XONN: XNOR-based Oblivious Deep Neural Network Inference

M. Sadegh Riazi
UC San Diego

Mohammad Samragh
UC San Diego

Hao Chen
Microsoft Research

Kim Laine
Microsoft Research

Kristin Lauter
Microsoft Research

Farinaz Koushanfar
UC San Diego

Abstract

Advancements in deep learning enable cloud servers to provide inference-as-a-service for clients. In this scenario, clients send their raw data to the server to run the deep learning model and send back the results. One standing challenge in this setting is to ensure the privacy of the clients' sensitive data. Oblivious inference is the task of running the neural network on the client's input without disclosing the input or the result to the server. This paper introduces XONN (pronounced /zʌn/), a novel end-to-end framework based on Yao's Garbled Circuits (GC) protocol, that provides a paradigm shift in the conceptual and practical realization of oblivious inference. In XONN, the costly matrix-multiplication operations of the deep learning model are replaced with XNOR operations that are essentially free in GC. We further provide a novel algorithm that customizes the neural network such that the runtime of the GC protocol is minimized without sacrificing the inference accuracy.

We design a user-friendly high-level API for XONN, allowing expression of the deep learning model architecture in an unprecedented level of abstraction. We further provide a compiler to translate the model description from high-level Python (i.e., Keras) to that of XONN. Extensive proof-of-concept evaluation on various neural network architectures demonstrates that XONN outperforms prior art such as Gazelle (USENIX Security'18) by up to 7×, MiniONN (ACM CCS'17) by 93×, and SecureML (IEEE S&P'17) by 37×. State-of-the-art frameworks require one round of interaction between the client and the server for each layer of the neural network, whereas, XONN requires a *constant* round of interactions for *any* number of layers in the model. XONN is first to perform oblivious inference on Fitnet architectures with up to 21 layers, suggesting a new level of scalability compared with state-of-the-art. Moreover, we evaluate XONN on four datasets to perform privacy-preserving medical diagnosis. The datasets include breast cancer, diabetes, liver disease, and Malaria.

1 Introduction

The advent of big data and striking recent progress in artificial intelligence are fueling the impending industrial automation revolution. In particular, Deep Learning (DL) —a method based on learning Deep Neural Networks (DNNs) —is demonstrating a breakthrough in accuracy. DL models outperform human cognition in a number of critical tasks such as speech and visual recognition, natural language processing, and medical data analysis. Given DL's superior performance, several technology companies are now developing or already providing DL as a service. They train their DL models on a large amount of (often) proprietary data on their own servers; then, an inference API is provided to the users who can send their data to the server and receive the analysis results on their queries. The notable shortcoming of this remote inference service is that the inputs are revealed to the cloud server, breaching the privacy of sensitive user data.

Consider a DL model used in a medical task in which a health service provider withholds the prediction model. Patients submit their plaintext medical information to the server, which then uses the sensitive data to provide a medical diagnosis based on inference obtained from its proprietary model. A naive solution to ensure patient privacy is to allow the patients to receive the DL model and run it on their own trusted platform. However, this solution is not practical in real-world scenarios because: (i) The DL model is considered an essential component of the service provider's intellectual property (IP). Companies invest a significant amount of resources and funding to gather the massive datasets and train the DL models; hence, it is important to service providers not to reveal the DL model to ensure their profitability and competitive advantage. (ii) The DL model is known to reveal information about the underlying data used for training [1]. In the case of medical data, this reveals sensitive information about other patients, violating HIPAA and similar patient health privacy regulations.

Oblivious inference is the task of running the DL model on the client's input without disclosing the input or the result to the server itself. Several solutions for oblivious inference have been proposed that utilize one or more cryptographic tools such as Homomorphic Encryption (HE) [2, 3], Garbled Circuits (GC) [4], Goldreich-Micali-Wigderson (GMW) protocol [5], and Secret Sharing (SS). Each of these cryptographic tools offer their own characteristics and tradeoffs. For example, one major drawback of HE is its *computational complexity*. HE has two main variants: Fully Homomorphic Encryption (FHE) [2] and Partially Homomorphic Encryption (PHE) [3, 6]. FHE allows computation on encrypted data but is computationally very expensive. PHE has less overhead but only supports a subset of functions or depth-bounded arithmetic circuits. The computational complexity drastically increases with the circuit's depth. Moreover, non-polynomial functionalities such as the ReLU activation function in DL cannot be supported.

GC, on the other hand, can support an arbitrary functionality while requiring only a *constant* round of interactions regardless of the depth of the computation. However, it has a high communication cost and a significant overhead for multiplication. More precisely, performing multiplication in GC has quadratic computation and communication complexity with respect to the bit-length of the input operands. It is well-known that the complexity of the contemporary DL methodologies is dominated by matrix-vector multiplications. GMW needs less communication than GC but requires many rounds of *interactions* between the two parties.

A standalone SS-based scheme provides a computationally inexpensive multiplication yet requires three or more independent (non-colluding) computing servers, which is a strong assumption. Mixed-protocol solutions have been proposed with the aim of utilizing the best characteristics of each of these protocols [7, 8, 9, 10]. They require secure conversion of secrets from one protocol to another in the middle of execution. Nevertheless, it has been shown that the cost of secret conversion is paid off in these hybrid solutions. Roughly speaking, the number of interactions between server and client (i.e., round complexity) in existing hybrid solutions is *linear* with respect to the depth of the DL model. Since depth is a major contributor to the deep learning accuracy [11], scalability of the mixed-protocol solutions with respect to the number of layers remains an unsolved issue for more complex, many-layer networks.

This paper introduces XONN, a novel end-to-end framework which provides a paradigm shift in the conceptual and practical realization of privacy-preserving interference on deep neural networks. The existing work has largely focused on the development of customized security protocols while using conventional fixed-point deep learning algorithms. XONN, for the first time, suggests leveraging the concept of the Binary Neural Networks (BNNs) in conjunction with the GC protocol. In BNNs, the weights and activations are restricted to binary (i.e, ±1) values, substituting the costly multiplications with simple XNOR operations during the inference phase. The XNOR operation is known to be *free* in the GC protocol [12]; therefore, performing oblivious inference on BNNs using GC results in the removal of costly multiplications. Using our approach, we show that oblivious inference on the standard DL benchmarks can be performed with minimal, if any, decrease in the prediction accuracy.

We emphasize that an effective solution for oblivious inference should take into account the deep learning algorithms and optimization methods that can tailor the DL model for the security protocol. Current DL models are designed to run on CPU/GPU platforms where many multiplications can be performed with high throughput, whereas, bit-level operations are very inefficient. In the GC protocol, however, bit-level operations are inexpensive, but multiplications are rather costly. As such, we propose to train deep neural networks that involve many bit-level operations but *no* multiplications in the inference phase; using the idea of learning binary networks, we achieve an average of 21× reduction in the number of gates for the GC protocol.

We perform extensive evaluations on different datasets. Compared to the Gazelle [10] (the prior best solution) and MiniONN [9] frameworks, we achieve 7× and 93× lower inference latency, respectively. XONN outperforms DeepSecure [13] (prior best GC-based framework) by 60× and CryptoNets [14], an HE-based framework, by 1859×. Moreover, our solution renders a *constant* round of interactions between the client and the server, which has a significant effect on the performance on oblivious inference in Internet settings. We highlight our contributions as follows:

- Introduction of XONN, the *first* framework for privacy preserving DNN inference with a *constant* round complexity that does not need expensive matrix multiplications. Our solution is the first that can be scalably adapted to ensure security against malicious adversaries.

- Proposing a novel conditional addition protocol based on Oblivious Transfer (OT) [15], which optimizes the costly computations for the network's input layer. Our protocol is 6× faster than GC and can be of independent interest. We also devise a novel network trimming algorithm to remove neurons from DNNs that minimally contribute to the inference accuracy, further reducing the GC complexity.

- Designing a high-level API to readily automate fast adaptation of XONN, such that users only input a high-level description of the neural network. We further facilitate the usage of our framework by designing a compiler that translates the network description from Keras to XONN.

- Proof-of-concept implementation of XONN and evaluation on various standard deep learning benchmarks. To demonstrate the scalability of XONN, we perform oblivious inference on neural networks with as many as 21 layers for the first time in the oblivious inference literature.

2 Preliminaries

Throughout this paper, scalars are represented as lowercase letters ($x \in \mathbb{R}$), vectors are represented as bold lowercase letters ($\mathbf{x} \in \mathbb{R}^n$), matrices are denoted as capital letters ($X \in \mathbb{R}^{m \times n}$), and tensors of more than 2 ways are shown using bold capital letters ($\mathbf{X} \in \mathbb{R}^{m \times n \times k}$). Brackets denote element selection and the colon symbol stands for all elements —$W[i,:]$ represents all values in the i-th row of W.

2.1 Deep Neural Networks

The computational flow of a deep neural network is composed of multiple computational layers. The input to each layer is either a vector (i.e., $\mathbf{x} \in \mathbb{R}^n$) or a tensor (i.e., $\mathbf{X} \in \mathbb{R}^{m \times n \times k}$). The output of each layer serves as the input of the next layer. The input of the first layer is the raw data and the output of the last layer represents the network's prediction on the given data (i.e., inference result). In an image classification task, for instance, the raw image serves as the input to the first layer and the output of the last layer is a vector whose elements represent the probability that the image belongs to each category. Below we describe the functionality of neural network layers.

Linear Layers: Linear operations in neural networks are performed in Fully-Connected (FC) and Convolution (CONV) layers. The vector dot product (VDP) between two vectors $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^n$ is defined as follows:

$$\mathsf{VDP}\ (\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n} \mathbf{w}[i] \cdot \mathbf{x}[i]. \qquad (1)$$

Both CONV and FC layers repeat VDP computation to generate outputs as we describe next. A fully connected layer takes a vector $\mathbf{x} \in \mathbb{R}^n$ and generates the output $\mathbf{y} \in \mathbb{R}^m$ using a linear transformation:

$$\mathbf{y} = W \cdot \mathbf{x} + \mathbf{b}, \qquad (2)$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^m$ is a bias vector. More precisely, the i-th output element is computed as $\mathbf{y}[\mathbf{i}] = \mathsf{VDP}\ (W[i,:], \mathbf{x}) + \mathbf{b}[i]$.

A convolution layer is another form of linear transformation that operates on images. The input of a CONV layer is represented as multiple rectangular channels (2D images) of the same size: $\mathbf{X} \in \mathbb{R}^{h1 \times h2 \times c}$, where $h1$ and $h2$ are the dimensions of the image and c is the number of channels. The CONV layer maps the input image into an output image $\mathbf{Y} \in \mathbb{R}^{h1' \times h2' \times f}$. A CONV layer consists of a weight tensor $\mathbf{W} \in \mathbb{R}^{k \times k \times c \times f}$ and a bias vector $\mathbf{b} \in \mathbb{R}^f$. The i-th output channel in a CONV layer is computed by sliding the kernel $\mathbf{W}[:,:,:,i] \in \mathbb{R}^{k \times k \times c}$ over the input, computing the dot product between the kernel and the windowed input, and adding the bias term $\mathbf{b}[i]$ to the result.

Non-linear Activations: The output of linear transformations (i.e., CONV and FC) is usually fed to an activation layer, which applies an element-wise non-linear transformation to the vector/tensor and generates an output with the same dimensionality. In this paper, we particularly utilize the Binary Activation (BA) function for hidden layers. BA maps the input operand to its sign value (i.e., $+1$ or -1).

