ZEN: Efficient Zero-Knowledge Proofs for Neural Networks

Boyuan Feng[1], Lianke Qin[1], Zhenfei Zhang[2], Yufei Ding[1], and Shumo Chu[1]

[1] UC Santa Barbara,
{boyuan,lianke,yufeiding,shumo}@ucsb.edu
[2] Manta Network,
zhenfei@manta.network

Abstract. In this paper, we present ZEN, a toolchain for producing efficient zero-knowledge proof systems of privacy-preserving verifiable neural network models. Taking an existing neural network as an input, ZEN produces a verifiable computation scheme for a classification task or a recognition task, namely ZEN_{class} and ZEN_{rec}. Both ZEN_{class} and ZEN_{rec} ensure the privacy, more precisely, the zero-knowledge property of the input data. In practice, this means removing the personal identifications, such as the facial image or other biometric data, from the attack surface. And thanks to three decades' consecutive efforts on zkSNARK from our community, the entire process is non-interactive and verifiable. Thus, our schemes potentially enable many important applications, ranging from trustless oracles for decentralized ledgers to privacy-preserving facial identification systems. To our best knowledge, ZEN is the first zero-knowledge neural network scheme that preserves the privacy of input data while delivering verifiable outputs.

To build efficient schemes with no additional accuracy loss, ZEN includes two major technical contributions. First, we propose a zkSNARK friendly quantization approach, which is semantically equivalent to the state-of-the-art quantization algorithm, yet brings significant savings in constraint size. Second, we propose a novel encoding scheme, namely stranded encoding, that encodes batched dot products, the workhorse of many matrix operations, using only a fraction of finite field elements. This brings sizable savings in terms of the number of constraints for the matrix operation circuits. Our end-to-end evaluation demonstrates the effectiveness of ZEN: compared with simply combining the state-of-the-art full quantization scheme with zkSNARK (ZEN-vanilla), ZEN has $3.68 \sim 20.99\times$ ($14.14\times$ on average) savings in the number of constraints (as a result, in prover time as well) thanks to our zkSNARK friendly quantization and stranded encoding.

1 Introduction

Neural network based software systems become integral parts of our daily life. While many of these systems, such as biometric-based recognition, bring us great convenience, they also become attack surfaces and result in privacy leakages. There is an increasing number of hacks and leakages on sensitive facial recognition data [Gau, O'F, She]. With on-going more adoptions of neural networks, the privacy leakage concern would be ever-growing. In major countries around the world, legislation efforts, such as GDPR, have been made to protect personal identifiable information. Privacy-preserving cryptographic methods have been a major candidate for legislation compliance.

In recent years, the advancement in zero-knowledge proof systems, especially zkSNARKs (zero-knowledge Succinct ARgument of Knowledge) [PHGR13, BCTV14, AHIV17, BBB+18, WTS+18, BBHR18, BCR+19, XZZ+19, BCG+20] makes efficient, verifiable while zero-knowledge computation possible. One natural question to ask is:

> *Can we leverage zero-knowledge proof systems to construct privacy-preserving, verifiable neural network schemes?*

Many prior works achieve privacy-preserving machine learning via secure multi-party computation (MPC) and/or homomorphic encryption (HE) [GDL+16, JVC18, DSC+19, JKLS18, MZ17, LJLA17, RRK18, SS15]. Zero-knowledge neural network schemes are also applicable to all these scenarios. Additionally, they also spark many new use cases due to the fact that (1) the neural network computation result will be publicly verifiable; (2) the whole process is non-interactive. For example, it can be used as a trustless oracle and a Know-Your-Customer (KYC) mechanism for decentralized ledgers [Wik]; it can also be used in privacy-preserving facial identification systems for verifiable auditability.

1.1 Our Contributions

In this paper, we present ZEN: a ZEro-knowledge proof system for Neural networks. ZEN is the first work in this domain that aims for privacy-preserving, verifiable inference. In short, ZEN makes the following contributions:

- ZEN *schemes:* The definitions of privacy-preserving, verifiable inference schemes based neural networks, that cover both classification and recognition, namely ZEN_{class} and ZEN_{rec}.
- *zkSNARK friendly quantization:* a series of zkSNARK friendly quantization algorithms that save the number of constraints in a proof system, while introducing no additional accuracy loss.
- *Optimizing matrix operations using stranded encoding:* a new encoding scheme for expressing matrix operations over finite fields, which leads to significant savings in the number of constraints in a proof system.
- ZEN *implementation:* An open-sourced toolchain (Fig. 1) that implements ZEN_{class} and ZEN_{rec} with zkSNARK friendly quantization and stranded encoded matrix operations. Our evaluation shows that ZEN brings $3.68 \sim 20.99\times$ ($14.14\times$ on average) savings in the number of constraints compared with a vanilla implementation of the neural network on zkSNARK.

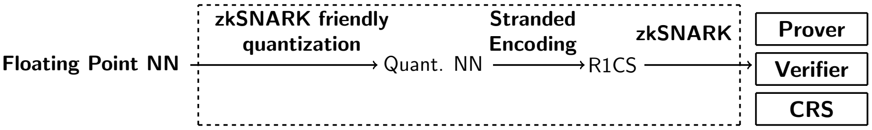

Fig. 1: Overview of ZEN tool chain.

1.2 Our Techniques

Privacy-preserving, verifiable inference schemes Based on potential use cases, we formally define two privacy-preserving, verifiable inference schemes, namely,

- ZEN_{class}, a verifiable classification scheme; and
- ZEN_{rec}, a verifiable recognition scheme.

Collectively, those two schemes cover the two most widely used tasks for neural networks (§3). Different from existing verifiable machine learning schemes such as zkDT [ZFZS20], SafetyNet [GGG17], and vCNN [LKKO20], both our schemes ensure that the results of classification and recognition are verifiable while the inputs (typically some personal identifiable biometric data) remain private; hence, compliant with legislation.

In both schemes, we assume that the prover and the verifier "agree" on the neural network model to be used. In our classification scheme (ZEN_{class}), the prover needs to convince the verifier that she has the secret input such that a certain classification result is obtained. Similarly, in our recognition scheme (ZEN_{rec}), we assume the prover and the verifier also "agree" on a reference "ground truth" embedding. For example, in the case of face recognition, this embedding can be viewed as the neural network model's output on a user's face image when setting up the recognition. However, the prover only knows a commitment to the reference embedding. This prevents the embedding from leaking personal identifiable biometric data [CJM20]. For this reason, ZEN_{rec} could be potentially used in privacy-preserving biometric-based authentication and trustless oracle for decentralized ledgers.

zkSNARK friendly quantization Prior to our work, a prohibitive factor of applying zero-knowledge proof systems over neural networks is the unfitted data type of neural network models. Popular neural network models use arithmetic operations on signed floating numbers and are therefore not compatible with existing zero-knowledge proof systems that operate over finite fields. Existing work on neural network quantization [JKC+18, DMM+18, MNA+18, SCC+19, LTA16] cannot be directly employed here for two reasons: first,

Table 1: Overall result

Dataset	**Model**	ZEN-vanilla **(K Constraints)**	ZEN **(K Constraints)**	**Saving** (×)	ZEN.prove **(s)**	ZEN.verify **(ms)**
MNIST	**ShallowNet**	408	111	3.68	7.01	5.51
CIFAR-10	**LeNet-5-small**	16,975	938	18.08	39.37	5.45
CIFAR-10	**LeNet-5-medium**	85,292	5,540	15.38	383.01	5.49
ORL	**LeNet-Face-small**	57,505	2,737	20.99	174.21	5.05
ORL	**LeNet-Face-medium**	274,643	17,506	15.67	1880.21	5.07
ORL	**LeNet-Face-large**	610,950	55,365	11.03	11334.31	5.17

part of the inference operations over the output of the quantized models remain on signed floating numbers; second, they still require operations such as divisions that are non-atomic to zkSNARK systems.

To overcome the first obstacle, we integrate a state-of-the-art full quantization algorithm [JKC+18] to the existing zkSNARK library as our baseline system: given a floating-point neural network, producing a neural network model consisting of only unsigned integer operations (§4.1). We show that although being fairly inefficient, one can already build a proof system for generic neural networks with the fully quantized neural network models. We call the proof systems generated by directly applying these fully quantized neural network models to an existing zkSNARK library ZEN-vanilla. It serves as an important stepping stone for the rest of the work.

We then introduced two important zkSNARK friendly optimizations (§4.2) to the full quantization algorithm, namely, *sign-bit grouping*, and *remainder-based verification*. The core idea behind both optimizations is to use *algebraic equalities* to avoid bit-decompositions, another non-atomic operation for zkSNARK, as possible. With sign-bit grouping, we completely eliminate the bit-decompositions due to element-wise zero-comparisons in each layer. With remainder-based verification, we reduce the bit-decompositions caused by divisions (due to the scale factor) in each layer. Both optimizations bring significant savings in terms of the number of constraints in the generated circuits. In addition, since these optimizations only use *algebraic equalities*, the resulted quantized neural network models are still equivalent to the quantized models from [JKC+18]. As a result, our solution does not incur any additional accuracy loss.

Optimizing matrix operation circuits using stranded encoding One crucial observation we have is: most quantized neural network works well with 8-bit unsigned integers, while most zkSNARKs use ellipitic curves (e.g. BLS12-381 [BLS02]) with an underlying finite field of order $\approx 2^{254}$; given such a field, can we encode matrix operations more efficiently by multiplexing a single finite field element for multiple elements in the matrix? This is analogous to the SIMD (Single Instruction Multiple Data) technique that is widely used in modern CPUs and GPUs [PH12].

The answer is affirmative. However, simply stacking 8-bit unsigned integers in finite field elements would not work. To solve this problem with subtlety, we propose a novel encoding scheme, namely *stranded encoding*, which encodes batched vector dot products with fewer field operations (§5). To compute s dot products simultaneously, i.e., given ($|A_j| = |B_j| = n, j \in \{1, \ldots, s\}$) as input, compute

$$(A_1 \cdot B_1), (A_2 \cdot B_2), \ldots, (A_s \cdot B_s),$$

a naïve encoding requires $2ns$ field elements. Our stranded encoding encodes these dot product operations with $2n$ field elements:

$$x_i = \sum_{j=1}^{s} a_{j,i} \delta^{\phi(j)}, \ y_i = \sum_{j=1}^{s} b_{j,i} \delta^{\phi(j)}$$

where $x_i, y_i \in \mathbb{F}_p, i \in \{1, \ldots, n\}$. For appropriate $\phi(\cdot)$ (see Definition 4), these s dot products could be extracted from the following quantity

$$\begin{aligned} \sum_{i=1}^{n} x_i y_i = (A_1 \cdot B_1)\delta^{2\phi(1)} + \ldots + (A_2 \cdot B_2)\delta^{2\phi(2)} + \\ \ldots + (A_s \cdot B_s)\delta^{2\phi(s)} \end{aligned}$$

Table 2: Case study: benefits of optimizations on the number of constraints for the four layers in LeNet-5-Medium CIFAR-10. Opt. Lv1 only includes sign-bit grouping; Opt. Lv2 adds remainder-based verification; Opt. Lv3 adds stranded encoding. See 6.1.

Kernels	ZEN-vanilla	Opt. Lv1	Opt. Lv2	Opt. Lv3
Conv	69,692,928	28,630,272	8,082,688	5,195,008
FC	394,000	219,706	132,490	54,906
AvgPool	14,925,312	4,982,976	7,872	7,872
ReLU	114,227	97,920	97,920	97,920

A caveat here is to pack as many matrix coefficients into a field element, while still allowing a proper $\phi(\cdot)$ to extract $A_1 \cdot B_1, \dots, A_s \cdot B_s$ correctly. We formally formulate this as a discrete optimization problem; and develop a cost model for stranded encoding, so that our implementation automatically chooses the optimal batch size s for any instance.

