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\(_{\text{class}}\) and ZEN\(_{\text{rec}}\). Both ZEN\(_{\text{class}}\) and ZEN\(_{\text{rec}}\) 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\(_{\text{class}}\) and ZEN\(_{\text{rec}}\).
- **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\(_{\text{class}}\) and ZEN\(_{\text{rec}}\) with zkSNARK friendly quantization and stranded encoded matrix operations. Our evaluation shows that ZEN brings \(3.68 \sim 20.99 \times (14.14 \times \text{on average})\) savings in the number of constraints compared with a vanilla implementation of the neural network on zkSNARK.

![Fig. 1: Overview of ZEN tool chain.](image)

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\(_{\text{class}}\), a verifiable classification scheme; and
- ZEN\(_{\text{rec}}\), 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\(_{\text{class}}\)), 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\(_{\text{rec}}\)), 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\(_{\text{rec}}\) 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,
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 proof system for generic neural networks with the 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 elliptic curves (e.g. BLS12-381 [BLS02]) with an underlying finite field of order $2^{254}$; 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:

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv</td>
<td>69,692,928</td>
<td>28,630,272</td>
<td>8,082,688</td>
<td>5,195,008</td>
</tr>
<tr>
<td>FC</td>
<td>394,000</td>
<td>219,706</td>
<td>132,490</td>
<td>54,906</td>
</tr>
<tr>
<td>AvgPool</td>
<td>14,925,312</td>
<td>4,982,976</td>
<td>7,872</td>
<td>7,872</td>
</tr>
<tr>
<td>ReLU</td>
<td>114,227</td>
<td>97,920</td>
<td>97,920</td>
<td>97,920</td>
</tr>
</tbody>
</table>

A caveat here is to pack as many matrix coefficients into a field element, while still allowing a proper $\phi(\cdot)$ to extract $A_1 \cdot B_1, \ldots, A_s \cdot B_s$ 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 $3.67 \sim 21.01 \times$ (14.27 \times 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. 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 \((\text{Gen, Prove, Verify, } |\pi|)\) 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 | 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. | \(O(C \log^2 C)\) | \(O(C)\) | \(O(\log^2 C)\) | 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 \((a, (1, y_1, \ldots, y_m)) : (b, (1, y_1, \ldots, y_m))\) for \(a, b \in \mathbb{F}_p^{m+1}\).

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 \(C : \mathbb{F}_p^n \times \mathbb{F}_p^h \rightarrow \mathbb{F}_p^l\) can be defined by the relation \(\mathcal{R}_C = \{(x, w) \in \mathbb{F}_p^n \times \mathbb{F}_p^h : C(x, w) = 0\}\) and the language \(\mathcal{L}_C = \{x \in \mathbb{F}_p^n : \exists a \in \mathbb{F}_p^{h} \text{ s.t. } C(x, w) = 0\}\).

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

- \(\text{Gen}(1^\lambda, C) \rightarrow (pk, vk)\). Using a security parameter \(\lambda\) and an \(\mathbb{F}_p\)-arithmetic circuit \(C\) as inputs, the key generator \(\text{Gen}\) randomly samples a proving key \(pk\) and a verification key \(vk\). These keys are considered as public parameters \(pp := (pk, vk)\), and can be used any number of times to prove/verify the membership in \(\mathcal{L}_C\).

- \(\text{Prove}(pk, x, w) \rightarrow \pi\). Taking a proving key \(pk\), and any \((x, w) \in \mathcal{R}_C\) as inputs, the \(\text{Prove}\) algorithm generates a non-interactive proof \(\pi\) for the statement \(x \in \mathcal{L}_C\).

- \(\text{Verify}(vk, x, \pi) \rightarrow \{0, 1\}\). Taking the verification key \(vk\), public input \(x\), and proof \(\pi\), the \(\text{Verify}\) 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 \(\lambda\), 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 \(1 - \text{negl}(\lambda)\) in the following experiment: \((vk, pk) \leftarrow (1^\lambda, C); \pi \leftarrow \text{Prove}(pk, x, w); 1 \leftarrow \text{Verify}(vk, x, \pi)\).