Batch Normalization: A batch normalization (BN) layer is typically applied to the output of linear layers to normalize the results. If a BN layer is applied to the output of a CONV layer, it multiplies all of the i-th channel's elements by a scalar $\boldsymbol{\gamma}[i]$ and adds a bias term $\boldsymbol{\beta}[i]$ to the resulting channel. If BN is applied to the output of an FC layer, it multiplies the i-th element of the vector by a scalar $\boldsymbol{\gamma}[i]$ and adds a bias term $\boldsymbol{\beta}[i]$ to the result.

Pooling: Pooling layers operate on image channels outputted by the CONV layers. A pooling layer slides a window on the image channels and aggregates the elements within the window into a single output element. Max-pooling and Average-pooling are two of the most common pooling operations in neural networks. Typically, pooling layers reduce the image size but do not affect the number of channels.

2.2 Secret Sharing

A secret can be securely shared among two or multiple parties using Secret Sharing (SS) schemes. An SS scheme guarantees that each share does not reveal any information about the secret. The secret can be reconstructed using all (or subset) of shares. In XONN, we use additive secret sharing in which a secret S is shared among two parties by sampling a random number $\widehat{S}_1 \in_R \mathbb{Z}_{2^b}$ (integers modulo 2^b) as the first share and creating the second share as $\widehat{S}_2 = S - \widehat{S}_1\ mod\ 2^b$ where b is the number of bits to describe the secret. While none of the shares reveal any information about the secret S, they can be used to reconstruct the secret as $S = \widehat{S}_1 + \widehat{S}_2\ mod\ 2^b$. Suppose that two secrets $S^{(1)}$ and $S^{(2)}$ are shared among two parties where party-1 has $\widehat{S}_1^{(1)}$ and $\widehat{S}_1^{(2)}$ and party-2 has $\widehat{S}_2^{(1)}$ and $\widehat{S}_2^{(2)}$. Party-i can create a share of the sum of two secrets as $\widehat{S}_i^{(1)} + \widehat{S}_i^{(2)}\ mod\ 2^b$ without communicating to the other party. This can be generalized for arbitrary (more than two) number of secrets as well. We utilize additive secret sharing in our Oblivious Conditional Addition (OCA) protocol (Section 3.3).

2.3 Oblivious Transfer

One of the most crucial building blocks of secure computation protocols, e.g., GC, is the Oblivious Transfer (OT) protocol [15]. In OT, two parties are involved: a sender and a receiver. The sender holds n different messages $m_j,\ j = 1...n$, with a specific bit-length and the receiver holds an index (ind) of a message that she wants to receive. At the end of the protocol, the receiver gets m_{ind} with no additional knowledge about the other messages and the sender learns nothing about the selection index. In GC, 1-out-of-2 OT is used where $n = 2$ in which case the selection index is only one bit. The initial realizations of OT required costly public key encryptions for each run of the protocol. However, the OT Extension [16, 17, 18] technique enables performing OT using more efficient symmetric-key encryption in conjunction with a *fixed* number of base OTs that need public-key encryption. OT is used both in the OCA protocol as well as the Garbled Circuits protocol which we discuss next.

2.4 Garbled Circuits

Yao's Garbled Circuits [4], or GC in short, is one of the generic two-party secure computation protocols. In GC, the result of an arbitrary function $f(.)$ on inputs from two parties can be computed without revealing each party's input to the other. Before executing the protocol, function $f(.)$ has to be described as a Boolean circuit with two-input gates.

GC has three main phases: garbling, transferring data, and evaluation. In the first phase, only one party, the Garbler, is involved. The Garbler starts by assigning two randomly generated l-bit binary strings to each wire in the circuit. These binary strings are called *labels* and they represent semantic values 0 and 1. Let us denote the label of wire w corresponding to the semantic value x as L_x^w. For each gate in the circuit, the Garbler creates a four-row garbled table as follows. Each label of the output wire is encrypted using the input labels according to the truth table of the gate. For example, consider an AND gate with input wires a and b and output wire c. The last row of the garbled table is the encryption of L_1^c using labels L_1^a and L_1^b.

Once the garbling process is finished, the Garbler sends all of the garbled tables to the Evaluator. Moreover, he sends the correct labels that correspond to input wires that represent his inputs to the circuit. For example, if wire w^* is the first input bit of the Garbler and his input is 0, he sends L_0^*. The Evaluator acquires the labels corresponding to her input through 1-out-of-2 OT where Garbler is the sender with two labels as his messages and the Evaluator's selection bit is her input for that wire. Having all of the garbled tables and labels of input wires, the Evaluator can start decrypting the garbled tables one by one until reaching the final output bits. She then learns the plaintext result at the end of the GC protocol based on the output labels and their relationships to the semantic values that are received from the Garbler.

3 The XONN Framework

In this section, we explain how neural networks can be trained such that they incur a minimal cost during the oblivious inference. The most computationally intensive operation in a neural network is matrix multiplication. In GC, each multiplication has a *quadratic* computation and communication cost with respect to the input bit-length. This is the major source of inefficiency in prior work [13]. We overcome this limitation by changing the learning process such that the trained neural network's weights become binary. As a result, costly multiplication operations are replaced with XNOR gates which are essentially free in GC. We describe the training process in Section 3.1. In Section 3.2, we explain the operations and their corresponding Boolean circuit designs that enable a very fast oblivious inference. In Section 4, we elaborate on XONN implementation.

3.1 Customized Network Binarization

Numerical optimization algorithms minimize a specific cost function associated with neural networks. It is well-known that neural network training is a non-convex optimization, meaning that there exist many locally-optimum parameter configurations that result in similar inference accuracies. Among these parameter settings, there exist solutions where both neural network parameters and activation units are restricted to take binary values (i.e., either $+1$ or -1); these solutions are known as Binary Neural Netowrks (BNNs) [19].

One major shortcoming of BNNs is their (often) low inference accuracy. In the machine learning community, several methods have been proposed to modify BNN functionality for accuracy enhancement [20, 21, 22]. These methods are devised for *plaintext* execution of BNNs and are not efficient for oblivious inference with GC. We emphasize that, when modifying BNNs for accuracy enhancement, one should also take into account the implications in the corresponding GC circuit. With this in mind, we propose to modify the number of channels and neurons in CONV and FC layers, respectively. Increasing the number of channels/neurons leads to a higher accuracy but it also increases the complexity of the corresponding GC circuit. As a result, XONN provides a trade-off between the accuracy and the communication/runtime of the oblivious inference. This tradeoff enables cloud servers to customize the complexity of the GC protocol to optimally match the computation and communication requirements of the clients. To customize the BNN, XONN configures the per-layer number of neurons in two steps:

- Linear Scaling: Prior to training, we scale the number of channels/neurons in all BNN layers with the same factor (s), e.g., $s = 2$. Then, we train the scaled BNN architecture.
- Network Trimming: Once the (uniformly) scaled network is trained, a post-processing algorithm removes redundant channels/neurons from each hidden layer to reduce the GC cost while maintaining the inference accuracy.

Figure 1 illustrates the BNN customization method for an example baseline network with four hidden layers. Network trimming (pruning) consists of two steps, namely, Feature Ranking and Iterative Pruning which we describe next.

Feature Ranking: In order to perform network trimming, one needs to sort the channels/neurons of each layer based on their contribution to the inference accuracy. In conventional neural networks, simple ranking methods sort features based

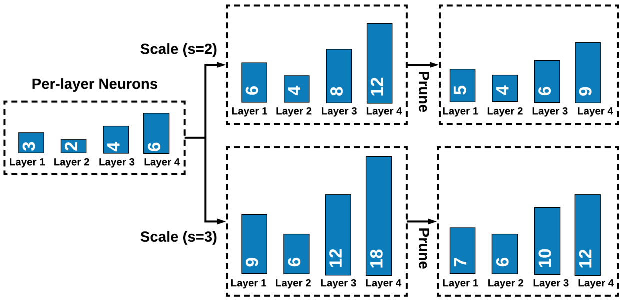


Figure 1: Illustration of BNN customization. The bars represent the number of neurons in each hidden layer.

on absolute value of the neurons/channels [23]. In BNNs, however, the weights/features are either $+1$ or -1 and the absolute value is not informative. To overcome this issue, we utilize first order Taylor approximation of neural networks and sort the features based on the magnitude of the gradient values [24]. Intuitively, the gradient with respect to a certain feature determines its importance; a high (absolute) gradient indicates that removing the neuron has a destructive effect on the inference accuracy. Inspired by this notion, we develop a feature ranking method described in Algorithm 1.

Iterative Pruning: We devise a step-by-step algorithm for model pruning which is summarized in Algorithm 2. At each step, the algorithm selects one of the BNN layers l^* and removes the first p^* features with the lowest importance (line 17). The selected layer l^* and the number of pruned neurons p^* maximize the following reward (line 15):

$$reward(l,p) = \frac{c_{curr} - c_{next}}{e^{a_{curr} - a_{next}}}, \qquad (3)$$

where c_{curr} and c_{next} are the GC complexity of the BNN before and after pruning, whereas, a_{curr} and a_{next} denote the corresponding validation accuracies. The numerator of this reward encourages higher reduction in the GC cost while the denominator penalizes accuracy loss. Once the layer is pruned, the BNN is fine-tuned to recover the accuracy (line 18). The pruning process stops once the accuracy drops below a pre-defined threshold.

3.2 Oblivious Inference

BNNs are trained such that the weights and activations are binarized, i.e., they can only have two possible values: $+1$ or -1. This property allows BNN layers to be rendered using a simplified arithmetic. In this section, we describe the functionality of different layer types in BNNs and their Boolean circuit translations. Below, we explain each layer type.

Binary Linear Layer: Most of the computational complexity of neural networks is due to the linear operations in CONV and FC layers. As we discuss in Section 2.1, linear operations are realized using vector dot product (VDP). In BNNs, VDP operations can be implemented using simplified circuits. We categorize the VDP operations of this work into two classes: (i) Integer-VDP where only one of the vectors is binarized and the other has integer elements and (ii) Binary-VDP where both vectors have binary (± 1) values.

Algorithm 1 XONN Channel Sorting for CONV Layers

Inputs: Trained BNN with loss function $\mathscr{L}$, CONV layer l with output shape of $h1 \times h2 \times f$, subsampled validation data and labels $\{(\mathbf{X_1}, z_1), \ldots, (\mathbf{X_k}, z_k)\}$
Output: Indices of the sorted channels: $\{i_0, \ldots, i_f\}$

```
G ← zeros(k × h1 × h2 × f)                    ▷ define gradient tensor
for $i = 1, \ldots, k$ do
    $\mathscr{L} = \mathscr{L}(\mathbf{X_i}, z_i)$                       ▷ evaluate loss function
    $\nabla_Y = \frac{\partial \mathscr{L}}{\partial Y^l}$          ▷ compute gradient w.r.t. layer output
    $\mathbf{G}[i,:,:,:] \leftarrow \nabla_Y$                          ▷ store gradient
end for
$\mathbf{G_{abs}} \leftarrow |\mathbf{G}|$                 ▷ take elementwise absolute values
$\mathbf{g_s} \leftarrow zeros(f)$                   ▷ define sum of absolute values
for $i = 1, \ldots, f$ do
    $\mathbf{g_s}[i] \leftarrow sum(\mathbf{G_{abs}}[:,:,:,i])$
end for
$\{i_0, \ldots, i_f\} \leftarrow sort(\mathbf{g_s})$
return $\{i_0, \ldots, i_f\}$
```

Integer-VDP: For the first layer of the neural network, the server has no control over the input data which is not necessarily binarized. The server can only train binary weights and use them for oblivious inference. Consider an input vector $\mathbf{x} \in \mathbb{R}^n$ with integer (possibly fixed-point) elements and a weight vector $\mathbf{w} \in \{-1, 1\}^n$ with binary values. Since the elements of the binary vector can only take $+1$ or -1, the Integer-VDP can be rendered using additions and subtractions. In particular, the binary weights can be used in a selection circuit that decides whether the pertinent integer input should be added to or subtracted from the VDP result.