Implementation and evaluation We implemented both ZEN-vanilla and the fully optimized ZEN toolchain. We summarize our benchmark results in Table 1. We see an improvement of 3.67∼21.01× (14.27× on average), depending on the inference model. Further discussions will be given in the corresponding sections.

As one shall see, while the verification speed is more or less stable, the proving cost increases drastically with the increase of the number of constraints. We make various optimizations that reduce the number of constraints. See Table 2 for a highlight of optimizations on LeNet-5-Small for CIFAR-10.

Our code is open-sourced, publicly available on GitHub. [3]

Our underlying zero-knowledge proof scheme is from the celebrated work of Groth [Gro16]. We remark that the selection of underlying zero-knowledge proof systems is largely orthogonal to our ZEN design. In particular, our optimization is independent of the underlying proving system, and we expect to see similar gains from our optimizations for other proving systems. With [Gro16], our proof size is always a constant, i.e., 196 bytes for our choice of parameters. Verify a proof can be done within 5∼7 *ms* in all cases. This feature may be particularly appealing in practical use cases such as blockchains, where decisions (whether authentication passes or not) need to be made almost instantly.

1.3 Related work

MPC, FHE, DP for secure machine learning Many research efforts [CJM20, HSLA20, HSC+20, JL17, CKM17, GRSY20, ZFZS20] have been devoted to the security and privacy in machine learning recently. These works largely fall into three categories. The first approach [DSC+19, JKLS18, GDL+16] utilizes homomorphic encryption to execute machine learning models on encrypted data homomorphically. The second approach [MZCS18, BDK+18, DSZ15, JVC18] builds upon the multi-party computations (MPC), enabling multiple parties with local datasets to learn the same machine learning model for the aggregated datasets, while preserving privacy for individual's data. The third approach [ACG+16, ZYCW20, BNS19] adopts differential privacy (DP) to ensure that the individual data points in a large dataset will not be leaked even if they have been utilized to train a machine learning model.

While all these work provide privacy of the data during training and inferencing, they are largely orthogonal to our ZEN in that, on top of data privacy, we also aim to guarantee the integrity of computation by a succinct, non-interactive proof and an efficient verification.

In the zero-knowledge proof domain, prior to our work, [ZFZS20] explores zero-knowledge proof for machine learning. Their work has a limited scope and focuses on a simple decision tree model. [LKKO20] also uses zero-knowledge proofs for verifiable machine learning. There, the proof system is used to guarantee the integrity of the computation while zero-knowledge property of the input is absent. In other words, their input to the neural network model is neither binded nor hided. Our solution, to some extend, covers a superset of use cases in [LKKO20]. To the best of our knowledge, our work is the first efficient zero-knowledge proof for neural networks.

[3] https://github.com/UCSB-TDS/ZEN

Table 3: Comparison of zero-knowledge proof systems, where ($\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify}, |\pi|$) denote the trusted setup algorithm, the prover algorithm, the verification algorithm, and the proof size. C is the size of a log-space uniform circuit with depth d, n is the size of its input. The concrete numbers indicate the performance of a circuit computing the root of a Merkle tree of 256 leaves (the CRH used in this Merkle tree is SHA256) [4].

Construction	Gen	Prove	Verify	$\|\pi\|$	Gen	Prove	Verify	$\|\pi\|$
LibSNARK [BCTV14, BCG+20]	$O(C)$	$O(C \log C)$	$O(1)$	$O(1)$	1027s	360s	2×10^{-3}s	0.19KB
Ligero [AHIV17]	N.A.	$O(C \log C)$	$O(C)$	$O(\sqrt{C})$	N.A.	400s	4s	1,500KB
Bulletproofs [BBB+18]	N.A.	$O(C)$	$O(C)$	$O(\log C)$	N.A.	2,555s	98s	2KB
Hyrax [WTS+18]	N.A.	$O(C \log C)$	$O(\sqrt{n} + d \log C)$	$O(\sqrt{n} + d \log C)$	N.A.	1,041s	9.9s	185KB
LibSTARK [BBHR18]	N.A.	$O(C \log^2 C)$	$O(\log^2 C)$	$O(\log^2 C)$	N.A.	2,022s	0.044s	395KB
Aurora [BCR+19]	N.A.	$O(C \log C)$	$O(C)$	$O(\log^2 C)$	N.A.	3,199s	15.2s	174.3KB
Libra [XZZ+19]	$O(n)$	$O(C)$	$O(d \log C)$	$O(d \log C)$	210s	201s	0.71s	51KB

Zero-knowledge proof systems A large body of zero-knowledge proof systems [BCTV14, AHIV17, BBB+18, WTS+18, BBHR18, BCR+19, XZZ+19, BCG+20] have been proposed to facilitate the design and implementation of programs under the zero-knowledge setting. These systems usually show diverse prover time, verification time, and proof size, leading to trade-offs in these three dimensions. In particular, we focus on the scheme developed by Groth [Gro16] and the ark-snark implementation [ark, BCG+20] to provide constant size proofs and millisecond-level verifications. We stress again that the selection of zero-knowledge proof systems is largely orthogonal to our ZEN design; our design is applicable to other proof systems and will deliver various performance preference, suiting dedicated use cases.

Neural network quantization Quantizing neural networks [LDC+20, SCC+19, LTA16], i.e., reducing the data bit-width, is a common method in practice that reduces memory consumption and accelerates the intensive neural network computation. One popular approach [SCC+19, DMM+18, MNA+18] replaces the float32 data in neural network kernels with variable-precision data (*e.g.*, float16) and gain speedup during the training procedure. Note that floating-point arithmetic remains after quantization and will not be efficient in a zero-knowledge proof system. Another approach [JKC+18, GGSS19] turns to fixed-point models. They efficiently utilize integer-arithmetic instructions on ARM CPUs and mobile accelerators. However, these work either build upon huge look-up tables or require intensive division operations; both are expensive/prohibitive for zkSNARKs.

1.4 Paper organization

§2 presents necessary background for this paper. We introduce our privacy-preserving, verifiable classification and recognition schemes based on neural networks in §3. Next, §4 and §5 present our zkSNARK friendly quantization method and stranded encoding respectively. Finally, implementation and evaluations are reported in §6.

2 Background

The major cryptographic building block used in this paper is zkSNARK, formally *publicly-verifiable preprocessing zero-knowledge Succinct Non-interactive ARgument of Knowledge*. We briefly introduce zkSNARK in §2.1, and compare different schemes of zkSNARKs and explain our choice in §2.2.

2.1 What is a zkSNARK?

We informally define zkSNARK in the context of arithmetic circuit satisfiability. A more formal definition can be found in [BCI+13].

We denote a finite field of order p as $\mathbb{F}_p$. An $\mathbb{F}_p$-arithmetic circuit is a circuit whose inputs and outputs are from $\mathbb{F}_p$. We consider circuits that have an input $x \in \mathbb{F}_p^n$ and a witness $w \in \mathbb{F}_p^h$. We restrict the circuits to the ones with only *bilinear gates*, *i.e.* addition, multiplication, negation, and constant gates with input $y_1, \ldots, y_m$ is bilinear if the output is $\langle \vec{a}, (1, y_1, \ldots, y_m)\rangle \cdot \langle \vec{b}, (1, y_1, \ldots, y_m)\rangle$ for $\vec{a}, \vec{b} \in \mathbb{F}_p^{m+1}$.

It is not hard to convert boolean circuits to arithmetic circuits via bit decomposition. We define arithmetic circuit satisfiability as follows:

Definition 1. *The arithmetic circuit satisfiability of an $\mathbb{F}_p$-arithmetic circuit $C : \mathbb{F}_p^n \times \mathbb{F}_p^h \to \mathbb{F}_p^l$ can be defined by the relation $\mathcal{R}_C = \{(x, w) \in \mathbb{F}_p^n \times \mathbb{F}_p^h : C(x, w) = 0^l\}$ and the language $\mathcal{L}_C = \{x \in \mathbb{F}_p^n : \exists\ a \in \mathbb{F}^h$ s.t. $C(x, w) = 0^l\}$.*

A *zkSNARK* for $\mathbb{F}_p$-arithmetic circuit satisfiability is a triple of polynomial time algorithms, namely $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$:

- $\mathsf{Gen}(1^\lambda, C) \to (\mathsf{pk}, \mathsf{vk})$. Using a security parameter λ and an $\mathbb{F}_p$-arithmetic circuit C as inputs, the key generator Gen randomly samples a *proving key* pk and a *verification key* vk. These keys are considered as public parameters $\mathsf{pp} := (\mathsf{pk}, \mathsf{vk})$, and can be used any number of times to prove/verify the membership in $\mathcal{L}_C$.
- $\mathsf{Prove}(\mathsf{pk}, x, w) \to \pi$. Taking a proving key pk, and any $(x, w) \in R_C$ as inputs, the Prove algorithm generates a non-interactive proof π for the statement $x \in \mathcal{L}_C$.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to \{0, 1\}$. Taking the verification key vk, public input x, and proof π, the Verify algorithm output 1 is the verification success, *i.e.* the verifier is convinced that $x \in \mathcal{L}_C$.

Remark 1. In practice, the prover may segment the proving processing into multiple stages. For example, it may commit to an input and publish the commitment first, and at a later stage generates the proof. It may even commit different parts of the inputs independently and separately. +For simplicity, for the rest of the paper, we will model this whole process as a single function. It is straightforward to see that our security features remain intact in this simplified model.

A zk-SNARK has the following properties:

Completeness. For any security parameter λ, any $\mathbb{F}_p$ arithmetic circuit C, and $(x, w) \in \mathcal{R}_C$, an honest prover can convince the verifier, namely that the verifier will output 1 with probability $1 - negl(\lambda)$ in the following experiment: $(\mathsf{vk}, \mathsf{pk}) \leftarrow (1^\lambda, C)$; $\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$; $1 \leftarrow \mathsf{Verify}(\mathsf{vk}, x, \pi)$.

Succinctness. An honestly-generated proof π has $O_\lambda(1)$ bits and $\mathsf{Verify}(\mathsf{vk}, x, \pi)$ runs in time $O_\lambda(|x|)$ [5].

Proof of Knowledge. If the verifier accepts a proof output by a computationally bounded prover, the prover must know a witness for a given instance. This is also called *soundness against bounded provers*. More precisely, for every $poly(\lambda)$-size adversary $\mathcal{A}$, there is a $poly(\lambda)$-size extractor $\mathcal{E}$ such that $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1$ and $(x, w) \notin \mathcal{R}_C$ with probability $negl(\lambda)$ in the following experiment: $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(1^\lambda, C)$; $(x, \pi) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{vk})$; $w \leftarrow \mathcal{E}(\mathsf{pk}, \mathsf{vk})$.

Zero Knowledge. An honestly generated proof is zero knowledge. Specifically, there is a $poly(\lambda)$-size simulator Sim such that for all stateful $poly(\lambda)$-size distinguishers $\mathcal{D}$, the probability of $\mathcal{D}(\pi) = 1$ on an honest proof and on a simulated proof is indistinguishable.