**Succinctness.** An honestly-generated proof \(\pi\) has \(O_\lambda(1)\) bits and \(\text{Verify}(vk, x, \pi)\) runs in time \(O_\lambda(|x|)\).

**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 \(\text{poly}(\lambda)\)-size adversary \(A\), there is a \(\text{poly}(\lambda)\)-size extractor \(E\) such that \(\text{Verify}(vk, x, \pi) = 1\) and \((x, w) \notin \mathcal{R}_C\) with probability \(\text{negl}(\lambda)\) in the following experiment: \((pk, vk) \leftarrow \text{KeyGen}(1^\lambda, C); (x, \pi) \leftarrow A(pk, vk); w \leftarrow E(pk, vk)\).

**Zero Knowledge.** An honestly generated proof is zero knowledge. Specifically, there is a \(\text{poly}(\lambda)\)-size simulator \(\text{Sim}\) such that for all stateful \(\text{poly}(\lambda)\)-size distinguishers \(D\), the probability of \(D(\pi) = 1\) 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 \(\lambda\).
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 \( X_i \)

\[
X_i = f_i(X_{i-1}|W_i), \quad i \in \{1, 2, \ldots, n\}
\]

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 \( X_n \) from the \( n \)th layer and selects the prediction \( \hat{y} = \arg\max_i X_n \) 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, \( \mathcal{N} : V_1 \rightarrow V_2 \), 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, \( \mathcal{Q} : \mathbb{F}_p^{d_1} \rightarrow \mathbb{F}_p^{d_2} \), 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, \( \text{COMM} : \{0,1\}^{O(\lambda)} \times \{0,1\}^* \rightarrow \{0,1\}^{O(\lambda)} \). 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\textsubscript{class}: zero-knowledge verifiable neural network based classification

Motivated by the application in the introduction, our zero-knowledge verifiable neural network classification scheme (ZEN\textsubscript{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 Q be a quantized neural network that represents the mapping \( Q : \mathbb{F}_p^d \rightarrow [M] \), where \([M]\) is the set of all possible classifications. A zero-knowledge verifiable neural network based classification scheme (ZEN\textsubscript{class}) can be defined as the following algorithms:

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

\( \pi \) is a zk-SNARK proof that the prover produces for the following NP statement:

Protocol 1 (NP statement for ZEN\textsubscript{class}) \textit{Given a commitment of input cm, “I” know a secret input a and a secret opening r such that:}

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

We define ZEN\textsubscript{class} formally as follows:

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

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

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

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

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

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

\[
\Pr \left[ \begin{array}{l}
D(cm, y_a, \pi) \leftarrow ZEN_{\text{class}}, \text{Prove}(pk,a,r), \\
(m, y_a, \pi) \leftarrow ZEN_{\text{class}}, \text{Prove}(pk,a,r)
\end{array} \right] = 1
\]
• The probability that \( D(cm, y_a, \pi) = 1 \) on a simulated proof:

\[
\Pr[D(cm, y_a, \pi) = 1] = \Pr[(pk, vk) \leftarrow \text{Sim}(1^\lambda, Q), \ (a, r) \leftarrow D(pk, vk), \ (cm, y_a, \pi) \leftarrow ZEN_{\text{class}}, \text{Sim}(pk, a, r)]
\]

### 3.2 ZEN\(_{\text{rec}}\): 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 \( L \).

If the loss is smaller than certain specified threshold (\( L(y_a, y_g) \leq \tau \)), 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\(_{\text{class}}\)) scheme assumes that the prover and the verifier agree on a neural network model, a commitment to the ground truth embedding \( cm_g \leftarrow \text{COMM}(s, y_g) \) where \( y_g \) is the ground truth embedding and \( s \) is some random opening, the distance metric \( L \), and the threshold \( \tau \). 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 \( L \), evaluated over both the \( y_a \) and the agreed and committed ground true \( y_g \), is less than or equal to \( \tau \). Again, the prover cannot alter her inputs once committed (binding), while her inputs remain private (hiding) during the whole process.

