Secure Evaluation of Quantized Neural Networks

Anders Dalskov*, Daniel Escudero*, Marcel Keller†
*Aarhus University, {anderspkd,escudero}@cs.au.dk †CSIRO's Data61, mks.keller@gmail.com

***Abstract*—Image classification using Deep Neural Networks that preserve the privacy of both the input image and the model being used, has received considerable attention in the last couple of years. Recent work in this area have shown that it is possible to perform image classification with realistically sized networks using e.g., Garbled Circuits as in XONN (USENIX '19) or MPC (CrypTFlow, Eprint '19). These, and other prior work, require models to be either trained in a specific way or postprocessed in order to be evaluated securely.**

We contribute to this line of research by showing that this postprocessing can be handled by *standard* Machine Learning frameworks. More precisely, we show that *quantization* as present in Tensorflow suffices to obtain models that can be evaluated directly and *as-is* in standard off-the-shelve MPC. We implement secure inference of these quantized models in MP-SPDZ, and the generality of our technique means we can demonstrate benchmarks for a wide variety of threat models, something that has not been done before. In particular, we provide a comprehensive comparison between running secure inference of large ImageNet models with active and passive security, as well as honest and dishonest majority. The most efficient inference can be performed using a passive honest majority protocol which takes between 0.9 and 25.8 seconds, depending on the size of the model; for active security and an honest majority, inference is possible between 9.5 and 147.8 seconds.

***Summary of Changes**: This version of the preprint adds a full implementation of inference in MPC. Assi Barak has been removed from the authors at his request.*

I. Introduction

Machine Learning (ML) models are becoming more relevant in our day-to-day lives due to their ability to perform predictions on several types of data. Neural Networks (NNs), and in particular Convolutional Neural Networks (CNNs), have emerged as a promising solution for many real-life problems such as facial recognition [45], image and video analysis for self-driving cars [6] and even for playing games (most readers probably know of *AlphaGo* [58] which in 2016 beat one of the top Go players). CNNs have also found applications within areas of medicine. [22], for example, demonstrates that CNNs are as effective as experts at detecting skin cancers from images, and [20] investigated using CNNs to examine chest x-rays.

Many applications that use Machine Learning to infer something about a piece of data, does so on data of sensitive nature, such as in the two examples cited above. In such cases the ideal would be to allow the input data to remain private. Conversely, and since model training is by far the most expensive part of deploying a model in practice,[1] preserving model privacy may be desirable as well.

[1]For example, the network by Yang et al. [67] costs between $61 000 and $250 000 to train according to https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-models/.

In order to break this apparent contradiction (performing computation on data that is ought to be kept secret) tools like *secure multiparty computation* (MPC) can be used. Using such tools, image classification can be performed so that it discloses neither the image to the model owner, nor the model to the input owner. In the client-server model this is achieved by letting the data owner and the model owner *secret-share* their input towards a set of servers, who then run the computation over these shares. It is nevertheless worth noting that an adversary with black-box access to a model, such as in the Machine Learning As A Service scenario, who can request predictions on arbitrary inputs can steal the model with near-perfect fidelity in some cases [62, 64]. We consider this an orthogonal work to secure inference using MPC techniques.

Research in the area of secure evaluation of CNNs has been rich during the last couple of years [24, 55, 56, 47, 50, 38, 63, 54, 44]. The main goal of this prior research has been to reduce the performance gap between evaluating a CNN in the clear and doing it securely. Current state of the art solutions rely on for example Garbled Circuits, such as the XONN [54] or MPC [44]. Both of these works manage to evaluate large ImageNet type models (tens of layers and 1000 classes) with reasonable efficiency. Moreover, the CrypTFlow framework [44] also support secure inference with malicious security albeit by relying on a secure hardware assumption.

Evaluating these models securely still takes in the order of seconds, which is definitely too slow for applications that require real-time image classification. However, for applications like [20, 22] this delay is definitely acceptable and in any event faster than having a doctor manually examine the image. (In practice, a human would probably still examine the image. However ML provides an appealing aid for the doctor.)

A. Towards practical secure inference.

The ultimate goal of all previous work on secure inference is to obtain a system—or at least move towards a system—that can be used for practical tasks.

While inference in the clear can work with floating point numbers and still be efficient, the same cannot be said for Secret Sharing based MPC which are better suited for modular integer-based computation. One can map a set of floating-point numbers with finite precision to integers and use a modulus that is large enough so that overflows do not occur (effectively emulating integer arithmetic), as done by Gilad-Bachrach et al. [24]. However, this approach is prohibitive when many multiplications are to be performed. On the other hand, one can also use fixed-point arithmetic, i.e. a real number $x \in \mathbb{R}$ is approximated by $x \approx 2^{-\ell}\hat{x}$ where $\hat{x} \in \mathbb{Z}_M$ for some ℓ, M, where ℓ, the precision, is the same for all values. Under this

representation, the arithmetic is performed over the integer representation $\hat{x}$. In order to perform multiplications in this representation, truncation protocols are needed. Different to plaintext computation, this truncation is more expensive than the multiplication.

Another source of complexity arises from the fact that non-linear operations that are trivial in the clear, such as comparisons, become expensive or even impossible when computed securely. The consequence of this is that common activation functions such as Rectified Linear activation or sign functions suddenly impose a significant overhead when compute securely. Depending on the system, previous solutions use e.g., garbled circuits to compute these functions [50, 47, 38] or rely on ad-hoc approximations (e.g., $x \mapsto x^2$ instead of $x \mapsto \max(x, 0)$) [24] that affects both training and the accuracy of the model.

1) Model sizes: The paragraphs above argue that there are some limitations that restricts the size and kind of models that it is possible to evaluate securely. These limitations are reflected in the type and size of models that are used to illustrate practicality in previous work on secure inference. In particular, previous work only evaluate small networks on small datasets, such as Cifar10 or MNIST. This choice is understandable—secure evaluation impose a significant overhead and more realistic models typically require 100s of millions of multiply-add operations—but it leaves open questions as to the efficiency for larger models. More importantly—in particular when one considers that the secure evaluation process requires modifying the networks as described above—it is not clear if the results with respect to accuracy for these smaller models, also carry over to larger models.

There are essentially two ways of addressing these issues: Either one needs to evaluate the larger models and so show that the modifications that are made do not affect the accuracy (or if they do, to what extent). This approach is taken in the recent and concurrent works XONN [54] and CrypTFlow [44]. Another approach is to evaluate models without modifying them. This approach allows one to argue that any guarantees about accuracy which are reported by the tool used for training the model will still be valid when evaluated securely.

2) Usability: A final obstacle for practical deployment has to do with usability of the secure evaluation solution, and is an obstacle that arise as a result of the challenges pointed out so far. In particular, and as a result of previous work using custom tailored protocols and a special purpose training phase, each solution carries with it a *full set of tools* that the end-user needs to become familiar with. For the purpose of having a usable system, this is clearly unreasonable: it is simply not realistic to expect an end-user to relearn a new framework every time a better secure solution is released. For this reason, some recent work [44, 2] strive to make their solution work with standard Tensorflow models.

A second issue with respect to usability, arises because different end-users will have different requirements in terms of security of the system. Because previous solutions develop specifically tailored protocols for secure evaluation, only one threat model is supported. To the best of our knowledge, it is only CrypTFlow [44] that also support active security. However CrypTFlow relies on SGX to obtain active security and so it remains an open problem of getting efficient secure inference without relying on a secure hardware assumption.

B. Our Contribution

Improvements to the viability of evaluating Machine Learning models in a secure way has been driven by advances in MPC; however, recent research in the area of Machine Learning itself may be beneficial towards this goal. At a high level, our main contribution is showing that existing techniques from the realm of Machine Learning plays very nicely with recent advances in MPC towards the goal of getting efficient inference. More precisely, the research area of *quantization* provides us with a *well tested* and *widely deployed* approach to performing *practically relevant* prediction tasks securely.

Quantization is motivated by the deployment of ML models in resource-constrained environments like mobile phones or embedded devices, and aims at reducing the size of neural networks by lowering the precision of the values involved. Besides lowering precision, another goal of quantization is to simplify the arithmetic used, so that only integer and bit-wise operations are used. These techniques, quite coincidentally, are highly convenient when we consider a secure implementation using MPC.

In this work we demonstrate that models as trained and output by Tensorflow can be evaluated directly and *as-is* using more or less off-the-shelve MPC. This illustrates a particularly attractive and user-friendly approach to secure evaluation as (1) models can be trained by users that do not have to be familiar with the secure protocol, and (2) that users can select a secure protocol that more closely matches the threat model that they need. The first point has seen some treatment in previous work, such as MiniONN [47] or CrypTFlow [44]. The second point has been examined in previous work such as XONN [54], which permits standard techniques to obtain active security (although they do not implement it), or CrypTFlow which achieve active security by relying on SGX. However, supporting a wider array of threat models while allowing for arbitrary Tensorflow models have not been achieved yet.

Our contributions can be summarized as follows:

1) We demonstrate that currently deployed and actively researched methods of quantization from Machine Learning solve many of the problems currently faced when developing MPC based secure inference. We employ a specific quantization scheme developed by Jacob et al. [37]. The reason for choosing this particular scheme, is that it (1) solves issues faced when performing secure inference using MPC, and (2) that it is already present in Tensorflow. In relation to the latter point, using this quantization scheme means we can use standard tools for training models without having to develop specialized tools that later convert the trained model into something which can be run by the MPC (this is unlike concurrent work such as CryptTFlow [44] which, while it works with standard Tensorflow models, still require a specific piece of software which converts the model into something that can be run securely).

2) Because of the generality of our approach—in particular that it is black box with respect to the underlying MPC—we provide benchmarks and implementations using several different protocols. This, we argue, is important for practical deployment, as it allows the user to pick from a variety of different threat and system models in order to best fit their need. Consequently, we are also the first to provide secure inference benchmarks for large ImageNet models (in particular, models of the MobileNets architecture [32]) that enjoy active security without relying on hardware assumptions. (As pointed out earlier, CrypTFlow achieve malicious security as well, but rely on SGX to obtain malicious security.)
3) Lastly, we develop a simple truncation protocol, which might be of independent interest, that allows one to compute a right shift by a *secret* value at essentially the cost of a public right shift. The downside is that computation needs to take place over a potentially larger modulus.

C. Techniques

1) Quantization: As mentioned in the preceding section, the core of our work is the theory of quantization. In a nutshell, this allows a set of real numbers $\{\alpha_1, \ldots, \alpha_n\} \in \mathbb{R}$ to be represented by a set of integers $\{a_1, \ldots, a_n\} \in \mathbb{Z}_M$ in a way so that basic operations such as additions and multiplications are preserved, at least up to some extent. In the context of CNNs the motivation of using quantization is to reduce storage: Instead of storing a set of real numbers (say, 32-bit floating-point numbers), one can replace this set by small integers; in practice either 8 or 16-bits.

