Secure Inference of Quantized Neural Networks

by

Haripriya Mehta

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 18, 2020

Certified by . . . . . .
Anantha P. Chandrakasan
Dean of School of Engineering, Vannevar Bush Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Secure Inference of Quantized Neural Networks

by

Haripriya Mehta

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Running image recognition algorithms on medical datasets raises several privacy concerns. Hospitals may not have access to an image recognition model that a third party may have developed, and medical images are HIPAA protected and thus, cannot leave hospital servers. However, with secure neural network inference, hospitals can send encrypted medical images as input to a modified neural network that is compatible with leveled fully homomorphic encryption (LHE), a form of encryption that can support evaluation of degree-bounded polynomial functions over encrypted data without decrypting it, and Brakerski/Fan-Vercauteren (BFV) scheme - an efficient LHE cryptographic scheme which only operates with integers. To make the model compatible with LHE with the BFV scheme, the neural net weights, and activations must be converted to integers through quantization and non-linear activation functions must be approximated with low-degree polynomial functions. This paper presents a pipeline that can train real world models such as ResNet-18 on large datasets and quantize them without significant loss in accuracy. Additionally, we highlight customized quantize inference functions which we will eventually modify to be compatible with LHE and measure the impact on model accuracy.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Dean of School of Engineering, Vannevar Bush Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Anantha Chandrakasan, for giving me this opportunity to work on an extremely cool project and for his consistent support and guidance throughout my research as well as confidence in me.

Thank you to Dr. Chiraag Juvekar, who was endlessly patient with me and would explain a topic in multiple ways until I would understand it and for helping me figure out the solution to my problems when I was stuck in my research.

Thank you to Leo, who would always take the time to meet with me and explain confusing cryptographic concepts and terms.

Thank you to Kyungmi, who walked me through working with virtual machines on GCP.

Thank you to Saroja, who taught me a lot about homomorphic encryption on our walks to Oath after our weekly meetings.

Thank you to my parents and my brother, who have always supported me throughout my academic journey and always encouraged me to do my personal best.

Thank you to Prof. Nazli Choucri and Prof. Phillip Isola, who piqued my interest in cybersecurity and computer vision respectively.

Thank you to my friends and TAs, who have helped clarify my concepts in machine learning, given me project related advice and have always been a wonderful sounding board for my ideas.

Thank you to the MIT Electrical Engineering and Computer Science Department, and my academic advisor Professor Vladimir Bulovic, for guiding me throughout my undergraduate and graduate years.

Contents

1 Introduction **13**

2 Background **17**

2.1 Convolutional Neural Networks 17

2.1.1 ResNet-18 21

2.2 Secure Neural Network Inference 24

2.2.1 Garbled Circuits 24

2.2.2 Homomorphic Encryption 26

2.2.3 BFV-compatible CNN 28

3 Related Work **29**

3.1 Multi-party Secure computation based methods 29

3.1.1 SecureML 29

3.1.2 DeepSecure 30

3.2 Homomorphic Encryption based methods 30

3.2.1 CryptoNets 30

3.2.2 CryptoDL 30

3.3 MPC and HE based methods 30

3.3.1 MiniONN 31

3.3.2 Gazelle 31

4 Training and Quantization Pipeline **33**

4.1 Training 34

4.1.1 Dataset 35
4.1.2 Building an Estimator 36
4.1.3 Model 37
4.1.4 Quantization 38
4.1.5 Results 39

5 Plaintext Inference of Quantized Model 41
5.1 Code Validation 45
5.2 Replicating Operators in Python 46
5.2.1 Quantize Operator 46
5.2.2 Add Operator 47
5.2.3 Other Operators 52

6 Conclusions and Future Work 55
6.1 Future Work 55

A Garbled Circuits and Secure Yelp 57
A.1 Formulation 57
A.2 Circuit Equivalent 58
A.3 Circuit as a DAG 60
A.4 Results 62

List of Figures

1-1 The hospital sends encrypted images as input to the modified model. The encrypted output is then decrypted and the hospital obtains classification of inputted image. 14

1-2 Our training and quantization pipeline to produce a modified model that can take in encrypted input. The model, as is, is not compatible with LHE with BFV scheme due to floating point weights and activations and presence of non-linear layers. Modifications to the model, including post-training quantization and customizing non-polynomial layers, according to methods presented in the thesis, are required for LHE and BFV compatibility. 15

2-1 Example of different types of layers found in convolutional neural networks. 18

2-2 An example of 1D Convolution. Point-wise multiplication and accumulation is performed by shifting the filter (shown in blue) across the tensor (shown in peach). We can add zero padding to ensure that the size of the feature is maintained, as seen in steps 1 and 5. 19

2-3 Example of max pooling layer given stride 2 and 2×2 filters. 20

2-4 Example of fully connected layer with 2 image classes. W represents the weights and b represents the bias. 21

2-5 The Identity Block is shown in (a). The Convolutional Block is shown in (b). Both are types of residual connections found in ResNet. 23

2-6 An OR gate with wires and labels are shown in (a). The truth table of the OR gate is shown in (b). A will first replace the 0s and 1s in the truth table with the labels as shown in (c) and then encrypt the circuit by creating a garbled truth table as shown in (d). 25

4-1 Training and Quantization Pipeline. 34

5-1 Portion of the quantized model shown in Netron. 43

5-2 Properties of Conv2D operator shown in Netron. 44

5-3 The steps taken in the Add Operator. To obtain the final output, we clip the raw output to be between and including the activation minimum and maximum values that we establish. 48

A-1 Our boolean circuit to find closest restaurants securely. A_x and A_y represent the user's location, B_x and B_y represent the restaurant's location and C_x and C_y represent the output, i.e. the nearest restaurant to the the user's location. 60

A-2 A circuit with two AND gates shown in(a). The associated directed acyclic graph with the circuit is shown in (b) 61

List of Tables

2.1 ResNet-18 Architecture 22

4.1 Pros and Cons of using Post Training Quantization 33

4.2 Comparison of accuracy of unquantized and quantized models 39

A.1 100 restaurant in the database 62

Chapter 1

Introduction

In the past couple of decades, deep learning has seen many breakthroughs specifically in the field of image recognition owing to larger datasets, invention of GPUs[17] and TPUs[15] and better algorithms. Due to its success in several other domains, image recognition can also be used to identify potential diseases in medical images to optimize tasks that normally doctors would have undertaken such as detecting breast cancer in mammograms[26]. However, important privacy concerns arise in the medical sector.

Hospitals may not have access to an image recognition model, that a third party, e.g. a startup, may develop. Sending the HIPAA protected medical images to the cloud as input to a third party neural network would violate patient privacy. Selling the model to the hospital would also not be feasible as it would prevent the third party from monetizing the model any further once the model is given away. Furthermore, the model may reveal private information from patient data that the model has been trained on, violating patient privacy.

Hospitals can, instead, encrypt the medical images using a secret key and send these encrypted images to the cloud as seen in Figure 1-1. The original neural network, which has been pretrained on unencrypted images, however, must be modified so that it can take an encrypted image as an input and run inference on this image. The encrypted prediction can then be sent to the hospital, where it can then be decrypted with the secret key.

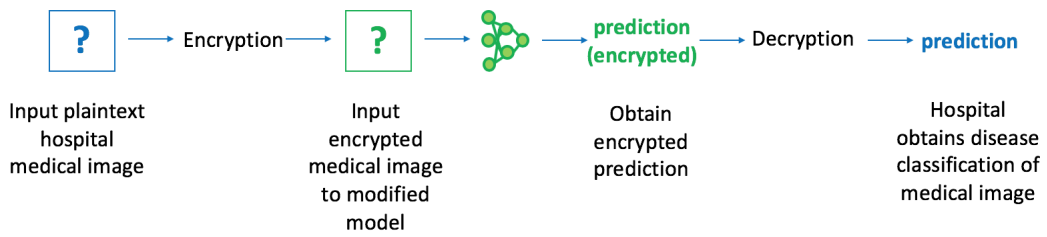


Figure 1-1: The hospital sends encrypted images as input to the modified model. The encrypted output is then decrypted and the hospital obtains classification of inputted image.

Leveled fully homomorphic encryption (LHE) is a form of encryption that can support evaluation of degree-bounded polynomial functions on ciphertext (encrypted data) without having access to the secret key or decrypting the data. An efficient LHE cryptographic scheme is the Brakerski/Fan-Vercauteren (BFV) scheme which only operates with integers. In order to create a secure neural network inference framework that is compatible with the BFV scheme, we developed a pipeline (seen in Figure 1-2) that first, takes in a dataset and trains large models such as ResNet-18 in the typical manner using floating point arithmetic. Then the model is quantized to have 8 bit integer weights and activations. We also customized quantized inference functions so that, in the future, we can make the model fully compatible with LHE by approximating non-linear activation functions with low-degree polynomial functions and measure the impact of modified functions on model accuracy.

I conducted my work at Microsystems Technology Laboratories (MTL) in collaboration with Dr. Chiraag Juvekar, research affiliate at MTL and Mr. Leo de Castro, graduate student at CSAIL. I will focus on my individual contributions in this thesis:

In the following chapters, I will:

- Provide background on deep neural networks and relevant cryptographic primitives (Chapter 2).
- Present related work on secure neural network inference(Chapter 3).
- Detail the components of the training and quantization pipeline (Chapter 4).
- Highlight custom implementation of quantized inference functions (Chapter 5).

- Discuss potential next steps of the project(Chapter 6).