2.2 zkSNARK schemes and implementations

zkSNARK's security can be reduced to knowledge-of-exponent and variants of Diffie-Hellman assumptions in bilinear groups [Gro10, BB04, Gen04]. Although the knowledge-of-exponent assumption is considered fairly strong, Gentry and Wichs showed that assumptions from this class are likely to be inherent for efficient, non-interactive arguments for NP relations [GW11].

There are a number of zero-knowledge proof systems proposed in recent years [BCTV14, AHIV17, BBB+18, WTS+18, BBHR18, BCR+19, XZZ+19, BCG+20]. Table 3 lists some of them and compares the trade-offs of different systems. In this paper, we use the ark-snark implementation [ark] that was part of

[4] The concrete numbers are from [XZZ+19].

[5] $O_\lambda(\cdot)$ hides a fixed polynomial factor in λ.

[BCG+20]. We will use the scheme by Groth [Gro16], commonly referred to as Groth16, to generate and verify proofs. This scheme is state-of-the-art in terms of proof size and verifier efficiency. Nonetheless, we reiterate that our work can be easily adapted to other proof systems at the back-end and the proposed front-end optimizations remain effective.

2.3 Neural network based classification and recognition

Neural network (NN) is an important type of machine learning algorithm with wide applications in computer vision [HZRS16, LBBH98] and natural language processing [DS20, WJI+20]. It generally contains a sequence of neural network layers to extract features and make predictions. During computation, these layers are usually executed sequentially and each layer takes input from the computation results of the previous layer. Formally, given a sequence of neural network layer $f_i(\cdot|W_i)$ that is parameterized by the weight W_i, each layer consumes an input feature X_{i-1} from the previous layer and generates a feature map X_i

$$X_i = f_i(X_{i-1}|W_i), i \in \{1, 2, ..., n\}$$

Here, X_0 is the input image and n is the number of neural network layers. To extract non-linear features, an activation function is usually applied after each neural network layer to introduce nonlinearity. The most popular one is the ReLU function that element-wisely compares features with zero ($i.e.$, $\text{ReLU}(X_i) = max(X_i, 0)$).

Two important workloads of NNs are the classification [HZRS16, LBBH98] and the recognition [SYS+20, WWZG20]. While these two workloads enjoy the same performance benefit from neural network layers, they are largely differentiated by the final layer for prediction. Given c candidate classes ($e.g.$, cat, dog, and house), the classification task aims to classify an image into one of these classes by predicting directly the probabilities of each class. Formally, given a NN with n layers, the classification task takes the features $X_n \in \mathcal{R}^c$ from the n^{th} layer and selects the prediction $\hat{y} = \text{argmax}_i X_n$ as the class i with the highest predicted probability.

The recognition task aims to compare two images and identify whether these two images belong to a same category. It is widely used for facial recognition that compares two images and decides whether these two images contain a same person. This task usually extracts the high-level features of two images and computes their distance for comparison. In particular, given the features $X_n^{(1)}$ and $X_n^{(2)}$ from two images, the recognition flags them as the same category if their distance is larger than a pre-defined threshold (=0.5 by default). Note that the recognition task usually uses a large dimension of X_n to maintain fine-grained features for comparison.

3 Zero-Knowledge Proofs for Neural Network based Classification and Recognition

In this section, we present our constructions of verifiable neural network based classification and recognition with zero knowledge.

A neural network, $\mathcal{N} : V_1 \to V_2$, is a function from one vector space to another. In practice, both V_1 and V_2 are defined on floating-point numbers, such as float16 or float32. To embed a neural network to arithmetic circuits, we propose a zkSNARK friendly quantization. This allows us to convert a neural network, defined on floating-point numbers, to a *fully quantized neural network*, where both the feature map and the weight of all neurons are unsigned integers, and thus could be easily embedded in finite fields. A fully quantized neural network, $\mathcal{Q} : \mathbb{F}_p^{d_1} \to \mathbb{F}_p^{d_2}$, is a mapping from a size d_1 vector over $\mathbb{F}_p$ to a size d_2 vector on the same field. A neural network model can be trained for different tasks, such as classification, regression, and recognition.

Apart from zk-SNARK, we use a cryptographic commitment scheme as a building block, formally, $\mathsf{COMM} : \{0,1\}^{O(\lambda)} \times \{0,1\}^* \to \{0,1\}^{O(\lambda)}$. We require the scheme to be both binding and hiding. Looking ahead, we will be using Pedersen commit which is statistically hiding and computationally binding under well-accepted assumptions.

3.1 ZEN$_{class}$: zero-knowledge verifiable neural network based classification

Motivated by the application in the introduction, our zero-knowledge verifiable neural network classification scheme (ZEN$_{class}$) assumes that the prover and the verifier agree on a neural network model prior to communication. This includes the network architecture, weight of each layer, and the choice of activation functions, etc. During our instantiation, we use Groth16 method to generate proofs. This implies that the prover and the verifier also need to agree on certain common reference string (CRS) that depends on the network model. During the whole process, the prover keeps the input private, but will publicly commit to the input data (binding). At a later time, the prover generates a proof for the result of the classification. Upon receiving the commitment and the proof, the verifier decides if the proof is valid or not. During the whole process, the prover's input data is kept secret (hiding).

Let $\mathcal{Q}$ be a quantized neural network that represents the mapping $\mathcal{Q} : \mathbb{F}_p^d \to [M]$, where $[M]$ is the set of all possible classifications. A zero-knowledge verifiable neural network based classification scheme (ZEN$_{class}$) can be defined as the following algorithms:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{class}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$: given a security parameter λ and a quantized neural network model $\mathcal{Q}$ for classification, randomly generate a proving key pk and a verification key vk.
- $(cm, y_a, \pi) \leftarrow \text{ZEN}_{class}.\mathsf{Prove}(\mathsf{pk}, a, r)$: given an input $a \in \mathbb{F}_p^d$ and a random opening r, the prover commits to the input with r, i.e., $cm \leftarrow \mathsf{COMM}(r, a)$, and runs the neural network classification to get $y_a \leftarrow \mathcal{Q}(a)$. Finally, the prover generates a proof for the above process.
- $\{0, 1\} \leftarrow \text{ZEN}_{class}.\mathsf{Verify}(\mathsf{vk}, cm, y_a, \pi)$: validate input a's classification result on model $\mathcal{Q}$ given the verification key vk, a's commitment, a's classification result y_a, and the zero-knowledge proof π.

π is a zk-SNARK proof that the prover produces for the following NP statement:

Protocol 1 (NP statement for ZEN$_{class}$) *Given a commitment of input cm, "I" know a secret input a and a secret opening r such that:*

- *The commitment is well-formed: $cm = \mathsf{COMM}(r, a)$.*
- *The classification result is valid: $y_a = \mathcal{Q}(a)$.*

We define ZEN$_{class}$ formally as follows:

Definition 2 (ZEN$_{class}$). *A scheme is a zero-knowledge verifiable neural network based classification if the following holds:*

- ***Completeness.*** *For any quantized neural network $\mathcal{Q}$ and an input $a \in \mathbb{F}_p^d$, $(\mathsf{pk}, \mathsf{vk}) \leftarrow ZEN_{class}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$, $(cm, y_a, \pi) \leftarrow ZEN_{class}.\mathsf{Prove}(\mathsf{pk}, a, r)$, it holds that:*

$$\Pr[ZEN_{class}.\mathsf{Verify}(\mathsf{vk}, cm, y_a, \pi) = 1] = 1$$

- ***Soundness.*** *For any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr\left[\begin{array}{l} (\mathsf{vk}, \mathsf{pk}) \leftarrow ZEN_{class}.\mathsf{Gen}(1^\lambda, \mathcal{Q}), \\ cm' \leftarrow \mathsf{COMM}(r, a), \\ (a', cm', y'_a, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pk}, \mathsf{vk}, r'), \\ 1 \leftarrow ZEN_{class}.\mathsf{Verify}(\mathsf{vk}, cm', y'_a, \pi'), \\ \mathcal{Q}(a') \neq y_a \quad for \quad a' \neq a \end{array}\right]$$

- ***Zero Knowledge.*** *An honestly-generated proof is perfect zero knowledge. For security parameter λ, $(\mathsf{pk}, \mathsf{vk}) \leftarrow ZEN_{class}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$, PPT distinguisher $\mathcal{D}$, there exists a PPT simulator Sim such that the following probabilities are indistinguishable (at most differs by negl(λ)):*
  - *The probability that $\mathcal{D}(cm, y_a, \pi) = 1$ on an honest proof:*

$$\Pr\left[\begin{array}{l|l} & (\mathsf{pk}, \mathsf{vk}) \leftarrow \\ & \quad ZEN_{class}.\mathsf{Gen}(1^\lambda, \mathcal{Q}), \\ \mathcal{D}(cm, y_a, \pi) & (a, r) \leftarrow \mathcal{D}(\mathsf{pk}, \mathsf{vk}), \\ = 1 & (cm, y_a, \pi) \leftarrow \\ & \quad ZEN_{class}.\mathsf{Prove}(\mathsf{pk}, a, r) \end{array}\right]$$

- *The probability that $\mathcal{D}(cm, y_a, \pi) = 1$ on a simulated proof:*

$$\Pr\left[\begin{array}{l|l} \mathcal{D}(cm, y_a, \pi) & (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Sim}(1^\lambda, \mathcal{Q}), \\ = 1 & (a, r) \leftarrow \mathcal{D}(\mathsf{pk}, \mathsf{vk}), \\ & (cm, y_a, \pi) \leftarrow \\ & \quad ZEN_{class}.\mathsf{Sim}(\mathsf{pk}, a, r) \end{array}\right]$$

3.2 ZEN$_{rec}$: zero-knowledge verifiable neural network based recognition

A second use case of neural networks that we study in this paper is recognition. Neural network based recognition tasks usually consist of two steps. Taking face recognition as an example, the first step is to use a neural network to map an input face image a to a face embedding, represented by a vector y_a. The second step is to compare this face embedding with a so-called ground truth embedding y_g via a distance metric $\mathcal{L}$. If the loss is smaller than certain specified threshold ($\mathcal{L}(y_a, y_g) \leq \tau$), then the recognition succeeds.

As motivated in the introduction, zero-knowledge verifiable neural network based recognition can be applied to many scenarios to preserve the privacy of sensitive input, such as biometrics information. Our zero-knowledge verifiable neural network based recognition (ZEN$_{class}$) scheme assumes that the prover and the verifier agree on a neural network model, a commitment to the ground truth embedding $cm_g \leftarrow \mathsf{COMM}(s, y_g)$ where y_g is the ground truth embedding and s is some random opening, the distance metric $\mathcal{L}$, and the threshold τ. Similar to the previous case, they also agree on a CRS for our proving system. The prover generates a proof to convince the verifier that she has a secret input y_a such that the output of the distance metric $\mathcal{L}$, evaluated over both the y_a and the agreed and committed ground true y_g, is less than or equal to τ. Again, the prover cannot alter her inputs once committed (binding), while her inputs remain private (hiding) during the whole process.

