Yue Niu*, Ramy E. Ali, and Salman Avestimehr

3LegRace: Privacy-Preserving DNN Training over TEEs and GPUs

Abstract: Leveraging parallel hardware (e.g. GPUs) for deep neural network (DNN) training brings high computing performance. However, it raises data privacy concerns as GPUs lack a trusted environment to protect the data. Trusted execution environments (TEEs) have emerged as a promising solution to achieve privacy-preserving learning. Unfortunately, TEEs' limited computing power renders them not comparable to GPUs in performance. To improve the trade-off among privacy, computing performance, and model accuracy, we propose an *asymmetric* model decomposition framework, AsymML, to (1) accelerate training using parallel hardware; and (2) achieve a strong privacy guarantee using TEEs and differential privacy (DP) with much less accuracy compromised compared to DP-only methods. By exploiting the low-rank characteristics in training data and intermediate features, AsymML asymmetrically decomposes inputs and intermediate activations into low-rank and residual parts. With the decomposed data, the target DNN model is accordingly split into a *trusted* and an *untrusted* part. The trusted part performs computations on low-rank data, with low compute and memory costs. The untrusted part is fed with residuals perturbed by very small noise. Privacy, computing performance, and model accuracy are well managed by respectively delegating the trusted and the untrusted part to TEEs and GPUs. We provide a formal DP guarantee that demonstrates that, for the same privacy guarantee, combining asymmetric data decomposition and DP requires much smaller noise compared to solely using DP without decomposition. This improves the privacy-utility trade-off significantly compared to using only DP methods without decomposition. Furthermore, we present a rank bound analysis showing that the low-rank structure is preserved after each layer across the entire model. Our extensive evaluations on DNN models show that AsymML delivers 7.6× speedup in training compared to the TEE-only executions while ensuring privacy. We also demonstrate that AsymML is effective in protecting data under common attacks such as model inversion and gradient attacks.

Keywords: Privacy-Preserving Machine Learning, TEE

DOI 10.56553/popets-2022-0105

Received 2022-02-28; revised 2022-06-15; accepted 2022-06-16.

1 Introduction

Deep neural networks (DNNs) are acting as an essential building block in various applications such as computer vision (CV) [1, 2] and natural language processing (NLP) [3]. Efficiently training a DNN model usually requires a large training dataset and sufficient computing resources. In many real applications, datasets are locally collected and not allowed to be publicly accessible, while training is computation-intensive and hence usually offloaded to parallel hardware (e.g., GPUs). Considering that data transfer can be hacked or a runtime memory with sensitive data can be accessed by third parties, such practice poses serious privacy concerns.

The need for private data protection has motivated privacy-preserving machine learning methods such as machine learning with differential privacy (DP) [4, 5, 11] and machine learning using trusted execution environments (TEEs) [7, 33, 35].

DP-based methods usually defend against membership inference and model-inversion attacks [8, 9] that aim to infer or reconstruct the training data through the DNN models [4]. Specifically, by injecting noise to the gradients during training, DP-based methods reduce the correlation between the model parameters and the training data. Therefore, reconstructing training data through the model becomes more challenging. In addition to applying DP to the models, [10, 12] also apply DP directly to the input data during training. However, DP alone usually compromises too much accuracy to achieve strong privacy guarantees [25].

Unlike such DP methods, TEEs provide a direct hardware solution to protect data from any untrusted en-

***Corresponding Author: Yue Niu:** Electrical and Computer Engineering Department, University of Southern California, yueniu@usc.edu

Ramy E. Ali, Salman Avestimehr: Electrical and Computer Engineering Department, University of Southern California, reali, avestime@usc.edu

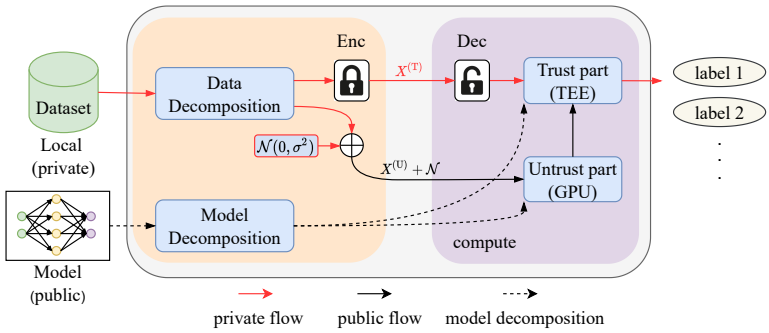


Fig. 1. An Overview of AsymML is depicted. In AsymML, the models are asymmetrically decomposed into trusted and untrusted parts. The trusted part performs computations on low-rank part $X^{(\mathrm{T})}$ at small compute/memory cost; while $X^{(\mathrm{U})}$ perturbed by small noise is offloaded onto the untrusted part with little critical information involved.

tities. TEE-based methods do not compromise accuracy as in DP. Trusted platforms such as Intel Software Guard Extensions (SGX) [6] and Arm TrustZone [14] create a sufficiently secure runtime environment, where sensitive data can be stored and processed. By performing all computations in TEEs, any untrusted access to internal memory is forbidden. However, the computing performance is severely affected when the TEEs are solely leveraged due to their limited storage and computation capabilities as a result of not having GPU support. Such executions are known as the TEE-only executions. A natural solution for this problem is to leverage both TEEs and the untrusted GPUs while protecting the privacy of the data. This idea was leveraged in [7] to develop a privacy-preserving framework known as Slalom for DNN inference. However, Slalom does not support DNN training, which is a more challenging and computationally-intensive task.

Contributions – To efficiently perform private training in a heterogeneous system with *trusted* TEE-enabled CPUs and fast *untrusted* accelerators, we propose a new training framework, AsymML, as shown in Figure 1. By exploiting the potential low-rank structure in the inputs and intermediate features X, AsymML first *asymmetrically* decomposes the inputs and the features into a low-rank part $X^{(\mathrm{T})}$ and a residual $X^{(\mathrm{U})}$. AsymML then accordingly decomposes the model into trusted and untrusted parts, in which the trusted part is fed with $X^{(\mathrm{T})}$, and the untrusted part is fed with $X^{(\mathrm{U})}$ perturbed by a very small noise. When the inputs and intermediate features have a low-rank structure, the trusted part with $X^{(\mathrm{T})}$ incurs small computation and memory costs, while the untrusted part handles most computations with little privacy revealed. The DNN model training is performed by respectively delegating the trusted and untrusted part to TEEs and GPUs, where TEEs protect the privacy, and GPUs guarantee the computing performance. We provide a formal DP guarantee that shows that, for the same privacy level, the asymmetric decomposition together with DP requires much smaller noise compared to solely leveraging DP methods. This implies that the decomposition improves the privacy-utility trade-off significantly. We also present a theoretical analysis showing that the low-rank structure is preserved after each layer in the DNN model, which ensures efficient asymmetric decomposition. In summary, our contributions are as follows.

1. We propose a privacy-preserving training framework that decomposes the data, the intermediate features and the models into two parts which decouples the information from the computations. The decomposition is based on a lightweight singular value decomposition (SVD) algorithm along with the Gaussian DP mechanism to protect the privacy further.
2. We provide essential theoretical analyses that show that a) AsymML achieves strong DP guarantee with a very small noise added compared to the case where there is no decomposition, and b) the low-rank structure in intermediate features is well-preserved after each layer in a DNN model.
3. We demonstrate AsymML is robust against common machine learning attacks, such as a strong model inversion attack that uses residual data as prior knowledge [40], and an attack that leverages the gradients [50].
4. We implement AsymML in a heterogeneous system with TEE-enabled CPUs and fast GPUs that supports various models. Our extensive experiments show that AsymML achieves up to 7.6× training speedup in VGG and 5.8× speedup in ResNet variants compared to the TEE-only executions.

Organization. The rest of this paper is organized as follows. In Section 2, we provide a brief overview of the closely-related works. We then present AsymML and its overhead analysis in Section 3. Our theoretical analyses on privacy and low-rank structure are provided in Section 4. In Section 5, we present several attack models. In Section 6, we provide extensive experiments to evaluate the performance, the overheads and the privacy guarantees of AsymML. Section 7 discusses the potential applications and limitations of AsymML. Finally, in Section 8, we discuss some concluding remarks.

2 Related works

Ensuring data privacy is a critical issue in offloading machine learning tasks to distributed and cloud-based systems. Many solutions have been developed to address such an issue, which falls into three main categories: differential privacy [4], crypto-based methods [23], and hardware solutions such as TEEs [6].

Differential privacy. Model training using differential privacy (DP) [4, 5, 11] aims to defend against membership inference [8] and model inversion attacks [9]. A well-known DNN training algorithm with DP is proposed in [4]. It follows a typical DP procedure and approximates the training objective function using a Gaussian noise mechanism. By injecting noise into the gradients during the training, DP reduces correlation between model parameters and training datasets to some extent. Therefore, with such a "noisy" model, it becomes more challenging to predict if a data record exists in the training dataset (membership inference attacks); or to directly reconstruct the training dataset through the trained model (model inversion attacks). However, DP usually greatly compromises accuracy to achieve strong privacy guarantees [25]. In addition to applying DP to models, [10, 12] directly add random noise to the input data, therefore hiding plain data from untrusted parties. However, the accuracy of these methods degrades significantly as we require stronger privacy guarantees.

Crypto-based methods. Machine learning with encrypted data has been recently investigated using various techniques such as homomorphic encryption (HE) [24, 29, 53], coded computing [20] and multi-party computing (MPC) [30, 31, 39]. These methods first encrypt the inputs, and then perform the computations in the encrypted domain. Despite their effectiveness in preserving privacy, these techniques still face many challenges. For example, machine learning with HE incurs much more computation costs compared to normal training flow using plain data. As a result, it is usually impracticable for most current DNN training or inference. On the other hand, coded computing [20] does not introduce noticeable compute costs, but requires a strong condition that a certain number of compute nodes cannot collude in a distributed system. Such an assumption may not hold in practice. Similar condition is also required in MPC settings. Moreover, current privacy-preserving coded computing approaches are limited to simple machine learning models such as logistic regression [20, 51].

Hardware solutions. In addition to algorithmic designs, recent works such as [32, 33, 35–37] have proposed privacy-preserving machine learning methods by leveraging trusted execution environments (TEEs) such as Intel SGX [6] and Arm TrustedZone [14]. These solutions keep private data in a trusted runtime environment that forbids any untrusted access and then perform training and inference inside this environment. Therefore, they usually offer strong privacy guarantees. However, TEE-based solutions usually degrade the computing performance due to the lack of GPU support, therefore causing significant training and inference slowdowns. To speed up computing, Graviton [34] prototypes a TEE fabric in a GPU platform so that both privacy and computing performance are achieved. On the other hand, [7, 38, 39] propose solutions that leverage both TEE-enabled CPUs and untrusted GPUs to perform inference. In Slalom [7], for instance, the inputs to a convolution layer are first masked with noise in TEEs to ensure privacy and then are sent to GPUs. When the convolutions are done, the noisy outputs are transferred back to TEEs and unmasked using precomputed results. However, Slalom only supports inference. To further support training in TEEs, methods such as DaKnight [38], and Goten [39] combine TEE with MPC techniques and require non-colluding compute nodes. As a result, they still fail to protect privacy if such an assumption does not hold.

In addition to the single-user offloading setting considered in our work, federated learning (FL) considers a collaborative learning setting with multiple clients and a central server aiming to learn a global model without data sharing [15]. In FL, each client trains a model using the local dataset, while after a certain number of local iterations, participating clients send their local models to a central parameter server. By aggregating local models from clients using algorithms such as FedAvg [26], the server obtains an updated global model and then

broadcasts to clients again in the next round. Such a procedure usually repeats many times until convergence is reached. During training with FL, sensitive datasets collected by each client are always stored in local memory, while the remote server can only access model parameters. Therefore, the datasets are protected as long as the local system is trusted. However, FL still faces many challenges in practice. For example, learning can be very difficult if data are highly heterogeneous [27]. Furthermore, FL cannot protect data against model inversion attacks. Therefore, FL still needs to be combined with methods such as DP [28] or secure aggregation [13] to ensure stronger data protection.

3 AsymML

In this section, we present AsymML. We start with the threat model considered in our work and then describe AsymML in detail, including data and model decomposition in DNNs through a lightweight SVD approximation. Finally, we provide the computation and the memory costs of AsymML.

Notations – We use lowercase letters for scalars and vectors, and uppercase letters for matrices and tensors. $\overline{X}$ denotes a 2D matrix flattened from a multidimensional tensor X, while $\left\|\overline{X}\right\|_F$ denotes the Frobenius norm of the matrix $\overline{X}$. $\overline{X}^*$ denotes the transpose of $\overline{X}$. X_i denotes i-th slice of a tensor $X \in \mathbb{R}^{N\times h\times w}$, $X_{i,:,:}$, while $X_{i,j}$ denotes (i,j)-th slice, $X_{i,j,:}$. We use $\circledast$ to denote convolution, and $\cdot$ for matrix multiplication. For a DNN model, given a loss function $\mathcal{L}$, $\nabla_W \mathcal{L}$ denotes gradients of the loss w.r.t parameters W. Finally, we use log to denote the logarithm to the base 2.

