VeriDL: Integrity Verification of Outsourced Deep Learning Services

Boxiang Dong[1], Bo Zhang[2], and Hui (Wendy) Wang[3]✉

[1] Montclair State University, Montclair, NJ dongb@montclair.edu
[2] Amazon Inc., Seattle, WA bzhanga@amazon.com
[3] Stevens Institute of Technology, Hoboken, NJ Hui.Wang@stevens.edu

Abstract. Deep neural networks (DNNs) are prominent due to their superior performance in many fields. The deep-learning-as-a-service (DLaaS) paradigm enables individuals and organizations (clients) to outsource their DNN learning tasks to the cloud-based platforms. However, the DLaaS server may return incorrect DNN models due to various reasons (e.g., Byzantine failures). This raises the serious concern of how to verify if the DNN models trained by potentially untrusted DLaaS servers are indeed correct. To address this concern, in this paper, we design VeriDL, a framework that supports efficient correctness verification of DNN models in the DLaaS paradigm. The key idea of VeriDL is the design of a small-size cryptographic proof of the training process of the DNN model, which is associated with the model and returned to the client. Through the proof, VeriDL can verify the correctness of the DNN model returned by the DLaaS server with a deterministic guarantee and cheap overhead. Our experiments on four real-world datasets demonstrate the efficiency and effectiveness of VeriDL.

Keywords: Deep learning, integrity verification, deep-learning-as-a-service

1 Introduction

The recent abrupt advances in deep learning (DL) [1, 10] have led to breakthroughs in many fields such as speech recognition, image classification, text translation, etc. However, this success crucially relies on the availability of both hardware and software resources, as well as human expertise for many learning tasks. As the complexity of these tasks is often beyond non-DL-experts, the rapid growth of DL applications has created a demand for cost-efficient, off-shelf solutions. This motivated the emerge of the *deep-learning-as-a-service* (DLaaS) paradigm which enables individuals and organizations (clients) to outsource their data and deep learning tasks to the cloud-based service providers for their needs of flexibility, ease-of-use, and cost efficiency.

Despite providing cost-efficient DL solutions, outsourcing training to DLaaS service providers raises serious security and privacy concerns. One of the major issues is the *integrity* of the deep neural network (DNN) models trained by the

server. For example, due to Byzantine failures such as software bugs and network connection interruptions, the server may return a DNN model that does not reach its convergence. However, it is difficult for the client to verify the correctness of the returned DNN model easily due to the lack of hardware resources and/or DL expertise.

In this paper, we consider the *Infrastructure-as-a-Service* (IaaS) setting where the DLaaS service provider delivers the computing infrastructure including servers, network, operating systems, and storage as the services through virtualization technology. Typical examples of IaaS settings are Amazon Web Services[1] and Microsoft Azure[2]. In this setting, a client outsources his/her training data T to the DLaaS service provider (server). The client does not need to store T locally after it is outsourced (i.e., the client may not have access to T after outsourcing). The client also has the complete control of the infrastructure. He can customize the configuration of the DNN model M, including the network topology and hyperparameters of M. Then the server trains M on the outsourced T and returns the trained model M to the client. As the client lacks hardware resources and/or DL expertise, a third-party verifier will authenticate on behalf of the client if M returned by the server is *correct*, i.e., M is the same as being trained locally with T under the same configuration. Since the verifier may not be able to access the private training data owned by the client, our goal is to design a lightweight verification mechanism that enables the verifier to authenticate the correctness of M without full access to T.

A possible solution is that the *verifier* executes the training process independently. Since the verifier does not have the access to the client's private data, he has to execute training on the private data encrypted by homomorphic encryption (HE) [6,9]. Though correct, this solution can incur expensive overhead due to the high complexity of HE. Furthermore, since HE only supports polynomial functions, some activation functions (e.g., ReLU, Sigmoid, and Tanh) have to be approximated by low-degree polynomials when HE is used, and thus the verifier cannot compute the exact model updates. On the other hand, the existing works on verifying the integrity of DNNs (e.g., SafetyNets [5] and VeriDeep [8] hold a few restrictions on the activation function (e.g., it must be polynomials with integer coefficients) and data type of weights/inputs (e.g., they must be integers). We do not have any assumption on activation functions and input data types. Furthermore, these existing works have to access the original data, which is prohibited in our setting due to privacy protection.

Our contributions. We design VeriDL, a framework that supports efficient verification of outsourced DNN model training by a potentially untrusted DLaaS server which may return wrong DNN model as the result. VeriDL provides the *deterministic* correctness guarantee of remotely trained DNN models without any constraint on the activation function and the types of input data. The key idea

[1] Amazon Web Services: https://aws.amazon.com/
[2] Microsoft Azure: https://azure.microsoft.com/en-us/

of VeriDL is that the server constructs a cryptographic proof of the model updates, and sends the proof along with the model updates to the verifier. Since the proof aggregates the intermediate model updates (in compressed format) during training, the verifier can authenticate the correctness of the trained model by using the proof only. In particular, we make the following contributions. First, we design an efficient procedure to construct the cryptographic proof whose size is significantly smaller than the training data. The proof is constructed by using *bilinear pairing*, which is a cryptographic protocol commonly used for aggregate signatures. Second, we design a lightweight verification method named VeriDL that can authenticate the correctness of model updates through the cryptographic proof. By using the proof, VeriDL does not need access to the training data for correctness verification. Third, as the existing bilinear mapping methods cannot deal with the weights in DNNs that are decimal or negative values, we significantly extend the bilinear mapping protocol to handle decimal and negative values. We formally prove that VeriDL is secure against the attacker who may have full knowledge of the verification methods and thus try to escape from verification. Last but not least, we implement the prototype of VeriDL, deploy it on a DL system, and evaluate its performance on four real-world datasets that are of different data types (including non-structured images and structured tabular data). Our experimental results demonstrate the efficiency and effectiveness of VeriDL. The verification by VeriDL is faster than the existing DNN verification methods [6, 9] by more than three orders of magnitude.

