Fast Lossless Neural Compression with Integer-Only Discrete Flows

Siyu Wang [1] **Jianfei Chen** [1] **Chongxuan Li** [2] **Jun Zhu** [1] **Bo Zhang** [1]

Abstract

By applying entropy codecs with learned data distributions, neural compressors have significantly outperformed traditional codecs in terms of compression ratio. However, the high inference latency of neural networks hinders the deployment of neural compressors in practical applications. In this work, we propose Integer-only Discrete Flows (IODF), an efficient neural compressor with integer-only arithmetic. Our work is built upon integer discrete flows, which consists of invertible transformations between discrete random variables. We propose efficient invertible transformations with integer-only arithmetic based on 8-bit quantization. Our invertible transformation is equipped with learnable binary gates to remove redundant filters during inference. We deploy IODF with TensorRT on GPUs, achieving $10\times$ inference speedup compared to the fastest existing neural compressors, while retaining the high compression rates on ImageNet32 and ImageNet64.

1. Introduction

As a growing amount of data is produced every day, efficient lossless compression is of significance in storing and transmitting them. Shannon's source coding theorem (Shannon, 1948) states that the average code length needed to encode data is lower bounded by entropy of its distribution:

$$\mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}}[|c(\mathbf{x})|] \geq \mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}}[-\log p_{\mathcal{D}}(\mathbf{x})], \tag{1}$$

where $|c(\mathbf{x})|$ is the code length and $p_{\mathcal{D}}$ is the data distribution. Based on the insight that the optimal code length for a single symbol $\mathbf{x}$ is $-\log p_{\mathcal{D}}(\mathbf{x})$, many efficient entropy codecs (Huffman, 1952; Duda, 2009; 2013) have been developed, and they have achieved nearly optimal code length given known data distribution. However, the data distribution is generally unknown in practice.

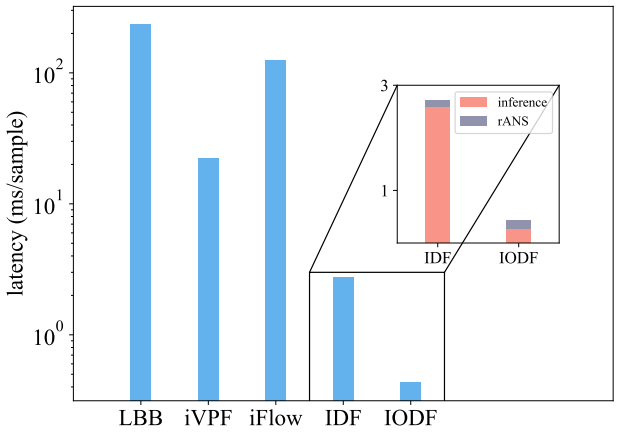


Figure 1. Encoding latency of different neural compressors measured by milliseconds per sample. Statistics of LBB, iVPF, and iFlow are directly picked from (Zhang et al., 2021c;b). The inner figure shows inference and entropy coding latency of IDF and IODF separately. For IDF, inference is the time bottleneck in the encoding process. IODF significantly improves the inference speed, making it comparable to entropy coding.

In the machine learning community, various likelihood-based generative models have been developed, including normalizing flows (Dinh et al. (2017); Ho et al. (2019a); Chen et al. (2020b); Lu et al. (2021), NFs), variational auto-encoders (Kingma & Welling (2013); Rezende et al. (2014), VAEs), auto-regressive models (Domke et al. (2008); Larochelle & Murray (2011); Salimans et al. (2017), ARMs), and diffusion models (Sohl-Dickstein et al. (2015); Ho et al. (2020); Song et al. (2021), DPMs). These deep generative models (DGMs) are powerful density estimators. With a model distribution close enough to the data distribution, many efficient lossless *neural compressors* have been explored (Hoogeboom et al., 2019; van den Berg et al., 2020; Ho et al., 2019b; Townsend et al., 2019a; Kingma et al., 2021). Neural compressors have achieved superior compression ratios on standard datasets than traditional lossless compression methods, such as JPEG2000 (Group et al.,

[1]Dept. of Comp. Sci. & Tech., BNRist Center, Tsinghua-Bosch Joint ML Center, Tsinghua University; Peng Cheng Laboratory [2]Gaoling School of AI, Renmin University of China; Beijing Key Lab of Big Data Management & Analysis Methods, Beijing, China. Correspondence to: Jianfei Chen <jianfeic@tsinghua.edu.cn>, Jun Zhu <dcszj@tsinghua.edu.cn>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

2000), PNG (Boutell & Lane, 1997) and ELIF (Sneyers & Wuille, 2016).

Despite the high compression rates, existing neural compressors still suffer from a low coding bandwidth, which has hindered practical applications. Figure 1 shows the encoding latency of several NFs-based neural compressors on the ImageNet32 dataset, of which Integer Discrete Flows (IDF) (Hoogeboom et al., 2019) is the fastest. However, the coding bandwidth of IDF is still more than an order of magnitude slower than traditional image codecs such as JPEG2000.

Technically, by viewing both image data and latent variables in a discrete integer space, IDF designs a bijective mapping between them for exact likelihood inference. Then it utilizes the rANS (Duda, 2009; 2013) coding algorithm to encode images into bits stream. Inference and entropy encoding are two separate steps of compression with IDF. Figure 1 illustrates that inference is over ten times slower than entropy coding for IDF. Thus, we seek to improve the coding bandwidth by accelerating the inference of flow-based models.

In this work, we present integer-only discrete flow (IODF), an efficient discrete flow for neural compression. We leverage quantization methods (Jacob et al., 2018; Esser et al., 2020) to accelerate the inference. Unlike IDF, which defines discrete bijections with expensive continuous neural networks, IODF performs its basic operations with the efficient *integer* arithmetic. Furthermore, IODF is equipped with learnable binary gates to identify and prune out redundant computations during inference. Our experiments demonstrate that IODF achieves up to $10\times$ inference speedup compared to IDF. In summary, our contributions include:

- We propose an efficient integer-only neural architecture for discrete flows. The architecture is carefully designed to be hardware-friendly to allow fast inference. We propose an algorithm to train such integer-only architectures, where IODF achieves comparable density estimation performance with the full precision IDF.
- We propose to prune IDF with learnable integer (more specifically, binary) gates. By removing redundant filters, we reduce FLOPs of IDF from 7.2G to 3.2G with only a tiny increase in bits per dimension (bpd).
- We deploy IODF with integer-only computational kernels on a Tesla T4 GPU using the TensorRT library (NVIDIA, 2018). We show that with integer arithmetic and pruning, IODF can achieve up to $10\times$ speedup compared to IDF during inference. IODF makes a step forward towards practical application of deep generative models in data compression.

2. Related Work

Coding with DGMs Based on the *change-of-variable formula*, NFs perform exact data distribution inference by designing bijective maps between data $\mathbf{x}$ and latent representation $\mathbf{z}$. Combining with entropy coding algorithms, NFs are applied to data lossless compression. Different from IDF (Hoogeboom et al., 2019; van den Berg et al., 2020) which model $\mathbf{x}$, $\mathbf{z}$ both with discrete integers, Ho et al. (2019b) dicretize continuous variables and propose a *local bits back* (LBB) scheme for compression with a general class of NFs which model $\mathbf{x}$, $\mathbf{z}$ as continuous. Zhang et al. (2021c;b) aim to handle the problem that continuous NFs are numerically non-invertible in data compression, and they propose novel NFs with numerically invertible transformations. Although these continuous flow-based models have achieved higher compression rates than IDF, their inference and coding procedures are generally more complex and slower. Currently, hierarchical VAEs attain theoretical code lengths comparable to NFs (Maaløe et al., 2019; Ho et al., 2019a) and DPMs achieve the best code lengths (Kingma et al., 2021) (2.49 bits/dim on CIFAR10, 3.72 bits/dim on ImageNet32 and 3.40 bits/dim on ImageNet64). However, the bandwidth of the corresponding compressors is much lower than IDF ($\sim$1MB/s) and far from practical demand, e.g., Townsend et al. (2019b) compresses at $\sim$ 0.02MB/s and Kingma et al. (2021) requires expensive computation due to a large number of timesteps with a network forward in each timestep. Moreover, continuous NFs, VAEs and DPMs based compression algorithms rely on the bits back coding scheme and thus suffer from non-negligible auxiliary bits when there are only a small amount of data to compress. Meanwhile, ARMs attain theoretical code lengths similar to DPMs, but decoding is extremely slow due to their serial sampling procedure. Overall, IDF is the most efficient neural compressor at the cost of a weak compression rate loss.