3.1 Threat model

Based on the capabilities of common TEE platforms such as Intel SGX [6], we consider the following threat model. 1) An adversary may compromise the OS where the TEE is running. However, it cannot breach the TEE environment, 2) the adversary may access hardware disk, runtime stack, memory outside TEEs and communication between trusted and untrusted environments, 3) the adversary may obtain the model parameters and the gradients during training and then analyze underlying relations with the training data, 4) the adversary may obtain data in untrusted environments and infer information in training data, and 5) the adversary may gain some knowledge of the training dataset (e.g., labels), and use public/online resources to improve its attack performance.
However, we do not consider side-channel attacks that compromise TEEs by probing physical signals such as power consumption and electromagnetic leaks.

3.2 Asymmetric data and model decomposition using SVD

AsymML decomposes compute-intensive and memory-intensive modules, especially convolutional layers in modern DNNs [1, 2], and then assigns each part to a suitable platform. At a high level, AsymML starts with decomposing a convolutional layer such that the computation involving privacy-sensitive information is performed in TEEs while the residual part is offloaded to GPUs. Specifically, the input of a convolutional layer denoted by X is decomposed into a low-rank part $X^{(\mathrm{T})}$ and a residual part $X^{(\mathrm{U})}$ as shown in Fig. 2. The residual part $X^{(\mathrm{U})}$ is then protected by adding a small noise as we discuss in Section 4 to ensure privacy.
When the convolutions in GPUs and TEEs are completed, outputs from GPUs denoted by $Y^{(\mathrm{U})}$ are merged into TEEs denoted by $Y^{(\mathrm{T})}$, and then followed by a non-linear layer (a pooling layer might be also needed). Outputs after a non-linear layer will be then re-decomposed before proceeding to the next convolution layer. During the whole forward/backward pass, the low-rank part in TEEs is never exposed, which effectively prevents crucial parts of data from being leaked.

We now explain the decomposition in detail. For a convolutional layer with input $X \in \mathbb{R}^{N\times h\times w}$ and kernels $W \in \mathbb{R}^{M\times N\times k\times k}$, where N and M are the number of input and output channels, and h, w and k are the size of inputs and kernels respectively, the i-th output channel is computed as follows[1]

$$Y_i = \mathrm{Conv}(X, W_i) = \sum_{j=1}^{N} X_j \circledast W_{i,j}. \tag{1}$$

By splitting the input X into $X^{(\mathrm{T})}$ and $X^{(\mathrm{U})}$, AsymML decomposes the convolution into a trusted convolution $\mathrm{Conv_T}$ and an untrusted convolution $\mathrm{Conv_U}$ as follows

$$Y_i = \mathrm{Conv_T}(X^{(\mathrm{T})}, W_i) + \mathrm{Conv_U}(X^{(\mathrm{U})}, W_i). \tag{2}$$

The *asymmetric* decomposition is inspired by an observation that channels in inputs and intermediate activa-

1 The batch size and the bias are omitted here for simplicity.

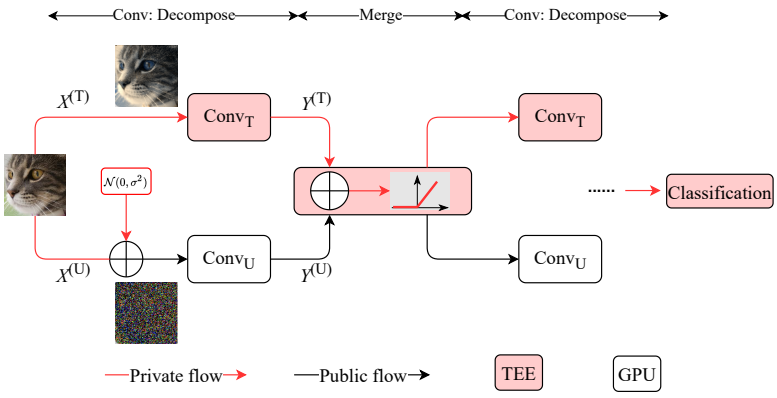


Fig. 2. An illustration of the model decomposition in AsymML is depicted. First, the input of a convolutional layer X is decomposed into a low-rank part $X^{(\mathrm{T})}$ and a residual part $X^{(\mathrm{U})}$ that is protected by Gaussian noise. The convolution of the trusted and the untrusted parts are then performed in TEEs and GPUs, respectively. Finally, the results are combined in TEEs followed by a non-linear layer and then the decomposition is performed again and so on. Therefore, training using AsymML behaves like a "three-legged race".

tions are usually highly correlated. By exploiting such channel correlations, X can be decomposed in a way that a low-rank tensor $X^{(\mathrm{T})}$ keeps most information, while $X^{(\mathrm{U})}$ stores the residuals. Therefore, the complexity of $\mathrm{Conv_T}$ is significantly reduced with little privacy compromised.

To extract a low-rank tensor $X^{(\mathrm{T})}$, we first apply singular value decomposition (SVD) to $\overline{X} \in \mathbb{R}^{N\times hw}$ flattened from X as

$$\overline{X} = U \cdot \mathrm{diag}(s) \cdot V^*, \tag{3}$$

where the *principal* channels are stored in V while the corresponding singular values in s denote the importance of each principal channel. The j-th channel in $X^{(\mathrm{T})}$ is obtained from the first $\mathcal{R}$ most principal channels as follows

$$X_j^{(\mathrm{T})} = \sum_{p=1}^{\mathcal{R}} s_p \cdot U(j,p) \cdot X_p^{'} \equiv \sum_{p=1}^{\mathcal{R}} a_{j,p} \cdot X_p^{'}, \tag{4}$$

where $X_p^{'} \in \mathbb{R}^{h\times w}$ is a 2D matrix reshaped from the p-th column of V and $a_{j,p} = s_p \cdot U(j,p)$. On the other hand, the residual part $X_j^{(\mathrm{U})}$ is given by

$$X_j^{(\mathrm{U})} = X_j - X_j^{(\mathrm{T})} = \sum_{p=\mathcal{R}+1}^{N} a_{j,p} \cdot X_p^{'}. \tag{5}$$

To further strengthen the privacy guarantee, $X^{(\mathrm{U})}$ is perturbed using a very small Gaussian noise as

$$\mathcal{M}(X^{(\mathrm{U})}) = X^{(\mathrm{U})} + Z \tag{6}$$

before being sent to the GPUs, where $Z_{i,j}$ are independent $\mathcal{N}(0,\sigma^2)$ random variables.

With the low-rank input $X^{(\mathrm{T})}$, the forward and backward passes of $\mathrm{Conv_T}$ can be reformulated in a way such that the complexity depends on the number of principal channels $\mathcal{R}$ as follows.

– **Forward.** During a forward pass, the outputs in TEEs are calculated as follows

$$\begin{aligned} Y_i^{(\mathrm{T})} &= \sum_{j=1}^{N} X_j^{(\mathrm{T})} \circledast W_{i,j} = \sum_{j=1}^{N}\sum_{p=1}^{\mathcal{R}} a_{j,p} X_p^{'} \circledast W_{i,j} \\ &= \sum_{p=1}^{\mathcal{R}} X_p^{'} \circledast \sum_{j=1}^{N} a_{j,p} W_{i,j} \equiv \sum_{p=1}^{\mathcal{R}} X_p^{'} \circledast W_{i,p}^{'}, \end{aligned} \tag{7}$$

where $W_{i,p}^{'} = \sum_{j=1}^{N} a_{j,p} W_{i,j}$. According to Eq. (7), $\mathrm{Conv_T}$ essentially is a convolution operation with $\mathcal{R}$ input channels, and the kernels $W^{'}$ that are obtained by regrouping W.

– **Backward.** During a backward pass, given gradient $\nabla_Y \mathcal{L}$, $\nabla^{(\mathrm{T})}_{W_{i,j}} \mathcal{L}$ in TEEs is computed as

$$\nabla^{(\mathrm{T})}_{W_{i,j}} \mathcal{L} = X_j^{(\mathrm{T})} \circledast \nabla_{Y_i} \mathcal{L} = \sum_{p=1}^{\mathcal{R}} a_{j,p} \cdot X_p^{'} \circledast \nabla_{Y_i} \mathcal{L}, \tag{8}$$

where $X_p^{'} \circledast \nabla_{Y_i} \mathcal{L}$ for $p = 1, 2, \cdots, \mathcal{R}$ are first computed. $\nabla^{(\mathrm{T})}_{W_{i,j}} \mathcal{L}$ is obtained by a simple linear transformation with $a_{j,p}$. Hence, the computation complexity of backward passes also depends on $\mathcal{R}$.

On the other hand, the untrusted forward output $Y_i^{(\mathrm{U})}$ and the backward gradient $\nabla^{(\mathrm{U})}_{W_{i,j}} \mathcal{L}$ in GPUs are the same as in the classical convolutional layers but with a different input $\mathcal{M}(X^{(\mathrm{U})})$. The final result Y_i and $\nabla_{W_{i,j}} \mathcal{L}$ are obtained by simply adding the results of the TEEs and the GPUs. In addition, as computing $\nabla_X \mathcal{L}$ does not involve the input X, it is offloaded onto GPUs.

3.3 Lightweight SVD approximation

Performing exact SVD on $X \in \mathbb{R}^{N\times h\times w}$ incurs a significant complexity of $O(Nh^2w^2 + N^2hw)$, which goes against our objective of reducing the complexity in TEEs. To reduce the complexity, we propose a lightweight SVD approximation that reduces complexity to only $O(\mathcal{R}Nhw)$.

SVD essentially is an algorithm that finds two vectors $\boldsymbol{u}^{(i)}$ and $\boldsymbol{v}^{(i)}$ to minimize $\left\|\overline{X}^{(i-1)} - \boldsymbol{u}^{(i)} \cdot \boldsymbol{v}^{(i)*}\right\|_F$, where $\overline{X}^{(i-1)}$ is the remaining $\overline{X}$ with $i-1$ most principal components extracted. $\left\{\boldsymbol{u}^{(i)}\right\}_{i=1}^{hw}$ and $\left\{\boldsymbol{v}^{(i)}\right\}_{i=1}^{N}$ are both orthogonal set of vectors. AsymML, however, does not require such orthogonality. Therefore, SVD can be then relaxed as

$$\begin{aligned} &\overline{X}^{(\mathrm{T})} = \underset{\overline{X}^{(\mathrm{T})}}{\arg\min} \quad \left\|\overline{X} - \overline{X}^{(\mathrm{T})}\right\|_F^2, \\ &\text{s.t. rank}\left(\overline{X}^{(\mathrm{T})}\right) \le \mathcal{R}, \quad \overline{X}^{(\mathrm{U})} = \overline{X} - \overline{X}^{(\mathrm{T})}. \end{aligned} \tag{9}$$

Using alternating optimization [17], each component i in $\overline{X}^{(\mathrm{T})}$ can be obtained as described in Algorithm 1[2]. Due to the fast convergence of alternating optimization, given suitable initial values (e.g. output channels from the previous ReLU/Pooling layer), we have experimentally observed that the maximum number of iterations max_iter to reach a near-optimal solution $\left\{\boldsymbol{u}^{(i)}\right\}_{i=1}^{\mathcal{R}}$ and $\left\{\boldsymbol{v}^{(i)}\right\}_{i=1}^{\mathcal{R}}$ is typically $1 \sim 2$ (See Appendix D). Finally, it is worth noting that the computation complexity of this algorithm is much less than exact SVD as it only increases linearly with $\mathcal{R}$.

Algorithm 1: Lightweight SVD Approx.

Data: $\mathcal{R}, \overline{X}, \left\{\boldsymbol{u}_0^{(i)}, \boldsymbol{v}_0^{(i)} \mid i = 1, \cdots, \mathcal{R}\right\}$, max_iter
Result: $\overline{X}^{(\mathrm{T})}, \overline{X}^{(\mathrm{U})}$
Initialize $\overline{X}^{(\mathrm{T})}$ as 0;
for i ***in*** $1, \cdots, \mathcal{R}$ **do**
  for j ***in*** $1, \cdots,$ max_iter **do**
    /* Alternating optimization */
    $\boldsymbol{u}_j^{(i)} = \frac{\overline{X} \cdot \boldsymbol{v}_{j-1}^{(i)}}{\left\|\boldsymbol{v}_{j-1}^{(i)}\right\|_F^2}$;
    $\boldsymbol{v}_j^{(i)} = \frac{\overline{X}^* \cdot \boldsymbol{u}_j^{(i)}}{\left\|\boldsymbol{u}_j^{(i)}\right\|_F^2}$;
  end
  $\overline{X}^{(\mathrm{T})} = \overline{X}^{(\mathrm{T})} + \boldsymbol{u}_j^{(i)} \cdot \boldsymbol{v}_j^{(i)*}$;
  $\overline{X} = \overline{X} - \boldsymbol{u}_j^{(i)} \cdot \boldsymbol{v}_j^{(i)*}$;
end
$\overline{X}^{(\mathrm{U})} = \overline{X}$;

3.4 Overhead analysis

The computation and memory costs in TEEs are of great concern as they decide AsymML's performance in a heterogeneous system. In this section, we break down these costs and compare the costs in TEEs and GPUs.