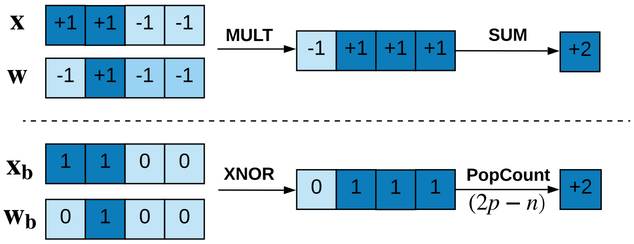


Figure 2: Equivalence of Binary-VDP and **XnorPopcount**.

Binary-VDP: Consider a dot product between two binary vectors $\mathbf{x} \in \{-1, +1\}^n$ and $\mathbf{w} \in \{-1, +1\}^n$. If we encode each element with one bit (i.e., $-1 \rightarrow 0$ and $+1 \rightarrow 1$), we obtain binary vectors $\mathbf{x}_b \in \{0, 1\}^n$ and $\mathbf{w}_b \in \{0, 1\}^n$. It has been shown that the dot product of $\mathbf{x}$ and $\mathbf{w}$ can be efficiently computed using an *XnorPopcount* operation [19]. Figure 2 depicts the equivalence of VDP$(\mathbf{x}, \mathbf{w})$ and

Algorithm 2 XONN Iterative BNN Pruning

Inputs: Trained BNN with n overall CONV and FC layers, minimum accuracy threshold θ, number of pruning trials per layer t, subsampled validation data and labels $data_V$, training data and labels $data_T$
Output: BNN with pruned layers

```
$\mathbf{p} \leftarrow zeros(n-1)$                                  ▷ current number of pruned neurons/channels per layer
$a_{curr} \leftarrow Accuracy(BNN, data_V|\mathbf{p})$              ▷ current BNN validation accuracy
$c_{curr} \leftarrow Cost(BNN|\mathbf{p})$                          ▷ current GC cost
while $a_{curr} > \theta$ do                                        ▷ repeat until accuracy drops below $\theta$
    for $l = 1, \dots, n-1$ do                                      ▷ search over all layers
        $inds \leftarrow Rank(BNN, l, data_V)$                      ▷ rank features via Algorithm 1
        $f \leftarrow$ Number of neurons/channels                    ▷ number of output neurons/channels
        for $p = \mathbf{p}[l], \mathbf{p}[l] + \frac{f}{t}, \dots, f$ do   ▷ search over possible pruning rates
            $BNN_{next} \leftarrow Prune(BNN, l, p, inds)$          ▷ prune $p$ features with lowest ranks from the $l$-th layer
            $a_{next} \leftarrow Accuracy(BNN_{next}, data_V|\mathbf{p}[\mathbf{1}], \dots, \mathbf{p}[\mathbf{l}] = p, \dots, \mathbf{p}[\mathbf{n-1}])$   ▷ validation accuracy if pruned
            $c_{next} \leftarrow Cost(BNN_{next}|\mathbf{p}[\mathbf{1}], \dots, \mathbf{p}[\mathbf{l}] = p, \dots, \mathbf{p}[\mathbf{n-1}])$   ▷ GC cost if pruned
            $reward(l,p) = \frac{c_{curr} - c_{next}}{e^{(a_{curr} - a_{next})}}$   ▷ compute reward given that $p$ features are pruned from layer $l$
        end for
    end for
    $\{l^*, p^*\} \leftarrow \arg\max_{l,p} reward(l,p)$            ▷ select layer $l^*$ and pruning rate $p^*$ that maximize the reward
    $\mathbf{p}[\mathbf{l}^*] \leftarrow p^*$                        ▷ update the number of pruned features in vector $\mathbf{p}$
    $BNN \leftarrow Prune(BNN, l^*, p^*, inds)$                     ▷ prune $p^*$ features with lowest ranks from the $l^*$-th layer
    $BNN \leftarrow Fine\text{-}tune(BNN, data_T)$                  ▷ fine-tune the pruned model using training data to recover accuracy
    $a_{curr} \leftarrow Accuracy(BNN, data_V|\mathbf{p})$          ▷ update current BNN validation accuracy
    $c_{curr} \leftarrow Cost(BNN|\mathbf{p})$                      ▷ update current GC cost
end while
return $BNN$
```

$XnorPopcount(\mathbf{x}_b, \mathbf{w}_b)$ for a VDP between 4-dimensional vectors. First, element-wise XNOR operations are performed between the two binary encodings. Next, the number of set bits p is counted, and the output is computed as $2p - n$.

Binary Activation Function: A Binary Activation (BA) function takes input x and maps it to $y = Sign(x)$ where $Sign(\cdot)$ outputs either $+1$ or -1 based on the sign of its input. This functionality can simply be implemented by extracting the most significant bit of x.

Binary Batch Normalization: in BNNs, it is often useful to normalize feature x using a Batch Normalization (BN) layer before applying the binary activation function. More specifically, a BN layer followed by a BA is equivalent to:

$$y = Sign(\gamma \cdot x + \beta) = Sign(x + \frac{\beta}{\gamma}),$$

since γ is a positive value. The combination of the two layers (BN+BA) is realized by a comparison between x and $-\frac{\beta}{\gamma}$.

Binary Max-Pooling: Assuming the inputs to the max-pooling layers are binarized, taking the maximum in a window is equivalent to performing logical OR over the binary encodings as depicted in Figure 3. Note that average-pooling layers are usually not used in BNNs since the average of multiple binary elements is no longer a binary value.

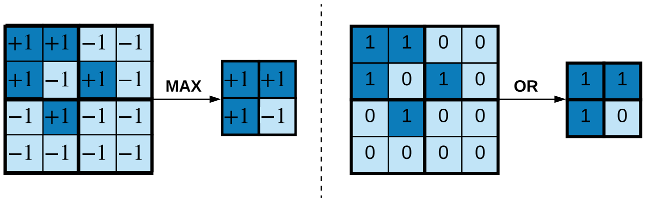


Figure 3: The equivalence between Max-Pooling and Boolean-OR operations in BNNs.

Figure 4 demonstrates the Boolean circuit for Binary-VDP followed by BN and BA. The number of non-XOR gates for binary-VDP is equal to the number of gates required to render the tree-adder structure in Figure 4. Similarly, Figure 5 shows the Integer-VDP counterpart. In the first level of the tree-adder of Integer-VDP (Figure 5), the binary weights determine whether the integer input should be added to or subtracted from the final result within the "Select" circuit. The next levels of the tree-adder compute the result of the integer-VDP using "Adder" blocks. The combination of BN and BA is implemented using a single *comparator*. Compared to Binary-VDP, Integer-VDP has a high garbling cost which is linear with respect to the number of bits. To mitigate this problem, we propose an alternative solution based on Oblivious Transfer (OT) in Section 3.3.

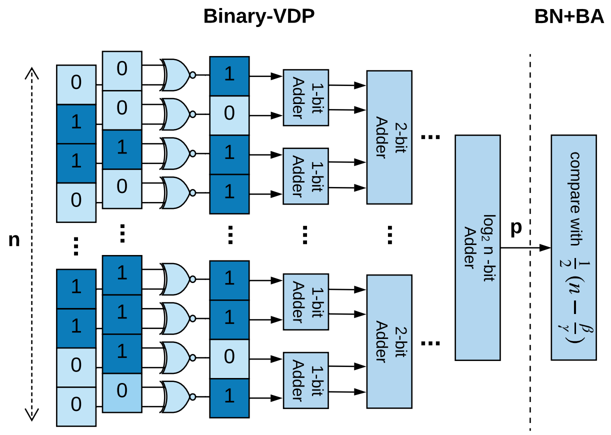


Figure 4: Circuit for binary-VDP followed by comparison for batch normalization (BN) and binary activation (BA).

3.3 Oblivious Conditional Addition Protocol

In XONN, all of the activation values as well as neural network weights are binary. However, the input to the neural network is provided by the user and is not necessarily binary. The first layer of a typical neural network comprises either an FC or a CONV layer, both of which are evaluated using oblivious Integer-VDP. On the one side, the user provides her input as non-binary (integer) values. On the other side, the network parameters are binary values representing -1 and 1. We now demonstrate how Integer-VDP can be described as an OT problem. Let us denote the user's input as a vector $\mathbf{v_1}$ of n (b-bit) integers. The server holds a vector of n binary values denoted by $\mathbf{v_2}$. The result of Integer-VDP is a number "y" that can be described with

$$b' = \left\lceil \log_2(n \cdot (2^b - 1)) \right\rceil$$

bits. Figure 6 summarizes the steps in the OCA protocol. The first step is to *bit-extend* $\mathbf{v_1}$ from b-bit to b'-bit. In other words, if $\mathbf{v_1}$ is a vector of *signed* integer/fixed-point numbers, the most significant bit should be repeated $(b'-b)$-many times, otherwise, it has to be zero-padded for most significant bits. We denote the bit-extended vector by $\mathbf{v_1^*}$. The second step is to create the two's complement vector of $\mathbf{v_1^*}$, called $\overline{\mathbf{v_1^*}}$. The client also creates a vector of n (b'-bit) randomly generated numbers, denoted as $\mathbf{r}$. She computes element-wise vector subtractions $\mathbf{v_1^*} - \mathbf{r}\ mod\ 2^{b'}$ and $\overline{\mathbf{v_1^*}} - \mathbf{r}\ mod\ 2^{b'}$. These two vectors are n-many pair of messages that will be used as input to n-many 1-out-of-two OTs. More precisely, $\overline{\mathbf{v_1^*}} - \mathbf{r}\ mod\ 2^{b'}$ is a list of first messages and $\mathbf{v_1^*} - \mathbf{r}\ mod\ 2^{b'}$ is a list of second messages. The server's list of selection bits is $\mathbf{v_2}$. After n-many OTs are finished, the server has a list of n transferred numbers called $\mathbf{v_t}$ where

$$\mathbf{v_t}[i] = \begin{cases} \overline{\mathbf{v_1^*}}[i] - \mathbf{r}[i]\ mod\ 2^{b'} & if\ \ \mathbf{v_2}[i] = 0 \\ \mathbf{v_1^*}[i] - \mathbf{r}[i]\ mod\ 2^{b'} & if\ \ \mathbf{v_2}[i] = 1 \end{cases} \quad i = 1,\ \dots,\ n.$$

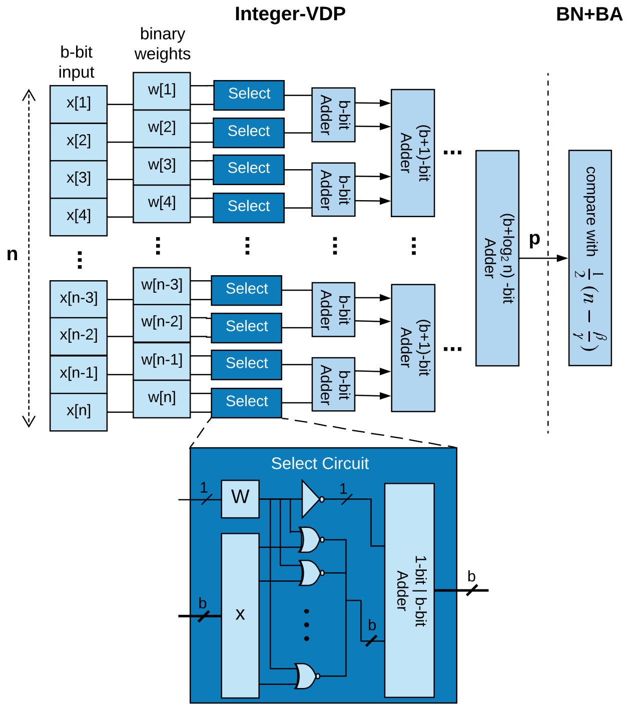


Figure 5: Circuit for Integer-VDP followed by comparison for batch normalization (BN) and binary activation (BN).

