in the RNN 0 layer propagates to deeper layers such as RNN 1 and the rest, explaining why the rest of the layers also experience more error. Since the model is larger, there are more levelled FHE operations in each layer, which explains why there is an increase in error. The levelled operations in the MNIST RNN model are all additions and subtractions. To mitigate the error, it is possible to add intermediate bootstraps with the identity function in each dot-product operation to limit the maximum error. This comes at a cost of more latency (at least double since one intermediate bootstrap per dot-product translates to doubling the total amount of bootstraps). It is also possible to increase the size of the parameters as a last resort. Nevertheless, according to the literature, the model is the deepest RNN ever evaluated over encrypted data using CGGI, achieving a comparable accuracy and an impressive latency per sample.

Since the MNIST RNN model contains roughly two million parameters, with respect to the state-of-the-art described in table 2.2, we achieve the most efficient runtime. We summarize the differences between the MNIST RNN evaluations over encrypted data and the state-of-the-art in table G.3 in appendix G. We present the runtimes of several models evaluated over the encrypted MNIST dataset (except SHE RNN, which is evaluated over the Penn Treebank dataset) and estimate the number of parameters according to the evidence in their respective papers. The models we compare against have different numbers of parameters and thus, different reported runtimes. To normalize the runtimes, we extrapolate the model runtimes for each paper to match the execution of equivalent models with the same number of parameters as the MNIST RNN (around 1.91M). We extrapolate by dividing 1.9M parameters by each model's number of parameters, and multiplying by the reported runtimes. We assume the extrapolation is fairly correct considering the models in FHE-DiNN [13] and Concrete-DiNN [20] exhibit a linear relationship between the number of parameters and their associated runtimes for each of their models.

Our MNIST RNN model outperforms the state-of-the-art (REDsec [34]) latency by 8.3 times. Moreover, REDsec performs its experiments using eight GPUs while we use only two, thus making our effective latency even faster. Concrete-DiNN uses eight of the same GPUs we use (Nvidia A100), and our regular model implementation, with a 55% larger depth than NN-20, outperforms Concrete-DiNN by nearly 30 times. The NN-100 model has the closest depth to our enlarged model, yet it obtains an accuracy of only 83%, while our enlarged model achieves 87.56% accuracy and a 34 times lower latency. The regular and enlarged MNIST RNN models achieve a 1.11x and 5.24x larger depth than the current

Type of Model	Set ID	Top-1 Accuracy		Top-5 Accuracy		Latency (s) (100^{th} Sample)
		Encrypted	Plaintext	Encrypted	Plaintext	
GreySignum	1	30%	53%	41%	77%	1188
	2	28%		44%		504
Signum	2	28%	53%	39%	70%	531

Table 4.6: **Accuracy and latency measurements for various SpeakerID RNN inferences over encrypted data**. The measurements are taken for the first 100 samples of the VoxCeleb1 test dataset. The latency is recorded for the 100^{th} sample, which is 188 timesteps long.

Type of Model	Set ID	Percent Difference			
		RNN 0	RNN 1	SA Dense 0	SA Dense 1
GreySignum	1	5.94%	13.48%	1.61%	12.12%
	2	7.99%	14.98%	1.73%	13.19%
Signum	2	11.59%	17.19%	26.94%	15.00%

Table 4.7: **Error metrics for each layer in various SpeakerID inferences over encrypted data**. The measurements are taken for the first 100 samples of the VoxCeleb1 test dataset. The Set ID column contains the IDs of the parameter sets in table G.2. "SA" refers to self attention and "DP" refers to dot-product.

state-of-the-art for RNNs, the SHE RNN [78]. While the accuracy achieved by SHE is around 9% greater than the regular MNIST RNN accuracy, we achieve a latency nearly 3000 times smaller, with two stacked RNNs rather than one.

In summary, the MNIST RNN achieves comparable accuracy and at least 8.3 times better latency than the state-of-the-art. For the regular MNIST RNN, we observe that the difference between the distributions in each layer of the plaintext and encrypted runs is minimal, allowing the the difference between plaintext and encrypted accuracy to be within 0.13%. According to the literature, it is also the first time multi-layer, stacked RNNs of greater than 25 timesteps have been executed over encrypted data, while maintaining at least 128-bit security.

4.4.2.2 SpeakerID RNN Over Encrypted Data

We present the accuracy and latency results for running two different types of SpeakerID RNN models in table 4.4 over the VoxCeleb1 test dataset. Due to the large runtime latency of each sample and resource constraints, we show the results for the first 100 samples only. The length of sequences vary from 130 to 702 timesteps for the first 100 samples. We experiment with two types of SpeakerID RNN models. The first uses the GreySignum activation function from section 4.2.2 as the activation function for each layer, while the

second uses the regular Signum function. Since the SpeakerID RNN is wider and deeper than the MNIST RNN, we anticipate larger errors, which are shown in table 4.7. We utilize a precision of $p = 2^{11}$, or 11 bits, which has a domain that just about fits the largest pre-activation distribution in the Dense 0 layer. As a result, we experiment with parameter sets 1 and 2. We also experimented with parameter set 3, which resulted in too much error and a worse accuracy, so it is disregarded. Parameter sets 1 and 2 have ring dimensions of 2^{12} and 2^{11} respectively. Our precision parameter is equivalent to the ring dimension of parameter set 2 and half of set 1, thus the rounding intervals include one coefficient and two, respectively. In contrast, the precision parameter $p = 2^6$ used in the MNIST RNN tests yields a rounding interval of 2^6 and 2^5 for parameter sets 1 and 2 respectively, allowing much more tolerance for error in every query, and specifically for a query of 0. The Signum function is a very forgiving function with respect to error tolerance since any value that maps close to the center of the LUT, effectively has the whole domain of N as its rounding interval. Unfortunately, this is not true for values around 0. Thus, we expect the accuracy of the SpeakerID RNNs to be much less than the MNIST RNN with respect to the difference between plaintext and encrypted runs. In section 4.2.2, we introduced the GreySignum function to introduce this type of error in the model during training so it can augment itself around it. Thus, we also expect the GreySignum encrypted runs to be closer in accuracy and less in error with respect to the plaintext runs, than the Signum encrypted runs.

We begin with the GreySignum model. Using set 2, we obtain an accuracy of 28% after 100 inferences over encrypted data, which is -25% different from the plaintext run. For the top-5 accuracy, we achieve a higher accuracy of 44% over encrypted data, which is also -33% different from the plaintext run. Our expectations from the previous paragraph are correct—the accuracy is much lower, taking a hit because of the larger precision. It is evident why the authors of CGGI suggest a precision of 8-bits as being the maximum supported bit width for practicality. In table 4.7, the cause of this decrease in accuracy can be seen in the RNN and self attention layers. We do not include MAE metrics and results for the other layers since they appear to be dependent on the number of timesteps. The layers shown in 4.7 all appear to be independent of the number of timesteps. For reference, figure G.2 in appendix G shows the MAE and percent difference metrics for the rest of the layers with respect to timesteps for this same run, over most of the test dataset. For a more detailed discussion surrounding error propagation in RNNs as the number of timesteps increase, refer to appendix G.4.2. For the sequential layers like the RNNs and self attention dense layers, we calculate the percent difference for all the values across all of the timesteps. The GreySignum error appears to be around 8% in the first RNN and increases to 15% in the second RNN. However, for the Signum run, which uses the same parameter set 2, the error increases by 3.6% and 2.2% for each RNN layer respectively, causing a much larger increase of 25.2% in the self attention Dense 0 layer. The GreySignum function thus helps

mitigate the growth of error in the model as a result of the Signum function and higher precision, as anticipated. The top-1 accuracy remains the same, however, which could be a result of the use of a smaller subset of the the test dataset, comprising less than 2% of the total size of the test dataset. The positive difference in top-5 accuracy suggests this as well, since top-5 accuracy is the measure of how high a model places less confident, correct labels. We predict that a GreySignum evaluation would outperform a Signum evaluation over a larger portion of the dataset. We only had the resources and time to evaluate the GreySignum run over 92% of the test dataset and plan to do the same for the Signum run in the future. In fact, the GreySignum evaluation achieves an encrypted accuracy of 30.57% over the first 7382 samples of the test dataset.

For the GreySignum run using parameter set 1, there is a clear decrease in all error metrics in table 4.7 with respect to the GreySignum run using parameter set 2, as well as an increase in top-1% accuracy. Given more samples, and judging by the better error metrics, we anticipate the accuracy to rise, however, at the cost of much more latency. For the runs with parameter set 2, we achieve latencies of 504 and 531 seconds. With parameter set 1, the latency increases to 1188 seconds. Thus, set 2 is preferred. More research is required into finding a way to make OAR regularization work for these types of networks. If we can successfully use OAR regularization, then the accuracy metrics will go up and the error will decrease, as shown by the MNIST RNN evaluation. The latencies we achieve are measured for the 100^{th} run, which is a sequence of 188 timesteps. We observe that increase in latency is linear with respect to the number of timesteps. It is important to note that we do not perform any model pipelining for the RNN and self attention dense layers. We perform parallelization on the bootstrapping front for each timestep, but not for multiple timesteps at once. If we were to scale the number of GPUs, we could achieve incredible speeds, suggested by the benchmarks in appendix G.5. The number of bootstraps in sample 100, according to our model and implementation, is 652,152. The number of bootstraps in the attention dot-product layer is $(ts/2) \cdot 768 \cdot 4$ according to our architecture, since we are using our novel multiplication method where the number of bootstraps is equal to the number of multiplications in the layer. For 188 timesteps, there are 288,768 bootstraps in the multiplication layer, spanning 44% of the total number of bootstraps, which is the main cause of the increased latency in our results with respect to the MNIST RNN. If we could parallelize the total number of these bootstraps using multiple GPUs, the latency would be minimized. For example, in figure G.4 in appendix G, for parameter set 2, the lower bound for latency is around 100 ciphertexts per A100 GPU (since the A100 contains around 100 streaming multiprocessors), running at 75ms. Thus, to enable complete parallelization, we would need $288,768/100 \approx 2888$ A100s, which we could round up to the next power-of-two for simplicity and more tolerance for more timesteps. With 4096 A100s, we could run the attention layer within 100ms, decreasing the entire inference runtime by an order of magnitude to an estimated 25 seconds. With pipelining, we can make the total runtime

of the sequential layers relative to the lower bound (75ms) times the number of timesteps. Thus, our implementation is linearly scalable with respect to decreasing latency and number of GPUs, a result of the naive bootstrapping parallelization in Concrete.

With respect to the state-of-the-art, we can only compare the SpeakerID RNN model with the one layer, 300 unit, 25 timesteps RNN in SHE [78]. Firstly, our model contains multiple layers of RNNs, each with 768 units. Furthermore, inference is performed over sequences of a minimum length of 130 timesteps, while the highest number of timesteps resulting in an accurate encrypted response is 426. Generally, our model performs well for models within the 150-250 timestep range, as seen in figure G.1. SHE achieves remarkable accuracy at the cost of large latency, which is less desired for RNNs. Our model is the first RNN that includes a form of attention that can be evaluated over encrypted data. If we were to use the method for ciphertext-ciphertext multiplication in [20], our latency in the multiplication layer would double, since the number of bootstraps would double, which would increase the total latency by at least 44%. If we were to use fixed-point signed integer arithmetic methods in other papers such as REDsec [34], SHE [78], and TAPAS [121], our accuracy would definitely increase, but our latency would increase by orders of magnitude as well. If we were to use quantization methods such as those defined in FHE-DiNN [13], we would not achieve accuracy near to the one we have already achieved since that type of quantization does not work for RNNs. Thus, we believe that we have achieved the current state-of-the-art in large-scale, non-interactive RNN evaluation over encrypted data with respect to balancing good accuracy and low latency. We do not compare to CKKS methods, such as [101], since the evaluated RNNs are segmented and we require that latency is more important than throughput of inferences. However, we recommend CKKS implementations for small-scale, non-latency restricted RNNs with limited timesteps (up to 5, if you want to stack RNNS) due to the precise floating-point nature of CKKS and its packing structure.

4.5 Summary

Converting RNNs to CGGI-compatible runs requires various contributions in different fields such as RNN quantization and unique operations specific to CGGI. In this chapter, we built on the key technique that solves the accumulation problem in RNNs, namely OAR regularization, to train and convert two large-scale RNNs: (1) the MNIST RNN, a fixed-size model with about 2M parameters, and (2) the SpeakerID RNN, a roughly 12M parameter model that incorporates attention to enable processing of variable-length sequences.

The first contribution we made was the novel four-step quantization process, which OAR regularization relies upon. This process is a quantization-aware training method to train RNNs into low-precision, binary-ternary networks that operate under modular, signed integer arithmetic. It quantizes the activations to binary numbers in $\{-1, +1\}$ and the network parameters to ternary numbers in $\{-1, 0, +1\}$. In order to perform effective

ternarization, we developed a new method of determining the ternarization threshold in section 4.1.1, that expands upon existing methods without scaling the parameters to fixed-point integers (not supported by CGGI natively). To mitigate exploding gradients caused by binary and ternary quantization of RNNs, we proposed a process that scales the RNN gradients during backpropagation and not during the forward step in section 4.1.2, which keeps the network operating under CGGI-compatible arithmetic. Finally, in section 4.1.3, we outlined the four-step quantization process which, briefly, (1) trains a network ready for quantization, then (2) retrains the network under the Signum activation function, (3) retrains the network again using ternarized inputs, and finally (4) quantizes the parameters and then retrains with OAR regularization (in the case of the MNIST RNN) or without (in the case of the SpeakerID RNN). We presented our quantization results for both RNNs in sections 4.4.1, where we demonstrated successful quantization of both RNNs. The MNIST RNN achieves a final accuracy of around 94%, which we showed is 40% better than applying the quantization strategy all in one step. The SpeakerID RNN achieves a full evaluation accuracy of 57%, which, similarly, is around 30% better than applying all quantization steps in one step, demonstrating the effectiveness of the four-step process. Since the literature heavily favours fixed-point quantization, we are among some of the first works to investigate and successfully implement this type of pure integer quantization without any normalization, floating-point operations, and shifting operations.

Our second contribution was a set of CGGI operations that enabled us to convert the quantized RNNs into CGGI-compatible RNNs. In order to implement attention over encrypted data, it is necessary to perform ciphertext-ciphertext multiplication. Unfortunately, this type of multiplication is not natively supported by CGGI between LWE ciphertexts. In section 4.2.1, we presented the various methods to multiply two ciphertexts that exist and showed that the best way involves a set of two PBS operations, which when scaled to many multiplications in scenarios such as the evaluation of attention layers, becomes a bottleneck. Due to the binary nature of the multiplications in our SpeakerID RNN, we proposed a new type of multiplication method that requires only one PBS operation. Another issue in the CGGI scheme is querying the "0" value in PBS operations. The "0" value is the border between the negacyclic region of the LUT and the regular region, which becomes a problem when the number of consecutive operations on a ciphertext increases and the underlying FHE error increases as a result. Unfortunately, it causes the PBS operation to output incorrect signs. To mitigate this error, we introduce it to the training process, allowing the model to adjust itself as it becomes aware of this error. We introduced the GreySignum activation function which randomly assigns "+1" or "-1" to queries of "0", similar to noisy queries in the LWE domain. Finally, in section 4.2.3, we discussed a way to divide the number of operations in the final layer to preserve precision and eliminate the possibility of overflow. We presented our results in sections 4.4.2, analyzing the accuracy and latency measurements for both RNNs. We found that the MNIST RNN achieves a latency of only

about 2 seconds using two Nvidia A100 GPUs, with an inference accuracy over encrypted data that is negligibly different than the same inference over plaintext data. We showed that OAR regularization is the key to obtaining these remarkable results. Consequently, we obtain new state-of-the-art results for latency while being the first to implement a multi-layer, stacked RNN, capable of performing inference over encrypted sequences much greater than 25 timesteps in length, while maintaining 128-bit security. With respect to the SpeakerID RNN model, the results demonstrate that it is the first RNN model with attention to be successfully performed over encrypted data. Additionally, it is the largest and longest RNN evaluated over encrypted data. While more work is necessary to increase the accuracy, we showed that massive parallelization of about 4096 GPUs can theoretically decrease the latency of 504 seconds by an order of magnitude to an estimated 25 seconds.

Through our method of quantization and the proposed CGGI methods, we enable the practical evaluation of large-scale, recurrent neural networks over encrypted data in a non-interactive setting. Our results, while promising, suggest that there is room for improvement. We discussed ways of improving some of the methods throughout our discussion and provide general future directions in section 5.2.