Figure 3 shows computation and memory costs in TEEs and GPUs for the case where $\mathcal{R}/N = 1/16$. As described in Section 3.2, the computation complexity of Conv_T in TEEs increases with the number of principal channels $\mathcal{R}$, while the complexity of Conv_U in GPUs is the same as that of the original convolutional layer. In addition to Conv_T, all element-wise operations (ReLU, Pooling, etc) and the lightweight SVD are performed in TEEs. As for the memory cost, the inputs and outputs of all element-wise operations are stored in TEEs, together with the inputs $X^{(\mathrm{T})}$ and the outputs of the convolution Conv_T, $Y^{(\mathrm{T})}$. On the other hand, inputs $\mathcal{M}(X^{(\mathrm{U})})$, outputs $Y^{(\mathrm{U})}$ of Conv_U and the corresponding gradients $\nabla_X\mathcal{L}$ and $\nabla_Y\mathcal{L}$ are stored in GPUs.

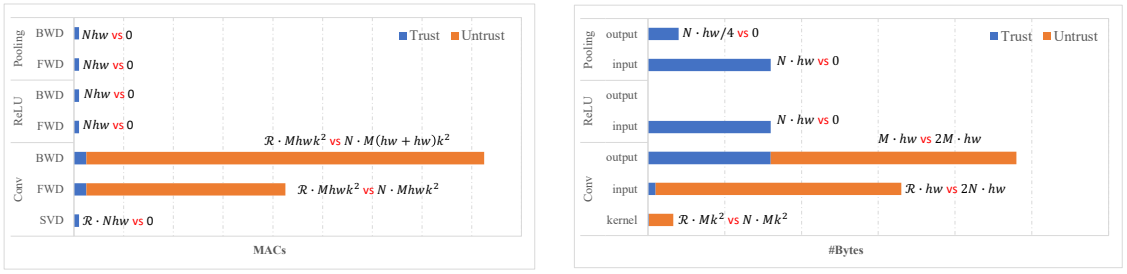
4 Theoretical guarantees

In this section, we first analyze the low-rank structure of the intermediate activations in NNs, and show that such a low-rank structure is preserved after linear operators such as convolutions in convolutional neural networks (CNNs). Then, we characterize the differential privacy guarantee of AsymML when the input $X^{(\mathrm{U})}$ is perturbed by a small Gaussian noise Z. We further demonstrate that AsymML requires much smaller noise for the privacy budget compared to directly adding noise to original inputs as in [12].

4.1 Low-rank structure in NNs

We first define a metric termed as *SVD-channel entropy* to formally quantify the low-rank structure in the intermediate features. Then, we show how the SVD-channel entropy changes in a DNN model after each layer. Inspired by SVD entropy [18] and Rényi entropy [19], we define SVD-channel entropy based on singu-

2 In the actual implementation, $\overline{X}^{(\mathrm{T})}$ is stored as a list of vectors $\left\{\boldsymbol{u}^{(i)}, \boldsymbol{v}^{(i)} \mid i = 1, \cdots, \mathcal{R}\right\}$, rather than as a matrix.



(a) Computation in TEEs and GPUs

(b) Memory in TEEs and GPUs

Fig. 3. The computation and the memory costs in TEEs (Trusted) and GPUs (Untrusted) ($\frac{\mathcal{R}}{N} = \frac{1}{16}$, pooling kernel size is 2)

lar values obtained in (3). Given the singular values $\{s_p(X)|\ p = 1, \cdots, N\}$ of an input matrix X with N channels, the SVD-channel entropy of X, denoted by μ_X, is defined as follows.

Definition 1. *(SVD-Channel Entropy). The SVD-channel entropy of a matrix X is given by*

$$\mu_X = -\log\left(\sum_{j=1}^{N} \bar{s}_j^2(X)\right), \tag{10}$$

where $\bar{s}_j(X) = \frac{s_j(X)}{\sum_{p=1}^{N} s_p(X)}$ is the j-th normalized singular value.

Next, we show in Lemma 1 and Theorem 1 that μ_X defined above indicates the number of *principal* channels actually needed to approximately reconstruct the original data X.

Lemma 1. *The SVD-channel entropy of an input X with N input channels is bounded as $0 \le \mu_X \le \log N$.*

If we use the first $\lceil 2^{\mu_X} \rceil$ most principal channels to reconstruct X, then Theorem 1 shows that such a reconstruction is sufficient to approximate X.

Theorem 1. *Given a matrix X with SVD-channel entropy μ_X, and assuming the j-th singular value is given as $s_j(X) = a \cdot b^{j-1}$, for some constants $a > 0$, $0 < b < 1$, if we use the $\mathcal{R} = \lceil 2^{\mu_X} \rceil$ most principal channels to reconstruct X, then we have $\frac{\sum_{j=1}^{\mathcal{R}} s_j^2(X)}{\sum_{j=1}^{N} s_j^2(X)} \ge 0.97$.*

Remark 1. The assumption in Theorem 1 that $s_j(X) = a \cdot b^{j-1}$ usually hold in natural images, where the data have highly correlated channels. In such cases, the singular values usually decay exponentially [21, 22].

The proofs of Lemma 1 and Theorem 1 are provided in Appendix A. In the rest of the paper, we regard $\mathcal{R}$ as the sufficient number of channels to represent X.

SVD-channel entropy in CNNs – In DNNs, given input data with low-rank structure, it is crucial to measure how such structure (measured using SVD-channel entropy) changes after every layer so that we can systematically set the number of principal channels in TEEs. In CNNs, convolutional, batch normalization, pooling and non-linear layers such as ReLU are the basic layers. In this section, we mainly analyze how these basic operators change the structure of the data. All proofs are provided in Appendix A.

We now start with the convolutional layers.

Convolutional layer – For a convolutional layer with input $X \in R^{N \times h \times w}$, we first bound the SVD-channel entropy of the outputs with 1×1 kernels in Theorem 2 and then we extend this to the general case of $k \times k$ kernels in Theorem 3.

Theorem 2. *For a convolution layer, given input $X \in R^{N \times h \times w}$ with SVD-channel entropy μ_X, kernel $W \in R^{M \times N \times 1 \times 1}$, then the SVD-channel entropy of the output $Y \in R^{M \times h' \times w'}$ is upper-bounded as follows*

$$\mu_Y \le \log \lceil 2^{\mu_X} \rceil.$$

Theorem 2 implies that low-rank structure is still preserved in the outputs for convolutional layers with 1×1 kernels. Therefore, before a convolutional layer, if $\lceil 2^{\mu} \rceil$ principal channels are used in TEEs, the same number of channels is still sufficient to approximate the outputs.

For a convolutional layer with a $k \times k$ kernel, it can be viewed as a 1×1 convolutional layer with k^2 different "input channels". Each "input channel" is a patch of X_j, denoted as $\left\{\hat{X}_{j,q,r} | 1 \le q, r \le k\right\}$, as shown in Figure 4. Given $X \in R^{N \times h \times w}$, $W \in R^{M \times N \times k \times k}$, the i-th output

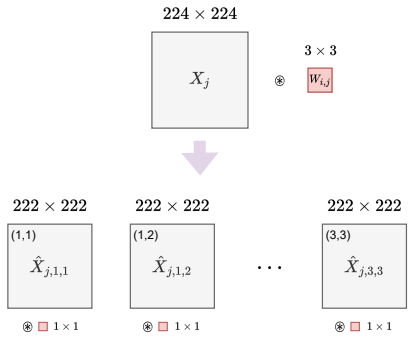


Fig. 4. An illustration of the conversion from $k \times k$ convolution to 1×1 convolutions with k^2 different input channels.

channel can be rewritten as follows

$$Y_i = \sum_{j=1}^{N} W_{i,j} \circledast X_j = \sum_{j=1}^{N} \sum_{q=1,r=1}^{k,k} W_{i,j,q,r} \circledast \hat{X}_{j,q,r}, \quad (11)$$

where $W_{i,j,q,r}$ is a 1×1 kernel after this conversion. Therefore, a $k \times k$ convolution with N input channels is equivalent to a 1×1 convolution with $N \cdot k^2$ "input channels": $\left\{\hat{X}_{j,q,r} | 1 \le j \le N, 1 \le q, r \le k\right\}$. To simplify the notation, we denote $\hat{X}_{:,q,r}$ as the (q,r)-th patch of all channels, and $\hat{X}_{j,:,:}$ as all patches of the j-th channel.

We now show in Theorem 3 that, knowing the SVD-channel entropy of $\hat{X}_{j,:,:}$ for $j = 1, \cdots, N$, the SVD-channel entropy of the outputs with $k \times k$ convolution can be accordingly bounded.

Theorem 3. *Given data X with SVD-channel entropy μ_X, $\mathcal{R} = \lceil 2^{\mu_X} \rceil$, $\mu_{\hat{X}_{j,:,:}} = \hat{\mu}_j$ for $1 \le j \le N$ and WLOG suppose that $\hat{\mu}_1 \le \hat{\mu}_2 \le \cdots \le \hat{\mu}_N$, then the SVD-channel entropy of $\hat{X}$ satisfies*

$$\mu_{\hat{X}} \le \log(\sum_{j=1}^{\mathcal{R}} \lceil 2^{\hat{\mu}_j} \rceil) \cong \mu_X + \bar{\mu},$$

where $\bar{\mu} = \frac{\sum_{j=1}^{\mathcal{R}} \hat{\mu}_j}{\mathcal{R}}$. Furthermore, the SVD-channel entropy of the output Y after convolution with $W \in R^{M \times N \times k \times k}$ is upper-bounded as

$$\mu_Y \le \log(\textstyle\sum_{j=1}^{\mathcal{R}} \lceil 2^{\hat{\mu}_j} \rceil)$$

Based on Theorem 3, with $k \times k$ kernels, the SVD-channel entropy of the outputs increases with $\mu_{\hat{X}_{j,:,:}}$ for $j = 1, \cdots, \mathcal{R}$. Intuitively, if k is larger, more patches are included, then $\mu_{\hat{X}_{j,:,:}}$ will be higher. More numerical analyses are presented in Section 6.2.

BatchNorm layer – Batch normalization is commonly used in CV models to reduce the *internal covariate shift*. According to Theorem 4, the SVD-channel entropy of outputs is almost the same as that of inputs in a BatchNorm layer.

Theorem 4. *Given data X with SVD-channel entropy μ_X, then the SVD-channel entropy of output Y after a batch normalization layer is upper-bounded as*

$$\mu_Y \le \log(\lceil 2^{\mu_X} \rceil + 1).$$

Hence, the low-rank structure of inputs is still preserved after batch normalization.

ReLU layer – As a non-linear layer, ReLU is the most commonly used operator in DNNs. Theoretically bounding the SVD-channel entropy after ReLU is infeasible. Instead, we empirically measure the SVD-channel entropy after a ReLU in Appendix C and show that the ReLU layers also preserve the low-rank structure.

Pooling layer – Pooling operations include max and average pooling. Max pooling is a also non-linear operator. Similarly, we provide an empirical experiment that shows that a max pooling layer does not greatly change the low-rank structure either (See Appendix C). Besides, the max and the average pooling layers usually deliver similar performance. As stated in Theorem 5, the SVD-channel entropy after an average pooling layer is less than the one before pooling. Therefore, pooling layers still preserve the low-rank structure.

Theorem 5. *For an average pooling layer, given input $X \in R^{N \times h \times w}$ with SVD-channel entropy μ_X, the SVD-channel entropy in output Y satisfies*

$$\mu_Y \le \log \lceil 2^{\mu_X} \rceil .$$

4.2 Privacy guarantees

We now provide the DP guarantee of AsymML. AsymML adds a small Gaussian noise to $X^{(\mathrm{U})}$ to ensure privacy. While the data stored in TEEs are protected by a secure hardware, the Gaussian mechanism further protects the information in the GPUs.

First, for a dataset $\mathcal{X} = \left\{X^1, X^2, \cdots, X^i, \cdots, X^B\right\}$, we define the neighboring dataset $\mathcal{X}' = \left\{X^1, X^2, \cdots, 0, \cdots, X^B\right\}$ for any possible i-th record removed [4]. During training, we sample a batch $\mathcal{S} \in \mathcal{X}$ with size b. The ℓ_2-sensitivity of the lightweight SVD is given by $\Delta_2 = \sup_{\mathcal{X},\mathcal{X}'} \left\| \mathcal{X}^{(U)} - \mathcal{X}'^{(U)} \right\| \le \xi \sup_i \left\| X^i \right\|$, where ξ is the upper bound of the ratio between the Frobenius norms of $X^{(\mathrm{U})}$ and X. Theorem 6 states that AsymML is (ϵ, δ)-differentially private when a Gaussian noise $\mathcal{N}(0, 2\Delta_2^2 \ln(1.25q/\delta)/\epsilon^2 . I)$ is added, where

$q = b/B$ is the sampling probability and I is the identity matrix.