Finally, the client computes $y_1 = \sum_{i=1}^{n} \mathbf{r}[i]\ mod\ 2^{b'}$ and the server computes $y_2 = \sum_{i=1}^{n} \mathbf{v_t}[i]\ mod\ 2^{b'}$. By OT's definition, the receiver (server) gets only one of the two messages from the sender. That is, based on each selection bit (a binary weight), the receiver gets an additive share of either the sender's number or its two's complement. Upon adding all of the received numbers, the receiver computes an additive share of the Integer-VDP result. Now, even though the sender does not know which messages were selected by the receiver, she can add all of the randomly generated numbers $\mathbf{r}[i]$s which is equal to the other additive share of the Integer-VDP result. Since all numbers are described in the two's complement format, subtractions are equivalent to the addition of the two's complement values, which are created by the sender at the beginning of OCA. Moreover, it is possible that as we accumulate the values, the bit-length of the final Integer-VDP result grows accordingly. This is supported due to the bit-extension process at the beginning of the protocol. In other words, all additions are performed in a larger ring such that the result does not overflow.

Note that all numbers belong to the ring $\mathbb{Z}_{2^{b'}}$ and by definition, a ring is closed under addition, therefore, y_1 and y_2 are true additive shares of $y = y_1 + y_2\ mod\ 2^{b'}$. We described the OCA protocol for one Integer-VDP computation. As we outlined in Section 3.2, all linear operations in the first layer of the DL model (either FC or CONV) can be formulated as a series of Integer-VDPs.

```
Sender:
  (1) Bit-extends all elements of $\mathbf{v_1}$ and creates $\mathbf{v_1^*}$
  (2) Creates two's complement of $\mathbf{v_1^*}$ : $\overline{\mathbf{v_1^*}}$
  (3) Creates random vector $\mathbf{r}$ : same size as $\overline{\mathbf{v_1^*}}$
  (4) Creates list of first messages as $\mathbf{m^2} = \overline{\mathbf{v_1^*}} - \mathbf{r}\ mod\ 2^{b'}$
  (5) Creates list of second messages as $\mathbf{m^1} = \mathbf{v_1^*} - \mathbf{r}\ mod\ 2^{b'}$
Sender & Receiver:
  (6) Parties engage in Oblivious Transfer (OT)
      Sender puts $\mathbf{m^1}$ and $\mathbf{m^2}$ as message vectors
      Receiver puts $\mathbf{v_2}$ vector as selection bits
Receiver:
  (7) Gets vector $\mathbf{v_t}$ where:
      $\mathbf{v_t}[i] = \begin{cases} \overline{\mathbf{v_1^*}}[i] - \mathbf{r}[i]\ mod\ 2^{b'} & (\text{if } \mathbf{v_2}[i] = 0) \\ \mathbf{v_1^*}[i] - \mathbf{r}[i]\ mod\ 2^{b'} & (\text{if } \mathbf{v_2}[i] = 1) \end{cases}$
Sender:
  (8) Computes her additive share of VDP result as:
      $y_1 = \sum_{i=1}^{n} \mathbf{r}[i]\ mod\ 2^{b'}$
Receiver:
  (9) Computes his additive share of VDP result as:
      $y_2 = \sum_{i=1}^{n} \mathbf{v_t}[i]\ mod\ 2^{b'}$
```

Figure 6: Oblivious Conditional Addition (OCA) protocol.

In traditional OT, public-key encryption is needed for each OT invocation which can be computationally expensive. Thanks to the Oblivious Transfer Extension technique [16, 17, 18], one can perform many OTs using symmetric-key encryption and only a fixed number of public-key operations.

Required Modification to the Next Layer. So far, we have shown how to perform Integer-VDP using OT. However, we need to add an "addition" layer to reconstruct the true value of y from its additive shares before further processing it. The overhead of this layer, as well as OT computations, are discussed next. Note that OCA is used only for the first layer and it does not change the overall constant round complexity of XONN since it is performed only once regardless of the number of layers in the DL model.

Comparison to Integer-VDP in GC. Table 1 shows the computation and communication costs for two approaches: (i) computing the first layer in GC and (ii) utilizing OCA. OCA removes the GC cost of the first layer in XONN. However, it adds the overhead of a set of OTs and the GC costs associated with the new ADD layer.

Table 1: Computation and communication cost of OCA.

Costs	GC	OCA	
{Sender, Receiver}		OT	ADD Layer
Comp. (AES ops)	$(n+1) \cdot b \cdot \{2, 4\}$	$n \cdot \{1, 2\}$	$b' \cdot \{2, 4\}$
Comm. (bit)	$(n+1) \cdot b \cdot 2 \cdot 128$	$n \cdot b$	$b' \cdot 2 \cdot 128$

3.4 Security of XONN

We consider the Honest-but-Curious (HbC) adversary model consistent with all of the state-of-the-art solutions for oblivious inference [7, 8, 9, 10, 13, 25]. In this model, neither of the involved parties is trusted but they are assumed to follow the protocol. Both server and client cannot infer any information about the other party's input from the entire protocol transcript. XONN relies solely on the GC and OT protocols, both of which are proven to be secure in the HbC adversary model in [26] and [15], respectively. Utilizing binary neural networks does not affect GC and OT protocols in any way. More precisely, we have changed the function $f(.)$ that is evaluated in GC such that it is more efficient for the GC protocol: drastically reducing the number of AND gates and using XOR gates instead. Our novel Oblivious Conditional Addition (OCA) protocol (Section 3.3) is also based on the OT protocol. The sender creates a list of message pairs and puts them as input to the OT protocol. Each message is an additive share of the sender's private data from which the secret data cannot be reconstructed. The receiver puts a list of selection bits as input to the OT. By OT's definition, the receiver learns nothing about the unselected messages and the sender does not learn the selection bits.

During the past few years, several attacks have been proposed that extract some information about the DL model by querying the server many times [1, 27, 28]. It has been shown that some of these attacks can be effective in the black-box setting where the client only receives the prediction results and does not have access to the model. Therefore, considering the definition of an oblivious inference, these type of attacks are out of the scope of oblivious inference frameworks. However, in Appendix A, we show how these attacks can be thwarted by adding a simple layer at the end of the neural network which adds a negligible overhead.

Security Against Malicious Adversaries. The HbC adversary model is the standard security model in the literature. However, there are more powerful security models such as security against covert and malicious adversaries. In the malicious security model, the adversary (either the client or server) can deviate from the protocol at any time with the goal of learning more about the input from the other party. One of the main distinctions between XONN and the state-of-the-art solutions is that XONN can be automatically adapted to the malicious security using cut-and-choose techniques [29, 30, 31]. These methods take a GC protocol in HbC and readily extend it to the malicious security model. This modification increases the overhead but enables a higher security level. To the best of our knowledge, there is no practical solution to extend the customized mixed-protocol frameworks [7, 9, 10, 25] to the malicious security model. Our GC-based solution is more efficient compared to the mixed-protocol solutions and can be upgraded to the malicious security at the same time.

4 The XONN Implementation

In this section, we elaborate on the garbling/evaluation implementation of XONN. All of the optimizations and techniques proposed in this section do not change the security or correctness in anyway and only enable the framework's scalability for large network architectures.

We design a new GC framework with the following design principles in mind: (i) *Efficiency:* XONN is designed to have a minimal data movement and low cache-miss rate. (ii) *Scalability:* oblivious inference inevitably requires significantly higher memory usage compared to plaintext evaluation of neural networks. High memory usage is one critical shortcoming of state-of-the-art secure computation frameworks. As we show in our experimental results, XONN is designed to scale for very deep neural networks that have higher accuracy compared to networks considered in prior art. (iii) *Modularity:* our framework enables users to create Boolean description of different layers separately. This allows the hardware synthesis tool to generate more optimized circuits as we discuss in Section 4.1. (iv) *Ease-to-use:* XONN provides a very simple API that requires few lines of neural network description. Moreover, we have created a compiler that takes a Keras description and automatically creates the network description for XONN API.

XONN is written in C++ and supports all major GC optimizations proposed previously. Since the introduction of GC, many optimizations have been proposed to reduce the computation and communication complexity of this protocol. Bellare et al. [32] have provided a way to perform garbling using efficient fixed-key AES encryption. Our implementation benefits from this optimization by using Intel AES-NI instructions. Row-reduction technique [33] reduces the number of garbled tables from four to three. Half-Gates technique [34] further reduces the number of rows in the garbled tables from three to two. One of the most influential optimizations for the GC protocol is the *free-XOR* technique [12] which makes XOR, XNOR, and NOT almost free of cost. Our implementation for Oblivious Transfer (OT) is based on libOTe [35].

4.1 Modular Circuit Synthesis and Garbling

In XONN, each layer is described as multiple invocations of a *base* circuit. For instance, linear layers (CONV and FC) are described by a VDP circuit. MaxPool is described by an OR circuit where the number of inputs is the window size of the MaxPool layer. BA/BN layers are described using a comparison (CMP) circuit. The memory footprint is significantly reduced in this approach: we only create and store the base circuits. As a result, the connection between two invocations of two different base circuits is handled at the software level.

We create the Boolean circuits using TinyGarble [36] hardware synthesis approach. TinyGarble's technology libraries are optimized for GC and produce circuits that have low number of non-XOR gates. Note that the Boolean circuit description of the contemporary neural networks comprises between millions to billions of Boolean gates, whereas, synthesis tools cannot support circuits of this size. However, due to XONN modular design, one can synthesize each base circuit separately. Thus, the bottleneck transfers from the synthesis tool's maximum number of gates to the system's memory. As such, XONN effectively scales for any neural network complexity regardless of the limitations of the synthesis tool as long as enough memory (i.e., RAM) is available. Later in this section, we discuss how to increase the scalability by dynamically managing the allocated memory.