Chapter 5

Concluding Remarks

5.1 Summary Of Contributions

In this thesis, we set out with the goal of enabling the evaluation of large-scale, recurrent neural networks over encrypted data. Some of our requirements included evaluation in the non-interactive setting and the use of parallelization to decrease the latency of inferences. Due to the larger depth of RNNs with respect to other neural networks, we chose the CGGI [25] mathematical foundation for FHE encryption since it possesses the most efficient bootstrapping method that can simultaneously evaluate any function while refreshing the ciphertexts concurrently. It also provides a user-friendly GPU implementation in its flagship library, Concrete [21]. In order to use the CGGI scheme, it was necessary to quantize RNNs, which is a challenge in and of itself due to exploding gradients and large accumulations in pre-activation distributions. Most of the work that surrounds the quantization of RNNs recommends normalization, fixed-point precision, and large accumulation precision, all of which are methods not supported by the CGGI scheme. The CGGI scheme only supports modular, signed integer arithmetic, without the ability to perform right-shift operations. Thus, one of the contributions of this thesis was a **four-step process to quantize RNNs**. In our method, we quantized RNNs to binary activations and ternary parameters. The four-step process solves the exploding gradients problem by proposing a **novel method to scale the gradients during backpropagation** but not during the forward step. The four-step process also proposes a **general method of finding a ternarization threshold**, which builds upon existing work. However, when quantizing to integers, the accumulation problem still exists. Specifically, due to the modular nature of signed integers, large accumulations can overflow into the negative domain and cause activation functions to output incorrect values.

In response, the second major contribution of this thesis was the **Overflow-Aware Activity Regularizer (OAR)**. By observing the properties of the Signum function, we discovered that overflowing integers an additional time, either by increasing or decreasing the already overflowed values, reverses the sign of the overflowed integer, causing the Signum

function to output the correct sign. Therefore, given a specified bit precision and a model that has finished the third-step in the four-step quantization process, OAR regularization pushes pre-activations to the *correct* regions through minimizing the regularization loss objective. Our results show that the pre-activations successfully move to the correct regions in the evaluation of a novel RNN architecture we refer to as the **MNIST RNN**. The MNIST RNN consists of two vanilla RNNs, and two dense layers, evaluated over the MNIST dataset. It also consists of around two million parameters. Our results indicate that OAR regularization is necessary to maintain low-precision evaluation, which allows us to use more efficient CGGI security parameters as a by-product.

We also evaluate a novel RNN for the speaker identification task, which we refer to as the **SpeakerID RNN**, over the VoxCeleb1 dataset [90]. This RNN is similar to the MNIST RNN, with the only difference being the **addition of a multi-headed, self attention layer** between the second RNN and the first dense layer. This model contains around twelve million parameters and is significantly harder to quantize than the MNIST RNN. We quantize the RNN successfully using the four-step process. In order to convert these types of quantized RNNs, our final contributions include the proposal of several operations specific to CGGI. To enable attention over encrypted data for the first time according to the literature, we propose a **novel ciphertext-ciphertext multiplication operation that reduces the number of necessary PBS operations from two to one**. To reduce the error caused by PBS evaluations of the value "0", we introduce the **GreySignum activation function** as a noise inducer to the RNN that allows the RNN to mitigate the error during training. Finally, to reduce the required precision of the final layer, we propose a method to **decompose the dot-products into several smaller dot-products**, thus preventing the occurrence of overflow.

The combination of these methods allows us to achieve a new state-of-the-art latency of 2 seconds using two Nvidia A100 GPUs when performing inference with the MNIST RNN, a two million parameter and 31-layer deep model, over encrypted data. Due to OAR regularization, the MNIST RNN outputs an accuracy over encrypted data that is negligibly different than the accuracy for the same inference over plaintext data. We also provide the first successful implementation of multi-layered, stacked RNNs over sequences greater than 25 timesteps in length, while achieving two orders of magnitude better latency compared to the current state-of-the-art for RNN evaluation over encrypted data, SHE [78]. We also successfully evaluate the SpeakerID RNN over encrypted data, which is considerably larger than the MNIST RNN, at twelve million parameters in size. This evaluation is the first time RNNs with attention layers have been evaluated over encrypted data. We achieve remarkable latency and provide a theoretical argument for scaling the number of GPUs and thus, reducing the latency by several orders of magnitude.

Using our proposed methods, practitioners can experiment with executing larger RNNs over encrypted data. There is definitely much more work to be done with respect to main-

taining accuracy, however, this work offers a good first step towards the future goal to one day be able to operate any type of RNN, of any length, over encrypted data.

5.2 Future Directions

In this thesis, we presented a novel four-step quantization process that appears to work well for RNNs. However, we did not experiment with other types of networks, such as CNNs, which could be of great interest for other applications such as image recognition. Since the parallelization of the bootstrapping procedure provided by Concrete is fundamental to our gains in reduced latency, it would dramatically reduce the latency in other types of models as well. Alongside our quantization process, OAR regularization was also fundamental to enabling accurate as well as fast executions of RNNs. It is worth researching activation functions other than the Signum, such as ReLU, to see if there exists a similar method that can mitigate overflow in the same way. In other words, is there a general method that uses overflow to its advantage rather than assuming it is a disadvantage?

A natural extension to this research is applying our methods to more complicated gated RNNs such as GRU and LSTM, since these are the main types of RNNs used in most applications. When creating the RNNs used in this research, we modeled their structure after common LSTM architectures for speech recognition, in anticipation of being able to perform speech recognition. Instead, we focused on performing a simpler task of speaker identification. Two additional avenues of investigation would be to focus on speaker diarisation and speech recognition.

In our experimentation, we were unable to perform OAR regularization for the SpeakerID RNN model. This is a necessary next-step in future research. It is possible that attention is the cause, for which we would need to investigate the propagation of gradients and analyze why it may be resistant to this type of regularization. Similarly, there was a larger loss in accuracy when quantizing the SpeakerID RNN as opposed to the MNIST RNN, which we predict may be caused by attention. Further investigation on the impacts and interaction of the attention layer with our method of quantization is a necessary next step.

Finally, with respect to attention as well, our novel ciphertext-ciphertext multiplication method, while more efficient than the standard multiplication method, still contributes to 44% of the latency in each inference due to the vast number of bootstraps in the multiplication layer with respect to the rest of the network. Research needs to be directed towards finding an efficient, native multiplication method in CGGI that does not cause a magnification of noise. This would dramatically reduce the added latency as a result of attention and enable the evaluation of popular transformer architectures using CGGI.

Bibliography

[1] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *ArXiv* 1603.04467 (2016).

[2] Abbas Acar et al. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Computing Surveys* 51.4 (2018), Article 79. ISSN: 0360-0300. DOI: 10.1145/3214303. URL: https://doi.org/10.1145/3214303.

[3] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9.3 (2015). ISSN: 1862-2976. DOI: 10.1515/jmc-2015-0016. URL: https://dx.doi.org/10.1515/jmc-2015-0016.

[4] Arash Ardakani et al. "Learning Recurrent Binary/Ternary Weights". In: *ArXiv* abs/1809.11086 (2019). DOI: 10.48550/ARXIV.1809.11086. URL: https://arxiv.org/abs/1809.11086.

[5] Frederik Armknecht et al. "A Guide to Fully Homomorphic Encryption". In: *IACR Cryptology ePrint Archive* 2015.1192 (2015). URL: https://eprint.iacr.org/2015/1192.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *ArXiv* 1607.06450 (2016).

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ArXiv* (2014). DOI: 10.48550/ARXIV.1409.0473. URL: https://arxiv.org/abs/1409.0473.

[8] Maya Bakshi and Mark Last. "CryptoRNN - Privacy-Preserving Recurrent Neural Networks Using Homomorphic Encryption". In: *Cyber Security Cryptography and Machine Learning*. Ed. by Shlomi Dolev et al. Be'er Sheva, Israel: Springer International Publishing, 2020, pp. 245–253. ISBN: 978-3-030-49785-9.

[9] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation". In: *ArXiv* 1308.3432 (2013).

[10] Loris Bergerat et al. “Parameter Optimization & Larger Precision for (T) FHE”. In: *Cryptology ePrint Archive* 2022.704 (2022). Version 20220620:172436. URL: https://eprint.iacr.org/archive/2022/704/20220620:172436.

[11] Christopher M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. New York: Springer, 2006. ISBN: 978-0-387-31073-2.

[12] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. “Evaluating 2-DNF Formulas on Ciphertexts”. In: *Theory of Cryptography*. Ed. by Joe Kilian. Springer Berlin Heidelberg, 2005, pp. 325–341. ISBN: 978-3-540-30576-7.

[13] Florian Bourse et al. “Fast Homomorphic Evaluation of Deep Discretized Neural Networks”. In: *Advances in Cryptology – CRYPTO 2018*. Ed. by Hovav Shacham and Alexandra Boldyreva. Cham: Springer International Publishing, 2018, pp. 483–512. ISBN: 978-3-319-96878-0.

[14] Zvika Brakerski. “Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP”. In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*. Springer-Verlag, 2012, pp. 868–886. DOI: 10.1007/978-3-642-32009-5_50. URL: https://doi.org/10.1007/978-3-642-32009-5_50.

[15] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325. DOI: 10.1145/2090236.2090262. URL: https://doi.org/10.1145/2090236.2090262.

[16] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *ArXiv* 2005.14165 (2020).

[17] Gizem Selcan Cetin et al. *cuFHE: CUDA-accelerated Fully Homomorphic Encryption Library*. 2018. URL: https://github.com/vernamlab/cuFHE.

[18] Kumar Chellapilla, Sidd Puri, and Patrice Simard. “High Performance Convolutional Neural Networks for Document Processing”. English. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by Lorette Guy. La Baule, France: Suvisoft, Oct. 2006. URL: https://inria.hal.science/inria-00112631.

[19] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8.

[20] Ilaria Chillotti, M. Joye, and Pascal Paillier. "Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks". In: *IACR Cryptology ePrint Archive* 2021.91 (2021). DOI: https://doi.org/10.1007/978-3-030-78086-9_1.

[21] Ilaria Chillotti et al. "CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE". In: *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. Vol. 15. 2020.

[22] Ilaria Chillotti et al. "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds". English. In: *Advances in Cryptology – ASIACRYPT 2016. ASIACRYPT 2016. Lecture Notes in Computer Science*. Ed. by Takagi T. Cheon J.H. Vol. 10031 LNCS. Hanoi, Vietnam: Springer Verlag, 2016, pp. 3–33. DOI: 10.1007/978-3-662-53887-6_1. URL: https://hal-cea.archives-ouvertes.fr/cea-01832762.

[23] Ilaria Chillotti et al. "Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE". In: *Advances in Cryptology – ASIACRYPT 2021*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Singapore, Singapore: Springer International Publishing, 2021, pp. 670–699. DOI: 10.1007/978-3-030-92078-4_23. URL: https://link.springer.com/chapter/10.1007%2F978-3-030-92078-4_23.

[24] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption Library*. 2016. URL: https://tfhe.github.io/tfhe/.

[25] Ilaria Chillotti et al. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *Journal of Cryptology* 33.1 (Jan. 2020), pp. 34–91. ISSN: 1432-1378. DOI: 10.1007/s00145-019-09319-x. URL: https://doi.org/10.1007/s00145-019-09319-x.

[26] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *ArXiv* 1406.1078 (2014).

[27] European Commission. *General Data Protection Regulation (2016/679)*. Apr. 2016. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN#d1e40-1-1.

[28] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: training deep neural networks with binary weights during propagations". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems*. Vol. 2. Montreal, Canada: MIT Press, 2015, pp. 3123–3131. URL: https://dl.acm.org/doi/10.5555/2969442.2969588.

[29] Matthieu Courbariaux et al. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *ArXiv* 1602.02830 (2016). DOI: 10.48550/ARXIV.1602.02830.

[30] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *ArXiv* 1810.04805 (2019).

[31] Ruizhou Ding et al. “Regularizing Activation Distribution for Training Binarized Deep Networks”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 11400–11409.

[32] Léo Ducas and Daniele Micciancio. “FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second”. In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Springer Berlin Heidelberg, 2015, pp. 617–640. ISBN: 978-3-662-46800-5.

[33] Junfeng Fan and Frederik Vercauteren. “Somewhat Practical Fully Homomorphic Encryption”. In: *IACR Cryptology ePrint Archive* 2012.144 (2012). URL: http://eprint.iacr.org/2012/144.

[34] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. “REDsec: Running Encrypted DNNs in Seconds”. In: *IACR Cryptology ePrint Archive* 2021.1100 (2021).

[35] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251. URL: https://doi.org/10.1007/BF00344251.

[36] Steven D. Galbraith. “Elliptic Curve Paillier Schemes”. In: *Journal of Cryptology* 15.2 (Jan. 2002), pp. 129–138. ISSN: 1432-1378. DOI: 10.1007/s00145-001-0015-6. URL: https://doi.org/10.1007/s00145-001-0015-6.

[37] J. G. Garson. “The Metric System of Identification of Criminals, as used in Great Britain and Ireland”. In: *The Journal of the Anthropological Institute of Great Britain and Ireland* 30 (1900), pp. 161–198. ISSN: 09595295. DOI: 10.2307/2842627. URL: http://www.jstor.org/stable/2842627.

[38] Craig Gentry. “A fully homomorphic encryption scheme”. PhD Thesis. Stanford University, 2009.

[39] Craig Gentry and Shai Halevi. “Implementing Gentry’s Fully-Homomorphic Encryption Scheme”. In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Springer Berlin Heidelberg, 2011, pp. 129–148. ISBN: 978-3-642-20465-4.

[40] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Springer Berlin Heidelberg, 2013, pp. 75–92. ISBN: 978-3-642-40041-4.

[41] Amir Gholami et al. “A Survey of Quantization Methods for Efficient Neural Network Inference”. In: *ArXiv* 2103.13630 (2021). DOI: 10.48550/arXiv.2103.13630.

[42] Ran Gilad-Bachrach et al. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Balcan Maria Florina and Q. Weinberger Kilian. Vol. 48. PMLR, 2016, pp. 201–210. URL: http://proceedings.mlr.press/v48/gilad-bachrach16.html.

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

[44] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *ArXiv* 1308.0850 (2013). URL: http://arxiv.org/abs/1308.0850.

[45] Alex Graves. "Sequence Transduction with Recurrent Neural Networks". In: *ArXiv* 1211.3711 (2012). URL: https://ui.adsabs.harvard.edu/abs/2012arXiv1211.3711G.

[46] Alex Graves et al. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks". In: *Proceedings of the 23rd international conference on Machine learning*. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, pp. 369–376. DOI: 10.1145/1143844.1143891. URL: https://doi.org/10.1145/1143844.1143891.

[47] Antonio Guimarães, Edson Borin, and Diego F. Aranha. "MOSFHET: Optimized Software for FHE over the Torus". In: *IACR Cryptology ePrint Archive* 2022.515 (2022). URL: https://ia.cr/2022/515.

[48] K. He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778. ISBN: 1063-6919. DOI: 10.1109/CVPR.2016.90.

[49] Qinyao He et al. "Effective Quantization Methods for Recurrent Neural Networks". In: *ArXiv* 1611.10176 (2016). URL: https://arxiv.org/abs/1611.10176.

[50] Y. He et al. "Streaming End-to-end Speech Recognition for Mobile Devices". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 6381–6385. ISBN: 2379-190X. DOI: 10.1109/ICASSP.2019.8682336.

[51] D. O. Hebb. *The organization of behavior; a neuropsychological theory*. Oxford, England: Wiley, 1949. URL: https://doi.org/10.1002/sce.37303405110.

[52] Geoffrey Hinton. "The forward-forward algorithm: Some preliminary investigations". In: *ArXiv* 2212.13345 (2022).

[53] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural Comput.* 18.7 (2006), pp. 1527–1554. ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.7.1527. URL: https://doi.org/10.1162/neco.2006.18.7.1527.

[54] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[55] J J Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences* 79.8 (Apr. 1982), pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554. URL: https://doi.org/10.1073/pnas.79.8.2554.

[56] Lu Hou and James Tin-Yau Kwok. "Loss-aware Weight Quantization of Deep Networks". In: *ArXiv* 1802.08635 (2018). URL: https://arxiv.org/abs/1802.08635.

[57] Lu Hou, Quanming Yao, and James Tin-Yau Kwok. "Loss-aware Binarization of Deep Networks". In: *ArXiv* 1611.01600 (2017). URL: https://arxiv.org/abs/1611.01600.

[58] Lu Hou et al. "Normalization Helps Training of Quantized LSTM". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2021, pp. 7346–7356.