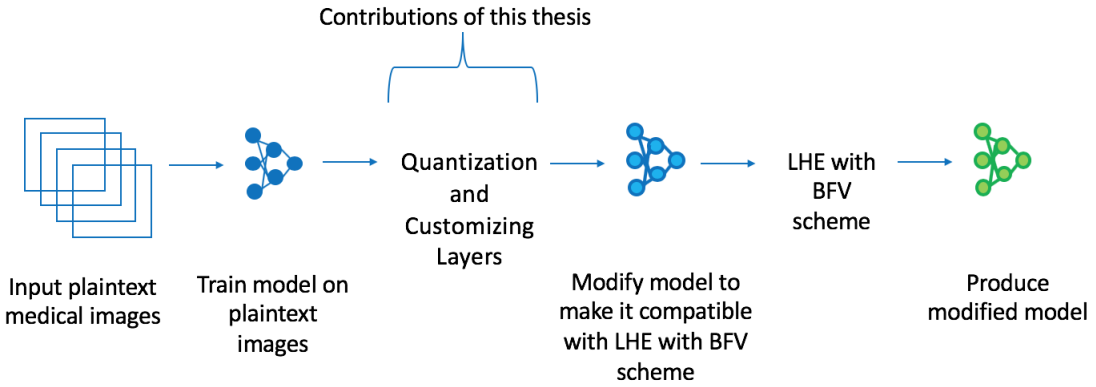


Figure 1-2: Our training and quantization pipeline to produce a modified model that can take in encrypted input. The model, as is, is not compatible with LHE with BFV scheme due to floating point weights and activations and presence of non-linear layers. Modifications to the model, including post-training quantization and customizing non-polynomial layers, according to methods presented in the thesis, are required for LHE and BFV compatibility.

Chapter 2

Background

Secure neural network inference on medical images involves two major components - an underlying neural network for image recognition as well as the cryptographic primitives used to transform the neural network such that an accurate encrypted prediction is outputted for an encrypted image.

Convolutional neural networks (CNN) are a particular type of neural networks used for image recognition and other applications. We can transform convolutional neural networks using cryptographic primitives to a form that can run inference on encrypted images, as we will see in the following section. We focus on ResNet-18, an example of a CNN that we tested our quantization pipeline and inference functions on, and two cryptographic primitives that are used in Gazelle [16] to modify the CNN - garbled circuits and homomorphic encryption.

2.1 Convolutional Neural Networks

CNNs are similar to feed forward networks in that they have learnable weights and biases. However, they fare better than feed forward neural networks for image recognition, as CNNs can capture the spatial and temporal dependencies due to weights being shared. Figure 2-1 shows an example of a CNN.

The input to a CNN is an image represented as a 3-D tensor with shape `[width, height, depth]`. The depth is the number of channels in the image. An RGB image

has 3 channels, for example.

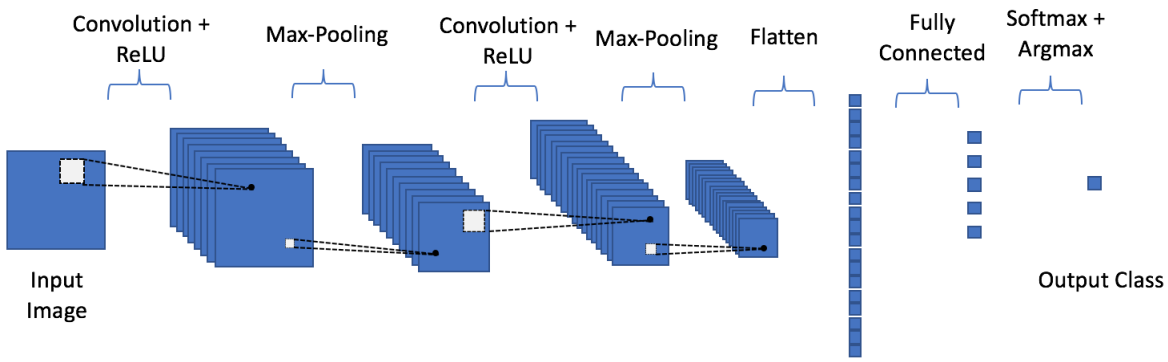


Figure 2-1: Example of different types of layers found in convolutional neural networks.

The following layers are common to CNNs, even though their sequencing and parameters may vary between different CNNs. Each layer transforms a tensor of activations to another tensor through a differentiable function.

1. **Convolutional Layers**

   In the convolutional layer, we perform point-wise multiplications and accumulations by shifting a square filter across the input image or output of a previous layer. If the stride is greater than one, the filter will skip some pixels. The depth of the output of this layer is equal to number of filters convolved with the image. Zero padding may be required prior to convolution if the size of each dimension has reduced, in order to maintain the height and width of the output shape. A 1-D example of a convolutional layer, which can be easily extended to multi dimensions, is seen in Figure 2-2. This layer is a linear layer.

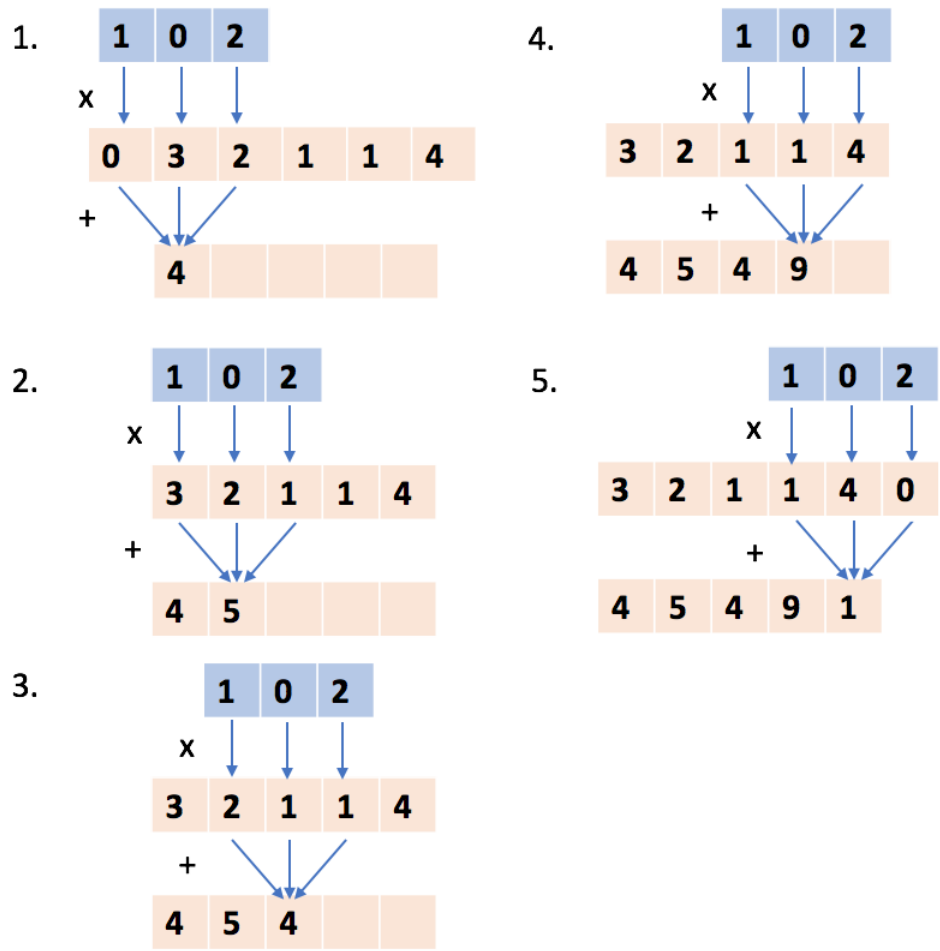


Figure 2-2: An example of 1D Convolution. Point-wise multiplication and accumulation is performed by shifting the filter (shown in blue) across the tensor (shown in peach). We can add zero padding to ensure that the size of the feature is maintained, as seen in steps 1 and 5.

2. **ReLU**

   Rectified Linear Unit (ReLU) is a non-linear layer that outputs $max(0, x)$ with input x. This elementwise activation function is generally applied to the output of the convolutional layer.

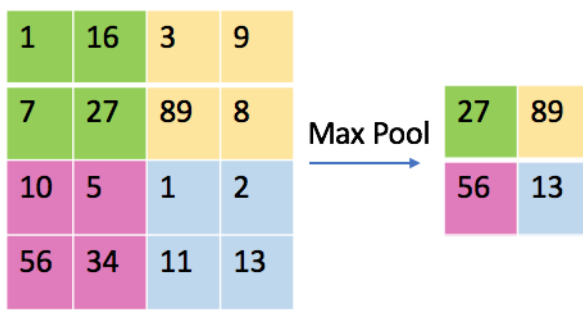


Figure 2-3: Example of max pooling layer given stride 2 and 2 × 2 filters.

3. **Pooling Layers**

   Pooling layers decreases the size of the convolved feature. Two most common types of pooling are MaxPooling and AvgPooling. In MaxPooling, the maximum value is returned from designated portions of the feature as seen in Figure 2-3, while in AvgPooling, the average value is returned. Pooling is a non-linear function.

4. **Fully Connected Layers**

   The intermediate output is flattened following the final pooling layer. This flattened layer is then densely connected to the fully connected layer, which has as many nodes as there are image classes that we have trained on. Figure 2-4 shows an example of a fully connected layer. y_1 and y_2 are obtained as follows:

$$y_1 = x_1 w_{1,1} + x_2 w_{2,1} + x_3 w_{3,1} + x_4 w_{4,1} + b_1$$

$$y_2 = x_1 w_{1,2} + x_2 w_{2,2} + x_3 w_{3,2} + x_4 w_{4,2} + b_2$$

   This layer is thus, a linear layer.