Pruning and Quantization Pruning and quantization are popular methods for reducing the memory and latency of deep neural networks (DNNs). Pruning strategies generally follow a procedure that first trains a network to convergence, scores the weight parameters (usually based on l_1/l_2-norms), prunes out low-scored parameters, and finetunes the pruned networks (LeCun et al., 1990; Han et al., 2015; Li et al., 2017; Lebedev & Lempitsky, 2016; Wen et al., 2016; He et al., 2017; Frankle & Carbin, 2019). Although various pruning methods have been developed for classification models, there is little attention on pruning NFs. Quantization represents parameters and activations in low precision format instead of 32-bit floating-point numbers (Courbariaux et al., 2015; Hubara et al., 2016; Rastegari et al., 2016; Zhou et al., 2016; Jacob et al., 2018; Choi et al., 2018; Dong et al., 2019; Esser et al., 2020; Chen et al., 2020a; Van Baalen et al., 2020). While researches in low-bit neural networks mainly focus on the quantization of discriminative models, there have been recent attempts to introduce quantization techniques into generative models.

Bird et al. (2021) binarize the majority of weights and activations in deep hierarchical VAE and NFs while retaining a valid probabilistic model. However, binarizing weights and activations leads to significant performance degradation Moreover, they use *simulated quantization* and all operations are still performed in floating points, which can not improve inference speed in reality. Ballé et al. (2018) explores integer networks and quantization in generative models for lossy compression, aiming to address the problem that floating-point arithmetic are not deterministic across different platforms.

Briefly, current DGMs based neural compressors still suffer from high inference latency, among which IDF is the most time efficient. On the other hand, pruning and quantization techniques for NFs need further exploration. Our work aims to speed up inference of IDF with integer computations and pruning out redundant computations.

3. Background: Integer Discrete Flows

Normalizing flows (NFs) provide a general framework for learning probability distributions over continuous and discrete random variables. In the discrete case, NFs consider $\mathbf{x}$ to be a discrete random variable with unknown distribution $p_X(\mathbf{x})$. Then NFs construct an invertible transformation $f : \mathcal{X} \mapsto \mathcal{Z}$, mapping $\mathbf{x}$ to latent representation $\mathbf{z} = f(\mathbf{x})$ on which we impose a tractable density $p_Z(\mathbf{z})$. Then the density of $\mathbf{x}$ can be obtained by the change-of-variables formula:

$$p_X(\mathbf{x}) = \sum_{z \in \{f^{-1}(z) = x\}} p_Z(\mathbf{z}).$$

Consider that $f : \mathcal{X} \mapsto \mathcal{Z}$ is invertible, the summation set contains only $\mathbf{z} = f(\mathbf{x})$, so the probability mass of $\mathbf{x}$ is given by

$$p_X(\mathbf{x}) = p_Z(\mathbf{z}).$$

IDF (Hoogeboom et al., 2019) assumes that both $\mathbf{x}$ and $\mathbf{z}$ lie in the $d-$dimensional integer space so $\mathcal{X} = \mathcal{Z} = \mathbb{Z}^d$, and the prior distribution $p_Z(\mathbf{z})$ is chosen as factorized discrete logistic distribution in the form

$$p_Z(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{s}) = \prod_{i=1}^{\forall} \sigma\left(\frac{z_i + \frac{1}{2} - \mu_i}{s_i}\right) - \sigma\left(\frac{z_i - \frac{1}{2} - \mu_i}{s_i}\right) \quad ,$$

where $\sigma(\cdot)$ denotes the sigmoid function. To obtain an invertible function, IDF designs its basic building block as an additive coupling layer (Dinh et al., 2017):

$$\begin{matrix} \mathbf{z}_a \\ \mathbf{z}_b \end{matrix} = \begin{matrix} \mathbf{x}_a \\ \mathbf{x}_b + \lfloor t_\theta(\mathbf{x}_a) \rceil \end{matrix} \quad . \tag{2}$$

Here $\mathbf{x}$ is split into two parts $\mathbf{x}_a \in \mathbb{Z}^m, \mathbf{x}_b \in \mathbb{Z}^n$ with $m + n = d$; likewise for $\mathbf{z}_a, \mathbf{z}_b$. $t_\theta(\cdot)$ is a neural network parameterized by θ. The network $t_\theta(\cdot) : \mathbb{R}^m \to \mathbb{R}^n$ defines a continuous mapping, which is projected to the discrete domain by the rounding operator $\lfloor \cdot \rceil$. We defer the optimization of neural networks with the rounding operator to Sec. 4.2.

IDF defines a discrete invertible mapping with a continuous function $t_\theta(\cdot)$, but the network $t_\theta(\cdot)$ still operates in the continuous domain internally. This makes IDF slow since expensive float-point operations are performed within the network.

4. Methodology

To make neural compression algorithms more efficient, we propose integer-only discrete flows (IODF). IODF consists of a novel network architecture for $t_\theta(\cdot)$, where most of the computations are achieved by efficient integer operations. IODF also prunes redundant convolution filters with learnable binary gates, which are again implemented with integer operations. We discuss how to train such integer-only networks. Finally, we propose hardware-friendly optimizations to maximize the efficiency of the integer arithmetic in IODF, including a reconsideration of the backbone architecture and a carefully implemented shortcut path.

4.1. Integer-Only Residual Block

We first present the basic integer-only building block of IODF. The methodology is mostly based on existing works on neural network quantization (Jacob et al., 2018; Esser et al., 2020), but we present the details below in the normalizing flow context.

In IODF, each network $t_\theta(\cdot)$ is made of a sequence of L *integer-only residual blocks* $t_\theta(\mathbf{x}) = t_\theta^{(1)} \circ \cdots \circ t_\theta^{(L)}(\mathbf{x})$, where each block is defined as

$$t_\theta^{(l)}(\mathbf{x}) = \mathrm{ReLU}(Q(\mathbf{x}) + \mathrm{Conv}(\mathrm{ReLU}(\mathrm{Conv}(Q(\mathbf{x}))))). \tag{3}$$

Note that this ResNet-like architecture (He et al., 2016) differs from the DenseNet architecture (Huang et al., 2017) used in IDF, which we will explain in Sec. 4.3.

In the integer-only residual block, all the tensors are represented with a hybrid numerical format, where a *quantizer* Q is used to convert floating-point tensors to the hybrid format. For a real-valued tensor $\mathbf{r}$, the quantizer outputs

$$\tilde{\mathbf{r}} := Q(\mathbf{r}) = s_{\mathbf{r}} \hat{\mathbf{r}} \approx \mathbf{r}, \tag{4}$$

where $s_{\mathbf{r}}$ is a real-valued *scale* scalar and $\hat{\mathbf{r}}$ is an integer tensor. The scale captures the wide common range of the numerical values, and $\hat{\mathbf{r}}$ encodes the actual value. In IODF, $\hat{\mathbf{r}}$ consists of 8-bit signed integers in $\{-128, \ldots, +127\}$ or $\{0, \ldots, +255\}$, depending on whether the tensor is non-negative. The Conv, ReLU, and addition operations are de-

fined with this hybrid format, and they can be implemented efficiently with integer-only arithmetic. With a little abuse of notation, we still call numbers in this hybrid format 8-bit integers, though it can represent non-integer values with the scale scalar. We defer the discussion of the implementation of quantizer to Sec. 4.2.