Theorem 6. *AsymML is (ϵ, δ)-differentially private when a Gaussian noise $\mathcal{N}(0, 2\Delta_2^2 \ln(1.25q/\delta)/\epsilon^2.I)$ is added to the residuals, for $0 < \epsilon, 0 < \delta \leq q$, where b is the batch size, $q = b/B$ is the sampling probability and $\Delta_2 \leq \xi \sup_i \left\|X^i\right\|$ with SVD.*

Remark 2. *Compared to directly adding noise to the original data X, AsymML reduces the required noise variance by ξ^2. Therefore, AsymML significantly improves the utility-privacy trade-off (See also the experiments of Section 6.3).*

5 Attack models

Attack models are a crucial part in evaluating a privacy-preserving NN training/inference algorithm. An attacker usually uses any relevant information available such as model parameters, gradients, and any public prior information to reveal the private information being protected. In NNs, attacks reconstructing training data using the model parameters or the gradients are one of the strongest attack methods. The reason for the success of such attacks arises from leveraging the correlations between training data and model parameters as well as gradients during the training.

In this section, we consider two attacks that aim to reconstruct the target training dataset. The first attack is a model inversion (MI) attack [40] that uses *well-trained* models and some prior knowledge (noisy residual data $\mathcal{M}(\mathcal{X}^{(U)})$ in our case) to find synthetic images similar to images in the target training dataset. The second attack is a gradient inversion attack [50]. It finds synthetic data that can result in similar gradients as the original training dataset. While a model inversion attack is mainly targeting a well-trained model, a gradient inversion attack can happen at any stage, especially for an initial model or a pre-trained model [50]. In addition, it is worth noting that common membership inference attacks that leverage output probabilities [8] fail in AsymML since the logits are secured in TEEs.

Notations. We use $\mathcal{X}_\mathrm{t}$ to denote the target training dataset, and, $\mathcal{Z}_\mathrm{p}$ to denote the prior knowledge known to the attacker, including some public dataset $\mathcal{X}_\mathrm{p}$ and the residuals $\mathcal{M}(\mathcal{X}^{(U)})$ available in untrusted platforms. Note that $\mathcal{X}_\mathrm{p}$ does not overlap with $\mathcal{X}_\mathrm{t}$, but it may contain objects with similar labels. Finally, $\mathcal{M}_\mathrm{t}$ represents the target model under attack.

5.1 Model inversion attack

We now consider the model inversion attack.

Assumptions – We assume that the attacker has access to the target model $\mathcal{M}_\mathrm{t}$. Further, the attacker uses the noisy residuals $\mathcal{M}(\mathcal{X}^{(U)})$ as prior information to help reconstruct the training dataset $\mathcal{X}_\mathrm{t}$. In addition, the attacker knows general information such as label information. Therefore, it can find relevant resources (e.g. online images) to learn the target data distribution.

We design the attacker as shown in Fig. 5a and Fig. 5b. Specifically, the attack has two stages as follows.

1. **Prior knowledge distillation**. In this stage, the attacker trains a generative adversarial network (GAN) with the public prior knowledge $\mathcal{Z}_\mathrm{p}$.
   Fig. 5b shows the generator G architecture. It consists of two branches. The first branch B1 is used for learning features from the residual noisy data $\mathcal{M}(\mathcal{X}^{(U)})$. The second branch B2 is used for generating latent features. On the other hand, the discriminator D is a classical CNN. The detailed architectures are provided in Appendix B. The Wasserstein-GAN loss function is used during training as follows

$$\min_G \max_D L(G, D) = E_{x \in \mathcal{X}_\mathrm{p}}[D(x)] - E_z[D(G(z, \mathcal{M}(\mathcal{X}^{(U)})))]. \quad (12)$$

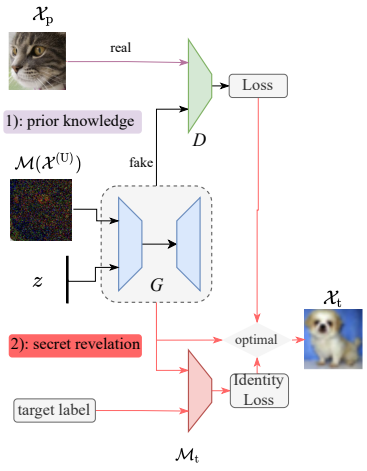
2. **Secret revelation**. In this stage, with the noisy residual data $\mathcal{M}(\mathcal{X}^{(U)})$ from the target dataset $\mathcal{X}_\mathrm{t}$ along with the labels, the attacker uses the generator to reconstruct images that achieve the highest accuracy in the target model $\mathcal{M}_\mathrm{t}$.
   We find the optimal latent vector z that generates an almost-real image to the discriminator, while achieving the lowest identity loss in the target model $\mathcal{M}_\mathrm{t}$, namely

$$\hat{z} = \arg\min_z L_\mathrm{prior} + \lambda L_\mathrm{id}, \quad (13)$$

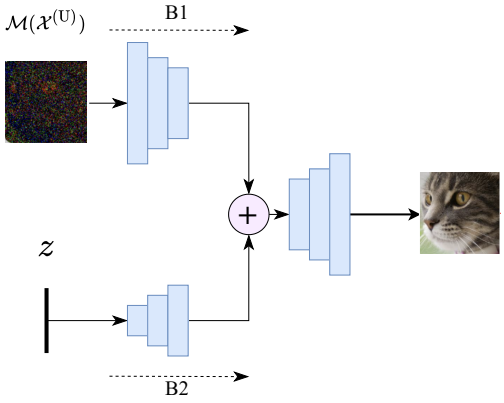
   where L_prior is the loss in the discriminator, L_id is the loss in the target model, and λ controls the weight between L_prior and L_id. These losses are given by

$$L_\mathrm{prior} = -D(G(z)), \quad L_\mathrm{id} = -\log[\mathcal{M}_t(G(z))], \quad (14)$$

   where $\mathcal{M}_t(G(z))$ is the output probability from the target model.



(a) An illustration of the model inversion attack using the residual data $X^{(U)}$ and any public relevant data as prior information. The attacker first trains a GAN model using prior knowledge (e.g., $\mathcal{M}(\mathcal{X}^{(U)})$). Then, with the residual data $\mathcal{M}(\mathcal{X}^{(U)})$ from the target training dataset along with the labels, the attacker uses the generator G to reconstruct images that achieve the highest accuracy in the target model $\mathcal{M}_t$



(b) An illustration of the generator architecture that uses residual data $\mathcal{M}(\mathcal{X}^{(U)})$ to reconstruct images. The generator consists of two branches: B1 for extracting features from residual data $\mathcal{M}(\mathcal{X}^{(U)})$, and B2 for generating latent features. Then, these two branches are merged and further up-sampled using deconvolution to generate outputs that have similar distribution as the training dataset.

Fig. 5. Model inversion attack using $\mathcal{M}(\mathcal{X}^{(U)})$ as prior knowledge.

5.2 Gradient inversion attack

Next, we consider a gradient version attack.

Assumptions – We assume the attacker has access to the gradients in the untrusted GPUs, $\nabla^{(U)}_{W_{i,j}}\mathcal{L}$, as well as the gradients on the inputs $\nabla_X\mathcal{L}$. Similar to the model inversion attack, the attacker can use $\mathcal{M}(\mathcal{X}^{(U)})$ as prior knowledge to help reconstruct synthetic images with similar gradients as the target dataset.

Figure 6 shows the procedures used in conducting a gradient inversion attack. Knowing the gradients $\nabla^{(U)}_W\mathcal{L}$ and $\nabla_X\mathcal{L}$ from target images, the attack first feeds $\mathcal{M}_t$ with the noisy residual data $\mathcal{M}(\mathcal{X}^{(U)})$. After obtaining the gradients $\nabla^{(U)}_W\mathcal{L}', \nabla_X\mathcal{L}'$, the ℓ_2 distance between the gradients is computed as follows

$$L = \left\|\nabla^{(U)}_W\mathcal{L} - \nabla^{(U)}_W\mathcal{L}'\right\|^2 + \lambda\left\|\nabla_X\mathcal{L} - \nabla_X\mathcal{L}'\right\|^2. \quad (15)$$

Based on L, the attacker computes the gradients on $\mathcal{M}(X^{(U)})$, and optimizes inputs using a common gradient descent method.

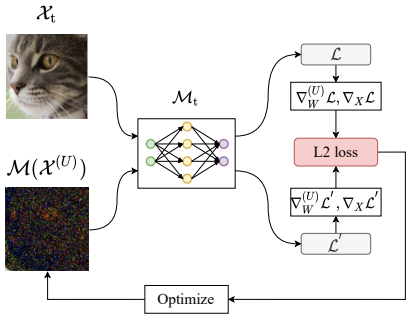


Fig. 6. An illustration of the gradient inversion attack that generates synthetic images with similar gradients as the target dataset.

6 Empirical evaluation

In this section, we evaluate AsymML in terms of the training accuracy, running time, robustness against attacks and information leakage. We perform our experiments on the following models and datasets.

For models, we consider VGG-16, VGG-19 [1], ResNet-18 and ResNet-34 [2]. For datasets, we consider the CIFAR-10 [41] and the ImageNet [42] datasets.

We illustrate the implementation of AsymML in detail in Section 6.1. Then, in Section 6.2, we study the effect of the kernel size on the SVD-channel entropy. In Section 6.3, we study the training accuracy of AsymML. In Section 6.4, we compare between AsymML and the baselines in terms of the running time. Finally, we present the privacy guarantee of AsymML and its capability under model inversion and gradient inversion attacks in Section 6.5.

6.1 AsymML implementation

We implement AsymML in a heterogeneous system as shown in Figure 7 with an Intel SGX enabled Xeon

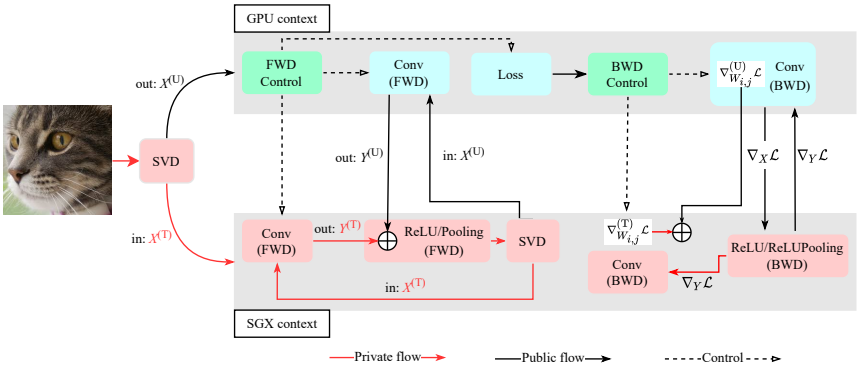


Fig. 7. Implementation of AsymML in a heterogeneous system with SGX enabled CPUs and GPUs. The forward and the backward passes in convolutional layers are coordinated by FWD and BWD Control, respectively.

CPUs [6] and NVIDIA RTX5000 GPUs working as untrusted accelerators. AsymML first reads a predefined model (e.g. from PyTorch [43]), decomposes it into trusted and untrusted parts, and then offloads them to TEEs and GPUs respectively. The operations in TEEs (e.g. Conv_T, ReLU, and Pooling) are supported by dedicated trusted functions. In order to reduce the CPU-GPU communications, a Pooling is fused with the preceding ReLU layer. Similar optimization also is applied to convolutional layers and the following batch normalization layers.

During training, SGX and GPU contexts are created for the trusted and the untrusted operations. We use PyTorch as a high-level coordinator to distribute the computations, activate GPU/SGX context, compute loss, and update the model parameters (See the forward and backward control in Figure 7).

During a forward pass, outputs of a convolutional layer in GPUs and TEEs will be merged in the following ReLU (and Pooling) layer in TEEs. A lightweight SVD is then applied to decompose the output activations into low-rank and residual parts, which are then fed into the next convolution layer.

During a backward pass, computing $\nabla_X \mathcal{L}$ for ReLU and Pooling layers is performed in TEEs, while computing $\nabla_X \mathcal{L}$ for convolutional layers is performed in GPUs. The partial gradients $\nabla^{(\text{T})}_{W_{i,j}} \mathcal{L}$ and $\nabla^{(\text{U})}_{W_{i,j}} \mathcal{L}$ in a convolutional layer are computed in GPUs and TEEs respectively and merged into TEEs.

6.2 SVD-channel entropy vs kernel size

In this subsection, we investigate the effect of the kernel size on the SVD-channel entropy. As Theorem 3 states, the upper bound of the SVD-channel entropy of outputs in a convolutional layer increases with the kernel size. More patches are generated with large kernels in each channel (See Fig. 4), therefore this leads to increasing SVD-channel entropy along the patches ($\mu_{\hat{X}_{j,:,:}}$).