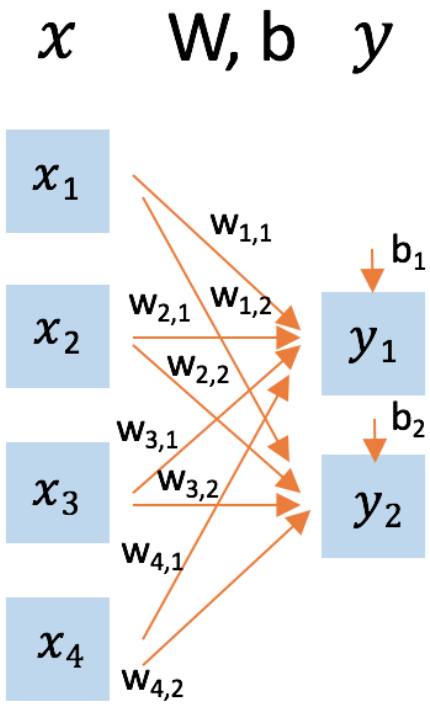


Figure 2-4: Example of fully connected layer with 2 image classes. W represents the weights and b represents the bias.

Finally, a non-linear activation function such as softmax is applied to the output of the last fully connected layer. Softmax normalizes the output into a probability distribution. The node with the highest probability determined using argmax is the predicted class of the image.

2.1.1 ResNet-18

The ImageNet Large Scale Visual Recognition Challenge[22] was an annual competition from 2010 and 2017 that involved the ImageNet dataset - a hand annotated dataset that consists of over 14 million images from over 21 thousand different categories. Each year, a training dataset consisting of over million images from thousand different classes was released for the image classification task.

These competitions brought about many state-of-the-art CNN implementations for image recognition including AlexNet[18], Inception[24], VGG[23] and ResNet[13],

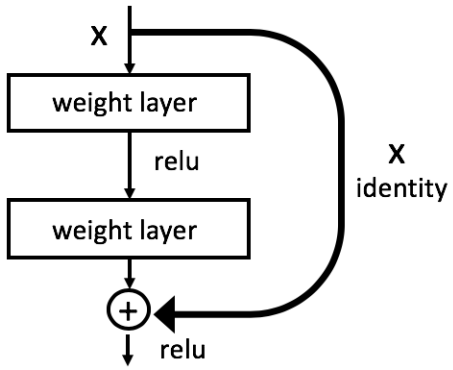
some of which even performed better at classification than the human eye[22]. In this thesis, we focused on testing our quantization pipeline with ResNet-18, the smallest ResNet model, which has eighteen layers, since that would be the fastest ResNet architecture to train. It is important to note, however, that our pipeline is also compatible with larger ResNet models such as ResNet-34, ResNet-50 and ResNet-101.

One major issue with deep neural networks is the problem of vanishing or exploding gradients. The way ResNet combats this problem is by residual or skip connections, which provide a path to propagate the gradient directly between non-adjacent layers. There are two types of residual blocks found in ResNet - identity block and the convolutional block as seen in Figure 2-5.

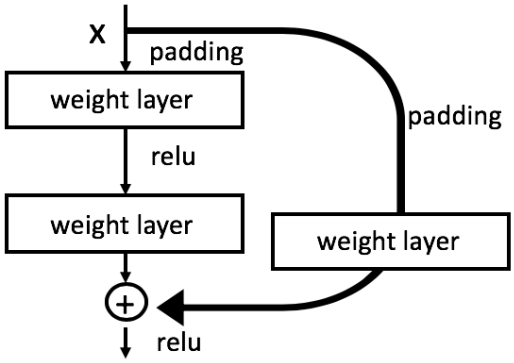
The identity block is used when the input activation has the same dimension as the output activation. The convolutional block is used when the dimensions are different. In ResNet-18, every convolutional block is followed by the identity block. The two blocks are represented by $\times 2$ in Table 2.1 where the entire architecture of the ResNet-18 is shown.

Table 2.1: ResNet-18 Architecture

Layer name	Output width and height	Filter size/Number of Filters
conv1	112×112	7×7, 64, stride 2
conv2	56×56	3×3 max pool, stride 2
		$\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 2$
conv3	28×28	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$
conv4	14×14	$\begin{bmatrix} 3\times 3, 256 \\ 3\times 3, 256 \end{bmatrix} \times 2$
conv5	7×7	$\begin{bmatrix} 3\times 3, 512 \\ 3\times 3, 512 \end{bmatrix} \times 2$
	1×1	average pool, 1000-d f.c. softmax



(a) Identity Block



(b) Convolutional Block

Figure 2-5: The Identity Block is shown in (a). The Convolutional Block is shown in (b). Both are types of residual connections found in ResNet.

2.2 Secure Neural Network Inference

Sending plaintext (unencrypted) medical images to a third-party violates patient privacy. These images must be encrypted with a cryptographic algorithm and an encryption key, which only the hospital has access to, before it is sent to the third-party CNN. The CNN, which has been trained on plaintext images, will not yield a correct prediction for the encrypted image, as is. The layers of the CNN must be modified such that the layers can compute over encrypted data. After the modified CNN is run on an encrypted medical image, the output of the modified CNN must be decrypted by the hospital using the encryption key to obtain the predicted classification of the medical image.

Two techniques which enable evaluating over encrypted data without decrypting the data include secure multi-party computation and homomorphic encryption.

2.2.1 Garbled Circuits

In secure multi-party computation, parties jointly compute a function over inputs that each party keeps private. The garbled circuit protocol [27] is a cryptographic protocol that is an approach to two-party secure computation, a subfield of secure multi-party computation which only involves two parties. A boolean circuit is used to express the function that will be computed using the garbled circuit protocol.

This protocol uses one-out-of-two oblivious transfer protocol, which involves a sender A with two messages, x_0 and x_1 and receiver B choosing one. B will pick an index i from $\{0, 1\}$ and A will send x_i without A knowing which index B selected and B not knowing the other message. The garbled circuit protocol is only limited to 2-input gates since A has two messages, from which B must choose.

The protocol is as follows:

1. The full circuit is known to both parties A and B.

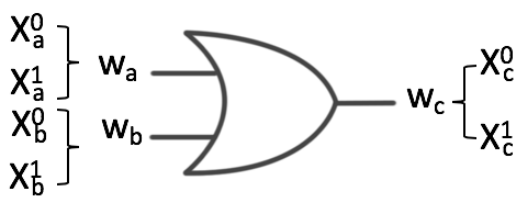
2. A garbles each logic gate in the circuit similar to Figure 2-6, which shows garbling of the OR logic gate. For each logic gate, he/she replaces 0s and 1s

with labels in the truth table and encrypts each output entry with labels of the two corresponding inputs. This new table is called the the garbled table. After A randomly permutes the rows in the garbled table for each logic gate, he/she sends each garbled table and the labels for his/her input to B.

3. B does not know the labels for his/her input. Through oblivious transfer, B will work with A to receive the labels for his/her input.

4. B evaluates the circuit and obtains the encrypted outputs.

5. A and B work together to learn the output.



(a) OR gate

w_a	w_b	w_c
0	0	0
0	1	1
1	0	1
1	1	1

(b) Truth table

w_a	w_b	w_c
X_a^0	X_b^0	X_c^0
X_a^0	X_b^1	X_c^1
X_a^1	X_b^0	X_c^1
X_a^1	X_b^1	X_c^1

(c) Truth table with labels

Garbled Table
$Enc_{X_a^0, X_b^0}(X_c^0)$
$Enc_{X_a^0, X_b^1}(X_c^1)$
$Enc_{X_a^1, X_b^0}(X_c^1)$
$Enc_{X_a^1, X_b^1}(X_c^1)$

(d) Garbled truth table

Figure 2-6: An OR gate with wires and labels are shown in (a). The truth table of the OR gate is shown in (b). A will first replace the 0s and 1s in the truth table with the labels as shown in (c) and then encrypt the circuit by creating a garbled truth table as shown in (d).

Garbled circuits are more efficient than other cryptographic primitives such as homomorphic encryption for non-linear computations[16]. However, the downside

to garbled circuits is that they are not reusable, i.e. the circuit can only be used once. Using the circuit more than once compromises the secrecy of the circuit[12]. Thus, due to poor scalability and efficiency, as well as memory constraints, garbled circuits cannot support the network bandwidth of large real world models such as ResNet-18[25].

Appendix A describes the computation of a non-linear function using garbled circuits.

2.2.2 Homomorphic Encryption

Homomorphic Encryption is a cryptographic primitive that can compute over encrypted data without access to the secret key. If $E(m)$ is the encryption of the message m, additive homomorphism is as follows:

$$E(m_1) + E(m_2) = E(m_1 + m_2)$$

Multiplicative homomorphism is as follows:

$$E(m_1) * E(m_2) = E(m_1 * m_2)$$

Fully homomorphic encryption (FHE)[11] is a type of homomorphic encryption that can support addition and multiplication on circuits of unbounded depth due to bootstrapping. In existing FHE schemes, ciphertext (encrypted data) contains some noise. Each additional computation over the data adds to the noise. If the noise crosses a certain threshold, decryption will not work correctly. With bootstrapping, however, the ciphertext is refreshed after each computation in order to reduce the noise and allow further operations to be performed.

Leveled fully homomorphic encryption (LHE)[7] also supports addition and multiplication, but only successfully runs on circuits of bounded depth as it does not use the very costly technique of bootstrapping, accumulating noise. Partially homomorphic encryption schemes only accomplish addition such as packed additively

homorphic encryption schemes (PAHE) or multiplication, but not both[9].