2 Preliminaries

Bilinear mapping. Let G and G_T be two multiplicative cyclic groups of finite order p. Let g be a generator of G. A bilinear group mapping e is defined as $e : G \times G \rightarrow G_T$, which has the following property: $\forall a, b \in \mathbb{Z}_p$, $e(g^a, g^b) = e(g, g)^{ab}$. In the following discussions, we use the terms bilinear group mapping and bilinear mapping interchangeably. The main advantage of bilinear mapping is that determining whether $c \equiv ab \bmod p$ *without the access to* a, b and c can be achieved by checking whether $e(g^a, g^b) = e(g, g^c)$, by given g, g^a, g^b, g^c.

Outsourcing framework. We consider the outsourcing paradigm that involves three parties: (1) a *data owner* (client) $\mathcal{O}$ who holds a private training dataset T; (2) a third-party service provider (server) $\mathcal{S}$ who provides infrastructure services to $\mathcal{O}$; and (3) a third-party verifier $\mathcal{V}$ who authenticates the integrity of $\mathcal{S}$' services. In this paradigm, $\mathcal{O}$ outsources T to $\mathcal{S}$ for training of a DNN model M. Meanwhile $\mathcal{O}$ specifies the configuration of M on $\mathcal{S}$' infrastructure for training of M. After $\mathcal{S}$ finishes training of M, it sends M to $\mathcal{V}$ for verification. Due to privacy concerns, $\mathcal{V}$ cannot access the private training data T for verification.

Basic DNN operations. In this paper, we only focus on deep feedforward networks (DNNs), and leave more complicated structures like convolutional and recurrent networks for the future work. In this section, we present the basic operations of training a DNN model. We will explain in Section 4 how to verify the

output of these operations. In this paper, we only concentrate on fully-connected neural networks, and refrain from convolutional networks or recurrent networks. However, our design can be adapted to more advanced network structures.

A DNN consists of several layers, including the input layer (data samples), the output layer (the predicted labels), and a number of hidden layers. During the feedforward computation, for the neuron n_k^ℓ, its weighted sum z_k^ℓ is defined as:

$$z_k^\ell = \begin{cases} \sum_{i=1}^{m} x_i w_{ik}^\ell & \text{if } \ell = 1 \\ \sum_{j=1}^{d_{\ell-1}} a_j^{\ell-1} w_{jk}^\ell & \text{otherwise,} \end{cases} \tag{1}$$

where x_i is the i-th feature of the input $\vec{x}$, and d_i is the number of neurons on the i-th hidden layer. The activation a_k^ℓ is calculated as follows:

$$a_k^\ell = \sigma(z_k^\ell), \tag{2}$$

where σ is the activation function. We allow a broad class of activation functions such as sigmoid, ReLU (rectified linear unit), and hyperbolic tangent.

On the output layer, the output o is generated by following:

$$o = \sigma(z^o) = \sigma(\sum_{j=1}^{d_L} a_j^L w_j^o), \tag{3}$$

where w_j^o is the weight that connects n_j^ℓ to the output neuron.

In this paper, we mainly consider the mean square error (MSE) as the cost function. For any sample $(\vec{x}, y) \in T$, the cost $C(\vec{x}, y; W)$ is measured as the difference between the label y and the output o:

$$C(\vec{x}, y; W) = C(o, y) = \frac{1}{2}(y - o)^2. \tag{4}$$

Then the error E is calculated as the average error for all samples:

$$E = \frac{1}{N} \sum_{(\vec{x},y)\in T} C(\vec{x}, y; W). \tag{5}$$

In the backpropagation process, gradients are calculated to update the weights in the neural network. According to the chain rule of backpropagation [10], for any sample $(\vec{x}, y)$, the error signal δ^o on the output neuron is

$$\delta^o = \nabla_o C(o, y) \odot \sigma'(z^o) = (o - y)\sigma'(z^o). \tag{6}$$

While the error signal δ_k^ℓ at the ℓ-th hidden layer is

$$\delta_k^\ell = \begin{cases} \sigma'(z_k^\ell) w_k^o \delta^o & \text{if } \ell = L, \\ \sigma'(z_k^\ell) \sum_{j=1}^{d_{\ell+1}} w_{kj}^{\ell+1} \delta_j^{\ell+1} & \text{otherwise.} \end{cases} \tag{7}$$

The derivative for each weight w_{jk}^{ℓ} is computed as:

$$\frac{\partial C}{\partial w_{jk}^{\ell}} = \begin{cases} x_j \delta_k^{\ell} & \text{if } \ell = 1 \\ a_j^{\ell-1} \delta_k^{\ell} & \text{otherwise.} \end{cases} \tag{8}$$

Then the weight increment Δw_{jk}^{ℓ} is

$$\Delta w_{jk}^{\ell} = -\frac{\eta}{N} \sum_{(\vec{x},y)\in T} \frac{\partial C}{\partial w_{jk}^{\ell}}, \tag{9}$$

where η is the learning rate. Finally, the weight is updated as

$$w_{jk}^{\ell} = w_{jk}^{\ell} + \Delta w_{jk}^{\ell}. \tag{10}$$

The DNN is iteratively optimized by following the above feedforward and backpropagation process until it reaches convergence, $|E_1 - E_2| \leq \theta$, where E_1 and E_2 are the error/loss of two consecutive epochs in the optimization process, and θ is a small constant.