Integer-Only Convolution The integer-only convolution is defined as $\mathbf{y} = \mathrm{Conv}(\mathbf{x}; \mathbf{W}, \mathbf{b})$, where $\mathbf{W}$ is a $C \times D \times k \times k$ convolution kernel tensor with C, D denoting the number of output / input channels, $\mathbf{b}$ is a C-dimensional bias vector, $\mathbf{x}$ is a $D \times h \times w$ input tensor, and $\mathbf{y}$ is a $C \times h' \times w'$ output tensor. $(\mathbf{y}, \mathbf{x}, \mathbf{W})$ are all integer tensors with a floating-point scale scalar, while $\mathbf{b}$ is kept in the floating-point format. The convolution is performed in the form

$$y_c = \sum_{c'=1}^{D} W_{c,c'} \sim x_{c'} + b_c, \quad c \in \{1, \dots, C\} \tag{5}$$

Here, $W_{c,c'}$ is a $k \times k$ 2-D kernel, b_c is a scalar, $x_{c'}$ is a $h \times w$ 2-D input feature map, y_c is a $h' \times w'$ 2-D output feature map, and $\sim$ denotes for 2D-convolution. In our architecture, we fix $C = D, h' = h, w' = w$ within a residual block.

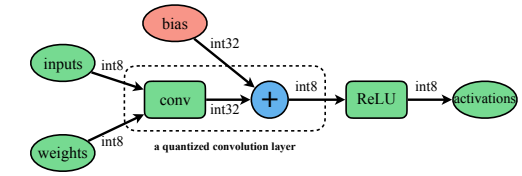
Using the hybrid format defined as Eqn. (4), we have $\mathbf{y} \approx s_{\mathbf{y}}\hat{\mathbf{y}}$, $\mathbf{x} \approx s_{\mathbf{x}}\hat{\mathbf{x}}$, $\mathbf{W} \approx s_{\mathbf{W}}\hat{\mathbf{W}}$. Plugging them into Eqn. (5) yields

$$\begin{aligned} s_{\mathbf{y}}\hat{y}_c \approx y_c &= \sum_{c'=1}^{D} W_{c,c'} \sim x_{c'} + b_c \approx \sum_{c'=1}^{D} \tilde{W}_{c,c'} \sim \tilde{x}_{c'} + b_c \\ &= \sum_{c'=1}^{D} s_{\mathbf{W}} \hat{W}_{c,c'} \sim (s_{\mathbf{x}}\hat{x}_{c'}) + b_c \\ &= s_{\mathbf{W}} s_{\mathbf{x}} \sum_{c'=1}^{D} \hat{W}_{c,c'} \sim \hat{x}_{c'} + b_c. \end{aligned}$$

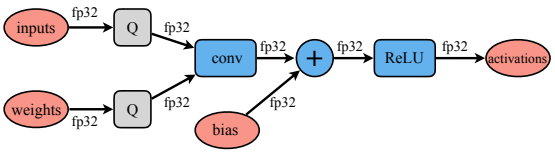
Reorganizing the terms, we have

$$\hat{y}_c \approx \frac{s_{\mathbf{W}} s_{\mathbf{x}}}{s_{\mathbf{y}}} \sum_{c'=1}^{D} \hat{W}_{c,c'} \sim \hat{x}_{c'} + \frac{b_c}{s_{\mathbf{y}}}. \tag{6}$$

Eqn. (6) can be implemented as the convolution of two signed 8-bit integer tensors $\hat{\mathbf{W}}$ and $\hat{\mathbf{x}}$, followed by element-wise floating-point multiplication and additions. This can be realized efficiently on GPUs as a single operation in the TensorRT library, as illustrated in Figure. 2(a). Matrix multiplications in the convolution are performed with the INT8 kernel, and the summation is performed by a 32-bit integer accumulator. Bias addition is also performed with 32-bit integers. The outputs are again quantized to an 8-bit integer using Eqn. 6. Because computations of a convolution layer are dominated by the multiplications in the summation, substituting floating-point operations with integer-arithmetic leads to a significant computation reduction when deploying on hardware.



(a) Integer-arithmetic inference.



(b) Fake quantization in training.

Figure 2. True quantization and fake quantization. Best viewed in color.

Integer-Only ReLU Given an input tensor $\mathbf{x} \approx s_{\mathbf{x}}\hat{\mathbf{x}}$, the ReLU is directly applied to the signed 8-bit integer part as $\hat{\mathbf{y}} = \mathrm{ReLU}(\hat{\mathbf{x}}) = \max\{0, \hat{\mathbf{x}}\}$. The scale scalar is not affected $s_{\mathbf{y}} = s_{\mathbf{x}}$. Hence, we have $\mathbf{y} \approx s_{\mathbf{y}}\hat{\mathbf{y}} = s_{\mathbf{x}} \max\{0, \hat{\mathbf{x}}\} \approx \max\{0, \mathbf{x}\} = \mathrm{ReLU}(\mathbf{x})$.

4.2. Training Integer-Only Residual Blocks

So far, we have defined the integer-only residual blocks with a general definition of the quantizer. We have not yet discussed how to train such blocks or implement the quantizer, which we shall do now. As mentioned in the last subsection, integer-only residual blocks rely on quantizers to convert real-valued tensors to integers. The conversion is lossy, which usually causes inaccurate outputs of the model. Additionally, the scale parameter in Eqn. 4 is crucial to the performance of quantized networks. This section focuses on fine-tuning the integer-only network with *simulated quantization training* (a.k.a. fake quantization) (Jacob et al., 2018).

The true integer-only inference must be deployed on hardware with special tools, which is not convenient for our training. So we implement fake quantization in PyTorch, which still uses floating-point operations but simulates the integer arithmetic by the quantizers as shown in Figure. 2(b). This corresponds to using the convolution defined in Eqn. (5).

Given a known scale s, we define the quantizer in Eqn. (4)

Table 1. Latency of floating-point and integer-only inference for convolutions with varying number of input and output channels. Obtained by averaging over 1000 runs (milliseconds).

IN CHN	OUT CHN	FP32	INT8	SPEEDUP
128	128	0.040	0.0039	10.2
64	256	0.035	0.0098	3.6
32	512	0.037	0.0131	2.8

in the specific form:

$$\tilde{\mathbf{r}} = Q(\mathbf{r}) = s\hat{\mathbf{r}} = s \cdot \left\lfloor \text{clip}\left(\frac{\mathbf{r}}{s}, -Q_N, Q_P\right) \right\rceil. \quad (7)$$

The clipping operation acts element-wise on the tensor $\mathbf{r}/s$:

$$\text{clip}\left(\frac{\mathbf{r}}{s}, -Q_N, Q_P\right) = \max\left(\min\left(\frac{\mathbf{r}}{s}, Q_P\right), Q_N\right). \quad (8)$$

We set $Q_N = -128, Q_P = 127$ for weight tensors and $Q_N = 0, Q_P = 255$ for non-negative activation tensors.