Let $\mathcal{Q}$ be the quantized neural network, represented via a mapping $\mathcal{Q} : \mathbb{F}_p^{d_1} \rightarrow \mathbb{F}_p^{d_2}$. Let $\mathcal{L} : \mathbb{F}_p^{d_2} \times \mathbb{F}_p^{d_2} \rightarrow \mathbb{F}_p$ be a distance metric over the embedding space, and $\tau \in \mathbb{F}_p$ be the agreed threshold. A zero-knowledge verifiable neural network based recognition scheme (ZEN$_{rec}$) can be defined as the following algorithms:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow$ ZEN$_{rec}$.$\mathsf{Gen}(1^\lambda, \mathcal{Q})$: given a security parameter λ and a quantized neural network model $\mathcal{Q}$ for recognition, randomly generate a proving key pk and a verification key vk.
- $(cm_g, cm_1, cm_2, \pi) \leftarrow$ ZEN$_{rec}$.$\mathsf{Prove}(\mathsf{pk}, a, y_g, r_1, r_2, s)$:
  given an input a, the ground truth embedding y_g, and some random secrets r_1, r_2 and s, the prover runs the neural network model $\mathcal{Q}$ on a to obtain the embedding result y_a, commits the input a to $cm_1 \leftarrow \mathsf{COMM}(r_1, a)$ and the output y_a to $cm_2 \leftarrow \mathsf{COMM}(r_2, y_a)$, respectively. The prover also commits the ground truth embedding y_a to $cm_g \leftarrow \mathsf{COMM}(y_a, s)$ using s. The prover generates a proof π for verifying these computations as well as $\mathcal{L}(y_a, y_g) \leq \tau$.
- $\{0, 1\} \leftarrow$ ZEN$_{rec}$.$\mathsf{Verify}(\mathsf{vk}, cm_g, cm_1, cm_2, \pi)$: validate the recognition result, given the commitment of the ground truth embedding (cm_g) the commitment of input a (cm_1), the commitment of a's neural embedding y_a (cm_2), and the zero-knowledge proof π.

Informally, π is a zero-knowledge proof of the following statement:

Protocol 2 (NP statement for ZEN$_{rec}$) *Given commitments cm_1, cm_2 and cm_g, "I" know some secret inputs a, y_a, y_g, r_1, r_2 and s such that:*

- *The following commitments are well-formed:*

$$\begin{aligned} cm_1 &= \mathsf{COMM}(r_1, a), \\ cm_2 &= \mathsf{COMM}(r_2, y_a), \\ cm_g &= \mathsf{COMM}(s, y_g) \end{aligned}$$

- *The neural embedding is valid: $y_a = \mathcal{Q}(a)$.*
- *The distance between a's embedding and the ground truth embedding is not greater than the threshold τ:*

$$\mathcal{L}(y_a, y_g) \leq \tau$$

we define ZEN$_{rec}$ formally as follows:

Definition 3 (ZEN$_{rec}$). *A scheme is a zero-knowledge verifiable neural network based recognition if the following holds:*

- ***Completeness.*** *For any quantized neural network $\mathcal{Q}$ and an input $a \in \mathbb{F}_p^d$, $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{rec}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$, $(cm_g, cm_1, cm_2, \pi) \leftarrow \text{ZEN}_{rec}.\mathsf{Prove}(\mathsf{pk}, a, y_g, r_1, r_2, s)$, it must hold that:*

$$\Pr[\text{ZEN}_{rec}.\mathsf{Verify}(\mathsf{vk}, cm_g, cm_1, cm_2, \pi) = 1] = 1$$

- ***Soundness.*** *For any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr\left[\begin{array}{l}(\mathsf{vk}, \mathsf{pk}) \leftarrow \text{ZEN}_{rec}.\mathsf{Gen}(1^\lambda, \mathcal{Q}),\\ (a', y_a', y_g', cm_g', cm_1', cm_2', \pi') \leftarrow\\ \quad \mathcal{A}(1^\lambda, \mathsf{pk}, \mathsf{vk}, r_1', r_2', s'),\\ cm_g' = \mathsf{COMM}(s', y_g'),\\ cm_1' = \mathsf{COMM}(r_1', a'),\\ cm_2' = \mathsf{COMM}(r_2', y_a'),\\ 1 \leftarrow \text{ZEN}_{rec}.\mathsf{Verify}(\mathsf{vk}, cm_g', cm_1', cm_2', \pi'),\\ \mathcal{Q}(a') = y_a,\\ \mathcal{L}(y_a, y_g) > \tau\end{array}\right]$$

- ***Zero Knowledge.*** *An honestly-generated proof is perfect zero knowledge. For security parameter λ, $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{rec}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$, PPT distinguisher $\mathcal{D}$, there exists a PPT simulator* Sim *such that the following probabilities are indistinguishable (at most differs by negl(λ)):*
  - *The probability that $\mathcal{D}(cm_g, cm_1, cm_2, \pi) = 1$ on an honest proof:*

$$\Pr\left[\begin{array}{c|l}\mathcal{D}(cm_g, cm_1, & (\mathsf{pk}, \mathsf{vk}) \leftarrow\\ cm_2, \pi) & \quad \text{ZEN}_{rec}.\mathsf{Gen}(1^\lambda, \mathcal{Q}),\\ = 1 & (a, y_g, r_1, r_2, s) \leftarrow \mathcal{D}(\mathsf{pk}, \mathsf{vk}),\\ & (cm_g, cm_1, cm_2, \pi) \leftarrow\\ & \quad \text{ZEN}_{rec}.\mathsf{Prove}(\mathsf{pk}, a, y_g, r_1, r_2, s)\end{array}\right]$$

  - *The probability that $\mathcal{D}(cm_g, cm_1, cm_2) = 1$ on a simulated proof:*

$$\Pr\left[\begin{array}{c|l}\mathcal{D}(cm_g, cm_1, & (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Sim}(1^\lambda, \mathcal{Q}),\\ cm_2, \pi) = 1 & (a, y_g, r_1, r_2, s) \leftarrow \mathcal{D}(\mathsf{pk}, \mathsf{vk}),\\ & (cm_g, cm_1, cm_2, \pi) \leftarrow\\ & \quad \mathsf{Sim}(\mathsf{pk}, a, y_g, r_1, r_2, s)\end{array}\right]$$

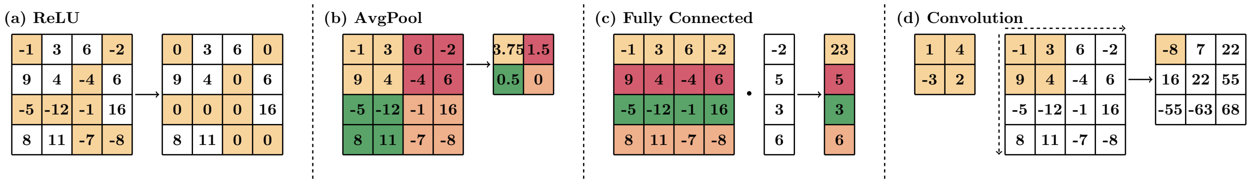

Fig. 2: Popular neural network kernels

4 zkSNARK Friendly Quantizaiton

In this section, we introduce our zkSNARK friendly quantization to fit floating-point neural network models to zkSNARK. Recall that popular neural networks require arithmetic computation on floating-point numbers (*e.g.*, float32). However, zero-knowledge systems work over finite fields where data is represented by large unsigned integers (*e.g.*, uint256). This poses special challenges on the operation and data, namely, converting floating points to non-negative integers, and handling divisions. To bridge this gap of numerical data types, we first integrate a full quantization scheme in [JKC$^+$18] to the existing zkSNARK library as our baseline system, ZEN-vanilla. Then, we introduce two zkSNARK friendly optimizations on top of our baseline system, to significantly reduce the number of constraints while maintaining an equivalent accuracy.

4.1 Our baseline quantization scheme

Amid a vast amount of existing work on neural network quantization [JKC+18, DMM+18, MNA+18, SCC+19, LTA16], applying neural network models in zkSNARKs requires full quantization: converting a floating-point neural network model to a neural network model consisting of only unsigned integer arithmetic. As a result, we choose the full quantization scheme in [JKC+18] as our baseline.

Given a weight matrix W and a data matrix X, a neural network kernel computes the output matrix Y as follows:

$$Y = WX, \quad W \in \mathbb{R}^{m \times n}, X \in \mathbb{R}^n, Y \in \mathbb{R}^m \tag{1}$$

The first step of quantization is to generate floating-point scale parameters ($s_Y, s_W, s_X \in \mathbb{R}$) and the lifted zero points ($z_Y, z_W, z_X \in \texttt{uint}$) for each matrix. As a result, we have the quantized representation[6]:

$$\begin{aligned} Y &= s_Y(Q_Y - z_Y\mathbf{J}_{m,1}) \quad W = s_W(Q_W - z_W\mathbf{J}_{m,n}) \\ X &= s_X(Q_X - z_X\mathbf{J}_{n,1}) \end{aligned}$$

During neural network computation, we can substitute Y, X, and W in Eq. 1 with their quantized representation:

$$s_Y(Q_Y - z_Y\mathbf{J}_{m,1}) = s_W s_X(Q_W - z_W\mathbf{J}_{m,n})(Q_X - z_X\mathbf{J}_{n,1})$$

The second step is to replace the floating-point scale parameters with unsigned integers and enable the full quantization computation:

$$\begin{aligned} M &= \lfloor 2^k \frac{s_W s_X}{s_Y} \rfloor \\ Q_Y - z_Y\mathbf{J}_{m,1} &= M(Q_W - z_W\mathbf{J}_{m,n})(Q_X - z_X\mathbf{J}_{n,1})/2^k \end{aligned} \tag{2}$$

By multiplying with 2^k with a large k (=22 by default), we preserve the precision of the floating-point scale parameters in an unsigned integer.

This quantization scheme establishes state-of-the-art accuracy on `uint8` quantization without introducing any additional accuracy loss, for a variety of real-world neural networks. We implement this quantization with the existing zkSNARK library as our baseline system, ZEN-vanilla.

4.2 Improving quantization using sign-bit grouping and remainder-based verification

ZEN-vanilla produces effective privacy-preserving, verifiable neural network models. However, ZEN-vanilla is not efficient due to its large number of constraints. We further introduce two zkSNARK friendly optimizations on top of the baseline quantization scheme, namely, sign-bit grouping and remainder-based verification. Both optimizations use algebraic equalities to reduce the number of expensive bit-decomposition operations in zkSNARK, while maintaining the semantics of the quantization. As a result, our techniques incur similar accuracy loss as [JKC+18]. Nonetheless, we note that [JKC+18] itself introduces accuracy loss.

Sign-bit grouping. In ZEN-vanilla, the constraints for a forward step on each layer is generated by Eq. 2. We first reformulate Eq. 2 to:

$$Q_Y = \; z_Y\mathbf{J}_{m,1} + M(Q_W - z_W\mathbf{J}_{m,n})(Q_X - z_X\mathbf{J}_{n,1})/2^k$$

Despite that both sides are guaranteed to be positive, each element of $(Q_W - z_W)$ and $(Q_X - z_X)$ may still be negative. As a result, one may still need $O(mn)$ sign-checks in the generated constraints, where each sign-check needs expensive bit-decomposition.

Instead, we use *sign-bit grouping* that uses the associativity of matrix multiplication to group operands of the same sign:

$$\begin{aligned} G_1 = Q_W Q_X,\ G_2 = z_X Q_W,\ G_3 = z_W Q_X,\ M' &= \lfloor \frac{z_Y 2^k}{M} \rfloor \\ Q_Y = M(G_1 + n z_W z_X \mathbf{J}_{m,1} + M'\mathbf{J}_{m,1} &- G_2 - G_3)/2^k \end{aligned} \tag{3}$$

[6] $\mathbf{J}_{k,l}$ represents a $k \times l$ matrix of ones.

Note that we add z_Y before subtraction such that all intermediate elements in the reformulated system is guaranteed to be positive. Now we can directly encode Eq. 3 on the finite field without the need of any sign check, and thus completely remove bit-decompositions. This optimization saves $O(mn \log p)$ constraints for $W \in \mathbb{R}^{m \times n}$, where p is the order of the finite field used by zkSNARK.

