ZEN: An Optimizing Compiler for Verifiable, Zero-Knowledge Neural Network Inferences

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. We present ZEN, the first optimizing compiler that generates efficient verifiable, zero-knowledge neural network inference schemes. ZEN generates two schemes: ZEN_{acc} and ZEN_{infer}. ZEN_{acc} proves the accuracy of a committed neural network model; ZEN_{infer} proves a specific inference result. Used in combination, these verifiable computation schemes ensure both the privacy of the sensitive user data as well as the confidentiality of the neural network models. However, directly using these schemes on zkSNARKs requires prohibitive computational cost. As an optimizing compiler, ZEN introduces two kinds of optimizations to address this issue: first, ZEN incorporates a new neural network quantization algorithm that incorporate two R1CS friendly optimizations which makes the model to be express in zkSNARKs with less constraints and minimal accuracy loss; second, ZEN introduces a SIMD style optimization, namely stranded encoding, that can encoding multiple 8bit integers in large finite field elements without overwhelming extraction cost. Combining these optimizations, ZEN produces verifiable neural network inference schemes with $\mathbf{5.43} \sim \mathbf{22.19}\times$ (15.35× on average) less R1CS constraints.

1 Introduction

From health care AI to machine translation, our civilization relies more and more on neural networks. With the increasing adoptions of neural networks, privacy leakage is ever-growing [Gau, O'F, She]. In particular, we need to protect two kinds of privacy: the privacy of the end user's sensitive data; and the intellectual properties of the neural network that many companies spent millions of dollars in training [Wig, PS].

In recent years, zero-knowledge proof systems, commonly known as zkSNARKs (zero-knowledge Succinct ARgument of Knowledge) [PHGR13, BCTV14, AHIV17, BBB+18, WTS+18, BBHR18, BCR+19, XZZ+19, BCG+20], become increasingly efficient. One natural question to ask is:

> *Can we build an optimizing compiler that leverages these powerful zkSNARKs to construct efficient privacy-preserving, verifiable neural network schemes?*

This question is important since a zkSNARK based privacy-preserving and verifiable neural network scheme has many desired and unique features, compared with other popular solutions in this domain, such as secure multi-party computation (MPC) and homomorphic encryption (HE) [GDL+16, JVC18, DSC+19, JKLS18, MZ17, LJLA17, RRK18, SS15]. To name a few, (1) the computation result will be publicly verifiable; (2) the whole process could be non-interactive.

Many potential applications require these two properties. For example, a patient may want to selectively disclose the diagnosis result by an AI doctor without leaking sensitive personal information. Blockchain oracles may want to put public verifiable results of neural network based inference such as facial recognition, natural language processing on the blockchain to interact with smart contracts, yet not leak the input information.

To support these promising applications, we first propose verifiable neural network inference schemes that protect both the privacy of input and the confidentiality of neural network model, namely ZEN_{acc} and ZEN_{infer}. They can work in combination: during the setup phase (ZEN_{acc}), the neural network inference service provider first commit the model, then take a random challenge from the user to prove the accuracy of the committed model; then, the service provider can provide verifiable inference using the committed model without leaking the input data (ZEN_{infer}).

However, naively implementing these schemes on zkSNARKs incurs prohibitive cost. The neural network that is useful in the real world is usually defined on floating-point numbers, which is not the first-class citizens in zkSNARKs. Despite previous attempts to improve the efficiency of verifiable floating-point computation [SVP+12], there is still a huge performance gap between the floating-point computation compared with "native" arithmetic computations in finite field. Additionally, inference tasks using modern neural network models require a significant amount of computation, which makes the constraints size very large and thus prohibitive performance.

To address this challenge, we develop the first end to end optimizer that compiles a floating-point pyTorch model to R1CS constraints, a common intermediate representation (IR) that is supported by a variety of zkSNARK back-ends. First, we observed that neural network quantization has been extensively studied [JKC+18, DMM+18, MNA+18, SCC+19, LLQ+19, WPQ+18, SJG+20] and widely deployed [BSB, Alf, PyT, Ten]. As a result, instead of adapting zkSNARKs to floating-point computation, we adapt the state of art neural network quantization technique to convert floating-point neural network models to quantized models which only contains low precision integers. To optimize the number of the constraints in R1CS of the quantized models, we propose two "lossless" R1CS friendly optimizations: sign-bit grouping and remainder based verification, which reduces the constraint sizes without any accuracy loss compared with models generated by the state-of-the-art quantization technique.

Since quantized neural network models work well on low precision integers (8bit or 16bit) and the underlying zkSNARKs usually works on large finite field element (e.g. 254bit), an intuitive idea is to encode many integers in a single finite field element, similar to SIMD (Single Instruction Multiple Data). However, simply stacking low precision integers in finite field elements would not work since extracting these integers requires expensive bit decompositions, which out-weights the savings. To solve this problem, we a novel encoding technique, namely *stranded encoding*, that could bring a SIMD style optimization to the batched dot product. We also develop an analytical cost model for stranded encoding so that the encoding scheme can always choose the best parameter based on the size of neural network kernels. Combining R1CS friendly optimizations and stranded encoding, ZEN brings up to 22.19× savings in the number of constraints compared with a vanilla implementation of the neural network models in zkSNARK.

1.1 Our Contributions

In this paper, we present ZEN: an optimizing compiler for verifiable neural networks with zero-knowledge. To our best knowledge, ZEN is the first work of its kind. In short, ZEN makes the following contributions:

- ZEN *schemes:* We propose two privacy-preserving, verifiable inference schemes for neural networks, namely ZEN_{acc} and ZEN_{infer}. These schemes can be used in many promising applications where a publicly verifiable inference result is essential. Additionally, they provide cryptographic privacy guarantees (zero-knowledge) for both the **sensitive input** and the **neural network models**.
- *R1CS friendly quantization:* The first compilation challenge that ZEN faces is converting a floating-point neural network to a *fully quantized* model, so that it could be expressed in Rank-1 Constraint Systems (R1CS), a common intermediate representation used by many proving systems. We proposed a new quantization algorithm, which incorporates two R1CS friendly optimizations, namely sign-bit grouping and remainder-based verification. As a result, compared with the state-of-the-art quantization schemes, ZEN brings up to 73.9× **savings** in R1CS constraints for convolution kernel and up to 8.4× **reduction** for fully connected kernel **without any additional accuracy loss**.
- *Stranded encoding of R1CS Constraints:* To further improve the number of R1CS constraints in the compilation result, ZEN incorporates a new *stranded encoding* technique, which optimally encodes multiple low-precision integers (8bit), that are common to quantized neural networks, with a single finite field element (usually 254bit). A caveat here is that encoding methods always come with some extra cost for extraction. Simply stacking the low-precision integers in the finite field does not work in our scenario, since the extraction coast is prohibitively high. Our stranded encoding mechanism comes with an efficient extraction. Additionally, our mechanism is adaptive, in that it always employees optimal encoding parameters, for any given input. Empirically evaluation shows that stranded encoding leads to up to 2.2× improvement in R1CS constraints for convolution kernel and 3.4× improvement for fully connected kernel.

– ZEN *toolchain:* We build an open-sourced toolchain [FQZ+] (Fig. 1) that takes a floating-point PyTorch model and converts it to ZEN schemes with all the above optimizations. Our evaluation shows that, without incurring any additional accuracy loss, ZEN brings 5.43 ∼ 22.19× (15.35× on average) savings, in the number of constraints, compared with a vanilla implementation of the neural network models in zkSNARK.