To optimize the network parameters with the quantizer, we leverage learned step-size quantization (LSQ) (Esser et al., 2020). LSQ treats the scalar s as a learnable parameter, which is updated by a gradient-based optimization algorithm. Back-propagation through quantizer is performed with the Straight Through Estimator (STE)(Bengio et al., 2013) in the form $\partial \lfloor \mathbf{u} \rceil / \partial \mathbf{u} = \mathbf{I}$ for real-valued vectors $\mathbf{u}$. Thus we have

$$\frac{\partial \mathcal{L}}{\partial \mathbf{r}} = \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{r}}} \frac{\partial \tilde{\mathbf{r}}}{\partial \mathbf{r}} = \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{r}}}, \ \mathbf{r} = \mathbf{W} \text{ or } \mathbf{x}, \quad (9)$$

where $\mathcal{L}$ denotes for the objective function and $\mathbf{r}$ applies to $\mathbf{W}$ or $\mathbf{x}$. With Eqn. (9) we can perform gradients back-propagation in IODF normally.

For scale parameters, the gradient of s can be calculated as follows,

$$\frac{\partial \tilde{\mathbf{r}}}{\partial s} = \begin{cases} (-\mathbf{r}/s + \lfloor \mathbf{r}/s \rceil) \odot \mathbb{I}(-Q_N < \mathbf{r}/s < Q_P) \\ -Q_N \cdot \mathbb{I}(\mathbf{r}/s < -Q_N) \\ Q_P \cdot \mathbb{I}(\mathbf{r}/s > Q_P) \end{cases}$$

where $\mathbb{I}(\cdot)$ is an indicator function that returns a tensor of the same shape as $\mathbf{r}$, and $\odot$ is the element-wise multiplication. Applying the chain rule, we have

$$\frac{\partial \mathcal{L}}{\partial s} = \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{r}}} \frac{\partial \tilde{\mathbf{r}}}{\partial s}$$

We adopt the gradient re-scaling trick introduced in (Esser et al., 2020), multiplying the gradient of s by a scale factor $g = 1/\sqrt{CQ_P}$, where C is the number of channels. This gradient re-scaling trick helps to stabilize the learning of the scale parameter s.

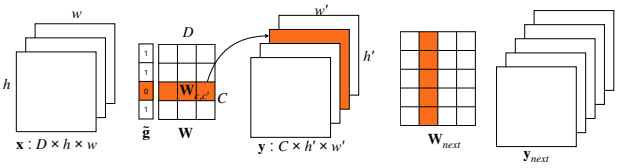


Figure 3. A gated convolution layer with C D k k kernel weights $\mathbf{W}$. The weights are represented by a C-by-D chessboard with each element as a 2-D k-by-k kernel. Each gate entry $\mathbf{g}$ determines whether to disable a filter (a row of kernels) in the kernel weights. Disabling a filter leads to removing the output feature map and corresponding kernels of the successive convolution that act on this feature map.

4.3. A More Efficient Architecture for Quantization

To make the inference of the integer-only model more efficient on hardware, we improve the network architecture. We first replace dense blocks in IDF with residual blocks for their more regular architecture and fewer connections across layers. DenseNets are more memory-intensive yet less computation-intensive (Zhang et al., 2021a) since it has many concatenation operations, which lead to many expensive quantization and dequantization operations during inference. Furthermore, DenseNets have many convolutions with a small number of input / output channels, which have unsatisfactory INT8 speedup. Tab. 1 shows the inference latency of floating-point and INT8 convolution layers for a varying number of input / output channels. All convolutions use 3×3 kernels and 16×16 input / output feature maps, so their FLOPs are kept the same. The floating-point inference latency of these three convolutions is similar, but the integer inference latency differs significantly. When the channels of input and output feature maps are identical, integer arithmetic can bring better speedup than when there is a big difference between input and output channels. Convolutions in ResNets are mainly of the former shapes, while those in DenseNets are the opposite. Therefore, residual blocks are better building blocks than dense blocks for IODF.

4.4. Learnable Binary-Gated Convolution

Neural networks have many redundant computations. Many channels die during training, and they are rarely used for inference. This problem is particularly severe for flow-based models since each transformation step has a separate network, and the required network width may vary for each transformation step. IODF addresses this problem by adding learnable binary gates to integer-only convolutions, where the masked gates can be removed at the inference time.

Formally, we denote a learnable binary gate by $\tilde{\mathbf{g}} = b(\mathbf{g}) := \mathbb{I}(\mathbf{g} > 0.5)$, where $\mathbf{g} \in [0, 1]^C$. Then a gated convolution is

defined as (omitting the bias):

$$\mathbf{y} = \mathrm{GConv}(\mathbf{x}; \mathbf{W}, \mathbf{g}) \coloneqq B(\tilde{\mathbf{g}}) \odot \mathrm{Conv}(\mathbf{x}; \mathbf{W}) = \sum_{c'=1}^{\not{K}} (\tilde{g}_c W_{c,c'}) \sim x_{c'}, \tag{10}$$

where $B(\tilde{\mathbf{g}})$ is a broadcast operation to a $C \times h' \times w'$ tensor with entries $B(\tilde{\mathbf{g}})_{c,i,j} = \tilde{g}_c, \forall i = 1, \ldots, h', j = 1, \ldots, w'$.

Figure 3 illustrates the process of pruning out a filter with a binary gate. $\tilde{g}_c = 0$ relates to disabling the filter $W_{c,:}$ and zeroing a output feature map y_c. Then the corresponding entries in the next convolution layer's weight that apply on this feature map are also removed. Therefore, pruning out m out of C filters of a convolution will reduce m/C computations for both current and the next layer.

Training Binary-Gated Convolution In the direction of obtaining good gates which contain as many zeros as possible while doing little harm to the performance of the whole model, we optimize $\mathbf{g}$ based on both the original training objective and the l_1-norm of $\tilde{\mathbf{g}}$, i.e., the number of ones in the vector $\tilde{\mathbf{g}}$. Let $\mathcal{L}_{IDF}$ be the original IDF objective function, the objective of IODF is formalized as follows,

$$\mathcal{L}(\mathbf{X};\{\mathbf{W}\},\{\mathbf{g}\}) = \mathcal{L}_{IDF}(\mathbf{X};\{\mathbf{W}\},\{\mathbf{g}\}) + \sum^{\not{X}} \lambda \|\tilde{\mathbf{g}}\|_1, \tag{11}$$

Here $\{\mathbf{W}\}, \{\mathbf{g}\}$ denote for the sets of all convolution kernel matrices and all gates vectors respectively. All gate vectors $\mathbf{g}$ are initialized as $\alpha\mathbf{1}$ with $0.5 \leq \alpha < 1$ so original gated convolution retains all filters. The $\mathcal{L}_{IDF}$ term tends to keep all entries of $\tilde{\mathbf{g}}$ close to 1 while $\|\tilde{\mathbf{g}}\|_1$ term pushes the gates to be sparse. λ acts as a strength hyper-parameter balancing them. See Appendix B.2 for settings of λ in different parts of our model. Taking derivative of $\mathcal{L}$ w.r.t. $\mathbf{g}$, we have

$$\frac{\partial \mathcal{L}}{\partial \mathbf{g}} = \frac{\partial \tilde{\mathbf{g}}}{\partial \mathbf{g}} \quad \frac{\partial \mathcal{L}_{IDF}}{\partial \tilde{\mathbf{g}}} + \lambda \frac{\partial \|\tilde{\mathbf{g}}\|_1}{\partial \tilde{\mathbf{g}}}$$

$\partial \mathcal{L}_{IDF} / \partial \tilde{\mathbf{g}}$ can be obtained by backward-propagating through the neural network. $\partial \|\tilde{\mathbf{g}}\|_1 / \partial \tilde{\mathbf{g}} = \mathbf{1}$ since $\tilde{\mathbf{g}}$ is binary. We adopt STE to make gradients flow through the binarize operation by taking $\partial \tilde{\mathbf{g}} / \partial \mathbf{g} = \mathbf{I}$. Based on the gradients $\partial \mathcal{L} / \partial \mathbf{g}$, gates can be optimized with gradient-based algorithms to achieve a good balance between efficiency and density modeling performance.