Verification protocol. We adapt the definition of the integrity verification protocol [11] to our setting:

Definition 21 (Deep Learning Verification Protocol) *Let W be the set of weight parameters in a DNN, and T be a collection of data samples. Let ΔW be the parameter update after training the DNN on T. The authentication protocol is a collection of the following four polynomial-time algorithms:* **genkey** *for key generation,* **setup** *for initial setup,* **certify** *for verification preparation, and* **verify** *for verification.*

- $\{s_k, p_k\} \leftarrow \mathbf{genkey}()$*: It outputs a pair of secret and public key;*
- $\{\gamma\} \leftarrow \mathbf{setup}(\mathbf{T}, \mathbf{s_k}, \mathbf{p_k})$*: Given the dataset T, the secret key s_k and the public key p_k, it returns a single signature γ of T;*
- $\{\pi\} \leftarrow \mathbf{certify}(\mathbf{T}, \mathbf{W_0}, \mathbf{\Delta W}, \mathbf{p_k})$*: Given the data collection T, the initial DNN model parameters W_0, the model update ΔW, and a public key p_k, it returns the proof π;*
- $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathbf{verify}(\mathbf{W_0}, \mathbf{\Delta W}, \pi, \gamma, \mathbf{p_k})$*: Given the initial DNN model parameters W_0, the model update ΔW, the proof π, the signature γ, and the public key p_k, it outputs either* accept *or* reject.

In this paper, we consider the adversary who has full knowledge of the authentication protocol. Next, we define the security of the authentication protocol against such adversary.

Definition 22 *Let* **Auth** *be an authentication scheme* {**genkey**, **setup**, **certify**, **verify**}*. Let* **Adv** *be a probabilistic polynomial-time adversary that is only given p_k and has unlimited access to all algorithms of* **Auth***. Then, given a DNN with*

initial parameters W_0 and a dataset T, **Adv** *returns a wrong model update $\Delta W'$ and a proof π': $\{\Delta W', \pi'\} \leftarrow \mathbf{Adv}(D, W_0, p_k)$. We say* **Auth** *is secure if for any p_k generated by the* **genkey** *routine, for any γ generated by the* **setup** *routine, and for any probabilistic polynomial-time adversary* **Adv**, *it holds that*

$$Pr(accept \leftarrow \mathbf{verify}(W_0, \Delta W', \pi', \gamma, p_k)) \leq negli(\lambda),$$

where $negli(\lambda)$ is a negligible function in the security parameter λ. Intuitively, **Auth** is secure if with negligible probability the incorrect model update can escape from verification.

3 Problem Statement

Threat model. In this paper, we consider the server $\mathcal{S}$ that may return incorrect trained model due to various reasons. For example, the learning process might be terminated before it reaches convergence due to the system's Byzantine failures (e.g., software bugs and network issues). $\mathcal{S}$ may also be incentivized to halt the training program early in order to save the computational cost and seek for a higher profit. Given the untrusted nature of the remote server, it is thus crucial for the client to verify the correctness of the returned DNN model before using the model for any decision-making task.

Problem statement. We consider the problem setting in which the data owner $\mathcal{O}$ outsources the training set T on the server. $\mathcal{O}$ also can specify the configuration of the DNN model M whose initial parameters are specified by W_0. The server $\mathcal{S}$ trains M until it reaches convergence (a local optima), and outputs the model update $\Delta W = f(T; W_0)$. However, with the presence of security threats, the model update ΔW returned by the server may not be a local optima. Therefore, our goal is to design an integrity verification protocol (Def. 21) that enables a third-party verifier $\mathcal{V}$ to verify if ΔW helps the model reach convergence without the access to the private training data.

4 Authentication Method

In this section, we explain the details of our authentication protocol. The **genkey** protocol is straightforward: the data owner $\mathcal{O}$ picks a pairing function e on two sufficiently large cyclic groups G and G_T of order p, a generator $g \in G$, and a secret key $s \in \mathbb{Z}_p$. Then it outputs a pair of secrete and public key (s_k, p_k), where $s_k = s$, and $p_k = \{g, G, G_T, e, v, H(\cdot)\}$, where $v = g^s \in G$, and $H(\cdot)$ is a hash function whose output domain is $\mathbb{Z}_p$. $\mathcal{O}$ keeps s_k private and distributes p_k to the other involved parties. In the following discussions, we only focus on the **setup**, **certify** and **verify** protocols.