[59] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *ArXiv* 1502.03167 (2015). DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[60] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713. ISBN: 2575-7075. DOI: 10.1109/CVPR.2018.00286.

[61] Marc Joye. "Guide to Fully Homomorphic Encryption over the [Discretized] Torus". In: *IACR Cryptology ePrint Archive* 2021.1402 (2021). URL: https://eprint.iacr.org/2021/1402.

[62] Henry J. Kelley. "Gradient Theory of Optimal Flight Paths". In: *ARS Journal* 30.10 (1960), pp. 947–954. DOI: 10.2514/8.5282. URL: https://arc.aiaa.org/doi/abs/10.2514/8.5282.

[63] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. *CIFAR-10 (Canadian Institute for Advanced Research)*. URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[65] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units". In: *ArXiv* 1504.00941 (2015). URL: https://ui.adsabs.harvard.edu/abs/2015arXiv150400941L.

[66] Yann LeCun, Corinna Cortes, and C. J. Burges. *MNIST handwritten digit database*. 2010. URL: http://yann.lecun.com/exdb/mnist.

[67] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

[68] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[69] Yann LeCun et al. "Handwritten digit recognition: applications of neural network chips and automatic learning". In: *IEEE Communications Magazine* 27.11 (1989), pp. 41–46. ISSN: 1558-1896. DOI: 10.1109/35.41400.

[70] E. H. Lee et al. "LogNet: Energy-efficient neural networks using logarithmic computation". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017, pp. 5900–5904. ISBN: 2379-190X. DOI: 10.1109/ICASSP.2017.7953288.

[71] Michela Lezzi. "Practical Privacy-Preserving Data Science With Homomorphic Encryption: An Overview". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 3979–3988. DOI: 10.1109/BigData50022.2020.9377989.

[72] Fengfu Li and Bin Liu. "Ternary Weight Networks". In: *ArXiv* 1605.04711 (2016). DOI: 10.48550/ARXIV.1605.04711. URL: https://arxiv.org/abs/1605.04711.

[73] Jian Li and Raziel Álvarez. "On the quantization of recurrent neural networks". In: *ArXiv* 2101.05453 (2021). DOI: 10.48550/arXiv.2101.05453.

[74] Tailin Liang et al. "Pruning and quantization for deep neural network acceleration: A survey". In: *Neurocomputing* 461 (Oct. 2021), pp. 370–403. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2021.07.045. URL: https://www.sciencedirect.com/science/article/pii/S0925231221010894.

[75] Aristidis Likas, Nikos Vlassis, and Jakob J. Verbeek. "The global k-means clustering algorithm". In: *Pattern Recognition* 36.2 (Feb. 2003), pp. 451–461. ISSN: 0031-3203. DOI: 10.1016/S0031-3203(02)00060-2. URL: https://www.sciencedirect.com/science/article/pii/S0031320302000602.

[76] Zhouhan Lin et al. "Neural Networks with Few Multiplications". In: *ArXiv* 1510.03009 (2016). DOI: 10.48550/ARXIV.1510.03009. URL: https://arxiv.org/abs/1510.03009.

[77] Seppo Linnainmaa. "Taylor expansion of the accumulated rounding error". In: *BIT Numerical Mathematics* 16.2 (June 1976), pp. 146–160. ISSN: 1572-9125. DOI: 10.1007/BF01931367. URL: https://doi.org/10.1007/BF01931367.

[78] Qian Lou and Lei Jiang. "SHE: A Fast and Accurate Deep Neural Network for Encrypted Data". In: *NeurIPS 2019*. Ed. by H. Wallach et al. Vol. 32. Advances in Neural Information Processing Systems. 2019. URL: https://proceedings.neurips.cc/paper/2019/file/56a3107cad6611c8337ee36d178ca129-Paper.pdf.

[79] D. Luebke. "CUDA: Scalable parallel programming for high-performance scientific computing". In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 2008, pp. 836–838. ISBN: 1945-8452. DOI: 10.1109/ISBI.2008.4541126.

[80] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13190-5.

[81] Chiara Marcolla et al. "Survey on Fully Homomorphic Encryption, Theory, and Applications". In: *Proceedings of the IEEE* 110.10 (2022), pp. 1572–1609. ISSN: 1558-2256. DOI: 10.1109/JPROC.2022.3205665.

[82] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Computational Linguistics* 19.2 (1993), pp. 313–330. URL: https://aclanthology.org/J93-2004.

[83] Paulo Martins, Leonel Sousa, and Artur Mariano. "A Survey on Fully Homomorphic Encryption: An Engineering Perspective". In: *ACM Computing Surveys* 50.6 (2017), Article 83. ISSN: 0360-0300. DOI: 10.1145/3124441. URL: https://doi.org/10.1145/3124441.

[84] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[85] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. "Regularizing and optimizing LSTM language models". In: *ArXiv* 1708.02182 (2017).

[86] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: The MIT Press, 1969. ISBN: 978-0-262-63022-1.

[87] Asit K. Mishra et al. "WRPN: Wide Reduced-Precision Networks". In: *ArXiv* 1709.01134 (2017). DOI: 10.48550/ARXIV.1709.01134.

[88] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.

[89] Morris Dworkin et al. "Advanced Encryption Standard (AES)". In: *Federal Information Processing Standards (NIST FIPS)* (Nov. 2001). DOI: 10.6028/NIST.FIPS.197.

[90] Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. "VoxCeleb: A Large-Scale Speaker Identification Dataset". In: *Interspeech*. Stockholm, Sweden, 2017, pp. 2616–2620. DOI: 10.21437/Interspeech.2017-950.

[91] Arsha Nagrani et al. "Voxceleb: Large-scale speaker verification in the wild". In: *Computer Speech & Language* 60 (Mar. 2020). ISSN: 0885-2308. DOI: 10.1016/j.csl.2019.101027. URL: https://www.sciencedirect.com/science/article/pii/S0885230819302712.

[92] Christopher Olah. *Understanding LSTM Networks*. Personal Blog. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[93] Bogdan Opanchuk et al. *NuFHE: A GPU implementation of fully homomorphic encryption on torus*. 2019. URL: https://nufhe.readthedocs.io/en/latest/.

[94] OpenAI. "GPT-4 Technical Report". In: *ArXiv* 2303.08774 (2023).

[95] Joachim Ott et al. "Recurrent Neural Networks With Limited Numerical Precision". In: *ArXiv* 1611.07065 (2016). DOI: 10.48550/ARXIV.1608.06902.

[96] P. J. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 1558-2256. DOI: 10.1109/5.58337.

[97] Adam Paszke et al. "PyTorch: an imperative style, high-performance deep learning library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019, Article 721.

[98] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. ISSN: 1532-4435.

[99] R. Podschwadt et al. "A Survey of Deep Learning Architectures for Privacy-Preserving Machine Learning with Fully Homomorphic Encryption". In: *IEEE Access* (2022). ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3219049.

[100] Robert Podschwadt and Daniel Takabi. "Classification of Encrypted Word Embeddings using Recurrent Neural Networks". In: *Proceedings of the PrivateNLP 2020: Workshop on Privacy in Natural Language Processing - Colocated with WSDM 2020*. Houston, Texas, Feb. 2020, pp. 27–31. URL: https://ceur-ws.org/Vol-2573/PrivateNLP_Paper3.pdf.

[101] Robert Podschwadt and Daniel Takabi. "Non-interactive Privacy Preserving Recurrent Neural Network Prediction with Homomorphic Encryption". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. Sept. 2021, pp. 65–70. ISBN: 2159-6190. DOI: 10.1109/CLOUD53861.2021.00019. URL: https://ieeexplore.ieee.org/abstract/document/9582263.

[102] Robert Podschwadt, Daniel Takabi, and Peizhao Hu. "SoK: Privacy-preserving Deep Learning with Homomorphic Encryption". In: *ArXiv* 2112.12855 (Dec. 2021). URL: https://arxiv.org/pdf/2112.12855.pdf.

[103] Haotong Qin et al. "Binary Neural Networks: A Survey". In: *Pattern Recognition* 105 (2020). URL: https://doi.org/10.1016/j.patcog.2020.107281.

[104] Alec Radford et al. "Improving Language Understanding by Generative Pre-Training". In: *OpenAI Blog* (2018). URL: https://openai.com/research/language-unsupervised.

[105] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: *OpenAI Blog* (2019). URL: https://openai.com/research/better-language-models.

[106] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 873–880. DOI: 10.1145/1553374.1553486. URL: https://doi.org/10.1145/1553374.1553486.

[107] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Springer International Publishing, 2016, pp. 525–542. ISBN: 978-3-319-46493-0.

[108] Christian Rechberger and Roman Walch. "Privacy-Preserving Machine Learning Using Cryptography". In: *Security and Artificial Intelligence: A Crossdisciplinary Approach*. Ed. by Lejla Batina et al. Springer International Publishing, 2022, pp. 109–129. ISBN: 978-3-030-98795-4. DOI: 10.1007/978-3-030-98795-4_6. URL: https://doi.org/10.1007/978-3-030-98795-4_6.

[109] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *37th ACM STOC*. Baltimore, MD, USA: Association for Computing Machinery, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603. URL: https://doi.org/10.1145/1060590.1060603.

[110] Oded Regev. "The Learning with Errors Problem (Invited Survey)". In: *2010 IEEE 25th Annual Conference on Computational Complexity*. June 2010, pp. 191–204. ISBN: 1093-0159. DOI: 10.1109/CCC.2010.26.

[111] Eric Rescorla. *RFC 8446: The Transport Layer Security (TLS) Protocol*. USA, 2018. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[112] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation, Academia Press* (1978), pp. 169–179.

[113] Ronald L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[114] Babak Rokh, Ali Azarpeyvand, and Ali Reza Khanteymoori. "A Comprehensive Survey on Model Quantization for Deep Neural Networks". In: *ArXiv* 2205.07877 (2022). DOI: 10.48550/ARXIV.2205.07877. URL: https://arxiv.org/abs/2205.07877.

[115] Frank Rosenblatt. *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*. Washington: Spartan Books, 1962.

[116] Frank Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65 (1958), pp. 386–408. ISSN: 1939-1471. DOI: 10.1037/h0042519.

[117] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[118] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y.

[119] H. Sak et al. "Learning acoustic frame labeling for speech recognition with recurrent neural networks". In: *ICASSP*. South Brisbane, QLD, Australia, Apr. 2015, pp. 4280–4284. ISBN: 2379-190X. DOI: 10.1109/ICASSP.2015.7178778.

[120] Tim Salimans and Diederik P. Kingma. "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks". In: *ArXiv* 1602.07868 (2016). URL: http://arxiv.org/abs/1602.07868.

[121] Amartya Sanyal et al. "TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Dy Jennifer and Krause Andreas. Vol. 80. PMLR, 2018, pp. 4490–4499. URL: https://proceedings.mlr.press/v80/sanyal18a.html.

[122] Eyyüb Sari, Vanessa Courville, and Vahid Partovi Nia. "iRNN: Integer-only Recurrent Neural Network". In: *ArXiv* 2109.09828 (2021). URL: https://arxiv.org/abs/2109.09828.

[123] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge: Cambridge University Press, 2014. ISBN: 978-1-107-05713-5.

[124] Jingyi Shen and M. Omair Shafiq. "Short-term stock market price trend prediction using a comprehensive deep learning system". In: *Journal of Big Data* 7.1 (Aug. 2020), p. 66. ISSN: 2196-1115. DOI: 10.1186/s40537-020-00333-6. URL: https://doi.org/10.1186/s40537-020-00333-6.

[125] Xingjian Shi et al. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015.

[126] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015. URL: http://arxiv.org/abs/1409.1556.

[127] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, 2018. ISBN: 0-262-03924-9.

[128] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. ISBN: 1063-6919. DOI: 10.1109/CVPR.2015.7298594.

[129] Ashish Vaswani et al. "Attention Is All You Need". In: *ArXiv* 1706.03762 (2017). DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[130] Peiqi Wang et al. "HitNet: hybrid ternary recurrent neural network". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. Montréal, Canada: Curran Associates Inc., 2018, pp. 602–612.

[131] Paul J. Werbos. "Applications of advances in nonlinear sensitivity analysis". In: *System Modeling and Optimization*. Ed. by R. F. Drenick and F. Kozin. Springer Berlin Heidelberg, 1982, pp. 762–770. ISBN: 978-3-540-39459-4.

[132] Bernard Widrow and Marcian E Hoff. *Adaptive switching circuits*. Tech. rep. Stanford University, Stanford Electronics Labs, 1960. URL: https://apps.dtic.mil/sti/pdfs/AD0241531.pdf.

[133] Zama. *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*. 2022. URL: https://github.com/zama-ai/tfhe-rs.

[134] Dongqing Zhang et al. "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Springer International Publishing, 2018, pp. 373–390. ISBN: 978-3-030-01237-3.

[135] Shuchang Zhou et al. “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients”. In: *ArXiv* 1606.06160 (2016). DOI: 10.48550/ARXIV.1606.06160. URL: https://arxiv.org/abs/1606.06160.

[136] Chenzhuo Zhu et al. “Trained ternary quantization”. In: *ArXiv* 1612.01064 (2016).

Appendix A

Long Short-Term Memory

The gates of the LSTM cell are implemented as feed-forward networks with `sigmoid` activation functions. The properties of `sigmoid` align with the desired behaviour of the gates. The `sigmoid` function maps inputs to the range between zero and one. When multiplied element-wise with the input information, it can constrict or encourage the flow of information based on the value of the `sigmoid` output. Values close to zero will constrict the flow, while values close to one will encourage it. The network also has the ability to open or close gates while remaining numerically stable. Large values in both the negative and positive directions are limited by the asymptotic nature of `sigmoid` and its derivative which is essential to mitigating the exploding and vanishing gradients problem. The `sigmoid` and `tanh` functions can be represented by the following equations:

$$\sigma(x) = \frac{1}{1 - e^{-x}} \tag{A.1}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}, \tag{A.2}$$

where $\forall x \in \mathbb{R}, \sigma(x) \in (0, 1), \tanh(x) \in (-1, 1)$.

Each gate learns to regulate information flow as a function of the concatenated input and immediate state vectors. The concatenation operation is essential to allow the network to control each gate based on each element of both vectors rather then the elements of some alternative combination of them. The location where the gates are applied in the cell allow each gate to differentiate. To aid in our explanation, figure A.1 illustrates the structure of the LSTM cell while equations A.3 to A.8 below represent the mathematical structure of

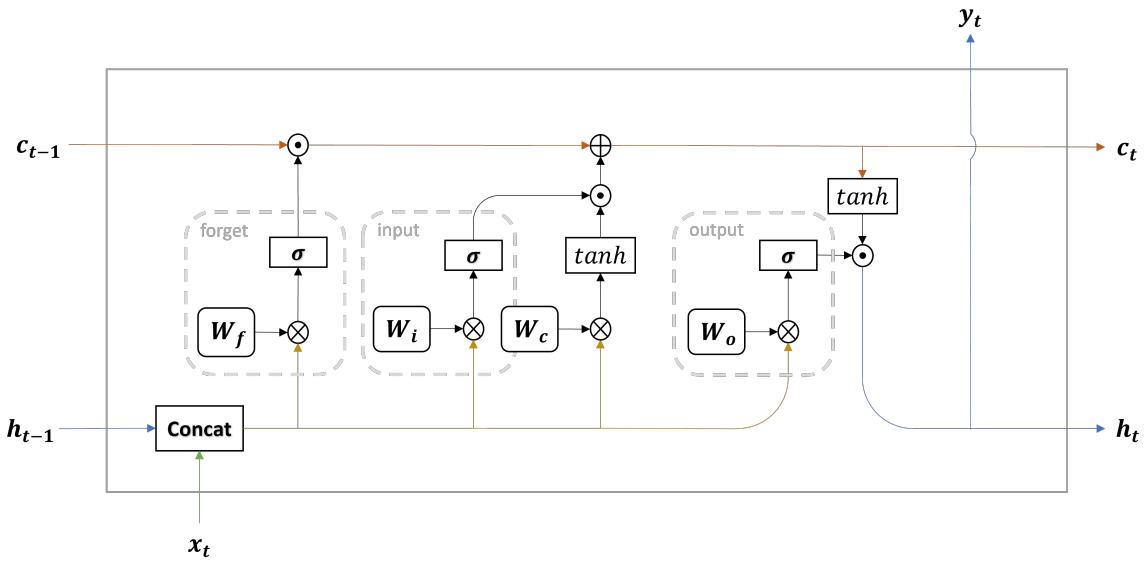


Figure A.1: **The Long Short-Term Memory RNN Cell.** This figure illustrates the structure of one LSTM cell at time t. Rectangular boxes surround functions applied to internal signals while rounded rectangular boxes surround the parameters of the cell. Colors are used to track the evolution of important signals in the cell. This figure is adapted from [92].

the cell:

$$\boldsymbol{f_t} = \sigma(\boldsymbol{W_f} \times (\boldsymbol{h_{t-1}}||\boldsymbol{x_t}) + \boldsymbol{b_f}) \tag{A.3}$$

$$\boldsymbol{i_t} = \sigma(\boldsymbol{W_i} \times (\boldsymbol{h_{t-1}}||\boldsymbol{x_t}) + \boldsymbol{b_i}) \tag{A.4}$$

$$\boldsymbol{c'_t} = \tanh(\boldsymbol{W_c} \times (\boldsymbol{h_{t-1}}||\boldsymbol{x_t}) + \boldsymbol{b_c}) \tag{A.5}$$

$$\boldsymbol{c_t} = \boldsymbol{f_t} \cdot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \cdot \boldsymbol{c'_t} \tag{A.6}$$

$$\boldsymbol{o_t} = \sigma(\boldsymbol{W_o} \times (\boldsymbol{h_{t-1}}||\boldsymbol{x_t}) + \boldsymbol{b_o}) \tag{A.7}$$

$$\boldsymbol{y_t} = \boldsymbol{h_t} = \boldsymbol{o_t} \cdot \tanh(\boldsymbol{c_t}), \tag{A.8}$$

where $\boldsymbol{x_t} \in \mathbb{R}^m$, $\boldsymbol{W_f}, \boldsymbol{W_i}, \boldsymbol{W_o} \in \mathbb{R}^{(m+n)\times n}$, all other vectors $\in \mathbb{R}^n$, n is the number of hidden units and m is the number of input units of the cell. The concatenation operation in this set of equations is across the first dimension.