In a nutshell, while many existing works focus on improving the zero-knowledge proof system backends, ZEN demonstrates a powerful new approach: co-designing the domain-specific algorithms (neural network inference) and constraints compilation process. This new approach brings an orders of magnitude improvement on the performance.

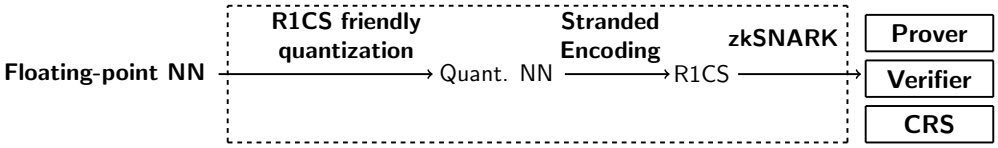

Fig. 1: Overview of ZEN tool chain.

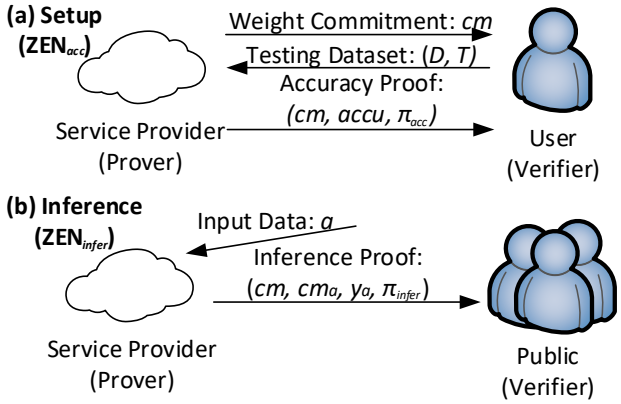

Fig. 2: Privacy-preserving, verifiable inference workflow.

1.2 Our Techniques

Privacy-preserving, verifiable inference schemes Neural networks are eating the world. We focus on how to let service providers, such as medical AI service, identity service, blockchain oracles [EJN], to provide verifiable proof of neural network inference result, without leaking either the user's privacy or the service providers neural network models.

To enable these privacy-preserving verifiable neural network inference applications, we propose two schemes:

– ZEN$_{acc}$, a verifiable NN accuracy scheme for classification and recognition workloads;
– ZEN$_{infer}$, a verifiable NN inference scheme for classification and recognition workloads.

The performance results are shown in Table 1 and Table 2. More details will be given in the corresponding section.

These two schemes are compatible with each other (Fig. 2). In a typical example, the service provider firstly demonstrates the effectiveness of her model during the setup phase using ZEN$_{acc}$. This is an interactive

Model-Dataset	Constraints (K)	Linear Combinations (K)	Setup (s)	Comm. (s)	Prove (s)	Verify (s)	CRS Size (GB)
ShallowNet-MNIST	4,380	18,008	154.87	0.310	147.07	0.097	1.395
LeNet-small-CIFAR	4,061	19,863	137.67	0.255	125.53	0.023	1.144
LeNet-medium-CIFAR	23,681	130,054	1033.59	1.069	1000.79	0.106	7.059
LeNet-Face-small-ORL	20,607	95,406	787.79	1.226	766.76	0.103	6.359
LeNet-Face-medium-ORL	83,416	474,277	4915.39	3.779	4710.13	0.372	25.531
LeNet-Face-large-ORL	318,353	1,818,988	20,000*	15.058	20,000*	1.5*	100*

Table 1: Overall performance of ZEN_{infer}. * indicates an estimated value.

Model-Dataset	Constraints (K)	Linear Combinations (K)	Setup (s)	Comm. (s)	Prove (s)	Verify (s)	CRS Size (GB)
ShallowNet-MNIST	13,181	75,210	291.67	0.366	256.02	0.17	1.75
LeNet-small-CIFAR	88,294	675,365	466.21	0.265	413.24	0.13	2.21
LeNet-medium-CIFAR	563,545	5,762,645	1511.20	1.069	1323.67	0.21	8.13
LeNet-Face-small-ORL	284,341	2,395,061	1316.51	1.081	1158.21	0.21	7.43
LeNet-Face-medium-ORL	1,809,239	21,266,440	5350.75	3.737	4863.41	0.47	26.61
LeNet-Face-large-ORL	5,792,208	77,689,757	20,000*	14.587	20,000*	1.5*	100*

Table 2: Overall performance of ZEN_{acc} scheme

protocol: she needs to first commit the neural network model into a commitment cm. Then, a user or a trusted third party sends her a random challenge: a dataset and truth label pair (D, T). Next, the service provider returns a zero-knowledge proof π_{acc}, proving that the committed neural network maintains an accuracy $accu$ on the input D.

Once this (one time) setup is completed, the service provider begins her service using ZEN_{infer}: the service provider collects the input data a from the end user, then generates a zero-knowledge proof π_{infer} to attest the result of inference using the previously committed model.

Both ZEN_{acc} and ZEN_{infer} leak no information of either the neural network model or the sensitive user input by the zero-knowledge property of zkSNARK.

Model	Commitment (K)	ZEN vanilla (K)	ZEN_{infer} (K)	Saving (×)
ShallowNet-MNIST	4,314	364	67	5.43
LeNet-small-CIFAR	3,289	16,809	772	21.77
LeNet-medium-CIFAR	18,306	85,126	5,375	15.84
LeNet-Face-small-ORL	18,022	57,352	2,585	22.19
LeNet-Face-medium-ORL	66,063	274,490	17,354	15.82
LeNet-Face-large-ORL	263,140	610,797	55,212	11.06

Table 3: Overall saving on the number of constraints of ZEN vanilla and ZEN_{infer}. Constraints from commitment are also shown above. The unit of number of constraints is thousand (K).

R1CS friendly quantization A crucial part of ZEN is converting neural network models with floating-points to arithmetic circuit (R1CS constraints) on a given finite field. This part is named *quantization*. Although neural network quantization has been extensively studied [JKC+18, DMM+18, MNA+18, SCC+19, LLQ+19, WPQ+18, SJG+20] and widely deployed [BSB, Alf, PyT, Ten], these existing works do not apply to zkSNARKs, for the following reasons. First, most schemes are *partial quantization*, in which part of the quantized models remain on signed floating numbers; second, most quantization schemes require operations such as divisions, which are non-atomic in an R1CS; last, directly applying the state-of-art full quantization algorithm [JKC+18] will produce a prohibitive number of R1CS constraints.

We address these challenges by proposing an improved, dedicated full quantization algorithm that minimizes the R1CS constraints. In this quantization algorithm, we incorporate two R1CS friendly optimizations

(§4.2), namely, *sign-bit grouping*, and *remainder-based verification*, to the [JKC+18] algorithm. The core idea behind both optimizations is to use *algebraic equalities* to bypass the expensive bit-decompositions caused by non-atomic operations such as comparisons and divisions:

- With the sign-bit grouping, we completely eliminate the bit-decompositions from element-wise zero-comparisons for *all* kernels.
- With the remainder-based verification, we reduce the bit-decompositions caused by divisions (due to the scale factor) up to 8.4× and 73.9× for fully connected and convolution kernels, respectively.

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 do not incur any additional accuracy losses, compared with the original quantization algorithm [JKC+18].

Stranded encoding of R1CS Constraints This technique comes from a fundamental observation: most quantized neural networks work with low precision (e.g. 8-bit) unsigned integers; on the other hand, most zkSNARKs use ellipitic curves (e.g. BLS12-381 [BLS02]) with an underlying finite field of order $\approx 2^{254}$. It makes sense that if we encode multiple matrix entries into a single finite field element, we should be able to reduce the number of constraints. This is analogous to the SIMD (Single Instruction Multiple Data) technique that is widely used in modern CPUs and GPUs [PH12].