Binary-Gated Residual Blocks To prune out filters, we need to carefully consider related layers which can be influenced by the filter removal. For simple network architectures, pruning across consecutive layers is relatively straightforward, as shown in Figure. 3. However, pruning residual blocks is a little more complicated since the addition operation in Eqn. 3 requires the two operands, i.e., the shortcut and the convolution output, to have the same number of feature maps. This is not guaranteed in general since the convolution has a trainable gate. We solve this problem by performing the addition in the original unpruned feature map with a *scatter add (*SAdd*)* operation:

$$t_\theta^{(l)}(\mathbf{x}) = \mathrm{ReLU}(\mathrm{SAdd}(Q(\mathbf{x}), \mathrm{GConv}(\mathrm{ReLU}(\mathrm{GConv}(Q(\mathbf{x}))))).$$

The scatter add operation directly sums up the unpruned $Q(\mathbf{x})$ and the sparse pruned output of the gated convolution by maintaining indices representing which feature maps are removed. As a result, convolution layers within the residual blocks can be pruned arbitrarily without considering the alignment with the shortcut path.

Algorithm 1 Training IODF

Input: r_{target}, remaining target proportion of FLOPs and $\mathbf{X}$, the training dataset.
#Stage 1:
$\mathbf{W} \longleftarrow$ InitializeParameter()
Train $\mathcal{L}_{IDF}(\mathbf{X}; \{\mathbf{W}\}, \{\mathbf{1}\})$ to convergence
$F_0 \longleftarrow$ CalculateFLOPs($\mathbf{W}$)
#Stage 2:
$\mathbf{g} \longleftarrow \alpha\mathbf{1}$, $\lambda \longleftarrow$ InitializeLambda()
Train $\mathcal{L}(\mathbf{X}; \{\mathbf{W}\}, \{\mathbf{g}\})$ until CalculateFLOPs($\mathbf{W}, \mathbf{g}$)$< r_{target} F_0$
#Stage 3:
Fine-tune $\mathcal{L}_{IDF}(\mathbf{X}; \{\mathbf{W}\}, \mathbf{g})$ with fixed $\mathbf{g}$
#Stage 4:
Fine-tune the model with fake quantization applied to activations
#Stage 5:
Fine-tune the model with fake quantization applied to activations and weights

4.5. Training Workflow

We propose a 5-stage training algorithm for IODF, as shown in Alg. 1. We first train a full-precision, non-gated model to convergence by optimizing $\mathcal{L}_{IDF}(\mathbf{X}; \{\mathbf{W}\}, \{\mathbf{1}\})$. Next, we insert learnable binary gates into convolutions, set the strength hyper-parameter λ, and train the gated model by optimizing $\mathcal{L}(\mathbf{X}; \{\mathbf{W}\}, \{\mathbf{g}\})$. After removing all zeroed-out input and output filters in $\mathbf{W}$, we obtain a *pruned* model. We train the gated model until the FLOPs of the pruned model reaches our target. Then, we fine-tune the pruned model by optimizing $\mathcal{L}_{IDF}(\mathbf{X}; \{\mathbf{W}\}, \{\mathbf{g}\})$, keeping the gates fixed. The final two stages fine-tune the model to use integer-only arithmetic by incorporating fake quantization.

4.6. Deployment on Hardware

So far, IODF is still at the simulation level with quantizers mimicking the behaviors of integer arithmetic. To earn true

inference speedup effects, we must deploy IODF on specific hardware that support accelerated integer operations. As an example, NVIDIA T4 GPUs attain 16 times as many integer operations per second as floating point numbers[1]. In this work, the T4 GPU is selected due to more convenient software support, specifically, the TensorRT library (NVIDIA, 2018). Notably, we do not rely upon any unique features of T4, so IODF can be successfully deployed to other hardware and achieve the corresponding acceleration effect.

5. Experiments

To illustrate the efficiency and capacity of IODF, we conduct two sets of experiments regarding to the architecture design, filter pruning, and integer-only inference. The models are trained with PyTorch (Paszke et al., 2019) implementation and latency results are measured by deploying on a Tesla T4 GPU with the TensorRT library. Density estimation performance is reported in bits per dimension (bpd). We compare IODF with IDF on ImageNet32 and ImageNet64 (Deng et al., 2009) dataset[2]. The flow architecture is taken from IDF, which has 3 levels of flow steps at the resolution 16×16, 8×8, and 4×4 on ImageNet32, and 4 levels of flow steps at the resolution 32×32, 16×16, 8×8, and 4×4. Each resolution level has 8 additive coupling layers. Batch normalization is not used in both models. Following IDF, we adopt the rezero trick (Kingma & Dhariwal, 2018) to realize identity mapping initialization which is helpful to improving training stability. The models are trained 100 epochs for ImageNet32 and 50 epochs for ImageNet64. See Appendix B.1 for architecture and training details. Open-source code is available at `https://github.com/thu-ml/IODF`.

5.1. Network Architecture and Filter Pruning

Network Architecture IDF and IODF differ by the network architecture adopted in the additive coupling layer. We first compare the DenseNet architecture used in IDF (denoted as IDF-Dense) and the more hardware-friendly ResNet architecture discussed in Sec. 4.3 (denoted as IDF-Res). The models are used in full precision. Table 2 compares these two architectures. IDF-Res and IDF-Dense achieve similar bpd under comparable FLOPs and number of parameters, indicating that replacing DenseNet with ResNet will not sacrifice much modeling capacity. However, IDF-Res is much more efficient than IDF-Dense with integer arithmetic, as we shall see in Sec. 5.2.

[1] `https://www.nvidia.com/en-us/data-center/tesla-t4/`

[2] There are two different versions of ImageNet32 and ImageNet64 datasets. We use the down-sampled ImageNet datasets from `https://image-net.org/data/downsample/Imagenet32_train.zip`, following Grcić et al. (2021); Hazami et al. (2022). Hoogeboom et al. (2019) use the datasets downloaded from `http://image-net.org/small/train_32x32.tar`.

Table 2. Overall evaluation results on test datasets (measured in bits per dimension) of IDF-DenseNets, IDF-ResNets, and pruned models of different FLOPs pruning ratio on ImageNet32 and ImageNet64. FLOPs are measured in floating-point operations.

MODEL	#FLOPS	#PARAMETERS	BPD
IMAGENET32			
IDF-DENSE	6.43G	58.4M	3.890
IDF-RES	7.15G	62.2M	3.916
IDF-RES-PRUNED1	5.67G	38.2M	3.916
IDF-RES-PRUNED2	4.25G	23.5M	3.920
IDF-RES-PRUNED3	3.28G	17.0M	3.930
IDF-RES-PRUNED4	1.60G	7.5M	4.048
IMAGENET64			
IDF-DENSE	26.27G	84.3M	3.629
IDF-RES	29.09G	84.5M	3.630
IDF-RES-PRUNED1	23.18G	39.1M	3.642
IDF-RES-PRUNED2	17.27G	26.3M	3.665
IDF-RES-PRUNED3	12.35G	18.6M	3.700

Filter Pruning Next, we study the effectiveness of the learned binary gates proposed in Sec. 4.4 on pruning redundant filters. We insert learnable binary gates into a well-trained IDF-Res model and perform training stage 2 and stage 3 depicted in Alg. 1 until it satisfies targeted FLOPs and prune out 0-gated filters as illustrated in Sec. 4.4. See Appendix B.2 for more details about optimization parameters. We set different targeted FLOPs reduction (20%, 40%, 60%, 80% of original IDF-Res) and obtain several pruned models IDF-Res-Pruned$\{1,2,3,4\}$. Table 2 compares the FLOPs and density estimation performance of the pruned models. Binary gated convolution can effectively reduce the number of parameters and FLOPs of IDF-Res with little harm to modeling capacity. 60% computations can be cut with only a 0.015 bpd drop on ImageNet32. For ImageNet64, pruning is relatively harder, 57.6% reduced FLOPs can cause a 0.07 bpd increase from 3.630 of IDF-Res to 3.700 of IDF-Res-Pruned3.