There are three gates: the *forget*, *input*, and *output* gates, each encapsulated by a dashed rectangular box. The position of the *forget gate* is strategically chosen to regulate the magnitude of elements in the previous long-term state $\boldsymbol{c_{t-1}}$ that enter the cell. Conceptually, the *forget gate* instructs the network to disregard units that could potentially have a detrimental impact on the output and numerical stability of the cell. The *forget gate* uses the element-wise multiplication of the output units from its `sigmoid` function and $\boldsymbol{c_{t-1}}$ to update the memory. The parameters used to linearly transform the concatenated input and immediate state vectors in the *forget* gate are $\boldsymbol{W_f}$ and $\boldsymbol{b_f}$. The bias parameter $\boldsymbol{b_f}$ is not

displayed in figure A.1 but is assumed to be an additional row in $\boldsymbol{W_f}$ instead. Likewise, the *input* and *output* gates have distinct parameter sets, with each parameter subscript indicating its respective gate. The calculation of the *forget gate* is represented by equation A.3.

The *input gate* controls the influence of the current inputs and immediate state on the memory of the cell. While its structure is the same as its sister gates, the element-wise multiplication of its outputs with a feed-forward network transformation of the input and immediate state vectors (shown in equation A.5) causes it to regulate the flow of new information. As with other RNNs, the `tanh` function is used as the activation function for learning hidden representations in the LSTM cell. The regulated updated information resulting from the *input gate* is combined with the long-term state of the cell using an addition operation, resembling the addition operation in vanilla RNN cells, but with a gating mechanism. This completes the transformation of $\boldsymbol{c_{t-1}}$ to $\boldsymbol{c_t}$, the updated long-term memory fed into the next cell, represented by equation A.6. The calculation of the *input gate* is represented by equation A.4.

Finally, the *output gate* decides what elements of the transformed state $\boldsymbol{c_t}$ to output. At this point, the updated state contains elements from the immediate inputs to the cell and the long-term memory. To maintain stability and non-linearity, the `tanh` function is applied to $\boldsymbol{c_t}$. The *output gate* is element-wise multiplied with the activation of the cell and the result is assigned to the current output $\boldsymbol{y_t}$ and immediate state $\boldsymbol{h_t}$ of the cell (equation A.8). The calculation of the *output gate* is represented by equation A.7.

Appendix B

Notes On Security

B.1 Security Of FHE

It is important to understand the type of security provided by fully homomorphic encryption schemes in order to claim the evaluation of machine learning algorithms over encrypted data as privacy-preserving. Fully homomorphic encryption schemes are considered **semantically secure** [81]. Semantic security is achieved by encryption schemes when their encryption algorithms are at least resilient to chosen plaintext attacks. FHE schemes provide a type of security against chosen plaintext attacks referred to as "indistinguishable under chosen plaintext attack" (IND-CPA) security.

During an IND-CPA attack, an adversary is given a public key and has access to the encryption algorithm. The adversary is able to encrypt as many plaintexts as he desires. At a certain point, the adversary *chooses* two plaintexts m_0 and m_1 and sends them to the "challenger" (the entity they wish to attack). The challenger randomly selects one of the two plaintexts m_b where $b \in \{0, 1\}$ and sends its encryption $\text{Enc}(m_b)$ under the FHE scheme using the same public key to the adversary. The FHE scheme is secure under IND-CPA if the adversary cannot distinguish between encryptions of m_0 and m_1 made by him and the challenger's encryption of m_b. In other words, he cannot, with probability greater than $1/2$, determine the correct value of b. In fact, it is estimated to take him on the order of 2^λ operations to break the scheme. λ is referred to as the **security parameter** of FHE schemes, defining a set of FHE parameters that maintain at least that security level. Often, $\lambda = 128$ is considered a good, minimum security level with larger values denoting greater security.

Furthermore, FHE schemes must be *circular* secure [81]. This type of security requires that IND-CPA security holds for encryptions of a challenger's secret key as well. Circular security is necessary for FHE schemes since the ability to bootstrap relies on public knowledge of the encryption of a challenger's secret key. It is also important to understand that the nature of FHE makes it malleable. Since evaluation keys and public keys are public knowledge, any adversary can encrypt a message and augment the target ciphertext using

basic FHE operations. However, it is possible to keep the integrity of an FHE ciphertext by encrypting the ciphertext using another quantum secure, non-malleable encryption algorithm such as AES-256 [89]. This stops the ciphertext from being manipulated at rest and in transport.

B.2 Threat Model Of Privacy-Preserving Neural Networks

To understand why privacy-preserving neural networks provide enhanced security, it is important to define a threat model. According to [108], there are two types of threat models in PPML. The first is the *honest but curious* model, providing the weakest level of security. In this model, we assume that the parties involved in the protocol (privacy-preserving inference in this case) perform all steps in the protocol honestly, but try to learn something from the intermediate values they see while executing the protocol. PPML with FHE is assumed to be secure under this model since any value an adversary observes is encrypted under FHE and is thus semantically secure. There is a stronger level of security called *malicious security* where parties can deviate from the protocol in a malicious way [108]. If the protocol is resistant to malicious attacks such as the manipulation of ciphertexts before they are sent, then it is secure under this type of model. With respect to SMC, there are protocols that satisfy this, while for PPML with FHE, there is still research ongoing in this area [108]. Therefore, in this thesis, we assume security under the *honest but curious* threat model.

Appendix C

Detailed Notes On CGGI Construction

C.1 Key Generation

Given a security parameter λ (defined in appendix B.1), we evaluate $k = k(\lambda)$, $n = n(\lambda)$, and $N = N(\lambda)$ to define the main security parameters of the scheme (k, n, N). We assume that N is a power-of-two and all polynomials to follow are of order-N. The k parameter defines the number of polynomials representing the secret key and public $a_i(x)$ values. In the definition of RLWE in section 2.4.1, we assumed $k = 1$ whereas in a *General*-LWE setting, which combines both RLWE and LWE into one definition, $b(x) = \sum_{i=0}^{k-1} a_i(x) \cdot s_i(x) + e(x)$ where $a_i(x), s_i(x), e(x), b(x) \in \mathcal{T}_q$, and $k \geq 1$ [20]. A larger k increases the security. However, in this thesis, we only consider the $k = 1$ case which is assumed from this point forward. We sample $\boldsymbol{s} \sim U(\mathbb{B}^n)$ and $s(x) \sim U(\mathbb{B}(x))$ to be the LWE and RLWE secret keys respectively. We also define error distributions $\chi_{\text{LWE}} = \mathcal{N}(0, \sigma^2_{\text{LWE}})$ and $\chi_{\text{RLWE}} = \mathcal{N}(0, \sigma^2_{\text{RLWE}})$ over $\mathbb{R}$ for $\sigma_{\text{LWE}} = \sigma_{\text{LWE}}(\lambda)$ and $\sigma_{\text{RLWE}} = \sigma_{\text{RLWE}}(\lambda)$. For each security level λ, a set of parameters $(k, n, N, \sigma_{\text{LWE}}, \sigma_{\text{RLWE}}, p)$ is provided to the user, usually in the form of a table extracted by running the LWE estimator [3]. For example, in [10], tables 4-9 give optimal parameters for each security level, which are considered secure against the best known attacks and can tolerate precision levels up to p. According to [10], current CGGI algorithms and minimum security requirements limit the maximum value for ω to 8 to ensure computation does not become impractical (a higher precision results in more computational complexity). In summary, given λ, the key generation process outputs public information $(k, n, N, \chi_{\text{LWE}}, \chi_{\text{RLWE}}, p, q)$, and private information $(\boldsymbol{s}, s(x))$.

C.2 Encoding And Encryption

Let $\upsilon \in \mathbb{Z}_p$ and $\upsilon(x) \in \mathcal{R}_p$. We define the following encoding operation that maps messages to (R)LWE-compatible plaintexts:

$$\text{Enc}_{q,p}(\upsilon) = \Delta_p \cdot \upsilon \pmod{q} \tag{C.1}$$

where $\Delta_p = q/p$, and υ can be a scalar or polynomial (where mod $\phi(x)$ also applies).

We can make two observations from the encryption operation. Firstly. it is a sequence of three sub-operations: (1) *lift* a message υ to the MSB of the Ω-bit torus, (2) add the error to the LSB of the torus, and (3) *mask* the result with a dot product (or polynomial multiplication) between the public a values and secret key. We refer to this last operation as *masking* the ciphertext. Secondly, from section 2.3.1, we know that chaining several levelled operations on the same ciphertext increases the noise. Since the error is in the LSB of the unmasked ciphertext, we can imagine the error increasing one bit at a time until it overflows into the message which occupies ω bits of the MSB in the ciphertext. This causes incorrect decryption and it is necessary to avoid this type of scenario.

C.3 Decryption And Decoding

Given two plaintexts, $\mu \in \frac{q}{p}\mathbb{Z}_q$ and $\mu(x) \in \mathcal{P}_q$, we define the following decoding functions for both LWE and RLWE settings:

$$\text{Dec}_{q,p}(\mu + e_q) = \left\lceil \frac{(\mu + e_q) \bmod q}{\Delta_p} \right\rfloor \pmod{p} \tag{C.2}$$

$$\text{Dec}_{q,p}(\mu(x) + e_q(x)) = \left\lceil \frac{(\mu(x) + e_q(x)) \bmod q \cdot \phi(x)}{\Delta_p} \right\rfloor \pmod{p \cdot \phi(x)} \tag{C.3}$$

Using this construction, we are guaranteed (on a fresh ciphertext) that encoding, encrypting, decrypting, and decoding a message returns the correct message as long as $\|e_q\|_\infty < \frac{q}{2p}$ [20] (this applies to the coefficients of an RLWE example as well). With the accumulation of larger amounts of error, the probability of correctness upon decryption decreases. For certain applications, such as those in this thesis, a correct decryption is not always necessary. For example, evaluating the sign of a value using a programmable bootstrap allows for a larger tolerance of error and therefore, relaxes the bound.

Appendix D

Detailed Notes On CGGI Operations

D.1 Key Switching

The keyswitch key is allowed to be public information since at its core, it is an encryption of the secret key, and referring to appendix B.1, it is assumed that FHE schemes provide circular security. Given a keyswitch key $\text{ksk}_{\boldsymbol{s}_1}(\boldsymbol{s}_0)$ for LWE secret keys $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$, and an input ciphertext $\text{ct} = \text{LWE}_{n,q,\chi_{\text{LWE}}}(\mu, \boldsymbol{s}_0)$, we create a trivial ciphertext $(\mathbf{0}, b_0)$ where $\mathbf{0} = [0, 0, ..., 0] \in \{0\}^n$ and evaluate:

$$\text{KS}(\text{ct}, \text{ksk}_{\boldsymbol{s}_1}(\boldsymbol{s}_0)) = (\mathbf{0}, b) - \langle \boldsymbol{a}, \text{ksk}_{\boldsymbol{s}_1}(\boldsymbol{s}_0) \rangle = \text{LWE}_{n,q,\chi_{\text{LWE}}}(\mu, \boldsymbol{s}_1) \tag{D.1}$$

There are many steps we are skipping in the keyswitch process such as decomposition of the public $\boldsymbol{a}$ values and the type of matching encryption over decomposed $\boldsymbol{s}_0$ values. However, since we only use the operation at an abstract level within this thesis, we will not discuss them. We refer the reader to [25] for a more in-depth overview of this operation.

One of the most important keyswitching operations in this thesis is the keyswitch from a bootstrapped LWE ciphertext with secret key $\boldsymbol{s}'$, back to the secret key $\boldsymbol{s}$ that was originally used in the encryption of the value before bootstrapping. In section 2.4.4, where we discuss the programmable bootstrap in detail, we mention that an RLWE ciphertext, encrypted under secret key $s(x)$ and containing the lookup table we want to evaluate, is rotated so that the first coefficient becomes the correct output as queried by the input LWE ciphertext. To complete the PBS operation, we perform a **sample extraction** operation of the first coefficient which is not a levelled operation, but rather, a transformation of the coefficient to an LWE ciphertext. The resulting LWE ciphertext is encrypted under a secret key $\boldsymbol{s}'$, which is a vector representing the coefficients of $s(x)$. After the sample extraction, we can perform the necessary keyswitch from $\boldsymbol{s}'$ back to $\boldsymbol{s}$, following equation D.1.

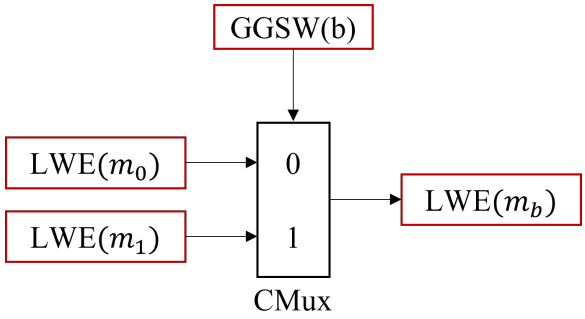


Figure D.1: **The Ciphertext Multiplexer (CMux)**. An operator in CGGI that homomorphically selects a ciphertext using an encrypted bit. All shapes outlined in red are considered ciphertexts.

D.2 Ciphertext Multiplexer

As shown in figure D.1, the CMux operator contains three variables: $\mathrm{ct}_0 = \mathrm{LWE}_{n,q,\chi_{\mathrm{LWE}}}(\mu_0, \boldsymbol{s})$, $\mathrm{ct}_1 = \mathrm{LWE}_{n,q,\chi_{\mathrm{LWE}}}(\mu_1, \boldsymbol{s})$, and $\mathrm{ct}_{\mathrm{sel}} = \mathrm{GGSW}(b, \boldsymbol{s})$ where $b \in \{0, 1\}$. Note that "GGSW" is a type of ciphertext defined in the original CGGI paper [25]. The following equation shows how to homomorphically evaluate the CMux operation:

$$\mathrm{ct}_b = \mathrm{CMux}\left(\mathrm{ct}_0, \mathrm{ct}_1, \mathrm{ct}_{\mathrm{sel}}\right) = \mathrm{ct}_{\mathrm{sel}} \odot \left(\mathrm{ct}_1 - \mathrm{ct}_0\right) + \mathrm{ct}_0 \tag{D.2}$$

The equation outputs $\mathrm{ct}_b = \mathrm{ct}_0$ if $b = 0$ and $\mathrm{ct}_b = \mathrm{ct}_1$ otherwise. The CMux operation plays an essential role in the blind rotation operator.