As alluded earlier, extraction remains problematic. Indeed, simply stacking many 8-bit unsigned integers into a finite field element would not work since the extraction cost will out-weight the benefits. Instead, we propose a novel encoding scheme, namely *stranded encoding*, which encodes *batched vector dot products* with fewer field operations (§5). Given $\{A_j, B_j\}_{j=1}^{s}$ as inputs, where $|A_j| = |B_j| = n$, to compute s dot products simultaneously, i.e.,

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

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 parameters δ and $\phi(\cdot)$ (see Definition 2), these s dot products could be extracted from the following quantity

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

and the cost is not much different from a single extraction.

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 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 To evaluate the effectiveness of ZEN, we implemented both ZEN-vanilla, a straight-forward implementation of existing full quantization scheme [JKC+18], and the fully optimized ZEN toolchain. We summarize our benchmark results in Table 3. We see an improvement of 5.43∼22.19× (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 4 for a highlight of optimizations on LeNet-5-Small for CIFAR-10.

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

Table 4: Case study: benefits of optimizations on the number of constraints for the four kernels in LeNet-5-Medium CIFAR-10. Opt. Lv1 only includes sign-bit grouping; Opt. Lv2 adds remainder-based verification; Opt. lv3 adds stranded encoding.

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

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., 192 bytes for our choice of parameters. Verifying a proof can be done in subsecond 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

Verifiable machine learning We first compare ZEN with existing verifiable machine learning systems. Ginger [SVP+12] is an argument system that supports floating-point computation. Despite the improvement, Ginger's concrete efficiency is still far behind verifying arithmetic computation that can be directly expressed in finite fields. As a result, ZEN's approach takes advantage of the existing quantization techniques, which avoid expensive zero-knowledge proof systems for floating-point values.

SafetyNet [GGG17] implements a specialized interactive proof protocol for verifiable execution of a class of deep neural networks. Compared with SafetyNet, ZEN has three major advantages: first, ZEN's inference schemes is non-interactive; second, SafetyNet still leaks the neural network model, such as the weights, which are usually treated as trading secrets of the service providers; last, SafetyNet only supports quadratic activation functions, which are usually considered less effective than the widely used ReLU activation function [MHN13].

zkDT [ZFZS20] implements verifiable inference and accuracy schemes on decision trees. zkDT proposes protocols with tree-specialized commitment design. Verifiable inference and accuracy schemes for neural networks pose a whole different set of technical challenges. We also note that neural network inference itself has more complicated application scenarios, such as computer vision, natural language processing, and computational biology, compared with a limited scope of decision tree models.

vCNN [LKKO20] proposes a verifiable inference scheme for neural networks with zero-knowledge. ZEN differentiates from vCNN in the following aspects. First, ZEN includes an accuracy statement, so that the effectiveness of the neural network models from the service provider is verifiable, rather than being blindly trusted. Second, ZEN's optimizations are applicable to quantization for different neural network kernels (fully connected, convolution, ReLu, and pooling) while vCNN only optimizes convolution; Last, ZEN's optimization can be easily extended to any existing SNARKs that supports R1CS, without the need of introducing a mixing of QAP [GGPR13], QPP [KPP+14] and CP-SNARKs [CFQ19] as in vCNN.

To the best of our knowledge, ZEN is the first end-to-end optimizing compiler that compiles neural network models to R1CS constraints.

Other privacy models for 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. 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 on 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. The privacy models of these works are largely orthogonal to ZEN since none of these work can provide a *publicly verifiable* inference result with both user input data and the service provider model private.

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 various use cases. These systems usually come with diverse setups, verification time, and proof size, leading to trade-offs in these dimensions. In particular, we focus on the scheme developed by Groth [Gro16] and the Arkworks 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 R1CS-based proof systems and will deliver various performance preference, suiting dedicated use cases.

Neural network quantization Quantization has been widely studied [JKC+18, DMM+18, MNA+18, SCC+19, LLQ+19, WPQ+18, SJG+20] to accelerate neural networks by replacing `float32` data with low-precision data (*e.g.*, `float16`). Apple and XORNet.ai [BSB, Alf] have deployed quantization of neural network to accelerate neural network on devices with resource constraints. Facebook [PyT] and Google [Ten] have integrated quantization as standard feature of PyTorch and TensorFlow due to its wide usage. However, existing quantization techniques are usually not R1CS friendly due to two reasons. First, most quantization techniques [DMM+18, MNA+18, SCC+19] involve floating-point data. However, applying neural network models in zkSNARKs requires full quantization: converting a floating-point neural network model to a neural network model consisting of only integer arithmetic. Second, for a few quantization techniques [JKC+18] with full quantization, it still involves negative values and division operations, which cannot be efficiently supported in zkSNARKs. We use the full quantization scheme in [JKC+18] as in our baseline system ZEN-vanilla, since it establishes state-of-the-art accuracy on `uint8` quantization with minimal accuracy loss, for a variety of real-world neural networks.

1.4 Paper organization

?? 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 R1CS friendly quantization method and stranded encoding respectively. Finally, implementation and evaluations are reported in §6.

2 background

We briefly recall zk-SNARK in §2.1. Then, we explain the necessary background on neural networks to understand our optimizations in §2.2.

2.1 Cryptography

The major cryptographic building block used in this paper is zk-SNARK, formally *(publicly-verifiable, preprocessing,) zero-knowledge Succinct Non-interactive ARgument of Knowledge*. A zk-SNARK is defined 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 zk-SNARK 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.

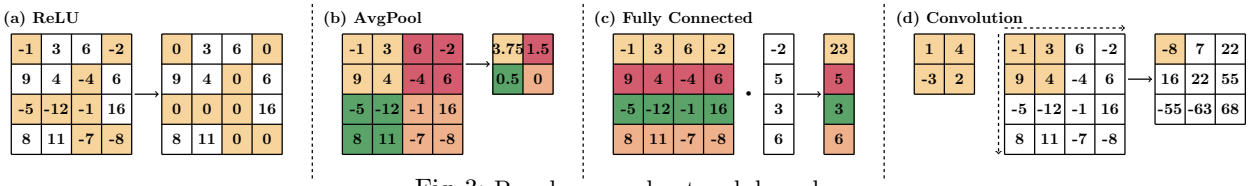

Fig. 3: Popular neural network kernels

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 $O_\lambda(|x|)$ [3].
- **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.

zk-SNARK'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]. In this paper, we use Arkworks implementation [ark] that was part of [BCG+20]. We will use the scheme by Groth [Gro16], commonly referred to as Groth16, to generate and verify proofs. This scheme is the state-of-the-art in terms of proof size and verifier efficiency.

[2] The concrete numbers are from [XZZ+19].
[3] $O_\lambda(\cdot)$ hides a fixed polynomial factor in λ.

In a typical Groth16-type of proving system, the statements that are to be proved are translated into a so-called Rank-1 Constraint System (R1CS). To prove that the prover knows some secret inputs that satisfy the given statements, is then converted into the satisfiability of the R1CS over any points over the field. The proof is therefore a demonstration that the R1CS is satisfied at a random point, which, as a prior, is agreed upon a trusted setup (result into the so-called *common reference string*) and remains unknown to both prover and verifier. Without going into further details, we note that the overall cost of the proving system is dominated by the number of constraints in the R1CS.

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.