In this thesis, we focus on LHE since it is not as computationally expensive as FHE and also more powerful than PAHE. One of the main cryptographic schemes present within LHE is Brakerski/Fan-Vercauteren (BFV) [10], which only works with integers. However, neural networks have floating points weights and activations. As a result, we must quantize the network to make it compatible with BFV.

BFV

Instantiating BFV requires specifying the following parameters[2] - polynomial ring $R = \mathbb{Z}[x]/f(x)$ where $f(x) = x^d + 1$ and $d = 2^n$, ciphertext modulus q, plaintext modulus p, $\Delta = \lfloor \frac{q}{t} \rfloor$ where t is an integer between 1 and q, ring R_q and a noise distribution χ from R.

The secret key sk is generated by sampling an element s uniformly at random from χ.

The public key is generated by sampling an element a uniformly at random from R_q and error e from χ to obtain following public key:

$$pk = ([-(a \cdot s + e)]_q, a)$$

A message m is encrypted by parsing $pk[0]$ and $pk[1]$. Given u, e_1, e_2 sampled from χ, we output the following ciphertext:

$$ct = ([pk[0] \cdot u + e_1 + \Delta \cdot m]_q, [pk[1] \cdot u + e_2]_q)$$

The ciphertext can be decrypted by setting s to sk and computing

$$\left[\left\lfloor \frac{t \cdot [ct[0] + ct[1] \cdot s]_q}{q} \right\rceil\right]_t$$

The elements of ciphertext can be interpreted as coefficients of a polynomial as follows:

$$[ct[0] + ct[1] \cdot s]_q = \Delta \cdot m + v$$

where v is the noise in the ciphertext and

This scheme defines two functions, $\mathsf{EvalAdd}$ and $\mathsf{EvalMult}$ such that

$$\mathsf{EvalAdd}(\mathsf{ct}_1, \mathsf{ct}_2) = \Delta \cdot [m_1 + m_2]_t + v$$

$$\mathsf{EvalMult}(\mathsf{ct}_1, \mathsf{ct}_2) = \Delta \cdot [m_1 \cdot m_2]_t + v$$

Thus, given two ciphertexts, we can compute the sum or product over these ciphertexts.

2.2.3 BFV-compatible CNN

As aforementioned, linear layers such as convolution layers and fully connected layers involve multiplication and addition of tensors. They can, thus, be supported by homomorphic encryption[16].

However, non-polynomial activation functions such as ReLU cannot be supported unless such activation functions are approximated by low-degree polynomials[8]. To make our neural network fully compatible with BFV, we have to modify our quantized inference functions. As a result, it is important that we have an implementation of quantized inference functions where we are easily able to modify the activation functions and test how different activation functions affect the accuracy. In the following chapters, we will explore how we quantized a network to be compatible with BFV scheme and how we customized the quantize inference functions.

Chapter 3

Related Work

Much work has been done in the field of secure neural network inference [25][20][8]. Papers written regarding privacy-preserving deep learning mostly use multi-party secure computation (MPC) or homomorphic encryption (HE) or a combination of the two. We highlight some of the major papers and some of their strengths and drawbacks.

3.1 Multi-party Secure computation based methods

Methods based on multi-party secure computation require interactivity between the parties involved.

3.1.1 SecureML

SecureML[20] uses oblivious transfer, garbled circuits and secret sharing, a method where each party receives only a share of the secret to preserve the privacy of neural network inference. This paper implements a simple neural network without any convolutional layers.

3.1.2 DeepSecure

Deep Secure[21] uses oblivious transfer and garbled circuits and works with convolutional neural networks. The drawback of using garbled circuits for secure inference, as mentioned in the previous chapter, is that they cannot support real world models.

3.2 Homomorphic Encryption based methods

The downside to multi-party secure computation based methods, such as SecureML and DeepSecure, is high network latency and high bandwidth usage[5]. We instead opt to use homomorphic encryption based methods. Following are some papers that use homomorphic encryption for secure neural network inference:

3.2.1 CryptoNets

CryptoNets [8] was the first example of secure neural network inference using homomorphic encryption. It specifically uses LHE with Brakerski-Gentry-Vaikuntanathan scheme (BGV)[6], a scheme like BFV that only works with integers. CryptoNets uses square function to approximate ReLU activation function prior to training. However, accuracy is not very high for very deep neural networks where number of activation layers and other non-linear layers is large.

3.2.2 CryptoDL

CryptoDL[14] uses low degree polynomials to approximate activation functions in CNNs, thereby improving upon the bandwidth per inference and latency.

3.3 MPC and HE based methods

There are some methods that use a combination of multi-party secure computation and homomorphic encryption.

3.3.1 MiniONN

MiniONN[19] uses garbled circuits and secret sharing for inference and a weak form of lattice-based additively homomorphic encryption for precomputation. Modifications to activation function take place prior to inference and not prior to training, as in CryptoNets.

3.3.2 Gazelle

Gazelle[16] greedily optimizes the cryptographic primitives used in the neural network by using garbled circuits for non-linear layers such as activation and pooling layers and PAHE for linear layers such as convolutional and fully connected layers. It achieves three orders of magnitude faster run-time than CryptoNets and 20x faster than MiniONN.

As in other garbled circuit based methods, Gazelle cannot support the network bandwidth of large real world networks such as ResNet-18. Running secure neural network inference with real world networks will potentially be much more feasible with the contributions of this thesis, which are as follows:

1. **Training and Quantization pipeline**
   We can train real world neural networks on large datasets in TFRecord format on TPU and then quantize the model in order for it to be compatible with BFV(a LHE scheme which only works with integers) without a significant loss in accuracy. Unlike other secure neural network inference methods such as CryptoNets which involve the modification of non-polynomial activation functions to low-degree polynomials prior to training, we train models without modification and thus, ensure higher accuracy on real world models with several activation layers.

2. **Customized quantize inference functions**
   With customized quantize inference functions, we will easily be able to approximate non-linear activation functions with low-degree polynomial functions prior to inference for the model to be compatible with homomorphic encryption.

We detail our pipeline and inference functions in the following chapters.

Chapter 4

Training and Quantization Pipeline

We needed to quantize the weights and activations to make the neural network compatible with BFV. We first had to determine the quantization method we wanted to use as that would influence how we would train our model.

Two main quantization methods are found in popular Python machine learning libraries PyTorch[4] and Tensorflow[3]:

1. Quantization-aware – Quantization takes places during training

2. Post-training quantization – Quantization takes place after training

In quantization - aware training, the model is trained with floating point weights and activations. Quantization error is modeled in both forward and backward passes using fake quantization modules. After the model has been trained, it is converted to a quantized model with conversion functions provided by Python machine learning

Table 4.1: Pros and Cons of using Post Training Quantization

Pros	Cons
Easy to use	Lower model accuracy than Quantization Aware Training
Only small unlabeled dataset required for calibration	

libraries. Though quantization-aware methods often yield a higher accuracy, post-training quantization methods are more standardized and can be easily applied to any model without any customization. Post-training quantization only requires a small set of unlabelled data for calibration whereas, quantization-aware training requires the entire labelled training data set. Furthermore, we found that quantizing the model post training did not contribute to a significant loss in accuracy. Table 4.1 summarizes the pros and cons of using post-training quantization as opposed to quantization-aware methods.

We initially worked with Pytorch's implementation of post-training quantization. However, training the model for additional epochs post quantization was required in order to improve the accuracy of the model. In contrast, Tensorflow's implementation successfully quantized the network after training with just a few lines of code. As a result, we opted to use Tensorflow's post-training quantization implementation.

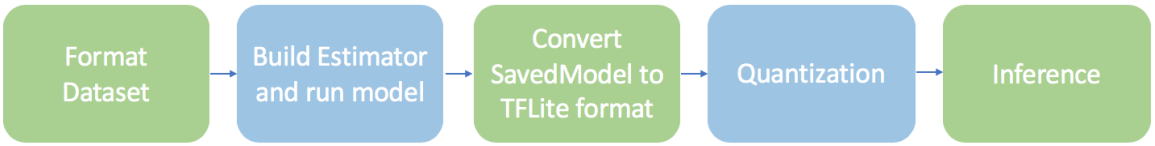


Figure 4-1: Training and Quantization Pipeline.

4.1 Training

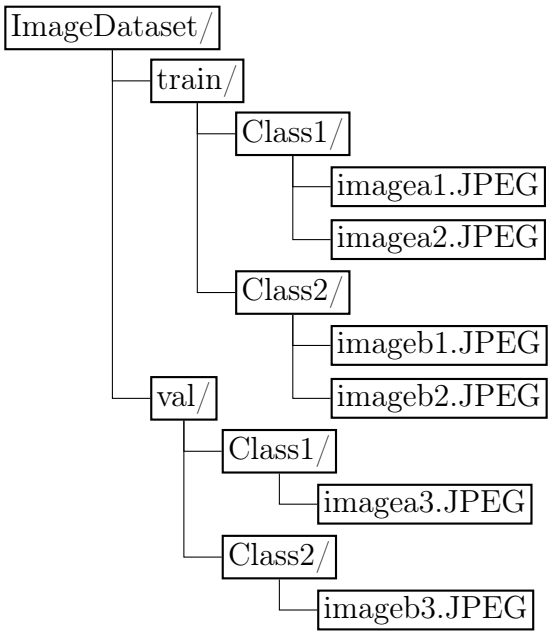
When working with large datasets and real world networks, running the model on CPU or even GPU is not optimal as it can take days for the network to train. Training ResNet-18 for one epoch on a single GPU on ImageNet dataset took us 1.6 hours. Training ResNet-18 for 200 epochs would take around 13 days. Google Cloud Platform's (GCP) TPU (Tensor Processing Unit) can shorten the full training time to hours, if the dataset and model are in the appropriate formats. TPU is optimized for Tensorflow giving us further reason to choose Tensorflow over PyTorch. PyTorch code needs to be compatible with XLA to run on TPU. We found that modifying the

code added additional and unnecessary complexity and we constantly ran into errors.