Overview of our Approach. We design a verification method that only uses a short proof of the results for verification. Consider a data owner $\mathcal{O}$ that has

a private dataset T. Before transferring T to the server, $\mathcal{O}$ executes the *setup* protocol to generate a short signature γ of T, and disseminate γ to the verifier $\mathcal{V}$. $\mathcal{O}$ also sets up a DNN model M with initial weights W_0. Then $\mathcal{O}$ outsources M (with W_0) and the training dataset T to $\mathcal{S}$. After receiving T and M with its initial setup, the server $\mathcal{S}$ optimizes M and obtains the model updates ΔW. Besides returning ΔW to the verifier $\mathcal{V}$, $\mathcal{S}$ sends two errors E_1 and E_2, where E_1 is the error when the model reaches convergence as claimed (computed by Eqn. 5) and E_2 is the error by running an additional round of backpropagation and feedforward process after convergence. Furthermore, $\mathcal{S}$ follows the *certify* protocol and constructs a short cryptographic proof π of E_1 and E_2. The proof π includes: (1) the cryptographic digest π_T of the samples, and (2) the intermediate results of feedforward and backpropagation processes in computing E_1 and E_2. The verifier $\mathcal{V}$ then runs the *verify* protocol and checks the correctness of ΔW by the following three steps:

- *Authenticity verification of* π_T: $\mathcal{V}$ checks the integrity of π_T against the dataset signature γ that is signed by $\mathcal{O}$;
- *Authenticity verification of* E_1 *and* E_2: Without access to the private data T, $\mathcal{V}$ verifies if both errors E_1 and E_2 are computed honestly from T, by using π_T and the other components in the proof π;
- *Convergence verification*: $\mathcal{V}$ verifies if E_1 and E_2 satisfy the convergence condition (i.e., whether ΔW helps the model to reach convergence).

Next, we discuss the **Setup**, **Certify** and **Verify** protocols respectively. Then we discuss how to deal with decimal and negative weights.

4.1 Setup Protocol

Based on the public key, we define the following function for the data owner $\mathcal{O}$ to calculate a synopsis for each sample $(\vec{x}, y)$ in T. In particular,

$$d(\vec{x}, y)) = H(g^{x_1}||g^{x_2}||\ldots||g^{x_m}||g^y), \tag{11}$$

where $x_1, x_2, \ldots, x_m$ are the features, y is the label, and g is the group generator.

With the help the secret key s, $\mathcal{O}$ generates the signature γ for $(\vec{x}, y)$ with $\tau = d(\vec{x}, y))^s$. Then instead of sharing the large amount of signatures with the verifier, $\mathcal{O}$ creates an aggregated signature $\gamma = \pi_{i=1}^{n} \tau_i$, where τ_i is the signature for the i-th sample in the training data T. Then γ serves as a short signature of the whole dataset T.

4.2 Certify Protocol

To enable the verifier to verify E_1 and E_2 without access to the private samples $T = \{(\vec{x}, y)\}$, our Certify protocol construct a *proof* π as following: $\pi = \{\pi_E, \pi_W, \pi_T\}$, where

- $\pi_E = \{E_1, E_2\}$, i.e., π_E stores the errors of the model.

- $\pi_T = \{\{g^{x_i}\}, g^y | \forall(\vec{x}, y) \in T\}$, i.e., π_T stores the digest of original data $\{\vec{x}\}$ and $\{y\}$. Storing the digest but not the original data is to due to the privacy concern in the outsourcing setting (Sec 2).
- $\pi_W = \{\{\Delta w_{jk}^1\}, \{z_k^1\}, \{\hat{z}_k^1\}, g^{\delta^o}, \{\delta_k^L\}\}$, where Δw_{jk}^1 is the weight updated between the input and *first* hidden layer by one round of backpropagation after the model reaches convergence, z_k^1 and $\hat{z}_k^1$ are the weighted sum of the neuron n_k^1 (Eqn. 1) at convergence and one round after convergence respectively, δ^o and $\{\delta_k^L\}$ are the error signals at output and the last hidden layer at convergence respectively. Intuitively, π_W stores a subset of model outputs at the final two rounds (i.e., the round reaching convergence and one additional round afterwards).

4.3 Verify Protocol

The verification process consists of four steps: (1) authenticity verification of π_T; (2) one feedforward to verify the authenticity of E_1; (3) one backpropagation to update weights and another feedforward to verify the authenticity of E_2; and (4) verification of convergence, i.e. if $|E_1 - E_2| \leq \theta$, where θ is a pre-defined threshold for termination condition. Next, we discuss these steps in details.

Step 1. Verification of π_T: The verifier firstly verifies the authenticity of π_T, i.e., the digest of training samples. In particular, the verifier checks whether the following is true: $\Pi_{d(\vec{x},y)\in\pi_T} e((\vec{x}, y), v) \stackrel{?}{=} e(\gamma, g)$, where $d(\cdot)$ is the synopsis function (Eqn. (11)), $v = g^s$ is a part of the public key, γ is the aggregated signature provided by the data owner. If π_T passes the verification, $\mathcal{V}$ is assured that the digests in π_T are calculated from the intact dataset T.

Step 2. Verification of E_1: First, the verifier $\mathcal{V}$ verifies if the weighted sum $\{z_k^1\}$ at the final round is correctly computed. Note that $\mathcal{V}$ is aware of w_{ik}^1. $\mathcal{V}$ also obtains $\{g^{x_i}\}$ and $\{z_k^1\}$ from π_W in the proof. Then to verify the correctness of $\{z_k^1\}$, for each z_k^1, $\mathcal{V}$ checks if the following is true:

$$\Pi e(g^{x_i}, g^{w_{ik}^1}) \stackrel{?}{=} e(g, g)^{z_k^1}. \tag{12}$$

Once $\mathcal{V}$ verifies the correctness of $\{z_k^1\}$, it calculates the activation of the hidden layers and thus the output o (Eqns. (2) and (3)). Next, $\mathcal{V}$ checks if the following is true:

$$\Pi_{(\vec{x},y)\in D} e(g^{y-o}, g^{y-o}) \stackrel{?}{=} e(g, g)^{2NE_1}, \tag{13}$$

where $g^{y-o} = g^y * g^{-o}$. Note that g^y is included in the proof. $\mathcal{V}$ can compute g^{-o} by using o computed previously.