Figure 4 displays the distribution of the percentage of remained filters among different coupling layers in a pruned model trained on ImageNet32. Most filters that are pruned out concentrate in the third flow level of IDF-Res, even we set the regularization strength λ according to the size of feature maps, i.e., λ is larger for convolutions in the first flow level whose feature map size is 16×16 and smaller for those in the third flow level whose feature map size is 4×4. We observe this phenomenon in all the pruned models. One possible explanation is that these filters act on feature maps of very small size, which have minor functionalities

Table 3. Evaluate density estimation, compression rate, and compression latency of IDF-DenseNets, IDF-ResNets, and IODF models respectively. Likelihood and compression rate are measured in bits per dimension (raw data is 8 bits/dimension). Inference latency is measured in milliseconds per sample (lower is better). Compression bandwidth is measured in MB/s (higher is better). The last row in each part shows the speedup of IODF compared to IDF-DenseNets. * denotes a failure in deployment with TensorRT.

	Analytic bpd	Coding bpd	Inference Latency 4	8	16	32	64	Compression Bandwidth 4	8	16	32	64
ImageNet32												
IDF-Dense	3.890	3.900	8.38	5.08	4.08	*	*	0.31	0.54	0.70	*	*
IDF-Res	3.916	3.926	4.19	3.19	3.59	2.54	2.93	0.56	0.79	0.79	1.12	1.00
8bit IDF-Dense	3.911	3.921	5.38	2.90	1.74	1.20	0.99	0.46	0.86	1.21	2.18	2.67
8bit IDF-Res	3.923	3.934	2.08	1.09	0.64	0.44	0.36	0.91	1.78	2.96	4.76	5.98
Pruned IDF-Res	3.936	3.947	3.27	2.04	1.60	1.33	1.29	0.72	1.14	1.59	2.01	2.15
IODF	3.968	3.979	1.79	0.94	0.54	0.34	0.27	0.98	1.91	3.45	5.41	7.08
Speedup	-	-	4.7	5.4	7.6	*	*	3.2	3.6	4.9	*	*
ImageNet64												
IDF-Dense	3.635	3.638	18.65	15.45	13.93	*	*	0.59	0.73	0.83	*	*
IDF-Res	3.637	3.640	12.50	11.89	9.30	8.84	8.64	0.82	0.93	1.22	1.32	1.35
8bit IDF-Dense	3.663	3.666	8.98	5.57	4.35	3.67	3.34	1.02	1.66	2.32	2.83	3.11
8bit IDF-Res	3.663	3.673	3.03	2.02	1.61	1.41	1.31	1.83	3.47	4.72	5.57	6.83
Pruned IDF-Res	3.657	3.666	7.75	6.45	6.55	5.79	5.71	1.21	1.59	1.70	1.93	2.00
IODF	3.685	3.695	2.79	1.71	1.34	1.15	1.06	2.22	3.72	4.64	7.03	7.93
Speedup	-	-	6.9	9.0	10.4	*	*	3.8	5.1	5.6	*	*

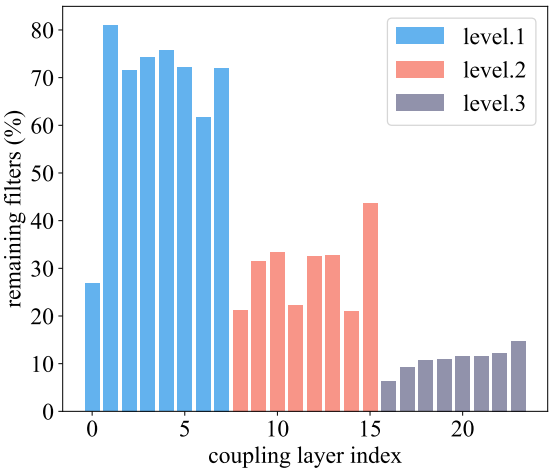


Figure 4. Number of remaining filters in different coupling layers after pruning. Layers in different flow levels are distinguished by their colors.

in density estimation. Another possible answer is that, due to the multi-scale architecture of the IDF model, part of the objective only relies on the first and second flow levels. Hence, parameters in these levels play more critical roles in density estimation.

5.2. Latency Evaluation of IODF

Now we consider the full IODF with integer-only arithmetic. We conduct a set of experiments to display density estimation performance and inference speedup of quantized models. We use unsigned per-tensor quantization for activations and signed per-channel quantization for weight tensors[3]. IODF is trained with the complete 5-stage procedure depicted in Alg. 1. The first convolution layer within each coupling transformation step is not quantized. Afterwards, we deploy these low-precision models on a Tesla T4 GPU and evaluate their inference latency. See Appendix for the detailed environment setup. For rigorous comparison, INT8 models and FP32 models are built into inference engines with the TensorRT library. We consider the following models: (1) pure FP32 IDF-Dense and IDF-Res; (2) INT8 quantized IDF-Dense and IDF-Res; (3) FP32 IDF-Res with half of the FLOPs pruned; and (4) IODF, which is quantized INT8 IDF-Res with half of the FLOPs pruned. We evaluate latency for different batch sizes.

Table 3 shows the overall results. We see that the INT8 inference of IODF is $5.9\times$ faster on ImageNet32 and $8.7\times$ faster on ImageNet64 than the baseline IDF-Dense on average. IODF achieves up to $10.4\times$ speedup with a batch size 16 on ImageNet64. Comparing IDF-Res and IDF-Dense and their INT8 versions, we see that the model architecture improvement in Sec. 4.3 is necessary for efficient inference. Pure FP32 inference of IDF-Res is faster than IDF-Dense even the former has more parameters and FLOPs. Additionally, INT8 inference of IDF-Res is on average $5.3\times$ faster than FP32 inference while only $2.3\times$ faster for IDF-Dense.

[3]Per-tensor quantization uses one scale and offset parameters for the activation tensor. Per-channel quantization has a different scale and offset for each channel of weight tensor.

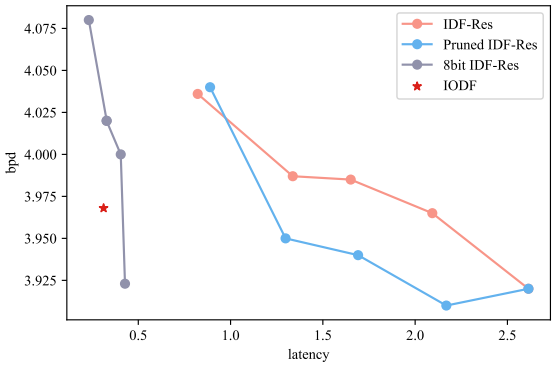


Figure 5. Bpd-latency trade-off of IDF models with a varying number of filters (IDF-Res) and their corresponding quantized models (8bit IDF-Res), different-sized models pruned from a single large IDF model (Pruned IDF-Res), and IODF. Models are trained on ImageNet32 and evaluated with batch size=32.

For larger batch sizes, inference latency per sample is lower, and the speedup effect of IODF is more remarkable. This is promising in commercial applications since real-world images are mainly high resolution. Limited by hardware, training generative models is impossible on such large images directly. Thus we train the model on smaller patches and perform encoding in a patch-based manner, which naturally gives rise to a large batch size scenario.

We also implement rANS(Duda, 2009; 2013) on the CPU to do actual encoding and report the actual size of compressed images as coding bpd. The coding BPD aligns well with the analytic bpd. Furthermore, the compression bandwidth is determined by both the inference latency and the cost of running rANS. IODF achieves 5.6× higher bandwidth than IDF-Dense. The speedup is lower than that for inference latency due to our suboptimal CPU implementation of rANS. An optimized GPU implementation of rANS should fill the gap, which we leave as future work.