Remainder-based verification. Observe that in Eq. 3, we still need to perform divisions. However, the division operation is non-atomic in zkSNARK systems. To naïvely support this division operation, we need to first conduct the expensive bit-decomposition. Then, we need to drop the n least significant bits and pack the rest back to enforce equality in Eq. 3. While this strategy allows verifying Eq. 3, it would introduce heavy overhead from the bit decomposition.

By contrast, we propose a *remainder-based verification optimization* to avoid the high overhead from bit decomposition. We first use an extra matrix R to store the division remainder. During verification, we will utilize this remainder matrix to avoid the division in zkSNARK systems. Formally, we have the following verification procedure:

$$Q_Y 2^k + R = M(G_1 + nz_W z_X \mathbf{J}_{m,1} + M' \mathbf{J}_{m,1} - G_2 - G_3) \tag{4}$$

As a result, we can verify the computation without the need of any division operations. This optimization saves $O(m \log p)$ constraints ($Y \in \mathbb{R}^m$, p is the order of the finite field used by zkSNARK).

4.3 Apply zkSNARK friendly quantization to different kernels

We introduce the quantization of individual neural network kernels. We illustrate these neural network kernels in Fig. 2. Since the fully connected kernels and convolution kernels can be implemented as dot products, we can directly use Eq. 4 to verify these two kernels. In the following section, we will show the quantization of the ReLU kernel and the average pool kernel.

ReLU kernel. The ReLU kernel contains only maximum operations, and is used to extract nonlinear neural network features, as illustrated in Fig. 2(a). Formally, given a quantized matrix represented in a triple (Q, s, z), where Q is the quantized matrix ($Q \in \mathbb{N}^{c_{in} \times m \times n}$), s is the scale parameter ($s \in \mathbb{R}$), and z is the zero point $z \in \mathbb{N}$. We compute the ReLU kernel by element-wisely applying the maximum comparison

$$Q_{\text{ReLU}} = \max(Q, z\mathbf{J}_{c_{in},m,n}))$$

The key insight is that z is an integer value corresponding to the lifted zero in the floating-point data. Note that this design involves only integer arithmetics, and avoids the conversion between floating-point and integer completely.

Average pool kernel. The average pool kernel computes the average values among a set of integers. It is useful to summarize neural network features across spatial dimensions, as illustrated in Fig. 2(b). Formally, given a matrix in quantized representation (Q, s, z) ($Q \in \mathbb{N}^{c_{in} \times m \times n}$), and a pooling parameter r, the average pooling operator splits the data into a set of $r \times r$ grid and computes the average in each grid

$$\bar{q}_{c,i,j} = \sum_{p=0}^{r-1} \sum_{t=0}^{r-1} q_{c,ri+p,rj+t}/r^2, c \in \{1, 2, ..., c_{in}\}$$
$$i \in \{1, 2, ..., \lfloor \frac{m}{r} \rfloor\}, j \in \{1, 2, ..., \lfloor \frac{n}{r} \rfloor\}$$

We observe that this kernel cannot be easily supported in zkSNARK systems. First, the average pooling operator contains division operation, which is not generally supported in zkSNARK systems. Second, even if division for certain pooling parameters (*e.g.*, $r = 2$) can be conducted with bit operations, it may still lead to non-integer outputs after division. To this end, we incorporate the remainder-based verification strategy to average pool kernel. In particular, we first use an extra scalar γ to store the division remainder and use the following verification for the average pooling layer

$$\bar{q}_{c,i,j} r^2 + \gamma = \sum_{p=0}^{r-1} \sum_{t=0}^{r-1} q_{c,ri+p,rj+t}, c \in \{1, 2, ..., c_{in}\}$$
$$i \in \{1, 2, ..., \lfloor \frac{m}{r} \rfloor\}, j \in \{1, 2, ..., \lfloor \frac{n}{r} \rfloor\}$$

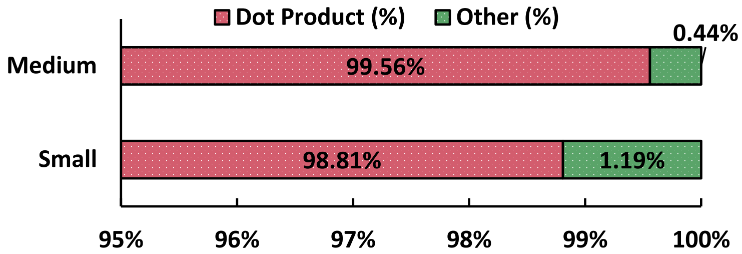

Fig. 3: Dot product/other computation Ratio in LeNet-5-Medium and LeNet-5-Small.

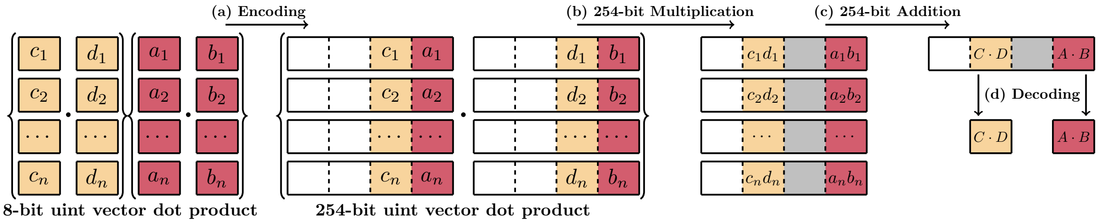

Fig. 4: Stranded encoding with batch size $s = 2$

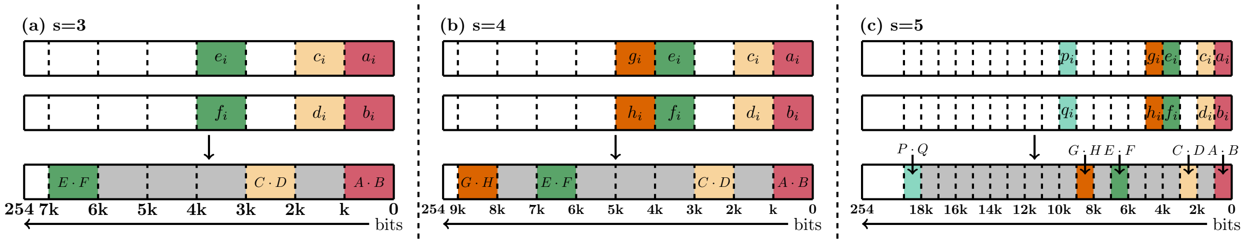
(a) s=3
(b) s=4
(c) s=5
e_i c_i a_i
f_i d_i b_i
$E \cdot F$ $C \cdot D$ $A \cdot B$
g_i h_i $G \cdot H$
p_i q_i $P \cdot Q$
254 7k 6k 5k 4k 3k 2k k 0
254 9k 8k 7k 6k 5k 4k 3k 2k k 0
254 18k 16k 14k 12k 10k 8k 6k 4k 2k 0
bits

Fig. 5: Data layout for variable batch size s. Each block has k bits and can store a value up to 2^k. k is a hyperparameter that is generally larger than 8 to accumulate the dot product and avoid overflow.

5 Optimizing Matrix Operation Circuits using Stranded Encoding

In this section, we propose *stranded encoding*, a general methodology of optimizing matrix operation circuits for zkSNARKs. Our profiling on neural networks shows that matrix operation, especially dot products, consumes most computation in neural networks, as shown in Fig. 3.

One important observation that we make is: neural network models can be effectively quantized to models consisting of small integers, such as uint8 or uint16, while the underlying finite field is usually much larger (e.g. $\approx 2^{254}$ in case of BLS12-381 [BLS02]). From this observation, we propose a new encoding scheme, namely stranded encoding, that could encode multiple low precision integers in the larger finite fields.

Naïve encoding. One intuitive solution is, to encode $A \cdot B$ where $A = [a_1, a_2], B = [b_1, b_2], a_i, b_i \in \texttt{uint8}$, we can use finite field elements x and y ($x, y \in \mathbb{F}_p, p \geq 2^{16}$) to encode A and B:

$$x = a_1 + a_2\delta$$
$$y = b_1 + b_2\delta$$

where $\delta \geq 2^{16}$. This encoding is already additive homomorphic, i.e., $A + B = [a_1 + b_1, a_2 + b_2]$ since $x + y = (a_1 + b_1) + (a_2 + b_2)\delta$, from which $a_1 + b_1$ and $a_2 + b_2$ can be easily extracted. This naïve encoding is not multiplicative homomorphic though. Take dot product computation $A \cdot B = a_1b_1 + a_2b_2$ as an example. We know that

$$xy = a_1b_1 + (a_1b_2 + a_2b_1)\delta + a_2b_2\delta^2$$

To get $A \cdot B$, we need to extract each $a_i b_i$ separately from xy. This costs $O(n), n = |A| = |B|$, which defeats the purpose of the encoding.

Stranded encoding. To address this problem, we propose a stranded encoding of low precision integers in finite field elements. The core idea of stranded encoding is to encode multiple matrix operations at the same time. For example, to better encode $A \cdot B = a_1 b_1 + a_2 b_2$ and $C \cdot D = c_1 d_1 + c_2 d_2$, we could first encode the low-precision integers in finite fields as follow:

$$\begin{aligned} x_1 &= a_1 + c_1\delta \quad & x_2 &= a_2 + c_2\delta \\ y_1 &= b_1 + d_1\delta \quad & y_2 &= b_2 + d_2\delta \end{aligned}$$

with sufficiently large δ ($\delta \geq 2^{17}$). Now, $A \cdot B$ and $C \cdot D$ can be all easily extracted from $\sum x_i y_i$, since:

$$\begin{aligned} x_1 y_1 &= a_1 b_1 + (a_1 d_1 + c_1 d_1)\delta + c_1 d_1 \delta^2 \\ x_2 y_2 &= a_2 b_2 + (a_2 d_2 + c_2 d_2)\delta + c_2 d_2 \delta^2 \\ x_1 y_1 + x_2 y_2 &= (a_1 b_1 + a_2 b_2) + (\ldots)\delta \\ &\quad + (c_1 d_1 + c_2 d_2)\delta^2 \end{aligned}$$

We can extract $A \cdot B$ and $C \cdot D$ from $x_1 y_1 + x_2 y_2$ by mod δ and extracting the lowest 9 bits.

It is not hard to see that this stranded encoding can be easily extended to the case that vector length $|A| = |B| = |C| = |D| = n > 2$, as illustrated in Fig. 4:

$$A \cdot B = \sum_{i=1}^{n} a_i b_i, \quad C \cdot D = \sum_{i=1}^{n} c_i d_i$$

We encode x_i and y_i as:

$$\begin{aligned} x_i &= a_i + c_i\delta, \quad i \in \{1, 2, \ldots, n\} \\ y_i &= b_i + d_i\delta, \quad i \in \{1, 2, \ldots, n\} \end{aligned}$$

Here, we set $\delta = 2^k, k = 2w_{in} + \log n$, where w_{in} is the bit width of the low precision unsigned integer and n is the size of the vector. We need to add n here to catch the possible overflow of accumulating n w_{in}-bit unsigned integers. Now we have:

$$\begin{aligned} \sum_{i=1}^{n} x_i y_i &= \sum_{i=1}^{n} a_i b_i + (\ldots)\delta + \left(\sum_{i=1}^{n} c_i d_i\right)\delta^2 \\ &= A \cdot B + (\ldots)\delta + (C \cdot D)\delta^2 \end{aligned} \tag{5}$$

Finally, we can decode the dot products $A \cdot B$ and $C \cdot D$ with bit operations

$$\begin{aligned} A \cdot B &= \left(\sum_{i=1}^{n} x_i y_i\right) \bmod \delta \\ C \cdot D &= \left(\sum_{i=1}^{n} x_i y_i\right) \gg 2k \end{aligned}$$

where *mod* is the module operation and $\gg 2k$ indicates right-shift by $2k$ bits. While the decoding adds more overhead, this overhead is amortized as we increase the number of batched operations in stranded encoding.