For the sake of illustration, we describe the process we took on training ResNet-18 with ImageNet on `v3-8` TPU, a TPU node with 8 TPU v3 cores. We ran the code on a GCP Debian virtual machine preinstalled with TensorFlow 1.15. We show our entire training and quantization pipeline in figure 4-1.

4.1.1 Dataset

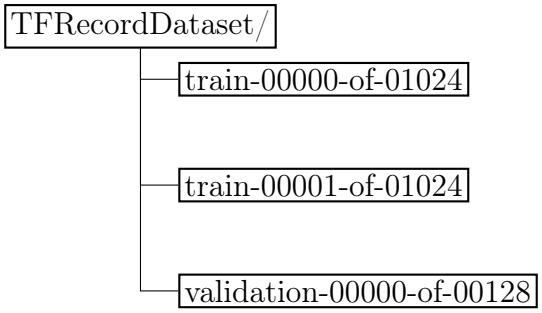
The plaintext image dataset we trained on was organized as follows:



Datasets such as ImageNet, however, are quite large containing over a million images in its training set. Feeding in images to the TPU when the dataset is in the format described above takes a lot of time and makes the TPU extremely inefficient. To speed up the time it takes for the TPU to read the images, we convert the image dataset that is organized in the above file structures to TFRecord format using a script on GPU. The TFRecord format serializes the data and allows the data to be read linearly. It is important to split the dataset into an appropriate number of TFRecord files. Generating one TRFrecord file for the training dataset and one for the validation dataset prevented the TPU from being efficient. For ImageNet, we divided training image set into 1024 files and validation image set into 128 files.

The TFRecord files that were fed into the TPU was organized in the following file structure:

The file structure for the TFRecord files was as follows:



4.1.2 Building an Estimator

In order to make training and subsequent quantization process streamlined, it was important that our ResNet-18 model was in the TensorFlow Estimator format[1]. The Estimator format involves four different actions and only runs on CPU and GPU:

- Training
- Evaluation
- Prediction
- Export for Serving

Since the model and datasets we worked with were large and required the use of TPU, the estimator was converted to TPUEstimator object. This required only the revision of a few lines of code.

It took 3-5 minutes to train each epoch of ResNet-18 on ImageNet on TPU using the TFRecord dataset format with a TPUEstimator object. The model was trained for 200+ epochs and a checkpoint was saved at every epoch. It achieved 70.512% accuracy on 1000 classes of ImageNet.

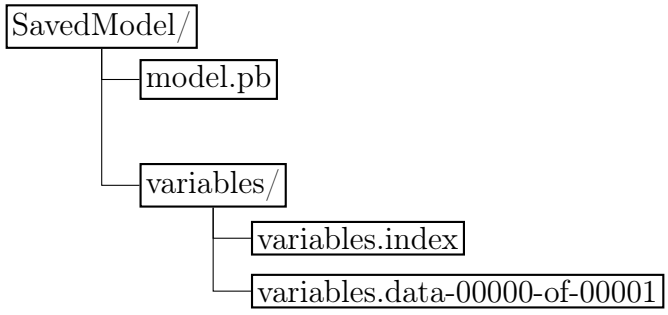
After the last epoch, the model was saved to the SavedModel format using a serving function. Since the model in the SavedModel format was going to be quantized,

it was important that all functions in the model were quantizable. All ResNet-18 functions such as padding, pooling and convolution are standard functions that can be quantized with TensorFlow Lite(a library that optimizes Tensorflow code for small devices such as mobile phones and IoT devices). However, any functions used for image preprocessing such as cropping and resizing are not supported for quantization. As a result, our serving function did not have any preprocessing code and only took in an image in the NHWC format `(batch number, height of image, width of image, number of channels)`. In our particular case, we used `(None, 224, 224, 3)` to have the same height, width and number of channels as the images we trained on.

The functions used to normalize the images (subtracting the mean, μ and dividing by the standard deviation, σ) was part of our model training code and thus, did not have to be included in the serving function either. Subtraction is quantized in TensorFlow Lite, but division is not. Instead, we multiplied by $\frac{1}{\sigma}$ when training the model as multiplication is quantizable.

4.1.3 Model

The model was saved in the SavedModel format which has the following file structure.



The SavedModel format can be automatically converted to the TFLite format, without any additional information. Once our model was in TFLite format, we were able to quantize it using the code in TFLite library.

4.1.4 Quantization

Types of post-training quantization methods available in TensorFlow Lite Library include dynamic range quantization, full integer quantization and float16 quantization. We opted for full integer quantization of weights and activations since this form of quantization resulted in the smallest model and highest speedup. Floating point tensors are approximated using the following formula:

$$\mathsf{real\_value} = (\mathsf{int8\_value} - \mathsf{zero\_point}) * \mathsf{scale}$$

In TensorFlow's quantization specification, weights are represented by symmetric quantization and the $\mathsf{zero\_point}$ for per-tensor and per-axis weights is equal to 0. Activations are represented by asymmetric quantization and the $\mathsf{zero\_point}$ for per-tensor activations are in range [-128, 127].

In order to quantize the TensorFlow Lite model, we needed a representative dataset to create quantized values with an accurate range of activations. We found that 100 images from ImageNet, which was the number of images suggested by TensorFlow, to be sufficient for conversion. Then the model can be converted to a quantized model with a floating input or a quantized input file with a integer input. The quantization converted weights and activations to int8, making the quantized model about 4 times smaller than the original trained model.

Inference

In order to ensure that quantization had properly worked, we determined the accuracy of classifying a small subset of the validation ImageNet images with the quantized model. We used the Python API from TensorFlow lite to run inference. Images were resized to 224x224 and any pre-processing (cropping, resizing, etc.) done on training images was done on the images in this set before inputting image through the quantized model. The accuracy attained was about 70.438% indicating that the quantization had indeed worked. Quantization caused the model accuracy to decrease

slightly - by 0.074%.

4.1.5 Results

The steps we took to train and quantize ResNet-18 were nearly identical for each of the image recognition models we trained and quantized. In addition to ResNet-18, we trained smaller networks with MNIST dataset and CIFAR-10 dataset and determined the accuracy of the quantized networks on validation images. We also quantized a large pretrained network VGG-16 and tested on 5000 images from ImageNet validation set. We summarize the results that we have obtained in table 4.2.

Model	Dataset (# of classes)	Accuracy (Unquantized)	Accuracy (Quantized)	Change in Accuracy
ResNet-18	ImageNet (1000)	70.51%	70.44%	-0.07%
Custom CNN	MNIST (10)	95.19%	95.17%	-0.02%
Custom CNN	Cifar-10 (10)	79.32%	78.94%	-0.38%
VGG-16	ImageNet (1000)	62.04%	62.08%	0.04%

Table 4.2: Comparison of accuracy of unquantized and quantized models

Post-training quantization resulted in less than 0.5% drop in accuracy for smaller and real world networks, and in some cases resulted in an increase in accuracy.

Chapter 5

Plaintext Inference of Quantized Model

An image recognition model that has been trained and quantized must undergo further modifications prior to inference in order to be completely compatible with LHE and obtain a prediction for an encrypted image. Though we used Python API to run inference, the TensorFlow Lite interpreter itself is implemented in C++ and uses gemmlowp, a library for low precision matrix multiplication.

In order to make each layer in the model more accessible and easier to modify, we wanted to implement the interpreter code in Python using NumPy (a library for computing on large, multi-dimensional arrays and matrices). With an implementation in Python, we can quickly test different low-degree polynomial approximations for non-linear functions to make the model compatible with homomorphic encryption and measure the impact these modifications have on the accuracy of the model.

We can also use our interpreter implemented in Python to understand how quantized inputs and outputs are processed through each operator. Complexity in the TensorFlow Lite code arises from the need to handle edge cases with C integer types and from the fact that the implementation target is the ARM Neon instruction set. As a result, the quantized inference code must work around the peculiarities of that instruction set. Since these issues are not relevant to secure inference, we want to simplify the underlying implementation and make the code simpler before we tested

it with homomorphic encryption.

We first used Netron to view the quantized model and its layers. A portion of the quantized model is shown in Figure 5-1. The following operators are in the ResNet-18 model and needed to be implemented in Python - Add, Average Pool 2D, Conv 2D, Fully Connected, Max Pool 2D, Multiply, Softmax, Pad, Subtract, Arguments of the Maxima, Quantize. Some of these layers such as Add and Conv 2D are merged with ReLU layer to form Add+ReLU and Conv 2D+ReLU. The type (int8, int32, float, etc.) and sizes of each input and output of each operator are in the Netron file as seen in Figure 5-2. Some operators have additional information pertaining to that operator such as the type of padding used, the stride width, stride height, etc. If applicable, the inputs and outputs also have quantization related information, from which we can obtain the `zero_point` and the `scale`.