Research in developing quantization schemes for several ML models and understanding the effect of these techniques in the accuracy is extensive in the ML community, as can be seen from the recent survey by Guo [29]. We choose to focus our work on the quantization scheme by Jacob et al. [37], which works by mapping a real number $\alpha_i \in \mathbb{R}$ to $x_i \in \mathbb{Z}_M$ such that $\alpha_i \approx m \cdot (x_i - z)$, where $m \in \mathbb{R}$ and $z \in \mathbb{Z}_M$ are parameters depending only on the set being quantized and the bound M. This affine mapping, which can be seen as a shifted version of fixed-point arithmetic, turns out to preserve accuracy quite well even when using 8-bit integers [42].

By making use of this quantization scheme we can implement the core operations used in CNNs, like convolutions and fully connected layers, by using mostly integer-only arithmetic. Most of these operations rely on taking dot products, which can be done very easily with this quantization scheme by taking the corresponding integer dot product over the quantized values and then performing a truncation afterwards. Moreover, the effect of using this type of quantization on the accuracy of the model has been studied extensively already and has been found to be very small. For example, the models we consider in this work achieve up to 70.7% top-1 accuracy and 89.5% accuracy[2]

[2]For ImageNet models, top-1 refers to the accuracy with which the model correctly classifies an image, while top-5 means that the correct label is within the top 5 predictions.

In must be said that in technical terms this quantization technique is not much different from using fixed-point arithmetic directly (for instance, this quantization method requires truncation just like fixed-point does), which is what most previous works in privacy-preserving ML have done.

2) Secure Computation: We implement our inference protocol in MP-SPDZ [17], and we benchmark several different ImageNet networks. This allows us to illustrate the trade-offs between security and performance, in order to get a better understanding on the right protocol to choose for a given application.

We benchmark these networks using a total of different 8 protocols, and present the results in detail in Section III-B. The protocols are distinguished based on the following factors:

- Active vs. passive security
- Computation modulo a power of two vs. modulo a prime
- Dishonest majority vs. Honest majority

We use MP-SPDZ because it provides an efficient implementation of most of the necessary primitives as well as high-level programming interface for it, and we added missing routines like truncation by a secret shift, as shown in Section III-D5, and cheap sums-of-products, which forms the basis of the most efficient setting (honest majority over a ring). The MP-SPDZ compiler supports a high level Python-like language and so it is easy to extend it with support for more models in the future.

D. Related Work

Secure evaluation of Neural Networks can be traced back to at least the work by Orlandi et al. [5, 51] which present a solution based on HE techniques. Several later works rely on HE techniques either in full or in part. CryptoNets [24] use Leveled Homomorphic Encryption (LHE), which necessitates bounding the number of operations a priori. In addition, HE only permits evaluation of polynomials and as such cannot compute e.g., the Rectified Linear activation functions (the function $x \mapsto \max(0, x)$) and the authors therefor rely on the approximation $x \mapsto x^2$. However, and as pointed out by Gilad-Bachrach et al. [24], such an approximation makes training difficult for larger networks, the issue being that the derivative of x^2 is unbounded. Chabanne et al. [10] improve upon CryptoNets by evaluating networks with 6 hidden layers (as opposed to only 2 as in Gilad-Bachrach et al.). More recently, Bourse et al. [7] obtain faster evaluation albeit for a smaller network (one and two hidden layers) by combining FHE and Discretized Neural Networks (i.e., networks where weights are in $\{1, -1\}$).

One of the downsides of HE based solutions are their inefficiency and inability to handle common activation functions. Gazelle [39] combines garbled circuits (GC) with additive HE (AHE) in order to obtain a more efficient system. The boost in efficiency is attributed to an efficient method of switching between the AHE scheme and a GC, where the former is used to compute convolutions and fully connected layers, while the latter is used to compute the network's activation functions.

The idea of using multiple different protocols to achieve faster predictions have been used before [39]. MiniONN [47] develops a technique for turning a pretrained model into an

oblivious one, which can be evaluated using a mix of HE, additive secret sharing and GC. Chameleon [55], which is an extension of the ABY framework by Demmler et al. [19], likewise use secret sharing for matrix operations and GC for activation functions. More recently, ABY3 by Mohassel and Rindal [49], also benchmark secure evaluation (albeit the authors do not implement full inference) in a framework that relies on a mix of secret sharing, boolean (i.e., GMW) and garbled circuits.

Finally, like solutions relying purely on HE have been considered before, so has solutions that rely purely on GC or MPC; the latter of which is most relevant to this work. DeepSecure [56] is perhaps the first work to take a pure GC based approach for evaluating Neural Networks. More recently XONN [54] builds a very efficient GC based solution by noting that Binarized Neural Networks [34] (i.e., networks with weights that are bits) can be evaluated very efficiently. XONN shows that evaluating deep networks ($>$ 20 layers) is possible. A different approach is taken by Ball et al. [2] where the authors use the arithmetic garbling technique of Ball et al. [3] to evaluate Neural Networks. Pure MPC based solutions have been studied in SecureML [50], which employs a three-party honest majority protocol. A major performance boost in SecureML can be attributed to the way fixed point arithmetic is handled, where the authors show that it is possible to just have parties perform the truncation locally. SecureNN [63] can be seen as an extension of SecureML where both three- and four-party protocols (both with one corrupted party) are used. Concurrently to this work, CrypTFlow [44] builds a system on top of SecureNN that is capable of evaluating very large networks ($>$100 layers) in reasonable time. Another very attractive feature of CrypTFlow is that it provides a more complete framework that accepts standard Tensorflow trained models as input (hence the name).

1) Quantization in prior work: Whether implicitly or explicitly, most prior work already uses some form of quantization. For instance, replacing directly floating-point by fixed-point numbers can already be seen as quantization. More often than not, however, this conversion is done in a very naive manner where the primary goal has been to fit the model parameters to the secure framework without further consideration about any potential impact it might have on the model's accuracy. Relatively little work has made explicit use of quantization in the context of securely evaluating Machine Learning models. One example is the recent work by Bourse et al. [7], where the authors use a quantization technique that is similar to the one described by Courbariaux and Bengio [11]. Sanyal et al. [57] use the same techniques. Nevertheless, their work lies in the FHE domain, which differs from multiparty computation. For instance, the fact that the weights are kept in the clear by the model owner changes the way the computation is performed, and allows them to use only additions and subtractions. XONN [54], which as mentioned is based on Garbled Circuits, use a quantization scheme which converts weights into bits [34]. For this to work, the authors need to increase the number of neurons of the network and a large part of the above work is dedicated to describing how this scaling can be performed. CrypTFlow [44] employ what can be seen as a custom fixed-point-to-floating-point conversion protocol (called Athos) that automatically converts the floating point weights of a Tensorflow model into a fixed points representation, where the parameters are chosen so as to not compromise on the models original accuracy.

2) Frameworks for secure evaluation: Several previous works provide what can be viewed as a more complete framework for secure evaluation. The first of these is MiniONN [47] which provides techniques for converting existing models into models that can be evaluated securely. The authors demonstrate this framework by converting and running several models for interesting problem domains, such as Language modeling, as well as more standard problems such as hand writing recognition (MNIST) and image recognition (Cifar10). CrypTFlow [44] also provides more complete framework. As already mentioned above, the first step in their framework is a protocol for converting an Tensorflow trained model into a model that can later be evaluated securely using a protocol based on SecureNN [63].

E. Outline of the Document

In Section II we give a brief introduction to Neural Networks after which we describe the quantization scheme we will be using. In Section III we provide a self-contained description of our protocol for secure inference, describing the basic building blocks. We discuss implementation details and present benchmarks in Section IV, and conclude in Section V.

II. Deep Learning and Quantization

Deep learning models are at the core of many real-world tasks like computer vision, natural language processing and speech recognition. However, in spite of their high accuracy for many such tasks, their usage on embedded devices like mobile phones, which have tight resource constrains, becomes restricted by the large amount of storage required to store the model and the high amount of energy consumption when carrying the computations that are typically done over floating-point numbers. To this end, researchers in the machine learning community have developed techniques that allow weights to be represented by low-width integers instead of the usual 32-bit floating-point numbers, and quantization is recognized to be the most effective such technique when the storage/accuracy ratio is taken into account.

Quantization allows the representation of the weights and activations to be as low as 8 bits, or even 1 bit in some cases [11, 53].[3] This is a long-standing research area, with initial works already dating back to the 1990s [23, 4, 61, 48], and this extensive research body have enabled modern quantized neural networks to have essentially the same accuracy as their full-precision counterparts [12, 68, 26, 30, 52], even with large CNN architectures like AlexNet [43], VGGNet [59], GoogleNet [60] and ResNet [31].

[3]Furthermore, some quantization techniques also allow to represent gradients with a small number of bits, which effectively allows for quantized training of neural networks. However, this is still in a very early stage, and since we are focused only on inference in this work, we do not present such techniques.

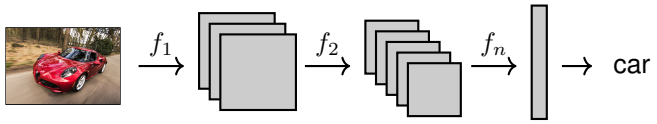

Fig. 1: Visualization of a Convolutional Neural Network.

A. Notation

For a value $\boldsymbol{x} \in \mathbb{R}^{N_1 \times N_2 \times N_3}$ we use $\boldsymbol{x}[i, j, c] \in \mathbb{R}$ to denote taking i'th value across the first dimension, the j'th value across the second dimension and the c'th value across the last dimension. In a similar way, we might write $\boldsymbol{x}[\cdot, \cdot, c] \in \mathbb{R}^{N_1 \times N_2}$ to denote the matrix obtained by fixing a specific value for the last dimension. A real value interval is denoted by $[a, b]$ and a discrete interval by $[a, b]_{\mathbb{Z}}$. We define *clamping* of a value $x \in \mathbb{R}$ to the interval $[a, b]$, denoted by $\mathsf{clamp}_{a,b}(x)$, by setting $x \leftarrow a$ if $x < a$, $x \leftarrow b$ if $x > b$ and otherwise $x \leftarrow x$. (Clamping to a discrete interval is similarly defined.) We denote by $\mathbb{N}_\ell$ the set $\{1, \ldots, \ell\}$.

B. Deep Learning

An artificial Neural Network, or simply Neural Network (NN) for short, is a machine learning model that is used to obtain predictions on some data that has proven to be very successful at specific tasks like character recognition, data processing and image classification. In a very general setting, a neural network is an ordered set of functions $(f_1, \ldots, f_n)$ where $f_i : D_{i-1} \to D_i$, with D_i some space of the form $\mathbb{R}^{N_1 \times \cdots \times N_{\ell_i}}$, for some $\ell_i > 0$. An element of such a set is known as a *tensor*, and each function f_i is known as a *layer*. The input to the neural network is a tensor $\boldsymbol{x} \in D_0$, and the output is $\boldsymbol{y} = f_n \circ \cdots \circ f_1(\boldsymbol{x}) \in D_n$. For convolutional neural networks (CNNs) in practice D_0 could be $\mathbb{R}^{128 \times 128 \times 3}$ to represent 128×128 images with 3 color channels (RGB), and the output set D_n could be a vector where the i-th entry represents the probability that the input image has a given label indexed by i. See Figure 1 for a visual representation of a neural network.