D.3 Blind Rotation

To perform the blind rotation operation, a series of CMux operations are chained together. The general idea is to decompose $t \in \mathbb{Z}$ into its binary representation $\boldsymbol{t}_b \in \mathbb{B}^{\gamma}$, where $\gamma = \log_2(N)$, generate an array of encrypted ciphertexts of the bits, and perform a series of CMux operations. If we want to rotate by t coefficients to the left, we can express the rotation as,

$$b^{(rot)}(x) = x^{-t} b(x) = x^{-\sum_{i=0}^{\gamma-1} t_i 2^i} b(x) = x^{-t_0} \left(x^{-t_1 2} \left(x^{-t_2 2^2} \left(\cdots \left(x^{-t_{\gamma-1} 2^{\gamma-1}} b(x)\right)\right)\right)\right) \tag{D.3}$$

This equation produces a chain of rotations of $b(x)$ by several monomials, in sequence. The brackets in equation D.3 show the chain of operations to rotate $b(x)$. Let us consider a

generic monomial $m(x)$ and $x^{-t_i 2^i}$. Multiplying both values yields two possibilities:

$$m(x) \cdot x^{-t_i 2^i} = \begin{cases} m(x) & \text{if } t_i = 0 \\ m(x) \cdot X^{-2^i} & \text{if } t_i = 1 \end{cases} \tag{D.4}$$

where $i \in [0, \gamma)$. If we are provided with a GGSW encryption of t_i, and two RLWE ciphertexts encrypting $m(x)$ and $m(x) \cdot X^{-2^i}$, we can output a selection of either ciphertext based on the encryption of t_i. Given $\text{ct}_0 = \text{RLWE}_{k,N,q,\chi_{\text{RLWE}}}(m(x), s(x))$ and $\text{ct}_t = GGSW(t_i, s(x))$, we evaluate $\text{ct}_1 = \text{ct}_0 \cdot x^{-2^i}$, which is a rotation by 2^i coefficients to the left. Therefore, equation D.4 can be transformed into,

$$\text{ct}^{(\text{rot})} = \text{ct}_0 \cdot x^{-\text{ct}_t 2^i} = \text{CMux}\left(\text{ct}_0, \text{ct}_1, \text{ct}_t\right) \tag{D.5}$$

Assuming ct_0 is the ciphertext we would like to rotate by an encrypted amount of coefficients, and assuming $t_{\text{enc}} = [GGSW(t_0, s(x)), GGSW(t_1, s(x)), ..., GGSW(t_{\gamma-1}, s(x))]$, then according to equation D.3, we rotate by one monomial through an evaluation of equation D.5, then we rotate the result with another monomial, and we continue until the last monomial. The brackets in equation D.3 show the order of operations, however, any monomial can be used to perform a rotation in any order due to the commutative property of multiplication. Therefore, there are γ CMux operations to perform a blind rotation of an encrypted $t \in \mathbb{Z}$.

D.4 Programmable Bootstrapping

Equation 2.19 contains implied modulus switching operations. Since $b \in \mathbb{Z}_q$, and V_I contains only $2N$ values, we need to switch the modulus of b from q to $2N$. The following function shows how to perform a modulus switch of $a \in \mathbb{Z}_q$ from q to a scalar $\omega < q$:

$$\text{ModSwitch}(a, \omega) = \left\lceil \frac{\omega \cdot a}{q} \right\rfloor \pmod{\omega} \tag{D.6}$$

Therefore, to switch the modulus of an LWE ciphertext $\text{ct} = (\boldsymbol{a}, b)$, you perform the above operation for b and for each $a_i \in \boldsymbol{a}$.

In equation 2.19, the decryption operation is placed in the exponent for rotation. According to the equation as it is shown, the multi-sum $\sum_{i=0}^{n} a_{i,2N} \cdot s_i$ must be performed publicly with a private secret key, an impossible combination. Examining the bit decomposition operation in appendix D.3, we notice that the secret key $\boldsymbol{s}$ is also an array of bits and the multi-sum being evaluated is similar to the multi-sum being evaluated in the blind rotation section with a_i replacing 2^i. Since we assume CGGI provides circular security, we

can thus encrypt each $s_i \in \boldsymbol{s}$ under GGSW and obtain the following bootstrapping key:

$$\text{bsk}(\boldsymbol{s}) = [\text{GGSW}(s_0, s(x)), \text{GGSW}(s_1, s(x)), ..., \text{GGSW}(s_{n-1}, s(x))] \tag{D.7}$$

where $s(x)$ is an RLWE secret key whose coefficients will be the secret key for sample-extracting LWE ciphertexts after rotation. The bootstrapping key is public and generated during the key generation phase in CGGI. This allows any server to perform a blind rotation with the multi-sum defined above on any ciphertext encrypted under secret key $\boldsymbol{s}$.

After rotating V_I by $-(b_{2N} - \sum_{i=0}^{n} a_{i,2N} \odot \text{bsk}_i)$ coefficients, we are left with an RLWE ciphertext where the first coefficient is equal to $f_I(\mu_{2N} + e_{2N}) = \mu_p$. It is required for $p < 2N < q$, in order to allow $\mu_{2N} = \mu_p$. Another observation is that after performing a sample extraction of the first coefficient, the new LWE ciphertext is encrypted under the coefficients of $s(x)$. In order to continue performing operations such as bootstrapping on the ciphertext, it is necessary perform a key switch from the coefficients of $s(x)$ to $\boldsymbol{s}$. For this reason, it is common for a bootstrapping operation to be followed by a key switch operation. In fact, for ideal noise propagation, it is suggested to perform this set of operations atomically [10].

Appendix E

Summary Of CGGI DNN Papers

FHE-DiNN [13]

The FHE-DiNN paper is considered one of the most important and groundbreaking papers in this area. In general, it shows how to evaluate any DNN by converting it to a discretized deep neural network (DiNN) compatible with CGGI, and proposes several optimizations/additions to the CGGI scheme. One of the additions it proposes is the programmable bootstrap itself. To convert any DNN to a DiNN, inputs and weights must first be quantized. The quantization strategy used in [13] is a symmetric strategy centered around 0 with bounds $[\alpha, \beta]$ where for $\tau \in \mathbb{N}$, $\alpha = \beta = \tau$. They introduce the following quantization equation:

$$\text{processWeight}(w, \tau) = \tau \cdot \left\lceil \frac{w}{\tau} \right\rfloor \tag{E.1}$$

where $w \in \mathbb{R}$ is any weight. The authors make note to mention that the choice of τ is important since a large τ would increase the resolution of the quantization (as can be deduced from equation E.1) and increase the quantization accuracy, but would also increase the message space, or p as defined in section 2.4.2. Increasing the message space of a ciphertext also increases the message space needed to handle an accumulation of that ciphertext with another. If an accumulation ciphertext surpasses its message space, it wraps around the modulus, an effect of integer arithmetic. Quantization in FHE-DiNN is a form of post training quantization. This means that the model is converted as-is to the quantization domain.

In FHE-DiNN, the input representation of each integer input is an encryption of itself by a separate LWE ciphertext. Inputs to DiNNs are therefore matrices or arrays of LWE ciphertexts. By keeping the model weights in plaintext form and ciphertexts as arrays, matrix multiplication is implemented using the standard algorithm from its definition: for each row $\boldsymbol{w}_i \in \mathbb{Z}_p^n$ of $\boldsymbol{W} \in \mathbb{Z}_p^{m \times n}$, the element in the output vector $\boldsymbol{W} \cdot \boldsymbol{v}$ is the dot product $\langle \boldsymbol{w}_i, \boldsymbol{v} \rangle$ where $\boldsymbol{v} \in \text{LWE}^n$, LWE refers to an LWE ciphertext, the dot product is a sequence of scalar multiplications and ciphertext-ciphertext additions, n is the input dimension, and m

is the output dimension. The authors accumulate the dot product additions into a regular ciphertext with the same message space.

The authors quantize activations through binarization. They set the activation functions to the Signum function (equation 2.24) which they implement by performing a programmable bootstrap of each pre-activation ciphertext. The bootstrap, as mentioned in section 2.4.4, applies a blind rotation over the RLWE ciphertext with each coefficient encrypting the number 1, thus outputting the sign. Since the ring dimension $N = 2^{10}$, the maximum precision of the bootstrapping operation is $p = 2^{10}$. Thus, accumulation ciphertexts must be able to handle inputs between $[-2^9, 2^9 - 1]$, otherwise, the sign may be incorrect.

In FHE-DiNN, two models are evaluated. Both are one layer, feed-forward models with 30 and 100 hidden units, referred to as FHE-DiNN30 ($\sim 24,000$ parameters) and FHE-DiNN100 ($\sim 80,000$ parameters) respectively. The models are trained on the MNIST [66] dataset to perform handwritten digit classification, and achieve plaintext, test accuracies of 93.55% and 96.43% respectively. The models output logits rather than applying `softmax` in the encrypted domain. In fact, all of the papers reviewed in this section output logits rather than `softmax` which is one of the reasons why they cannot guarantee the privacy of the model.

Over encrypted data, both models are executed using 80-bit security in CGGI. They achieve accuracies of 93.71% and 96.35% respectively, showing a minimal difference to the plaintext execution accuracies. With respect to efficiency, FHE-DiNN30 executes in 515ms while FHE-DiNN100 executes in 1.679s, both using a four-core, Intel Core i7-4720HQ CPU at 2.60GHz. The authors do not mention any type of hardware acceleration (CPU multithreading or GPU).

TAPAS [121]

The TAPAS paper proposes a different method for transforming DNN models to CGGI models. The major difference between this paper and FHE-DiNN is the input representation. For each input integer, TAPAS decomposes the integer into its d-bit representation and encrypts each bit as an LWE ciphertext. More formally, $\forall x \in \mathbb{Z}_{2^d}$, we define $\boldsymbol{x} \in \{0,1\}^d$ such that,

$$x = \sum_{i=0}^{d-1} x_i \cdot 2^{d-1-i}, \tag{E.2}$$

where $x_i \in \boldsymbol{x}$. From this definition, each integer x can be represented in CGGI by transforming $\boldsymbol{x} = [x_0, x_1, ..., x_{d-1}]$ to $\boldsymbol{x}_{\text{CGGI}} = [\text{LWE}(x_0, \boldsymbol{s}), \text{LWE}(x_1, \boldsymbol{s}), ..., \text{LWE}(x_{d-1}, \boldsymbol{s})]$ for any secret key $\boldsymbol{s}$.

TAPAS performs binarization of all the weights and activations by applying the same method proposed in the BNN paper [29] (refer to section 2.5.3 for more details). This

means that their quantization method is to perform quantization aware training, since BNN trains a binarized network. TAPAS also replaces each -1 with 0 in the binarized weights which turns a matrix multiplication into a series of additions, simplifying the circuit. The authors suggest ternarization could simplify the network. They perform an evaluation using ternarization in their paper, discussed further below.

Since the representation of activations is in arrays of ciphertexts encrypting bits, additions must be implemented as circuits. For example, circuits such as full/half adders and ripple-carry adders must be implemented. The original version of CGGI [22] allowed the evaluation of gates such as AND, XOR, XNOR, etc. All of these gates were performed using gate bootstrapping, a topic discussed in section 2.3.2. Some of these circuits contain several gates and thus, several bootstrap operations per single addition. Considering the massive amount of additions in a matrix multiplication between binary values, this type of operation is very slow, as can be seen by their evaluation metrics. Nevertheless, TAPAS introduces two ways of performing additions: a "reduce tree adder", which is a binary tree of half and ripple-carry adders, and a "sorting network". For details on these methods, refer to [121]. To perform the sign bit, TAPAS only needs to use the encrypted ciphertext in the first index of the ciphertext array since, being the sign bit, it defines whether the full set of bits represents a negative or positive integer, making it a very simple operation. Accumulation is not a problem in these types of networks since it occurs over an array of ciphertexts. These types of networks can also perform power-of-two division since right shifting is as simple as shifting the indices of the ciphertext array.

TAPAS evaluates its strategy using the MNIST dataset, similar to FHE-DiNN. The authors utilize a feed-forward model with three dense layers of 2048 units each, and an output layer of 10 units, one for each digit. In total, this model contains about 10,000,000 parameters. Unfortunately, there is no mention of the security level they use for their CGGI implementation. They construct a full-precision network that achieves 99.8% accuracy, while their BNN network utilizing 80% ternary weights achieves 97.3% accuracy. In terms of timing analysis, TAPAS only evaluates a sequential model without model-level parallelism that executes one inference in 65.1 hours, a very poor result. These results include parallelism across 16 CPU cores in their adder circuits only. It is unclear whether the other operations utilize any thread-level parallelism. The other results they provide are estimations given model-level parallelism. Even with an estimation, they suggest a fully parallelized model would execute an inference in 147s.

SHE [78]

SHE is a similar paper to TAPAS with respect to the input representation. SHE quantizes their inputs using 5-bit, power-of-two log-scale quantization. In basic terms, they perform a non-uniform quantization. They utilize post training quantization. For more information on this specific type of quantization, refer to [70]. Therefore, each value of the network is

represented by an array of five ciphertexts, similar to TAPAS. The arithmetic they use to evaluate the network is fixed-point as a result of the array of ciphertexts. Weights are also 5-bit power-of-two log-quantized. As a result of this type of quantization, logical shifts of the ciphertexts in the ciphertext arrays replace multiplications/divisions.

For matrix multiplications, multiplications are substituted by shifts and additions are computed using an adder circuit, similar to TAPAS. The authors use the gate-bootstrapping version of CGGI to perform the adder circuits. They also create max-pooling and ReLU circuits that they evaluate using gate-bootstrapping. Therefore, the activation function they use for all evaluated models is the ReLU function. However, they claim to operate these circuits in a levelled setting. This means that the gate evaluations are performed *without bootstrapping*. The authors suggest that CGGI ciphertexts can handle several levelled operations until they need to be bootstrapped. Accumulation is also not a problem due to the bit representation of activations.

SHE evaluates several different types of models on different types of datasets. They evaluate CNN and RNN models, being the only FHE paper to our knowledge that uses CGGI to evaluate RNNs. In section 2.6.3, we expand on the implementation of RNNs in SHE. SHE evaluates their method on the MNIST dataset using a CNN model from [42]. For information on the structure of the model, refer to the SHE paper. The number of parameters in this model is approximately 130,000. The authors set the CGGI scheme to 152-bit security. Using the CGGI model, an accuracy of 99.54% and a runtime of 9s are obtained. In the FHE setting, they obtain a runtime of 3 hours. There is no mention of any parallelization in the paper, however the authors utilize a ten-core, Intel Xeon E7-4850 CPU to perform their experiments. The high level of accuracy is suggestive of the fixed-point quantization administered, while operation in a levelled setting decreases runtime by three orders of magnitude.

Concrete-DNN [20]

Concrete-DNN is a paper that is similar to FHE-DiNN. The authors stray away from representing integers through their bit-decomposition, rather, they represent quantized integers as integers in $\mathbb{Z}_p$ (one ciphertext rather than d ciphertexts). They do not specify what type of quantization they use for inputs, weights, and activations, however, we can assume that they employ a type of post-training quantization since they mention that their network benefits from not having to be retrained. Matrix multiplication and accumulation occurs as it would in plaintext, using CGGI operations such as scalar multiplication and addition. The authors evaluate models that are 20 (NN-20), 50 (NN-50), and 100 (NN-100) dense layers deep on the MNIST dataset. For a security setting of 128-bits and using eight NVIDIA A100 Tensor Core GPUs, they achieve runtimes of 7.53s, 18.89s, and 37.65s for each network, respectively. They also achieve 97.1%, 94.7%, and 83% accuracies, respectively. It is difficult to qualify these results since the authors do not mention the size or

structure of the models. They only mention that the models contain mixtures of CNN and feed-forward layers, with at least 92 active neurons per layer. However, we can approximate upper bounds of around 230,000 parameters for NN-20, 480,000 parameters for NN-50, and 910,000 parameters for NN-100 by assuming every layer is fully connected.

REDsec [34]

In REDsec, the authors implement the same input representation as TAPAS and SHE, representing each quantized integer with 8 ciphertexts, encrypting 8 bits. Their quantization strategy uses quantization aware training to ternarize the parameters of the network. The inputs are quantized using a symmetric quantization centered at zero. The authors propose a new method for multiplication of bits with ternary weights and an interesting application of an existing method to perform additions. Instead of performing additions using full, half, or ripple-carry adders, the authors combine the decomposed ciphertexts into one ciphertext. They then accumulate in the integer domain using ciphertext-ciphertext addition, and decompose homomorphically back into d ciphertexts. This allows them to perform expensive addition operations at the cost of only one bootstrap rather than several. This also allows them to perform the Signum activation function the same way as in TAPAS, which is much more efficient than performing a bootstrap. For information on homomorphic decomposition, or "bridging", refer to [34].

