FLOW++: IMPROVING FLOW-BASED GENERATIVE MODELS WITH VARIATIONAL DEQUANTIZATION AND ARCHITECTURE DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Flow-based generative models are powerful exact likelihood models with the benefit of efficient sampling and inference. Despite their computational efficiency, flow-based models generally have much worse density modeling performance compared to state-of-the-art autoregressive models. In this paper, we carefully investigate 3 design choices employed by prior flow-based models that turn out to be limiting: (1) uniform noise is a sub-optimal dequantization choice that hurts both training and generalization loss; (2) commonly used affine coupling flows are not expressive enough; (3) conv-net based conditioning architecture of flows fails to capture the global image context. Based on our findings, we propose Flow++, a set of alternative design choices that significantly improve the density modelling capacity of flow-based models. Our validation metrics - 3.13 bits per dim (b.p.d) on CIFAR10, 3.91 b.p.d on 32x32 Imagenet, have started to close the significant performance gap that has so far existed between autoregressive models and flow-based models.

1 INTRODUCTION

Advances in deep generative models – such as latent variable models like variational autoencoders (Kingma & Welling, 2013), implicit generative models like GANs (Goodfellow et al., 2014), and exact likelihood models like PixelRNN/CNN (van den Oord et al., 2016a;c), Image Transformer (Parmar et al., 2018), PixelSNAIL (Chen et al., 2017), NICE, RealNVP, and Glow (Dinh et al., 2014; 2016; Kingma & Dhariwal, 2018) – have enabled us to model high dimensional raw observations from complex real-world datasets, from natural images and videos, to audio signals and natural language (Karras et al., 2017; Kalchbrenner et al., 2016b; Oord et al., 2016a; Kalchbrenner et al., 2016a; Vaswani et al., 2017).

Autoregressive models, a certain subclass of exact likelihood models, have recently achieved state-of-the-art density estimation performance on many challenging real-world datasets, but suffer from slow sampling time due to their autoregressive structure (Oord et al., 2016b; Salimans et al., 2017; Chen et al., 2017; Parmar et al., 2018). Non-autoregressive flow-based models (which we will refer to as "flow models") form another subclass of exact likelihood models. These models, such as NICE, RealNVP, and Glow, are efficient for sampling, but have so far lagged behind autoregressive models in density estimation benchmarks (Dinh et al., 2014; 2016; Kingma & Dhariwal, 2018).

Within the family of exact likelihood models, it is currently impossible to have all 3 of the following attractive characteristics: (a) Fast enough generation for real-world use cases; (b) Efficient inference, likelihood evaluation at any given data point $\mathbf{x}$, for maximum-likelihood (M-projection) training; (c) Expressive modelling capacity as measured by standard density modelling benchmarks like CIFAR10 and 32x32 ImageNet. Autoregressive models achieve (b) and (c) but are very slow at generation due to the sequential dependency between dimensions; Inverse Autoregressive models (Kingma et al., 2016b) have (a) and possibly (c) (Oord et al., 2017) but can't be trained efficiently under maximum-likelihood framework; flow models enjoy efficient generation (a) and inference (b) but are lacking behind in terms of capacity.

In the hope of creating an ideal likelihood-based generative model that achieves (a,b,c), we seek to close the density modelling performance gap between flow models and autoregressive models by

first identifying the inefficiency in common flow model designs and then presenting better alternatives. In subsequent sections, we will first formally define the generic flow model family and then describe our new flow model, Flow++, which is powered by an improved training procedure for continuous likelihood models and a number of architectural extensions of the coupling layer defined by Dinh et al. (2014; 2016); Kingma & Dhariwal (2018).

2 FLOW MODELS

A flow model f is constructed as an invertible transformation that maps observed data $\mathbf{x}$ to a standard Gaussian latent variable $\mathbf{z} = f(\mathbf{x})$, as in nonlinear independent component analysis (Bell & Sejnowski, 1995; Hyvärinen et al., 2004; Hyvärinen & Pajunen, 1999). The key idea in the design of a flow model is to form f by stacking individual simple invertible transformations (Dinh et al., 2014; 2016; Kingma & Dhariwal, 2018; Rezende & Mohamed, 2015; Kingma et al., 2016b; Louizos & Welling, 2017). Explicitly, f is constructed by composing a series of invertible flows as $f(\mathbf{x}) = f_1 \circ \cdots \circ f_L(\mathbf{x})$, with each f_i having a tractable inverse and a tractable Jacobian determinant. This way, sampling is efficient, as it can be performed by computing $f^{-1}(\mathbf{z}) = f_L^{-1} \circ \cdots \circ f_1^{-1}(\mathbf{z})$ for $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and training by maximum likelihood is computationally efficient too, since

$$\log p(\mathbf{x}) = \log \mathcal{N}(f_1 \circ \cdots f_L(\mathbf{x}); \mathbf{0}, \mathbf{I}) + \sum_{i=1}^{L} \log \left| \frac{\partial f_i}{\partial f_{i-1}} \right| \tag{1}$$

is easy to compute and differentiate with respect to the parameters of the flows f_i.

3 FLOW++

In this section, we describe three main axes of modelling inefficiencies in prior flow models: (1) uniform noise is a sub-optimal dequantization choice that hurts both training loss and generalization; (2) commonly used affine coupling flows are not expressive enough; (3) conv-net based parametrization of flows fails to capture the global image context. Our proposed model, Flow++, consists of a set of improved design choices: (1) variational flow-based dequantization as opposed to uniform dequantization; (2) logistic mixture CDF coupling flows; (3) self-attention in the coupling flows.

3.1 DEQUANTIZATION VIA VARIATIONAL INFERENCE

Many real-world datasets, such as CIFAR10 and ImageNet, are recordings of continuous signals quantized into discrete representations. Fitting a continuous density model to discrete data, however, will produce a degenerate solution that places all probability mass on discrete datapoints (Uria et al., 2013). A common solution to this problem is to first convert the discrete data distribution into a continuous distribution via a process called "dequantization," and then model the resulting continuous distribution using the continuous density model (Uria et al., 2013; Dinh et al., 2016; Salimans et al., 2017).

3.1.1 UNIFORM DEQUANTIZATION

Dequantization is usually performed in prior work by adding uniform noise to the discrete data over the width of each discrete bin: if each of the D components of the discrete data $\mathbf{x}$ takes on values in $\{0, 1, 2, \ldots, 255\}$, then the dequantized data is given by $\mathbf{y} = \mathbf{x} + \mathbf{u}$, where $\mathbf{u}$ is drawn uniformly from $[0, 1)^D$. Theis et al. (2015) note that training a continuous density model p_{model} on uniformly dequantized data $\mathbf{y}$ can be interpreted as maximizing a lower bound on the log-likelihood for a certain discrete model P_{model} on the original discrete data $\mathbf{x}$:

$$P_{\text{model}}(\mathbf{x}) \coloneqq \int_{[0,1)^D} p_{\text{model}}(\mathbf{x} + \mathbf{u})\, d\mathbf{u} \tag{2}$$

The argument of Theis et al. (2015) proceeds as follows. Letting P_{data} denote the original distribution of discrete data and p_{data} denote the distribution of uniformly dequantized data, Jensen's

inequality implies that

$$\mathbb{E}_{\mathbf{y}\sim p_{\text{data}}}\left[\log p_{\text{model}}(\mathbf{y})\right] = \sum_{\mathbf{x}} P_{\text{data}}(\mathbf{x}) \int_{[0,1)^D} \log p_{\text{model}}(\mathbf{x}+\mathbf{u})\, d\mathbf{u} \tag{3}$$

$$\leq \sum_{\mathbf{x}} P_{\text{data}}(\mathbf{x}) \log \int_{[0,1)^D} p_{\text{model}}(\mathbf{x}+\mathbf{u})\, d\mathbf{u} \tag{4}$$

$$= \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[\log P_{\text{model}}(\mathbf{x})\right] \tag{5}$$

Consequently, maximizing the log-likelihood of the continuous model on uniformly dequantized data cannot lead to the continuous model degenerately collapsing onto the discrete data, because its objective is bounded above by the log-likelihood of a discrete model.

3.1.2 VARIATIONAL DEQUANTIZATION

While uniform dequantization successfully prevents the continuous density model p_{model} from collapsing to a degenerate mixture of point masses on discrete data, it asks p_{model} to assign uniform density to unit hypercubes $\mathbf{x} + [0,1)^D$ around the data $\mathbf{x}$. It is difficult and unnatural for smooth function approximators, such as neural network density models, to excel at such a task. To sidestep this issue, we now introduce a new dequantization technique based on variational inference.