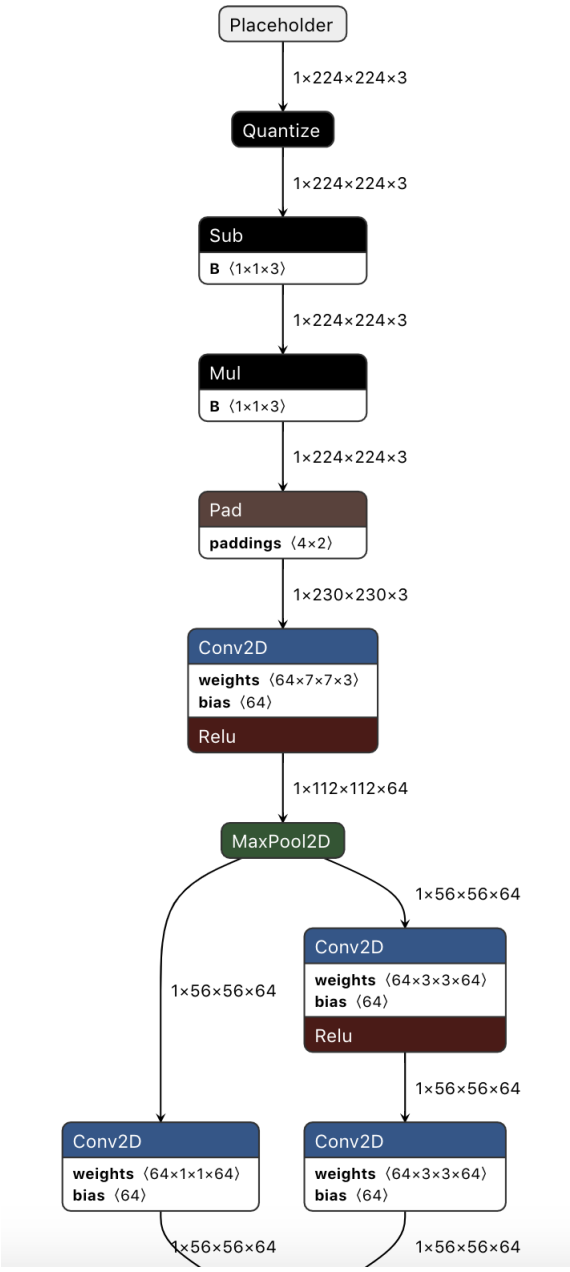


Figure 5-1: Portion of the quantized model shown in Netron.

NODE PROPERTIES

type	Conv2D
location	4

ATTRIBUTES

dilation_h_factor	1
dilation_w_factor	1
fused_activation...	RELU
padding	VALID
stride_h	2
stride_w	2

INPUTS

input	name: **Pad**
	type: **int8[1,230,230,3]**
	quantization: **-2.114842653274536 ≤ 0.018629100173711777**
	location: **4**
weights	name: **conv2d/kernel**
bias	name: **conv2d/Conv2D_bias**

OUTPUTS

output	name: **Relu**
	type: **int8[1,112,112,64]**
	quantization: **0 ≤ 0.1656312197446823 * (q - -128) ≤ 42.2359**
	location: **19**

Figure 5-2: Properties of Conv2D operator shown in Netron.

5.1 Code Validation

In order to validate each operator implemented in our version of the interpreter implemented in Python, we had to ensure that given an input, the output obtained from the Python code matches the output obtained from C++ code in the original library. According to TensorFlow Lite documentation, only the final output is exposed in the Python API and not intermediate outputs for inference so there was no way to obtain the outputs of each operator in the model unless we modified the TensorFlow library.

In order to modify the C++ code and obtain these intermediate outputs, we built TensorFlow library from source. The files and folders that were relevant to the interpreter code are located in the `lite` directory of TensorFlow and are as follows:

1. `interpreter.cc` and `interpreter.h` which had C++ code for interpreter
2. `python` directory which contains the Python API
3. `kernel` directory which contains the code for each of the operators

Additional code pertaining to the operators is found in `fixedpoint.h` in the `fixedpoint` directory of the gemmlowp code.

Initially, we experimented with printing the output from C++ code, but this would involve printing the output for each operator. Our idea was not scalable if we wanted to implement other operators in Python in the future. A more long term solution would be to add functionality to the Python API such that anytime we run inference, we could call the output of any operator and save the matrix in NumPy format.

Since the documentation stated that only the tensor of output nodes can be called from the Python API, we decided to label each intermediate node as an output node in C++ code. In the C++ code, we created a function `MarkAllAsOutputs`, in which the list of outputs are first cleared so as not to have the final output appear at the beginning of the list. Then, all the nodes are added to the output list in order that they appear in the model. We then created a new function in Python API `mark_all_as_outputs` that would mark all the intermediate nodes as output. As a

result, when we infer on an image, we can determine the output tensor of any kernel by indexing into the list of outputs.

5.2 Replicating Operators in Python

Within the `kernel` directory, there are several files pertaining to the kernels as well as an `internal` directory which contains two sub-directories, `optimized` and `reference`. Code within the operator files in `kernel` directory calls optimized code if the device the code is run on, uses ARM Neon instruction set. We seek to replicate the reference code for our interpreter in Python as it is straightforward.

Each operator has its own file in the `kernel` directory and is comprised of the following steps:

1. Init - Initializes the inputs and the outputs
2. Free - Frees memory
3. Prepare - Encompasses variety of important functions such as determining if broadcasting is necessary, obtaining multipliers and shifts for inputs and outputs, calculating activation range, etc.
4. Eval - Determines which function to use based on shape of inputs and outputs and whether device runs ARM Neon instruction set; calls a function in the `internal` directory

We include in detail the steps taken for two of the operators - quantize and add operator.

5.2.1 Quantize Operator

The very first step is quantizing the image input. z_{out} represents the zero point and s_{out} represents the scale of the output tensor. An unclamped output is obtained by

$$\mathsf{z}_{out} + \left\lfloor \frac{\mathsf{input} - \mathsf{tensor}}{\mathsf{s}_{out}} \right\rceil$$

Finally, we clip the values in the unclamped output tensor to values in the int8 range (from -128 to 127).

5.2.2 Add Operator

The ResNet-18 model contains several Add+Relu operators in the model. We replicate the steps taken within the Add operator. There are two inputs to each add operator and one output.

In the quantized Add operator, inputs cannot be directly added to obtain the outputs as the input tensors may have different scales. We must take a series of steps in order to normalize the tensors to the same scale and sum the inputs. We first obtain an integer multiplier and shift value for each input based on both inputs' scale. Each input is then modified, if necessary, to have the shape shape. The input's zero point is summed to the input and the input is then left shifted. This left shifted value is multiplied with the integer multiplier. Only the highest 31 bits are retained and then shifted according to the shift value, yielding scaled inputs that can be summed to obtain a raw output. The raw output is then clipped to obtain the final output.

We summarize the steps taken in the Add Operator in Figure 5-3 and explain each step in more detail below.

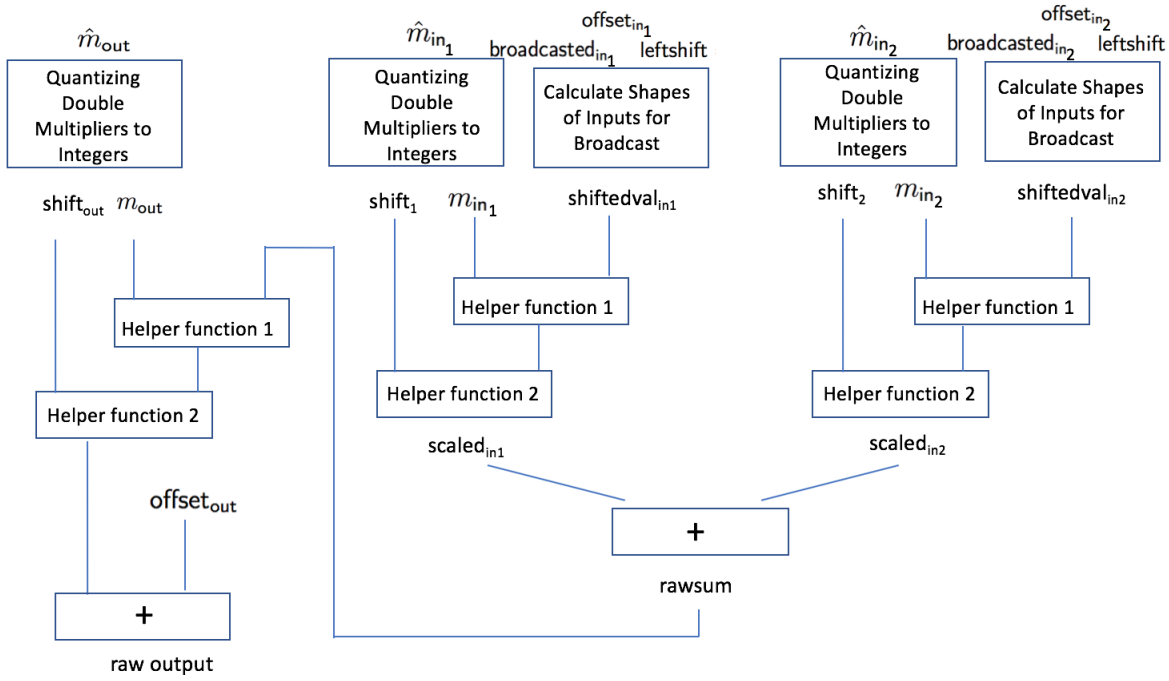


Figure 5-3: The steps taken in the Add Operator. To obtain the final output, we clip the raw output to be between and including the activation minimum and maximum values that we establish.