Some of the types of layers considered in practice include affine layers like fully connected and convolutional layers, non-linear activations like ReLU and ReLU6, and down-sampling layers like average or max pooling. Below we discuss some of the layers we will consider in this work.

1) Two-Dimensional Convolutional Layer: A two-dimensional convolution is an operation performed between two three-dimensional tensors to produce another three-dimensional tensor. The parameters for a two-dimensional convolution are the input dimensions I_h, I_w, I_d, the output dimensions O_h, O_w, O_d and the dimension of the windows W_h, W_w. On input $\boldsymbol{x} \in \mathbb{R}^{I_h \times I_w \times I_d}$, a weight tensor $\boldsymbol{w} \in \mathbb{R}^{O_d \times W_h \times W_w \times I_d}$ and a bias vector $\boldsymbol{b} \in \mathbb{R}^{O_d}$, the convolution performs the following operations.

1) For each $(i, j) \in \mathbb{N}_{O_h} \times \mathbb{N}_{O_w}$, a sub-tensor of $\boldsymbol{x}$, denoted by $\boldsymbol{x}_{i,j} \in \mathbb{R}^{W_h \times W_w \times I_d}$, is extracted. We will not dive into the details of how this is done besides saying that it depends on some extra parameters like the *stride* and the *padding type*, which are part of the architecture.[4]
2) The entry indexed by $(i, j, k) \in \mathbb{N}_{O_h} \times \mathbb{N}_{O_w} \times \mathbb{N}_{O_d}$ of the output is computed as $\mathsf{dot}(\boldsymbol{x}_{i,j}\boldsymbol{w}[k, \cdot, \cdot, \cdot]) + \boldsymbol{b}[k]$, where dot represents the sum of the point-wise products of the two inputs.

2) ReLU and ReLU6: It should be clear from the description of the convolutional layer above that this operation is affine, and any composition of such functions would result in an affine function as well. The purpose of non-linear activation functions like ReLU and ReLU6, as the name suggests, is to break this linear relation between input and output. ReLU (REctified Linear Unit) takes as input a tensor and applies the following to each one of its entries: if the entry is negative then replace it with 0, and leave it unchanged otherwise. ReLU6 works in a similar way, but if the entry is greater than 6 then it is replaced by 6. This can be seen as clamping the input to the interval $[0, 6]$.

3) Average and Max Pooling: A pooling operation takes as input a tensor $\boldsymbol{x} \in \mathbb{R}^{I_h \times I_w \times I_o}$ and returns a tensor $\boldsymbol{y} \in \mathbb{R}^{O_h \times O_w \times O_d}$ with $O_h < I_h$, $O_w < I_w$ and $O_d = I_d$, which can be seen as a *down-sampling* operation. Each entry $\boldsymbol{y}[i, j, k]$ is computed by applying a given operation to the entries of some sub-matrix $\boldsymbol{x}_{i,j}[\cdot, \cdot, k]$ of $\boldsymbol{x}[\cdot, \cdot, k]$, which is extracted in a similar way as in a convolution. For average pooling the operation is just the mean of the entries in the matrix, and for max pooling the maximum of the entries is returned.

4) Output: The output of a CNN is typically a vector. The index with the maximum value in this vector represents the most likely label corresponding to the input, and this vector can be normalized into a probability distribution via a monotonous function like Softmax so that the value corresponding to index i is the probability that the input has label i.

5) Batch Normalization: The networks we will be working with use Batch Normalization [36], which is a technique used to speed up training. The idea is to normalize the inputs to each activation: instead of computing $g(x)$ for input x and activation function g, we instead compute $g(y)$ where

$$y = \gamma \left(\frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \beta, \tag{1}$$

where γ, β are parameters learned during training, and μ_B, σ_B^2 is the mean and variance, respectively, of a batch B of which x is a member. Consider an input $y = xW + b$ to g. During inference, we can "fold" the batch normalization parameters into the weights, which is done by using W' and b' defined as

$$W' = \frac{\gamma W}{\sigma}, \qquad b' = \gamma \left(\frac{b - \mu}{\sigma} \right) + \beta.$$

It is straight forward to verify that using $y' = xW' + b'$ yields the expression in Eq. (1).

[4] See https://cs231n.github.io/convolutional-networks/#conv for an elementary and thorough explanation.

C. Google's Quantization Scheme

The goal of this section is to provide an overview of the quantization technique of Jacob et al. [37] (see also [42]) that we will be relying on to get efficient secure inference. While this particular quantization scheme might not be state of the art, or even the best for all choices of secure inference (e.g., XONN [54] rely on a different scheme to get efficient inference) we choose this particular scheme for the following reasons: It is implemented in Tensorflow (more precisely, TFLite [28]) and as such we get an user friendly, widely available and well documented way of training models that can securely evaluated. The fact that Tensorflow can be used to directly train models for our framework is very handy indeed as it removes the need to develop custom tooling that has little to do with the secure framework itself. Moreover, Tensorflow provide several pre-trained ImageNet models which provides a very good point of reference for not only our benchmarks, but for future work that wish to compare against us. Indeed, few if any previous work on secure inference provide pretrained models which makes an accuracy oriented comparison very hard.[5]

It bears mentioning that this section does not aim at providing a comparison or treatment of the different quantization schemes that exist. For a broader survey of the various of the different quantization techniques that exist we direct the reader to Guo [29].

We note that this scheme is benefitial for MPC since it simplifies the activations and the arithmetic needed to evaluate a CNN. However, the original goal of Jacob et al. was to reduce the size of the models, rather than simplifying the arithmetic or the activations. Unfortunately, we do not get the benefits in the size reduction since, even if the network can be stored using 8-bit integers, arithmetic must be done modulo 2^{32} and even 2^{64} in some cases.

1) Quantization and De-Quantization: The scheme comes in two variants, one for 8-bit integers and another one for 16-bit integers. In this work we focus in the former, and we provide our description only in that setting.

Let $m \in \mathbb{R}$ and $z \in [0, 2^8)_{\mathbb{Z}}$ and consider the function $\mathsf{dequant}_{m,z} : [0, 2^8)_{\mathbb{Z}} \to \mathbb{R}$ given by $\mathsf{dequant}_{m,z}(x) = m \cdot (x - z)$. This function transforms the interval $[0, 2^8)_{\mathbb{Z}}$ injectively into the interval $I = [-m \cdot z, m \cdot (2^8 - 1 - z))$ and as such it admits and inverse $\mathsf{quant}_{m,z}$ mapping elements in the image of $\mathsf{dequant}_{m.z}$ into $[0, 2^8)_{\mathbb{Z}}$. We define the quantization of a number $\alpha \in I$ to be $\mathsf{quant}_{m,z}(\alpha')$, where α' is closest number to α such that α' is in the image of $\mathsf{dequant}_{m,z}$.

The constants m, z above are the parameters of the quantization, and are known as the *scale* and the *zero-point*, respectively. This quantization method will be applied on a per-tensor basis, i.e. each individual tensor $\boldsymbol{\alpha}$ has a single pair m, z associated to it. These parameters are determined at training time by recording the ranges on which the entries of a given tensor lie, and computing m, z such that the interval $[-m \cdot z, m \cdot (2^8 - 1 - z))$ is large enough to hold these values.

[5]This is especially the case if it is not clear exactly how the model was trained and which training and test data was used.

See Figure 2 for a visualization of this quantization method, and see Jacob et al. [37] for details.

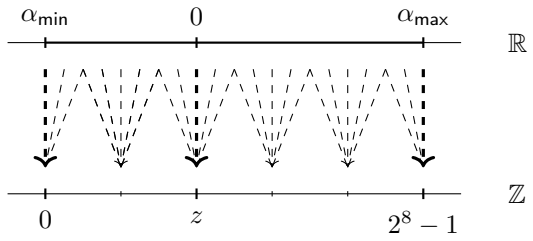

Fig. 2: Visualization of the Quantization Scheme by Jacob et al. [37]. The continuous interval on top is mapped to the discrete interval below, and multiple numbers may map to the same integer due to the rounding.

2) Dot Products: Computing dot products is a core arithmetic operation in any CNN. In this section we discuss how to do this with the quantization method described above.

Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_N)$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_N)$ be two vectors of numbers with quantization parameters (m_1, z_1) and (m_2, z_2), respectively. Let $\gamma = \sum_{i=1}^{N} \alpha_i \cdot \beta_i$, and suppose that γ is part of a tensor whose quantization parameters are (m_3, z_3). Let $c = \mathsf{quant}_{m_3,z_3}(\gamma)$, $a_i = \mathsf{quant}_{m_1,z_1}(\alpha_i)$ and $b_i = \mathsf{quant}_{m_2,z_2}(\beta_i)$. It turns out we can compute c from all the a_i, b_i by using integer-only arithmetic and fixed-point multiplication, as shown in the following.

Since $\gamma \approx m_3 \cdot (c - z_3)$, $\alpha_i \approx m_1(a_i - z_1)$ and $\beta_i \approx m_2(b_i - z_2)$, it holds that

$$m_3 \cdot (c - z_3) \approx \gamma = \sum_{i=1}^{N} \alpha_i \cdot \beta_i \approx \sum_{i=1}^{N} m_1 \cdot (a_i - z_1) \cdot m_2 \cdot (b_i - z_3).$$

Hence, we can approximate c as

$$c = z_3 + \frac{m_1 \cdot m_2}{m_3} \cdot \sum_{i=1}^{N} (a_i - z_1) \cdot (b_i - z_2) \tag{2}$$

The summation $s = \sum_{i=1}^{N} (a_i - z_1) \cdot (b_i - z_2)$ involves integer-only arithmetic and it is guaranteed to fit in $16 \log N$ bits, since each summand, being the product of two 8-bit integers, fits in 16 bits. However, since $m = (m_1 m_2)/m_3$ is a float, the product $m \cdot s$ cannot be done with integer-only arithmetic. This product is handled in TFLite by essentially transforming m into a fixed-point number and then performing fixed-point multiplication, rounding to the nearest integer. More precisely, m is first normalized as $m = 2^{-n} m''$ where $m'' \in [0.5, 1)$,[6] and then m'' is approximated as $m'' \approx 2^{-31} m'$, where m' is a 32-bit integer. This is highly accurate since $m'' \geq 1/2$, so there are at least 30 bits of relative accuracy.

Thus, given the above, the multiplication $m \cdot s$ is done by computing the integer product $m \cdot s$, which fits in 64 bits since both m and s use at most 32 bits (if $N \leq 2^{16}$), and

[6]Jacob et al. [37] find that in practice $m \in [0, 1)$, which is the reason why such normalization is possible. We also confirm this observation in our experiments, although it is not hard to extend this to the general case (in fact, TFLite already supports it).

then multiplying by 2^{-n-31} followed by a rounding-to-nearest operation. Finally, addition with z_3 is done as simple integer addition.

If the quantization parameters for γ were computed correctly, it should be the case, by construction, that the result c lies in the correct interval $[0, 2^8)_{\mathbb{Z}}$. However, due to the different rounding errors that can occur above, this may not be the case. Thus, the result obtained with the previous steps is clamped into the interval $[0, 2^8)_{\mathbb{Z}}$.