Again, we are interested in modeling D-dimensional discrete data $\mathbf{x} \sim P_{\text{data}}$ using a continuous density model p_{model}, and we will do so by maximizing the log-likelihood of its associated discrete model $P_{\text{model}}(\mathbf{x}) \coloneqq \int_{[0,1)^D} p_{\text{model}}(\mathbf{x}+\mathbf{u})\, d\mathbf{u}$. Now, however, we introduce a dequantization noise distribution $q(\mathbf{u}|\mathbf{x})$, with support over $\mathbf{u} \in [0,1)^D$. Treating q as an approximate posterior, we have the following variational lower bound, which holds for all q:

$$\mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[\log P_{\text{model}}(\mathbf{x})\right] = \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[\log \int_{[0,1)^D} q(\mathbf{u}|\mathbf{x}) \frac{p_{\text{model}}(\mathbf{x}+\mathbf{u})}{q(\mathbf{u}|\mathbf{x})}\, d\mathbf{u}\right] \tag{6}$$

$$\geq \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[\int_{[0,1)^D} q(\mathbf{u}|\mathbf{x}) \log \frac{p_{\text{model}}(\mathbf{x}+\mathbf{u})}{q(\mathbf{u}|\mathbf{x})}\, d\mathbf{u}\right] \tag{7}$$

$$= \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}} \mathbb{E}_{\mathbf{u}\sim q(\cdot|\mathbf{x})}\left[\log \frac{p_{\text{model}}(\mathbf{x}+\mathbf{u})}{q(\mathbf{u}|\mathbf{x})}\right] \tag{8}$$

We will choose q itself to be a conditional flow-based generative model of the form $\mathbf{u} = q_{\mathbf{x}}(\boldsymbol{\epsilon})$, where $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) = \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \mathbf{I})$ is Gaussian noise. In this case, $q(\mathbf{u}|\mathbf{x}) = p(q_{\mathbf{x}}^{-1}(\mathbf{u})) \cdot \left|\partial q_{\mathbf{x}}^{-1}/\partial \mathbf{u}\right|$, and thus we obtain the objective

$$\mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[\log P_{\text{model}}(\mathbf{x})\right] \geq \mathbb{E}_{\mathbf{x}\sim P_{\text{data}},\boldsymbol{\epsilon}\sim p}\left[\log \frac{p_{\text{model}}(\mathbf{x}+q(\mathbf{x},\boldsymbol{\epsilon}))}{p(\boldsymbol{\epsilon})\left|\partial q_{\mathbf{x}}/\partial \boldsymbol{\epsilon}\right|^{-1}}\right] \tag{9}$$

which we maximize jointly over p_{model} and q. When p_{model} is also a flow model $\mathbf{x} = f^{-1}(\mathbf{z})$ (as it is throughout this paper), it is straightforward to calculate a stochastic gradient of this objective through pathwise derivative, as $f(\mathbf{x} + q(\mathbf{x}, \boldsymbol{\epsilon}))$ is differentiable with respect to the parameters of f and q.

Notice that the lower bound for uniform dequantization – eqs. (3) to (5) – is a special case our variational lower bound – eqs. (6) to (8) – when the dequantization distribution q is a uniform distribution that ignores dependence on $\mathbf{x}$. Because the gap between our objective (8) and the true expected log-likelihood $\mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[\log P_{\text{model}}(\mathbf{x})\right]$ is exactly $\mathbb{E}_{\mathbf{x}\sim P_{\text{data}}}\left[D_{\text{KL}}\left(q(\mathbf{u}|\mathbf{x}) \parallel p_{\text{model}}(\mathbf{u}|\mathbf{x})\right)\right]$, using a uniform q both forces the variational lower bound to stay loose due to inexpressiveness of q and forces p_{model} to unnaturally place uniform density over each hypercube $\mathbf{x} + [0,1)^D$. Using an expressive flow-based q, on the other hand, allows p_{model} to place density in each hypercube $\mathbf{x} + [0,1)^D$ according to a much more flexible distribution $q(\mathbf{u}|\mathbf{x})$. This is a more natural task for p_{model} to perform, improving both training and generalization loss.

3.2 IMPROVED COUPLING LAYERS

Recent progress in the design of flow models has involved carefully constructing flows to increase their expressiveness while preserving tractability of the inverse and Jacobian determinant computa-

tions. One example is the invertible 1×1 convolution flow, whose inverse and Jacobian determinant can be calculated and differentiated with standard automatic differentiation libraries (Kingma & Dhariwal, 2018). Another example, which we build upon in our work here, is the affine coupling layer (Dinh et al., 2016). It is a parameterized flow $\mathbf{y} = f_\theta(\mathbf{x})$ that first splits the components of $\mathbf{x}$ into two parts $\mathbf{x}_1, \mathbf{x}_2$, and then computes $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$, given by

$$\mathbf{y}_1 = \mathbf{x}_1, \qquad \mathbf{y}_2 = \mathbf{x}_2 \cdot \exp(\mathbf{a}_\theta(\mathbf{x}_1)) + \mathbf{b}_\theta(\mathbf{x}_1) \tag{10}$$

Here, $\mathbf{a}_\theta$ and $\mathbf{b}_\theta$ are outputs of a neural network that acts on $\mathbf{x}_1$ in a complex, expressive manner, but the resulting behavior on $\mathbf{x}_2$ always remains an elementwise affine transformation – effectively, $\mathbf{a}_\theta$ and $\mathbf{b}_\theta$ together form a data-parameterized family of invertible affine transformations. This allows the affine coupling layer to express complex dependencies on the data while keeping inversion and log-likelihood computation tractable. Using $\cdot$ and $\exp$ to respectively denote elementwise multiplication and exponentiation,

$$\mathbf{x}_1 = \mathbf{y}_1, \qquad \mathbf{x}_2 = (\mathbf{y}_2 - \mathbf{b}_\theta(\mathbf{y}_1)) \cdot \exp(-\mathbf{a}_\theta(\mathbf{y}_1)), \qquad \log \left| \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right| = \mathbf{1}^\top \mathbf{a}_\theta(\mathbf{x}_1) \tag{11}$$

3.2.1 Expressive coupling transformations with continuous mixture CDFs

We found in our experiments that density modeling performance of these coupling layers could be improved by augmenting the data-parameterized elementwise affine transformations by more general nonlinear elementwise transformations. For a given scalar component x of $\mathbf{x}_2$, we apply the cumulative distribution function (CDF) for a mixture of K logistics – parameterized by mixture probabilities, means, and log scales $\boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{s}$ – followed by an inverse sigmoid and an affine transformation parameterized by a and b:

$$x \longmapsto \sigma^{-1}\left(\text{MixLogCDF}(x; \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{s})\right) \cdot \exp(a) + b \tag{12}$$

$$\text{where} \quad \text{MixLogCDF}(x; \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{s}) \coloneqq \sum_{i=1}^{K} \pi_i \sigma\left((x - \mu_i) \cdot \exp(-s_i)\right) \tag{13}$$

The transformation parameters $\boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{s}, a, b$ for each component of $\mathbf{x}_2$ are produced by a neural network acting on $\mathbf{x}_1$. This neural network must produce these transformation parameters for each component of $\mathbf{x}_2$, hence it produces vectors $\mathbf{a}_\theta(\mathbf{x}_1)$ and $\mathbf{b}_\theta(\mathbf{x}_1)$ and tensors $\boldsymbol{\pi}_\theta(\mathbf{x}_1), \boldsymbol{\mu}_\theta(\mathbf{x}_1), \mathbf{s}_\theta(\mathbf{x}_1)$ (with last axis dimension K). The coupling transformation is then given by:

$$\mathbf{y}_1 = \mathbf{x}_1, \qquad \mathbf{y}_2 = \sigma^{-1}\left(\text{MixLogCDF}(\mathbf{x}_2; \boldsymbol{\pi}_\theta(\mathbf{x}_1), \boldsymbol{\mu}_\theta(\mathbf{x}_1), \mathbf{s}_\theta(\mathbf{x}_1))\right) \cdot \exp(\mathbf{a}_\theta(\mathbf{x}_1)) + \mathbf{b}_\theta(\mathbf{x}_1) \tag{14}$$

where the formula for computing $\mathbf{y}_2$ operates elementwise.

The inverse sigmoid ensures that the inverse of this coupling transformation always exists: the range of the logistic mixture CDF is $(0, 1)$, so the domain of its inverse must stay within this interval. The CDF itself can be inverted efficiently with bisection, because it is a monotonically increasing function. Moreover, the Jacobian determinant of this transformation involves calculating the probability density function of the logistic mixtures, which poses no computational difficulty.

3.2.2 Expressive conditioning architectures by introducing self-attention

In addition to improving the expressiveness of the elementwise transformations on $\mathbf{x}_2$, we found it crucial to improve the expressiveness of the conditioning on $\mathbf{x}_1$ – that is, the expressiveness of the neural network responsible for producing the elementwise transformation parameters $\boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{s}, \mathbf{a}, \mathbf{b}$. Our best results were obtained by stacking convolutions and multi-head self attention into a gated residual network (Mishra et al., 2018; Chen et al., 2017), in a manner resembling the Transformer (Vaswani et al., 2017) with pointwise feedforward layers replaced by 3×3 convolutional layers. Our architecture is defined as a stack of blocks. Each block consists of the following two layers connected in a residual fashion, with layer normalization (Ba et al., 2016) after each residual connection:

$$\text{Conv} = \text{input} \to \text{nonlinearity} \to \text{conv}_{3\times3} \to \text{nonlinearity} \to \text{gate}$$
$$\text{Attn} = \text{input} \to \text{conv}_{1\times1} \to \text{MultiHeadSelfAttention} \to \text{gate}$$

where gate refers to a 1×1 convolution that doubles the number of channels, followed by a gated linear unit (Dauphin et al., 2016). The convolutional layer is identical to the one used by PixelCNN++

(Salimans et al., 2017), and the multi-head self attention mechanism we use is identical to the one in the Transformer (Vaswani et al., 2017). (We always use 4 heads in our experiments, since we found it to be effective early on in our experimentation process.)