2.2 Neural network based classification and recognition

Neural network (NN) brings significant advancement in computer vision [HZRS16, LBBH98], natural language processing [DS20, WJI+20] and computational biology [SEJ+19, SEJ+20]. Abstractly, An NN can be viewed as a function $Y = f(X_0)$ that takes an input $X_0 \in \mathbb{R}^{c\times h\times w}$ and generates an output $Y \in \mathbb{R}^m$: $f = f_l \circ f_{l-1} \circ \cdots \circ f_1$ is usually composed by a sequence of kernels. Each kernel f_i (also called a single layer in NN) is usually one of the following 4 "elementry" matrix operations (illustrated in Fig. 3):

1. *ReLU* [MHN13] kernel applies a simple non-linear activation function (*i.e.*, $\mathrm{ReLU}(X) = max(X, 0)$) element-wisely to the input matrix. This kernel enables NNs to learn non-linearity patterns of the input data.
2. *Average pool* kernel splits the input data spatially into a set of $r \times r$ grid (r=2 in Fig. 3(b)) and computes the mean over all values in each grid. This kernel spatially summarizes local image features and extract high-level features to facilitate pattern recognition.
3. *Fully connected* kernel takes two inputs (*i.e.*, a weight matrix $W \in \mathbb{R}^{m\times n}$ and a data matrix $X \in \mathbb{R}^n$) and computes an output matrix $W \cdot X \in \mathbb{R}^m$. This fully connected kernel mixes signals from individual pixels and extra high-level features based on the learnable weight matrix W.
4. *Convolution* kernel slides a weight matrix W along the height and width of the input matrix X, retrieves a slice of input matrix with the same spatial size (2×2 in Fig. 3(d)), and computes the dot product to generate a real number. This kernel mixes local signals (*e.g.*, local edges and angles) in contrast to fully connected kernel that mixes global signals over all pixels. Note that convolution kernel can be transformed into matrix-matrix multiplications [CWV+14], which can be computed similar as fully connected kernel.

Two prominent use cases of NNs are classification [HZRS16, LBBH98] and recognition [SYS+20, WWZG20]. The classification task puts an input into one of m candidate classes (*e.g.*, cat, dog, and house). In this case, the final layer f_l is a softmax function that outputs $Y = [y_1, y_2, ..., y_m]$ which satisfies $0 \le y_j \le 1$ and $y_1 + \cdots y_m = 1$. As a result, y_j can be treated as the probability that the input image X_0 has class j. The classification result is the class with the highest predicted probability: $\hat{y} = \mathrm{argmax}_j\, y_j$.

The recognition task compares an input X to a reference input X_R and decides whether the input and the reference input are a same object. Different from classification, the final layer f_l takes neural embeddings of both the input and the reference, and then computes their distance d in a metric space (*e.g.*, Eculidean space). These two objects are decided as a same object, if their distance d is smaller than a pre-defined threshold τ (e.g. 0.5).

3 ZKP for accuracy and inference

In this section, we present our constructions for accuracy and inference. We first introduce a verifiable neural network accuracy scheme that proves the accuracy of neural network models without revealing the weights of the models (§3.1). We then introduce a verifiable neural network inference scheme for classification and recognition (§3.2). Here, we consider a quantized neural network $\mathcal{Q} : \mathbb{F}_p^{d_1} \to \mathbb{F}_p^{d_2}$ which is a mapping from a size d_1 vector over $\mathbb{F}_p$ to a size d_2 vector on the same field. While neural networks are usually floating-point models, we defer our R1CS friendly quantization to §4.

We remark that we will use Groth16 method to generate proofs. This implies that the prover and the verifier need to agree on certain common reference string (CRS) that is generated through either a trusted third party, or a multiparty computation protocol. For a dedicated verifier use case, it may also be sufficient for the verifier to generate the CRS.

3.1 $\mathbf{ZEN}_{acc}$

We present ZEN_{acc}, a *zero-knowledge, verifiable neural network accuracy scheme*, that works for the following typical scenario: the prover firstly commits to a private neural network, then proves the prediction accuracy of the committed neural network, on either a public or a verifier-chosen testing dataset. The verifier learns nothing about the model except its prediction accuracy.

Formally, we consider a testing dataset $\mathcal{D} = \{a_1, a_2, \ldots, a_n\}$ and the corresponding truth labels $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ for the testing dataset. In classification task, $t_i \in \{1, 2, ..., m\}$ indicates whether the image a_i contains object t_i. In recognition task, $t_i \in \{0, 1\}$ indicates whether the current image a_i contains the same person as a reference image a_{ref}. Given a neural network $\mathcal{Q}$, we define a *classification computation routine* as: a) running the neural network Q on the testing dataset $\mathcal{D}$ provided by the verifier to get prediction results $\mathcal{Y} \leftarrow \mathcal{Q}(\mathcal{D})$; b) comparing $\mathcal{Y}$ with their truth label $\mathcal{T}$ to obtain the accuracy, denoted by *accu*, of the neural network prediction.

For recognition task, we additionally define a distance metric $\mathcal{L} : \mathbb{F}_p^{d_2} \times \mathbb{F}_p^{d_2} \rightarrow \mathbb{F}_p$ over the embedding space, $\tau \in \mathbb{F}_p$ be the agreed threshold, and a ground truth embedding y_g. Given a neural network $\mathcal{Q}$, we define a *recognition computation routine* as: a) running the neural network $\mathcal{Q}$ on $\mathcal{D}$ to obtain the embedding results $\mathcal{Y} = \{y_1, y_2, \ldots, y_n\}$; b) for each y_i in $\mathcal{Y}$, calculate $\mathcal{L}(y_i, y_g) \leq \tau$ to obtain the recognition prediction results $\mathcal{R}$; c) comparing $\mathcal{R}$ and $\mathcal{T}$ and obtaining the recognition accuracy results *accu* on testing dataset $\mathcal{D}$.

Based on these two computation routines, we define a zero-knowledge, verifiable neural network accuracy scheme (ZEN_{acc}) as the following algorithms:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{acc}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$: given a security parameter λ and a quantized neural network model $\mathcal{Q}$ for classification or recognition tasks, randomly generate a proving key pk and a verification key vk.
- $cm \leftarrow \text{ZEN}_{acc}.\mathsf{Commit}(\mathcal{Q}, r)$: given a random opening r, the prover commits to the neural network $\mathcal{Q}$ with r, i.e., $cm \leftarrow \mathsf{COMM}(r, \mathcal{Q})$.
- $(accu, \pi_{acc}) \leftarrow \text{ZEN}_{acc}.\mathsf{Prove}(\mathsf{pk}, \mathcal{D}, \mathcal{T}, \mathcal{Q})$: Upon receiving the commitment, the verifier sends a testing dataset $\mathcal{D}$ to the prover. Then the prover runs either classification or recognition computation routine for each data sample in $\mathcal{D}$, compares with the predictions with their truth labels in $\mathcal{T}$ and outputs *accu*, the prediction accuracy of the neural network $\mathcal{Q}$. Finally, the prover generates a corresponding proof π_{acc} for this computation routine.
- $\{0, 1\} \leftarrow \text{ZEN}_{acc}.\mathsf{Verify}(\mathsf{vk}, cm, \mathcal{D}, \mathcal{T}, accu, \pi_{acc})$: given the verification key vk and proof π_{acc}, verifies if the following statements are correct: the number of correct predictions on dataset $\mathcal{D}$ with model $\mathcal{Q}$ is *accu*; *cm* is a commitment to $\mathcal{Q}$.