Let \( Q \) be the quantized neural network, represented via a mapping \( Q : \mathbb{F}_p^d \rightarrow \mathbb{F}_p^d \). Let \( L : \mathbb{F}_p^d \times \mathbb{F}_p^d \rightarrow \mathbb{F}_p \) 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\(_{\text{rec}}\)) can be defined as the following algorithms:

- \((pk, vk) \leftarrow ZEN_{\text{rec}}, \text{Gen}(1^\lambda, Q)\): given a security parameter \( \lambda \) and a quantized neural network model \( Q \) for recognition, randomly generate a proving key \( pk \) and a verification key \( vk \).
- \((cm_a, cm_1, cm_2, \pi) \leftarrow ZEN_{\text{rec}}, \text{Prove}(pk, a, y_g, r_1, r_2, s)\): 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 \( Q \) on \( a \) to obtain the embedding result \( y_a \), commits the input \( a \) to \( cm_1 \leftarrow \text{COMM}(r_1, a) \) and the output \( y_a \) to \( cm_2 \leftarrow \text{COMM}(r_2, y_a) \), respectively. The prover also commits the ground truth embedding \( y_a \) to \( cm_g \leftarrow \text{COMM}(y_a, s) \) using \( s \). The prover generates a proof \( \pi \) for verifying these computations as well as \( L(y_a, y_g) \leq \tau \).
- \([0, 1] \leftarrow ZEN_{\text{rec}}, \text{Verify}(vk, cm_g, cm_1, cm_2, \pi)\): 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 \( \pi \).

Informally, \( \pi \) is a zero-knowledge proof of the following statement:

**Protocol 2 (NP statement for ZEN\(_{\text{rec}}\))** 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:
  
  \[
  cm_1 = \text{COMM}(r_1, a),
  cm_2 = \text{COMM}(r_2, y_a),
  cm_g = \text{COMM}(s, y_g)
  \]

- The neural embedding is valid: \( y_a = Q(a) \).
- The distance between \( a \)'s embedding and the ground truth embedding is not greater than the threshold \( \tau \):
  
  \[
  L(y_a, y_g) \leq \tau
  \]

we define ZEN\(_{\text{rec}}\) 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 $Q$ and an input $a \in \mathbb{R}^d$, $(pk, vk) \leftarrow$ ZEN$_{rec}$.Gen($1^\lambda$, $Q$), $(cm_g, cm_1, cm_2, \pi) \leftarrow$ ZEN$_{rec}$.Prove($pk, a, y_g, r_1, r_2, s$), it must hold that:
  \[
  \Pr[\text{ZEN$_{rec}$.Verify}(vk, cm_g, cm_1, cm_2, \pi) = 1] = 1
  \]

- **Soundness.** For any PPT adversary $A$, the following probability is negligible in $\lambda$:

\[
\begin{bmatrix}
\Pr[(pk, vk) \leftarrow \text{ZEN$_{rec}$.Gen}(1^\lambda, Q), (a', y_{g'}, y_{g'}, cm'_g, cm'_1, cm'_2, \pi') \leftarrow \text{A}(1^\lambda, pk, vk, r_1, r_2, s')], \\
\text{cm}_g' = \text{COMM}(s', y_{g'}), \\
\text{cm}_1' = \text{COMM}(r_1', a'), \\
\text{cm}_2' = \text{COMM}(r_2', y_{g'}), \\
1 \leftarrow \text{ZEN$_{rec}$.Verify}(vk, cm'_g, cm'_1, cm'_2, \pi'), \\
Q(a') = y_a, \\
L(y_a, y_{g'}) > \tau
\end{bmatrix}
\]