With these blocks in hand, the network that outputs the elementwise transformation parameters is simply given by stacking blocks on top of each other, and finishing with a final convolution that increases the number of channels to the amount needed to specify the elementwise transformation parameters.

4 EXPERIMENTS

So far we have identified 3 key areas that impact a flow model's performance: dequantization, coupling layer and conditioning architecture. In this section, we will first perform ablation experiments that confirm our improved design choice in each area independently contributes to improved performance. And then we show that Flow++, with all 3 improvements, achieve new flow model state-of-the-art density modelling performance on both CIFAR10 and 32x32 ImageNet. Finally we present example generative samples from Flow++ and compare them with autoregressive models' samples.

All experiments were run using weight normalization with data-dependent initialization (Salimans & Kingma, 2016). We also used the invertible 1 convolution flows of Kingma & Dhariwal (2018), as well as a variant of their "actnorm" flow that normalizes all activations independently (instead of normalizing per channel).

4.1 ABLATIONS

We ran the following ablations of our model on unconditional CIFAR10 density estimation: variational dequantization vs. uniform dequantization; logistic mixture coupling vs. affine coupling; and stacked self-attention vs. convolutions only.

Due to computation resource limit, we are not able to compare the performance of fully converged models of all variants. In fig. 5 and table 1, we compare the performance of these ablations relative to Flow++ at approximately 160 epochs of training, which was not enough for these models to converge, but far enough to see their relative performance differences. Switching from our variational dequantization to the more standard uniform dequantization costs the most: approximately 0.125 bits/dim. The remaining two ablations both cost approximately 0.05 bits/dim: switching from our logistic mixture coupling layers to affine coupling layers, and switching from our hybrid convolution-and-self-attention architecture to a pure convolutional residual architecture.

These experiments confirm that improvement in each axis independently contributes to better performance. The most surprising of them is probably the effect that different dequantization can have on both training and generalization loss. In fact, if we run Importance Sampling to further tighten the variational dequantization bound on our final CIFAR10 model, a modest number of importance samples (1024, compared to the 10000 used by Kingma et al. (2016b)) can reveal a gap of approximately 0.025 bits/dim, indicating that further gain might be possible given an even better dequantization proposal distribution.

Table 1: CIFAR10 ablation results after 164 epochs of training. Models not converged for the purposes of ablation study.

Ablation	**bits/dim**
uniform dequantization	3.305
no self-attention	3.233
affine coupling	3.226
Flow++ (not converged for ablation)	**3.180**

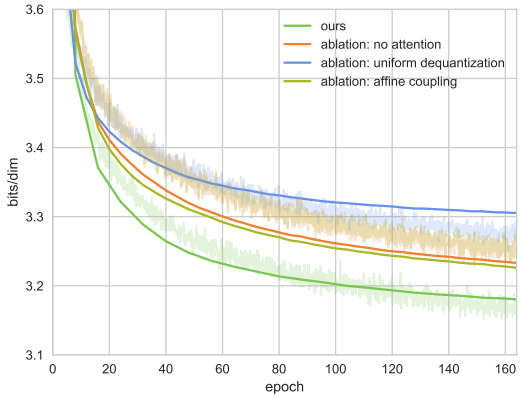


Figure 1: Ablation training (light) and validation (dark) curves on unconditional CIFAR10 density estimation. These runs are not fully converged, but the gap in density estimation performance is already visible.

Table 2: Unconditional image modeling results

Model family	**Model**	**CIFAR10 bits/dim**	**ImageNet 32x32 bits/dim**
Flows and VAEs	RealNVP (Dinh et al., 2016)	3.49	4.28
	Glow (Kingma & Dhariwal, 2018)	3.35	4.09
	Flow++ (ours)	**3.13**	**3.91**
	IAF-VAE (Kingma et al., 2016b)	3.11	–
Autoregressive	Multiscale PixelCNN (Reed et al., 2017)	–	3.95
	PixelCNN/RNN (van den Oord et al., 2016a)	3.14	3.86
	Gated PixelCNN (van den Oord et al., 2016c)	3.03	3.83
	PixelCNN++ (Salimans et al., 2017)	2.92	–
	Image Transformer (Parmar et al., 2018)	2.90	3.77
	PixelSNAIL (Chen et al., 2017)	2.85	3.80

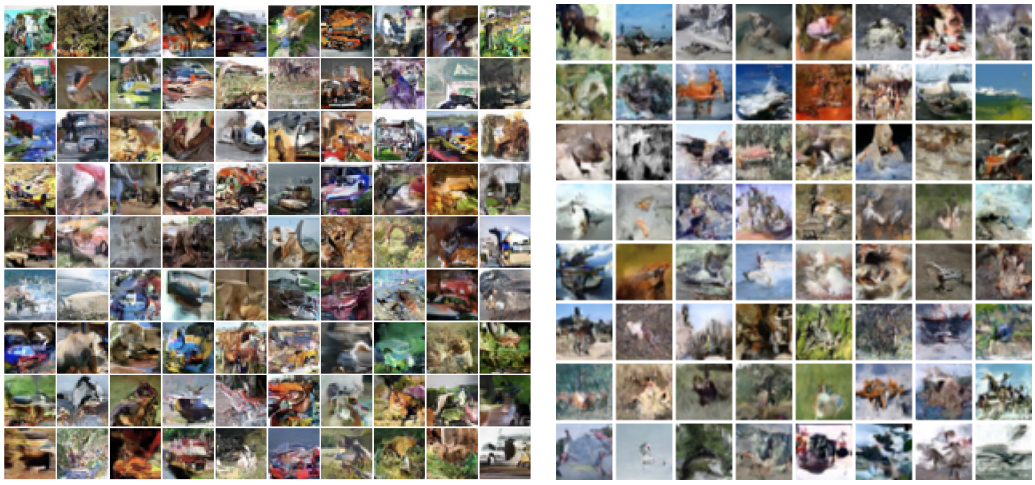
4.2 Density Modeling

Here we show in table 2 that Flow++ is able to achieve new state-of-the-art density modelling results among Flow-based models and has performance that's on-par with the first generation of PixelCNN models (van den Oord et al., 2016a). As of submission, our models have not fully converged due to computational constraint and we expect further performance gain in future revision of this manuscript.

4.3 Visualizations

We present the samples from our trained density models of Flow++ on CIFAR 10 and 32 x 32 Imagenet in Figures 2 and 3 respectively. We also import the generated samples of a PixelRNN (CNN) trained on these datasets from van den Oord et al. (2016b) (specifically, Figure 7 on page 7 in `https://arxiv.org/pdf/1601.06759.pdf`. We see that the samples generated by Flow++ on both the datasets match the perceptual quality of PixelCNN samples. This shows that the Flow++ model can capture both local and global dependencies at the level of a PixelCNN and is capable of generating diverse samples on large datasets.

To understand why this result is significant, one should take into account the sampling time comparison between PixelCNN and Flow++. See table 3. The numbers for O(N) and O(logN) PixelCNN



(a) PixelCNN (b) Flow++

Figure 2: CIFAR 10 Samples. Left: samples from a PixelCNN taken from van den Oord et al. (2016a). Right: samples from Flow++. We see that Flow++ is able to capture local dependencies and generate good samples at the quality level of PixelCNN, but with the advantage of efficient sampling: 122x more efficient than a Fast PixelCNN++ - refer to table 3 for more details.

(a) PixelCNN (b) Flow++

Figure 3: 32x32 Imagenet Samples. Left: samples generated from a PixelRNN taken from van den Oord et al. (2016a). Right: Flow++ samples. 32 x 32 Imagenet is a much larger dataset compared to CIFAR 10. Note that the samples from a flow-based generative model Flow++ can match the sample diversity of an autoregressive model like PixelRNN on such a large dataset.

are taken from the Multiscale PixelCNN paper (Reed et al., 2017) while the sampling time of Fast PixelCNN++ is taken from Ramachandran et al. (2017), which already uses advanced activations caching to improve on the original PixelCNN++'s sampling speed. Note that our density estimation metrics are better than that of Multiscale PixelCNN as shown in Table 2. As a caveat, these comparisons are not entirely accurate since the experiments corresponding to each model were carried out on different GPU configurations. However, it is well known in deep generative modeling that autoregressive models are much slower and inefficient for sampling when compared to flow-based generative models Kingma et al. (2016a); Kingma & Dhariwal (2018). With this context, it becomes

easy to appreciate the importance of why it makes sense to keep improving the sample quality and density estimation benchmarks of flow-based generative models and that Flow++ is a step in that direction.