3.2 $\mathbf{ZEN}_{infer}$

Our zero-knowledge, verifiable neural network inference scheme (ZEN_{infer}) assumes the prover keeps the neural network private, but will publicly commit to it (binding). At a later time, the prover generates a proof for the result of the classification or recognition. Upon receiving the model commitment and the proof, the verifier checks if the proof is valid or not. During the whole process, the prover's neural network is kept secret (hiding).

Formally, we consider a public input $a \in \mathbb{F}_p^d$ and a neural network $\mathcal{Q}$. We define the *classification inference routine* as running the neural network $\mathcal{Q}$ on the input a to get $y_a \leftarrow \mathcal{Q}(a)$. For the recognition task, we additionally have the distance metric $\mathcal{L}$, agreed threshold τ, and the ground truth embedding y_g. We define the *recognition inference routine* as running the neural network $\mathcal{Q}$ on a to obtain the embedding y_a and calculating $\mathcal{L}(y_a, y_g) \leq \tau$ to obtain the recognition prediction result.

Based on these two inference routines, we define a zero-knowledge, verifiable neural network based inference scheme (ZEN_{infer}) as the following algorithms:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{infer}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$: 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.
- $(cm, cm_a, y_a, \pi_{inf}) \leftarrow \text{ZEN}_{infer}.\mathsf{Prove}(\mathsf{pk}, a, r, s, \mathcal{Q})$: given an input $a \in \mathbb{F}_p^d$ and a random opening r and s, the prover commits to the input with r and the neural network model with s, i.e., $cm_a \leftarrow \mathsf{COMM}(r, a)$ and $cm \leftarrow \mathsf{COMM}(r, \mathcal{Q})$, respectively. Then the prover runs the classification or the recognition inference routine to get a result. Finally, the prover generates a proof π_{inf} for the above process.
- $\{0, 1\} \leftarrow \text{ZEN}_{infer}.\mathsf{Verify}(\mathsf{vk}, cm, cm_a, y_a, \pi)$: validate input a's inference result on model $\mathcal{Q}$ given the verification key vk, a's commitment, $\mathcal{Q}$'s commitment, a's inference result, and a zero-knowledge proof π_{inf}.

4 R1CS Friendly Quantizaiton

In this section, we introduce our R1CS friendly quantization. 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 the full quantization scheme in [JKC+18] to the existing zkSNARK library and achieve our baseline system, ZEN-vanilla. Then, we introduce two R1CS friendly optimizations on top of our baseline system, to significantly reduce the number of constraints while maintaining an equivalent accuracy. Last, we demonstrate these R1CS friendly quantizations can be applied to 4 major neural network kernels: fully connected, convolution, average pool, and ReLU.

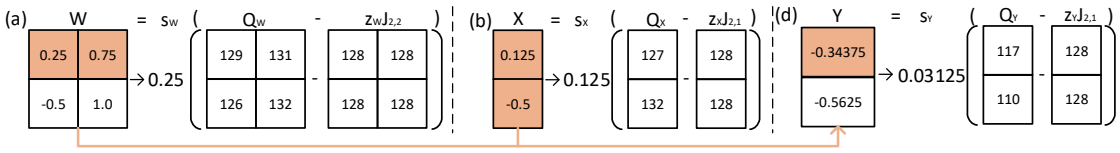

Fig. 4: Illustration of Partial Quantization when computing $Y = WX$.

4.1 Baseline: ZEN-vanilla

As discussed in §2.2, given a floating-point weight matrix W and a floating-point input 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 illustrated in Fig. 4. As a result, we have the quantized representation:

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

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

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:

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

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

Efficiency Challenges in ZEN-vanilla. While ZEN-vanilla produces effective privacy-preserving and verifiable neural network models, it is not efficient due to its large number of constraints. There are two main reasons coming from [JKC+18]. First, [JKC+18] is developed for signed `int8` on integer-only hardware, allowing negative elements in $(Q_W - z_W\mathbf{J}_{m,n})$ and $(Q_X - z_X\mathbf{J}_{n,1})$ of Eq. 2. However, zk-SNARK supports only finite field arithmetic and requires expensive sign checks for signed integers.As a result, for $mn + n$ elements in $(Q_W - z_W\mathbf{J}_{m,n})$ and $(Q_X - z_X\mathbf{J}_{n,1})$, one may still need $O(mn)$ sign-checks in the generated constraints, where each sign-check needs expensive bit-decomposition. Considering that Eq. 2 accounts for most computations in neural networks in terms of fully connected kernels and convolution kernels, this would lead to significant overhead. Second, [JKC+18] involves division operations while the division operation is non-atomic in zk-SNARK 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. 2. While this strategy allows verifying Eq. 2, it would introduce heavy overhead from the bit decomposition.

4.2 R1CS Friendly quantizations

In this section, we introduce two R1CS friendly optimizations on top of the baseline quantization scheme, namely, sign-bit grouping and remainder-based verification. Here, we focus on fully connected kernel and convolution kernel. Both can be viewed as matrix multiplications (Eq. 1). We defer the discussion of ReLU kernel and average pool kernel to the next section. 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. Our *sign-bit grouping* first reformulates Eq. 2 to:

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

Here, both sides are guaranteed to be positive. While $(Q_W - z_W)$ and $(Q_X - z_X)$ may have negative elements, we use the associativity of matrix multiplication to group operands of the same sign:

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

Note that we add z_Y before subtraction such that all intermediate elements in the reformulated system are 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 $Q_W \in \mathbb{R}^{m\times n}$, where p is the order of the finite field used by zk-SNARK.

Remainder-based verification. NN computation involves abundant division computations in average pool kernels (*e.g.*, division by 4) and quantization (*e.g.*, division by 2^k). These division computations usually lead to non-integers and make it costly to verify since zk-SNARK supports only integers. A naïve approach is to use expensive bit-decomposition operations in zk-SNARK. To efficiently verify division operation in zk-SNARKs, we propose a *remainder-based verification optimization* to avoid this high overhead. 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 zk-SNARK systems. Formally, instead of Eq. 3, we prove Eq. 4:

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

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).

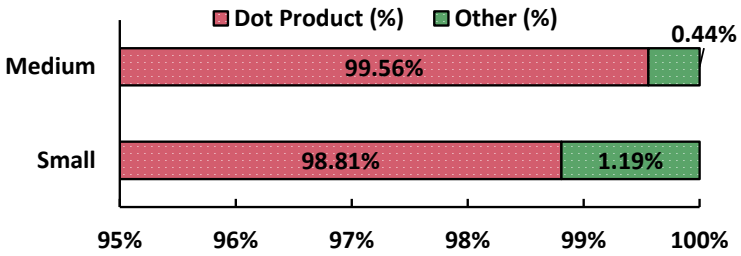

Fig. 5: Dot product/other computation Ratio in LeNet-5-Medium and LeNet-5-Small.

4.3 Adapting R1CS friendly quantization for average pool and ReLU kernels

Now, we show how to adapt the previously introduced R1CS friendly quantization techniques to average pool and ReLu kernels as well.

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. 3(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

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

Let's briefly summarize the obstacles. First, the average pooling operator contains division operation, which is not generally supported in zk-SNARK 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 aforementioned remainder-based verification strategy to the average pool kernels. In particular, we first use an extra scalar γ to store the division remainder and use the following verification for the average pooling kernel

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

ReLU kernel. The ReLU kernel contains only maximum operations, and is used to extract nonlinear neural network features, as illustrated in Fig. 3(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.

5 Optimizing Matrix Operation Circuits using Stranded Encoding

In this section, we propose the *stranded encoding*, a general methodology of optimizing matrix operation circuits for zk-SNARKs. Our profiling on neural networks shows that matrix operation, especially dot products, consumes most computation in neural networks, as shown in Fig. 5.