- **Zero Knowledge.** An honestly-generated proof is perfect zero knowledge. For security parameter $\lambda$, $(pk, vk) \leftarrow$ ZEN$_{rec}$.Gen($1^\lambda$, $Q$), PPT distinguisher $D$, there exists a PPT simulator $\text{Sim}$ such that the following probabilities are indistinguishable (at most differs by negl$(\lambda)$):
  - The probability that $D(cm_g, cm_1, cm_2, \pi) = 1$ on an honest proof:
  \[
  \Pr \left[ D(cm_g, cm_1, cm_2, \pi) = 1 \middle| \begin{bmatrix}(pk, vk) \leftarrow \text{ZEN$_{rec}$.Gen}(1^\lambda, Q), (a, y_g, r_1, r_2, s) \leftarrow D(pk, vk), (cm_g, cm_1, cm_2, \pi) \leftarrow \text{ZEN$_{rec}$.Prove}(pk, a, y_g, r_1, r_2, s) \end{bmatrix} \right] = 1
  \]
  - The probability that $D(cm_g, cm_1, cm_2, \pi) = 1$ on a simulated proof:
  \[
  \Pr \left[ D(cm_g, cm_1, cm_2, \pi) = 1 \middle| \begin{bmatrix}(pk, vk) \leftarrow \text{Sim}(1^\lambda, Q), (a, y_g, r_1, r_2, s) \leftarrow D(pk, vk), (cm_g, cm_1, cm_2, \pi) \leftarrow \text{Sim}(pk, a, y_g, r_1, r_2, s) \end{bmatrix} \right]
  \]

(a) ReLU
\[
\begin{array}{cccc}
-1 & 3 & 6 & -2 \\
9 & 4 & -6 & -6 \\
-5 & -1 & -1 & 16 \\
8 & 11 & -7 & 16
\end{array}
\]
(b) AvgPool
\[
\begin{array}{cccc}
-1 & 3 & 6 & -2 \\
9 & 4 & -6 & -6 \\
-5 & -1 & -1 & 16 \\
8 & 11 & -7 & 16
\end{array}
\]
(c) Fully Connected
\[
\begin{array}{cccc}
-1 & 3 & 6 & -2 \\
9 & 4 & -6 & -6 \\
-5 & -1 & -1 & 16 \\
8 & 11 & -7 & 16
\end{array}
\]
(d) Convolution
\[
\begin{array}{cccc}
-1 & 3 & 6 & -2 \\
9 & 4 & -6 & -6 \\
-5 & -1 & -1 & 16 \\
8 & 11 & -7 & 16
\end{array}
\]

Fig. 2: Popular neural network kernels

4 zkSNARK Friendly Quantization

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:

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

(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 $(z_Y, z_W, z_X \in \text{uint})$ for each matrix. As a result, we have the quantized representation:\

$$ Y = s_Y (Q_Y - z_Y J_{m,1}) \quad W = s_W (Q_W - z_W J_{m,n}) \quad X = s_X (Q_X - z_X J_{n,1}) $$

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

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

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

$$ M = \lfloor 2^k \frac{s_W s_X}{s_Y} \rfloor $$

$$ Q_Y - z_Y J_{m,1} = M (Q_W - z_W J_{m,n}) (Q_X - z_X J_{n,1}) / 2^k $$

(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:

$$ Q_Y = z_Y J_{m,1} + M (Q_W - z_W J_{m,n}) (Q_X - z_X J_{n,1}) / 2^k $$

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:

$$ G_1 = Q_W Q_X, \quad G_2 = z_X Q_W, \quad G_3 = z_W Q_X, \quad M' = \left\lfloor \frac{z_Y 2^k}{M} \right\rfloor $$

$$ Q_Y = M (G_1 + nz_W z_X J_{m,1} + M' J_{m,1} - G_2 - G_3) / 2^k $$

(3)

$^6$ $J_{k,l}$ 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 na"ively 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:

\[
Q_Y g^k + R = M(G_1 + n z_w z_X J_{m,1} + M' J_{m,1} - G_2 - G_3)
\]