Table 3: Sampling Time Comparisons for generating 32x32 images in parallel

Model	Wall Clock Time (secs)
Flow++	**0.324**
Fast PixelCNN++	39.58
O(N) PixelCNN	120.0
O(logN) PixelCNN	1.17
O(logN) PixelCNN in-graph	1.14

5 Related Work

Likelihood-based models constitute a large family of deep generative models. There are methods based on variational inference (Kingma & Welling, 2013; Rezende et al., 2014; Kingma et al., 2016b) that allow for efficient approximate inference and sampling, but do not admit exact inference of log likelihood computation. There are also a large class of models that admit exact log likelihood evaluation, which we call exact likelihood models in this work. These exact likelihood models are typically specified as invertible transformations that are parametrized by neural networks (Deco & Brauer, 1995; Larochelle & Murray, 2011; Uria et al., 2013; Dinh et al., 2014; Germain et al., 2015; Oord et al., 2016b; Salimans et al., 2017; Chen et al., 2017).

There are prior works that aim to improve the sampling speed of deep autoregressive models. In particular, the Multiscale PixelCNN (Reed et al., 2017) modifies the PixelCNN to be non-fully-expressive by introducing conditional independence assumptions among pixels in a way that permits sampling in a logarithmic number of steps, rather than linear. Such a change in the autoregressive structure allows faster sampling but also makes some statistical patterns impossible to capture by definition, hence reducing the capacity of the density modeling.

There are also recent works that aim to improve the the expressiveness of the coupling layers used in flow models. Glow (Kingma & Dhariwal, 2018) shows improved density estimation performance on standard benchmarks by introducing an invertible 1x1 convolution coupling layer. Kingma & Dhariwal (2018) also shows that very large flow models can be trained to produce photorealistic faces. Müller et al. (2018) introduces piecewise polynomial coupling layers that are similar in spirit to our mixture of logistics coupling layer. They found piecewise polynomial couplings to be more expressive than affine couplings, but reported little performance gains in density estimation. We leave a detailed comparison between our coupling layer and the piecewise polynomial CDFs for future work.

6 Conclusion

We presented Flow++, a set of techniques that start to close the performance gap between flow models and autoregressive models. Aside from the specific design choices proposed here, we hope the key axes of considerations identified here will help guide future research of flow models: (a) dequantization choice; (b) coupling layer design; (c) conditioning architecture. We have shown through ablation studies that improvement along any axis will help improve a flow model's final performance and we believe more performance gains are possible as the research community continues to make strides in these directions.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Anthony J Bell and Terrence J Sejnowski. An information-maximization approach to blind separation and blind deconvolution. *Neural computation*, 7(6):1129–1159, 1995.

Xi Chen, Nikhil Mishra, Mostafa Rohaninejad, and Pieter Abbeel. Pixelsnail: An improved autoregressive generative model. *arXiv preprint arXiv:1712.09763*, 2017.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*, 2016.

Gustavo Deco and Wilfried Brauer. Higher order statistical decorrelation without information loss. *Advances in Neural Information Processing Systems*, pp. 247–254, 1995.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. *arXiv preprint arXiv:1502.03509*, 2015.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Aapo Hyvärinen and Petteri Pajunen. Nonlinear independent component analysis: Existence and uniqueness results. *Neural Networks*, 12(3):429–439, 1999.

Aapo Hyvärinen, Juha Karhunen, and Erkki Oja. *Independent component analysis*, volume 46. John Wiley & Sons, 2004.

Nal Kalchbrenner, Lasse Espheholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. eural machine translation in linear time. *arXiv preprint arXiv:1610.00527*, 2016a.

Nal Kalchbrenner, Aaron van den Oord, Karen Simonyan, Ivo Danihelka, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Video pixel networks. *arXiv preprint arXiv:1610.00527*, 2016b.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

Diederik P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *arXiv preprint arXiv:1807.03039*, 2018.

Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *Proceedings of the 2nd International Conference on Learning Representations*, 2013.

Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improving variational inference with inverse autoregressive flow. In *Advances in Neural Information Processing Systems*, 2016a.

Diederik P Kingma, Tim Salimans, and Max Welling. Improving variational inference with inverse autoregressive flow. *arXiv preprint arXiv:1606.04934*, 2016b.

Hugo Larochelle and Iain Murray. *The Neural Autoregressive Distribution Estimator*. AISTATS, 2011.

Christos Louizos and Max Welling. Multiplicative normalizing flows for variational bayesian neural networks. *arXiv preprint arXiv:1703.01961*, 2017.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *International Conference on Learning Representations (ICLR)*, 2018.

Thomas Müller, Brian McWilliams, Fabrice Rousselle, Markus Gross, and Jan Novák. Neural importance sampling. *arXiv preprint arXiv:1808.03856*, 2018.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016a.

Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *International Conference on Machine Learning (ICML)*, 2016b.

Aaron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis C Cobo, Florian Stimberg, et al. Parallel wavenet: Fast high-fidelity speech synthesis. *arXiv preprint arXiv:1711.10433*, 2017.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Łukasz Kaiser, Noam Shazeer, and Alexander Ku. Image transformer. *arXiv preprint arXiv:1802.05751*, 2018.

Prajit Ramachandran, Tom Le Paine, Pooya Khorrami, Mohammad Babaeizadeh, Shiyu Chang, Yang Zhang, Mark A Hasegawa-Johnson, Roy H Campbell, and Thomas S Huang. Fast generation for convolutional autoregressive models. *arXiv preprint arXiv:1704.06001*, 2017.

Scott Reed, Aron van den Oord, Nal Kalchbrenner, Sergio Gmez Colmenarejo, Ziyu Wang, Dan Belov, and Nando de Freitas. Parallel multiscale autoregressive density estimation, 2017.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *Proceedings of The 32nd International Conference on Machine Learning*, pp. 1530–1538, 2015.

Danilo J Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *ICML*, pp. 1278–1286, 2014.

Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *arXiv preprint arXiv:1602.07868*, 2016.

Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844*, 2015.

Benigno Uria, Iain Murray, and Hugo Larochelle. Rnade: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems*, pp. 2175–2183, 2013.

Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016a.

Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2016b.

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328*, 2016c.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