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

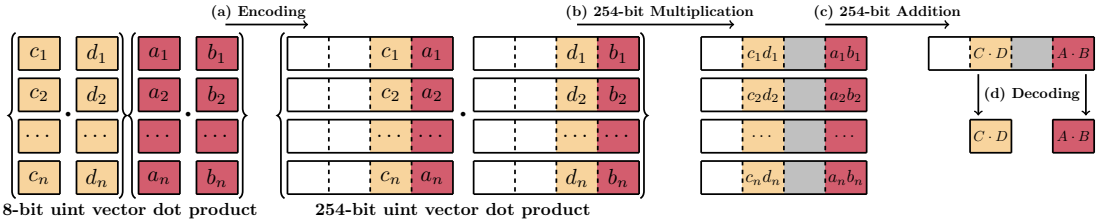

Fig. 6: Stranded encoding with batch size $s = 2$

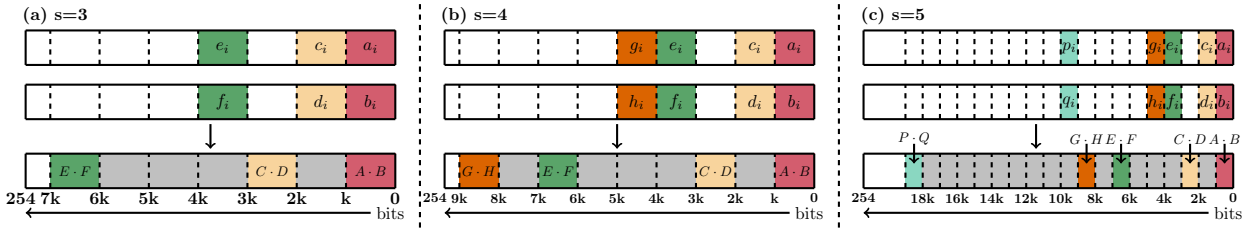

Fig. 7: Data layout for variable batch size s. Each block has k bits and can store a value up to 2^k. k is a hyperparameter that is generally larger than 8 to accumulate the dot product and avoid overflow.

(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 into a single finite field element.

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

$$x = a_1 + a_2\delta, \qquad y = b_1 + b_2\delta.$$

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_ib_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 $A \cdot B = a_1b_1 + a_2b_2$ and $C \cdot D = c_1d_1 + c_2d_2$, 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 $\sum x_iy_i$, since:

$$\begin{aligned} x_1y_1 &= a_1b_1 + (a_1d_1 + c_1d_1)\delta + c_1d_1\delta^2 \\ x_2y_2 &= a_2b_2 + (a_2d_2 + c_2d_2)\delta + c_2d_2\delta^2 \\ x_1y_1 + x_2y_2 &= (a_1b_1 + a_2b_2) + (\ldots)\delta \\ &\quad + (c_1d_1 + c_2d_2)\delta^2 \end{aligned}$$

We can extract $A \cdot B$ and $C \cdot D$ from $x_1y_1 + x_2y_2$ 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. 6:

$$A \cdot B = \sum_{i=1}^{n} a_ib_i, \quad C \cdot D = \sum_{i=1}^{n} c_id_i$$

We encode x_i and y_i as:

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

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:

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

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

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

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

$$x_i = a_i + c_i\delta + e_i\delta^2, \quad y_i = b_i + d_i\delta + f_i\delta^2.$$

Then, the multiplication becomes:

$$\begin{aligned} x_i y_i =& a_i b_i + (a_i d_i + b_i c_i)\delta + (c_i d_i + a_i f_i + b_i e_i)\delta^2 \\ &+ (c_i f_i + d_i e_i)\delta^3 + e_i f_i \delta^4 \end{aligned}$$

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 δ^2. To solve this problem, we use the following encoding instead for $i \in \{1, ..., n\}$:

$$x_i = a_i + c_i\delta + e_i\delta^3, \quad y_i = b_i + d_i\delta + f_i\delta^3.$$

Now it is not hard to see:

$$x_i y_i = a_i b_i + (\ldots)\delta + (c_i d_i)\delta^2 + \ldots + (e_i f_i)\delta^6 \tag{6}$$

As a result:

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

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

Definition 2 (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 $x_i, y_i \in \mathbb{F}_p, p \geq 2^{w_{out}}, i \in \{1, \ldots, n\}$ as follows:*

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

where $\delta = 2^{2w_{in}+\log n}$ *and* $\phi(\cdot) : \{1, \ldots, s\} \to \mathbb{N}$ *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:

$$\sum_{i=1}^{n} x_i y_i = (A_1 \cdot B_1)\delta^{2\phi(1)} + \ldots + (A_2 \cdot B_2)\delta^{2\phi(2)} + \ldots + (A_s \cdot B_s)\delta^{2\phi(s)} \tag{9}$$

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 ends 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. 7). 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 5.

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 zk-SNARK implementation we use, bit decomposition of a finite field element 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 shall not do stranded encoding since the amortized cost is greater than 1; for $632/3 \leq n \leq 4096$, we choose a batch size of 4; and for $n > 4096$, we choose a batch size of 3. We believe this cost function is useful when determining both the integer precision during the neural network quantization and the underlying field field used by a zk-SNARK.

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}	-

Table 5: Largest supported vector size n_{max} in stranded encoding for different batch size s ($\omega_{in} = 8$ and $\omega_{out} = 254$). '-' indicates not supported.

6 Evaluation

We first describe ZEN's implementation details and evaluation settings in §6.1. Next, we demonstrate the effectiveness of our proposed optimizations on reducing constraints in §6.2. Then, we show the benefits of reducing linear combinations in §6.3. Last, we show the end-to-end results on the number of constraints as well as accuracy in §6.4.

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

Table 6: Neural Networks used in our evaluation.

6.1 Implementation and Evaluation Settings

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 zk-SNARK friendly quantizations, and produces a quantized neural network model. Circuit generators generate individual components of circuit. We implemented circuit generators for FC, Conv, average pooling, and ReLU kernels. Our system also includes a commitment circuit generator from the underlying zk-SNARK system we used. We implement stranded encoding as part of 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_{infer} scheme. Our system uses arkworks' implementation [ark] of Groth16 scheme [Gro16] as the underlying zk-SNARK. We choose the BLS12-381[BLS02] as the underlying curve for in Groth16.

Datasets. We select three popular datasets (MNIST, CIFAR-10, and ORL) used by many secure machine learning projects [DSC+19, ZYCW20, GDL+16, JVC18, JKLS18]. 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,

- MNIST is a large dataset for handwritten digit 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 kernels and one ReLU kernel. LeNet has three convolutional kernels, two fully connected kernels, and four ReLU kernels. These variants have different kernel sizes for adapting to different sizes of inputs. The evaluation on these six variants demonstrates the performance of ZEN under diverse model sizes.

Experiment Configuration. All the evaluations run on a Microsoft Azure M32ls instance with 32 core Intel Xeon Platinum 8280M vCPU @ 2.70GHz and 256 GiB DRAM. We compile ZEN code using Rust 1.47.0 in release mode.

6.2 ZEN optimizations on reducing constraints

R1CS friendly quantization. We demonstrate the optimization benefits from R1CS friendly quantization on convolution (Conv) kernels and fully connected (FC) kernels with diverse kernel sizes. We choose Conv and FC kernels since they take up to 60% share in the total number of constraints. We report the benefits from sign-bit grouping and remainder-based verification for each kernel.