In the above example, we batch two dot product operations: $A \cdot B$, $C \cdot D$. We will discuss how to extend stranded encoding with a larger batch size next.

Stranded encoding with arbitrary batch sizes. Let batch size s be the number of batched dot product operations. To achieve further saving in the constraint size, we would like to extend s from 2 to larger batch sizes.

However, a naïve extension of the stranded encoding when batch size $s = 2$ would not work. For example, for $s = 3$, we encode $A \cdot B$, $C \cdot D$, $E \cdot F$ ($A = [a_1, \dots, a_n]$ etc.) as follows:

$$\begin{aligned} x_i &= a_i + c_i\delta + e_i\delta^2 \\ y_i &= b_i + d_i\delta + f_i\delta^2 \end{aligned}$$

Then, the multiplication becomes:

$$\begin{aligned} x_iy_i =& a_ib_i + (a_id_i + b_ic_i)\delta + (c_id_i + a_if_i + b_ie_i)\delta^2 \\ &+ (c_if_i + d_ie_i)\delta^3 + e_if_i\delta^4 \end{aligned}$$

It becomes very difficult to extract c_id_i from x_iy_i since c_id_i is "mixed" in the coefficient of δ^2. To solve this problem, we use the following encoding instead for $i \in \{1, ..., n\}$:

$$\begin{aligned} x_i &= a_i + c_i\delta + e_i\delta^3 \\ y_i &= b_i + d_i\delta + f_i\delta^3 \end{aligned}$$

Now it is not hard to see:

$$x_iy_i = a_ib_i + (\dots)\delta + (c_id_i)\delta^2 + \dots + (e_if_i)\delta^6 \tag{6}$$

As a result:

$$\sum_{i=1}^{n} x_iy_i = A \cdot B + (...)\delta + (C \cdot D)\delta^2 + \dots + (E \cdot F)\delta^6$$

In fact, stranded encoding can be generalized to an arbitrary batch size s (with constraints). Formally, we define stranded encoding as follows:

Definition 4 (Stranded encoding scheme). *For a series of dot product $A_1 \cdot B_1, \dots, A_s \cdot B_s$, where $A_j = [a_{j,1}, \dots, a_{j,n}], B_j = [b_{j,1}, \dots, b_{j,n}] (j \in \{1, \dots, s\})$ and $a_{j,i}, b_{j,i} \in [2^{w_{in}}]$, stranded encoding encodes these dot product operations in finite field elements $x_i, y_i \in \mathbb{F}_p, p \geq 2^{w_{out}}, i \in \{1, \dots, n\}$ as follows:*

$$x_i = \sum_{j=1}^{s} a_{j,i}\delta^{\phi(j)},\ y_i = \sum_{j=1}^{s} b_{j,i}\delta^{\phi(j)}$$

where $\delta = 2^{2w_{in} + \log n}$ and $\phi(\cdot) : \{1, \dots, s\} \to \mathbb{N}$ can be defined by the following optimization problem:

$$\begin{aligned} \min \quad & \phi(s) \\ s.t. \quad & \Omega_1 = \{\phi(1) + \phi(s),\ ...,\ \phi(s-1) + \phi(s)\} \\ & \Omega_2 = \{2\phi(1),\ 2\phi(2),\ ...,\ 2\phi(s-1), 2\phi(s)\} \\ & \Omega_1 \cap \Omega_2 = \emptyset \end{aligned} \tag{7}$$

In addition, n needs to satisfy the following constraint:

$$(2\phi(s) + 1)(2w_{in} + \log n) \leq w_{out} \tag{8}$$

As a result:

$$\begin{aligned} \sum_{i=1}^{n} x_iy_i = (A_1 \cdot B_1)\delta^{2\phi(1)} + \dots + (A_2 \cdot B_2)\delta^{2\phi(2)} + \\ \dots + (A_s \cdot B_s)\delta^{2\phi(s)} \end{aligned} \tag{9}$$

The core of this definition is to formulate stranded encoding as an optimization problem in Eq. 7. Intuitively, as shown in Eq. 9, Ω_2 is the set of exponents of δs in the terms of $x_i y_i$ that end up to be "useful", Ω_1 represents the set of exponents of δs in the terms of $x_i y_i$ that are going to be discarded. For example, in case of $s = 2$, $\phi(1) = 0, \phi(2) = 1, \Omega_1 = \{1\}, \Omega_2 = \{0, 2\}$. This can be verified in Eq. 5. In case of $s = 3$, $\phi(1) = 0, \phi(2) = 1, \phi(3) = 3, \Omega_1 = \{1, 3, 4, 5\}, \Omega_2 = \{0, 2, 6\}$. This can be verified in Eq. 6.

In addition, the constraint shown in Eq. 8 prevents the stranded encoding scheme from blowing up the finite fields. Since $\delta = 2^{2w_{in} + \log n} > max\{A_i \cdot B_i\}$, each term in Eq. 9 is non-overlapping in the final encoded bits (as shown in Fig. 5). Now, we only need to worry about the last term not exceeding the size of the finite field, which is captured by Eq. 8. We list the $\phi(s)$ for different s and their corresponding n_{max} in Table 4.

Cost based optimization. Now, we can analyze the benefits brought by stranded encoding in terms of the number of constraints. Since the encoding part is "free": the addition would not cost extra constraints. The major cost is decoding, which requires bit decomposition (generating $O(w_{out})$ constraints). For example, in the SNARK implementation we use, bit decomposition of a finite field element used in BLS12-381 generates 632 constraints. Then, the amortized cost of each element-wise multiplication in dot product is:

$$cost(s, n) = \frac{O(w_{out})}{sn} \tag{10}$$

where s is the batch size and n is size of the vectors to be dot producted. For the cost function listed in Eq. 10, we always choose the best batch size s for the given input. For example, in our setting, $w_{in} = 8, w_{out} = 254$, the fixed cost of bit decomposition is 632. For $n < 632/3$, we don't do stranded encoding since the amortized cost is greater than 1. For $632/3 \leq n \leq 4096$, we chose batch size 4. And for $n > 4096$, we choose batch size 3. We believe this cost function can be used to guide different choices of the integer precision of quantized neural network and different sizes of finite fields that are used by zkSNARK.

Table 4: Largest supported vector size n_{max} in stranded encoding for different batch size s ($\omega_{in} = 8$ and $\omega_{out} = 254$). '-' indicates not supported.

s	2	3	4	5
$\phi(s)$	1	3	4	9
$2\phi(s) + 1$	3	7	9	19
n_{max}	2^{68}	2^{20}	2^{12}	-

6 Evaluation

In this section, we evaluate ZEN over various datasets and popular neural network models.

Implementation. ZEN implementation consists of three major parts: a quantization engine, circuit generators, and a scheme aggregator. The quantization engine takes a pretrained floating-point pyTorch model, applies our zkSNARK friendly quantizations, and produces a quantized neural network. Circuit generators generate individual components of circuit. We implemented circuit generators for FC, Conv, average pooling, and ReLU kernels. Our system also include a SNARK friendly Pedersen commitment circuit generator from the underlying zkSNARK system we used. The stranded encoding is implemented in FC and Conv circuit generators. The scheme aggregator assembles all component circuits together, and produces the final zero-knowledge proof systems according to the specified ZEN$_{class}$ or ZEN$_{rec}$ scheme.

We implemented the ZEN prototype using $\sim 3,000$ lines of Rust code. Our system uses arkworks's implementation [ark] of Groth16 scheme [Gro16] as the underlying zkSNARK, to convert the generated circuits to CRS, prover and verifier. We choose the BLS12-381[BLS01] as the underlying curve for in Groth16.

Datasets. We select three popular datasets (MNIST, CIFAR-10, and ORL) used by many secure machine learning papers [DSC+19, ZYCW20, GDL+16, JVC18, JKLS18]. The characteristics of these datasets are summarized in Table 5. Among these datasets, MNIST and CIFAR-10 are used for the classification task and ORL is used for the recognition task (*e.g.*, face recognition). In particular,

Table 5: Datasets used in our evaluation.

Dataset	Task	#Images	Size	#Class
MNIST [LC10]	Classification	70, 000	$28 \times 28 \times 1$	10
CIFAR-10 [KNH]	Classification	60, 000	$32 \times 32 \times 3$	10
ORL [Cam02]	Recognition	400	$56 \times 46 \times 1$	40

Table 6: Neural Networks used in our evaluation.

Network	Number of Layers				# FLOPs (K)
	Conv	FC	Act	Pool	
ShallowNet	0	2	1	0	102
LeNet-5-small	3	2	4	2	530
LeNet-5-medium	3	2	4	2	7,170
LeNet-Face-small	3	2	4	2	2,880
LeNet-Face-medium	3	2	4	2	32,791
LeNet-Face-large	3	2	4	2	127,466

- MNIST is a large dataset for handwritten digits classification with 60, 000 training images and 10, 000 testing images. Images in MNIST are gray-scale of shape $28 \times 28 \times 1$.
- CIFAR-10 is a classification dataset with 10 classes (e.g., cat and dog). It contains 50, 000 training images and 10, 000 testing images of shape $32 \times 32 \times 3$.
- ORL dataset contains face images from 40 distinct subjects with diverse lighting, facial expression, and facial details. Since ORL dataset does not specify the training and testing dataset split, we randomly select 90% images as the training dataset and use the remaining 10% images as the testing dataset.

All images are stored with `uint8` data type and values are between 0 and 255.

Models. We use ShallowNet, a lightweight neural network model and a series of LeNet variants [LBBH98], as summarized in Table 6: ShallowNet contains two fully connected layers and one ReLU activation layer. LeNet has three convolutional layers, two fully connected layers, and four activation layers. These variants have different kernel sizes to adapt different sizes of inputs. The evaluation on these five variants demonstrates the performance of ZEN under diverse model sizes.

Experiment Configuration. All the evaluations run on a Microsoft Azure M16-8ms instance with 8 core Intel Xeon Platinum 8280M vCPU @ 2.70GHz and 437.5 GiB DRAM. We compile ZEN code using Rust 1.47.0 in release mode.

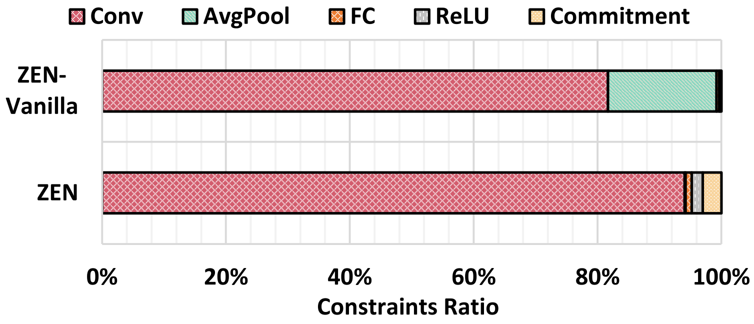

Fig. 6: Breakdown of constraints in LeNet-5-small on CIFAR-10 from both ZEN-vanilla and ZEN.