Step 3. Verification of E_2: This step consists of five-substeps. The first four substeps verify the correctness of weight increment in the backpropagation process, including the verification of error signal at the output layer, the verification of error signal at the last hidden layer, the verification of weight increments between all hidden layers, and verification of weight increments between the input

and the first hidden layer. The last substep is to verify the authenticity of E_2 based on the updated weights. Next, we discuss the details of these five substeps.

First, $\mathcal{V}$ verifies the correctness of g^{δ^o}. Following Eqn. (6), $\mathcal{V}$ can easily predict label y with δ^o. Therefore, π_W only includes g^{δ^o}. $\mathcal{V}$ verifies the following:

$$e(g^{-o}g^{y}, g^{-\sigma'(z^o)}) \stackrel{?}{=} e(g, g^{\delta^o}), \tag{14}$$

where g^{-o} and $g^{-\sigma'(z^o)}$ are computed by $\mathcal{V}$, and g^y and g^{δ^o} are from the proof.

Second, $\mathcal{V}$ verifies the correctness of δ_k^L (Eqn. (7)), i.e., the error signal on the k-th neuron on the last hidden layer, by checking if $e(g^{w_k^o \sigma'(z_k^L)}, g^{\delta^o}) \stackrel{?}{=} e(g,g)^{\delta_k^L}$, where $g^{w_{kj}^o \sigma'(z_k^L)}$ is computed by $\mathcal{V}$, and δ_k^L and g^{δ^o} are obtained from the proof.

Third, $\mathcal{V}$ calculates the error signal of other hidden layers by following Eqn. (7). Then with the knowledge of the activation on every hidden layer (by Step 2), $\mathcal{V}$ computes the derivatives of the weights (Eqn. 8) on the hidden layers to update the weights between consecutive hidden layers (Equations 9 - 10).

Fourth, $\mathcal{V}$ verifies the weight increment between input and the first hidden layer. We must note that $\mathcal{V}$ cannot compute $\frac{\partial C}{\partial w_{jk}^1}$ (Eqn. (8)) and Δw_{jk}^1 (Eqn. (9)) as it has no access to the input feature x_j. Thus $\mathcal{V}$ obtains Δw_{jk}^1 from the proof π and verifies its correctness by checking if the following is true:

$$\Pi_{(\vec{x},y)\in D} e(g^{x_j}, g^{\eta\delta_k^1}) \stackrel{?}{=} e(g^{\Delta w_{jk}^1}, g^{-N}). \tag{15}$$

Note that g^{x_j} and Δw_{jk}^1 are included in the proof, and $g^{\eta\delta_k^1}$ and g^{-N} are calculated by $\mathcal{V}$. After Δw_{jk}^1 is verified, $\mathcal{V}$ updates the weight by Eqn. (10). Finally, $\mathcal{V}$ verifies E_2 by following the same procedure of Step 2 on the updated weights.

Step 4. Verification of convergence: If E_1 and E_2 pass the authenticity verification, the verifier verifies the convergence of training by checking if $|E_1 - E_2| \leq \theta$, i.e., it reaches the termination condition.

We have the following theorem to show the security of VeriDL.

Theorem 1. *The authentication protocols of* VeriDL *is secure (Definition 22).*

We refer the readers to the extended version [3] of the paper for the detailed proof.

4.4 Dealing with Decimal & Negative Values

One weaknesses of bilinear pairing is that it cannot use decimal and negative values as the exponent in g^e. Therefore, the verification in Equations 12 - 15 cannot be performed easily. To address this problem, we extend the bilinear pairing protocol to handle decimal and negative values.

Decimal values. We design a new method that conducts decimal arithmetic in an integer field without accuracy loss. Consider the problem of checking if $b * c \stackrel{?}{=} e$, where b, c and e are three variables that may hold decimal values. Let L_T be the maximum number of bits after the decimal point allowed for any value. We define a new operator $f(\cdot)$ where $f(x) = x * 2^{L_T}$. Obviously, $f(x)$ must be an integer. We pick two cyclic groups G and G_T of sufficiently large order p such that $f(x)f(y) < Z_p$. Thus, we have $g^{f(x)} \in G$, and $e(g^{f(x)}, g^{f(y)}) \in G_T$. To make the verification in Eqn. (14) applicable with decimal values, we check if $e(g^{f(b)}, g^{f(c)}) \stackrel{?}{=} e(g, g)^{f(e)}$. Obviously, if $e(g^{f(b)}, g^{f(c)}) = e(g, g)^{f(e)}$, it is natural that $b * c = e$. The verification in Eqn. (12), (13) and (15) is accomplished in the same way, except that the involved values should be raised by 2^{L_T} times.

Negative values. Equations (12 - 15) check for a given pair of vectors $\vec{u}, \vec{v}$ of the same size, whether $\sum u_i v_i = z$. Note that the verification in Eqn. (14) can be viewed as a special form in which both $\vec{u}$ and $\vec{v}$ only include a single scalar value. Also note that u_i, v_i or z may hold negative values. Before we present our methods to deal with negative values, we first define an operator $[\cdot]$ such that $[x] = x \ mod \ p$. Without loss of generality, we assume that for any $\sum u_i v_i = z$, $-p < u_i, v_i, z < p$. We have the following lemma.