We present an empirical result on the SVD-channel entropy $\mu_{\hat{X}_{j,:,:}}$ with kernel sizes ranging from 1 to 11. In this experiment, the input data X are randomly sampled from ImageNet. As detailed in Section 4.1, for $k \times k$ kernels, k^2 input patches are generated. We compute the SVD-channel entropy along all patches. Fig. 8 shows the SVD-channel entropy across 10K randomly sampled images with different kernel size. The line plot indicates the mean value, while the violin plots show the distribution. With 3×3 kernels, the SVD-channel entropy increases by around 1, which implies that given $\mathcal{R}$ principal channels in inputs, $2\mathcal{R}$ principal channels are sufficient for outputs after a convolution layer with 3×3 kernel. Although it is noted that the SVD-channel entropy increases with the kernel size, small kernel sizes are commonly used in modern DNNs such as VGG and ResNet. Therefore, the SVD-channel entropy increment is well-managed.

6.3 Training accuracy

Next, we investigate the training performance. In the experiments, we follow standard data preprocessing procedure. The training hyperparameters are as follows: weight decay is 0.0005, momentum is 0.9, the maximum number of epochs is 200 for CIFAR-10 and 100 for ImageNet. The initial learning rate (lr) is 0.1. We use the cosine annealing learning rate decay strategy.

We consider the following four schemes.

1. **Training with low-rank data $X^{(\text{T})}$(Low-Rank)**. In Low-Rank, each convolutional layer is only fed with the low-rank part $X^{(\text{T})}$. Other operations are the same as the original model. Based on

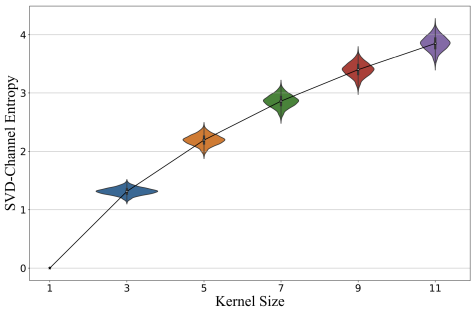


Fig. 8. The SVD-channel entropy along patches of a images is shown. The line plot shows the mean value, while the violin plot shows the distribution. A wide violin indicates large number of images are around a particular SVD-channel entropy level.

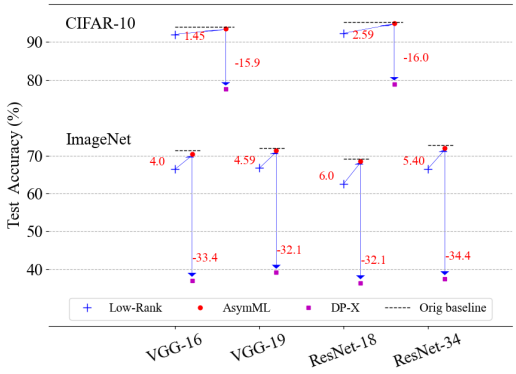


Fig. 9. The accuracy after training with only the low-rank part and original data is shown for CIFAR-10 and ImageNet. The low-rank training achieves good accuracy, but not is sufficient compared to the original models. Hence, the residual data $X^{(U)}$ is an indispensable component to filling this gap.

the theoretical analysis in Section 4.1 and the empirical evidence in Section 6.2, we compute $X^{(T)}$ as follows: for the first convolutional layer, $X^{(T)}$ only consists of the most principle channel ($\mathcal{R} = 1$); $\mathcal{R}$ is doubled when another convolutional layer (VGG) or a residual block (ResNet) is added, while $\mathcal{R}$ remains unchanged for a ReLU or a pooling layer.

2. **Training with noise added to the original data X (DP-X).** In DP-X and AsymML, we set $(\epsilon, \delta) = (1, 10^{-5})$. According to Theorem 6, we generate noise with the parameters given in Table 1.
3. **Training with noise added to the residuals $X^{(U)}$ (AsymML).** In AsymML, we add noise to the residual parts obtained by the light-weight SVD.
4. **Training with the original data X (Orig).** In Orig, the training is performed with the original data without SVD and without adding any noise. This results in the best accuracy, but that does not provide any privacy guarantee.

Fig. 9 shows the accuracy of Low-Rank, DP-X, AsymML and Orig on CIFAR-10 and ImageNet. Since CIFAR-10 is a very small dataset, we only train smaller models (VGG-16, ResNet-18) on it. On CIFAR-10, Low-Rank already achieves very high accuracy. AsymML further improves the accuracy by 1.45% and 2.59% in VGG-16 and ResNet-18, respectively. The final accuracy is almost the same as the original models. However, to achieve the same privacy guarantee, DP-X that directly adds noise to inputs suffers a significant accuracy drop.

Similar results are also observed on ImageNet. While Low-Rank training incurs a slightly larger accuracy drop, AsymML achieves almost the same accuracy as the original models. DP-X still fails to preserve accuracy under the same privacy budget.

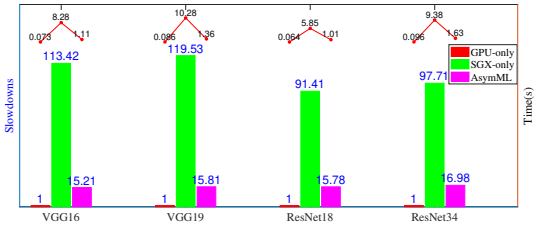
Table 1. Noise parameters for training in DP-X and AsymML.

	b	B	q	ξ	σ	
					AsymML	DP-X
CIFAR-10	32	50K	6e-4	0.05	0.12	2.5
ImageNet	128	1.2M	1e-4	0.05	0.11	2.2

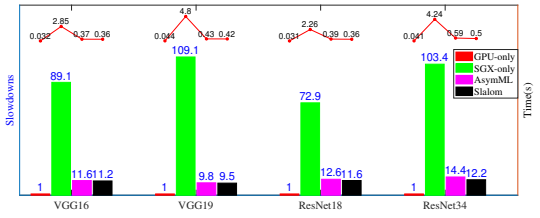
6.4 Runtime analysis

We conduct training and inference on ImageNet and compare the runtime with two baselines: GPU-only, TEE-only. Following the standard data pre-processing procedure, we resize each image into $3 \times 224 \times 224$. The batch size is set to 32. We compute the runtime as the average time of a forward and backward pass. Computing the number of principal channels $\mathcal{R}$ is TEEs is similar as in Section 6.3.

Training runtime. For training, we compare the runtime performance with the TEE-only and the GPU-only executions. Fig. 10a shows the actual runtime (red plots) and the relative slowdowns (bar plots) compared to the GPU-only execution. Compared to the TEE-only method, AsymML achieves $7.5-7.6\times$ speedup on VGG-16/VGG-19, and up to $5.8-5.8\times$ speedup on ResNet-18/34. By encoding most information in TEEs with very low-rank representations, AsymML achieves significant performance gains compared to TEE-only executions. Although AsymML shows around $15\times$ slowdown compared to *untrusted* GPU-only execution, AsymML protects the privacy of the training datasets unlike the GPU-only execution.



(a) The training time of AsymML compared to the SGX-only and the GPU-only executions is shown.



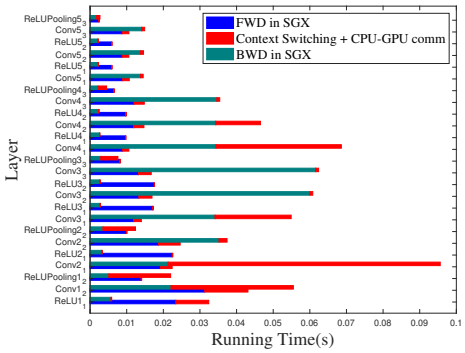
(b) The inference time of AsymML compared to the SGX-only execution, the GPU-only execution and Slalom is shown.

Fig. 10. Training and Inference performance on ImageNet. Bar plot shows slowdowns compared to GPU-only execution; red dotted lines are the corresponding running time.

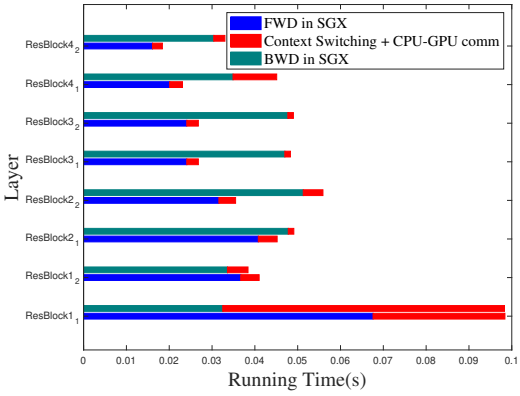
Inference runtime. We also compare AsymML to Slalom [7] in terms of inference time. While a part of the convolution is performed in TEEs, inference using AsymML is just slightly slower than Slalom. Therefore, the low-rank representation in TEEs does not incur significant additional costs.

Runtime breakdown. To better identify the main bottlenecks on a heterogeneous platform, we profile the running time of the forward and backward passes in AsymML. We break down the running time into computation time and communication time. We show the runtime breakdown for VGG-16 and ResNet-18 in Fig. 11. More results are provided in Appendix E.

- **Forward pass**. For the forward pass, the blue bars show the time spent in TEEs, while the red bars show CPU-GPU communication. In the early convolutional layers, due to the large number of features, the communication from GPUs to CPUs brings notable costs, which then becomes marginal in later layers.
- **Backward pass.** For the backward pass, the green bars show the time in TEEs, and the red bars show CPU-GPU communication. The additional cost for CPU-GPU communication is much more dominant, especially in early convolution layers. The reason is that during the backward pass, not only are input gradients $\nabla_X \mathcal{L}$ but also parameter gradients $\nabla_{W_{i,j}} \mathcal{L}$ transferred between CPUs and GPUs.



(a) The runtime of VGG-16 is shown.



(b) The runtime of ResNet-18 is shown.

Fig. 11. An illustration of the runtime breakdown in VGG-16 and ResNet-18 is shown.

6.5 Privacy protection

In this subsection, we evaluate AsymML's privacy protection against the two attacks of Section 5: the model inversion attack and the gradient inversion attack. We use CIFAR-10 to demonstrate the performance of AsymML under such attacks in all experiments. We use the same noise in Table 1. We first list the metrics used to evaluate the performance of AsymML and then we discuss detailed results.

Metrics. We consider the following metrics: peak signal-to-noise ratio (PSNR), structural similarity index (SSIM), and the accuracy under the target model ($\text{Acc}_{\mathcal{M}}$). PSNR is used to measure the pixel-wise similarity between two samples, while SSIM is used to measure the visual quality of a human visual system [49]. They are both widely used in image quality assessment. Large PSNR ($\leq \infty$) and SSIM (≤ 1) indicate more similarity between original and the reconstructed images. Finally, we also report accuracy on the target model $\mathcal{M}_{\text{t}}$ to test how it recognizes the reconstructed data.

Model inversion attack. In this experiment, we first partition the dataset into two parts. The first part is used as a public dataset $\mathcal{X}_\mathrm{p}$ to train the attack model and the other part $\mathcal{X}_\mathrm{t}$ is used as a private dataset to train the target model $\mathcal{M}_\mathrm{t}$. With $\mathcal{X}_\mathrm{p}$ and $\mathcal{X}_\mathrm{t}$ created from one dataset, we simulate the scenario that the attack model can learn general knowledge about the target dataset such as features relevant to the labels. We use $\mathcal{M}(X^{(\mathrm{U})})$ in the first layer ($X^{(\mathrm{U})}$ perturbed with small noise) as the prior information to the attack model as it reveals most information in inputs.

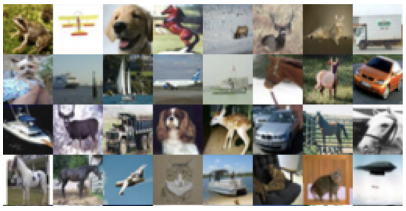
In the first stage of training the attack model, we train the GAN model using an Adam optimizer with weight decay 0.0005 and momentum 0.5/0.999 for both the generator G and the discriminator D. The batch size is set as 64. We set the learning rate as 0.0025 for G and as 0.01 for D to achieve the best training performance. In the second stage, we optimize the latent vector z using an SGD optimizer with learning rate 0.01. Each batch of latent vectors is optimized for 200 iterations.

Dataset	Target Model	PSNR	SSIM	$\mathrm{Acc}_{\mathcal{M}}$
CIFAR-10	VGG-16	9.43	0.12	10.74%
CIFAR-10	ResNet-18	9.31	0.09	10.56%

Table 2. Similarity between the reconstructed images and the original ones using various metrics in model inversion attacks, where $\mathrm{PSNR}_{\mathrm{max}} = \infty$, and $\mathrm{SSIM}_{\mathrm{max}} = 1.0$ for two identical images.