The authors use the cuFHE [17] library to perform CGGI operations and also provide many optimizations to the library. They configure cuFHE to utilize 110 bits of security and perform their experiments using eight NVIDIA T4 GPUs. They convert a small model with two dense layers, each with 96 hidden units, using the Signum function as the activation for each layer to a CGGI-compatible model. According to their description, the model contains roughly 86,000 parameters. Running their model on the MNIST achieves 93.1% accuracy and a runtime of 780ms. However, their execution is around 100x slower than plaintext evaluation. The authors compare their results against all of the papers mentioned in this section (except TAPAS) in table 4 of the paper [34]. While they achieve the lowest accuracy (around 3.3% less than FHE-DiNN), their implementation is the fastest.

Appendix F

Extensions To Overflow-Aware Regularizer Section

F.1 $\mathcal{L}_2$ vs OAR_1 vs OAR_2

The following equations define L1 and L2 regularization functions for parameter matrices:

$$\mathcal{L}_1(\boldsymbol{\theta}^{(i)}) = \sum_{j=0}^{n-1} \left|\theta_j^{(i)}\right| \tag{F.1}$$

$$\mathcal{L}_2(\boldsymbol{\theta}^{(i)}) = \sum_{j=0}^{n-1} \left(\theta_j^{(i)}\right)^2 \tag{F.2}$$

where i refers to a parameter matrix in the set of all parameter matrices, and n refers to the number of weights in the parameter.

In this section, the experiments evaluate which type of regularization is best for accuracy. There are three different types of regularization defined in this thesis that can push pre-activations to the correct regions in $\mathbb{Z}_p$. $\mathcal{L}_2$ regularization can be applied to pre-activations to push them towards the first correct region, specifically $[-\frac{p}{2}, \frac{p}{2} - 1]$. OAR_1 regularization pushes the pre-activations to all the correct regions, using the same magnitude of the gradient for each value. OAR_2 regularization is similar, except that it uses a gradient magnitude for each value relative to the distance of the value from the center of the incorrect region. Table F.1 shows the difference in accuracy when each regularizer is applied to the MNIST RNN. The results in the table are recorded after training for 1000 epochs with a precision setting of 6-bit. From the table, it is evident that $\mathcal{L}_2$ regularization does not help increase the accuracy. However, for each run performed with $\mathcal{L}_2$ regularization, the OAR metric for each layer achieves almost 100%. Inspecting the pre-activation distributions reveals that the values are correctly moved to the region $[-32, 31]$, yet the accuracy still does not increase. This observation, along with the better accuracy results for the runs with OAR, leads us to

Regularization Rate	$\mathcal{L}_2$	OAR_1	OAR_2
$1e^{-5}$	34%	25%	88%
$5e^{-5}$	31%	25%	91%
$1e^{-4}$	31%	26%	92%
$5e^{-4}$	30%	44%	92%
$1e^{-3}$	30%	74%	89%
$5e^{-3}$	31%	69%	86%

Table F.1: **MNIST RNN accuracy with $\mathcal{L}_2$, OAR_1, and OAR_2 activity regularization**.

conclude that OAR regularization is necessary to both move values to correct regions *and* retain accuracy.

Since we are using ternarization, a possible reason for the poor performance with $\mathcal{L}_2$ regularization is that most of the weights are pushed to zero, limiting the network's representational power. Effectively, this approach prunes the network, and the more the network is pruned, the lower the accuracy. In contrast, OAR regularization pulls weights towards zero and also pushes them outwards to other regions, causing less weights to be quantized as zero values. The number of quantized parameters that are zero divided by the total number of parameters is considered the pruning level. For instance, if this number is 80%, then 80% of the parameters in the model are zero. That being said, the pruning levels for the runs from table F.1 with $\mathcal{L}_2$, OAR_1, and OAR_2 regularization, and a regularization rate of $1e^{-3}$, are 84.96%, 77.58%, and 78.02% respectively. These numbers show that $\mathcal{L}_2$ regularization prunes the network more than OAR regularization (around 9% more in this case), suggesting that it could contribute to the observed lower accuracy.

Table F.1 shows that OAR_1 and OAR_2 are effective regularization techniques for achieving high OAR metrics and accuracy. The table also shows that OAR_1 is less effective than OAR_2. In addition to the table, figure F.1 shows the training curves of all the experiments in the table. On the right side of the table, the brackets aid in identifying the runs and their associated pre-activation regularization method. OAR_2 consistently outperforms OAR_1 in both the table and figure. The figure shows a better separation between the methods, with $\mathcal{L}_2$ at the bottom, OAR_1 in the middle, and OAR_2 at the top. Thus, for the rest of the experiments, we utilize OAR_2 as the main OAR regularizer.

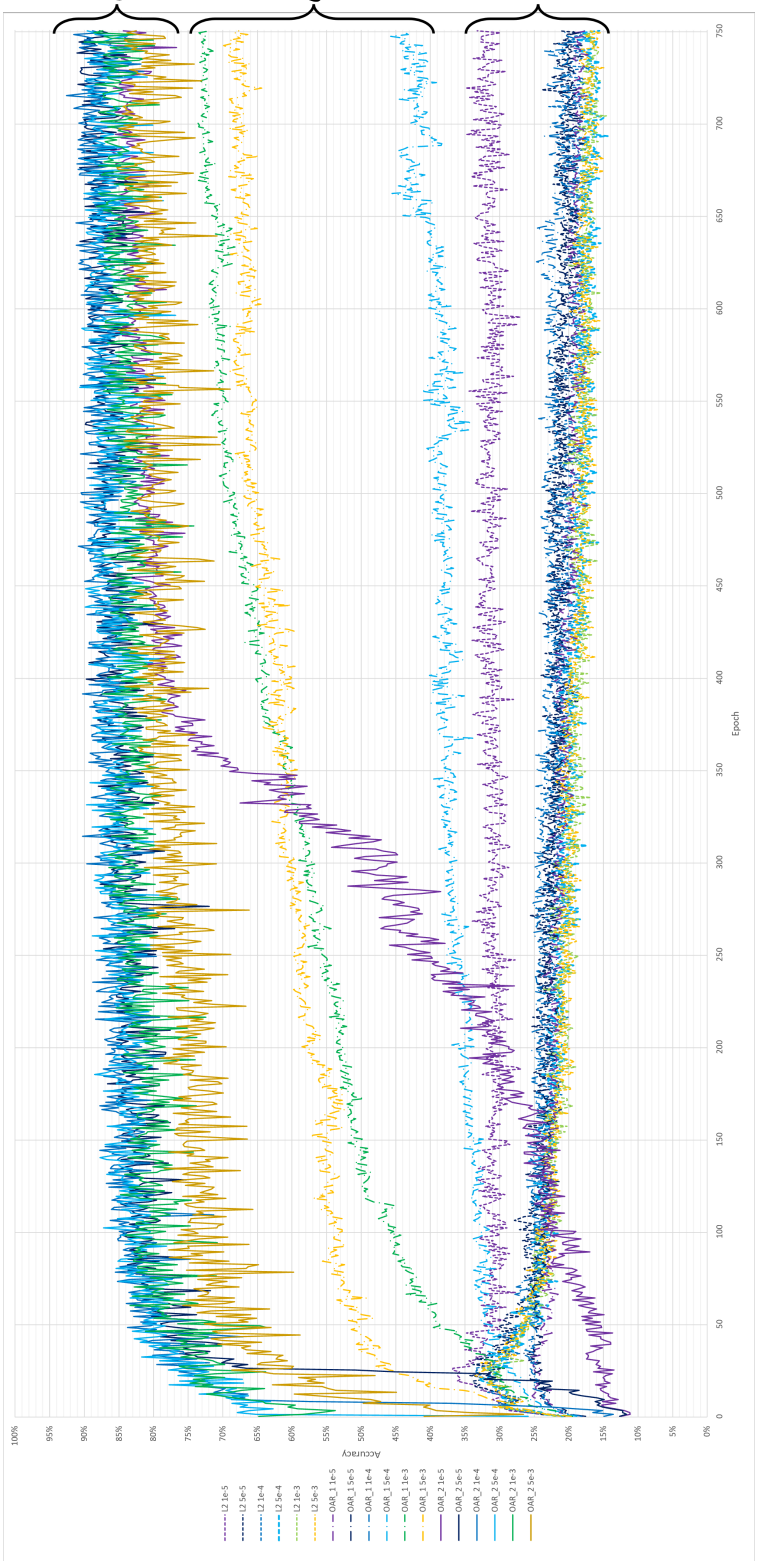


Figure F.1: **Comparison of three regularization techniques.** This figure illustrates the training results of using three different regularizers on the pre-activations of the MNIST RNN. OAR_1, OAR_2, and $\mathcal{L}_2$ are each used five times for different regularization rates. The metric displayed in the graph is the validation accuracy across the epochs. (*) In this region, there are also experiments with OAR_1 regularization. It is evident in the figure that OAR_2 performs the best with respect to accuracy, while both OAR regularizers perform much better than regular $\mathcal{L}_2$ regularization. This figure is discussed primarily in appendix F.1.

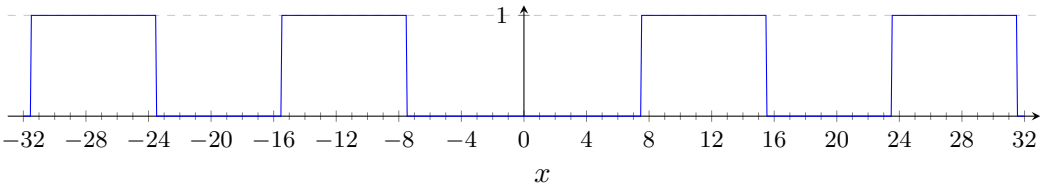


Figure F.2: **Incorrect Accumulations Metric**. This graph represents the function that flags incorrect values when calculating the incorrect accumulation metric. In each region where $f(x) = 1$, x is considered an incorrect value. Values that fall outside of these regions are considered correct.

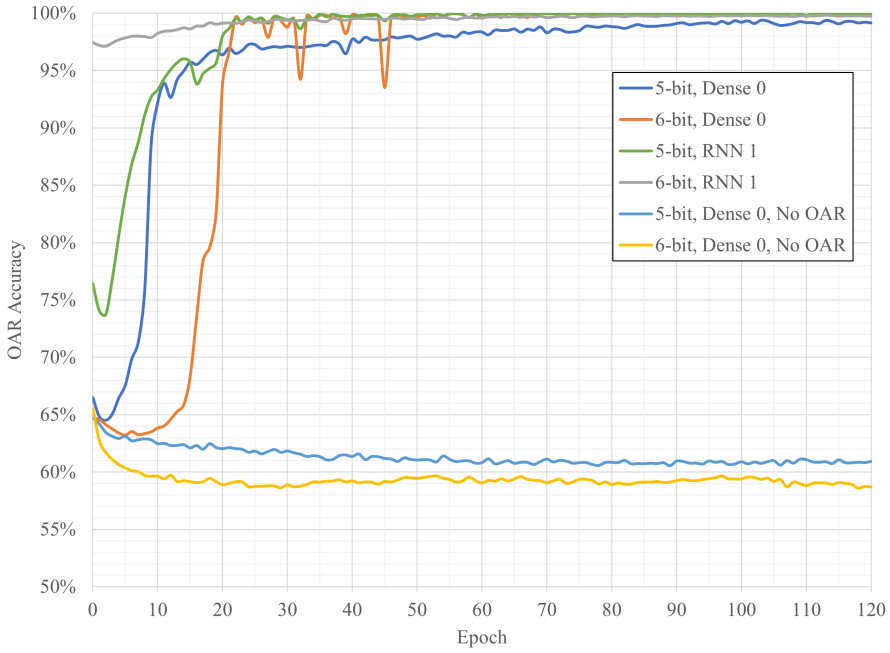
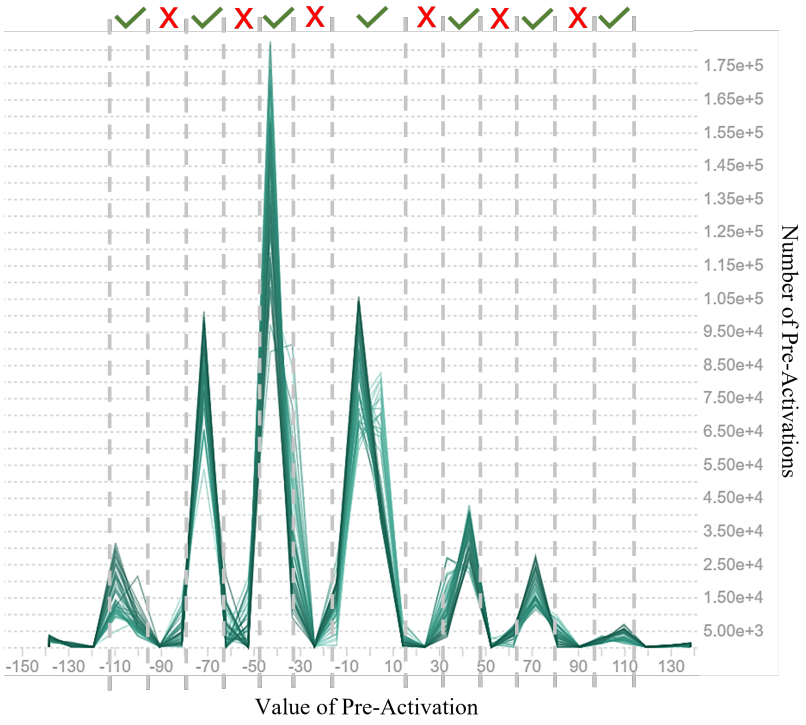
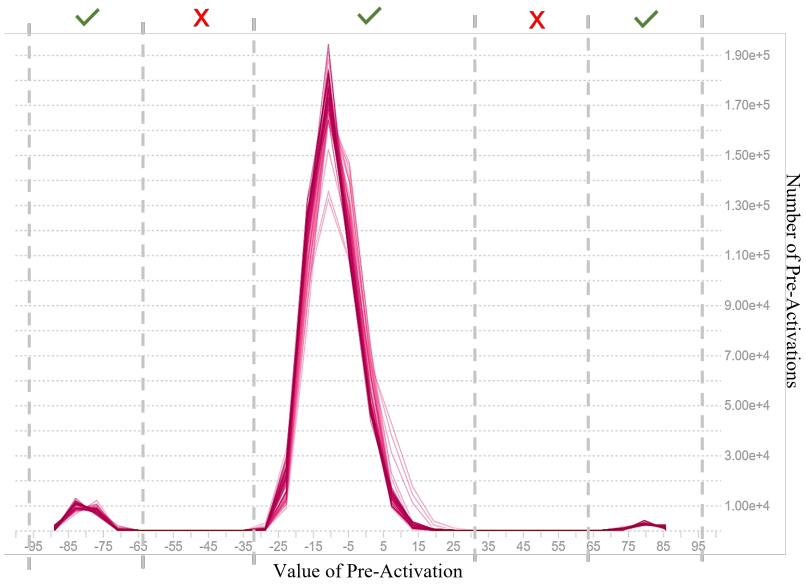


Figure F.3: **OAR metric evaluation.** OAR metric results of four quantizations of the MNIST RNN for the Dense 0 and RNN 1 layers. Two of the evaluations are without OAR regularization and the other two are with OAR regularization. Two evaluations are also in the 5-bit setting while the other two are in the 6-bit setting. The figure shows that the metric results do not improve for runs without OAR regularization and that they do improve for runs with OAR regularization.



(a) 5-bit



(b) 6-bit

Figure F.4: **Pre-Activation Distributions With OAR Regularization**. Training pre-activation distributions of the Dense 0 layer in the MNIST RNN, running with 5-bit and 6-bit OAR_2 regularization and a rate of $1e^{-4}$. The lighter lines indicate the distribution in a past epoch—the lighter the line, the older the epoch. A checkmark indicates a correct region and an "X" indicates an incorrect region. The figure shows that the pre-activation distributions successfully transition to correct accumulation regions as a result of OAR regularization.