7 APPENDIX A: SAMPLES

Figure 4: Samples from a Flow++ trained on CIFAR10 dataset

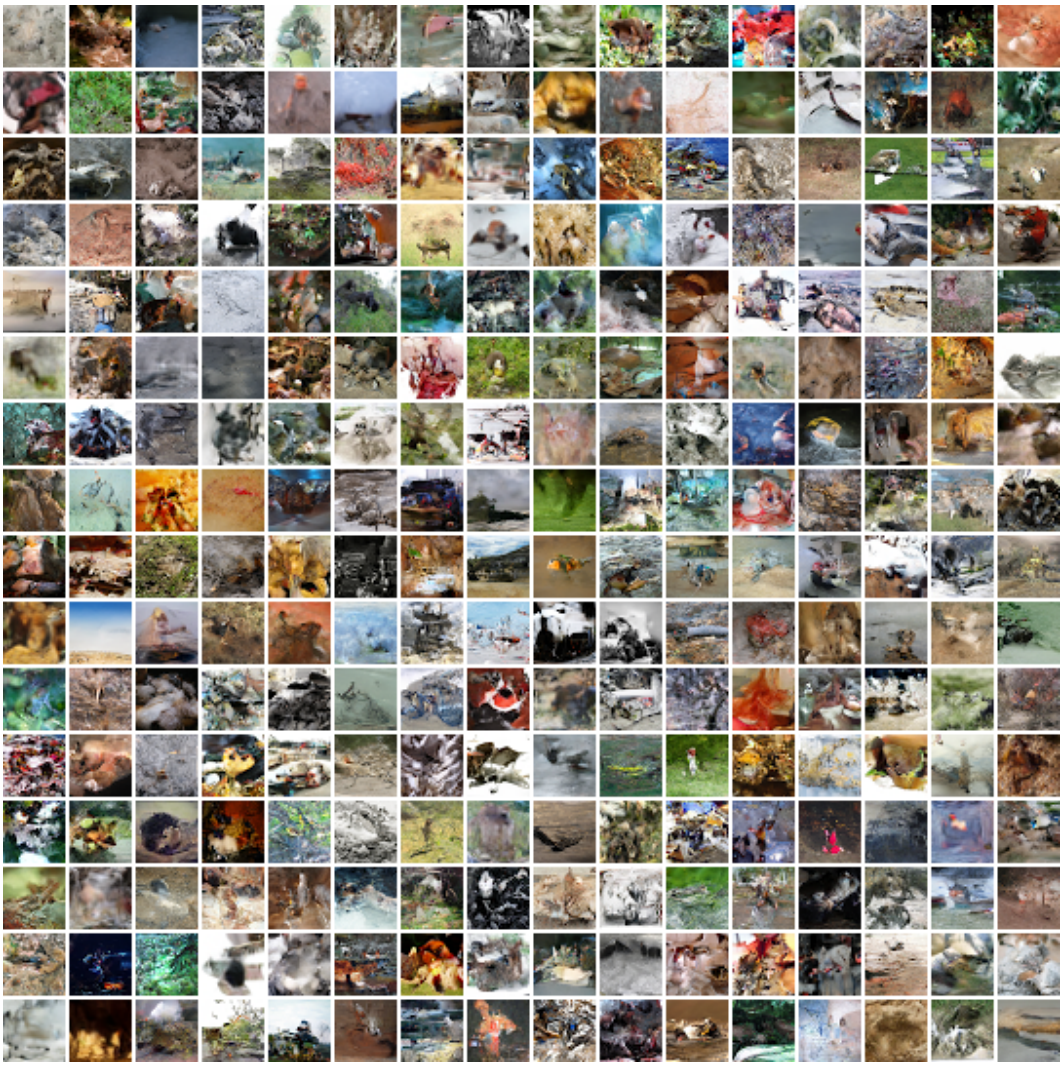


Figure 5: More Samples from a Flow++ trained on 32x32 Imagenet dataset

8 APPENDIX B: ARCHITECTURE

8.1 FLOWS

All our flows are described below (as self-contained as possible):

```
import math
import numpy as np
import tensorflow as tf
from tensorflow.contrib.framework.python.ops import arg_scope
from tqdm import tqdm

# nn module and its imports used is here are based on
    https://github.com/openai/pixel-cnn/blob/master/pixel_cnn_pp/nn.py

def safe_log(x):
   return tf.log(tf.maximum(x, 1e-22))

class tf_go:
   def __init__(self, x, debug=False):
      self.value = x
      self.debug = debug

   def __getattr__(self, name):
      def go(*args, **kwargs):
         try:
            if self.debug:
               print("Calling %s on %s" % (name, self.value))
            method = tf.__dict__[name]
            tf_val = method(*([self.value] + (list(args) if args else
                []), **kwargs)
            return tf_go(tf_val)
         except:
            print("Exception while calling %s on %s" % (name,
                self.value))
            raise

      return go

class Flow:
   def forward(self, x, **kwargs):
      raise NotImplementedError

   def inverse(self, y, **kwargs):
      raise NotImplementedError


class Inverse(Flow):
   def __init__(self, base_flow):
      self.base_flow = base_flow

   def forward(self, x, **kwargs):
      return self.base_flow.inverse(x, **kwargs)

   def inverse(self, y, **kwargs):
      return self.base_flow.forward(y, **kwargs)


class ImgProc(Flow):
   def forward(self, x, **kwargs):
      # assert ((x >= 0) & (x <= 256)).all()
      x = x * (.9 / 256) + .05 # [0, 256] -> [.05, .95]
      x = -tf.log(1. / x - 1.) # inverse sigmoid
      logd = np.log(.9 / 256) + tf.nn.softplus(x) + tf.nn.softplus(-x)
```

```
        logd = tf.reduce_sum(tf.reshape(logd, [int_shape(logd)[0], -1]), 1)
        return x, logd

    def inverse(self, y, **kwargs):
        y = tf.sigmoid(y)
        logd = tf.log(y) + tf.log(1. - y)
        y = (y - .05) / (.9 / 256) # [.05, .95] -> [0, 256]
        logd -= np.log(.9 / 256)
        logd = tf.reduce_sum(tf.reshape(logd, [int_shape(logd)[0], -1]), 1)
        return y, logd


class Normalize(Flow):
    def __init__(self, init_scale=1.):
        def f(input_, forward, init, ema):
            assert not isinstance(input_, list)
            if isinstance(input_, tuple):
                is_tuple = True
            else:
                assert isinstance(input_, tf.Tensor)
                input_ = [input_]
                is_tuple = False

            mus, invstds = [], []
            with arg_scope([nn.init_normalization], counters={}, init=init,
                ema=ema):
                for x in input_:
                    mu, invstd = nn.init_normalization(x)
                    assert mu.shape == invstd.shape == x.shape[1:]
                    mus.append(mu)
                    invstds.append(invstd * init_scale)

            logd = tf.fill([int_shape(input_[0])[0]],
                tf.add_n([tf.reduce_sum(tf.log(invstd)) for invstd in
                invstds]))
            if forward:
                out = [(x - mu[None]) * invstd[None] for (x, mu, invstd) in
                    zip(input_, mus, invstds)]
            else:
                out = [x / invstd[None] + mu[None] for (x, mu, invstd) in
                    zip(input_, mus, invstds)]
                logd = -logd

            if not is_tuple:
                assert len(out) == 1
                return out[0], logd
            return tuple(out), logd

        self.template = tf.make_template(self.__class__.__name__, f)

    def forward(self, x, init=False, ema=None, **kwargs):
        return self.template(x, forward=True, init=init, ema=ema)

    def inverse(self, y, init=False, ema=None, **kwargs):
        return self.template(y, forward=False, init=init, ema=ema)

class TupleFlip(Flow):
    def forward(self, x, **kwargs):
        assert isinstance(x, tuple)
        a, b = x
        return (b, a), None

    def inverse(self, y, **kwargs):
        assert isinstance(y, tuple)
        a, b = y
```

```
        return (b, a), None


class SpaceToDepth(Flow):
    def __init__(self, block_size=2):
        self.block_size = block_size

    def forward(self, x, **kwargs):
        return tf.space_to_depth(x, self.block_size), None

    def inverse(self, y, **kwargs):
        return tf.depth_to_space(y, self.block_size), None

class CheckerboardSplit(Flow):
    def __init__(self, bin=0, h_collapse=True):
        self.cf, self.ef, self.merge =
            self.checkerboard_condition_fn_gen(bin, h_collapse)

    def forward(self, x, **kwargs):
        return (self.cf(x), self.ef(x)), None

    def inverse(self, y, **kwargs):
        assert isinstance(y, tuple)
        cf_val, ef_val = y
        return self.merge(cf_val, ef_val), None

    @staticmethod
    def checkerboard_condition_fn_gen(bin, h_collapse):
        id = bin % 2

        def split_gen(bit):
            def go(x):
                shp = int_shape(x)
                if len(shp) != 4:
                    import IPython;
                    IPython.embed()
                assert len(shp) == 4
                half = (
                    tf_go(x).
                        transpose([0, 3, 1, 2]).
                        reshape([shp[0], shp[3], shp[1] * shp[2] // 2, 2]).
                        transpose([0, 2, 1, 3]).
                        value[:, :, :, bit]
                )
                if h_collapse:
                    return tf.reshape(
                        half,
                        [shp[0], shp[1], shp[2] // 2, shp[3]]
                    )
                else:
                    # collapse vertically
                    return (
                        tf_go(half).
                            reshape([shp[0], shp[1] // 2, 2, shp[2] // 2,
                                shp[3]]).
                            transpose([0, 1, 3, 2, 4]).
                            reshape([shp[0], shp[1] // 2, shp[2], shp[3]]).
                            value
                    )

            return go

        def merge(condition, effect):
            shp = int_shape(condition)
            assert len(shp) == 4
```

```
            xs = [condition, effect] if id == 0 else [effect, condition]
            if not h_collapse:
               xs = [
                  tf_go(x).
                     reshape([shp[0], shp[1], shp[2] // 2, 2, shp[3]]).
                     transpose([0, 1, 3, 2, 4]).
                     reshape([shp[0], shp[1] * 2, shp[2] // 2, shp[3]]).
                     value
                  for x in xs
               ]
               shp = int_shape(xs[0])
            vs = [
               tf_go(x).
                  transpose([0, 3, 1, 2]).
                  reshape([shp[0], shp[3], shp[1], shp[2], 1]).
                  value
               for x in xs
            ]
            return (
               tf_go(tf.concat(vs, axis=4)).
                  reshape([shp[0], shp[3], shp[1], shp[2] * 2]).
                  transpose([0, 2, 3, 1]).
                  value
            )

         return split_gen(id), split_gen((id + 1) % 2), merge


class Pointwise(Flow):
   def __init__(self, noisy_identity_init=None):
      def f(input_, forward, ema):
         assert not isinstance(input_, list)
         if isinstance(input_, tuple):
            is_tuple = True
         else:
            assert isinstance(input_, tf.Tensor)
            input_ = [input_]
            is_tuple = False

         out, logds = [], []
         for i, x in enumerate(input_):
            _, img_h, img_w, img_c = x.shape.as_list()
            if noisy_identity_init:
               # identity + gaussian noise
               initializer = (
                     np.eye(img_c) + noisy_identity_init *
                         np.random.randn(img_c, img_c)
               ).astype(np.float32)
            else:
               # random orthogonal
               initializer = np.linalg.qr(np.random.randn(img_c,
                   img_c))[0].astype(np.float32)
            W = nn.get_var_maybe_avg(
               'W{}'.format(i), ema, dtype=tf.float32,
               initializer=initializer,
               trainable=True
            )
            out.append(_nin(x, W if forward else tf.matrix_inverse(W)))
            logds.append(
               (1 if forward else -1) * img_h * img_w *
               tf.to_float(tf.log(tf.abs(tf.matrix_determinant(tf.to_double(W)))))
            )
         logd = tf.fill([input_[0].shape[0]], tf.add_n(logds))

         if not is_tuple:
```

```
            assert len(out) == 1
            return out[0], logd
         return tuple(out), logd

      self.template = tf.make_template(self.__class__.__name__, f)

   def forward(self, x, init=False, ema=None, **kwargs):
      return self.template(x, forward=True, ema=ema)

   def inverse(self, y, init=False, ema=None, **kwargs):
      return self.template(y, forward=False, ema=ema)


class Sigmoid(Flow):
   def forward(self, x, **kwargs):
      y = tf.sigmoid(x)
      logd = -tf.nn.softplus(x) - tf.nn.softplus(-x)
      return y, tf.reduce_sum(tf.layers.flatten(logd), axis=1)

   def inverse(self, y, **kwargs):
      x = -safe_log(tf.reciprocal(y) - 1.) # inverse sigmoid
      logd = -safe_log(y) - safe_log(1. - y)
      return x, tf.reduce_sum(tf.layers.flatten(logd), axis=1)

class Compose(Flow):
   def __init__(self, flows):
      self.flows = flows

   def _maybe_tqdm(self, iterable, desc, verbose):
      return tqdm(iterable, desc=desc) if verbose else iterable

   def forward(self, x, **kwargs):
      bs = int((x[0] if isinstance(x, tuple) else x).shape[0])
      logd_terms = []
      for i, f in enumerate(self._maybe_tqdm(self.flows, desc='forward
          {}'.format(kwargs),
                                     verbose=kwargs.get('verbose'))):
         assert isinstance(f, Flow)
         x, l = f.forward(x, **kwargs)
         if l is not None:
            assert l.shape == [bs]
            logd_terms.append(l)
      return x, tf.add_n(logd_terms) if logd_terms else tf.constant(0.)

   def inverse(self, y, **kwargs):
      bs = int((y[0] if isinstance(y, tuple) else y).shape[0])
      logd_terms = []
      for i, f in enumerate(
            self._maybe_tqdm(self.flows[::-1], desc='inverse
                {}'.format(kwargs), verbose=kwargs.get('verbose'))):
         assert isinstance(f, Flow)
         y, l = f.inverse(y, **kwargs)
         if l is not None:
            assert l.shape == [bs]
            logd_terms.append(l)
      return y, tf.add_n(logd_terms) if logd_terms else tf.constant(0.)
```