Table 2 lists the performance of the model inversion attack using different metrics. Based on PSNR and SSIM, the reconstructed images using $\mathcal{M}(\mathcal{X}^{(U)})$ as prior knowledge share very few similarities with the original ones. Furthermore, as reported in $\mathrm{Acc}_{\mathcal{M}}$ for target models VGG-16 and ResNet-18, with the $\mathcal{M}(\mathcal{X}^{(U)})$ as prior information, the target models still fail to match the reconstructed images with their true labels (low $\mathrm{Acc}_{\mathcal{M}}$). Fig. 12 further presents some reconstructed data samples (Fig. 12c) compared to the original ones (Fig. 12a). The target model is ResNet-18. Based on Fig. 12b, it is observed that the residual $\mathcal{M}(\mathcal{X}^{(U)})$ reveals little feature information about a specific class. As a result, even though it is used as prior knowledge when training GAN models in the MI attack, the reconstructed and original images still share few common features. Therefore, AsymML can effectively defend against this MI attack.

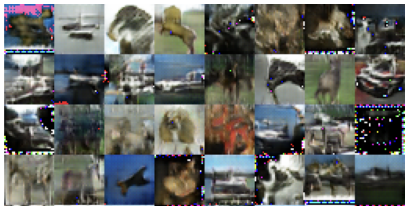
We further conduct a model inversion reconstruction using $\mathcal{X}^{(U)}$ as prior knowledge, as shown in Fig. 12d. It is interesting to observe that $\mathcal{X}^{(U)}$ without noise does provide a little useful information that helps reconstruct slightly better synthetic images. Therefore, adding small noise to $\mathcal{X}^{(U)}$ as in AsymML is very critical to further hide such information.



(a) The original data samples of CIFAR-10 dataset.

(b) The noisy residual data $\mathcal{M}(X^{(\mathrm{U})})$ of CIFAR-10 dataset.

(c) The reconstructed samples using $\mathcal{M}(\mathcal{X}^{(U)})$ in AsymML.



(d) The reconstructed samples using $\mathcal{X}^{(U)}$.

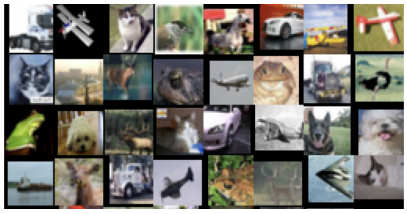
Fig. 12. The reconstructed data samples using the model inversion attack with residual data $\mathcal{M}(X^{(\mathrm{U})})$ as prior information, and ResNet-18 as the target model are shown.

Gradient inversion attack. In the experiments, we use $\mathcal{M}_\mathrm{t}$ with random initialized values as in [50]. $\mathcal{X}^{(U)}$ of the inputs (perturbed by small noise) is used as initial synthetic images.

During optimization, we use an L-BFGS optimizer with an initial learning rate of 0.1. The history vector size in L-BFGS is 100. The batch size is 64. The maximum number of iterations for one optimization is 200. λ is Eq (15) is 0.5. Fig 13 shows the original images and the reconstructed ones using this gradient inversion attack. It it observed that given $\mathcal{M}(X^{(\mathrm{U})})$ as ini-

tial value (Fig 13b), the gradient attack fails to generate similar images as the original ones (Fig 13c), even though the optimization loss is very small (Fig. 13e). It is worth noting that the original gradient attack method [50] achieves good reconstruction only on a single image, rather than on large batches. Such an observation is also aligned with our results.

We also use $\mathcal{X}^{(U)}$ without noise to perform a gradient inversion attack. As shown in Fig 13b, the reconstructed images are still very noisy, and share very few features with original ones.

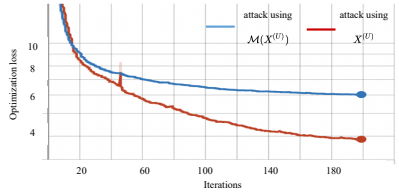


(a) The original data samples of CIFAR-10 dataset.

(b) The residual data $\mathcal{M}(X^{(\mathsf{U})})$ of CIFAR-10 dataset.

(c) The reconstructed samples using $\mathcal{M}(\mathcal{X}^{(U)})$ in AsymML.

(d) The reconstructed samples using $\mathcal{X}^{(U)}$.



(e) Optimization loss in the gradient inversion attack.

Fig. 13. The reconstructed data samples using the gradients inversion attack with residual data $\mathcal{M}(X^{(\mathsf{U})})$ as prior information, and ResNet-18 as the target model are shown.

7 Discussion

AsymML in this paper reduces the computation and memory costs in TEEs without significant privacy leaks. However, due to the additional communication overhead between GPUs and TEEs, the actual performance improvement does not exactly match the expectation in Fig. 3. As Fig. 16 in Appendix E shows, this additional overhead even dominates the runtime in layers with large-size features. Architectures such as unified memory access (UMA) [44, 45] can potentially resolve this issue by relieving CPU-GPU communication burden, where AsymML can also be leveraged.

AsymML is designed to protect privacy under the assumption that TEEs are secure against potential attacks such as side channel attacks [46]. The case that side channel attacks breach TEE's security [47] based on information gained from the implementation of a computer system (e.g. power consumption, electromagnetic leaks) are out of the scope of this work. Finally, it is worth noting that AsymML is compatible with security updates in hardware such as Intel SGX [6] and RISC-V Sanctum [48].

8 Conclusion

In this paper, we have proposed an asymmetric decomposition framework, AsymML, to decompose DNN models and offload computations onto trusted and untrusted fast hardware. The trusted part aims to preserve the main information in the data with manageable computation cost, and the untrusted part undertakes most computations. In such a way, AsymML makes the best use of each platform in a heterogeneous setting. We have then presented theoretical analysis showing that the low-rank structure is preserved in NNs, and AsymML achieves (ϵ, δ)-DP guarantee by adding a Gaussian noise to $X^{(\mathsf{U})}$. Our extensive experiments show that AsymML achieves gain up to 7.6× in training. We also show that AsymML provides strong privacy protection under model and gradient inversion attacks.

Acknowledgements

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. FASTNICS HR001120C0088, NSF grants CCF-1703575, CCF-1763673, CNS-2002874, and a gift from Intel/Avast/Borsetta via the PrivateAI institute. We would like to also acknowledge the valuable feedback and discussions by Dr. Matthias Schunter from Intel.

References

[1] Simonyan, K., Zisserman, A., Very deep convolutional networks for large-scale image recognition, arXiv:1409.1556 (2014)

[2] He, K., Zhang, X., Ren, S. and Sun, J., Deep residual learning for image recognition, IEEE CVPR (2016)

[3] Devlin, J., Chang, M.W., Lee, K., Toutanova and K., Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv:1810.04805 (2018)

[4] Abadi, M., Chu, A., et al, Deep learning with differential privacy, ACM SIGSAC conference on computer and communications security (2016).

[5] Papernot, N., Song, S., et al, Scalable private learning with pate, arXiv:1802.08908 (2018)

[6] Costan, V. and Devadas, S., 2016. Intel sgx explained, IACR Cryptol, ePrint Arch (2016)

[7] Tramer, F. and Boneh, D., Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware, International Conference on Learning Representations (2018)

[8] Shokri, R., Stronati, M., Song, C. and Shmatikov, V., Membership inference attacks against machine learning models, IEEE Symposium on Security and Privacy (2017)

[9] Fredrikson, M., Jha, S. and Ristenpart, T., Model inversion attacks that exploit confidence information and basic countermeasures, ACM SIGSAC CCS (2015)

[10] Wang, J., Zhang, J., et al, Not just privacy: Improving performance of private deep learning in mobile cloud, ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2018)

[11] Lee, J. and Kifer, D., Scaling up Differentially Private Deep Learning with Fast Per-Example Gradient Clipping. Proc. Priv. Enhancing Technol (2021).

[12] Mireshghallah, F., et al, Shredder: Learning noise distributions to protect inference privacy, International Conference on Architectural Support for Programming Languages and Operating Systems (2020)

[13] Bonawitz, K., Ivanov, V., Kreuter, B., et al, Segal, A. and Seth, K., Practical secure aggregation for privacy-preserving machine learning, ACM SIGSAC Conference on Computer and Communications Security (2017)

[14] Alves, T. and Felton, D., TrustZone: Integrated Hardware and Software Security, White Paper, ARM (2004)

[15] Konečný, J., McMahan, B. and Ramage, D., Federated optimization: Distributed optimization beyond the data center, arXiv:1511.03575 (2015)

[16] Wu, X., Fredrikson, M., Jha, S. and Naughton, J.F., A methodology for formalizing model-inversion attacks. In IEEE Computer Security Foundations Symposium (2016).

[17] Bezdek, J.C. and Hathaway, R.J., Convergence of alternating optimization, Neural, Parallel and Scientific Computations (2003)

[18] Alter, O., Brown, P.O. and Botstein, D., Singular value decomposition for genome-wide expression data processing and modeling, Proceedings of the National Academy of Sciences (2000)

[19] Jizba, P. and Arimitsu, T., Observability of Rényi's entropy. Physical Review E (2004)

[20] So, J., Güler, B. and Avestimehr, A.S., CodedPrivateML: A fast and privacy-preserving framework for distributed machine learning. IEEE Journal on Selected Areas in Information Theory (2021).

[21] Gao, X., Gao, F., Tao, D. and Li, X., Universal blind image quality assessment metrics via natural scene statistics and multiple kernel learning. IEEE Transactions on neural networks and learning systems (2013).

[22] Harchaoui, Z., Douze, M., et al, Large-scale image classification with trace-norm regularization. In IEEE Conference on Computer Vision and Pattern Recognition (2012).

[23] Graepel, T., Lauter, K. and Naehrig, M., ML confidential: Machine learning on encrypted data. In International Conference on Information Security and Cryptology (2012).

[24] Sun, X., Zhang, P., et al, Private machine learning classification based on fully homomorphic encryption. IEEE Transactions on Emerging Topics in Computing (2018).

[25] Bagdasaryan, E., Poursaeed, O. and Shmatikov, V., Differential privacy has disparate impact on model accuracy. Advances in Neural Information Processing Systems (2019).

[26] McMahan, B., Moore, et al, Communication-efficient learning of deep networks from decentralized data. In Artificial intelligence and statistics (2017).

[27] Zhao, Y., Li, M., et al, Federated learning with non-iid data. arXiv preprint arXiv:1806.00582 (2018).

[28] Wei, K., Li, J., et al, Federated learning with differential privacy: Algorithms and performance analysis. IEEE Transactions on Information Forensics and Security (2020).

[29] Jin, C., Al Badawi, A., et al, CareNets: Efficient homomorphic CNN for high resolution images. In NeurIPS Workshop on Privacy in Machine Learning (2019).

[30] Knott, B., Venkataraman, S., Hannun, A., Sengupta, S., Ibrahim, M. and van der Maaten, L., CrypTen: Secure multi-party computation meets machine learning. Advances in Neural Information Processing Systems (2021).

[31] Ma, X., Zhang, F., Chen, X. and Shen, J., Privacy preserving multi-party computation delegation for deep learning in cloud computing. Information Sciences (2018).

[32] Kunkel, R., Quoc, D.L., et al, Tensorscone: A secure tensorflow framework using intel sgx. arXiv preprint arXiv:1902.04413 (2019).

[33] Quoc, D.L., Gregor, F., et al, secureTF: a secure TensorFlow framework. In Proceedings of the 21st International Middleware Conference (2020).

[34] Volos, S., Vaswani, K. and Bruno, R., Graviton: Trusted execution environments on gpus. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI

18) (2018).
[35] Hanzlik, L., Zhang, Y., et al, Mlcapsule: Guarded offline deployment of machine learning as a service. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (2021).
[36] Zhang, C., Xia, J., et al, Citadel: Protecting Data Privacy and Model Confidentiality for Collaborative Learning with SGX. arXiv preprint arXiv:2105.01281 (2021).
[37] Yuhala, P., Felber, P., Schiavoni, V. and Tchana, A., Plinius: Secure and Persistent Machine Learning Model Training. arXiv preprint arXiv:2104.02987 (2021).
[38] Hashemi, H., Wang, Y. and Annavaram, M., DarKnight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (2021).
[39] Ng, L.K., Chow, S.S., et al, Goten: GPU-Outsourcing Trusted Execution of Neural Network Training. In Proceedings of the AAAI Conference on Artificial Intelligence (2021).
[40] Zhang, Y., Jia, R., et al, The secret revealer: Generative model-inversion attacks against deep neural networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (2021).
[41] Krizhevsky, A., Hinton, G. et al. Learning multiple layers of features from tiny images (2009).
[42] Deng, J., Dong, W., et al, Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition (2009).
[43] Paszke, A., Gross, S., et al, Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems (2019).
[44] Chu, H., AMD heterogeneous uniform memory access. Proceedings of the APU 13th Developer Summit (2013).
[45] Apple Inc., Apple at Work M1 Overview (2021).
[46] Standaert, F.X., Introduction to side-channel attacks. In Secure integrated circuits and systems (2010).
[47] Chen, G., Chen, S., et al, Sgxpectre attacks: Leaking enclave secrets via speculative execution. arXiv preprint arXiv:1802.09085 (2018).
[48] Costan, V., Lebedev, I. and Devadas, S., Sanctum: Minimal hardware extensions for strong software isolation. In 25th USENIX Security Symposium (2016).
[49] Wang, Z., Bovik, A.C., et al, Image quality assessment: from error visibility to structural similarity. IEEE transactions on image processing (2004).
[50] Zhu, L., Zhijian L., and Song H., Deep leakage from gradients. Advances in Neural Information Processing Systems (2019).
[51] Tang, Tingting, Ali, Ramy E, et al, Verifiable coded computing: Towards fast, secure and private distributed machine learning. arXiv preprint arXiv:2107.12958 (2021).
[52] Dwork, C. and Roth, A., The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci., (2014).
[53] Gilad-Bachrach, R., Dowlin, N., et al, Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In International conference on machine learning (2016).