3) Addition of bias: In the context of CNNs the dot products above will come from two-dimensional convolutions. However, these operations not only involve dot products but also the addition of a single number, the bias. In order to handle this in a smooth manner with respect to the dot product above, the scale for the bias is set as $m_1 m_2 / m_3$ and the zero-point it set to 0. This allows the quantized bias to be placed inside the summation s, involving no further changes to our description above.

4) Other layers: Other layers like ReLU, ReLU6 or max pooling, which involve only comparisons, can be implemented with relative ease directly on the quantized values, assuming these share the same quantization parameters. This is because if $\alpha = m(a - z)$ and $\beta = m(b - z)$, then $\alpha \leq \beta$ if and only if $a \leq b$, so the comparisons can be performed directly on the quantized values.

In fact, activations like ReLU6 (which is used extensively in the models we consider in this work) can be entirely fused into the dot product that precedes it, as shown in Section 2.4 of [37]. Since ReLU6 is essentially a clamping operation, it is possible, by carefully picking the quantization parameters, to make the clamping of the product to the interval $[0, 2^8)_{\mathbb{Z}}$ *also* take care of the ReLU6 operation. In short, if the zero-point is 0 and the scale is $6/255$, then we are guaranteed that $m(q - z) \in [0, 6]$ for any $q \in \{0, \ldots, 2^8 - 1\}$.

On the other hand, mathematical functions like sigmoid must be handled differently. We will not be concerned with this type of functions in this document since it is the case in practice that ReLU and ReLU6 (or similar activation functions) are typically enough.[7]

III. Quantized CNNs in MPC

In the previous section we discussed how quantization of neural networks works, or, more specifically, we discussed the quantization scheme by Jacob et al. [37]. Now, we turn to the discussion about how to implement these operations using MPC. However, before diving into the details of the protocols we use in this work, we describe the setting we consider for the secure evaluation of CNNs.

A. System and Threat Model

Like most previous work on secure inference using MPC, we consider a setting where both the model owner and client outsource their model, respectively input to a set of servers that perform that actual secure inference.

[7]See [37] for a discussion on quantization of mathematical functions.

		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
Dishonest Majority	Passive	OTSemi2k	OTSemiPrime
	Active	SPDZ2k	LowGear
Honest majority	Passive	Replicated2k	ReplicatedPrime
	Active	PsReplicated2k	PsReplicatedPrime

TABLE I: MPC protocols we use, classified depending on their security level (passive vs. active) and their arithmetic properties (modulo 2^k vs. modulo a prime)

We consider a setting of either two or three servers P_1, P_2 and P_3 depending on the setting (honest or dishonest majority) among which one is allowed to be corrupted.

The model and input owner each secret-shares their inputs to the servers at the beginning of the protocol execution. This preserves the privacy of this sensitive information under certain assumptions on the adversarial corruption. Then, the servers execute a secure multiparty computation protocol to evaluate the quantized model on the given input, obtaining shares of the output, which can then be sent to the party that is supposed to get the classification result.

Our techniques have the crucial feature that virtually *any* secret-sharing-based MPC protocol can be used as the underlying computation engine. More precisely, let R be either $\mathbb{Z}_{2^k}$ or $\mathbb{F}_p$, we assume a secret-sharing scheme $\langle \cdot \rangle$ over R for two or three parties (depending on the setting) withstanding one corruption, allowing local additions $\langle x + y \rangle = \langle x \rangle + \langle y \rangle$, together with a protocol for secure multiplication $\langle x \cdot y \rangle = \langle x \rangle \cdot \langle y \rangle$. In this work we consider several well-studied MPC protocols for different regimes: 2-party and 3-party computation (both withstanding one corruption), passive and active security, and computation modulo primes and powers-of-two. We refer to Section III-B for a thorough discussion on the protocols we consider in this work.

B. MPC Protocols

We consider a total of 8 MPC protocols to support the secure evaluation of the quantized CNNs, corresponding to all the possible combinations of active/passive, honest/dishonest majority and computation modolu a prime or a power-of-two. Table I contains an overview of which protocol is used in which security model. The rest of this subsection provides some details on each protocol.

Now that we abstracted away the provision of the input and the model, it remains to describe what MPC protocol the servers execute in order to compute the prediction securely. This includes the secret-sharing scheme used, and also how to handle multiplications, truncations and comparisons, among other details of the protocol.

In this work we leverage the capabilities of the MP-SPDZ framework to consider a total of 8 MPC protocols, each of them with its own advantages and disadvantages. Our protocols can be categorized in three different dimensions: amount of corruptions, type of corruptions and underlying algebraic structure. For the first dimension we distinguish between two cases: honest vs dishonest majority. In the first case the adversary is allowed to corrupt strictly less than half

of the parties, and we instantiate this case with 3 parties among which one corruption is tolerated. On the other hand, in the dishonest majority scenario the adversary can corrupt any number of parties as long as there is at least one remaining honest party, and here we consider 2 parties, which allows us to tolerate one corruption. While honest majority impose a stronger security assumption than dishonest majority, they tend to be simpler in their design and thus more efficient.

Regarding the type of corruption, we distinguish between passive and active corruptions. Actively secure protocols are preferred for sensitive applications since they remain secure even if the adversary deviates arbitrarily from the protocol specification, but they include an overhead with respect to their passive counterpart. Finally, the algebraic structure on which the computation takes place also plays an important role in terms of efficiency and protocol design, with protocols over $\mathbb{F}_p$ being easier to design and possibly implement, but protocols over $\mathbb{Z}_{2^k}$ providing some efficiency improvements in terms of basic arithmetic and bit-operations [16].

Given our considerations above, we proceed to describe each one of the protocols we consider in this work. We emphasize that the goal of implementing our CNN evaluation algorithm using such a wide range of protocols is to provide baselines for the performance-security tradeoff. A shared value under any of the protocols below is denoted by $\langle x \rangle$, and we only assume procedures for adding shared values (which do not involve communication) and for multiplying shared values. We also assume the existence of a procedure that allows servers to provide inputs: If server P_i has the value x, then $\mathsf{Input}(x)$ results in the parties having consistent shares $\langle x \rangle$.

1) Dishonest Majority: Protocols in the dishonest majority setting are often harder to develop and they are also more complex than honest majority ones. They are typically based in additive secret sharing and use authentication tags for active security to ensure that the openings of shared values are done correctly.

• SPDZ2k: This is the first actively secure protocol over $\mathbb{Z}_{2^k}$ in the dishonest majority setting, and it was proposed initially by Cramer et al. [13] and implemented subsequently by Damgård et al. [16]. This protocol can be seen as an extension of MASCOT [40] (itself being an extension of SPDZ, hence the name).

Multiplications in SPDZ2k are handled using multiplication triples, which are preprocessed using oblivious transfer like in MASCOT. Authentication is handled like in SPDZ, but with an addition that allows this method to work over $\mathbb{Z}_{2^k}$ which consists of working over the ring $\mathbb{Z}_{2^{k+s}}$ and using the upper s bits for authentication.

• OTSemi2k, OTSemiPrime: These protocols denote cut-down versions of SPDZ2k and MASCOT, respectively. In particular, they omit the usage of authentication tags and the so-called "sacrifice" where two triples are checked against each other and only of them can subsequently used in the protocol. There essentially remains the generation of multiplication triples using OT.

• LowGear: This is an actively secure protocol for computation modulo a prime. It uses semi-homomorphic encryption based on learning with errors. See Keller et al. [41] for details.

2) Honest Majority: Honest majority protocols are typically developed using Shamir Secret Sharing (for an arbitrary number of parties) or Replicated Secret Sharing (for small number of parties). Since we consider only a small number of servers we focus on the replicated SS instantiations.

• Replicated2k, ReplicatedPrime: This protocol secret-shares a value x among three parties P_1, P_2, P_3 by letting each P_i have random pairs (x_i, x_{i+1}) (indexes wrap around modulo 3) subject to $x \equiv x_1 + x_2 + x_3 \bmod M$, where $M = 2^k$ for the ring case and $M = p$ for the field case. The most efficient passively secure multiplication protocol to date is the one presented by Araki et al. [1], where the total communication involves 3 ring elements.

• PsReplicatedPrime: This protocol by Lindell and Nof [46] extends ReplicatedPrime to active security by preprocessing potentially incorrect triples and proceeding to the online phase using these, optimistically, checking their correctness at the end of the execution using sacrificing techniques.

• PsReplicated2k: This protocol by Eerikson et al. [21] is an extension of the one by Lindell et al. [46] to the ring setting. This is achieved by incorporating ideas by Cramer et al. [13] in order to adapt the post-sacrifice step by Lindell et al. to the ring $\mathbb{Z}_{2^k}$.

C. Building Blocks

For many applications, the multiplication protocol assumed for $\langle \cdot \rangle$ is not enough. In practice, many useful functionalities cannot be nicely expressed in terms of additions and multiplications and therefore, more often than not, researchers end up developing custom protocols for specific applications. As we argued in Section I-D, this also includes the case of secure evaluation of Neural Networks.

In our case, thanks to the quantization scheme by Jacob et al. [37] most of the operations in the evaluation of a quantized Neural Network become additions and multiplications, which are already supported by the MPC protocols we consider here. Furthermore, the multiplications have a very special structure: they are part of a dot product operation, which can be computed more efficiently by the passively secure protocols in the honest majority setting. However, the evaluation still requires non-arithmetic operations like truncations and comparisons, which are more expensive and require specialized subprotocols for their computation. We discuss these primitives now.

Some of these protocols have been already studied in the setting of computation over fields (e.g. [8, 14]), however, over rings, some of these operations are not possible anymore and therefore some ring-compatible variants are required. Furthermore, for the particular case of quantized Neural Networks it is also required to truncate by a *secret* amount, unlike the typical scenario in which this shift is public. Also, as a contribution of independent interest, we show how to reduce this computation to the well-studied case in which the shift is public.

We remark that we think of $x \in R$ as a signed integer in the interval $\left[-\frac{M}{2}, \frac{M}{2} - 1\right]$, where M is either p or 2^k.

D. MPC Primitives

1) Random Truncated Pairs: At the heart of many of the subprotocols below lies the creation of pairs $(\langle r\rangle, \langle r'\rangle)$, where $r \in R$ is uniformly random and $r' = r \bmod 2^m$ for some integer $0 < m < k$. This is achieved by sampling random shares $\langle r_i\rangle$ for $i = 1, \ldots, k$, where $r_i \in \{0, 1\}$, and then computing $\langle r\rangle = -2^{k-1}\langle r_{k-1}\rangle + \sum_{i=0}^{k-2} 2^i \langle r_i\rangle$ and $\langle r'\rangle = \sum_{i=0}^{m-1} 2^i \langle r_i\rangle$. We denote this procedure by $\mathsf{TruncPair}_R(m, k)$.