Additionally, we train IDF models with a varying number of filters (IDF-Res), prune a single large model to different extents (Pruned IDF-Res), and quantize the unpruned, different-sized IDF model (8bit IDF-Res). Figure 5 presents a bpd-latency trade-off curve of these models and shows that pruned IDF-Res and 8bit IDF-Res achieve better Pareto frontier than IDF-Res, and IODF is better than both.

We also conduct experiments to evaluate memory usage, generalization ability, and practical applicability of IODF. We evaluate the models' compression performance on the high-resolution image dataset CLIC [4] ($\sim 10^6$ pixels per image) (Agustsson & Timofte, 2017) and compare the performance with non-neural compression algorithms. As shown in Table 4, the models can generalize to realistic images well. Compared to IDF, IODF improves the bandwidth by ten times and reduces memory usage by 47%, with little compression rate drop. IODF can attain better compression ratio over the traditional PNG codec (Boutell & Lane, 1997), while being about 3 times slower than the CPU-based PNG compressor implemented in the `Pillow-SIMD` package. We also compare the speed with GPU implementations of JPEG2000 codecs, where the bandwidth of CUJ2K and Fastvideo JPEG2000 encoders are 60.5MB/s and 985MB/s, respectively[5].

Table 4. Compression performance on high resolution image dataset CLIC. (Models are trained on ImageNet64 and evaluated with batch size=32). Compression rate is measured in bpd, bandwidth in MB/s, and GPU memory usage in GB.

Model	IDF-Dense	IDF-Res	8bit IDF-Res	Pruned IDF-Res	IODF	PNG
BPD	**2.438**	2.430	2.499	2.451	2.505	3.62
Bandwidth	0.84	1.28	7.57	1.95	**9.17**	29.8
Memory	3.2	2.8	1.7	2.4	**1.7**	*

6. Conclusion

We propose Integer-Only Discrete Flows (IODF), an efficient flow-based neural compressor. We propose a hardware-friendly backbone architecture with integer-only residual blocks. By equipping integer-arithmetic convolutions with learnable binary gates, we prune out redundant filters, significantly reducing the number of parameters and the amount of computation. Furthermore, we directly deploy IODF on a Tesla T4 GPU and measure the encoding latency, showing that IODF achieves up to 10× speedup compared to IDF.

Acknowledgements

This work was supported by National Key Research and Development Project of China (No. 2021ZD0110502); NSF of China Projects (Nos. 62061136001, 61620106010, 62076145, U19B2034, U1811461, U19A2081, 6197222, 62106120); Beijing NSF Project (No. JQ19016); Beijing Outstanding Young Scientist Program NO. BJJWZYJH012019100020098; a grant from Tsinghua Institute for Guo Qiang; the NVIDIA NVAIL Program with GPU/DGX Acceleration; the High Performance Computing Center, Tsinghua University; and Major Innovation & Planning Interdisciplinary Platform for the "Double-First Class" Initiative, Renmin University of China.

[4] `http://compression.cc/tasks/#image`

[5] `https://www.fastcompression.com/benchmarks/benchmarks-j2k.htm`

References

Agustsson, E. and Timofte, R. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 126–135, 2017.

Ballé, J., Johnston, N., and Minnen, D. Integer networks for data compression with latent-variable models. In *International Conference on Learning Representations*, 2018.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Bird, T., Kingma, F. H., and Barber, D. Reducing the computational cost of deep generative models with binary neural networks. In *International Conference on Learning Representations*, 2021.

Boutell, T. and Lane, T. Png (portable network graphics) specification version 1.0. *Network Working Group*, pp. 1–102, 1997.

Chen, J., Gai, Y., Yao, Z., Mahoney, M. W., and Gonzalez, J. E. A statistical framework for low-bitwidth training of deep neural networks. In *Advances in Neural Information Processing Systems*, 2020a.

Chen, J., Lu, C., Chenli, B., Zhu, J., and Tian, T. Vflow: More expressive generative flows with variational data augmentation. In *International Conference on Machine Learning*, pp. 1660–1669. PMLR, 2020b.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 248–255. IEEE, 2009.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. Density estimation using real nvp. In *International Conference on Learning Representations*, 2017.

Domke, J., Karapurkar, A., and Aloimonos, Y. Who killed the directed model? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1–8, 2008.

Dong, Z., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 293–302, 2019.

Duda, J. Asymmetric numeral systems. *arXiv preprint arXiv:0902.0271*, 2009.

Duda, J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. Learned step size quantization. In *International Conference on Learning Representations*, 2020.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

Grcić, M., Grubišić, I., and Šegvić, S. Densely connected normalizing flows. In *Advances in Neural Information Processing Systems*, volume 34, pp. 23968–23982, 2021.

Group, J. P. E. et al. Jpeg2000 image coding system. *ISO/IEC FCD 15444-1*, 2000.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. In *Advances in Neural Information Processing Systems*, 2015.

Hazami, L., Mama, R., and Thurairatnam, R. Efficient-vdvae: Less is more. *arXiv preprint arXiv:2203.13751*, 2022.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1389–1397, 2017.

Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International Conference on Machine Learning*, pp. 2722–2730. PMLR, 2019a.

Ho, J., Lohn, E., and Abbeel, P. Compression with flows via local bits-back coding. In *Advances in Neural Information Processing Systems*, 2019b.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, 2020.

Hoogeboom, E., Peters, J., van den Berg, R., and Welling, M. Integer discrete flows and lossless compression. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4700–4708, 2017.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

Kingma, D. P. and Dhariwal, P. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems*, 2018.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2013.

Kingma, D. P., Salimans, T., Poole, B., and Ho, J. Variational diffusion models. In *Advances in Neural Information Processing Systems*, 2021.

Larochelle, H. and Murray, I. The neural autoregressive distribution estimator. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 29–37. JMLR Workshop and Conference Proceedings, 2011.

Lebedev, V. and Lempitsky, V. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2554–2564, 2016.

LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pp. 598–605, 1990.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017.

Lu, C., Chen, J., Li, C., Wang, Q., and Zhu, J. Implicit normalizing flows. In *International Conference on Learning Representations*, 2021.

Maaløe, L., Fraccaro, M., Liévin, V., and Winther, O. Biva: A very deep hierarchy of latent variables for generative modeling. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

NVIDIA. Tensorrt. https://developer.nvidia.com/tensorrt, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8026–8037, 2019.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pp. 525–542. Springer, 2016.

Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pp. 1278–1286. PMLR, 2014.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. In *International Conference on Learning Representations*, 2017.

Shannon, C. E. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

Sneyers, J. and Wuille, P. Flif: Free lossless image format based on maniac compression. In *2016 IEEE international conference on image processing (ICIP)*, pp. 66–70. IEEE, 2016.

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., and Ganguli, S. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, pp. 2256–2265. PMLR, 2015.

Song, J., Meng, C., and Ermon, S. Denoising diffusion implicit models. In *International Conference on Learning Representations*, 2021.

Townsend, J., Bird, T., and Barber, D. Practical lossless compression with latent variables using bits back coding. In *International Conference on Learning Representations*, 2019a.

Townsend, J., Bird, T., Kunze, J., and Barber, D. Hilloc: Lossless image compression with hierarchical latent variable models. In *International Conference on Learning Representations*, 2019b.

Van Baalen, M., Louizos, C., Nagel, M., Amjad, R. A., Wang, Y., Blankevoort, T., and Welling, M. Bayesian bits: Unifying quantization and pruning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 5741–5752, 2020.

van den Berg, R., Gritsenko, A. A., Dehghani, M., Sønderby, C. K., and Salimans, T. Idf++: Analyzing and improving integer discrete flows for lossless compression. In *International Conference on Learning Representations*, 2020.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *Advances in Neural Information Processing Systems*, 29:2074–2082, 2016.