Fig. 8 shows the constraint saving on Conv kernels. We observe that the number of constraints can be reduced by $3.4\times$ to $73.9\times$ with R1CS friendly quantization, including the sign-bit grouping and remainder-based verification. 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 $16 \times 4 \times 5 \times 5$. Fig. 9 shows similar constraint saving (from $1.1\times$ to $8.4\times$) for FC kernels from R1CS-friendly quantization.

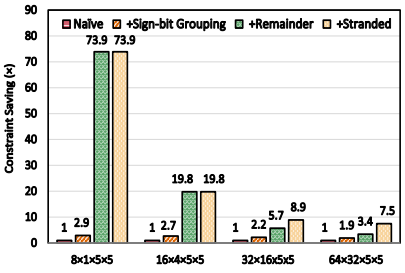

Fig. 8: Constraint saving on conv kernels. Shape is [# of in channels]×[# of out channels]×[kernel width]×[kernel height].

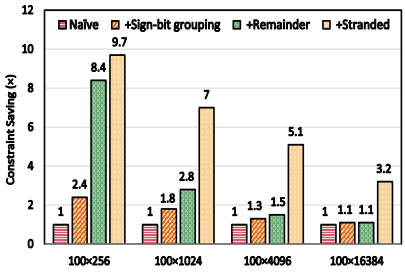

Fig. 9: Constraint saving FC kernels. Shape is [# of in channels]×[# of out channels].

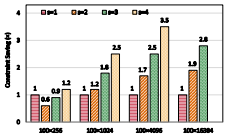

Fig. 10: Constraint saving on FC kernels with different batching size s. Shape is [# of in channels] × [# of out channels]. [4]

From Table 4 we know that AvgPool obtains 3.0× and 633× constraint reduction from the sign-bit grouping and remainder-based verification in R1CS-friendly quantization. ReLU obtains 1.2× constraint reduction from sign-bit grouping. This improvement is regardless of AvgPool and ReLU kernel size, because the constraint reduction for each division in AvgPool is fixed and independent of the number of division operations, and similarly for ReLU.

Stranded encoding. We first show the effectiveness of stranded encoding with the optimal batch s determined by the optimizer. Since stranded encoding can only be applied to FC kernel and Conv. kernel, we show constraint savings cause by stranded encoding in Fig. 8 and Fig. 9. When applied to Conv. kernel, stranded encoding didn't get triggered when the kernel sizes are small ($8 \times 1 \times 5 \times 5$ and $16 \times 4 \times 5 \times 5$). This is because the optimizer decides that the gain of stranded encoding will be negative. For larger size kernels, stranded encoding brings 1.6× to 2.2× constraint size savings on top of R1CS friendly quantization. We observe a similar trend when applying stranded encoding to FC kernel in Fig. 9: it brings additional 1.2× to 2.9× constraint reduction for FC kernels of different sizes. The larger FC kernel is, the more benefit from stranded encoding it gets. This is because the amortized cost of stranded encoding decreases proportionally as the kernel size increases.

Selecting the optimal batch size s in stranded encoding. We further demonstrate the benefits of stranded encoding under different batch sizes s (Fig. 10). Comparing across different batch sizes s, we notice that a larger batch size usually leads to higher savings. This result shows that we should choose a larger batch size s when the vector satisfies the corresponding length requirement. We also 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 a 2.8× constraint saving, which almost reaches the theoretical upper bound on constraint saving. Meanwhile, stranded encoding brings little benefits on small FC kernels (*e.g.*, 100×256). The reason is that the encoding and decoding overheads are constant across FC sizes and become relatively small on large FC sizes. This observation motivates the usage of $s = 1$ for these small FC kernels. Our optimizer maximizes the benefits from stranded encoding by automatically selecting the best batch size s.

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

Table 7: Accuracy comparison between floating-point (FP) models and R1CS friendly quantized models.

[4] We skip $s = 4$ for the FC kernel of shape 100×16384 since our stranded encoding with $s = 4$ requires a vector of length less than $2^{12} = 4096$, as described in Table 5

Case study: end-to-end results on LeNet-5-Medium. We show end-to-end results of different levels of optimizations on inference using LeNet-5-Medium on CIFAR-10 in Table 4. Starting from ZEN-vanilla, we add individual optimizations one by one 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. 9. These results show that our optimization techniques can significantly mitigate the intensive number of constraints on NN workloads and enable a more efficient deployment of ZEN.

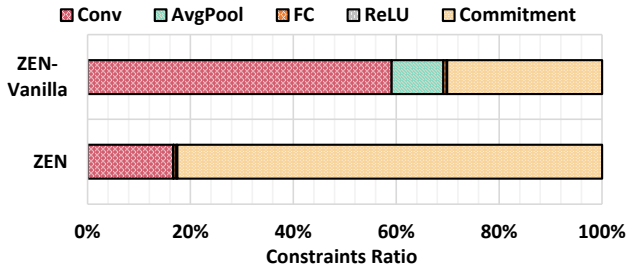

Fig. 11: Breakdown of constraints in LeNet-Face-large ORL dataset from both ZEN-vanilla and ZEN.

Constraint size breakdown by kernel type. We breakdown the number of constraints in LeNet-Face-Large on CIFAR-10 circuits generated by ZEN-vanilla and fully optimized ZEN in Fig. 11. We split the constraints into those from the commitment scheme and those from 4 different kinds of kernels. Overall, we observe that convolution kernels and fully connected kernels are the dominant sources of constraints in the ZEN-vanilla (59.2%) implementation. Since these two kinds of kernels heavily rely on dot products, this justifies our efforts on improving dot product circuit size by using stranded encoding. It is worth noting that commitment accounts for only 30.1% constraints in the ZEN-vanilla, but this ratio significantly rises to 82.6% 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 inference part. Further improving the commitment size to make ZEN even more efficient is our future work.

6.3 ZEN optimizations on reducing linear combinations

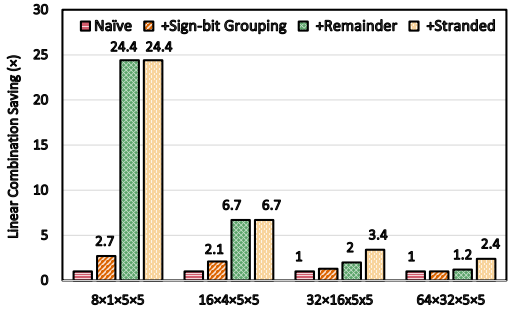

Fig. 12: LC saving on conv kernels. Shape is [# of in channels]×[# of out channels]×[kernel width]×[kernel height].

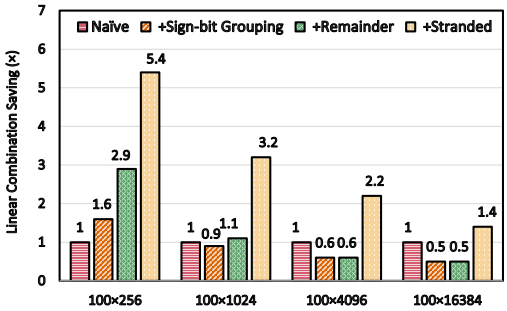

Fig. 13: LC saving on FC kernels. Shape is [# of in channels]×[# of out channels].

Despite the traditional wisdom that the prover time is mostly decided by the number of constraints, we find this is not the case for ZEN schemes on arkworks implementation of Groth16 protocol [ark], especially

on the bigger neural network models (which translates to a larger number of constraints). In Table 8, we breakdown the run time of ZEN$_{infer}$'s proving time into four parts: synthesizing constraints from circuits (Const. syn.), inlining linear combinations (Inline LCs), transforming R1CS to QAP (R1CS to QAP), and computation in groups (Comp. in Gs). Surprisingly, as the number of constraints goes up, the time spent on inlining linear combinations goes up and dominates the run time.