It then remains to show how to preprocess random shares $\langle b\rangle$ where $b \in \{0, 1\}$. This is achieved by letting parties P_i for $i = 1, 2$ sample a bit b_i and call $\mathsf{Input}(b_i)$ so that the parties obtain $\langle b_i\rangle$. Then the parties compute $\langle b\rangle$ where $b = b_1 \oplus b_2$ as $\langle b\rangle = \langle b_1\rangle + \langle b_2\rangle - \langle b_1\rangle \cdot \langle b_2\rangle$. Since at most one party is assumed to be corrupt (either in the honest or dishonest majority case), the resulting bit b is uniformly random.

The above procedure works for $R = \mathbb{F}_p$ or $R = \mathbb{Z}_{2^k}$. However, for active security, it is crucial to enforce that $b_i \in \{0, 1\}$, since in principle a corrupt party can input any $b_i \in R$. Fortunately, this can be easily checked by computing and opening $\langle b_i\rangle \cdot (1 - \langle b_i\rangle)$, and verifying that it equals 0. This works since $x \cdot (1 - x) = 0$ if and only if $x = 1$, both over $\mathbb{F}_p$ and $\mathbb{Z}_{2^k}$.

We remark that our bit-generation procedure does not scale well as more servers are involved, and we use it since we only consider two or three servers. For the more general case, the bit-generation procedures by Damgård [15] (for $\mathbb{F}_p$) and Damgård [16] (for $\mathbb{Z}_{2^k}$) are more suitable.

2) Comparison of Bitwise-Shared Values: In order the reduce the number of rounds we use a divide-and-conquer approach, that is we compare the lower and upper half independently and then combine the result. The base case is the comparison of two bits. This results in a logarithmic number of rounds while keeping the number of ANDs linear.

3) Modulo a Power of Two: The subprotocol from the previous section allows us to produce pairs $(\langle r\rangle, \langle r'\rangle)$ with $r' = r \bmod 2^m$. However, it is more useful to produce shares of $x' = x \bmod 2^m$ from a given shared value x. For this task we use the protocol Mod2m by Catrina and de Hoogh [8] presented below.

Protocol $\mathsf{Mod}_R(\langle x\rangle, m)$

The parties proceed as follows.

1) Let $(\langle r\rangle, \langle r'\rangle) \leftarrow \mathsf{TruncPair}_R(m, k)$. Furthermore, let $\langle r'\rangle_B$ be the shared bit-decomposition of $\langle r'\rangle$, which can be outputted by TruncPair.
2) Open $c \leftarrow 2^{k-1} + \langle x\rangle + \langle r\rangle$ and compute $c' = c \bmod 2^m$.
3) Let $\langle u\rangle \leftarrow \mathsf{BitLT}_R(c', \langle r'\rangle_B)$.
4) Return $\langle x'\rangle = c' - \langle r'\rangle + 2^m \langle u\rangle$.

4) Truncation by a Public Value: Now we discuss subprotocols for computing $\langle y\rangle$ from $\langle x\rangle$, where $y = \left\lfloor \frac{x}{2^m} \right\rfloor$. This is considered by Catrina and de Hoogh [8] by noticing that $\left\lfloor \frac{x}{2^m} \right\rfloor = \frac{x - x \bmod 2^m}{2^m}$. Hence, their protocol over fields, which we denote here by $\mathsf{Trunc}_{\mathbb{F}_p}(\langle x\rangle, m)$, simply calculates $\langle x \bmod 2^m\rangle \leftarrow \mathsf{Mod}_{\mathbb{F}_p}(\langle x\rangle, m)$, computes $\langle x - x \bmod 2^m\rangle$, and then multiplies by $1/2^m$. This last step is possible over the field $\mathbb{F}_p$ since 2^m is invertible in this structure, but not over $\mathbb{Z}_{2^k}$.

We develop a specific truncation method for the $\mathbb{Z}_{2^k}$ setting which follows the template by Catrina and de Hoogh but replaces the final multiplication by $1/2^m$ with a shift. The protocol is described below.

Protocol $\mathsf{Trunc}_{\mathbb{Z}_{2^k}}(\langle x\rangle, m)$

The parties proceed as follows.

1) Call $\langle x \bmod 2^m\rangle \leftarrow \mathsf{Mod}_{\mathbb{Z}_{2^k}}(\langle x\rangle, m)$ and let $\langle v\rangle = \langle x\rangle - \langle x \bmod 2^m\rangle$;
2) Sample $k-m$ random bits $\langle r_0\rangle, \ldots, \langle r_{k-m-1}\rangle \leftarrow \mathsf{RandBit}_{\mathbb{Z}_{2^k}}(\cdot)$ and compute $\langle r\rangle = \sum_{i=0}^{k-m-1} 2^i \langle r_i\rangle$.
3) Compute and open $c \leftarrow \langle v\rangle + 2^m \langle r\rangle$, and let $c' = \left\lfloor \frac{c}{2^m} \right\rfloor$.
4) Compute $\langle u\rangle \leftarrow \mathsf{BitLT}_{\mathbb{Z}_{2^k}}(c', \langle r\rangle_B)$.

5) Truncation by a Secret Value: The truncation protocol we have described in Section III-D4 assumes that the amount of bits to be truncated, m, is public. This is a natural setting and appears for instance in fixed-point multiplication, where m is equal to the amount of bits assigned for the decimal part. However, as we already argued in Section II, the quantization scheme we use here differs from traditional fixed-point arithmetic in that the parameters for the discretization are adaptively chosen for each particular layer of the network. As a side effect, these parameters become information of the model, and therefore they must not be revealed in the computation. As a result, truncation by secret amounts become necessary.

In this section we present our protocol for truncation by a secret amount. It takes as input a secret $\langle x\rangle$ and a shift m represented by $\langle 2^{M-m}\rangle$ where M is some public upper bound on m,[8] and outputs $\langle y\rangle$ where $y = \frac{x}{2^m}$.

Protocol $\mathsf{TruncPriv}_R(\langle x\rangle, \langle m\rangle)$

The parties proceed as follows.

1) Compute $\langle 2^{M-m} \cdot x\rangle = \langle 2^{M-m}\rangle \cdot \langle x\rangle$.
2) Return $\langle y\rangle \leftarrow \mathsf{Trunc}_R(\langle 2^{M-m} \cdot x\rangle, M)$.

The only requirement for this protocol to work is that $(M - m) + \log_2(|x|) < k$ In this case, it can be seen that $2^{M-m} \cdot x$ does not overflow and therefore the final truncation returns $\left\lfloor \frac{2^{M-m} \cdot x}{2^M} \right\rfloor = \left\lfloor \frac{x}{2^m} \right\rfloor$, as required.

6) Probabilistic Truncation: The most expensive part of the above protocols for truncation is the bit-wise comparison. In particular, it is the only part whose number of rounds depends on the bit lengths m and k. Catrina and Saxena [9] have proposed a protocol that achieves truncation in a constant number of rounds at the cost of a deterministic result. Instead, the result

[8] We may alternatively assume that m itself is shared. The conversion $\langle m\rangle \to \langle 2^{M-m}\rangle$ can be achieved then by first bit-decomposing $M - m$ as $\sum_i 2^i \cdot b_i$, computing shares of each b_i and then outputting $\langle 2^{M-m}\rangle = \prod_i (1 + \langle b_i\rangle \cdot (2^{2^i} - 1))$. However, since in our setting m is known by the client who has the model, it is simpler to assume that the client distributes $\langle 2M - m\rangle$ to begin with.

will be probabilistically rounded up or down depending on the how close the input is to either side. As an example consider the truncation of $x\|01$ by two bits, which would result in x with probability 0.75 and $x+1$ with probability 0.25.

Catrina and Saxena's protocol works only for prime order fields because it uses the inverse of 2^m. In the following we present an adaption for rings.

Protocol $\mathsf{TruncPr}_{\mathbb{Z}_{2^k}}(\langle x\rangle, m)$

Pre $m \in [-2^{k-1}, 2^{k-1}-1]$
Post $\langle x/2^m\rangle$ rounded according to text
Proceed as follows:

1) Generate k random bits $\langle b_i\rangle$ and compute $\langle r\rangle \leftarrow \sum_i \langle r_i\rangle \cdot 2^i$.
2) Open $c \leftarrow \langle x\rangle + \langle r\rangle$ and compute $c' \leftarrow (c/2^m) \bmod 2^{k-m-1}$.
3) Compute $\langle b\rangle \leftarrow \langle r_{k-1}\rangle \oplus (c/2^{k-1})$.
4) Output $c - \sum_{i=m}^{k-2} \langle r_i\rangle \cdot 2^{i-m} + b \cdot 2^{k-m-1}$.

The analysis is similar to the protocol for prime order fields, except that b accounts for the overflow of the addition $x + \sum_i^{k-2} r_i \cdot 2^i$.

7) Special probabilistic truncation: This protocol emulates the black-box probabilistic truncation above in the setting of semi-honest computation over a power of two with an honest majority. Informally, it changes from a symmetric three-party protocol to a two-party protocol where the third party generates correlated randomness used by the the other parties. This allows to generate random values of any bit length at once without the need to generate such random values bit-wise. The latter is the main cost in black-box probabilistic truncation because the communication is independent of the number of bits otherwise.

Protocol $\mathsf{TruncPrSp}_{\mathbb{Z}_{2^k}}(\langle x\rangle, m)$

P_3 proceeds as follows:

1) Sample random bits $\{r_i\}$ for $i \in [0, k-1]$.
2) Generate 2-out-of-2 sharings of $r = \sum_i r_i \cdot 2^i$, r_{k-1}, and $\sum_{i=m}^{k-2} r_i \cdot 2^{i-m}$, and send one share to P_1 and P_2 each.
3) Generate random $y_1, y_3 \in \mathbb{Z}_{2^k}$ and send y_1 to P_1 and y_3 to P_2.
4) Output (y_3, y_1).

P_1 and P_2 proceed as follows:

1) Convert $\langle x\rangle$ to a 2-out-of-2 sharing by P_1 computing $x_1 + x_2$ and P_2 proceeding with x_3.
2) Execute $\mathsf{TruncPr}$ as two-party computation using the random values received from P_3.
3) P_i: Let y_i' denote the share output by $\mathsf{TruncPr}$ and $\hat{y}_i$ the share received from P_3 (y_1 or y_3). Send $y_i' - \hat{y}_i$ to P_{2-i}. Denote the received value by $\tilde{y}_i$.
4) P_1 outputs $(y_1, y_1' - \hat{y}_1 + \tilde{y}_1)$, and P_2 outputs $(y_2' - \hat{y}_2 + \tilde{y}_2, y_3)$.

For correctness, we have to establish that the parties output a correct replicated secret sharing of the result. To establish the correct replicated secret sharing, consider

$$\begin{aligned} y_1' - \hat{y}_1 + \tilde{y}_1 &= y_1' - \hat{y}_1 + y_2' - \hat{y}_2 \\ &= \tilde{y}_2 + y_2' - \hat{y}_2. \end{aligned}$$

Furthermore,

$$\begin{aligned} y_1 + y_3 + y_1' - \hat{y}_1 + \tilde{y}_1 &= y_1 + y_3 + y_1' - \hat{y}_1 + y_2' - \hat{y}_2 \\ &= y_1 + y_3 + y_1' - y_1 + y_2' - y_2 \\ &= y_1' + y_2', \end{aligned}$$

which equals the result of $\mathsf{TruncPr}$ by definition.