Lemma 2. *For any pair of vectors $\vec{u}, \vec{v}$ of the same size, and $z = \sum u_i v_i$, we have*

$$\Big[\sum [u_i][v_i]\Big] = \begin{cases} z & \text{if } z \geq 0 \\ z + p & \text{otherwise.} \end{cases}$$

We omit the proof of Lemma 2 here due to the limited space; the proof can be found in the extended version [3] of the paper. Following Lemma 2, we have Theorem 3 to verify vector dot product operation in case of negative values based on bilinear pairing.

Theorem 3. *To verify $\sum u_i v_i \stackrel{?}{=} z$, it is equivalent to checking if*

$$\Pi e(g^{[u_i]}, g^{[v_i]}) \stackrel{?}{=} \begin{cases} e(g, g)^z & \text{if } z \geq 0 \\ e(g, g)^{(z+p)} & \text{otherwise.} \end{cases} \tag{16}$$

We omit the proof due to the limited space, and include it in the extended version [3]. Next, we focus on Eqn. (12) and discuss our method to handle negative values. First, based on Lemma 2, we can see that for any x_i and w_{ik}^1, if $x_i w_{ik}^1 \geq 0$, then $[x_i][w_{ik}^1] = x_i w_{ik}^1$; otherwise, $[x_i][w_{ik}^1] = x_i w_{ik}^1 + p$. Therefore, to prove $z_k^1 = \sum x_i w_{ik}^1$, the server includes a flag $sign_i$ for each x_i in the proof, where

$$sign_i = \begin{cases} + & \text{if } x_i \geq 0 \\ - & \text{otherwise.} \end{cases}$$

Meanwhile, for each z_k^1, the server prepares two values $p_k^1 = \sum_{i:x_i w_{ik}^1 \geq 0} x_i w_{ik}^1$ and $n_k^1 = \sum_{i:x_i w_{ik}^1 < 0} x_i w_{ik}^1$, and includes them in the proof.

In the verification phase, since the client is aware of w_{ik}^1, with the knowledge of $sign_i$ in the proof, it can tell if $x_i w_{ik}^1 \geq 0$ or not. So the client first verifies if

$$\Pi_{i:x_i w_{ik}^1 \geq 0} e(g^{[x_i]}, g^{[w_{ik}^1]}) \stackrel{?}{=} e(g,g)^{p_k^1}, \Pi_{i:x_i w_{ik}^1 < 0} e(g^{[x_i]}, g^{[w_{ik}^1]}) \stackrel{?}{=} e(g,g)^{n_k^1 + p},$$

where $g^{[x_i]}$ is included in the proof, and $g^{[w_{ik}^1]}$ is computed by the client. Next, the client checks if $p_k^1 + n_k^1 \stackrel{?}{=} z_k^1$.

5 Experiments

5.1 Setup

Hardware & Platform. We implement VeriDL in C++. We use the implementation of bilinear mapping from PBC library[3]. The DNN model is implemented in Python on TensorFlow. We simulate the server on a computer of 2.10GHz CPU, 48 cores and 128GB RAM, and the data owner and the verifier on 2 computers of 2.7GHz Intel CPU and 8GB RAM respectively.

Datasets. We use the following four datasets that are of different data types: (1) **MNIST** dataset that contains 60,000 image samples and 784 features; (2) **TIMIT** dataset that contains 4,620 samples of broadband recordings and 100 features; (3) **ADULT** dataset that includes 45,222 records and 14 features; and (4) **HOSPITAL** dataset that contains 230,000 records and 33 features.

Neural network architecture. We train a DNN with four fully connected hidden layers for the MNIST, ADULT and HOSPITAL datasets. We vary the number of neurons on each hidden layer from 10 to 50, and the number of parameters from 20,000 to 100,000. We apply sigmoid function on each layer, except for the output layer, where we apply softmax function instead. We optimize the network by using gradient descent with the learning rate $\eta = 0.1$. By default, the minibatch size is 100. We use the same DNN structure for the TIMIT dataset with ReLU as the activation function.

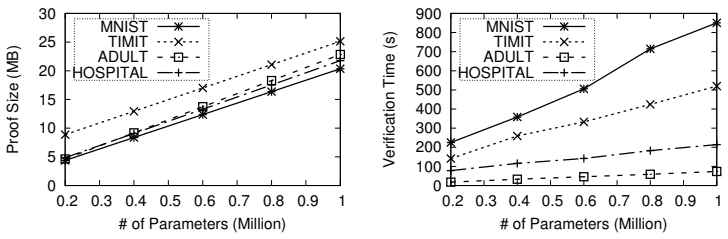
Basic and optimized versions of VeriDL. We implement two versions of VeriDL: (1) Basic approach (**B-VERIDL**): the proof of model updates is generated for every single input example $(\vec{x}, y)$; and (2) Optimized approach (**O-VERIDL**): the proof is generated for every unique value in the input $\{(\vec{x}, y)\}$.

Existing verification approaches for comparison. We compare the performance of VeriDL with two alternative approaches: (1) C_1**. Homomorphic encryption (LHE) vs. bilinear mapping:** When generating the proof, we use LHE to encrypt the plaintext values in the proof instead of bilinear mapping; (2) C_2**. Result verification vs. re-computation of model updates**

[3] https://crypto.stanford.edu/pbc/.