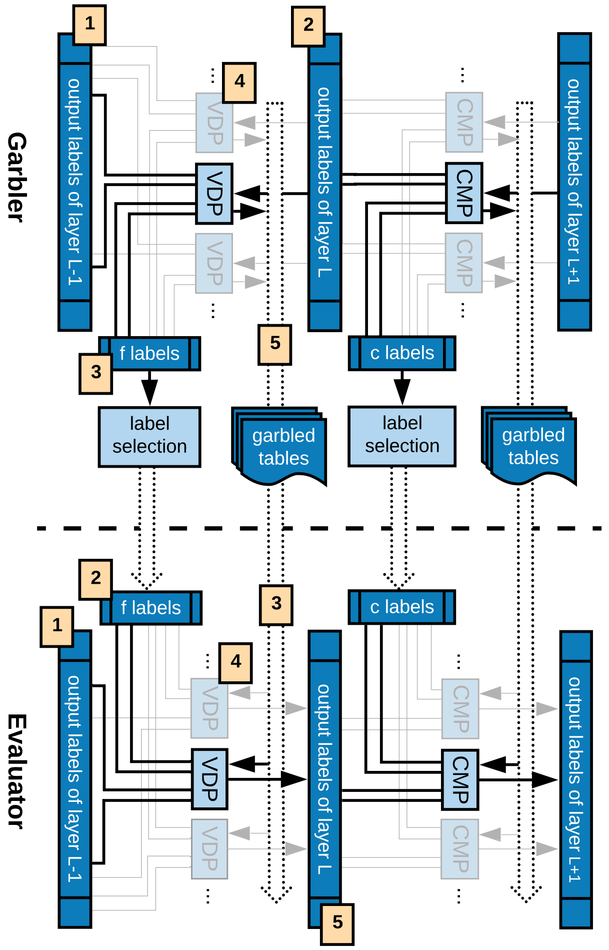


Figure 7: XONN modular and pipelined garbling engine.

Pipelined GC Engine. In XONN, computation and communication are pipelined. For instance, consider a CONV layer followed by an activation layer. We garble/evaluate these layers by multiple invocations of the VDP and CMP circuits (one invocation per output neuron) as illustrated in Figure 7. Upon finishing the garbling process of layer $L-1$, the Garbler starts garbling the L^{th} layer and creates the random labels for output wires of layer L. He also needs to create the random labels associated with his input (i.e., the weight parameters) to layer L. Given a set of input and output labels, Garbler generates the garbled tables, and sends them to the Evaluator as soon as one is ready. He also sends one of the two input labels for his input bits. At the same time, the Evaluator has computed the output labels of the $(L-1)^{th}$ layer. She receives the garbled tables as well as the Garbler's selected input labels and decrypts the tables and stores the output labels of layer L.

Dynamic Memory Management. We design the framework such that the allocated memory for the labels is released as soon as it is no longer needed, reducing the memory usage significantly. For example, without our dynamic memory management, the Garbler had to allocate 10.41GB for the labels and garbled tables for the entire garbling of BC1 network (see Section 7 for network description). In contrast, in our framework, the size of memory allocation never exceeds 2GB and is less than 0.5GB for most of the layers.

4.2 Application Programming Interface (API)

XONN provides a simplified and easy-to-use API for oblivious inference. The framework accepts a high-level description of the network, parameters of each layer, and input structure. It automatically computes the number of invocations and the interconnection between all of the base circuits. Figure 8 shows the complete network description that a user needs to write for a sample network architecture (the BM3 architecture, see Section 7). All of the required circuits are automatically generated using TinyGarble [36] synthesis libraries. It is worth mentioning that for the task of oblivious inference, our API is much simpler compared to the recent *high-level* EzPC framework [25]. For example, the required lines of code to describe BM1, BM2, and BM3 network architectures (see Section 7) in EzPC are 78, 88, and 154, respectively. In contrast, they can be described with only 6, 6, and 10 lines of code in our framework.

```
INPUT 28 1 8
CONV 5 16 1 0 OCA
ACT
MAXPOOL 2
CONV 5 16 1 0
ACT
MAXPOOL 2
FC 100
ACT
FC 10
```

```
Description:

INPUT #input_feature #channels #bit-length

CONV #filter_size #filters #stride
        #Pad #OCA (optional)

MAXPOOl #window_size

FC #output_neurons
```

Figure 8: Sample snippet code in XONN.

Keras to XONN Translation. To further facilitate the adaptation of XONN, a compiler is created to translate the description of the neural network in Keras [37] to the XONN format. The compiler creates the `.xonn` file and puts the network parameters into the required format (`HEX` string) to be read by the framework during the execution of the GC protocol. All of the parameter adjustments are also automatically performed by the compiler.

5 Related Work

One of the early solutions for oblivious inference is proposed by Barni et al. [38] that leverages homomorphic encryption. Howerver, this work leaks some information about the intermediate states of the computation. Orlandi et al. [39] improve upon this work and avoid any information leakage. In [40], authors propose a mechanism to also keep the topology of the neural network secret, at the cost of more computation overhead. A solution based on additively homomorphic encryption and garbled circuits is presented in [41].

CryptoNets [14] suggests the adaptation of Leveled Homomorphic Encryption (LHE) to perform oblivious inference. LHE is a variant of Partially HE that enables evaluation of depth-bounded arithmetic circuits. DeepSecure [13] is a privacy-preserving DL framework that relies on the GC protocol. CryptoDL [42] improves upon CryptoNets [14] and proposes more efficient approximation of the non-linear functions using low-degree polynomials. Their solution is based on LHE and uses mean-pooling in replacement of the max-pooling layer. Chou et al. propose to utilize the sparsity within the DL model to accelerate the inference [43].

SecureML [8] is a privacy-preserving machine learning framework based on homomorphic encryption, GC, and secret sharing. SecureML also uses customized activation functions and supports privacy-preserving training in addition to inference. Two non-colluding servers are used to train the DL model where each client XOR-shares her input and sends the shares to both servers. MiniONN [9] is a mixed-protocol framework for oblivious inference. The underlying cryptographic protocols are HE, GC, and secret sharing.

Chameleon [7] is a more recent mixed-protocol framework for machine learning, i.e., Support Vector Machines (SVMs) as well as DNNs. Authors propose to perform low-depth non-linear functions using the Goldreich-Micali-Wigderson (GMW) protocol [5], high-depth functions by the GC protocol, and linear operations using additive secret sharing. Moreover, they propose to use correlated randomness to more efficiently compute linear operations. EzPC [25] is a secure computation framework that enables users to write high-level programs and create a protocol based on both Boolean and Arithmetic shares. The back-end cryptographic engine is based on the ABY framework [44]. HyCC [45] is a high-level secure computation framework that complies ANSI C programs into efficient hybrid protocols.

Shokri and Shmatikov [46] proposed a solution for privacy-preserving collaborative deep learning where the training data is distributed among many parties. Their approach, enables clients to train their local model on their own training data and update the central model's parameters held by a central server. However, it has been shown that a malicious client can learn significant information about the other client's private data [47]. Google [48] has recently introduced a new approach for securely aggregating the parameter updates from multiple users. However, none of these approaches [46, 48] study the oblivious inference problem. An overview of related frameworks is provided in [49, 50].

Frameworks such as ABY^3 [51] and SecureNN [52] have different computation models and they rely on three (or four) parties during the oblivious inference. In contrast, XONN does not require an additional server for the computation. In E2DM framework [53], the model owner can encrypt and outsource the model to an untrusted server to perform oblivious inference. Concurrently and independently of ours, in TAPAS [54], Sanyal et al. study the binarization of neural networks in the context of oblivious inference. They report inference latency of 147 seconds on MNIST dataset with 98.6% prediction accuracy using custom CNN architecture. However, as we show in Section 7 (BM3 benchmark), XONN outperforms TAPAS by close to *three orders of magnitude*.

Gazelle [10] is the previously most efficient oblivious inference framework. It is a mixed-protocol approach based on additive HE and GC. In Gazelle, convolution operations are performed using the packing property of HE. In this approach, many numbers are packed inside a single ciphertext for faster convolutions. In Section 6, we briefly discuss one of the essential requirements that the Gazelle protocol has to satisfy in order to be secure, namely, *circuit privacy*.

High-Level Comparison. In contrast to prior art, we propose a DL-secure computation co-design approach. To the best of our knowledge, DeepSecure [13] is the only solution that preprocesses the data and network before the secure computation protocol. However, the preprocessing step reveals some information about the network parameters and structure of data. Moreover, another advantage of XONN is the constant round complexity regardless of the number of layers in the NN. It has been shown that round complexity is one of the important criteria in designing secure computation protocols [55] since the performance can significantly be reduced where the network latency is high. As we show in Section 7, XONN outperforms all previous solutions in inference latency. Table 2 summarizes a high-level comparison between state-of-the-art oblivious inference frameworks.

Table 2: High-Level Comparison of oblivious inference frameworks. “**C**”onstant round complexity. “**D**”eep learning/secure computation co-design. “**I**”ndependence of secondary server. “**U**”pgradeable to malicious security using standard solutions. “**S**”upporting any non-linear layer.

Framework	**Crypto. Protocol**	**C**	**D**	**I**	**U**	**S**
CryptoNets [14]	HE	✓	✗	✓	✗	✗
DeepSecure [13]	GC	✓	✓	✓	✓	✓
SecureML [8]	HE, GC, SS	✗	✗	✗	✗	✗
MiniONN [9]	HE, GC, SS	✗	✗	✓	✗	✓
Chameleon [7]	GC, GMW, SS	✗	✗	✗	✗	✓
EzPC [25]	GC, SS	✗	✗	✓	✗	✓
Gazelle [10]	HE, GC, SS	✗	✗	✓	✗	✓
XONN (This work)	GC, SS	✓	✓	✓	✓	✓

6 Circuit Privacy

In Gazelle [10], for each linear layer, the protocol starts with a vector $\mathbf{m}$ that is secret-shared between client $\mathbf{m_1}$ and server $\mathbf{m_2}$ ($\mathbf{m} = \mathbf{m_1} + \mathbf{m_2}$). The protocol outputs the secret shares of the vector $\mathbf{m}' = A \cdot \mathbf{m}$ where A is a matrix known to the server but not to the client. The protocol has the following procedure: (i) Client generates a pair (pk, sk) of public and secret keys of an additive homomorphic encryption scheme HE. (ii) Client sends $\mathsf{HE}.Enc_{pk}(\mathbf{m_1})$ to the server. Server adds its share ($\mathbf{m_2}$) to the ciphertext and recovers encryption of $\mathbf{m}$: $\mathsf{HE}.Enc_{pk}(\mathbf{m})$. (iii) Server homomorphically evaluates the multiplication with A and obtains the encryption of $\mathbf{m}'$. (iv) Server secret shares $\mathbf{m}'$ by sampling a random vector $\mathbf{r}$ and returns ciphertext $\mathbf{c} = \mathsf{HE}.Enc_{pk}(\mathbf{m}' - \mathbf{r})$ to the client. The client can decrypt $\mathbf{c}$ using private key sk and obtain $\mathbf{m}' - \mathbf{r}$.

Gazelle uses the Brakerski-Fan-Vercauteren (BFV) scheme [56, 57]. However, the vanilla BFV scheme does not provide circuit privacy. At high-level, the circuit privacy requirement states that the ciphertext $\mathbf{c}$ should not reveal any information about the private inputs to the client (i.e., A and $\mathbf{r}$) other than the underlying plaintext $A \cdot \mathbf{m} - \mathbf{r}$. Otherwise, some information is leaked. Gazelle proposes two methods to provide circuit privacy that are not incorporated in their implementation. Hence, we need to scale up their performance numbers for a fair comparison.