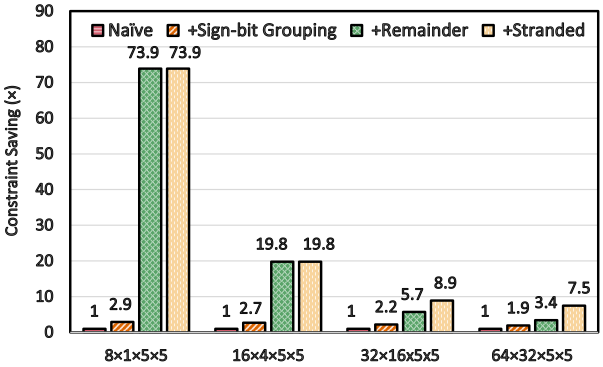

Fig. 7: Benefits on reducing conv-layer number of constraints. Shape is [# of in channels]×[# of out channels]×[kernel width]×[kernel height].

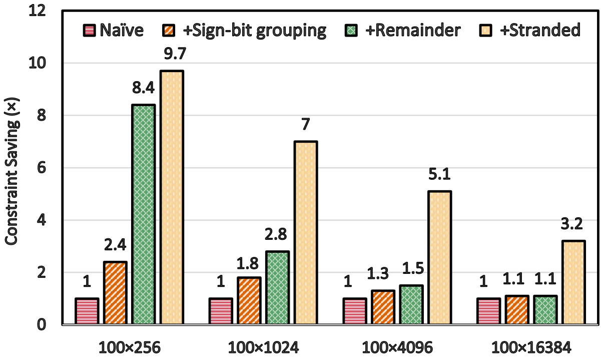

Fig. 8: Benefits on reducing FC-layer number of constraints. Shape is [# of in channels]×[# of out channels].

6.1 Micro-benchmarks.

In this section, we evaluate the performance improvements of our optimizations with a set of micro-benchmarks. As a common practice, we focus on the number of constraints since it dominants the computations in all stages. We defer end-to-end metrics evaluations (*e.g.*, commit time, prover time, verifier time, and proof size) to the next section.

Constraints breakdown by operators We compare the breakdown of number of constraints in generated LeNet-5-small on CIFAR-10 circuits using ZEN-vanilla and fully optimized ZEN in Fig. 6. We breakdown the constraints to those from the commitment scheme and those from 4 different kinds of kernels: convolutional, fully connected, average pooling, ReLU.

Overall, we observe that convolution kernels and fully connected kernels account for most constraints in both the ZEN-vanilla (82.2%) and fully optimized ZEN (95.1%) implementation. Since these two kinds of kernels heavily rely on dot product, this justifies our effort on using stranded encoding to improve batched dot product circuits. Additionally, average pooling kernels accounts for 17.5% constraints in the ZEN-vanilla, while only becomes almost negligible in the fully optimized ZEN. This demonstrates the significance of our remainder-based verification optimization. It is also worth noting that commitment accounts for only 0.2% constraints in the ZEN-vanilla, but this ratio rises to 2.9% constraints in ZEN. Note that the absolute number of constraints from commitment remains the same in both ZEN-vanilla and ZEN. This ratio change comes from optimizations in ZEN that significantly reduces the number of constraints from neural network part.

Evaluating optimizations on fully connected and convolutional kernels Since constraints caused by convolutional (Conv) kernels and fully connected (FC) takes up more than 80% share in total number of constraints in both ZEN-vanilla and fully optimized ZEN, we perform a detailed study on the effectiveness of optimizations on FC and Conv kernels. We chose Conv and FC kernels for a comprehensive study also because they can benefit from all three optimizations.

For each kernel of each size, we first report the number of constraints in ZEN-vanilla. Then, we report the numbers of constraints savings (×) from sign-bit grouping, remainder-based verification, and stranded encoding cumulatively. Note that, the stranded encoding implementation we use incorporates the cost based optimization already: it chooses an optimal batch s according to the data shape; it is not applied when there is no cost saving (this can be viewed as $s = 1$).

Fig. 7 shows the reduction in the number of constraints on Conv kernels. We observe that the number of constraints can be reduced by 7.5× to 73.9× with our optimizations.

Looking at individual kernels, we find that sign-bit grouping and remainder-based verification significantly bring benefits on diverse Conv kernels, especially on small Conv kernels of shape $8\times1\times5\times5$ and $16\times4\times5\times5$. We also notice that the stranded encoding optimization brings benefits on large kernels; and is not effective

Table 7: Overall performance of ZEN.

Dataset	**Model**	**Constraints (K)**	**Setup (s)**	**Comm. (ms)**	**Prove (s)**	**Verify (ms)**	**CRS Size (MB)**
MNIST	**ShallowNet**	111	9.07	18.29	7.01	5.51	35.81
CIFAR-10	**LeNet-5-sm.**	938	77.43	30.39	39.37	5.45	413.51
CIFAR-10	**LeNet-5-med.**	5,540	569.83	43.30	383.01	5.49	2,620.32
ORL	**LeNet-Face-sm.**	2,737	274.11	140.17	174.21	5.05	1,248.31
ORL	**LeNet-Face-med.**	17,506	2392.22	153.38	1880.21	5.07	8,217.77
ORL	**LeNet-Face-lg.**	55,365	13007.78	123.31	11334.31	5.17	25,788.50

for small kernels. As we discussed in the cost-based optimization in §5, stranded encoding may introduce overhead from encoding and decoding procedure. For small kernels, cost-based optimization will select the batch size $s = 1$ to avoid such overhead.

Fig. 8 shows similar savings on the number of constraints for FC kernels as the case on Conv kernels. Comparing with Fig. 7, we observe that stranded encoding brings more significant savings on FC kernels, especially on the ones with large kernel sizes. The insight is that the amortized cost of stranded encoding decreases proportionally as the kernel size increases. To this end, we can expect higher benefit from stranded encoding on kernels with larger sizes.

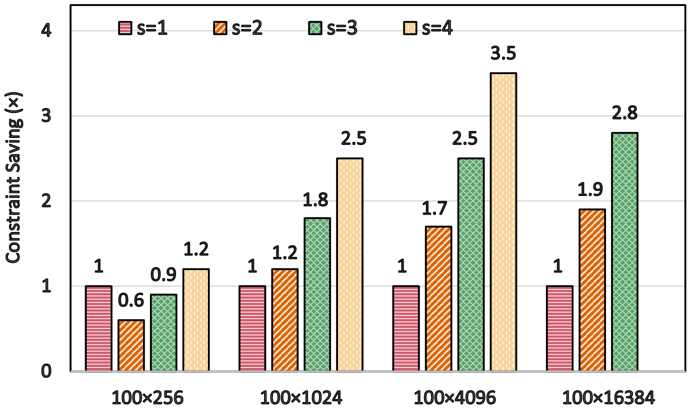

Fig. 9: Benefit from different batching size s for FC-layer. Shape is [# of in channels] × [# of out channels].

Benefits from different batching size s We further evaluate the benefits of stranded encoding under different batch size s in Fig. 9. We skip $s = 4$ for the FC kernel of shape 100×16384 since our BatchEngine with $s = 4$ requires a vector of length less than $2^{12} = 4096$, as described in Table 4.

Table 8: Overall saving on the number of constraints. The unit of number of constraints is thousand (K).

Dataset	**Model**	**Naive (K)**	**Optimized (K)**	**Saving (×)**
MNIST	**ShallowNet**	408	111	3.68
CIFAR-10	**LeNet-5-small**	16,958	938	18.08
CIFAR-10	**LeNet-5-medium**	85,210	5,540	15.38
ORL	**LeNet-Face-small**	57,444	2,737	20.99
ORL	**LeNet-Face-medium**	274,383	17,506	15.67
ORL	**LeNet-Face-large**	610,428	55,365	11.03

We observe that the constraint saving increases as the FC size grows. In particular, when the kernel shape is 100×16384, $s = 3$ can lead to 2.8× constraint saving, which almost reaches the theoretical upper bound

Table 9: Accuracy comparison between floating-point (FP) models and zkSNARK friendly quantized models.

Dataset	**Model**	**FP Acc. (%)**	**Quant. Acc. (%)**	**ΔAcc. (%)**
MNIST	**ShallowNet**	95.13	94.91	−0.22
CIFAR-10	**LeNet-5-small**	55.76	55.35	−0.41
CIFAR-10	**LeNet-5-medium**	64.23	63.68	−0.55
ORL	**LeNet-Face-small**	84.3	84.0	−0.3
ORL	**LeNet-Face-medium**	88.6	88.2	−0.4
ORL	**LeNet-Face-large**	91.6	92.1	0.5

on constraint saving. The reason is that the overhead incurred from the encoding and decoding procedure remains constant in ZEN, regardless the FC size. As the FC size increases, this overhead accounts for less ratio of constraints and the savings from stranded encoding increases. In the meanwhile, stranded encoding brings little benefits on small FC layers (*e.g.*, 100×256) due to the encoding and decoding overhead. This observation motivates the utilization of $s = 1$ for these small FC layers. Comparing across different batch size s, we notice that a larger batch size usually leads to higher saving. This result shows that we should choose a larger batch size s when the vector satisfies the corresponding length requirement.

Case study: end-to-end benefits of optimizations on LeNet-5-Medium We show the end-to-end benefits of different levels of optimizations on LeNet-5-Medium inference on CIFAR-10 dataset, as already shown with Table 2 in introduction. Starting from ZEN-vanilla, we add individual optimizations and show the total number of constraints from all NN layers. In ZEN-vanilla, LeNet-5-Medium has almost 85 million constraints. This large number of constraints comes from the intensive computation and bit decomposition in NNs. When all three optimizations are applied, ZEN significantly reduces the total number of constraints by 15.45×. This is similar to the constraint saving on popular NN kernels in Fig. 8 and Fig. 7. These results show that our optimization techniques can significantly mitigate the intensive number of constraints on NN workloads and enable more efficient deployment of ZEN.

Performance implication of number of linear combinations Apart from number of constraints, we also observed that the setup time and prover time is related to the number of linear combinations of public input and witnesses. However, the cost of linear combinations is system specific. We discuss the performance implications of number of linear combinations on ZEN (using ArkWorks SNARK [ark] as backend) in §A.

6.2 End-to-end performance.

In this section, we evaluate the end-to-end performance of a large set of NNs on various datasets. We provide a comprehensive evaluation of ZEN, including the number of constraints, setup time, commit time, prover time, verifier time, CRS Size, and accuracy.

Overall performance Table 10 shows the overall performance of ZEN on 6 neural networks with diverse numbers of constraints, ranging from 111 thousands to 55, 365 thousands. We observe that a model with a higher number of constraints comes with higher prover and setup time consumption and a larger CRS size. For a small model ShallowNet on MNIST with 111 K constraints, we have a short time of 9 seconds and 7 seconds for setup and proving, respectively; the CRS consists of 35.81 MB of data. The overall cost increases linearly with the number of constraints. For large models such as LeNet-Face-large on ORL with over 55, 365 thousands constraints, we require nearly 3.5 hours for setup, and over 3 hours for proving. We also require over 25, 788.5 MiB of data for CRS.

Overall saving on the number of constraints Table 8 shows the overall saving on the number of constraints with our optimizations. Overall, we can significantly reduce the number of constraints by 14.14× on average. This shows a similar saving on the number of constraints as our case study in the micro-benchmarks. On large models such as LeNet-Face-Large, we achieve a saving of 11.03×. On small models of LeNet-5-small and LeNet-Face-small, we achieve more than 18.08× saving. This is mainly due to the fact that kernels in these small models are dot products of vectors that can be drastically improved with our sign-bit grouping and remainder-based verification. This result is similar to our microbenchmark in Fig. 7 and Fig. 8.