(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

\[
Q_{ReLU} = \max(Q, z J_{c_{in},m,n})
\]

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 \( \gamma \) 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 $s = 2$

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 $A = [a_1, a_2], B = [b_1, b_2], a_i, b_i \in \text{uint8}$, 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 $x y$. 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:

$$
x_1 = a_1 + c_1 \delta \quad x_2 = a_2 + c_2 \delta \\
y_1 = b_1 + d_1 \delta \quad y_2 = b_2 + d_2 \delta
$$

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

$$
x_1 y_1 = a_1 b_1 + (a_1 d_1 + c_1 d_1) \delta + c_1 d_1 \delta^2 \\
x_2 y_2 = a_2 b_2 + (a_2 d_2 + c_2 d_2) \delta + c_2 d_2 \delta^2 \\
x_1 y_1 + x_2 y_2 = (a_1 b_1 + a_2 b_2) + (\ldots) \delta \\
+ (c_1 d_1 + c_2 d_2) \delta^2
$$

We can extract $A \cdot B$ and $C \cdot D$ from $x_1 y_1 + x_2 y_2$ by mod $\delta$ 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:

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

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:

$$
\sum_{i=1}^{n} x_i y_i = \sum_{i=1}^{n} a_i b_i + (\ldots) \delta + \left(\sum_{i=1}^{n} c_i d_i\right) \delta^2
$$

$$
= A \cdot B + (\ldots) \delta + (C \cdot D) \delta^2 \quad (5)
$$

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

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

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

\[
x_i = a_i + c_i \delta + e_i \delta^2 \\
y_i = b_i + d_i \delta + f_i \delta^2
\]

Then, the multiplication becomes:

\[
x_i y_i = a_i b_i + (a_i d_i + b_i e_i) \delta + (c_i d_i + a_i f_i + b_i e_i) \delta^2 \\
+ (c_i f_i + d_i e_i) \delta^3 + e_i f_i \delta^4
\]

It becomes very difficult to extract \( c_i d_i \) from \( x_i y_i \) since \( c_i d_i \) is “mixed” in the coefficient of \( \delta^2 \). To solve this problem, we use the following encoding instead for \( i \in \{1, \ldots, n\} \):

\[
x_i = a_i + c_i \delta + e_i \delta^3 \\
y_i = b_i + d_i \delta + f_i \delta^3
\]

Now it is not hard to see:

\[
x_i y_i = a_i b_i + (\ldots) \delta + (c_i d_i) \delta^2 + \ldots + (e_i f_i) \delta^6
\] (6)

As a result:

\[
\sum_{i=1}^{n} x_i y_i = A \cdot B + (\ldots) \delta + (C \cdot D) \delta^2 + \ldots + (E \cdot F) \delta^6
\]

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 \( A_1 \cdot B_1, \ldots, A_s \cdot B_s \), where \( A_j = [a_{j,1}, \ldots, a_{j,n}], B_j = [b_{j,1}, \ldots, b_{j,n}] \) \((j \in \{1, \ldots, s\})\) and \( a_{j,i}, b_{j,i} \in [2^{w_{in}}] \), stranded encoding encodes these dot product operations in finite field elements \( x_i, y_i \in \mathbb{F}_p, p \geq 2^{w_{in}}, i \in \{1, \ldots, n\} \) 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 \( \delta = 2^{w_{in}+\log n} \) and \( \phi(\cdot) : \{1, \ldots, s\} \to \mathbb{N} \) can be defined by the following optimization problem:

\[
\min \ \phi(s) \\
\text{s.t.} \ \Omega_1 = \{\phi(1) + \phi(s), \ldots, \phi(s-1) + \phi(s)\} \\
\Omega_2 = \{2\phi(1), 2\phi(2), \ldots, 2\phi(s-1), 2\phi(s)\} \\
\Omega_1 \cap \Omega_2 = \emptyset
\] (7)

In addition, \( n \) needs to satisfy the following constraint:

\[
(2\phi(s) + 1)(2^{w_{in}} + \log n) \leq w_{out}
\] (8)