Since we only aim for semi-honest security with honest majority, we have to show that each party does not learn any information about x if all parties follow the protocol. This is trivial for P_3 because they do not receive anything. For P_1 and P_2, the randomness received from P_3 is independent of x. Furthermore, the security of the two-party $\mathsf{TruncPr}$ execution follows by the black-box definition of it. Finally, $\tilde{y}_i$ does not reveal information because $\hat{y}_{2-i}$ is uniformly random and unknown to P_i.

8) Integer Comparison: This is derived from Catrina and de Hoogh [8] by replacing the truncation. Essentially, we extract the most significant bit from the difference of two values by exact truncation.

E. Putting it all Together

In this section we use the building blocks from Section III-C in order to securely evaluate a CNN that is quantized under the scheme from Section II-C. As we discussed there, evaluating a quantized CNN consists mostly of computing the expression in Eq. (2), followed by a clamping procedure. We describe these computations in this section, along with the other necessary pieces for the evaluation of a quantized CNN.

Recall from Section II that each weight tensor $\boldsymbol{a}$ in a quantized CNN has a scale $m \in \mathbb{R}$ and a zero-point $z \in \mathbb{Z}_{2^8}$ associated to it, such that $\alpha \approx m \cdot (a - z)$ is the actual floating-point numbers corresponding to each 8-bit integer a in the tensor. Also, biases are quantized in a similar manner but with a 32-bit integer instead, a zero point equal to 0, and a scale that depends on the inputs and output to the layer it belongs to, as explained in Section II-C2. We assume that the model owner, who knows all this information, distributes shares to the servers using the scheme described above of the quantized weights and biases of each layer in the network.[9] Also, the zero points associated to each tensor is shared towards the parties.

The scales of the model, on the other hand, are handled in a slightly different way. Each dot product in the quantized network requires a fixed-point multiplication by a factor $m = (m_1 \cdot m_2)/m_3$, borrowing the notation from Section II-C2. Recall that this product was handled by writing $m = 2^{-n-31} \cdot m'$, where m' is a 32-bit integer.

[9]Notice that these values are only 8-bit long in the clear, but the shares are 64-bit long. The reason is that, although the values are small, the computation must be carried without overflow. Therefore we cannot use a modulus that is smaller than the maximum possible intermediate value.

1) Secure Computation of a Quantized Dot Product: We show how to compute securely the expression in Eq. (2). Given the setting we described above, the parties have shares of the zero points z_1, z_2, z_3, the quantized inputs a_i, b_i for $i = 1, \ldots, N$, the integer scale m' and the power $2^{L-\ell}$, where $\ell = n + 31$ with $2^{-n-31} \cdot m' \approx m = (m_1 \cdot m_2)/m_3$, and L is an upper bound on ℓ.[10]

In order to compute the expression in Eq. (2), the parties begin by computing the dot product $\langle s \rangle = \sum_{i=1}^{N} (\langle a_i \rangle - \langle z_1 \rangle) \cdot (\langle b_i \rangle - \langle z_2 \rangle)$. As described in Section III-B, some underlying MPC protocols allow this computation to be done at the expense of one single multiplication. Otherwise, this must be done using N calls to the underlying multiplication protocol. Then, an additional secure multiplication is used in order to compute $\langle m \cdot s \rangle = \langle m \rangle \cdot \langle s \rangle$. Next, shares of $\left\lfloor 2^{-n-31} \cdot (m \cdot s) \right\rceil$ are computed from $\left\langle 2^{L-\ell} \right\rangle$ and $\langle m \cdot s \rangle$ using Protocol $\mathsf{TruncPriv}$ from Section III-D5, together with the observation that $\left\lfloor 2^{-m} \cdot x \right\rceil = \left\lfloor 2^{-m} \cdot x + 0.5 \right\rfloor = \left\lfloor 2^{-m} \cdot (x + 2^{m-1}) \right\rfloor$ for breaking a tie by rounding up.

Finally, addition with $\langle z_3 \rangle$ is local, and it is followed by the clamping method described in Section III-E2 below.

2) Clamping: For the final operation the parties hold $\langle x \rangle$ and need to compute $\langle y \rangle$ where $y = \mathsf{clamp}_{0,2^8-1}(x)$. This is done by comparing $\langle x \rangle$ to the limits (0 and 255) using the protocol from Section III-D8, followed by oblivious selection: If $s \in \{0, 1\}$, it holds trivially that $a_s = s \cdot (a_1 - a_0) + a_0$ for arbitrary a_0, a_1.

3) Average and Max Pooling: Average pooling involves computing $\langle y \rangle$ from $\langle x_1 \rangle, \ldots, \langle x_n \rangle$, where $y = \left\lfloor \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \right\rceil$. This can be achieved using Goldschmidt's algorithm [25], a widely used iterative algorithm for division. For its usage in the context of secure multiparty computation, see for example Catrina and Saxena [9]. It uses basic arithmetic as well as truncation, both of which we have been discussed already.

On the other hand, max pooling requires implementing the max function securely, which can be easily done by making use of a secure comparison protocol [8].

4) Output Layer: Once shares of the output vector are obtained (raw output, before applying Softmax), several options can be considered. The parties could open the vector itself towards the input owner and/or data owner so that they compute the Softmax function and therefore learn the probabilities for each label. However, this would reveal all the prediction vector, which could be undesirable in some scenarios. Thus, we propose instead to securely compute the argmax of the output array, and return this index, which returns the most likely label since exponentiation is a monotone increasing function.

Previous work, such as SecureML [50], replace the exponentiation in the Softmax function with ReLU operations, i.e. by computing $\mathsf{ReLU}(x)$ instead of e^x. More MPC friendly solutions exist, such as the spherical Softmax [18], which replaces e^x with x^2.

[10]Since $n \leq 32$ it suffices to take $M = 63$. In this case, given that $m \geq 31$, it follows that $2^{M-m} \leq 2^{32}$. According to Section III-D5, this imposes the restriction that the modulus for the computation must be at least $32+64 = 96$. In practice n is smaller than 32 and this bound can be improved.

IV. Implementation and Benchmarking

This section discusses our implementation and our performance results. We begin by describing in Section IV-A the families of CNNs we implement on this work, *MobileNets*. Then we discuss in Section IV-B the results we obtained by running these networks with our framework.

A. MobileNets Architecture

MobileNets [33] is a family of networks for the ImageNet challenge. These are networks of 28 hidden layers with 1000 outputs and are used to classify images. There are currently two versions of the MobileNets architecture, and our focus will be on the original V1 family. The structure of a MobileNetsV1 network is fairly straightforward: the input layer is a regular convolution (and in fact the only such one). This layer is immediately followed by 13 depthwise separable convolutions, i.e. 13 alternating depthwise and pointwise convolutions. The last two layers are an average pool followed by a fully connected layer, for a total of 28 layers (not counting a softmax at the end for turning the result into a probability distribution).

The architecture defines two hyper parameters that allow the user to scale the network in different ways, namely a *width multiplier* α and a *resolution multiplier* ρ. The width multiplier scales the input and output channels. I.e., the number of multiply-add instructions in each convolution becomes $\alpha I_d \cdot I_h \cdot I_w \cdot W_h \cdot W_w \cdot \alpha O_d$: the α essentially serves to thin the network. Note that α also affects the number of parameters and thus the model size. The resolution parameter on the other hand simply scales the input; at $\rho = 1.0$ the input is set to be 224×224 and pretrained networks exist for $\rho \approx 0.85$ (192×192), $\rho \approx 0.71$ (160×160) and $\rho \approx 0.57$ (128×128). Note that scaling ρ affects the number of multiply-add operations, but does not affect the size of the network. In the following, a network with a particular set of parameters is denoted as "`V1` α_S" where H denotes the height and width of the input (i.e., is a value related to ρ).

B. Implementation

We implement secure inference in the MP-SPDZ framework, which allows us to get timings for all the protocols described in Section III-B. These protocols run over either a prime p or a ring $\mathbb{Z}_{2^k}$. The prime is 128 bits while the k we use for the ring is 72 bits. As described in Section III-E, these arise because we need some extra space in order for the truncation by a secret shift to be correct. We arrive at 72 experimentally by computing the sizes of the shift and dot-products needed in the models we evaluate.

a) Experimental setup: We ran all our benchmarks on colocated c5.9xlarge AWS machines, each of which has 36 cores, 72gb of memory, a 10gpbs link between them and submillisecond latency. Throughout this section, communication is measured per party and all timings include preprocessing. Our code has been published as part of MP-SPDZ [17].[11]

[11]The scripts necessary to convert the published models and images can be found at https://github.com/anderspkd/SecureQ8.

Variant	Accuracy		Trunc.	Passive Security				Active Security			
				Dishonest Maj.		Honest Maj.		Dishonest Maj.		Honest Maj.	
	Top-1	Top-5		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
V1 0.25_128	39.5%	64.4%	Prob.	139.5	465.1	0.9	3.7	1264.9	1377.8	9.5	9.2
			Exact	203.2	564.5	1.7	4.3	1864.9	1592.0	11.1	10.9
V1 0.25_160	42.8%	68.1%	Prob.	214.5	713.2	1.5	5.9	1997.4	2070.4	15.0	14.6
			Exact	317.7	878.2	2.6	6.8	2916.1	2432.8	17.4	17.4
V1 0.25_192	45.7%	70.8%	Prob.	305.1	1020.2	2.1	8.4	2827.8	2875.0	20.8	20.5
			Exact	460.6	1245.5	3.7	9.7	4173.9	3389.8	24.5	24.3
V1 0.25_224	48.2%	72.8%	Prob.	417.6	1391.5	3.0	11.3	3825.6	3855.3	28.0	27.4
			Exact	614.1	1684.3	5.1	13.0	5629.6	4574.0	33.1	32.8
V1 0.5_128	54.9%	78.1%	Prob.	305.4	980.6	1.9	7.7	2731.5	2760.4	19.1	19.3
			Exact	430.1	1148.3	3.4	8.9	3950.3	3183.4	22.4	22.7
V1 0.5_160	57.2%	80.5%	Prob.	472.6	1499.5	3.4	12.4	4331.5	4277.2	30.0	30.6
			Exact	672.1	1802.6	5.6	14.2	6177.5	5006.9	35.2	35.7
V1 0.5_192	59.9%	82.1%	Prob.	676.1	2172.1	4.7	17.7	6194.6	6026.6	42.9	43.6
			Exact	978.0	2582.7	8.0	20.3	8924.5	7025.4	49.7	51.0
V1 0.5_224	61.2%	83.2%	Prob.	915.6	2947.8	6.5	24.0	8446.5	8112.9	57.2	59.8
			Exact	1320.6	3504.5	11.0	27.5	11962.2	9143.3	67.5	70.0
V1 0.75_128	55.9%	79.1%	Prob.	485.4	1538.5	3.3	12.2	4440.8	4152.6	29.1	31.0
			Exact	697.3	1813.6	5.4	14.0	6203.2	4754.5	34.1	35.7
V1 0.75_160	62.4%	83.7%	Prob.	775.7	2449.3	5.6	19.8	7018.5	6502.8	46.5	49.3
			Exact	1075.8	2851.4	9.0	22.4	9780.5	7491.2	54.2	56.5
V1 0.75_192	66.1%	86.2%	Prob.	1101.1	3419.4	8.2	28.1	10053.3	9145.2	67.0	71.3
			Exact	1536.9	4089.4	13.1	32.0	13991.2	10696.1	78.2	81.0
V1 0.75_224	66.9%	86.9%	Prob.	1487.2	4730.8	11.2	38.0	13634.5	12367.3	89.8	93.7
			Exact	2135.5	5602.5	18.3	43.3	18962.2	14370.4	105.1	108.6
V1 1.0_128	63.3%	84.1%	Prob.	709.4	2179.4	4.8	17.3	6381.8	5733.0	40.1	44.1
			Exact	968.5	2514.9	7.6	19.5	8797.0	6624.6	46.8	49.7
V1 1.0_160	66.9%	86.7%	Prob.	1101.8	3386.4	8.1	27.8	10142.0	9006.3	64.1	69.0
			Exact	1528.0	3991.2	12.8	31.2	13780.4	10357.2	74.8	79.9
V1 1.0_192	69.1%	88.1%	Prob.	1581.6	4900.8	11.9	39.7	14471.8	12778.3	91.6	97.5
			Exact	2214.8	5714.6	18.6	44.6	19725.0	14770.0	107.2	112.6
V1 1.0_224	70.0%	89.0%	Prob.	2147.3	6725.0	16.3	53.8	19691.6	17211.3	124.8	134.0
			Exact	2943.3	7728.7	25.8	60.5	26714.3	19910.4	147.8	154.4