8.2 UTILS: 1 - MOL CDF AND ITS INVERSION

```
# logistic.py
import numpy as np
import tensorflow as tf
```

```
def logistic_logpdf(*, x, mean, logscale):
    """
    log density of logistic distribution
    this operates elementwise
    """
    z = (x - mean) * tf.exp(-logscale)
    return z - logscale - 2 * tf.nn.softplus(z)


def logistic_logcdf(*, x, mean, logscale):
    """
    log cdf of logistic distribution
    this operates elementwise
    """
    z = (x - mean) * tf.exp(-logscale)
    return tf.log_sigmoid(z)

def mixlogistic_logpdf(*, x, prior_logits, means, logscales):
    """logpdf of a mixture of logistics"""
    assert len(x.get_shape()) + 1 == len(prior_logits.get_shape()) ==
        len(means.get_shape()) == len(
       logscales.get_shape())
    return tf.reduce_logsumexp(
       tf.nn.log_softmax(prior_logits, axis=-1) + logistic_logpdf(
          x=tf.expand_dims(x, -1), mean=means, logscale=logscales),
       axis=-1
    )

def mixlogistic_logcdf(*, x, prior_logits, means, logscales):
    """log cumulative distribution function of a mixture of logistics"""
    assert (len(x.get_shape()) + 1 == len(prior_logits.get_shape()) ==
          len(means.get_shape()) == len(logscales.get_shape()))
    return tf.reduce_logsumexp(
       tf.nn.log_softmax(prior_logits, axis=-1) + logistic_logcdf(
          x=tf.expand_dims(x, -1), mean=means, logscale=logscales),
       axis=-1
    )

def mixlogistic_sample(*, prior_logits, means, logscales):
    # Sample mixture component
    sampled_inds = tf.argmax(
       prior_logits -
           tf.log(-tf.log(tf.random_uniform(tf.shape(prior_logits),
           minval=1e-5, maxval=1. - 1e-5))),
       axis=-1
    )
    sampled_onehot = tf.one_hot(sampled_inds, tf.shape(prior_logits)[-1])
    # Pull out the sampled mixture component
    means = tf.reduce_sum(means * sampled_onehot, axis=-1)
    logscales = tf.reduce_sum(logscales * sampled_onehot, axis=-1)
    # Sample from the component
    u = tf.random_uniform(tf.shape(means), minval=1e-5, maxval=1. - 1e-5)
    x = means + tf.exp(logscales) * (tf.log(u) - tf.log(1. - u))
    return x

def mixlogistic_invcdf(*, y, prior_logits, means, logscales, tol=1e-10,
    max_bisection_iters=100):
    """inverse cumulative distribution function of a mixture of
        logistics"""
    assert len(y.shape) + 1 == len(prior_logits.shape) ==
        len(means.shape) == len(logscales.shape)
    dtype = y.dtype
    with tf.control_dependencies([assert_in_range(y, min=0., max=1.)]):
       y = tf.identity(y)
```

```
    def body(x, lb, ub, _last_diff):
        cur_y = tf.exp(mixlogistic_logcdf(x=x, prior_logits=prior_logits,
            means=means, logscales=logscales))
        gt = tf.cast(tf.greater(cur_y, y), dtype=dtype)
        lt = 1 - gt
        new_x = gt * (x + lb) / 2. + lt * (x + ub) / 2.
        new_lb = gt * lb + lt * x
        new_ub = gt * x + lt * ub
        diff = tf.reduce_max(tf.abs(new_x - x))
        return new_x, new_lb, new_ub, diff

    init_x = tf.zeros_like(y)
    maxscales = tf.reduce_sum(tf.exp(logscales), axis=-1, keepdims=True)
        # sum of scales across mixture components
    init_lb = tf.reduce_min(means - 20 * maxscales, axis=-1)
    init_ub = tf.reduce_max(means + 20 * maxscales, axis=-1)
    init_diff = tf.constant(np.inf, dtype=dtype)

    out_x, _, _, _ = tf.while_loop(
        cond=lambda _x, _lb, _ub, last_diff: last_diff > tol,
        body=body,
        loop_vars=(init_x, init_lb, init_ub, init_diff),
        back_prop=False,
        maximum_iterations=max_bisection_iters
    )
    assert out_x.shape == y.shape
    return out_x
```