The first method is to let the client and server engage in a two-party secure decryption protocol, where the input of client is sk and input of server is $\mathbf{c}$. However, this method adds communication and needs extra rounds of interaction. A more widely used approach is *noise flooding*. Roughly speaking, the server adds a large noise term to $\mathbf{c}$ before returning it to the client. The noise is big enough to drown any extra information contained in the ciphertext, and still small enough to so that it still decrypts to the same plaintext.

For the concrete instantiation of Gazelle, one needs to triple the size of ciphertext modulus q from 60 bits to 180 bits, and increase the ring dimension n from 2048 to 8192. The (amortized) complexity of homomorphic operations in the BFV scheme is approximately $O(\log n \; \log q)$, with the exception that some operations run in $O(\log q)$ amortized time. Therefore, adding noise flooding would result in a *3-3.6 times slow down* for the HE component of Gazelle. To give some concrete examples, we consider two networks used for benchmarking in Gazelle: MNIST-D and CIFAR-10 networks. For the MNIST-D network, homomorphic encryption takes 55% and 22% in online and total time, respectively. For CIFAR-10, the corresponding figures are 35%, and 10%[1]. Therefore, we estimate that the total time for MNIST-D will grow from 0.81s to 1.16-1.27s (network BM3 in this paper). In the case of CIFAR-10 network, the total time will grow from 12.9s to 15.48-16.25s.

[1] these percentage numbers are obtained through private communication with the authors.

7 Experimental Results

We evaluate XONN on MNIST and CIFAR10 datasets, which are two popular classification benchmarks used in prior work. In addition, we provide four healthcare datasets to illustrate the applicability of XONN in real-world scenarios. For training XONN, we use Keras [37] with Tensorflow backend [58]. The source code of XONN is compiled with GCC 5.5.0 using O3 optimization. All Boolean circuits are synthesized using Synopsys Design Compiler 2015. Evaluations are performed on (Ubuntu 16.04 LTS) machines with Intel-Core i7-7700k and 32GB of RAM. The experimental setup is comparable (but has less computational power) compared to the prior art [10]. Consistent with prior frameworks, we evaluate the benchmarks in the LAN setting.

7.1 Evaluation on MNIST

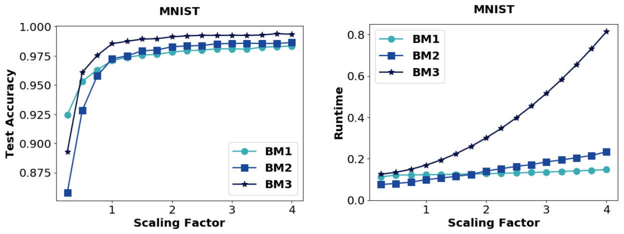
There are mainly three network architectures that prior works have implemented for the MNIST dataset. We convert these reference networks into their binary counterparts and train them using the standard BNN training algorithm [19]. Table 3 summarizes the architectures for the MNIST dataset.

Table 3: Summary of the trained binary network architectures evaluated on the MNIST dataset. Detailed descriptions are available in Appendix B.2, Table 13.

Arch.	Previous Papers	Description
BM1	SecureML [8], MiniONN [9]	3 FC
BM2	CryptoNets [14], MiniONN [9], DeepSecure [13], Chameleon [7]	1 CONV, 2 FC
BM3	MiniONN [9], EzPC [25]	2 CONV, 2MP, 2FC

Analysis of Network Scaling: Recall that the classification accuracy of XONN is controlled by scaling the number of neurons in all layers (Section 3.1). Figure 9a depicts the inference accuracy with different scaling factors (more details in Table 11 in Appendix B.2). As we increase the scaling factor, the accuracy of the network increases. This accuracy improvement comes at the cost of a higher computational complexity of the (scaled) network. As a result, increasing the scaling factor leads to a higher runtime. Figure 9b depicts the runtime of different BNN architectures as a function of the scaling factor s. Note that the runtime grows (almost) quadratically with the scaling factor due to the quadratic increase in the number of *Popcount* operations in the neural network (see $BM3$). However, for the $BM1$ and $BM2$ networks, the overall runtime is dominated by the constant initialization cost of the OT protocol ($\sim$ 70 millisecond).

GC Cost and the Effect of OCA: The communication cost of GC is the key contributor to the overall runtime of XONN. Here, we analyze the effect of the scaling factor on the total message size. Figure 10 shows the communication cost of GC for the BM1 and BM2 network architectures. As can be seen, the message size increases with the scaling factor. We also observe that the OCA protocol drastically reduces the message size. This is due to the fact that the first layer of BM1 and BM2 models account for a large portion of the overall computation; hence, improving the first layer with OCA has a drastic effect on the overall communication.



(a) (b)

Figure 9: Effect of scaling factor on (a) accuracy and (b) inference runtime of MNIST networks. No pruning was applied in this evaluation.

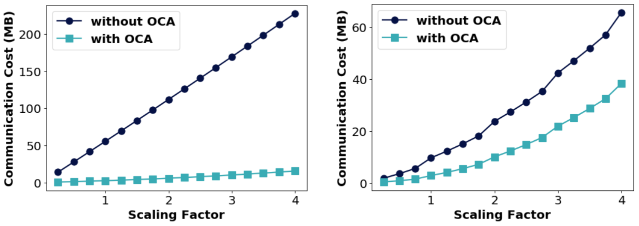


Figure 10: Effect of OCA on the communication of the BM1 (left) and BM2 (right) networks for different scaling factors. No pruning was applied in this evaluation.

Comparison to Prior Art: We emphasize that, unlike previous work, the accuracy of XONN can be customized by tuning the scaling factor (s). Furthermore, our channel/neuron pruning step (Algorithm 2) can reduce the GC cost in a post-processing phase. To provide a fair comparison between XONN and prior art, we choose a proper scaling factor and trim the pertinent scaled BNN such that the corresponding BNN achieves the same accuracy as the previous work. Table 4 compares XONN with the previous work in terms of accuracy, latency, and communication cost (a.k.a., message size). The last column shows the scaling factor (s) used to increase the width of the hidden layers of the BNN. Note that the scaled network is further trimmed using Algorithm 2.

In XONN, the runtime for oblivious transfer is at least $\sim$ 0.07 second for initiating the protocol and then it grows linearly with the size of the garbled tables; As a result, in very small architectures such as $BM1$, our solution is slightly slower than previous works since the constant runtime dominates the total runtime. However, for the $BM3$ network which has higher complexity than $BM1$ and $BM2$, XONN achieves a more prominent advantage over prior art. In summary, our solution achieves up to 7.7× faster inference (average of 3.4×) compared to Gazelle [10]. Compared to MiniONN [9], XONN has up to 62× lower latency (average of 26×) Table 4. Compared to EzPC [25], our framework is 34× faster. XONN achieves 37.5×, 1859×, 60.4×, and 14× better latency compared to SecureML [8], CryptoNets [14], DeepSecure [13], and Chameleon [7], respectively.

Table 4: Comparison of XONN with the state-of-the-art for the MNIST network architectures.

Arch.	Framework	Runtime (s)	Comm. (MB)	Acc. (%)	s
BM1	SecureML	4.88	-	93.1	-
	MiniONN	1.04	15.8	97.6	-
	EzPC	0.7	76	97.6	-
	Gazelle	0.09	0.5	97.6	-
	XONN	0.13	4.29	97.6	1.75
BM2	CryptoNets	297.5	372.2	98.95	-
	DeepSecure	9.67	791	98.95	-
	MiniONN	1.28	47.6	98.95	-
	Chameleon	2.24	10.5	99.0	-
	EzPC	0.6	70	99.0	-
	Gazelle	0.29	8.0	99.0	-
	XONN	0.16	38.28	98.64	4.00
BM3	MiniONN	9.32	657.5	99.0	-
	EzPC	5.1	501	99.0	-
	Gazelle	1.16	70	99.0	-
	XONN	0.15	32.13	99.0	2.00

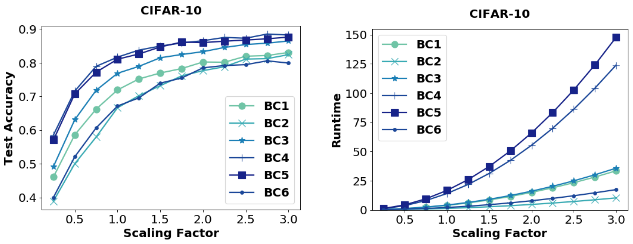
7.2 Evaluation on CIFAR-10

In Table 5, we summarize the network architectures that we use for the CIFAR-10 dataset. In this table, BC1 is the binarized version of the architecture proposed by MiniONN. To evaluate the scalability of our framework to larger networks, we also binarize the Fitnet [59] architectures, which are denoted as BC2-BC5. We also evaluate XONN on the popular VGG16 network architecture (BC6). Detailed architecture descriptions are available in Appendix B.2, Table 13.

Table 5: Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset.

Arch.	Previous Papers	Description
BC1	MiniONN[9], Chameleon [7], EzPC [25], Gazelle [10]	7 CONV, 2 MP, 1 FC
BC2	Fitnet [59]	9 CONV, 3 MP, 1 FC
BC3	Fitnet [59]	9 CONV, 3 MP, 1 FC
BC4	Fitnet [59]	11 CONV, 3 MP, 1 FC
BC5	Fitnet [59]	17 CONV, 3 MP, 1 FC
BC6	VGG16 [60]	13 CONV, 5 MP, 3 FC

Analysis of Network Scaling: Similar to the analysis on the MNIST dataset, we show that the accuracy of our binary models for CIFAR-10 can be tuned based on the scaling factor that determines the number of neurons in each layer. Figure 11a depicts the accuracy of the BNNs with different scaling factors. As can be seen, increasing the scaling factor enhances the classification accuracy of the BNN. The runtime also increases with the scaling factor as shown in Figure 11b (more details in Table 12, Appendix B.2).



(a) (b)

Figure 11: (a) Effect of scaling factor on accuracy for CIFAR-10 networks. (b) Effect of scaling factor on runtime. No pruning was applied in this evaluation.

Comparison to Prior Art: We scale the BC2 network with a factor of $s = 3$, then prune it using Algorithm 2. Details of pruning steps are available in Table 10 in Appendix B.1. The resulting network is compared against prior art in Table 6. As can be seen, our solution achieves 2.7×, 45.8×, 9.1×, and 93.1× lower latency compared to Gazelle, EzPC, Chameleon, and MiniONN, respectively.

Table 6: Comparison of XONN with prior art on CIFAR-10.

Framework	Runtime (s)	Comm. (MB)	Acc. (%)	s
MiniONN	544	9272	81.61	-
Chameleon	52.67	2650	81.61	-
EzPC	265.6	40683	81.61	-
Gazelle	15.48	1236	81.61	-
XONN	5.79	2599	81.85	3.00

7.3 Evaluation on Medical Datasets

One of the most important applications of oblivious inference is medical data analysis. Recent advances in deep learning greatly benefit many complex diagnosis tasks that require exhaustive manual inspection by human experts [61, 62, 63, 64]. To showcase the applicability of oblivious inference in real-world medical applications, we provide several benchmarks for publicly available healthcare datasets summarized in Table 7. We split the datasets into validation and training portions as indicated in the last two columns of Table 7. All datasets except Malaria Infection are normalized to have 0 mean and standard deviation of 1 per feature. The images of Malaria Infection dataset are resized to 32×32 pictures. The normalized datasets are quantized up to 3 decimal digits. Detailed architectures are available in Appendix B.2, Table 13 We report the validation accuracy along with inference time and message size in Table 8.