Accuracy drop from quantization Table 9 compares the accuracy between our quantized model (which is similar to [JKC+18]) and the original floating-point model. Given an float32 model, we generate the quantized uint8 model with our zkSNARK friendly quantization. We aim to minimize the accuracy loss from quantization while supporting zkSNARK systems with full quantization. The accuracy of the original floating-point model is out-of-scope for ZEN. Overall, the average accuracy loss of our quantization is 0.23% compared with the floating-point model, which is similar to [JKC+18]. This result shows that ZEN can effectively guarantee the integrity of computation and provide succinct proof on diverse NNs without incurring noticeable accuracy loss. Surprisingly, we observe a 0.5% accuracy improvement for LeNet-Face-large on the ORL dataset. This accuracy improvement comes from the regularization effect of quantization on neural networks, which mitigates over-fitting on large models.

References

ACG+16. Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, pages 308–318. ACM, 2016.

AHIV17. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *CCS*, pages 2087–2104. ACM, 2017.

ark. arkworks. arks-snark. https://github.com/arkworks-rs/snark.

BB04. Dan Boneh and Xavier Boyen. Secure identity based encryption without random oracles. In *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 443–459. Springer, 2004.

BBB+18. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society, 2018.

BBHR18. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.

BCG+20. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: enabling decentralized private computation. In *IEEE Symposium on Security and Privacy*, pages 947–964. IEEE, 2020.

BCI+13. Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, volume 7785 of *Lecture Notes in Computer Science*, pages 315–333. Springer, 2013.

BCR+19. Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 103–128. Springer, 2019.

BCTV14. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796. USENIX Association, 2014.

BDK+18. Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. Hycc: Compilation of hybrid protocols for practical secure computation. In *CCS*, pages 847–861. ACM, 2018.

BLS01. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, volume 2248, pages 514–532, 2001.

BLS02. Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *Security in Communication Networks, Third International Conference, SCN 2002, Amalfi, Italy, September 11-13, 2002. Revised Papers*, volume 2576 of *Lecture Notes in Computer Science*, pages 257–267. Springer, 2002.

BNS19. Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *NeurIPS*, pages 7948–7956, 2019.

Cam02. AT&T Laboratories Cambridge. The database of faces. 2002.

CJM20. Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In *Crypto*, pages 189–218, 2020.

CKM17. Hervé Chabanne, Julien Keuffer, and Refik Molva. Embedded proofs for verifiable neural networks. *IACR Cryptol. ePrint Arch.*, 2017:1038, 2017.

DMM+18. Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist Fomenko, and Vadim Pirogov. Mixed precision training of convolutional neural networks using integer operations. In *ICLR*, 2018.

DS20. Forrest Davis and Marten Van Schijndel. Recurrent neural network language models always learn english-like relative clause attachment. In *ACL*, pages 1979–1990. Association for Computational Linguistics, 2020.

DSC$^+$19. Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *PLDI*, pages 142–156. ACM, 2019.

DSZ15. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.

Gau. Matthew Gault. DHS Admits Facial Recognition Photos Were Hacked, Released on Dark Web. https://www.vice.com/en/article/m7jzbb/dhs-admits-facial-recognition-photos-were-hacked-released-on-dark-web.

GDL$^+$16. Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, volume 48, pages 201–210, 2016.

Gen04. Rosario Gennaro. Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks. In *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2004.

GGG17. Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *NIPS*, pages 4672–4681, 2017.

GGSS19. Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. Compiling kb-sized machine learning models to tiny iot devices. In *PLDI*, pages 79–95. ACM, 2019.

Gro10. Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

GRSY20. Shafi Goldwasser, Guy N. Rothblum, Jonathan Shafer, and Amir Yehudayoff. Interactive proofs for verifying machine learning. *Electron. Colloquium Comput. Complex.*, 27:58, 2020.

GW11. Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108. ACM, 2011.

HSC$^+$20. Yangsibo Huang, Zhao Song, Danqi Chen, Kai Li, and Sanjeev Arora. Texthide: Tackling data privacy for language understanding tasks. In Trevor Cohn, Yulan He, and Yang Liu, editors, *EMNLP*, pages 1368–1382, 2020.

HSLA20. Yangsibo Huang, Zhao Song, Kai Li, and Sanjeev Arora. Instahide: Instance-hiding schemes for private distributed learning. In *ICML*, 2020.

HZRS16. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society, 2016.

JKC$^+$18. Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, pages 2704–2713, 2018.

JKLS18. Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *CCS*, pages 1209–1222. ACM, 2018.

JL17. Robin Jia and Percy Liang. Adversarial examples for evaluating reading comprehension systems. In *EMNLP*, pages 2021–2031. Association for Computational Linguistics, 2017.

JVC18. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, pages 1651–1669. USENIX Association, 2018.

KNH. Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

LBBH98. Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

LC10. Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

LDC$^+$20. Liu Liu, Lei Deng, Zhaodong Chen, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Xing Hu, Yufei Ding, and Yuan Xie. Boosting deep neural network efficiency with dual-module inference. In *ICML*, 2020.

LJLA17. Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

LKKO20. Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network. *IACR Cryptol. ePrint Arch.*, 2020:584, 2020.

LTA16. Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *ICML*, page 2849–2858. JMLR.org, 2016.

MNA+18. Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *ICLR*, 2018.

MZ17. Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

MZCS18. Xu Ma, Fangguo Zhang, Xiaofeng Chen, and Jian Shen. Privacy preserving multi-party computation delegation for deep learning in cloud computing. In *Information Science*, 2018.

O'F. Kate O'Flaherty. China facial recognition database leak sparks fears over mass data collection. shorturl.at/OUW59.

PH12. David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

PHGR13. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society, 2013.

RRK18. Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

SCC+19. Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In *NeurIPS*, pages 4901–4910, 2019.

She. Xinmei Shen. Facial recognition data leaks are rampant in china as covid-19 pushes wider use of the technology. shorturl.at/kFHY1.

SS15. Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1310–1321, New York, NY, USA, 2015. Association for Computing Machinery.

SYS+20. Yichun Shi, Xiang Yu, Kihyuk Sohn, Manmohan Chandraker, and Anil K. Jain. Towards universal representation learning for deep face recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

Wik. Wikipedia. Know your customer. https://en.wikipedia.org/wiki/Know_your_customer.

WJI+20. Xin Eric Wang, Vihan Jain, Eugene Ie, William Yang Wang, Zornitsa Kozareva, and Sujith Ravi. Environment-agnostic multitask learning for natural language grounded navigation. In *ECCV*, 2020.

WTS+18. Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society, 2018.

WWZG20. Qiangchang Wang, Tianyi Wu, He Zheng, and Guodong Guo. Hierarchical pyramid diverse attention networks for face recognition. In *CVPR*, June 2020.

XZZ+19. Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. *IACR Cryptol. ePrint Arch.*, 2019:317, 2019.

ZFZS20. Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *CCS*, pages 2039–2053, 2020.

ZYCW20. Yuqing Zhu, Xiang Yu, Manmohan Chandraker, and Yu-Xiang Wang. Private-knn: Practical differential privacy for computer vision. In *CVPR*, June 2020.

Table 10: Overall performance of ZEN.

Dataset	**Model**	**Linear Combinations (K)**	**Constraints (K)**	**Setup (s)**	**Comm. (ms)**	**Prove (s)**	**Verify (ms)**	**CRS Size (MB)**
MNIST	**ShallowNet**	**561**	111	9.07	18.29	7.01	5.51	35.81
CIFAR-10	**LeNet-5-sm.**	**6,885**	938	77.43	30.39	39.37	5.45	413.51
CIFAR-10	**LeNet-5-med.**	**49,489**	5,540	569.83	43.30	383.01	5.49	2,620.32
ORL	**LeNet-Face-sm.**	**21,681**	2,737	274.11	140.17	174.21	5.05	1,248.31
ORL	**LeNet-Face-med.**	**169,147**	17,506	2392.22	153.38	1880.21	5.07	8,217.77
ORL	**LeNet-Face-lg.**	**601,396**	55,365	13007.78	123.31	11334.31	5.17	25,788.50

Table 11: Overall saving on the number of linear combinations. The unit of number of linear combinations is thousand (K).

Dataset	**Model**	**Naive (K)**	**Optimized (K)**	**Saving (×)**
MNIST	**ShallowNet**	1, 532	561	2.73
CIFAR-10	**LeNet-5-small**	51, 986	6, 885	7.55
CIFAR-10	**LeNet-5-medium**	264, 535	49, 489	5.34
ORL	**LeNet-Face-small**	175, 783	21, 681	8.11
ORL	**LeNet-Face-medium**	857, 551	169, 147	5.07
ORL	**LeNet-Face-large**	1, 961, 749	601, 396	3.26

A Discussion: Performance Implication of Number of Linear Combinations.

We include the impact of our optimizations on the number of linear combinations in Fig. 10 and Fig. 11. Overall, we observe that our optimizations can save the number of linear combinations from 1.4× to 24.4×. We see that stranded encoding consistently saves the number of linear combinations across various kernel sizes. Meanwhile, sign-bit grouping and remainder-based verification may slightly increase the number of linear combinations on certain NN kernels (*e.g.*, 100×4096 and 100×16384 in Fig. 11).

We show the overall performance of ZEN in Table 10 and Table 11. We observe that the latency of ZEN is related to both the number of linear combinations and the number of constraints, which is a system-specific limitation on the ArkWorks SNARK backend. Table 11 shows that the number of linear combinations varies significantly from 1, 532K to 1, 961, 749K. With our optimizations, we can save the number of linear combinations from 2.73× to 8.11× (5.34× on average).

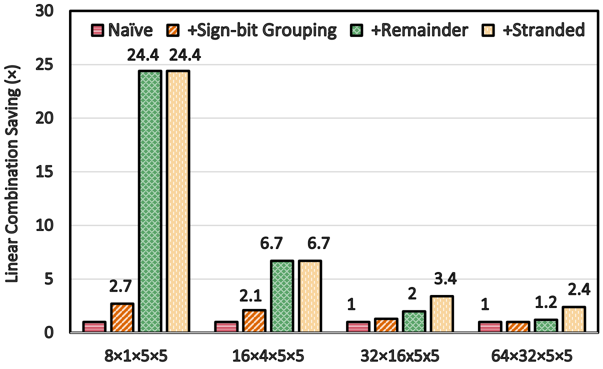

Fig. 10: Benefits on reducing conv-layer number of linear combinations. Shape is [# of in channels]×[# of out channels]×[kernel width]×[kernel height].

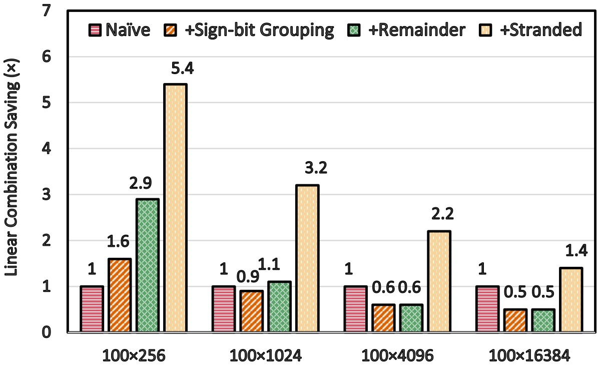

Fig. 11: Benefits on reducing FC-layer number of linear combinations. Shape is [# of in channels]×[# of out channels].