8.3 UTILS: 2 - NEURAL NETWORK ARCHITECTURAL BLOCKS

```
DEFAULT_FLOATX = tf.float32
STORAGE_FLOATX = tf.float32


def to_default_floatx(x):
    return tf.cast(x, DEFAULT_FLOATX)


def at_least_float32(x):
    assert x.dtype in [tf.float16, tf.float32, tf.float64]
    if x.dtype == tf.float16:
        return tf.cast(x, tf.float32)
    return x


def get_var(var_name, *, ema, initializer, trainable=True, **kwargs):
    """forced storage dtype"""
    assert 'dtype' not in kwargs
    if isinstance(initializer, np.ndarray):
        initializer = initializer.astype(STORAGE_FLOATX.as_numpy_dtype)
    v = tf.get_variable(var_name, dtype=STORAGE_FLOATX,
        initializer=initializer, trainable=trainable, **kwargs)
    if ema is not None:
        assert isinstance(ema, tf.train.ExponentialMovingAverage)
        v = ema.average(v)
    return v

def dense(x, *, name, num_units, init_scale=1., init, ema):
    # use weight normalization (Salimans & Kingma, 2016)
    with tf.variable_scope(name):
        assert x.shape.ndims == 2
```

```
        _V = get_var('V', shape=[int(x.shape[1]), num_units],
            initializer=tf.random_normal_initializer(0, 0.05),
                  ema=ema)
        _g = get_var('g', shape=[num_units],
            initializer=tf.constant_initializer(1.), ema=ema)
        _b = get_var('b', shape=[num_units],
            initializer=tf.constant_initializer(0.), ema=ema)
        _vinvnorm = tf.rsqrt(tf.reduce_sum(tf.square(_V), [0]))
        V, g, b, vinvnorm = map(to_default_floatx, [_V, _g, _b, _vinvnorm])
        # V, g, b = map(to_default_floatx, [_V, _g, _b])
        # vinvnorm = tf.rsqrt(tf.reduce_sum(tf.square(V), [0]))

        x0 = x = tf.matmul(x, V)
        x = (g * vinvnorm)[None, :] * x + b[None, :]

        if init: # normalize x
           m_init, v_init = tf.nn.moments(x, [0])
           scale_init = init_scale / tf.sqrt(v_init + 1e-8)
           with tf.control_dependencies([
              _g.assign(tf.cast(g * scale_init, dtype=_g.dtype)),
              _b.assign_add(tf.cast(-m_init * scale_init, dtype=_b.dtype))
           ]):
              g, b = map(to_default_floatx, [_g, _b])
              x = (g * vinvnorm)[None, :] * x0 + b[None, :]

        return x


def conv2d(x, *, name, num_units, filter_size=(3, 3), stride=(1, 1),
    pad='SAME', init_scale=1., init, ema):
   # use weight normalization (Salimans & Kingma, 2016)
   with tf.variable_scope(name):
      assert x.shape.ndims == 4
      _V = get_var('V', shape=[*filter_size, int(x.shape[-1]),
          num_units],
                initializer=tf.random_normal_initializer(0, 0.05),
                    ema=ema)
      _g = get_var('g', shape=[num_units],
          initializer=tf.constant_initializer(1.), ema=ema)
      _b = get_var('b', shape=[num_units],
          initializer=tf.constant_initializer(0.), ema=ema)
      _vnorm = tf.nn.l2_normalize(_V, [0, 1, 2])
      V, g, b, vnorm = map(to_default_floatx, [_V, _g, _b, _vnorm])

      W = g[None, None, None, :] * vnorm

      # calculate convolutional layer output
      input_x = x
      x = tf.nn.bias_add(tf.nn.conv2d(x, W, [1, *stride, 1], pad), b)

      if init: # normalize x
         m_init, v_init = tf.nn.moments(x, [0, 1, 2])
         scale_init = init_scale * tf.rsqrt(v_init + 1e-8)
         with tf.control_dependencies([
            _g.assign(tf.cast(g * scale_init, dtype=_g.dtype)),
            _b.assign_add(tf.cast(-m_init * scale_init, dtype=_b.dtype))
         ]):
            g, b = map(to_default_floatx, [_g, _b])
            W = g[None, None, None, :] * vnorm
            x = tf.nn.bias_add(tf.nn.conv2d(input_x, W, [1, *stride, 1],
                pad), b)

      return x
```

```
def nin(x, *, num_units, **kwargs):
   assert 'num_units' not in kwargs
   s = x.shape.as_list()
   x = tf.reshape(x, [np.prod(s[:-1]), s[-1]])
   x = dense(x, num_units=num_units, **kwargs)
   return tf.reshape(x, s[:-1] + [num_units])


def concat_elu(x):
   axis = len(x.get_shape()) - 1
   return tf.nn.elu(tf.concat([x, -x], axis))


def gate(x, *, axis):
   a, b = tf.split(x, 2, axis=axis)
   return a * tf.sigmoid(b)


def gated_resnet(x, *, name, a, nonlinearity=concat_elu, conv=conv2d,
    use_nin, init, ema, dropout_p):
   with tf.variable_scope(name):
      num_filters = int(x.shape[-1])

      c1 = conv(nonlinearity(x), name='c1', num_units=num_filters,
          init=init, ema=ema)
      if a is not None: # add short-cut connection if auxiliary input
          'a' is given
         c1 += nin(nonlinearity(a), name='a_proj',
             num_units=num_filters, init=init, ema=ema)
      c1 = nonlinearity(c1)
      if dropout_p > 0:
         c1 = tf.nn.dropout(c1, keep_prob=1. - dropout_p)

      c2 = (nin if use_nin else conv)(c1, name='c2',
          num_units=num_filters * 2, init_scale=0.1, init=init, ema=ema)
      return x + gate(c2, axis=3)


def attn(x, *, name, pos_emb, heads, init, ema, dropout_p):
   with tf.variable_scope(name):
      bs, height, width, ch = x.shape.as_list()
      assert pos_emb.shape == [height, width, ch]
      assert ch % heads == 0
      timesteps = height * width
      dim = ch // heads
      # Position embeddings
      c = x + pos_emb[None, :, :, :]
      # b, h, t, d == batch, num heads, num timesteps, per-head dim (C
          // heads)
      c = nin(c, name='proj1', num_units=3 * ch, init=init, ema=ema)
      assert c.shape == [bs, height, width, 3 * ch]
      # Split into heads / Q / K / V
      c = tf.reshape(c, [bs, timesteps, 3, heads, dim]) # b, t, 3, h, d
      c = tf.transpose(c, [2, 0, 3, 1, 4]) # 3, b, h, t, d
      q_bhtd, k_bhtd, v_bhtd = tf.unstack(c, axis=0)
      assert q_bhtd.shape == k_bhtd.shape == v_bhtd.shape == [bs, heads,
          timesteps, dim]
      # Attention
      w_bhtt = tf.matmul(q_bhtd, k_bhtd, transpose_b=True) /
          np.sqrt(float(dim))
      w_bhtt = tf.cast(tf.nn.softmax(at_least_float32(w_bhtt)),
          dtype=x.dtype)
      assert w_bhtt.shape == [bs, heads, timesteps, timesteps]
      a_bhtd = tf.matmul(w_bhtt, v_bhtd)
      # Merge heads
```

```
        a_bthd = tf.transpose(a_bhtd, [0, 2, 1, 3])
        assert a_bthd.shape == [bs, timesteps, heads, dim]
        a_btc = tf.reshape(a_bthd, [bs, timesteps, ch])
        # Project
        c1 = tf.reshape(a_btc, [bs, height, width, ch])
        if dropout_p > 0:
            c1 = tf.nn.dropout(c1, keep_prob=1. - dropout_p)
        c2 = nin(c1, name='proj2', num_units=ch * 2, init_scale=0.1,
            init=init, ema=ema)
        return x + gate(c2, axis=3)


def _norm(x, *, axis, g=None, b=None, e=1e-5):
    u = tf.reduce_mean(x, axis=axis, keepdims=True)
    s = tf.reduce_mean(tf.square(x - u), axis=axis, keepdims=True)
    x = (x - u) * tf.rsqrt(s + e)
    if g is not None and b is not None:
        x = x * g + b
    return x


def norm(x, *, scope, ema):
    """Layer norm over last axis"""
    with tf.variable_scope(scope):
        shape = [1] * (x.shape.ndims - 1) + [int(x.shape[-1])]
        g = get_var("g", ema=ema, shape=shape,
            initializer=tf.constant_initializer(1))
        b = get_var("b", ema=ema, shape=shape,
            initializer=tf.constant_initializer(0))
        return _norm(x, g=g, b=b, axis=-1)
```

8.4 MoL Coupling Layer with Self-Attention

```
class MixLogisticAttnCoupling(Flow):
    """
    CDF of mixture of logistics, followed by affine
    """

    def __init__(self, filters, blocks, use_nin, components, with_affine,
        use_final_nin=False, init_scale=0.1):
        self.components = components
        self.with_affine = with_affine
        self.scale_flow = Inverse(Sigmoid())

        def f(x, init, ema, dropout_p, verbose, context):
            if init and verbose:
                # debug stuff
                xmean, xvar = tf.nn.moments(x,
                    axes=list(range(len(x.get_shape()))))
                x = tf.Print(
                    x, [tf.shape(x), xmean, tf.sqrt(xvar), tf.reduce_min(x),
                        tf.reduce_max(x)],
                    message='{} (shape/mean/std/min/max)
                        '.format(self.template.variable_scope.name),
                    summarize=10,
                )
            B, H, W, C = x.shape.as_list()
            pos_emb = get_var('pos_emb', ema=ema, shape=[H, W, filters],
                              initializer=tf.random_normal_initializer(stddev=0.01),
                              trainable=True)
            x = conv2d(x, name='proj_in', num_units=filters, init=init,
                ema=ema)
            for i_block in range(blocks):
```

```
            with tf.variable_scope(f'block{i_block}'):
                x = norm(gated_resnet(x, name='conv', a=context,
                    use_nin=use_nin, init=init, ema=ema,
                                dropout_p=dropout_p), scope=f'ln1',
                                    ema=ema)
                x = norm(attn(x, name='attn', pos_emb=pos_emb, heads=4,
                    init=init, ema=ema, dropout_p=dropout_p),
                        scope=f'ln2', ema=ema)
        x = (nin if use_final_nin else conv2d)(x, name='proj_out',
            num_units=C * (2 + 3 * components),
                                    init_scale=init_scale, init=init,
                                        ema=ema)
        assert x.shape == [B, H, W, C * (2 + 3 * components)]
        x = tf.reshape(x, [B, H, W, C, 2 + 3 * components])

        s, t = tf.tanh(x[:, :, :, :, 0]), x[:, :, :, :, 1]
        ml_logits, ml_means, ml_logscales = tf.split(x[:, :, :, :, 2:],
            3, axis=4)
        ml_logscales = tf.maximum(ml_logscales, -7.)

        assert s.shape == t.shape == [B, H, W, C]
        assert ml_logits.shape == ml_means.shape == ml_logscales.shape
            == [B, H, W, C, components]
        return s, t, ml_logits, ml_means, ml_logscales

    self.template = tf.make_template(self.__class__.__name__, f)

def forward(self, x, init=False, ema=None, dropout_p=0.,
    verbose=True, context=None, **kwargs):
    assert isinstance(x, tuple)
    cf, ef = x
    s, t, ml_logits, ml_means, ml_logscales = self.template(
        cf, init=init, ema=ema, dropout_p=dropout_p, verbose=verbose,
            context=context
    )

    out = tf.exp(mixlogistic_logcdf(x=ef, prior_logits=ml_logits,
        means=ml_means, logscales=ml_logscales))
    out, scale_logd = self.scale_flow.forward(out)
    if self.with_affine:
        assert out.shape == s.shape == t.shape
        out = tf.exp(s) * out + t

    logd = mixlogistic_logpdf(x=ef, prior_logits=ml_logits,
        means=ml_means, logscales=ml_logscales)
    if self.with_affine:
        assert s.shape == logd.shape
        logd += s
    logd = tf.reduce_sum(tf.layers.flatten(logd), axis=1)
    assert scale_logd.shape == logd.shape
    logd += scale_logd

    assert out.shape == ef.shape == cf.shape
    return (cf, out), logd

def inverse(self, y, init=False, ema=None, dropout_p=0.,
    verbose=True, context=None, **kwargs):
    assert isinstance(y, tuple)
    cf, ef = y

    s, t, ml_logits, ml_means, ml_logscales = self.template(
        cf, init=init, ema=ema, dropout_p=dropout_p, verbose=verbose,
            context=context
    )
```

```
        out = ef
        if self.with_affine:
            out = tf.exp(-s) * (ef - t)
        out, invscale_logd = self.scale_flow.inverse(out)
        out = tf.clip_by_value(out, 1e-5, 1. - 1e-5)
        out = mixlogistic_invcdf(y=out, prior_logits=ml_logits,
            means=ml_means, logscales=ml_logscales)

        logd = mixlogistic_logpdf(x=out, prior_logits=ml_logits,
            means=ml_means, logscales=ml_logscales)
        if self.with_affine:
            assert s.shape == logd.shape
            logd += s
        logd = -tf.reduce_sum(tf.layers.flatten(logd), axis=1)
        assert invscale_logd.shape == logd.shape
        logd += invscale_logd

        assert out.shape == ef.shape == cf.shape
        return (cf, out), logd
```