TABLE II: Running time, in seconds, of securely evaluating some of the networks in the MobileNets family, in a LAN network. The first number in variant is the width multiplier and the second is the resolution multiplier. Top-1 accuracy measures when the truth label is predicted correctly by the model whereas Top-5 measures when the truth label is among the first 5 outputs of the model. Prob. and Exact refer to probabilistic truncation and nearest rounding, respectively.

C. Full model evaluation

We evaluate almost all available pre-trained V1 MobileNets models in all of our settings. For each model protocol and model combination, we run a secure prediction using probabilistic truncation and exact (rounding to nearest) truncation. The rationale behind presenting numbers for both methods of truncation, is to illustrate the increase in efficiency when a less exact method of truncation protocol is used. We stress that using a probabilistic may hurt the accuracy and that the accuracy reported by Tensorflow during training may not be guaranteed. results are presented in two tables: Table II presents running time in seconds, while Table III in the appendix shows the amount of data that each combination needs to communicate. Note that the figures include preprocessing where applicable because MP-SPDZ executes this on demand. The figures for active security with dishonest majority over $\mathbb{F}_p$ have been gathered using LowGear while all other dishonest-majority protocols using OT. However, they do not include a one-time key generation with active security because MP-SPDZ does not implement that.

Models with higher accuracy are larger, and so it is not surprising that accuracy and running time/communication are correlated. More interesting is the relationship between the different threat models. We see that honest-majority protocols greatly outperform dishonest-majority protocols. In the latter setting, the protocols require either oblivious transfer or homomorphic encryption, which is much more costly, than the pure secret sharing used otherwise. Another interesting difference is between modulo 2^k and modulo p protocols. Computation in the former is in some way simpler—in particular, it is supported in hardware (e.g., mod 2^{64} corresponds to using unsigned long integers). However, modulo p allows to use homomorphic, which reduces time for most models and significantly reduces communication for all models. A final note is with regard to the difference between the two honest majority protocols with passive security and with active security. Interestingly, there is a larger gap between the modulo p and modulo 2^k protocols in the passive version, than in the active version (cf Fig. 3). We attribute this difference to the fact that the simpler computations modulo 2^k are more pronounced when correctness does not need to be enforced (i.e., when the protocol only needs passive security).

1) Comparison with CrypTFlow: Although we do not use the same setup as CrypTFlow [44], a comparison is still possible. We note that the `0.50_128` model enjoys a similar accuracy as the SqueezeNet model that is evaluated in

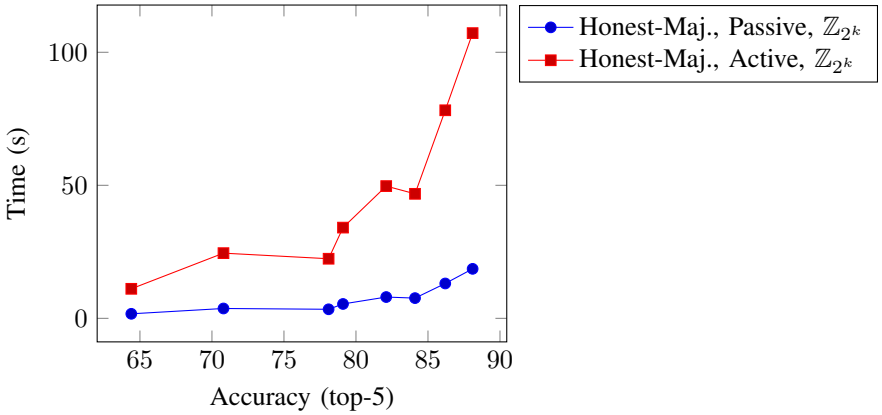

Fig. 3: Performance of securely evaluating some of the networks in the MobileNets family (the 128 and 192 variants) with passive and active security.

CrypTFlow. Comparing these two, we see that our approach achieves a significantly faster evaluation time for both active and passive security, albeit at a higher communication cost. Because of the differences in the setup, a more accurate comparison requires us to scale our results to match those in CrypTFlow.

D. WAN Benchmarks

We have also run the smallest model in a WAN setting where each party is located on a different continent. For computation over rings with probablistic truncation, the timings range from 110 seconds for passive honest-majority computation to 28,000 seconds for active dishonest-majority computation.

V. Conclusions

We show that it is possible to securely evaluate large and realistic networks, so called ImageNet networks, using more-or-less existing MPC protocols. Moreover, the networks we evaluate are unmodified and can be trained using standard Tensorflow. This work thus provides a very appealing approach to secure evaluation from an end-users perspective: First, because standard MPC suffices, it is possible to choose from a wider array of threat models than previous works allow. While the passive security honest majority setting is by far the most efficient, our benchmarks still provide an interesting insight into the exact trade-off one wants secure inference against dishonest majority. Second, the fact that models directly output by Tensorflow can be evaluated *without* modification, means that model designers can remain oblivious to the secure framework.

Finally, and like in recent and concurrent work, we obtain nice results by relying on quantization. Our work can therefore be seen as further evidence towards the fact that specific models greatly improve the range of models in terms of what can be evaluated securely.

A. Future Work

a) MPC/FHE-aware ML research: The quantization scheme used in TFLite is what enables us to achieve secure inference using standard MPC protocols. Quantization techniques are developed with the goal of making models smaller, however our work shows that it applies to other problem areas—secure inference in particular. We believe then that an interesting problem within the intersection of machine learning and security, is how to design quantization schemes that are "secure evaluation friendly" from the outset.

b) Training: Our protocol is designed for secure inference, and we leave it as future work to apply quantization techniques in order to realize secure training of quantized CNNs with little-to-no accuracy loss with respect to their floating-point counterparts. It is important to notice that the spectrum of research in quantized training of CNNs is much more reduced than in quantized inference, with only a few recent works like [35], [65] and [66]. Moreover, it is not clear yet how compatible these methods are with MPC or FHE techniques.

Acknowledgements

This work has been supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO), and the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC) and the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA).

We thank Adrià Gascón for fruitful discussions, and also NEC and BIU members, especially Prof. Benny Pinkas and Prof. Yehuda Lindell for very useful comments on this work. We thank Assi Barak (who was previously on this paper) for many helpful discussions in the initial phases of this project.

APPENDIX

A. MobileNets Blocks

We present below some of the different blocks present in the MobileNets family of networks.

a) Depthwise Separable Convolutions: The majority of the computation that a CNN performs, and the space that it uses, is tied to its convolutional operations. A regular convolution performs at its core two steps: first it filters the input using a set of trained weights, by moving each filter over the entire input; and second, a convolution combines the output of each filter application to produce a single output value. The whole operation can be viewed as an entry-wise product between a filter and each input in a specific window, followed by a summation of all the products. The price of a convolution is therefore $I_d \cdot I_h \cdot I_w \cdot W_h \cdot W_w \cdot O_d$. What the MobileNets models does instead, is to replace these convolutions with a *depthwise separable convolution*. At a high level, the idea behind a depthwise separable convolution is to split the two tasks outlined above into to separate operations.

More precisely, instead of performing a normal convolution, first a depthwise convolution is performed, which is like a regular convolution except it does not change the output depth; afterwards, a *pointwise* convolution is performed. A pointwise is a regular convolution with a 1×1 filter, which preserves the input dimensions but allows for scaling of the depth. The cost of the depthwise convolution is $I_h \cdot I_w \cdot I_d \cdot W_w \cdot W_h$ while the cost of the pointwise convolution is $O_d \cdot I_d \cdot I_h \cdot I_w$. Replacing a normal regular convolution with a Depthwise Separable convolution provides a saving of $1/O_d + 1/(W_h \cdot W_w)$ in terms of computation.

In MobileNetsV1, both the depthwise and pointwise convolution are followed by a batch normalization layer, and a ReLU6 activation.

b) ReLU6: As mentioned already in Section II-C4, the ReLU6 operation can be computed as part of the clamping when quantization is used and so this operation is also not explicit in the quantized version of a MobileNets model. (It is, however, needed in the floating point variant.)

B. Quantized training with Tensorflow

Tensorflow supports quantized training in two variants: *post-training quantization*[12] which quantizes and already trained floating point model, and *quantization aware training* [27] which performs training using floating point numbers, but inserts specific nodes in the model during training that lets the model "learn" that it will be quantized later on.

The following appendix illustrates via a small example how to train and quantize an MNIST model which can then be evaluated securely using. Training a model and obtaining a file which can be used as input to our protocol proceeds in three steps, each of which are simple enough that they in total require less than 100 lines of Python and shell code.

```
graph = tf.Graph()
sess = tf.Session(graph=graph)
keras.backend.set_session(sess)
with graph.as_default():
    # build model
    model = ... # keras model

    # create a quantized training graph. This
    # inserts fake quantization nodes that
    # emulate quantization when it's actually
    # used in the inference phase.
    tf.contrib.quantize.create_training_graph(
        graph
    )

    # run, compile and train the model
    sess.run(tf.global_variables_initializer())
    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    model.fit(x_train, y_train)

    # save stuff so we can resume training if
    # need be. Also needed when we want
    # to extract the quantized model
    saver = tf.train.Saver()
    saver.save(sess, checkpoint_directory)

    loss, accuracy = model.evaluate(
        x_test, y_test
    )
    print('\nEvaluation results:')
    print('test loss', loss)
    print('test accuracy', accuracy)
```

Fig. 4: Quantize aware training of a Keras model in Tensorflow.

a) Step 1. Quantize aware training: The first step is to train a model, ideally in a way which accounts for the quantization that is enable during inference. The details of how this training process is implemented (in particular, how to make the model aware of the quantization that happens during inference) are described in the [42]. Here we only focus on how to achieve this using Tensorflow.