We use the following notation: Floating point values are represented with a hat ($\hat{x}$).

$$
\begin{aligned}
\mathsf{offset}_{\mathsf{in}_i} &= -z_{\mathsf{in}_i} \\
\mathsf{offset}_{\mathsf{out}} &= z_{\mathsf{out}} \\
\mathsf{leftshift} &= 20 \\
S &= 2 \cdot \max(s_{\mathsf{in}_1}, s_{\mathsf{in}_2}) \\
\hat{m}_{\mathsf{in}_i} &= \frac{s_{\mathsf{in}_i}}{S} \\
\hat{m}_{\mathsf{out}} &= \frac{S}{(2^{\mathsf{leftshift}} * s_{\mathsf{out}})}
\end{aligned}
$$

This ensures that $0 \leq \hat{m}_{\mathsf{in}_i} \leq 1$.

Quantizing Double Multipliers to Integers

The following formulae show how a double scale value ($\hat{q}$) gets converted to a pair of integers (q, shift) such that $\hat{q} \approx q \cdot 2^{\mathsf{shift}}$. This step is taken for both $\hat{m}_{\mathsf{in}_i}$ to obtain m_i and $\mathsf{shift_i}$ and for $\hat{m}_{\mathsf{out}}$ to obtain m_{out} and $\mathsf{shift}_{\mathsf{out}}$.

$$
\begin{aligned}
(q, \mathsf{shift}) &= \mathsf{frexp}(\hat{q}) \\
q &= \lfloor q \cdot 2^{31} \rceil \\
(q, \mathsf{shift}) &= (0, 0), \text{ if } \hat{q} = 0 \\
(q, \mathsf{shift}) &= (q/2, \mathsf{shift} + 1), \text{ if } q = 2^{31} \\
(q, \mathsf{shift}) &= (0, 0), \text{ if } \mathsf{shift} < -31
\end{aligned}
$$

Calculate Shapes of Inputs for Broadcast

In order to calculate the shape of the output, we must first determine if shapes of outputs are the same. If they are same, the ouput takes the same shape. If they

are not, we prepend 1 as the dimension to the input with less dimension so that the shapes are now the same. The output takes this shape. The following step then takes place for both inputs.

$$\mathsf{shiftedval}_{\mathsf{in_i}} = (\mathsf{broadcasted}_{\mathsf{in}_i} + \mathsf{offset}_{\mathsf{in}_i}) * 2^{\mathsf{leftshift}}$$

Quantization Helper Functions

For each input, the first function takes in the $\mathsf{shiftedval}_{\mathsf{in_i}}$ and m_{in_i}.

Helper function 1

We represented the inputs of this function as a and b,

$$\begin{aligned}
\mathsf{overflow} &= (a = b) \wedge (a = -2^{31}) \\
ab &= a * b \\
\mathsf{nudge} &= \begin{cases} 2^{30} & \text{if } ab \geq 0 \\ 1 - 2^{30} & \text{otherwise} \end{cases} \\
\mathsf{ab_x2_high32} &= \left\lfloor \frac{ab + \mathsf{nudge}}{2^{31}} \right\rfloor \\
c &= \begin{cases} 2^{31} - 1 & \text{if } \mathsf{overflow} \\ \mathsf{ab_x2_high32} & \text{otherwise} \end{cases}
\end{aligned}$$

In summary, given (a, b), Helper function 1 computes $\frac{2ab}{2^{32}}$

Helper function 2

Helper function 2 takes in the output of Helper function 1 and $-\mathsf{shift}_{\mathsf{i}}$.

With inputs of Helper function 2 represented as `numerator` and `exponent`,

$$\mathsf{mask} = 2^{\mathsf{exponent}} - 1$$

$$\mathsf{remainder} = \mathsf{numerator} \wedge \mathsf{mask}$$

$$\mathsf{threshold} = \begin{cases} \lfloor \frac{\mathsf{mask}}{2} \rfloor + 1 & \text{if numerator} < 0 \\ \lfloor \frac{\mathsf{mask}}{2} \rfloor & \text{otherwise} \end{cases}$$

$$\mathsf{scaled} = \begin{cases} \lfloor \mathsf{numerator} * 2^{-\mathsf{exponent}} \rfloor + 1 & \text{if numerator} < 0 \\ \lfloor \mathsf{numerator} * 2^{-\mathsf{exponent}} \rfloor & \text{otherwise} \end{cases}$$

For each input, we obtain $\mathsf{scaled}_{\mathsf{in_i}}$. We sum the $\mathsf{scaled}_{\mathsf{in_i}}$ to obtain a rawsum. Then we repeat the quantization steps with the Helper function 1's input being the rawsum and m_{out} and the Helper function 2's input being the output of Helper function 1 and $\mathsf{shift}_{\mathsf{out}}$. To obtain the raw output, we sum the output of Helper function 2 with $\mathsf{offset}_{\mathsf{out}}$.

Calculate the Activation Minimum and Maximum values

To obtain the final output, we clip the raw output to values between and including the minimum and maximum values calculated from this function. The lower and upper limits are decided based on what activation function is fused to the addition operator (if there is one).

$$
\begin{aligned}
(q_{min}, q_{max}) &= (-128, 127) \\
(s, z) &= (s_{\mathsf{out}}, z_{\mathsf{out}}) \\
q(x) &= z + \lfloor \frac{x}{s} \rceil \\
(\mathsf{act}_{min}, \mathsf{act}_{max}) &= \begin{cases} (\max(q(0), q_{min}),\ q_{max}) & \mathsf{ReLU} \\ (\max(q(0), q_{min}),\ \min(q(6), q_{max})) & \mathsf{ReLU6} \\ (\max(q(-1), q_{min}),\ \min(q(1), q_{max})) & \mathsf{ReLU1} \\ (q_{min},\ q_{max}) & \mathsf{No\ Activation} \end{cases}
\end{aligned}
$$

5.2.3 Other Operators

Subtract and Multiply operators are very similar to the Add operator. For the Subtract operator, $\hat{m}_{\mathsf{in}_2}$ is multiplied by -1. For the Multiplier operator, instead of summing the $\mathsf{scaled}_{\mathsf{in}_i}$, we multiply them.

Other operators such as Conv 2D and Fully Connected use many of the similar functions found in the Add operator such as quantizing double multipliers to integers, but are much more complicated.

Max Pool 2D, Average Pool 2D are very similar to their non quantized counterparts with the output being clipped to the values after calculating the activation minimum and maximum values. Essentially, the scale and zero point of the final output only is relevant if ReLU comes after the pooling layer. Otherwise, the output is clipped to values between and including the int8 minimum and maximum values. Softmax is also similar to its non quantized counterpart. The only difference is that the output is summed with the zero point of the final output and then clipped to values between and including the int8 minimum and maximum values.

Simpler operators include the ArgMax operator, which does not involve quantization, and the Padding operator, which involves padding with the $\mathsf{z}_{\mathsf{out}}$.

In summary, coding the operators in Python gave us an opportunity to understand the quantization functions and easily modify the functions as necessary to simplify the model and make it compatible with LHE with BFV scheme.

Chapter 6

Conclusions and Future Work

In this thesis, we explored the modifications that are necessary for a third party model to take an encrypted medical image as an input. We presented a training and quantization pipeline, which involved converting image data to TFrecord format, training real world models such as ResNet-18 from scratch on TensorFlow using TPU, converting the model to a TensorFlow Lite model and then quantizing it to make the model compatible with the BFV scheme. With appropriate serving model function, we showed that we are able to run inference and quantize model with less than 0.5% loss in accuracy.

We then built TensorFlow from source and coded functions in C++ and Python to facilitate extracting inputs and outputs of each layer in our model. We also wrote inference functions in Python for each of the operators in ResNet-18 and validated our implementation with the inputs and output tensors that we had extracted.

6.1 Future Work

In the future, we would like to modify the inference functions and approximate non-linear activation functions such as ReLU with low-degree polynomial functions to make the model compatible with LHE. Edge cases in homomorphic encryption are rare so we would like to remove them and test the effect on accuracy.

Once we are satisfied with the modified inference functions, we would like to

test it with BFV and LHE and determine accuracy on encrypted images. After any necessary modifications, we would like to implement this in Gazelle 2.0 code-base. We would like to measure the end-to-end latency and bandwidth and compare with other state-of-the-art models.

Finally, we would like to test the Gazelle 2.0 system with OncoNet network from Professor Regina Barzilay's group, which has been trained on plaintext breast scan data from Massachusetts General Hospital. We would like to evaluate the accuracy of inferring on encrypted medical data.

In conclusion, we were able to show a promising method for quantizing neural networks and customizing inference functions that can potentially be used with BFV and leveled homomorphic encryption for hospitals to send encrypted medical data to third party for image recognition.

Appendix A

Garbled Circuits and Secure Yelp

This thesis goes over quantizing a CNN in order to make it compatible with homomorphic encryption. However, as mentioned previously, garbled circuits are more efficient for non-linear functions than homomorphic encryption. This appendix will talk through the use of garbled circuits for the following problem:

If a user were to input their location and if a third party were to have a database of restaurants, the user could find out what is the closest restaurant to it without divulging his/her own location to the third party or having access to the entire database of restaurants.

A.1 Formulation

If the location of person is defined as (x_p, y_p) and R is set of restaurants in our database, we would like to satisfy the following expression,

$$\min_{i \in R} \sqrt{(y_i - y_p)^2 + (x_i - x_p)^2}$$

and output the location of the closest restaurant to the user.