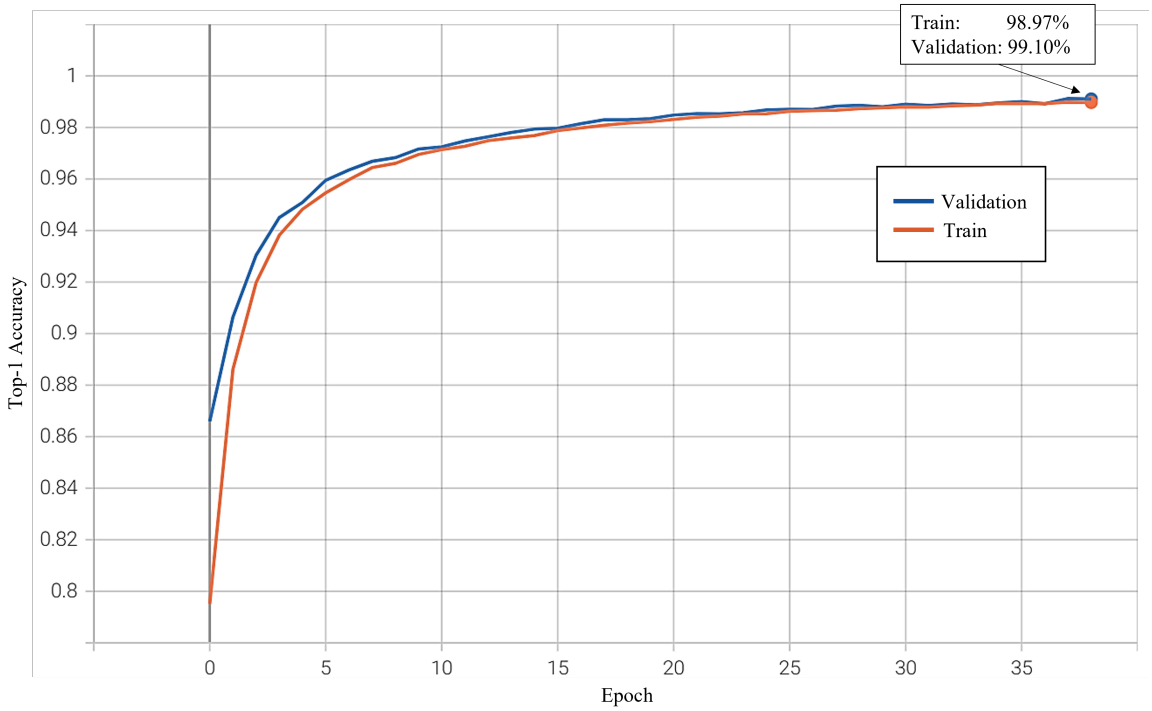


Figure F.5: **MNIST RNN full-precision accuracy.** Full-precision training and validation graph of the MNIST RNN over the MNIST dataset. For information regarding the model being trained, refer to section 3.3. This figure shows that the MNIST RNN achieves remarkable test and validation accuracy.

Appendix G

Additional CGGI-Conversion Information

G.1 Experimental Setup

	Machine A	Machine B
CPU	Intel i7-7820X, 8-Core, 3.60 GHz	AMD EPYC 7763, 64-Core, 3.1 GHz (x2)
GPU	Nvidia RTX 2080 Ti (x2)	Nvidia A100 40GB (x2)
RAM	132 GB	512 GB
OS	Ubuntu 22.04.2 LTS, 5.19.0-46-generic	Ubuntu 22.10, 5.19.0-46-generic
CUDA	11.2.152	12.1.105
Python	3.10.6	N/A
TensorFlow	2.10.1	N/A
QKeras	0.9.0	N/A
Rust	1.69.0	
Concrete-Core	https://github.com/irskid5/concrete-core/tree/3c9d247	

Table G.1: **Hardware and software experimental setup**. Standalone numbers indicate corresponding version numbers. (x2) indicates there are two components.

G.2 Concrete Security Parameters

G.2.1 Security Parameter Selection

To perform CGGI operations, there are a variety of parameters to choose from that provide different security levels and plaintext precision capabilities. For this thesis, we use the parameters provided in tables 4 and 5 in *Bergerat et al.* [10]. According to the authors,

these parameters maintain a security level of at least $\lambda = 128$ bits. The parameter sets we experiment with are shown in table G.2. The table contains three parameter sets, in descending order of n. Since $k = 1$ and the decomposition parameters are the same for each of the sets, the efficiency of bootstrapping, which is the most expensive CGGI operation, depends only on the value of n for this set of parameters specifically (refer to table G.2 for more details). Thus, the latency decreases for each parameter set down the table, as well as the precision. The first set is rated for a precision of 6-bit, whereas the other two sets are rated for 5-bit. In our experimentation, we find that all parameter sets are able to support 6-bit precision. However, since the LWE and RLWE variances increase with each operation, the amount of CGGI operations that can be applied consecutively, such as scalar multiplication and addition, decreases. Thus, set 1 can handle more operations in 6-bit than sets 2 and 3, and set 2 can handle more than set 3. Consequently, the size of the dot-products that can be evaluated is less for each parameter set as you go down the table. Since noise increases, and larger dot-products can increase the size of the domain of distributions, a larger N parameter becomes important. The ring dimension determines the size of the LUT for evaluating functions using the PBS operation. If the domain increases, then the amount of repetition in each mega-pack in the LUT should increase as well. As you go down the table, each parameter set has a lower ring dimension, resulting in a smaller rounding interval for each query. In summary, we have to balance between efficiency, depth of operations, and rounding error. In section 4.4, we evaluate our networks using all three sets and discuss the benefits and drawbacks of using each parameter set.

Set ID	$\log_2(p)$	n	σ^2_{LWE}	$\log_2(N)$	σ^2_{RLWE}
1	6	796	$3.72852e^{-12}$	12	$4.70198e^{-38}$
2	5	732	$3.87088e^{-11}$	11	$4.90564e^{-32}$
3	5	585	$8.35721e^{-09}$	10	$8.93436e^{-16}$

Table G.2: **Concrete security parameters for** $\lambda = 128$. $k = 1$ applies to all parameters. For more information on the meaning of these parameters, refer to section 2.4.2.

We have included benchmarking for each parameter set in appendix G.5 using machine B (details in table G.1). The latency of bootstrapping several groups of ciphertexts, specifically $\{32, 64, 128, 256, 512, 1024, 2048\}$, for three different parallelized hardware setups is shown in figures G.3 (parameter set 1), G.4 (parameter set 2), and G.5 (parameter set 3). All three graphs show that two A100 GPUs consistently outperform the other hardware setups. The single A100 and multi core setups perform similarly, suggesting that CPU parallelization is also a candidate for running efficient, large-scale RNNs over encrypted data. The latency results between the single and double A100 benchmarks in each figure show that when doubling the number of GPUs, the latency is halved, which is what is theoretically expected. This suggests the possibility of a linear scaling. Theoretically, by increasing the number of GPUs, or CPU cores, we can reduce the latency to a lower bound for each differently sized

group of ciphertexts, with the lower bound being the cost of bootstrapping one ciphertext. Due to resource constraints, we were not able to test a larger number of GPUs to verify this claim. It is also evident that the latency of each parameter set is less than the previous.

G.2.2 Effect Of Decomposition On PBS Latency

For the parameters in table G.2, a larger n causes increased PBS execution time, since the number of rotation steps performed sequentially in the PBS operation is equal to the size of the LWE secret key. Additionally, the decomposition parameters can also change the execution time. We have not talked about decomposition in this thesis, since it is not important to the overall goal. Briefly, the external product operation (which is utilized in the PBS and key-switching operations) occurs between encryptions of the decompositions by a certain basis and exponent (level) of the values being multiplied, rather than the values themselves. The rationale behind this operation is to mitigate the effects of multiplicative noise. For more information, refer to [25]. The decomposition parameters are represented by l_{PBS}, l_{KS}, $\log_2(\beta_{\text{PBS}})$, and $\log_2(\beta_{\text{KS}})$. The β parameters are the decomposition bases and the l parameters are the decomposition levels. The larger the level, the less efficient the external product (and the PBS/key-switching operations as a result), and the smaller the resulting noise. Therefore, there is a trade-off between noise and latency. For the security parameters in table G.2, the decomposition parameters are set to $l_{\text{PBS}} = 3$, $l_{\text{KS}} = 2$, $\log_2(\beta_{\text{PBS}}) = 14$, and $\log_2(\beta_{\text{KS}}) = 8$, which obtains the best trade-off in our experimentation.

G.3 MNIST RNN Evaluation

Paper & Model	Est. No. Parameters	Runtime (s)	Est. 1.9M Parameter Runtime (s)	Depth In Layers
FHE-DiNN100 [13]	80k	1.68	40.178	1
TAPAS torch7 [121]	10M	234,360	44,865	3
SHE CNN [78]	130k	9	132	4
SHE RNN	180k	576	6126	25
Concrete-DNN NN-20 [20]*	230k	7.53	62.675	20
Concrete-DNN NN-100*	910k	37.65	79.204	100
REDsec FF-96 [34]**	86k	0.78	17.363	2
This Work (Regular, Set 2)*	1.91M	4.86	4.86	31
This Work (Regular, Set 3)*		2.10	**2.10**	
This Work (Enlarged, Set 2)*	8.48M	21.72	4.90	131
This Work (Enlarged, Set 3)*		10.26	2.32	

Table G.3: **MNIST RNN latency comparisons to state-of-the-art**. (*) Using Nvidia A100 GPU acceleration, (**) using Nvidia T4 GPU acceleration, otherwise using CPU hardware.

G.4 SpeakerID RNN Evaluation

G.4.1 Effect Of Using Temperature Scale

In section 4.1.2, we discussed the problem of exploding gradients and our solution to the problem which adds temperature scaling to RNN gradients. Our experiments show that this problem is not as severe for smaller networks such as MNIST RNN, but becomes very severe for larger networks such as SpeakerID RNN. We cannot tell for sure why this may be an issue (it might not be the size of the model that causes this). It is possible that the addition of an attention layer increases the severity of this problem. A more detailed analysis is left for future work. For the following experiments, we run the fourth step of the quantization process as outlined in section 4.3.2, without OAR regularization, and adjust the temperature scale of the RNN layers only in both networks.

In table G.4, we present the results of running the MNIST RNN training with quantization for $s = 1$ (no temperature scale), to $s = 5$. The values are recorded after 2000 epochs. The gradient norms displayed are the average gradient norms for each layer and globally. The test accuracy results are also averaged. The averages are exponential moving

s	Test Accuracy	Average Gradient Norm						
		Global	RNN 0 (I)	RNN 0 (R)	RNN 1 (I)	RNN 1 (R)	Dense 0	Dense Out
1.00	87.9%	984.30	74.17	981.10	14.84	7.97	1.21	4.44
1.25	89.9%	12.84	1.43	11.45	2.58	1.67	1.12	4.16
1.50	90.6%	5.39	0.41	2.86	1.41	0.95	1.06	3.91
1.75	90.7%	4.59	0.22	1.53	1.05	0.71	1.05	3.91
2.00	91.1%	4.49	0.14	0.99	0.86	0.58	1.05	4.07
2.50	90.7%	4.29	0.08	0.53	0.63	0.43	1.04	4.01
3.00	90.7%	4.18	0.05	0.34	0.51	0.35	1.05	3.95
4.00	90.6%	4.10	0.03	0.18	0.37	0.25	1.05	3.91
5.00	90.6%	4.12	0.02	0.11	0.29	0.20	1.05	3.94

Table G.4: **Accuracy and gradient norm results for different temperature scales**. (I) and (R) indicate input and recurrent, displaying the gradient norms for $\boldsymbol{W}_{\boldsymbol{xh}}^{T}\boldsymbol{x}_{\boldsymbol{t}}$ and $\boldsymbol{W}_{\boldsymbol{hh}}^{T}\boldsymbol{h}_{\boldsymbol{t-1}}$ in the vanilla RNN cell (equation 2.4), respectively. The global gradient norm is the total norm for all gradients.

averages with smoothing weight 0.999, calculated and displayed by TensorBoard[1]. It is clear that although the gradient norms do not explode and cause the network to diverge, using no temperature scaling causes very large gradient norms, particularly in the recurrent gradients of the first RNN. The table shows that as the temperature scale is increased, the norms decrease and start to stabilize. The global norm stabilizes around a value of 4. The test accuracy also increases as we increase the temperature scale. This indicates that large gradients are detrimental to the training of an RNN.

Another phenomenon observed in this table is the continuing decrease of the RNN norms. This and the decrease in test accuracy suggests that too large of a temperature scale is also detrimental to training quantized RNNs. The temperature scale introduces vanishing gradients as it is increased, confirming our suspicions raised in section 4.1.2. Thus, finding a good temperature scale is necessary, both to gain good accuracy when training a quantized RNN, and to avoid turning an exploding gradients problem into a vanishing gradients problem. The model could benefit from a scheduler, similar to learning rate schedulers, that decays the temperature scale to 1 after a set of epochs to avoid vanishing gradients. We leave this for future work.

We do not perform the same analysis as above for the SpeakerID RNN since we have already observed the effects of increasing the temperature scale while the RNN is training. For the SpeakerID RNN, we observe a more severe problem. If we do not set a temperature scale, we cannot train the network due to divergence. The results show that after only three training steps, not epochs, the training loss and accuracy output `NaN`. By observing the gradient norms for the RNN layers in the first two training steps, we notice that the

[1] https://www.tensorflow.org/tensorboard

norms grow towards infinity, which explains the divergence. If we set the temperature scale to $s = 5$, the model converges during training. The temperature scale is therefore necessary in certain situations to enable convergence.

With respect to other works, the only relevant paper is HitNet [130] where the authors suggest adding a constant temperature scale to mitigate exploding gradients when quantizing GRU and LSTM networks. However, their implementation also applies the temperature scaling to the forward step as well. They are able to do this since their quantization process includes a mix of floating-point and fixed-point representations whereas our process does not allow us to use anything other than pure integers. Through our experimentation, we show that applying temperature scaling in the backward direction still allows us to quantize RNNs properly, which questions whether scaling in the forward direction is even necessary.

G.4.2 Accuracy Over Longer Sequences

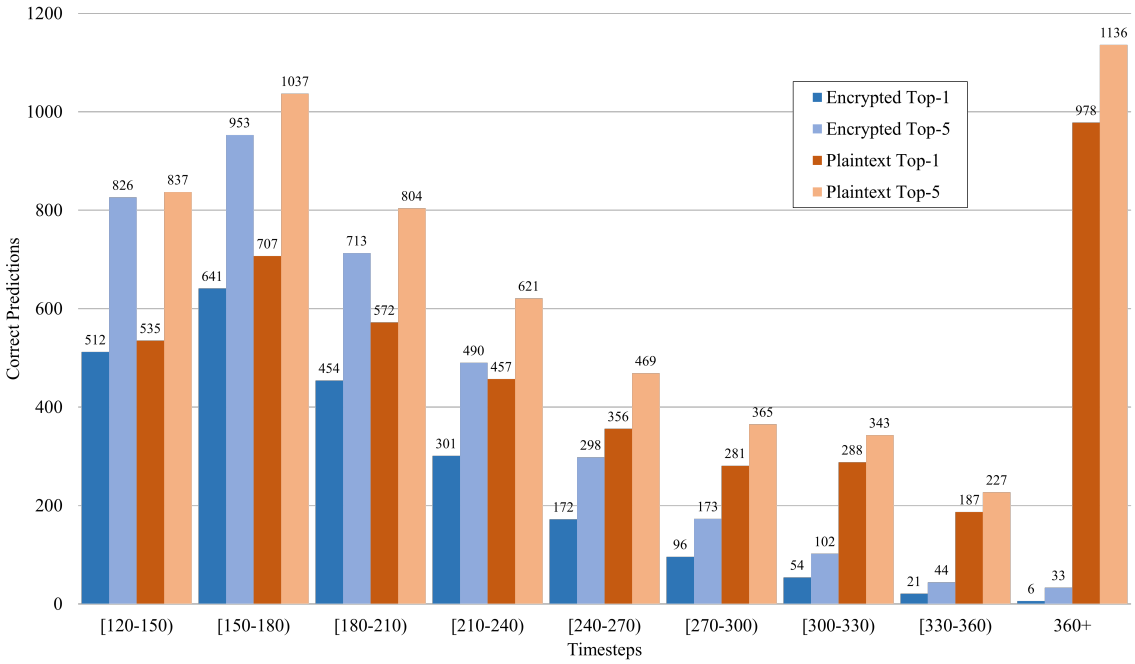


Figure G.1: **Number of correct inferences for input data of different sequence lengths using SpeakerID RNN**. Both encrypted and plaintext runs are shown. *Correct* for top-5 refers to the runs where the correct label is in the set of top-5 predictions of the run. The figure shows that the SpeakerID RNN performs well for sequence lengths up to 250 and fails for lengths past 250.