The code in Fig. 4 illustrates how Tensorflow can be used to train a model in a which takes quantization into account. Notice that the model is specified using the `Keras` high level framework, rather than directory in Tensorflow.

b) Step 2. Converting training checkpoints to a frozen graph: The code in Figure 4 generates a set of checkpoint files that can frozen in order to obtain a model file. That is, a file which contains both the description of the model as well as the weights.

c) Step 3. Quantizing the model: The final step consists of using the `tflite_convert`[13] tool that is part of TFLite to convert the frozen model into a quantized model file.

[12]https://www.tensorflow.org/lite/performance/post_training_quantization

[13]https://www.tensorflow.org/lite/convert/

Variant	Accuracy		Trunc.	Passive Security				Active Security			
				Dishonest Maj.		Honest Maj.		Dishonest Maj.		Honest Maj.	
	Top-1	Top-5		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
V1 0.25_128	39.5%	64.4%	Prob.	199.2	655.3	0.8	3.4	1748.4	282.4	5.3	8.7
			Exact	296.3	793.3	1.7	4.0	2578.2	335.3	7.5	10.3
V1 0.25_160	42.8%	68.1%	Prob.	311.2	1023.3	1.2	5.4	2731.2	423.9	8.3	13.6
			Exact	462.8	1239.0	2.6	6.2	4027.1	511.9	11.8	16.1
V1 0.25_192	45.7%	70.8%	Prob.	447.5	1471.8	1.7	7.7	3927.0	600.0	11.9	19.6
			Exact	665.6	1782.0	3.8	8.9	5792.2	723.3	16.9	23.2
V1 0.25_224	48.2%	72.8%	Prob.	608.5	2001.5	2.4	10.5	5339.9	811.3	16.2	26.6
			Exact	905.3	2423.7	5.2	12.2	7877.7	987.2	23.0	31.5
V1 0.5_128	54.9%	78.1%	Prob.	438.0	1399.4	1.5	6.9	3834.5	581.7	11.5	18.5
			Exact	631.8	1675.2	3.4	8.0	5492.6	687.4	16.0	21.7
V1 0.5_160	57.2%	80.5%	Prob.	684.6	2186.8	2.4	10.7	5993.7	899.7	18.0	28.8
			Exact	987.4	2617.6	5.3	12.4	8583.4	1075.6	25.0	33.9
V1 0.5_192	59.9%	82.1%	Prob.	984.8	3146.1	3.5	15.5	8621.7	1286.9	25.9	41.5
			Exact	1420.7	3766.4	7.6	17.9	12349.8	1533.4	35.9	48.7
V1 0.5_224	61.2%	83.2%	Prob.	1339.4	4279.6	4.7	21.0	11725.6	1744.6	35.2	56.4
			Exact	1932.6	5123.8	10.3	24.3	16799.2	2079.1	48.9	66.3
V1 0.75_128	55.9%	79.1%	Prob.	716.9	2234.5	2.3	10.3	6264.3	916.3	18.6	29.3
			Exact	1007.6	2648.1	5.1	11.9	8750.4	1074.9	25.4	34.1
V1 0.75_160	62.4%	83.7%	Prob.	1120.8	3492.4	3.6	16.1	9793.1	1428.3	29.1	45.7
			Exact	1574.8	4138.6	7.9	18.6	13676.4	1692.3	39.6	53.3
V1 0.75_192	66.1%	86.2%	Prob.	1612.4	5025.2	5.2	23.2	14089.1	2044.4	41.9	65.8
			Exact	2266.2	5955.4	11.4	26.8	19680.3	2432.0	57.0	76.6
V1 0.75_224	66.9%	86.9%	Prob.	2193.2	6836.2	7.1	31.5	19163.5	2783.4	57.0	89.5
			Exact	3082.9	8102.4	15.5	36.5	26773.0	3294.5	77.6	104.3
V1 1.0_128	63.3%	84.1%	Prob.	1035.9	3160.5	3.1	13.7	9037.9	1286.1	26.7	41.1
			Exact	1423.5	3712.0	6.7	15.9	12352.1	1514.8	35.6	47.6
V1 1.0_160	66.9%	86.7%	Prob.	1619.6	4940.2	4.8	21.5	14128.8	2009.9	41.8	64.3
			Exact	2224.9	5801.6	10.5	24.8	19306.2	2361.8	55.7	74.3
V1 1.0_192	69.1%	88.1%	Prob.	2330.3	7109.0	6.9	30.9	20328.5	2889.9	60.1	92.5
			Exact	3201.9	8349.3	15.2	35.7	27782.7	3400.7	80.2	106.9
V1 1.0_224	70.0%	89.0%	Prob.	3169.9	9671.5	9.4	42.0	27652.5	3928.3	81.7	125.8
			Exact	4356.2	11359.7	20.6	48.6	37798.3	4615.2	109.1	145.5

TABLE III: Communication complexity, in Gigabytes, of securely evaluating some of the networks in the MobileNets family, in a LAN network. The first number in variant is the width multiplier and the second is the resolution multiplier. Top-1 accuracy measures when the truth label is predicted correctly by the model whereas Top-5 measures when the truth label is among the first 5 outputs of the model.

REFERENCES

[1] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 805–817, Vienna, Austria, Oct. 24–28, 2016. ACM Press.

[2] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski. Garbled neural networks are practical. Cryptology ePrint Archive, Report 2019/338, 2019. https://eprint.iacr.org/2019/338.

[3] M. Ball, T. Malkin, and M. Rosulek. Garbling gadgets for boolean and arithmetic circuits. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 565–577, Vienna, Austria, Oct. 24–28, 2016. ACM Press.

[4] W. Balzer, M. Takahashi, J. Ohta, and K. Kyuma. Weight quantization in boltzmann machines. *Neural Networks*, 4(3):405–409, 1991.

[5] M. Barni, C. Orlandi, and A. Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pages 146–151. ACM, 2006.

[6] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[7] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512, Santa Barbara, CA, USA, Aug. 19–23, 2018. Springer, Heidelberg, Germany.

[8] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. A. Garay and R. D. Prisco, editors, *SCN 10: 7th International Conference on Security in Communication Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199, Amalfi, Italy, Sept. 13–15, 2010. Springer, Heidelberg, Germany.

[9] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In R. Sion, editor, *FC 2010: 14th*

International Conference on Financial Cryptography and Data Security, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50, Tenerife, Canary Islands, Spain, Jan. 25–28, 2010. Springer, Heidelberg, Germany.

[10] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. http://eprint.iacr.org/2017/035.

[11] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[12] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[13] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798, Santa Barbara, CA, USA, Aug. 19–23, 2018. Springer, Heidelberg, Germany.

[14] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In S. Halevi and T. Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304, New York, NY, USA, Mar. 4–7, 2006. Springer, Heidelberg, Germany.

[15] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, Sept. 9–13, 2013. Springer, Heidelberg, Germany.

[16] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, pages 1325–1343, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[17] Data61. MP-SPDZ - Versatile framework for multi-party computation. https://github.com/data61/MP-SPDZ.

[18] A. de Brébisson and P. Vincent. An exploration of softmax alternatives belonging to the spherical loss family. *arXiv preprint arXiv:1511.05042*, 2015.

[19] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*, San Diego, CA, USA, Feb. 8–11, 2015. The Internet Society.

[20] J. A. Dunnmon, D. Yi, C. P. Langlotz, C. Ré, D. L. Rubin, and M. P. Lungren. Assessment of convolutional neural networks for automated classification of chest radiographs. *Radiology*, 290(2):537–544, 2018.

[21] H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. Use your brain! arithmetic 3pc for any modulus with active security. Cryptology ePrint Archive, Report 2019/164, 2019. https://eprint.iacr.org/2019/164.

[22] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[23] E. Fiesler, A. Choudry, and H. J. Caulfield. Weight discretization paradigm for optical neural networks. In *Optical interconnections and networks*, volume 1281, pages 164–174. International Society for Optics and Photonics, 1990.

[24] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 201–210, 2016.

[25] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, 1964.

[26] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[27] Google. Quantization-aware training. https://github.com/tensorflow/tensorflow/tree/r1.13/tensorflow/contrib/quantize, 2019.

[28] Google. Tensorflow lite. https://www.tensorflow.org/lite/, 2019.

[29] Y. Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[30] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[32] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[34] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[35] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[36] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal

covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[37] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017.

[38] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1651–1669, 2018.

[39] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In W. Enck and A. P. Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 1651–1669, Baltimore, MD, USA, Aug. 15–17, 2018. USENIX Association.

[40] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, Oct. 24–28, 2016. ACM Press.

[41] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, Apr. 29 – May 3, 2018. Springer, Heidelberg, Germany.

[42] R. Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[44] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. Cryptflow: Secure tensorflow inference. Cryptology ePrint Archive, Report 2019/1049, 2019. https://eprint.iacr.org/2019/1049.

[45] S. Lawrence, C. Lee Giles, A. Chung Tsoi, and A. Back. Face recognition: A convolutional neural network approach. *Neural Networks, IEEE Transactions on*, 8:98 – 113, 02 1997.

[46] Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 259–276, Dallas, TX, USA, Oct. 31 – Nov. 2, 2017. ACM Press.

[47] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 619–631, Dallas, TX, USA, Oct. 31 – Nov. 2, 2017. ACM Press.

[48] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini. Fast neural networks without multipliers. *IEEE transactions on Neural Networks*, 4(1):53–62, 1993.

[49] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, Oct. 15–19, 2018. ACM Press.

[50] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[51] C. Orlandi, A. Piva, and M. Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007(1):037343, 2007.

[52] E. Park, J. Ahn, and S. Yoo. Weighted-entropy-based quantization for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7197–7205. IEEE, 2017.

[53] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, pages 525–542, 2016.

[54] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. In *USENIX Security*, August 2019.

[55] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, editors, *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*, pages 707–721, Incheon, Republic of Korea, Apr. 2–6, 2018. ACM Press.

[56] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. Deepsecure: scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 2:1–2:6, 2018.

[57] A. Sanyal, M. J. Kusner, A. Gascón, and V. Kanade. Tapas: Tricks to accelerate (encrypted) prediction as a service. *arXiv preprint arXiv:1806.03461*, 2018.

[58] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[59] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[60] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabi-

novich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[61] C. Z. Tang and H. K. Kwan. Multilayer feedforward neural networks with single powers-of-two weights. *IEEE Transactions on Signal Processing*, 41(8):2724–2727, 1993.

[62] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016.

[63] S. Wagh, D. Gupta, and N. Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, 2019.

[64] B. Wang and N. Z. Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy*, pages 36–52, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[65] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7686–7695, 2018.

[66] S. Wu, G. Li, F. Chen, and L. Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[67] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.

[68] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.