Zhang, C., Benz, P., Argaw, D. M., Lee, S., Kim, J., Rameau, F., Bazin, J.-C., and Kweon, I. S. Resnet or densenet? introducing dense shortcuts to resnet. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3550–3559, 2021a.

Zhang, S., Kang, N., Ryder, T., and Li, Z. iflow: Numerically invertible flows for efficient lossless compression via a uniform coder. In *Advances in Neural Information Processing Systems*, volume 34, 2021b.

Zhang, S., Zhang, C., Kang, N., and Li, Z. ivpf: Numerical invertible volume preserving flow for efficient lossless compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 620–629, 2021c.

Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

A. Asymmetric Numeral System

Asymmetric Numeral Systems (ANS) (Duda, 2009; 2013) is an approach to encoding a string of discrete symbols with a known distribution into a bit stream and decoding symbols from the bit stream. ANS is a kind of arithmetic coding algorithms, achieving approximate optimal code length, i.e. entropy of the distribution. Range-base ANS (rANS) is a variant of ANS with fast coding speed.

Let $S = (s_1, \ldots, s_n)$ be the input string of symbols with each symbol taken from the alphabet set $\mathcal{A} = \{a_1, \ldots, a_k\}$. Assume the distribution over alphabet is given by $\mathbf{p} = \{p_1, \ldots, p_k\}$ with $\sum_{i=1}^{k} p_i = 1$. Then a large integer M a chosen as total mass and integers $\{F_{a_1}, \ldots, F_{a_k}\}$ represent mass of each symbol in the alphabet, with $p_i \approx F_{a_i}/M$. Then define a cumulative mass $C_{a_i} = \sum_{j=1}^{i-1} F_{a_j}$. rANS keeps track of input symbols with a single integer state. Let X_t represent the state after rANS encodes t symbols in string S. X_0 is initialized to 0. When X_t comes, rANS update the state X_t based on X_{t-1} and s_t in the form

$$X_t = \lfloor \frac{X_{t-1}}{F_{s_t}} \rfloor * M + C_{s_t} + X_{t-1} \bmod F_{s_t}. \tag{12}$$

rANS decoder takes in a state X_t and retrieves previous state X_{t-1} and encoded symbol s_t. Consider that

$$X_t \bmod M = C_{s_t} + X_{t-1} \bmod F_{s_t} \tag{13}$$

must lies in $[C_{s_t}, C_{s_t} + F_{s_t})$. Thus the symbol s_{t+1} can be retrieved by

$$s_t = a_l \quad C_{a_l} \leq X_t \bmod M < C_{a_{l+1}}. \tag{14}$$

Then

$$X_{t-1} = \lfloor \frac{X_{t-1}}{F_{s_t}} * F_{s_t} \rfloor + X_{t-1} \bmod F_{s_t} = \lfloor \frac{X_t}{M} \rfloor * F_{s_t} + (X_t \bmod M - C_{s_t}) \tag{15}$$

B. Experimental Details

B.1. Network Architecture

The overall architecture of IODF is the same as IDF introduced in Sec. 3. The entire invertible transformation from x to z has L levels and each level is composed of D coupling layers. The neural network $t_\theta(\cdot)$ in each coupling layer consists of 8 residual blocks as Figure. 6 shows.

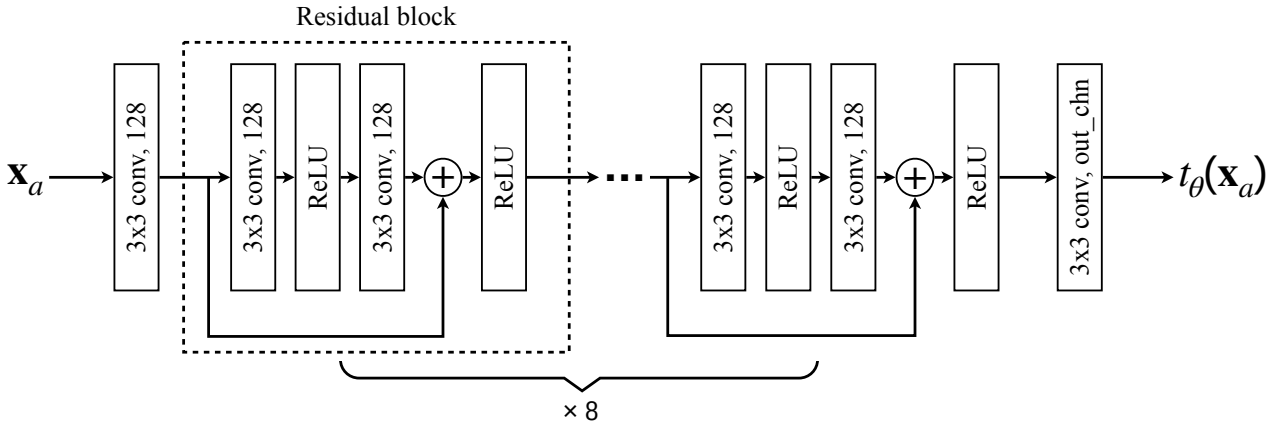


Figure 6. $t_\theta(\,)$ in each coupling transformation consist of 8 residual blocks and two convolutions. All convolutions use 3 3 kernel and 128 hidden channels. We set *padding=1* and *stride=1* in convolution for not changing the shape of feature maps.

The architecture for IDF-ResNets and optimization parameters are shown in Tab. 5.

Table 5. IDF-ResNets architecture and optimization parameters for each experiment.

DATASET	L	D	BATCHSIZE	TRAIN SAMPLES	OPTIMIZER	LR	LR DECAY	EPOCHS
IMAGENET32	3	128	512	1230000	ADAMAX	0.001	0.99	100
IMAGENET64	4	128	256	1230000	ADAMAX	0.0001	0.99	50

Table 6. We choose larger strength parameters for gated convolutions in shallower level.

DATASET	LEVEL 1	LEVEL 2	LEVEL 3	LEVEL 4
IMAGENET32	1	2	4	-
IMAGENET64	1	2	4	8

B.2. Optimization Parameters for Binary Gates and LSQ

In training model with gated convolutions, we initialize gates with $\alpha = 0.8$ and set $lr = 0.00005,\ lr\_decay = 0.99$. We set strength parameter λ in Eqn. 11 according to which layer the convolution locates in, as shown in Tab. 6. In fine-tuning pruned model, we set $lr = 0.00005,\ lr\_decay = 0.99$. Model with gated convolutions is trained for 50 epochs and pruned models are fine-tuned for 5 epochs.

For quantized model, we initialize scale parameters in quantizers data dependently in the form

$$s_{\mathbf{r}} = \frac{2^{\frac{1}{n_{\mathbf{r}}}\sum |r|}}{\sqrt{2^b - 1}},\ \ \mathbf{r} = \mathbf{W}\ or\ \mathbf{x}, \tag{16}$$

where b is the bit width and $n_{\mathbf{x}}$ denotes for the number of elements in tensor $\mathbf{x}$. We set $lr = 1e-4,\ lr\_decay = 0.99$ in simulated quantization training and quantized models are fine-tuned for 10 epochs.

B.3. Hardware and Software

The codes for our experiments are implemented with PyTorch (Paszke et al., 2019). The model implementation is based on IDF codes released by (Hoogeboom et al., 2019). rANS implementation is based on local bits back code released by (Ho et al., 2019b) in C language.

We train IODF using 8 Nvidia RTX 2080Ti GPUs. We build inference engine and evaluate the latency on a Tesla T4 GPU and Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz with TensorRT8.2.0.6, CUDA10.2.