The results allow us to observe whether or not the number of timesteps in each input sequence affects the accuracy results. We compile our results in figure G.1. In this figure, we display four histograms showing the number of correct inferences per sequence length interval over the first 7382 samples of the VoxCeleb1 test dataset (which has a total of 7972 samples). The first two histograms show the number of correct top-1 and top-5 inferences

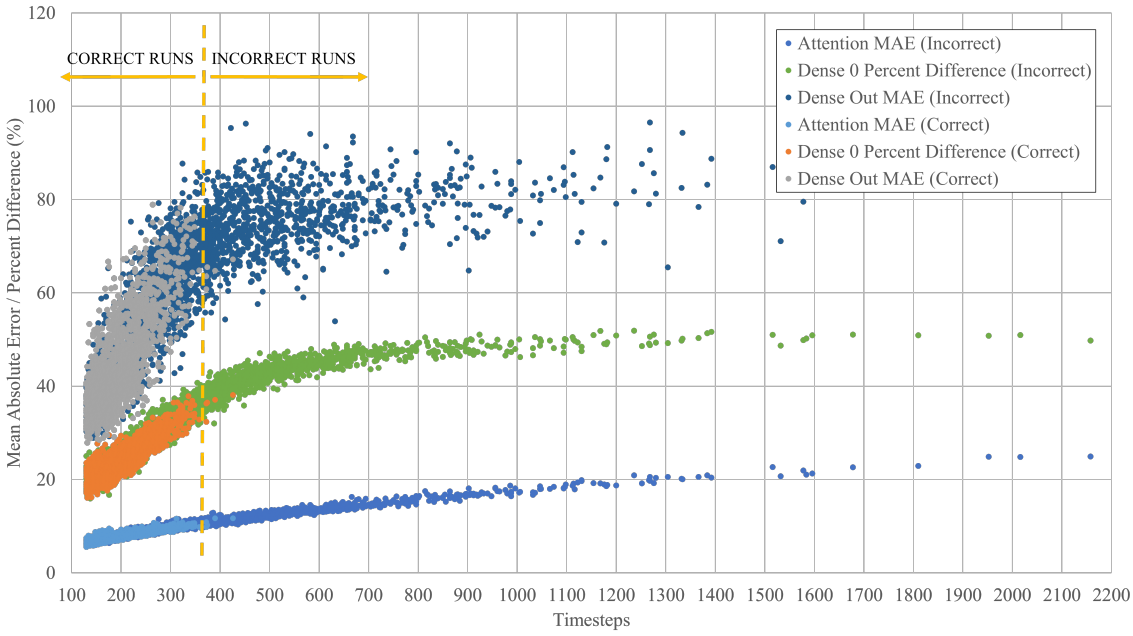


Figure G.2: **Error metrics between encrypted and plaintext run distributions per layer, per length of input sequence, for the SpeakerID RNN**. In layers where pre-activations are important, we calculate the mean absolute error. In layers where the activations are important, we calculate percent difference. The metrics are given for runs where the plaintext inference output is correct. Both correct and incorrect encrypted runs are displayed for comparison.

over encrypted data, respectively. Similarly, the last two histograms show the same results over plaintext data. The results show that encrypted runs severely underperform plaintext runs for sequence lengths greater than 360—there are 978 correct top-1 predictions over plaintext data and only 6 correct top-1 predictions over encrypted data. It is clear that pre-activations with a value of zero can sometimes cause incorrect outputs from the Signum activation function due to negacyclicity and the non-existant rounding intervals from using a precision equal to the ring dimension. In fact, because there are no rounding intervals, values around zero could also be mapped incorrectly (values such as ± 1, ± 2). Due to the sequential nature of RNNs, an error in one timestep gets propagated to future timesteps, causing an exponential increase in error. This is most likely the cause of why longer RNNs do not perform well over encrypted data.

To investigate this further, in figure G.2, we graph the sequence length with respect to MAE and percent difference in select layers of the encrypted runs to observe whether error could be causing this problem. We consider the attention layer (after the final matrix multiply), Dense 0, and Dense Out layers. The attention and Dense Out layers output accumulations due to matrix multiply operations. Thus, we calculate the MAE metric to see *how* different the distributions between the encrypted and plaintext runs are. For the Dense 0 layer, we calculate the percent difference over the activations. We plot the correct

runs against the incorrect runs, specifically those that output the correct run in the plaintext domain, to see if there is a difference. We observe that in all three metrics for both correct and incorrect runs, the error increases with the sequence length, suggesting that the RNNs are propagating errors in general (since they are the only sequential structure in the networks). We also observe that both the correct and incorrect runs follow the same trajectory. However, after the correct runs reach a certain error value, there are no more correct runs, leaving only incorrect runs. This occurs around the 300-400 timestep mark, which is what is expected, judging from figure G.1. It is evident that correct runs experience errors within a certain, smaller region. Any errors outside of that region translate to incorrect runs, suggesting that error propagation is the cause of larger sequences outputting incorrect predictions. Thus, reducing the error by either decomposing the scalar multiplication with m in the attention dot-product, or increasing the ring dimension to gain larger rounding intervals, are possible areas of future exploration. In either case, figure G.1 shows that the encrypted distribution follows the plaintext distribution quite well, except for the longer RNNs.

G.5 Concrete Benchmarks For The Programmable Bootstrap

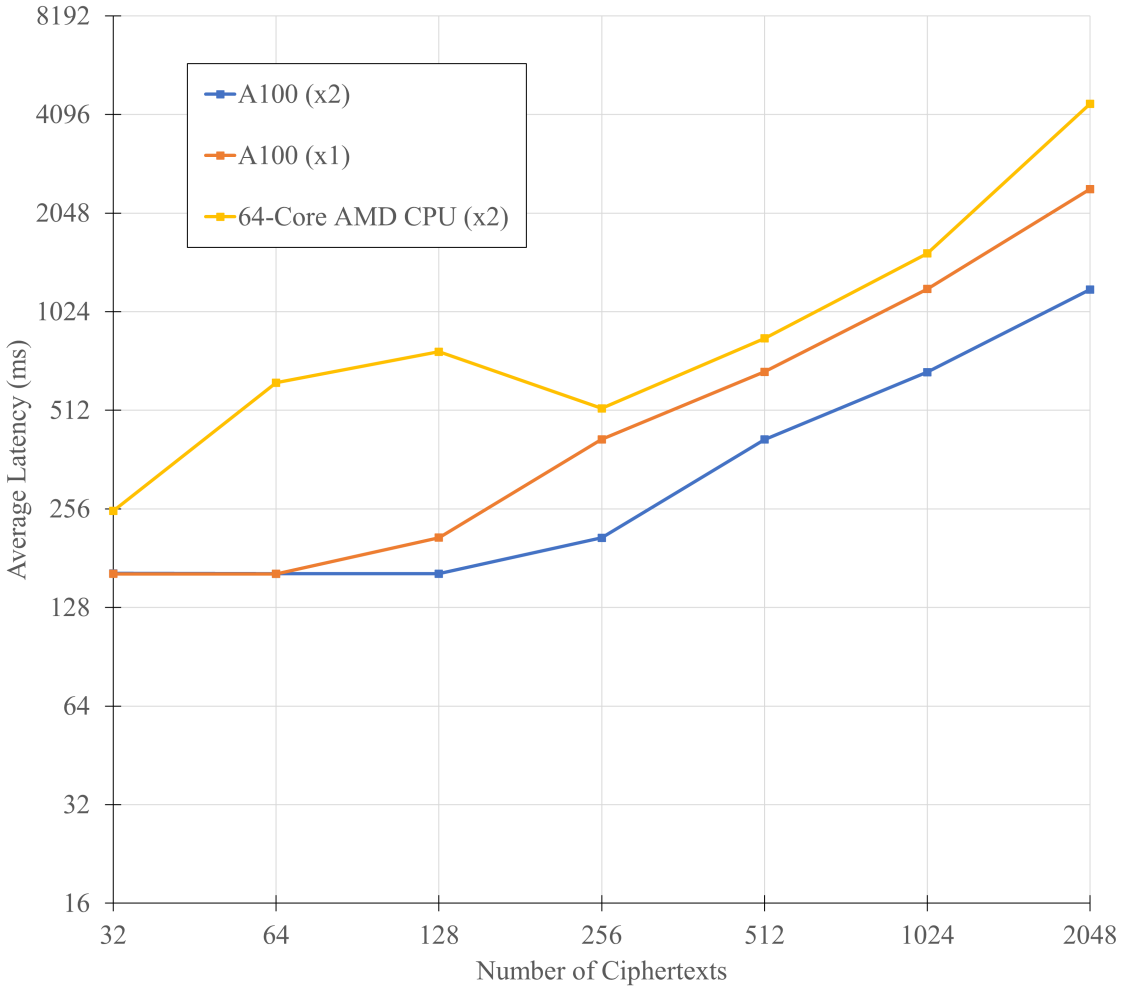


Figure G.3: **PBS latency using Concrete: Parameter Set 1**. This figure shows the benchmarking results for performing the programmable bootstrap operation using three different compute platforms (shown in the legend) and parameter set 1 from table G.2. This parameter set offers the slowest latency results out of all three parameter sets.

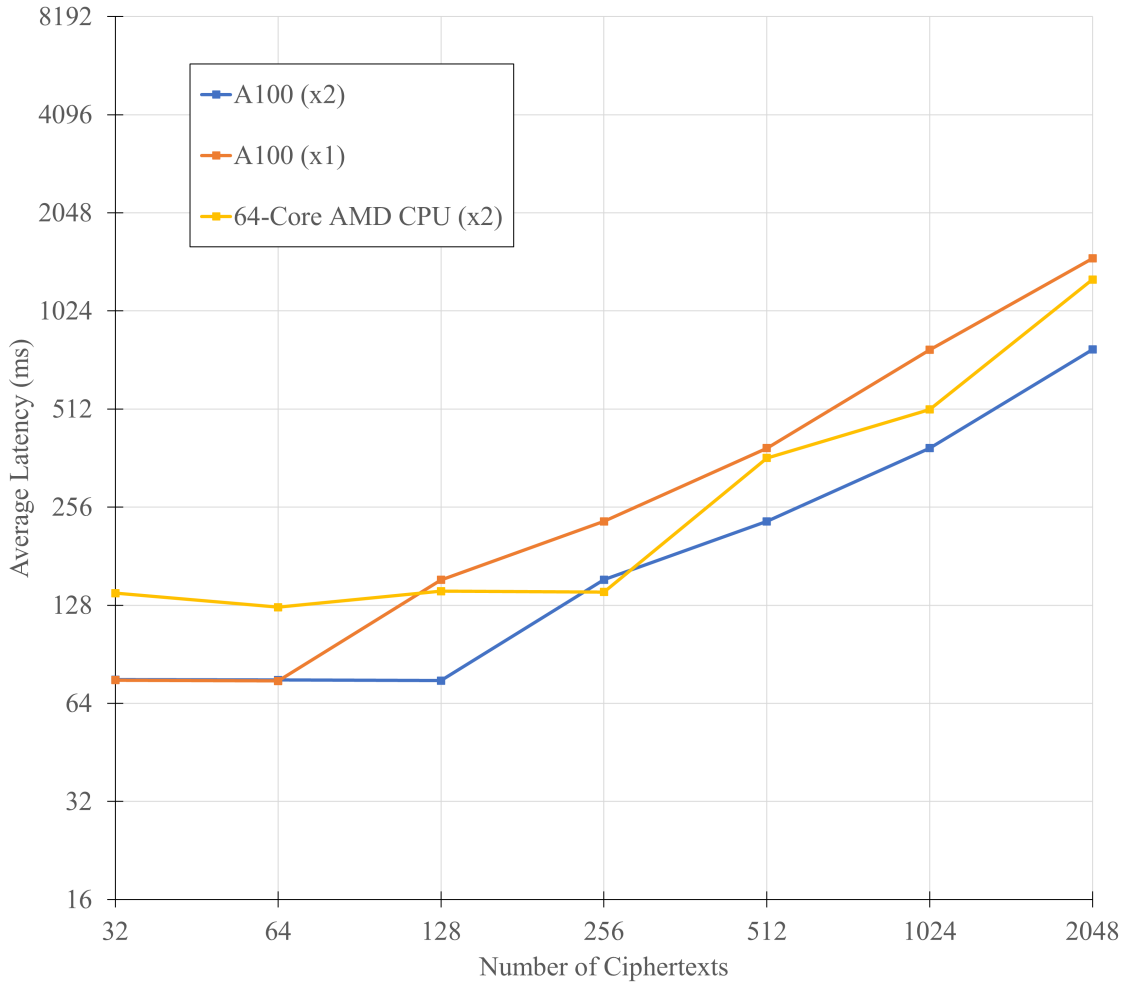


Figure G.4: **PBS latency using Concrete: Parameter Set 2**. This figure shows the benchmarking results for performing the programmable bootstrap operation using three different compute platforms (shown in the legend) and parameter set 2 from table G.2.

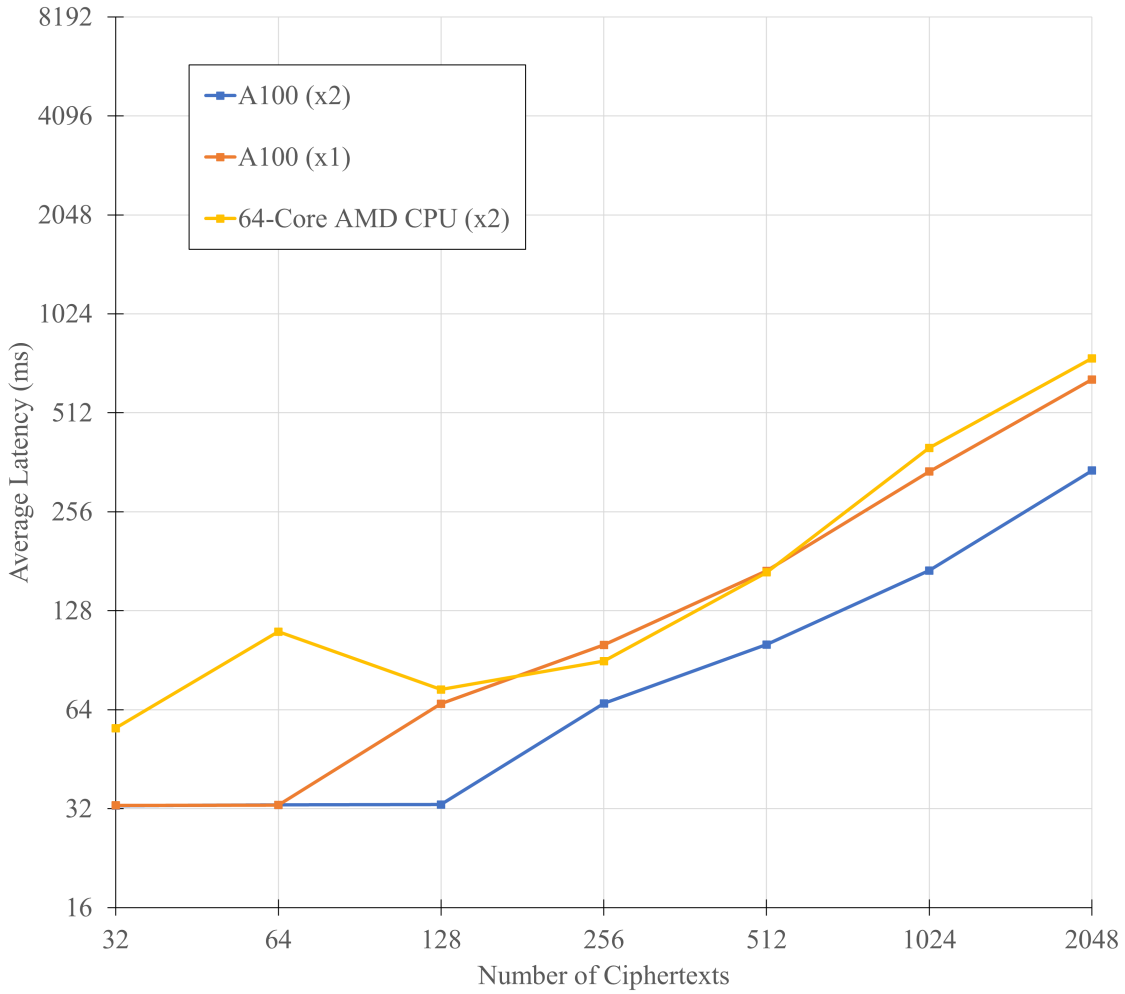


Figure G.5: **PBS latency using Concrete: Parameter Set 3**. This figure shows the benchmarking results for performing the programmable bootstrap operation using three different compute platforms (shown in the legend) and parameter set 3 from table G.2. This parameter set offers the fastest latency results out of all three parameter sets.