Table 7: Summary of medical application benchmarks.

Task	Arch.	Description	# of Samples	
			Tr.	Val.
Breast Cancer [65]	BH1	3 FC	453	113
Diabetes [66]	BH2	3 FC	615	153
Liver Disease [67]	BH3	3 FC	467	116
Malaria Infection [68]	BH4	2 CONV, 2 MP, 2 FC	24804	2756

Table 8: Runtime, communication cost (Comm.), and accuracy (Acc.) for medical benchmarks.

Arch.	Runtime (ms)	Comm. (MB)	Acc. (%)
BH1	82	0.35	97.35
BH2	75	0.16	80.39
BH3	81	0.3	80.17
BH4	482	120.75	95.03

8 Conclusion

We introduce XONN, a novel framework to automatically train and use deep neural networks for the task of oblivious inference. XONN utilizes Yao's Garbled Circuits (GC) protocol and relies on binarizing the DL models in order to translate costly matrix multiplications to XNOR operations that are free in the GC protocol. Compared to Gazelle [10], prior best solution, XONN achieves 7× lower latency. Moreover, in contrast to Gazelle that requires one round of interaction for each layer, our solution needs a constant round of interactions regardless of the number of layers. Maintaining constant round complexity is an important requirement in Internet settings as a typical network latency can significantly degrade the performance of oblivious inference. Moreover, since our solution relies on the GC protocol, it can provide much stronger security guarantees such as security against malicious adversaries using standard cut-and-choose protocols. XONN high-level API enables clients to utilize the framework with a minimal number of lines of code. To further facilitate the adaptation of our framework, we design a compiler to translate the neural network description in Keras format to that of XONN.

Acknowledgements We would like to thank the anonymous reviewers for their insightful comments.

References

[1] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security*, 2016.

[2] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[4] Andrew Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[5] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[6] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[7] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ASIACCS'18*, 2018.

[8] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[9] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS*, 2017.

[10] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. *USENIX Security*, 2018.

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. CVPR, 2015.

[12] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[13] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable provably-secure deep learning. *DAC*, 2018.

[14] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[15] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[16] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*, pages 145–161. Springer, 2003.

[17] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC*, 1996.

[18] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS*, 2013.

[19] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[20] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[21] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. ReBNet: Residual binarized neural network. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–64. IEEE, 2018.

[22] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.

[23] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[24] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[25] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation. *IACR Cryptology ePrint Archive*, 2017/1109, 2017.

[26] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[27] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*. ACM, 2015.

[28] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *S&P*. IEEE, 2017.

[29] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, 2012.

[30] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO 2013*, pages 18–35. Springer, 2013.

[31] Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology*, 29(2):456–490, 2016.

[32] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P*, 2013.

[33] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, 1999.

[34] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *EUROCRYPT*, 2015.

[35] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.

[36] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S&P*, 2015.

[37] François Chollet et al. Keras. `https://keras.io`, 2015.

[38] Mauro Barni, Claudio Orlandi, and Alessandro 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.

[39] Claudio Orlandi, Alessandro Piva, and Mauro Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007(1):037343, 2007.

[40] Ahmad-Reza Sadeghi and Thomas Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In *International Conference on Information Security and Cryptology*, pages 336–353. Springer, 2008.

[41] Mauro Barni, Pierluigi Failla, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Privacy-preserving ECG classification with branching programs and neural networks. *IEEE Transactions on Information Forensics and Security*, 6(2):452–468, 2011.

[42] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. Privacy-preserving machine learning as a service. *Proceedings on Privacy Enhancing Technologies*, 2018(3):123–142, 2018.

[43] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster CryptoNets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

[44] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[45] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 847–861. ACM, 2018.

[46] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *ACM CCS*, 2015.

[47] Briland Hitaj, Giuseppe Ateniese, and Fernando Pérez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. In *ACM CCS*, 2017.

[48] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM CCS*, 2017.

[49] M Sadegh Riazi, Bita Darvish Rouhani, and Farinaz Koushanfar. Deep learning on private data. *IEEE Security and Privacy (S&P) Magazine.*, 2019.

[50] M Sadegh Riazi and Farinaz Koushanfar. Privacy-preserving deep learning and inference. In *Proceedings of the International Conference on Computer-Aided Design*, page 18. ACM, 2018.

[51] Payman Mohassel and Peter Rindal. ABY3: a mixed protocol framework for machine learning. In *ACM CCS*, 2018.

[52] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: Efficient and private neural network training, 2018.

[53] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *ACM CCS*, 2018.

[54] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. TAPAS: Tricks to accelerate (encrypted) prediction as a service. In *International Conference on Machine Learning*, pages 4497–4506, 2018.

[55] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 578–590. ACM, 2016.

[56] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in cryptology–crypto 2012*, pages 868–886. Springer, 2012.

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

[58] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Operating Systems Design and Implementation (OSDI)*, 2016.

[59] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[60] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[61] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24, 2019.

[62] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[63] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. Predicting the sequence specificities of dna- and rna-binding proteins by deep learning. *Nature biotechnology*, 33(8):831, 2015.

[64] Alvin Rajkomar, Eyal Oren, Kai Chen, Andrew M Dai, Nissan Hajaj, Michaela Hardt, Peter J Liu, Xiaobing Liu, Jake Marcus, Mimi Sun, et al. Scalable and accurate deep learning with electronic health records. *npj Digital Medicine*, 1(1):18, 2018.

[65] Breast Cancer Wisconsin, accessed on 01/20/2019. `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data`.

[66] Pima Indians Diabetes, accessed on 01/20/2019. `https://www.kaggle.com/uciml/pima-indians-diabetes-database`.

[67] Indian Liver Patient Records, accessed on 01/20/2019. `https://www.kaggle.com/uciml/indian-liver-patient-records`.

[68] Malaria Cell Images, accessed on 01/20/2019. `https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria`.

A Attacks on Deep Neural Networks

In this section, we review three of the most important attacks against deep neural networks that are relevant to the context of oblivious inference [1, 27, 28]. In all three, a client-server model is considered where the client is the adversary and attempts to learn more about the model held by the server. The client sends many inputs and receives the inference results . He then analyzes the results to infer more information about either the network parameters or the training data that has been used in the training phase of the model. We briefly review each attack and illustrate a simple defense mechanism with negligible overhead based on the suggestions provided in these works.

Model Inversion Attack [27]. In the black-box access model of this attack (which fits the computational model of this work), an adversarial client attempts to learn about a prototypical sample of one of the classes. The client iteratively creates an input that maximizes the confidence score corresponding to the target class. Regardless of the specific training process, the attacker can learn significant information by querying the model many times.

Model Extraction Attack [1]. In this type of attack, an adversary's goal is to estimate the parameters of the machine learning model held by the server. For example, in a logistic regression model with n parameters, the model can be extracted by querying the server n times and upon receiving the confidence values, solving a system of n equations. Model extraction can diminish the pay-per-prediction business model of technology companies. Moreover, it can be used as a pre-step towards the model inversion attack.

Membership Inference Attack [28]. The objective of this attack is to identify whether a given input has been used in the training phase of the model or not. This attack raises certain privacy concerns. The idea behind this attack is that the neural networks usually perform better on the data that they were trained on. Therefore, two inputs that belong to the same class, one used in the training phase and one not, will have noticeable differences in the confidence values. This behavior is called *overfitting*. The attack can be mitigated using regularization techniques that reduce the dependency of the DL model on a single training sample. However, overfitting is not the only contributor to this information leakage.

Defense Mechanisms. In the prior state-of-the-art oblivious inference solution [9], it has been suggested to limit the number of queries from a specific client to limit the information leakage. However, in practice, an attacker can impersonate himself as many different clients and circumvent this defense mechanism. Note that all three attacks rely on the fact that along with the inference result, the server provides the confidence vector that specifies how likely the client's input belongs to each class. Therefore, as suggested by prior work [1, 27, 28], it is recommended to augment a *filter* layer that (i) rounds the confidence scores or (ii) selects the index of a class that has the highest confidence score.

1. *Rounding the confidence values:* Rounding the values simply means omitting one (or more) of the Least Significant Bit (LSB) of all of the numbers in the last layer. This operation is in fact *free* in GC since it means Garbler has to avoid providing the mapping for those LSBs.
2. *Reporting the class label:* This operation is equivalent to computing `argmax` on the last layer. For a vector of size c where each number is represented with b bits, `argmax` is translated to $c \cdot (2b+1)$ many non-`XOR` (`AND`) gates. For example, in a typical architecture for MNIST (e.g., BM3) or CIFAR-10 dataset (e.g., BC1), the overhead is 1.68E-2% and 1.36E-4%, respectively.

Note that the two aforementioned defense mechanisms can be augmented to any framework that supports non-linear functionalities [7, 9, 13]. However, we want to emphasize that compared to mixed-protocol solutions, this means that another round of communication is usually needed to support the filter layer. Whereas, in XONN the filter layer does not increase the number of rounds and has negligible overhead compared to the overall protocol.

B Experimental Details

B.1 Network Trimming Examples

Table 9 and 10 summarize the trimming steps for the MNIST and CIFAR-10 benchmarks, respectively.

Table 9: Trimming MNIST architectures.

Network	Property	Trimming Step				Change
		initial	step 1	step 2	step 3	
BM1 (s=1.75)	Acc. (%)	97.63	97.59	97.28	97.02	-0.61%
	Comm. (MB)	4.95	4.29	3.81	3.32	1.49× less
	Lat. (ms)	158	131	114	102	1.54× faster
BM2 (s=4)	Acc. (%)	98.64	98.44	98.37	98.13	-0.51%
	Comm. (MB)	38.28	28.63	24.33	15.76	2.42× less
	Lat. (ms)	158	144	134	104	1.51× faster
BM3 (s=2)	Acc. (%)	99.22	99.11	98.96	99.00	-0.22%
	Comm. (MB)	56.08	42.51	37.34	32.13	1.75× less
	Lat. (ms)	190	165	157	146	1.3× faster

Table 10: Trimming the BC2 network for CIFAR-10.

Property	Trimming Step				Change
	initial	step 1	step 2	step 3	
Acc. (%)	82.40	82.39	82.41	81.85	-0.55%
Com. (GB)	3.38	3.05	2.76	2.60	1.30× less
Lat. (s)	7.59	6.87	6.23	5.79	1.31× faster

B.2 Accuracy, Runtime, and Communication

Runtime and communication reports are available in Table 11 and Table 12 for MNIST and CIFAR-10 benchmarks, respectively. The corresponding neural network architectures are provided in Table 13. Entries corresponding to a communication of more than 40GB are estimated using numerical runtime models.

Table 11: Accuracy (Acc.), communication (Comm.), and latency (Lat.) for MNIST dataset. Channel/neuron trimming is not applied.