As a result:

\[
\sum_{i=1}^{n} x_i y_i = (A_1 \cdot B_1)\delta^{2\phi(1)} + \ldots + (A_2 \cdot B_2)\delta^{2\phi(2)} + \\
\ldots + (A_s \cdot B_s)\delta^{2\phi(s)}
\] (9)
Table 4: Largest supported vector size $n_{\text{max}}$ in stranded encoding for different batch size $s$ ($\omega_{\text{in}} = 8$ and $\omega_{\text{out}} = 254$). ‘-‘ indicates not supported.

<table>
<thead>
<tr>
<th>$s$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi(s)$</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>$2\phi(s)+1$</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>$n_{\text{max}}$</td>
<td>$2^\phi$</td>
<td>$2^\omega$</td>
<td>$2^w$</td>
<td>$2^s$</td>
</tr>
</tbody>
</table>

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 includes 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 ZENclass or ZENrec 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 Groth16.

Datasets. We select three popular datasets (MNIST, CIFAR-10, and ORL) used by many secure machine learning papers [DSC’19, ZYCW’20, 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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Task</th>
<th>#Images</th>
<th>Size</th>
<th>#Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>Classification</td>
<td>70,000</td>
<td>28 × 28 × 1</td>
<td>10</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>Classification</td>
<td>60,000</td>
<td>32 × 32 × 3</td>
<td>10</td>
</tr>
<tr>
<td>ORL</td>
<td>Recognition</td>
<td>400</td>
<td>56 × 46 × 1</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 6: Neural Networks used in our evaluation.

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

- 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 × 28 × 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 × 32 × 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.](image-url)
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 dominates 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 \( 8 \times 1 \times 5 \times 5 \) and \( 16 \times 4 \times 5 \times 5 \). We also notice that the stranded encoding optimization brings benefits on large kernels; and is not effective...
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] $\times$ [# 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 $100 \times 16384$ 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).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>Naive</th>
<th>Optimized</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>ShallowNet</td>
<td>408</td>
<td>111</td>
<td>3.68</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>LeNet-5-small</td>
<td>16,958</td>
<td>938</td>
<td>18.08</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>LeNet-5-medium</td>
<td>85,210</td>
<td>5,540</td>
<td>15.38</td>
</tr>
<tr>
<td>ORL</td>
<td>LeNet-Face-small</td>
<td>57,444</td>
<td>2,737</td>
<td>20.99</td>
</tr>
<tr>
<td>ORL</td>
<td>LeNet-Face-medium</td>
<td>274,383</td>
<td>17,506</td>
<td>15.67</td>
</tr>
<tr>
<td>ORL</td>
<td>LeNet-Face-large</td>
<td>610,428</td>
<td>55,365</td>
<td>11.03</td>
</tr>
</tbody>
</table>

We observe that the constraint saving increases as the FC size grows. In particular, when the kernel shape is $100 \times 16384$, $s = 3$ can lead to $2.8 \times$ constraint saving, which almost reaches the theoretical upper bound.
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., $100 \times 256$) 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


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


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


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


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


Table 10: Overall performance of ZEN.

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

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>Naive (K)</th>
<th>Optimized Saving (×)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MNIST</td>
<td>ShallowNet</td>
<td>1,532</td>
<td>561</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>LeNet-5-small</td>
<td>51,986</td>
<td>6,885</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>LeNet-5-medium</td>
<td>264,535</td>
<td>49,489</td>
</tr>
<tr>
<td>ORL</td>
<td>LeNet-Face-small</td>
<td>175,783</td>
<td>21,681</td>
</tr>
<tr>
<td>ORL</td>
<td>LeNet-Face-medium</td>
<td>857,551</td>
<td>169,147</td>
</tr>
<tr>
<td>ORL</td>
<td>LeNet-Face-large</td>
<td>1,961,749</td>
<td>601,396</td>
</tr>
</tbody>
</table>

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].](image)

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