Since we do not need to determine the exact distance between the person and the closest restaurant, but rather just the location of the closest restaurant, we can modify

the function by eliminating the square root. We then approximate our function by eliminating the squaring computations and modify the expression to:

$$\min_{i \in R} |y_i - y_p| + |x_i - x_p|$$

Thus, a function that returns the closest restaurant to a person would look something like the following if a person's location is inputted and a list of restaurants' locations is inputted:

Algorithm 1 Secure Yelp

```
procedure FindClosestRestaurant(person, restaurants):
    mindist ← ∞
    loc ← [0, 0]
    for i in restaurants do
        dist ← |i.x - person.x| + |i.y - person.y|
        if dist < mindist then
            mindist ← dist
            loc ← [i.x, i.y]
    return loc
```

A.2 Circuit Equivalent

In order to find the closest restaurants securely, we used boolean circuits and garbled them. We took the user's location as a 32 bit integer, where the leftmost 16 bits represented the x value of the user's location and the rightmost 16 bits represented the y value of the user's location. Each restaurant in the database was also similarly 32 bits, with leftmost 16 bits representing x value of the restaurant's and rightmost 16 bit representing the y value of the restaurant's location.

We concatenated the restaurant locations so the third party's input is a string of length $32n$ where n is the number of restaurants in the database. Thus the inputs to our boolean circuit were the user's location and the location of all the restaurants. We also had two additional inputs, 1 and 0 to represent power and ground. Together,

we had $32(n+1)+2$ inputs.

Because of the garbled circuits protocol, we were only limited to 2-input logic gates. We used the following gates in our circuit: AND gates, XOR gates, OR gates and XNOR gates. NOT gates were represented in the circuits by the input XOR 1. Due to Free-XOR optimization in garbled circuits protocol, which states that amount of computation does not rely on number of XOR gates, we also tried to maximize the number of XOR gates and minimize the number of other gates.

The minimum distance is first set to 17 bits of 1 and the closest location is first set to 32 bits of 0. Then we start calculating the distance between each restaurant and person. To calculate the distance between the x value of a restaurant and x value of the person, we first had to calculate which value was bigger so that the smaller number was subtracted from the larger number. We used a multi-bit comparator to compare the two values. Once we determine if the x value of the restaurant was equal to, less-than or greater-than, we used two multi-bit multiplexers, which had two inputs - the x value of the person and the restaurant. The selector bit in the first multiplexer was set to the less-than value obtained from the comparator. The selector bit in the second multiplexer was set to the greater-than value obtained from the comparator. Then we used a multi-bit subtractor to subtract the output of the second multiplexer from the output of the first multiplexer. The process is repeated to calculate the distance between the y value of the restaurant and y value of the person.

The distance between the x values and y values of the person and the restaurant are summed using a multi-bit adder. We then use a multi-bit comparator to compare the current minimum distance and the distance calculated between the restaurant and the person. A multi-bit multiplexer is used to select either the minimum distance or the distance between the restaurant and the person based on the value of the comparator. The output of the the multiplexer is compared with the distance between the person and the restaurant using a multi-bit comparator, thus determining if the minimum distance has changed. The equal to value of the comparator is the selector bit to a multi-bit comparator with two inputs: closest location and the restaurant location.

The restaurant location is thus selected if the distance between the restaurant and the person is now the minimum distance. The process is thus repeated with all the restaurants in the database. The circuit we designed is seen in Figure A-1.

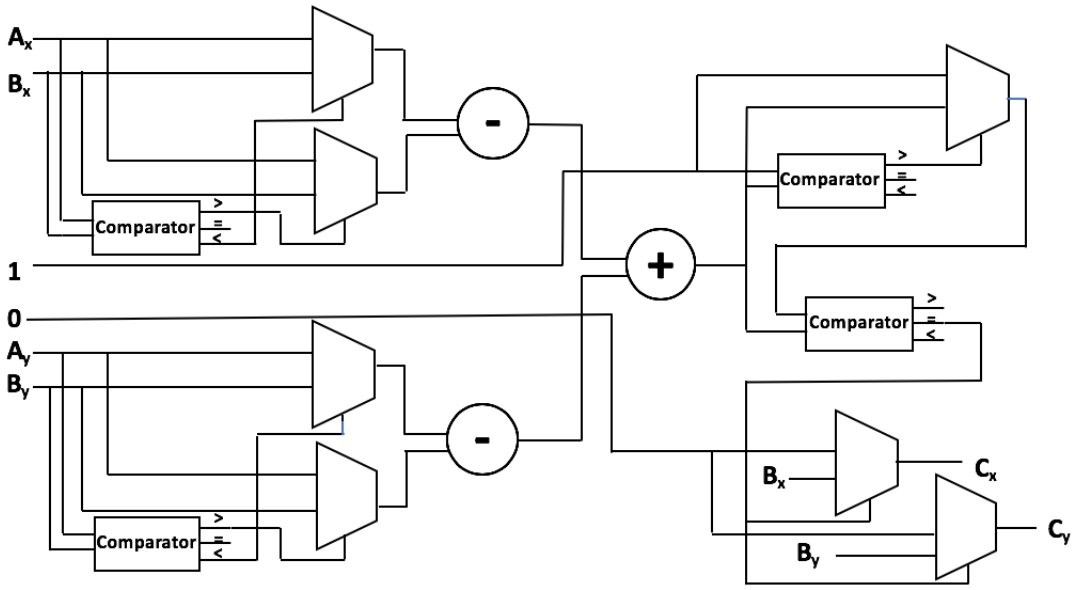
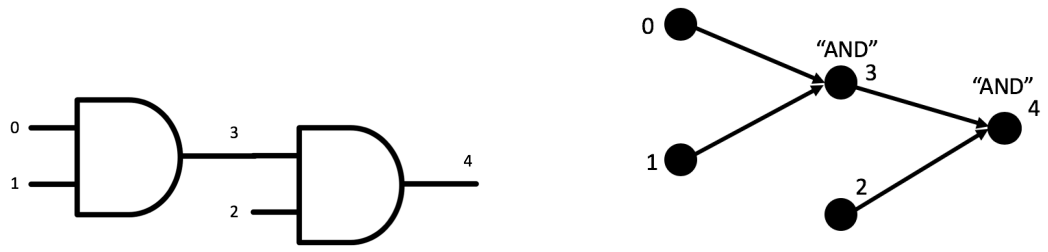


Figure A-1: Our boolean circuit to find closest restaurants securely. A_x and A_y represent the user's location, B_x and B_y represent the restaurant's location and C_x and C_y represent the output, i.e. the nearest restaurant to the the user's location.

It is important to note that each of the circuit components are composed of further components. Multi-bit comparators are composed of four-bit comparators. Multi-bit subtractor are composed of full subtractors which are made of half subtractors. Multi-bit adders are composed of full adders which are made of half adders. Multi-bit multiplexers are composed of single-bit multiplexers. All comparators, multiplexers, subtractors and adders are composed of the following logic gates - AND gates, XOR gates, OR gates and XNOR gates.

A.3 Circuit as a DAG

We had existing code to garble the circuit. There was a specific format needed to input the circuit to garble it. The circuit needed to bre represented by a list of tuples in the format (Number of input 1 node, Number of input 2 node, Number of output



(a) Circuit (b) Graph

Figure A-2: A circuit with two AND gates shown in(a). The associated directed acyclic graph with the circuit is shown in (b)

node, Name of Gate). The final 32 outputs of the circuit representing the closest restaurant to a person should be in the last 32 tuples of the list. To transform the circuit in this format, we modeled our circuit as a directed acyclical graph (DAG). The original inputs as well as the inputs and the outputs of each logic gate represented nodes in our graph. Each node (other than the original input nodes) is annotated with the logic gate that it is the output of. Edges were added between inputs and the outputs to represent the gate. Figure A-2 shows an example of a circuit and its associated DAG.

The graph is topologically sorted and nodes are renumbered and reordered such that the final output nodes are in the final 32 tuples in the list.

A.4 Results

Since garbled circuits are not reusable, the circuit grows linearly in size with the addition of each restaurant in the database. For n restaurants in the database, the `# of AND gates + # of XOR gates + # of OR gates + # of XNOR gates` is

$$714n + 417n + 273n + 72n$$

for a total number of $1476n$ gates. Table A.1 displays the number of gates needed if there are 100 restaurants in the database.

Table A.1: 100 restaurant in the database

# of AND gates	71400
# of XOR gates	41700
# of OR gates	27300
# of XNOR gates	7200

Bibliography

[1] Estimators : TensorFlow Core, https://www.tensorflow.org/guide/estimator [Accessed 17 May 2020].

[2] Homomorphic Encryption Standard 2018, https://projects.csail.mit.edu/HEWorkshop [Accessed 17 May 2020].

[3] Quantization aware training : TensorFlow Model Optimization, https://www.tensorflow.org/model_optimization/guide/quantization/training [Accessed 17 May 2020].

[4] Quantization, https://pytorch.org/docs/stable/quantization.html [Accessed 17 May 2020].

[5] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 483–512, Cham, 2018. Springer International Publishing.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. Technical Report MSR-TR-2016-3, February 2016.

[9] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. 01 2012.

[10] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[11] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, March 2010.

[12] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. Cryptology ePrint Archive, Report 2012/733, 2012. https://eprint.iacr.org/2012/733.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[14] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data, 2017.

[15] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[16] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference, 2018.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[19] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

[20] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, May 2017.

[21] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning, 2017.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[23] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[25] Harry Tanuwidjaja, Rakyong Choi, and Kwangjo Kim. *A Survey on Deep Learning Techniques for Privacy-Preserving*, pages 29–46. 09 2019.

[26] Adam Yala, Tal Schuster, Randy Miles, Regina Barzilay, and Constance Lehman. A deep learning model to triage screening mammograms: a simulation study. *Radiology*, 293(1):38–46, 2019.

[27] A. C.-C. Yao. Protocols for secure computations., In FOCS, volume 82, pages 160–164, 1982.