Arch.	s	Acc. (%)	Comm. (MB)	Lat. (s)
BM1	1	97.10	2.57	0.12
	1.5	97.56	4.09	0.13
	2	97.82	5.87	0.13
	3	98.10	10.22	0.14
	4	98.34	15.62	0.15
BM2	1	97.25	2.90	0.10
	1.50	97.93	5.55	0.12
	2	98.28	10.09	0.14
	3	98.56	21.90	0.18
	4	98.64	38.30	0.23
BM3	1	98.54	17.59	0.17
	1.5	98.93	36.72	0.22
	2	99.13	62.77	0.3
	3	99.26	135.88	0.52
	4	99.35	236.78	0.81

Table 12: Accuracy (Acc.), communication (Comm.), and latency (Lat.) for CIFAR-10 dataset. Channel/neuron trimming is not applied.

Arch.	s	Acc. (%)	Comm. (MB)	Lat. (s)
BC1	1	0.72	1.26	3.96
	1.5	0.77	2.82	8.59
	2	0.80	4.98	15.07
	3	0.83	11.15	33.49
BC2	1	0.67	0.39	1.37
	1.5	0.73	0.86	2.78
	2	0.78	1.53	4.75
	3	0.82	3.40	10.35
BC3	1	0.77	1.35	4.23
	1.5	0.81	3.00	9.17
	2	0.83	5.32	16.09
	3	0.86	11.89	35.77
BC4	1	0.82	4.66	14.12
	1.5	0.85	10.41	31.33
	2	0.87	18.45	55.38
	3	0.88	41.37	123.94
BC5	1	0.81	5.54	16.78
	1.5	0.85	12.40	37.29
	2	0.86	21.98	65.94
	3	0.88	49.30	147.66
BC6	1	0.67	0.65	2.15
	1.5	0.74	1.46	4.55
	2	0.78	2.58	7.91
	3	0.80	5.77	17.44

Table 13: Evaluated network architectures.

BM1

1	**FC** [input: 784, output: $128s$] + **BN** + **BA**
2	**FC** [input: $128s$, output: $128s$] + **BN** + **BA**
3	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BM2

1	**CONV** [input: $28 \times 28 \times 1$, window: 5×5, stride: 2, kernels: $5s$, output: $12 \times 12 \times 5s$] + **BN** + **BA**
2	**FC** [input: $720s$, output: $100s$] + **BN** + **BA**
3	**FC** [input: $100s$, output: 10] + **BN** + **Softmax**

BM3

1	**CONV** [input: $28 \times 28 \times 1$, window: 5×5, stride: 1, kernels: $16s$, output: $24 \times 24 \times 16s$] + **BN** + **BA**
2	**MP** [input: $24 \times 24 \times 16s$, window: 2×2, output: $12 \times 12 \times 16s$]
3	**CONV** [input: $12 \times 12 \times 16s$, window: 5×5, stride: 1, kernels: $16s$, output: $8 \times 8 \times 16s$] + **BN** + **BA**
4	**MP** [input: $8 \times 8 \times 16s$, window: 2×2, output: $4 \times 4 \times 16s$]
5	**FC** [input: $256s$, output: $100s$] + **BN** + **BA**
6	**FC** [input: $100s$, output: 10] + **BN** + **Softmax**

BC1

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $64s$, output: $30 \times 30 \times 64s$] + **BN** + **BA**
2	**CONV** [input: $30 \times 30 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $28 \times 28 \times 64s$] + **BN** + **BA**
3	**MP** [input: $28 \times 28 \times 64s$, window: 2×2, output: $14 \times 14 \times 64s$]
4	**CONV** [input: $14 \times 14 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $12 \times 12 \times 64s$] + **BN** + **BA**
5	**CONV** [input: $12 \times 12 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $10 \times 10 \times 64s$] + **BN** + **BA**
6	**MP** [input: $10 \times 10 \times 64s$, window: 2×2, output: $5 \times 5 \times 64s$]
7	**CONV** [input: $5 \times 5 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $3 \times 3 \times 64s$] + **BN** + **BA**
8	**CONV** [input: $3 \times 3 \times 64s$, window: 1×1, stride: 1, kernels: $64s$, output: $3 \times 3 \times 64s$] + **BN** + **BA**
9	**CONV** [input: $3 \times 3 \times 64s$, window: 1×1, stride: 1, kernels: $16s$, output: $3 \times 3 \times 16s$] + **BN** + **BA**
10	**FC** [input: $144s$, output: 10] + **BN** + **Softmax**

BC2

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
4	**MP** [input: $32 \times 32 \times 16s$, window: 2×2, output: $16 \times 16 \times 16s$]
5	**CONV** [input: $16 \times 16 \times 16s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
6	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
7	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
8	**MP** [input: $16 \times 16 \times 32s$, window: 2×2, output: $8 \times 8 \times 32s$]
9	**CONV** [input: $8 \times 8 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $6 \times 6 \times 48s$] + **BN** + **BA**
10	**CONV** [input: $6 \times 6 \times 48s$, window: 3×3, stride: 1, kernels: $48s$, output: $4 \times 4 \times 48s$] + **BN** + **BA**
11	**CONV** [input: $4 \times 4 \times 48s$, window: 3×3, stride: 1, kernels: $64s$, output: $2 \times 2 \times 64s$] + **BN** + **BA**
12	**MP** [input: $2 \times 2 \times 64s$, window: 2×2, output: $1 \times 1 \times 64s$]
13	**FC** [input: $64s$, output: 10] + **BN** + **Softmax**

BC3

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
4	**MP** [input: $32 \times 32 \times 32s$, window: 2×2, output: $16 \times 16 \times 32s$]
5	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $16 \times 16 \times 48s$] + **BN** + **BA**
6	**CONV** [input: $16 \times 16 \times 48s$, window: 3×3, stride: 1, kernels: $64s$, output: $16 \times 16 \times 64s$] + **BN** + **BA**
7	**CONV** [input: $16 \times 16 \times 64s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
8	**MP** [input: $16 \times 16 \times 80s$, window: 2×2, output: $8 \times 8 \times 80s$]
9	**CONV** [input: $8 \times 8 \times 80s$, window: 3×3, stride: 1, kernels: $96s$, output: $6 \times 6 \times 96s$] + **BN** + **BA**
10	**CONV** [input: $6 \times 6 \times 96s$, window: 3×3, stride: 1, kernels: $96s$, output: $4 \times 4 \times 96s$] + **BN** + **BA**
11	**CONV** [input: $4 \times 4 \times 96s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
12	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
13	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BC4

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $32 \times 32 \times 48s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 48s$, window: 3×3, stride: 1, kernels: $64s$, output: $32 \times 32 \times 64s$] + **BN** + **BA**
4	**CONV** [input: $32 \times 32 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $32 \times 32 \times 64s$] + **BN** + **BA**
5	**MP** [input: $32 \times 32 \times 64s$, window: 2×2, output: $16 \times 16 \times 64s$]
6	**CONV** [input: $16 \times 16 \times 64s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
7	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
8	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
9	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
10	**MP** [input: $16 \times 16 \times 80s$, window: 2×2, output: $8 \times 8 \times 80s$]
11	**CONV** [input: $8 \times 8 \times 80s$, window: 3×3, stride: 1, kernels: $128s$, output: $6 \times 6 \times 128s$] + **BN** + **BA**
12	**CONV** [input: $6 \times 6 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
13	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
14	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
15	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BC5

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
4	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $32 \times 32 \times 48s$] + **BN** + **BA**
5	**CONV** [input: $32 \times 32 \times 48s$, window: 3×3, stride: 1, kernels: $48s$, output: $32 \times 32 \times 48s$] + **BN** + **BA**
6	**MP** [input: $32 \times 32 \times 48s$, window: 2×2, output: $16 \times 16 \times 48s$]
7	**CONV** [input: $16 \times 16 \times 48s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
8	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
9	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
10	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
11	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
12	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
13	**MP** [input: $16 \times 16 \times 80s$, window: 2×2, output: $8 \times 8 \times 80s$]
14	**CONV** [input: $8 \times 8 \times 80s$, window: 3×3, stride: 1, kernels: $128s$, output: $8 \times 8 \times 128s$] + **BN** + **BA**
15	**CONV** [input: $8 \times 8 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $8 \times 8 \times 128s$] + **BN** + **BA**
16	**CONV** [input: $8 \times 8 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $8 \times 8 \times 128s$] + **BN** + **BA**
17	**CONV** [input: $8 \times 8 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $6 \times 6 \times 128s$] + **BN** + **BA**
18	**CONV** [input: $6 \times 6 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
19	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
20	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
21	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BC6

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
3	**MP** [input: $32 \times 32 \times 16s$, window: 2×2, output: $16 \times 16 \times 16s$]
4	**CONV** [input: $16 \times 16 \times 16s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
5	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
6	**MP** [input: $16 \times 16 \times 32s$, window: 2×2, output: $8 \times 8 \times 32s$]
7	**CONV** [input: $8 \times 8 \times 32s$, window: 3×3, stride: 1, kernels: $64s$, output: $8 \times 8 \times 64s$] + **BN** + **BA**
8	**CONV** [input: $8 \times 8 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $8 \times 8 \times 64s$] + **BN** + **BA**
9	**CONV** [input: $8 \times 8 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $8 \times 8 \times 64s$] + **BN** + **BA**
10	**MP** [input: $8 \times 8 \times 64s$, window: 2×2, output: $4 \times 4 \times 64s$]
11	**CONV** [input: $4 \times 4 \times 64s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
12	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
13	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
14	**MP** [input: $4 \times 4 \times 128s$, window: 2×2, output: $2 \times 2 \times 128s$]
15	**CONV** [input: $2 \times 2 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
16	**CONV** [input: $2 \times 2 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
17	**CONV** [input: $2 \times 2 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
18	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
19	**FC** [input: $128s$, output: $512s$] + **BN** + **BA**
20	**FC** [input: $512s$, output: $512s$] + **BN** + **BA**
21	**FC** [input: $512s$, output: 10] + **BN** + **Softmax**

BH1

1	**FC** [input: 30, output: 16] + **BN** + **BA**
2	**FC** [input: 16, output: 16] + **BN** + **BA**
3	**FC** [input: 16, output: 2] + **BN** + **Softmax**

BH2

1	**FC** [input: 8, output: 20] + **BN** + **BA**
2	**FC** [input: 20, output: 20] + **BN** + **BA**
3	**FC** [input: 20, output: 2] + **BN** + **Softmax**

BH3

1	**FC** [input: 10, output: 32] + **BN** + **BA**
2	**FC** [input: 32, output: 32] + **BN** + **BA**
3	**FC** [input: 32, output: 2] + **BN** + **Softmax**

BH4

1	**CONV** [input: $32 \times 32 \times 3$, window: 5×5, stride: 1, kernels: 36, output: $28 \times 28 \times 36$] + **BN** + **BA**
2	**MP** [input: $28 \times 28 \times 36$, window: 2×2, output: $14 \times 14 \times 36$]
3	**CONV** [input: $14 \times 14 \times 36$, window: 5×5, stride: 1, kernels: 36, output: $10 \times 10 \times 36$] + **BN** + **BA**
4	**MP** [input: $10 \times 10 \times 36$, window: 2×2, output: $5 \times 5 \times 36$]
5	**FC** [input: 900, output: 72] + **BN** + **BA**
6	**FC** [input: 72, output: 2] + **BN** + **Softmax**