A Proofs of all lemmas and theorems

In this appendix, we provide the proofs of all lemmas and theorems as follows.

1. Section A.1 presents the proof of Lemma 1.
2. Section A.2 provides the proof of Theorem 1.
3. Section A.3 provides the proof of Theorem 2.
4. Section A.4 provides the proof of Theorem 3.
5. Section A.5 presents the proof of Theorem 4.
6. Section A.6 presents the proof of Theorem 5.
7. Section A.7 provides the proof of Theorem 6.

A.1 Proof of Lemma 1

Proof. Let $Q = \sum_{i=1}^{N} s_i$ and $P = \sum_{i=1}^{N} s_i^2$, in which $s_1 \geq s_2 \geq \cdots \geq s_N$. We note that Q satisfies

$$\sqrt{P} \leq Q \leq \sqrt{NP}, \tag{16}$$

where the left equality holds when all singular values are zeros except s_1 and the right equality holds when all singular values are equal. According to Definition 1, we have

$$\begin{aligned}\mu_X &= -\log\left(\sum_{i=1}^{N} \bar{s}_i^2(X)\right) = -\log\left(\sum_{i=1}^{N} \frac{s_i^2}{Q^2}\right) \\ &= -\log(\sum_{i=1}^{N} s_i^2) + \log Q^2 = 2\log Q - \log P.\end{aligned}$$

Finally, from above equation, we have $0 \leq \mu_X \leq \log N$. □

A.2 Proof of Theorem 1

Proof. Let $Q = \sum_{i=1}^{N} s_i$ and $P = \sum_{i=1}^{N} s_i^2$, and assume that $s_1 > s_2 > \cdots > s_N$. According to the assumption, the i-th singular value is given as $s_i = a \cdot b^{i-1}$, where $a > 0, 0 < b < 1$. Hence, we have

$$\begin{aligned}Q &= \sum_{i=1}^{N} s_i = a \cdot \sum_{i=1}^{N} b^{i-1} = \frac{1-b^N}{1-b}, \\ P &= \sum_{i=1}^{N} s_i^2 = a^2 \cdot \sum_{i=1}^{N} b^{2(i-1)} = \frac{1-b^{2N}}{1-b^2},\end{aligned}$$

$$2^{\mu_X} = \frac{Q^2}{P} = \frac{(1+b)(1-b^N)}{(1-b)(1+b^N)}.$$

Let $\eta = \frac{\sum_{i=1}^{\mathcal{R}} s_i^2}{\sum_{j=1}^{N} s_j^2}$, which can be lower-bounded as follows

$$\eta(b,N) = \frac{1-b^{2\mathcal{R}}}{1-b^{2N}} \geq \frac{1-b^{2\cdot 2^{\mu_X}}}{1-b^{2N}} = \frac{1-b^{\frac{2Q^2}{P}}}{1-b^{2N}} = \frac{1-b^{\frac{2(1+b)(1-b^N)}{(1-b)(1+b^N)}}}{1-b^{2N}}.$$

Finally by minimizing the function $\eta(b,N)$, we get a minimum of 0.97. Therefore, with $\mathcal{R} = \lceil 2^{\mu_X} \rceil$ principal channels, the total energy in the reconstructed data is at least 97% of the total energy in the original data X. □

In the later theorems and proofs, we regard $\mathcal{R}$ as the sufficient number of principal channel to reconstruct X.

A.3 Proof of Theorem 2

Proof. Let $\mathcal{R} = \lceil 2^{\mu_X} \rceil$, then $X_j = \sum_{p=1}^{\mathcal{R}} a_{j,p} \cdot X_p^{'}$, where $X_p^{'}$ is the p-th principle channel of X, and $\overline{X'}_p$ denote a flatten vector from $X_p^{'}$. Then, the i-th output channel is given by

$$Y_i = \sum_{j=1}^{N} W_{i,j} \circledast X_j = \sum_{p=1}^{\mathcal{R}} (\sum_{j=1}^{N} W_{i,j} \cdot a_{j,p}) \circledast X_p^{'}.$$

All channels in Y can be then written as follows

$$Y = \{Y_1, \cdots, Y_M\} = \sum_{p=1}^{\mathcal{R}} \left\{ (\sum_{j=1}^{N} W_{1,j} \cdot a_{j,p}) \circledast X_p^{'}, \cdots \right\}.$$

Let

$$Y_p^{'} = \left\{ (\sum_{j=1}^{N} W_{1,j} \cdot a_{j,p}), \cdots, (\sum_{j=1}^{N} W_{M,j} \cdot a_{j,p}) \right\} \circledast X_p^{'},$$

then $Y = \sum_{p=1}^{\mathcal{R}} Y_p^{'}$. Since $W_{i,j}$ is a 1×1 kernel, $Y_p^{'}$ can be written as

$$Y_p^{'} = \left\{ (\sum_{j=1}^{N} W_{1,j} \cdot a_{j,p}), \cdots, (\sum_{j=1}^{N} W_{M,j} \cdot a_{j,p}) \right\} \cdot X_p^{'}.$$

For any two principal channels, $X_{p_1}^{'}, X_{p_2}^{'}, p_1 \neq p_2$, $\left\langle \overline{X'}_{p_1}, \overline{X'}_{p_2} \right\rangle = 0$. Therefore, $Y_{p_1}^{'}, Y_{p_2}^{'}$ are constructed by two orthogonal channels. Y only has at most $\mathcal{R}$ principle channels. Therefore, $\mu_Y \leq \log \mathcal{R} = \log \lceil 2^{\mu_X} \rceil$. □

A.4 Proof of Theorem 3

To prove Theorem 3, we first need a lemma to bound SVD-channel entropy for patches in all channels $\hat{X}_{:,q,r}$ is almost the same as that of the original data X Then according to Lemma 2, we can bound SVD-channel entropy after $k \times k$ convolutions.

Lemma 2. *Given an input $X \in R^{N \times h \times w}$ with SVD-channel entropy μ_X, $k \times k$ kernels, then for $\forall 1 \leq q, r \leq k$, SVD-channel entropy in $\hat{X}_{:,q,r}$ satisfies:*

$$\mu_{\hat{X}_{:,q,r}} \leq \log \lceil 2^{\mu_X} \rceil$$

Proof. Flatten $\hat{X}_{:,q,r}$, X as $\overline{\hat{X}}_{:,q,r}$ and $\overline{X}$. Then, $\overline{\hat{X}}_{:,q,r}$ can be regarded as $\overline{X}$ with at most $k^2 - (\frac{k+1}{2})^2$ columns reset as zeros. We use a set S_0 to denote these columns. Let $\mathcal{R} = \log \lceil 2^{\mu_X} \rceil$, then each row in $\overline{X}$ can be written as $\overline{X}_j = \sum_{p=1}^{\mathcal{R}} a_{j,p} \overline{X}_p^{'}$. Therefore, each row in $\overline{\hat{X}}_{:,q,r}$ can be written as $\overline{\hat{X}}_{j,q,r} = \sum_{p=1}^{\mathcal{R}} a_{j,p} \overline{\hat{X}}_{p,q,r}^{'}$, where $\overline{\hat{X}}_{p,q,r}^{'}$ is the same as $\overline{X}_p^{'}$ except for values in columns S_0 are zeros. Therefore, $\overline{\hat{X}}_{:,q,r}$ has at most $\mathcal{R}$ principle components. Namely, $\hat{X}_{:,q,r}$ has at most $\mathcal{R}$ principle channels. Hence, $\mu_{\hat{X}_{:,q,r}} \leq \mathcal{R} = \log \lceil 2^{\mu_X} \rceil$. □

With Lemma 2, we prove Theorem 3 as follows:

Proof. According to Lemma 2, for the (q,r)-th patches in all channels, $\hat{X}_{:,q,r}, \forall 1 \leq q, r \leq k$ can be constructed by at most $\mathcal{R}$ principle channels $\{\mathcal{U}_{1,q,r}, \cdots, \mathcal{U}_{\mathcal{R},q,r}\}$ as follows

$$\begin{cases} \hat{X}_{1,q,r} &= a_{1,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{1,\mathcal{R},q,r}\mathcal{U}_{\mathcal{R},q,r} \\ \quad \vdots & \\ \hat{X}_{N,q,r} &= a_{N,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{N,\mathcal{R},q,r}\mathcal{U}_{\mathcal{R},q,r}. \end{cases}$$

For all patches in channel j, $\hat{X}_{j,:,:} \forall 1 \leq j \leq N$ can also be constructed by principle channels $\left\{\mathcal{V}_{j,1}, \cdots, \mathcal{V}_{j,\hat{N}_j}\right\}$, where $\hat{N}_j = \left\lceil 2^{\hat{\mu}_j} \right\rceil$,

$$\begin{cases} \hat{X}_{j,1,1} &= b_{j,1,1,1}\mathcal{V}_{j,1} + \cdots + b_{j,p_j,1,1}\mathcal{V}_{j,\hat{N}_j} \\ \quad \vdots & \\ \hat{X}_{j,k,k} &= b_{j,1,k,k}\mathcal{V}_{j,1} + \cdots + b_{j,p_j,k,k}\mathcal{V}_{j,\hat{N}_j}. \end{cases}$$

By combining equation (1) and (2), we can express $\hat{X}_{:,q,r}$ as follows

$$\begin{cases} \hat{X}_{1,q,r} & = a_{1,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{1,\mathcal{R},q,r}\mathcal{U}_{\mathcal{R},q,r} \\ & = b_{1,1,q,r}\mathcal{V}_{1,1} + \cdots + b_{1,p_j,q,r}\mathcal{V}_{1,\hat{N}_1} \\ \vdots & \\ \hat{X}_{N,q,r} & = a_{N,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{N,\mathcal{R},q,r}\mathcal{U}_{\mathcal{R},q,r} \\ & = b_{N,1,q,r}\mathcal{V}_{N,1} + \cdots + b_{N,p_j,q,r}\mathcal{V}_{N,\hat{N}_N}. \end{cases}$$

In (A.4), the coefficients are obtained from SVD, therefore the determinant of any sub coefficient matrix is not zero. Hence, at most $\mathcal{R}$ sub-equations are needed to derive $\mathcal{U}_{:,q,r}$ from $\mathcal{V}_{:,:}$.

If we pick the equations with least number of principal channels $\mathcal{V}$, then each channel in $\hat{X}$ can be fully constructed by the selected $\mathcal{V}$. At most, the total number of principal channels $\mathcal{V}$ needed is $\sum_{j=1}^{\mathcal{R}} \hat{N}_j$. Then the total number of principle channels needed to construct $\hat{X}$ is at most $\sum_{j=1}^{\mathcal{R}} \hat{N}_j$.