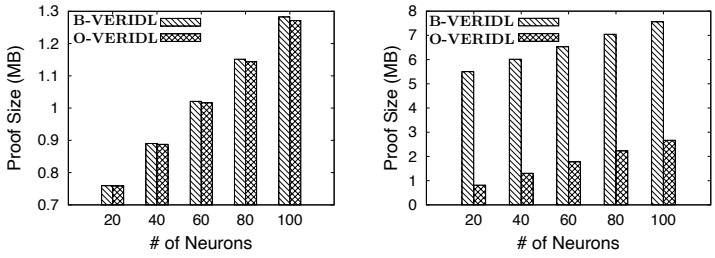
by privacy-preserving DL: The server encrypts the private input samples with homomorphic encryption. The verifeir executes the learning process on the encrypted training data, and compares the computed results with the server's returned updates. For both comparisons, we use three different implementations of HE. The first implementation is the Brakerski-Gentry-Vaikuntanathan (BGV) scheme provided by HElib library[4]. The second implementation is built upon the PALISADE library[5] that uses primitives of lattice-based cryptography for implementation of HE. The last one is built upon the Microsoft SEAL project [12], which provides a programming interface to lightweight homomorphic encryption.

5.2 Efficiency of VeriDL



(a) Proof size (b) Verification time

Fig. 1: Performance of VeriDL (minibatch size 100)

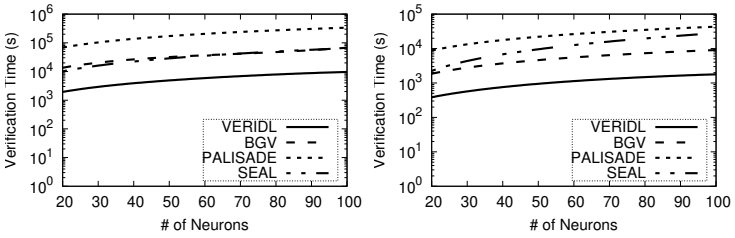


(a) MNIST dataset (b) TIMIT dataset

Fig. 2: Proof size

Proof size. The results of proof size of VeriDL on four datasets, with various number of neurons at each hidden layer, are shown in Figure 1 (a). In all settings, the proof size is small (never exceeds 25MB even with one million parameters).

[4] https://github.com/shaih/HElib.
[5] https://git.njit.edu/palisade/PALISADE/wikis/home



(a) MNIST dataset (b) TIMIT dataset

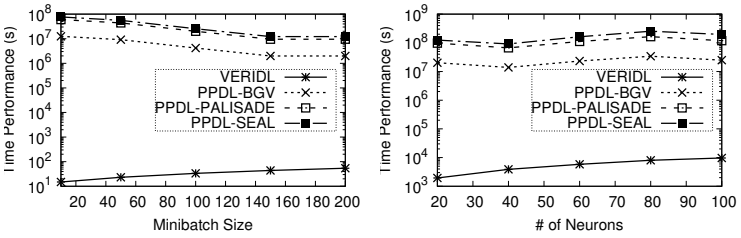
Fig. 3: Verification time (minibatch size 100)

This demonstrates the benefit of using bilinear pairing for proof construction. Second, we observe a linear increase in the proof size with the growth of the number of parameters. This is because the dominant components of proof size is the size of $\{\Delta w_{jk}^1\}$, $\{z_k^1\}$ and $\{\hat{z}_k^1\}$, which grows with the number of parameters.

Verification time. The results of verification time on all four datasets are shown in Figure 1 (b). First, the verification time is affordable even on the datasets of large sizes. Second, the verification time grows linearly with the number of hyperparameters. The reason is that the number of neurons on the first hidden layer increases linearly with the growth of parameters in the neuron network, while the verification time linearly depends on the input dimension and the number of neurons in the first hidden layer.

B-VeriDL VS. O-VeriDL. We compare the performance of the basic and optimized versions of VeriDL. Figure 2 demonstrates the proof size of B-VeriDL and O-VeriDL with various number of neurons at each hidden layer in the DNN model. In general, the proof size is small (less than 1.3MB and 8MB for MNIST and TIMIT datasets respectively). Furthermore, the proof size of O-VERIDL can be smaller than B-VERIDL; it is 20% - 26% of the size by B-VERIDL on TIMIT dataset. This demonstrates the advantage of O-VERIDL. The results also show that the proof size of both O-VERIDL and B-VERIDL gradually rises when the number of neurons increases. However, the growth is moderate. This shows that VeriDL can be scaled to large DNNs.

Comparison with existing approaches. We evaluate the verification time of different proof generation methods (defined by the comparison C_1 in Section 5.1) for various numbers of neurons on all four datasets, and report the results of MNIST and TIMIT datasets in Figures 3. The results on ADULT and HOSPITAL datasets are similar; we omit them due to the limited space. we observe that for all four datasets, VeriDL (using bilinear mapping) is more efficient than using HE (i.e., BGV, PALISADE and SEAL) in the proof. Thus bilinear mapping is a good choice as it enables the same function over ciphertext with cheaper cost. Besides, the time performance of both VeriDL and HE increases when the number of neurons in the network grows. This is expected as it takes more time to verify a more complex neural network. We also notice that all ap-



(a) Minibatch size (b) # of neurons

Fig. 4: Verification vs. re-computation of model updates

proaches take longer time on the MNIST dataset than the other datasets. This is because the MNIST dataset includes more features than the other datasets; it takes more time to complete the verification in Equations 12 - 15.