8.4.1 FLOW-BASED DEQUANTIZATION

```
class FlowbasedDequantWithPointwise(Flow):
    def __init__(self, *, pointwise_kwargs, coupling_kwargs):
        super(FlowbasedDequantWithPointwise, self).__init__()
        self.dequant_flow = Compose([
            CheckerboardSplit(),
            Normalize(), Pointwise(**pointwise_kwargs),
                MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
            Normalize(), Pointwise(**pointwise_kwargs),
                MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
            Normalize(), Pointwise(**pointwise_kwargs),
                MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
            Normalize(), Pointwise(**pointwise_kwargs),
                MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
            Inverse(CheckerboardSplit()),
        ])

        def shallow_processor(x, init, ema, dropout_p):
            this = CheckerboardSplit.checkerboard_condition_fn_gen(0,
                True)[0](x)
            that = CheckerboardSplit.checkerboard_condition_fn_gen(0,
                True)[1](x)
            processed_context = conv2d(tf.concat([this, that], 3),
                name='proj', num_units=32, init=init, ema=ema)
            for i in range(3):
                processed_context = gated_resnet(
                    processed_context, name=f'c{i}', init=init, ema=ema,
                        dropout_p=dropout_p, use_nin=False, a=None
                ) # , filter_size=[5, 5])
            return processed_context

        self.context_proc = tf.make_template("context_proc",
            shallow_processor)

    # x: [0, 256)
    def forward(self, x, init=False, ema=None, dropout_p=0.,
        verbose=True, **kwargs):
        from fun.sandboxes.hoj.tf_gm.flows import Gaussian, safe_log
        norm_x = x / 256
        noise_dist = Gaussian(dim=int(np.prod(x.shape[1:])))
        eps, eps_logli =
            noise_dist.sample_logli(noise_dist.prior_dist_info(x.shape[0]))
```

```
        context = self.context_proc(norm_x, init, ema, dropout_p)
        unbound_xd, logd = self.dequant_flow.forward(
            tf.reshape(eps, x.shape),
            context=context,
            init=init,
            ema=ema,
            dropout_p=dropout_p,
            verbose=verbose
        )
        xd = tf.nn.sigmoid(unbound_xd)
        sigmoid_logd = safe_log(xd) + safe_log(1. - xd)
        sigmoid_logd = tf.reduce_sum(tf.reshape(sigmoid_logd,
            [sigmoid_logd.shape[0], -1]), 1)
        return x + xd, logd + sigmoid_logd - eps_logli
```

8.5 CIFAR 10 FLOW++ ARCHITECTURE:

```
def construct(*, components, with_affine, blocks):
    # see MixLogisticAttnCoupling constructor
    pointwise_kwargs = dict(noisy_identity_init=0.001)
    dequant_coupling_kwargs = dict(filters=96, blocks=2, use_nin=True,
        components=components, with_affine=with_affine)
    coupling_kwargs = dict(filters=96, blocks=blocks, use_nin=True,
        components=components, with_affine=with_affine)

    dequant_flow = FlowbasedDequantWithPointwise(
        pointwise_kwargs=pointwise_kwargs,
            coupling_kwargs=dequant_coupling_kwargs
    )
    flow = Compose([
        ImgProc(),

        CheckerboardSplit(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Inverse(CheckerboardSplit()),

        SpaceToDepth(),

        ChannelSplit(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Inverse(ChannelSplit()),

        CheckerboardSplit(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(**pointwise_kwargs),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Inverse(CheckerboardSplit()),
    ])
    return dequant_flow, flow
```

```
max_lr = 1e-3
warmup_steps = 200

def lr_schedule(step):
   if step < warmup_steps:
      return max_lr * step / warmup_steps
   return max_lr

grad_clip = 1. # grad norm clipped at 1
dropout_p = 0.2
components = 32
with_affine = True
blocks = 10

init_batchsize=128 # for weight norm
total_batchsize=64 # across 8 GPUs
ema_decay = .999 # EMA for validation params (as in PixelCNN++)
```

8.6 32x32 IMAGENET FLOW++ ARCHITECTURE:

```
def gaussian_sample_logp(shape, dtype):
   eps = tf.random_normal(shape)
   logp = Normal(0., 1.).log_prob(eps)
   assert logp.shape == eps.shape
   logp = tf.reduce_sum(tf.layers.flatten(logp), axis=1)
   return tf.cast(eps, dtype=dtype), tf.cast(logp, dtype=dtype)


class Dequantizer(Flow):
   def __init__(self, dequant_flow):
      super().__init__()
      assert isinstance(dequant_flow, Flow)
      self.dequant_flow = dequant_flow

      def shallow_processor(x, *, init, ema, dropout_p):
         this = CheckerboardSplit.checkerboard_condition_fn_gen(0,
             True)[0](x)
         that = CheckerboardSplit.checkerboard_condition_fn_gen(0,
             True)[1](x)
         processed_context = conv2d(tf.concat([this, that], 3),
             name='proj', num_units=32, init=init, ema=ema)
         for i in range(3):
            processed_context = gated_resnet(
               processed_context, name='c{}'.format(i),
               a=None, dropout_p=dropout_p, ema=ema, init=init,
               use_nin=False
            ) # TODO filter_size=[5, 5])
         return processed_context

      self.context_proc = tf.make_template("context_proc",
          shallow_processor)

   def forward(self, x, init=False, ema=None, dropout_p=0.,
       verbose=True, **kwargs):
      eps, eps_logli = gaussian_sample_logp(x.shape,
          dtype=DEFAULT_FLOATX)
      unbound_xd, logd = self.dequant_flow.forward(
         eps,
         context=self.context_proc(x / 256.0 - 0.5, init=init, ema=ema,
             dropout_p=dropout_p),
         init=init, ema=ema, dropout_p=dropout_p, verbose=verbose
```

```
        )
        xd, sigmoid_logd = Sigmoid().forward(unbound_xd)
        assert x.shape == xd.shape and logd.shape == sigmoid_logd.shape ==
            eps_logli.shape
        return x + xd, logd + sigmoid_logd - eps_logli


def construct(*, filters, blocks, components, attn_heads, use_nin,
    use_ln):
    # see MixLogisticAttnCoupling constructor
    dequant_coupling_kwargs = dict(
        filters=filters, blocks=4, use_nin=use_nin, components=components,
            attn_heads=attn_heads, use_ln=use_ln
    )
    dequant_flow = Dequantizer(Compose([
        CheckerboardSplit(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**dequant_coupling_kwargs),
            TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**dequant_coupling_kwargs),
            TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**dequant_coupling_kwargs),
            TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**dequant_coupling_kwargs),
            TupleFlip(),
        Inverse(CheckerboardSplit()),
    ]))

    coupling_kwargs = dict(
        filters=filters, blocks=blocks, use_nin=use_nin,
            components=components, attn_heads=attn_heads, use_ln=use_ln
    )
    flow = Compose([
        ImgProc(),

        CheckerboardSplit(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Inverse(CheckerboardSplit()),

        SpaceToDepth(),

        ChannelSplit(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Inverse(ChannelSplit()),

        CheckerboardSplit(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
        Normalize(), Pointwise(),
            MixLogisticAttnCoupling(**coupling_kwargs), TupleFlip(),
```

```
        Inverse(CheckerboardSplit()),
    ])
    return dequant_flow, flow

max_lr = 1e-3
min_lr = 3e-4
warmup_steps = 10000
bs = 32

def lr_schedule(step, *, decay=0.99999995, min_lr=3e-4):
    global curr_lr
    if step < warmup_steps:
        curr_lr = max_lr * step / warmup_steps
        return max_lr * step / warmup_steps
    #return max_lr
    elif step > (warmup_steps * 4) and curr_lr > min_lr:
        curr_lr *= decay
        return curr_lr
    return curr_lr

global curr_lr
curr_lr = max_lr
use_ln = True
dropout_p = 0. # Image Transformer (SOTA on Imagenet 32x32) uses 0.1
    dropout.
filters = 128
blocks = 20
components = 32 # logistic mixture components
attn_heads = 4
init_bs=128 # for weight norm
max_grad_norm=1
ema_decay=.999222
```