Therefore, $\mu_{\hat{X}} \leq \log(\sum_{j=1}^{\mathcal{R}} \hat{N}_j) = \log(\sum_{j=1}^{\mathcal{R}} \lceil 2^{\hat{\mu}_j} \rceil)$. When $\hat{\mu}_j$ are close, $\mu_{\hat{X}} \doteq \eta + \bar{\mu}$, where $\bar{\mu}$ is the average of for $\hat{\mu}_j$. Finally, according to Theorem 2, $\mu_Y \leq \log \lceil 2^{\mu_{\hat{X}}} \rceil) \leq \log(\sum_{j=1}^{\mathcal{R}} \lceil 2^{\hat{\mu}_j} \rceil)$. □

A.5 Proof of Theorem 4

Proof. Let $\mathcal{R} = \lceil 2^{\mu_X} \rceil$, then input $X_j = \sum_{p=1}^{\mathcal{R}} a_{j,p} \cdot X_p^{'}$, where $X^{'}$ is the principle channels of X. With batch normalization operator, we have

$$Y_j = \frac{X_j - E[X_j]}{\sqrt{V[X_j] + \epsilon}} \cdot \gamma_i + \beta_i,$$

where $E[X_j]$ and $V[X_j]$ is the mean and variance in channel j across batches, γ_j and β_j are learnable parameters in BatchNorm layers and ϵ is a constant for numerical stability. We can then re-write Y_i in terms of $X_p^{'}$ as

$$\begin{aligned} Y_i &= \frac{\sum_{p=1}^{\mathcal{R}} a_{j,p} \cdot X_p^{'} - E[X_j]}{\sqrt{V[X_j] + \epsilon}} \cdot \gamma_j + \beta_j \\ &= \sum_{p=1}^{\mathcal{R}} \frac{\gamma_j a_{j,p}}{\sqrt{V[X_j] + \epsilon}} X_p^{'} - \frac{\gamma_j E[X_j]}{\sqrt{V[X_j] + \epsilon}} + \beta_j \\ &= \sum_{p=1}^{\mathcal{R}} \frac{\gamma_j a_{j,p}}{\sqrt{V[X_j] + \epsilon}} X_p^{'} - (\frac{\gamma_j E[X_j]}{\sqrt{V[X_j] - \epsilon}} - \beta_j) \cdot \mathbf{1} \end{aligned}$$

Therefore, the output channel Y_j can be reconstructed by at most $\mathcal{R}$ principal channels in X, plus a constant vector $\mathbf{1}$. Hence $\mu_Y \leq \log \lceil 2^{\mu_X} \rceil + 1$. □

A.6 Proof of Theorem 5

Proof. Let $\mathcal{R} = \lceil 2^{\mu_X} \rceil$, then input $X_j = \sum_{p=1}^{\mathcal{R}} a_{j,p} \cdot X_p^{'}$, where $X^{'}$ is the principle channels of X. With $k \times k$ average operator, we have

$$\begin{aligned} Y_i(h,w) &= \frac{1}{k^2} \sum_{h^{'}=1}^{k} \sum_{w^{'}=1}^{k} X_i((h-1)k + h^{'}, (w-1)k + w^{'}) \\ &= \frac{1}{k^2} \sum_{h^{'}=1}^{k} \sum_{w^{'}=1}^{k} \sum_{p=1}^{\mathcal{R}} a_{i,p} X_p^{'}((h-1)k + h^{'}, (w-1)k + w^{'}) \\ &= \sum_{p=1}^{\mathcal{R}} a_{i,p} \cdot \frac{\sum_{h^{'},w^{'}=1}^{k} X_p^{'}((h-1)k + h^{'}, (w-1)k + w^{'})}{k^2} \end{aligned}$$

Therefore, each channel in Y can be seen as an accumulation of $\mathcal{R}$ principle channels obtained by conducting average pooling on $X^{'}$. Therefore, Y can be constructed by at most $\mathcal{R}$ principle channels. Hence, $\mu_Y \leq \log \mathcal{R} = \log \lceil 2^{\mu_X} \rceil$ □

A.7 Proof of Theorem 6

Proof. Due to the random sampling process, two cases might arise: 1) if $X^i \notin \mathcal{S}$, $\left\| \mathcal{S}^{(U)} - \mathcal{S}^{'(U)} \right\| = 0$; 2) if $X^i \in \mathcal{S}$, $\left\| \mathcal{S}^{(U)} - \mathcal{S}^{'(U)} \right\| \leq \Delta_2$.

Since $\mathcal{S}$ and $\mathcal{S}^{'}$ can only differ at one position, without loss of generality, we assume X and $X^{'}$ are the data that might be different in $\mathcal{S}$ and $\mathcal{S}^{'}$. According to data decomposition, $X = X^{(\mathrm{T})} + X^{(\mathrm{U})}$, and $X^{'} = X^{'(T)} + X^{'(U)}$. With noise $z = \mathcal{N}(0, \sigma^2 I)$ added to $X^{(\mathrm{U})}$ and $X^{'(U)}$, we have

$$\mathcal{M}(X^{(\mathrm{U})}) = X^{(\mathrm{U})} + z, \quad \mathcal{M}(X^{'(U)}) = X^{'(U)} + z.$$

Given $0 < \epsilon \leq 1, \delta > 0$, we next focus on deriving the tail bound δ. We define C as

$$C = \log \frac{\Pr(\mathcal{M}(X^{(U)}) = X^{(\mathrm{U})} + z)}{\Pr(\mathcal{M}(X^{'(U)}) = X^{(\mathrm{U})} + z)}.$$

Then, we have

$$\begin{aligned} \Pr(|C| \geq \epsilon) = & \Pr(X^{(\mathrm{U})} \neq X^{'(\mathrm{U})}) \cdot \\ & \Pr(|C| \geq \epsilon | X^{(\mathrm{U})} \neq X^{'(\mathrm{U})}). \end{aligned}$$

Given $X^{(\mathrm{U})} \neq X^{'(\mathrm{U})}$, let $v = X^{(\mathrm{U})} - X^{'(\mathrm{U})}$, C can be written as

$$\begin{aligned} C &= \log \frac{\exp\left(-\left\|z\right\|^2 / 2\sigma^2\right)}{\exp\left(-\left\|z+v\right\|^2 / 2\sigma^2\right)} \\ &= -\frac{1}{2\sigma^2}(\left\|z\right\|^2 - \left\|z+v\right\|^2) \\ &= \frac{1}{2\sigma^2}(2\left\langle z, v \right\rangle + \left\|v\right\|^2), \end{aligned}$$

where $\langle ., . \rangle$ denotes the sum of element-wise product. It is easy to verify that C given $X^{(\mathrm{U})} \neq X'^{(\mathrm{U})}$ is a Gaussian random variable with mean $\frac{\|v\|^2}{2\sigma^2}$ and variance $\frac{\|v\|^2}{\sigma^2}$. Since $q = \Pr(X^{(\mathrm{U})} \neq X'^{(\mathrm{U})})$, $\Pr(|C| \geq \epsilon) \leq \delta$ is equivalent to $\Pr(|C| \geq \epsilon | X^{(\mathrm{U})} \neq X'^{(\mathrm{U})}) \leq \frac{1}{q}\delta = \delta'$.
Similar as the classical Gaussian mechanism, we set $\sigma = \frac{t\Delta_2}{\epsilon}$. Since $\|v\| \leq \Delta_2$, $\Pr(|C| \geq \epsilon | X^{(\mathrm{U})} \neq X'^{(\mathrm{U})})$ is equivalent to

$$\Pr\left(\left|STD(C|X^{(\mathrm{U})} \neq X'^{(\mathrm{U})})\right| \geq \frac{\epsilon\sigma}{\|v\|} - \frac{\|v\|}{2\sigma^2}\right)$$
$$\Leftrightarrow$$
$$\Pr\left(\left|STD(C|X^{(\mathrm{U})} \neq X'^{(\mathrm{U})})\right| \geq t - \frac{\epsilon}{2t}\right)$$

STD denotes the standardization of a distribution. Following the classical Gaussian mechanism [52], if we set $t = \sqrt{2\log(1.25/\delta')} = \sqrt{2\log(1.25q/\delta)}$, we get

$$\Pr(|C| \geq \epsilon) = \Pr(\left|STD(C|X^{(\mathrm{U})} \neq X'^{(\mathrm{U})})\right| \geq t - \frac{\epsilon}{2t}) \leq \delta.$$

□

B Model architecture for MI attack

In this appendix, we provide the model architectures of G and D in Table 3 and Table 4, respectively.

Type	Kernel	Channels	Stride	Padding	Output
		B1			
Conv	3	32	1	1	32×32
Conv	3	64	2	1	16×16
Conv	3	128	1	1	16×16
Conv	3	128	2	1	8×8
Conv	3	128	1	1	8×8
		B2			
DeConv	4	256	1	0	4×4
DeConv	4	128	2	1	8×8
		Decoder			
DeConv	4	128	2	1	16×16
DeConv	4	64	2	1	32×32
Conv	3	32	1	1	32×32
Conv	3	3	1	1	32×32

Table 3. The generator architecture (B1, B2, and the decoder) is shown. B1 is used to extract the features from the residual data $X^{(\mathrm{U})}$, while B2 is used to generate the latent features. The decoder then combines these features, and reconstructs the output.

Type	Kernel	Channels	Stride	Padding	Output
Conv	3	32	2	1	16×16
Conv	3	64	2	1	8×8
Conv	3	128	2	1	4×4
Conv	3	256	2	1	2×2
Conv	3	256	2	1	1×1

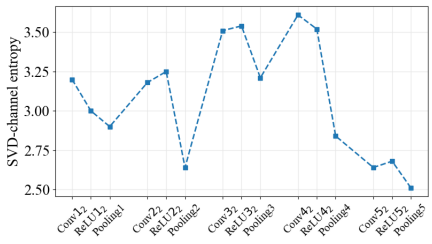
Table 4. The discriminator architecture is shown. Batch normalization and ReLU is applied after every convolution layer, similar to the generator.

C SVD-channel entropy in NN models

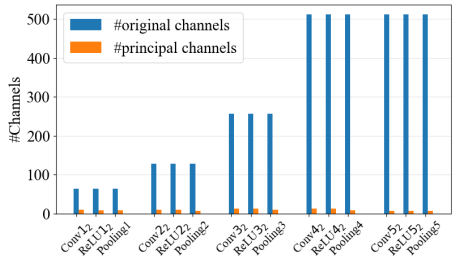
In this appendix, we empirically illustrate how the SVD-channel entropy changes across the layers in DNNs.
In Fig. 14, we show the average SVD-channel entropy and the required number of principal channels to approximate intermediate data across all batches in VGG-19/ImageNet with a randomly initialized model. We note the following two key observations. First, as shown in Figure 14a, the SVD-channel entropy before and after the ReLU layers barely changes. As for Pooling (Max-Pooling) layers, the SVD-channel entropy usually decreases since the features are down-sampled in these layers. Second, from Figure 14b, we observe that the required number of principal channels to reconstruct the original intermediate features is much less than the number of original kernels. Thus, these observations provide strong evidence for the existence of low-rank structure in NNs.

D Approximate SVD evaluation

In this appendix, we further evaluate the approximated light SVD in Algorithm 1.
To measure the efficacy of Algorithm 1, we calculate the total energy in the remaining data $\overline{X}^{(i)}$ after extracting the i most principal channels as shown in Equation (9). It is worth noting that minimizing such residual energy is also one of the objectives of the original SVD algorithm. To give a better comparison, we use relative energy, $\left\|\overline{X}^{(i)}\right\|_F^2 / \left\|\overline{X}\right\|_F^2$ and compare with SVD. Figure 15 shows the relative residual energy by performing SVD and our approximated algorithm in outputs after the first and the second convolution layer in VGG-16. The results are obtained by averaging multiple mini-batches using a randomly initialized model. We observe



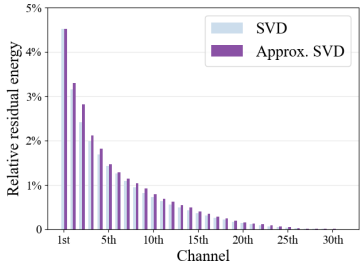
(a) SVD-channel entropy



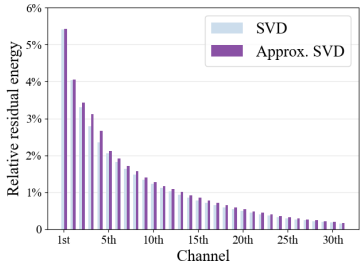
(b) Number of principal channels vs. original channels

Fig. 14. SVD-channel entropy in VGG-19/ImageNet.

that the extracted principal channels using the approximated SVD contain almost the same energy as using SVD. Therefore, such an approximation algorithm captures low-rank components in data as using the original SVD algorithm.



(a) 1st convolution layer.

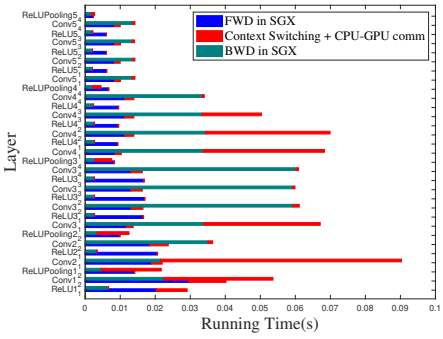


(b) 2nd convolution layer.

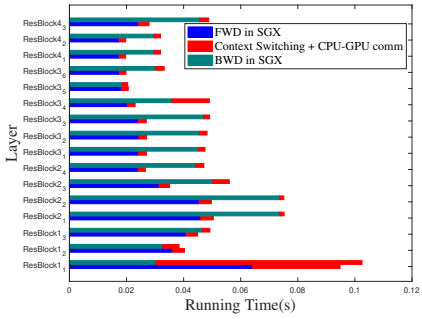
Fig. 15. Energy in residuals after 1st and 2nd convolution layer is shown in VGG-16 using SVD and the approximated SVD in Algorithm 1. The extracted principal channels using approximated SVD contain almost the same energy as using SVD.

E Runtime breakdown of VGG-19 and ResNet-34

In this appendix, we provide the runtime breakdown of VGG-19 and ResNet-34 as shown in Figure 16. Similar to in Figure 11, the communication between SGX and GPUs dominates the runtime in the early convolution layers, especially in backward passes, while it becomes marginal in the later layers.



(a) The runtime of VGG-19 is shown.



(b) The runtime of ResNet-34 is shown.

Fig. 16. The runtime breakdown in VGG-19 and ResNet-34 is shown.Time in data movement is relatively high in early convolutional layers, which then becomes marginal in later layers.