5.3 Verification vs. Re-computation of Model Updates

We perform the comparison C_2 (defined in Sec. 5.1) by implementing the three HE-based privacy-preserving deep learning (PPDL) approaches [6, 9, 12] and comparing the performance of VeriDL with them. To be consistent with [6, 9], we use the approximated ReLU as the activation function due to the fact that HE only supports low degree polynomials. Figure 4 shows the comparison results. In Figure 4 (a), we observe that VeriDL is faster than the three PPDL methods by more than three orders of magnitude. An interesting observation is that VeriDL and PPDL take opposite pattern of time performance when the minibatch size grows. The main reason is that when the minibatch size grows, VeriDL has to verify E_1 and E_2 from more input samples (thus takes longer time), while PPDL needs fewer epochs to reach convergence (thus takes less time). Figure 4 (b) shows the impact of the number of neurons on the time performance of both VeriDL and PPDL. Again, VeriDL wins the three PPDL methods by at least three orders of magnitude.

5.4 Robustness of Verification

To measure the robustness of VeriDL, we implement two types of server's misbehavior, namely *Byzantine failures* and *model compression attack*, and evaluate if VeriDL can catch the incorrect model updates by these misbehavior.

Byzantine failure. We simulate the Byzantine failure by randomly choosing 1% neurons and replacing the output of these neurons with random values. We generate three types of wrong model updates: (1) the server sends the wrong error E_1 with the proof constructed from correct E_1; (2) the server sends wrong E_1 with the proof constructed from wrong E_1; (3) the server sends correct E_1 and wrong E_2. Our empirical results demonstrate that VeriDL caught all wrong model updates by these Byzantine failures with 100% guarantee.

Model compression attack. The attack compresses a trained DNN network with small accuracy degradation [2,7]. To simulate the attack, we setup a fully-connected network with two hidden layers and sigmoid activation function. The model parameters are set by randomly generating 32-bits weights. We use ADULT dataset as the input. We simulate two types of model compression attacks: (1) the *low-precision floating points attack* that truncates the initial weights to 8-bits and 16-bits respectively and train the truncated weights; and (2) the *network pruning attack* that randomly selects 10% - 25% weights to drop out during training. For both attacks, we run 50 times and calculate the absolute difference between the error E_1' computed from the compressed model and the error E_1 of the correct model. From the results, we observe that the error difference produced by the low-precision attack is relatively high (with a 35% chance of larger than or equal to 0.02), and can be as large as 0.2. While the error differences of the network pruning attack are all between 0.002 and 0.01. In all cases, we have $|E_1' - E_1| \geq 10^{-9}$. We omit the results due to the limited space. We must note that given the DNN model is a 32-bit system, VeriDL can determine that $E_1' \neq E_1$ as long as $|E_1' - E_1| \geq 10^{-9}$. Therefore, VeriDL can detect the incorrect model updates by both network compression attacks, even though the attacker may forge the proof of E_1 to make E_1' pass the verification.

6 Related Work

Verified artificial intelligence (AI) [13] aims to design AI-based systems that are provably correct with respect to mathematically-specified requirements. SafetyNet [5] provides a protocol that verifies the execution of DL on an untrusted cloud. The verification protocol is built on top of the *interactive proof* (IP) protocol and arithmetic circuits. It places a few restrictions on DNNs, e.g., the activation functions must be polynomials with integer coefficients, which disables the activation functions that are commonly used in DNNs such as ReLU, sigmoid and softmax. Recent advances in zero-knowledge (ZK) proofs significantly reduce the verification and communication costs, and make the approach more practical to verify delegated computations in public [14]. ZEN [4] is the first ZK-based protocol that enables privacy-preserving and verifiable inferences for DNNs. However, ZEN only allows ReLU activation functions. We remove such strict assumption. *VeriDeep* [8] generates a few minimally transformed inputs named *sensitive samples* as fingerprints of DNN models. If the adversary makes changes to a small portion of the model parameters, the outputs of the sensitive samples from the model also change. However, VeriDeep only can provide a probabilistic correctness guarantee.

7 Conclusion and Future Work

In this paper, we design VeriDL, an authentication framework that supports efficient integrity verification of DNN models in the DLaaS paradigm. VeriDL

extends the existing bilinear grouping technique significantly to handle the verification over DNN models. The experiments demonstrate that VeriDL can verify the correctness of the model updates with cheap overhead.

While VeriDL provides a deterministic guarantee by verifying the output of *all* neurons in DNN, generating the proof for such verification is time costly. Thus an interesting direction to explore in the future is to design an alternative *probabilistic* verification method that provides high guarantee (e.g., with 95% certainty) but with much cheaper verification overhead.

References

1. Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.
2. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.
3. Boxiang Dong, Bo Zhang, and Hui (Wendy) Wang. Veridl: Integrity verification of outsourced deep learning services (extended version version). *arXiv preprint arXiv:2107.00495*, 2021.
4. Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: Efficient zero-knowledge proofs for neural networks. *IACR Cryptol. ePrint Arch.*, 2021:87, 2021.
5. Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems*, pages 4675–4684, 2017.
6. Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.
7. Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
8. Zecheng He, Tianwei Zhang, and Ruby B Lee. Verideep: Verifying integrity of deep neural networks through sensitive-sample fingerprinting. *arXiv preprint arXiv:1808.03277*, 2018.
9. Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.
10. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
11. Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *Annual Cryptology Conference*, pages 91–110, 2011.
12. Microsoft SEAL (release 3.5). `https://github.com/Microsoft/SEAL`, April 2020. Microsoft Research, Redmond, WA.
13. Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514*, 2016.
14. Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. *IACR Cryptol. ePrint Arch.*, 2021:76, 2021.