Fortunately, our optimizations reduce the number of linear combinations (LCs) as well. We demonstrate the effectiveness of our optimizations in terms of reducing the number of LCs in Fig. 12 and Fig. 13. We can observe a result that is similar to our savings of constraint sizes shown in §6.2. Overall, our optimizations save the number of LCs from 1.4× to 24.4× in ZEN$_{infer}$. We also observe that stranded encoding consistently saves the number of LCs across various kernel sizes. Meanwhile, sign-bit grouping and remainder-based verification may slightly increase the number of LCs on certain NN kernels (*e.g.*, 100×4096 and 100×16384 in Fig. 13).

Model-Dataset	Const. syn.	Inline LCs	R1CS to QAP	Comp. in Gs
ShallowNet-MNIST(4M)	21.6%	11.6%	17.3%	49.5%
LeNet-small-CIFAR(4M)	27.7%	16.2%	12.1%	44.0%
LeNet-medium-CIFAR(23M)	21.0%	28.9%	13.2%	36.9%
LeNet-small-ORL(20M)	23.1%	20.8%	14.9%	41.2%
LeNet-medium-ORL(83M)	17.9%	39.4%	12.5%	30.2%
LeNet-large-ORL(318M)	15.1%	67.4%	8.8%	15.7%

Table 8: Prover time breakdown of ZEN$_{infer}$ using Arkworks implementation of Groth16

6.4 End-to-end performance

In this section, we evaluate the end-to-end performance of ZEN in terms of accuracy, savings on total number of constraints, savings on total number of LCs, setup/proving time, and CRS sizes.

ZEN is accurate. In Table 7, we list the accuracy of our quantized model compared with the original floating-point PyTorch models. One can see the accuracy drop is minimal and some of them are within the error bound of different implementation of the same machine learning models. Additionally, compared with the state of the art full quantization scheme [JKC+18], ZEN has exactly the same accuracy thanks to our semantic preserving optimizations proposed in §4. As a result, we can conclude that ZEN maintains the accuracy of neural network models in practice.

Overall saving on the number of constraints. Table 3 shows the overall saving on the number of constraints with our optimizations. Overall, we can significantly reduce the number of constraints by 15.35× 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 11.06×. On small models of LeNet-5-small and LeNet-Face-small, we achieve more than 22.19× saving. This is mainly due to the fact that kernels in these small models are dot products of vectors which can be drastically improved with our sign-bit grouping and remainder-based verification. This result is similar to our microbenchmark in Fig. 8 and Fig. 9.

Overall saving on the number of LCs. Table 9 shows that the number of LCs in ZEN$_{infer}$ varies significantly from 509K to 766,147K. Our optimizations save the number of LCs from 2.34× to 8.03× (5.34× on average).

Model-Dataset	Commitment LCs(K)	ZEN vanilla LCs(K)	ZEN$_{infer}$ LCs(K)	Saving (×)
ShallowNet-MNIST	17,349	1,193	509	2.34
LeNet-small-CIFAR	13,487	51,231	6,375	8.03
LeNet-medium-CIFAR	73,405	263,779	56,649	4.66
LeNet-small-ORL	72,423	175,066	22,982	7.62
LeNet-medium-ORL	264,501	856,834	209,775	4.08
LeNet-large-ORL	1,052,862	1,944,147	766,125	2.54

Table 9: Overall saving on the # of linear combinations in ZEN$_{infer}$.

Setup time, prover time and CRS sizes. Table 1[5] shows the overall performance of ZEN_{infer} on 6 neural networks with diverse number of constraints, ranging from $4,061$ thousands to $318,505$ thousands. We observe that a model with a higher number of constraints comes with higher prover and setup time consumption and a larger Common Reference String (CRS) size. For a small model ShallowNet on MNIST with $4,380$ K constraints, we have a short time of 154 seconds and 147 seconds for setup and proving, respectively; the CRS consists of 1.395 GB of data. The overall cost increases as the number of constraints increases. For large models such as LeNet-Face-large on ORL with over $318,505$ thousand constraints, ZEN_{infer} requires nearly $20,000$ seconds for setup and proving respectively. Its CRS size is around 100 GB.

Due to the large memory and time consumption by ZEN_{acc}, we can parallelize the testing dataset inference step across multiple machines and add a prediction accuracy commitment sum check circuit. Table 2[6] shows the overall performance of ZEN_{acc} on 6 neural networks with a testing dataset size of 100. The constraints of ZEN_{acc} here consist of the commitment to the neural network model, the inferences on 100 images, and a commit to the final accuracy. The number of constraints of ZEN_{acc} ranges between $13,181$ K to $5,792,208$ K on 6 neural networks.

[5] In Table 1 and Table 2, we calculate the number of linear combination before inlining. It influences the time spent on inlining all linear combinations.

[6] We record the time spent on model commitment, inference on one image plus the final accuracy commitment check circuit as the ZEN_{acc} execution time.

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.

Alf. Anthony Alford. Apple acquires edge-focused ai startup xnor.ai. https://www.infoq.com/news/2020/01/apple-acquires-xnor-ai/.

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.

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.

BSB. Alan Boyle, Taylor Soper, and Todd Bishop. Exclusive: Apple acquires xnor.ai, edge ai spin-out from paul allen's ai2, for price in $200m range. https://www.geekwire.com/2020/exclusive-apple-acquires-xnor-ai-edge-ai-spin-paul-allens-ai2-price-200m-range/.

CFQ19. Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *CCS*, pages 2075–2092. ACM, 2019.

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.

CWV+14. Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

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.

EJN. Steve Ellis, Ari Juels, and Sergey Nazarov. Chainlink: A decentralized oracle network. https://link.smartcontract.com/whitepaper.

FQZ+. Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen. https://github.com/UCSB-TDS/ZEN.

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.

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

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.

KPP+14. Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: Faster verifiable set computations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 765–780, San Diego, CA, August 2014. USENIX Association.

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.

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.

LLQ+19. Rundong Li, Feng Liang, Hongwei Qin, Yan Wang, Rui Fan, and Junjie Yan. Fully quantized network for object detection. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019*, 2019.

MHN13. Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc ICML*, 2013.

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.

PS. Tony Peng and Michael Sarazen. The staggering cost of training sota ai models. https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-models/.

PyT. PyTorch. Pytorch quantization. https://pytorch.org/docs/stable/quantization.html.

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.

SEJ+19. Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Protein structure prediction using multiple deep neural networks in the 13th critical assessment of protein structure prediction (casp13). *Proteins: Structure, Function, and Bioinformatics*, 87(12):1141–1148, 2019.

SEJ+20. Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 2020.

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

SJG+20. Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks. In *International Conference on Learning Representations*, 2020.

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.

SVP+12. Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, pages 253–268. USENIX Association, 2012.

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.

Ten. TensorFlow. Post-training quantization. https://www.tensorflow.org/lite/performance/post_training_quantization.

Wig. Kyle Wiggers. Openai's massive gpt-3 model is impressive, but size isn't everything. https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt-everything/.

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.

WPQ+18. Yi Wei, Xinyu Pan, Hongwei Qin, Wanli Ouyang, and Junjie Yan. Quantization mimic: Towards very tiny cnn for object detection. In *15th European Conference on Computer Vision, ECCV 2018*, 2018.

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.