Pixel Adaptive Convolutions for Generative Modeling with Normalizing Flows

Anonymous Author(s)
Affiliation
Address
email

Abstract

Coupling based normalizing flows build flexible invertible functions by using half of an input to parametrize a bijective transformation on the second half. While in theory any normalizing flow layer can be used with coupling, such as invertible convolutions, the most successful coupling based flows only act elementwise on an input. This is because invertible convolutions cannot be parametrized as well as the elementwise functions can, nor can they be inverted as efficiently. We propose PAC flows, normalizing flows for images which use a new type of invertible layer based on pixel adaptive convolutions (PAC). In PAC flows, the pixel adaptive convolutions weight a convolutional filter differently at every pixel. This construction uses a comparable number of parameters as other methods and can be inverted quickly using a globally convergent iterative method. We demonstrate that PAC flows are well suited for coupling based flows, can be inverted efficiently and improve the performance of invertible convolutions. Our experiments demonstrate that PAC flows reduce the bits/dimension[1] achieved by equivalent normalizing flows with invertible standard convolutions on CIFAR-10 from 3.43 to 3.33.

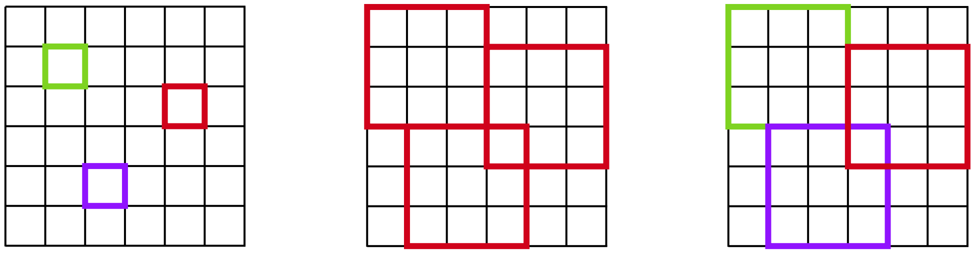
1 Introduction

Normalizing flows have been shown to produce high-quality generative models and are useful for ML tasks such as density estimation (Papamakarios et al. [2019], Chen et al. [2020], Durkan et al. [2019], Kingma and Dhariwal [2018], Dinh et al. [2017], Ho et al. [2019]). Introducing invertible convolutions in normalizing flows has the potential to further improve these generative models, given the widespread success of convolutions in other neural network architectures. However, thus far, normalizing flows with invertible convolutions have been limited in their representational capacity because transformations in normalizing flows must preserve the dimensionality of their inputs. This means that an input with C channels can only be transformed with a convolution with exactly C filters. This limits flexibility compared to standard convolutions that can increase the number of channels its input has in order to apply more filters. To address this shortcoming, we introduce pixel adaptive convolutions (PAC) in normalizing flows – which we will hence call PAC flows.

The main advantage that PAC flows has over existing invertible convolutions is the ability to learn a different filter for each location using coupling (see Fig.1). Coupling flows typically use simple invertible functions that are parametrized by one half of an input to modify the second half. Although any flow layer can be used with coupling, the most successful ones use elementwise functions (Durkan et al. [2019], Ho et al. [2019], Huang et al. [2018], Chen et al. [2020]). We hypothesize that the reason elementwise coupling outperforms existing invertible convolutions is that elementwise coupling applies a different function to each element while invertible convolutions apply the same

[1]Bits/dimension is a measure of data fit similar to negative log likelihood – the lower it is, the better the fit.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.



(a) Elementwise functions with spatially dependent parameters.

(b) Convolution with shared parameters.

(c) Convolution with spatially dependent parameters.

Figure 1: Coupling flows work well when their transformer can be adapted for every element of an input. This makes elementwise transformations (Fig.1a) a good fit. Convolutions on the other hand slide the same transformation across an input (Fig.1b). PAC flows are suited for coupling flows because they use pixel adaptive convolutions (Fig.1c) that apply a different filter to every location of the input. In the figures, each color represents a different function and the colored squares cover the pixels the function uses.

function. Ultimately the use of spatially varying convolutions allows for an increased representational capacity.

Our implementation of PAC flows is parameter efficient, tractable in the context of normalizing flows through the use of the PLU decomposition and easily inverted using a globally convergent iterative method. In our experiments we compare how elementwise affine transformations, invertible convolutions, and pixel adaptive convolutions perform under the same parameter budget. We find that the use of pixel adaptive convolutions provides a significant improvement in density estimation over comparable models. Furthermore, we demonstrate that PAC flows can be inverted quickly. In summary, our main contribution is the demonstration that pixel adaptive convolutions are well suited for coupling normalizing flows.

2 Related Work

Although there are variants of invertible convolutions, there does not exist a convolutional flow that can be parametrized as effectively as elementwise functions in coupling based normalizing flows. There are numerous elementwise coupling flows. The first popular coupling layers used affine transformations to achieve good performance on image generation tasks (Dinh et al. [2015, 2017]). Since then, more complex functions have been proposed to increase the capabilities of coupling based flows. Two popular functions are a mixture of logistic cumulative distribution functions used by Ho et al. [2019] and splines by Durkan et al. [2019]. The only condition elementwise functions must satisfy is that they need to be monotonic. Under this condition, they can always be inverted using 1d root finders such as the bisection method and the log Jacobian determinant is equal to the sum of the log derivatives. Our method is designed to get the best of elementwise coupling while using invertible convolutions and is in fact strictly more general than elementwise multiplication.

A tractable type of invertible convolutions is based on circular and symmetric convolutions. These kinds of convolutions can be computed an inverted in $O(N\log(N))$ time using an FFT and the log Jacobian determinant can be computed just as quickly. Finzi et al. use these convolutions as stand alone layers in an invertible convolutional neural network in conjunction with a smooth leaky ReLU bijection and nearest neighbor downsampling. Their method did not use coupling and showed that their architecture is not as suitable for generative modeling as existing architectures. Karami et al. [2019] also used circular convolutions, but with coupling. In particular, their conditioner network learns a single convolutional filter that is applied to an input. The authors mention that adding an elementwise multiplication after their filter results in a filtering scheme that varies over space and frequency, but this only varies the filters up to a scalar while our method uses almost entirely different filters at every spatial location.

Another kind of invertible convolution exploits triangular structure. Triangular matrices work well in normalizing flows because they can be inverted quickly using forward or backward substitution and their log Jacobian determinant is given by the sum of the log absolute values of the diagonal elements Papamakarios et al. [2019]. Zheng et al. [2018] also used autoregressive convolutions, but for 1D inputs and within a planar flow. Ma et al. [2019] uses masked convolutions to parametrize an elementwise shift and scale to transform an input rather than as an invertible convolutions.

The most similar prior work to ours is emerging convolutions (Hoogeboom et al. [2019]). Emerging convolutions compose autoregressive convolutions to yield a transformation with the same receptive field as a standard convolution. Our implementation of PAC flows can be seen as an improvement of emerging convolutions by using pixel adaptive convolutions and two other enhancements. The first is the use of locally masked convolutions (Jain et al. [2020]) which allows the use of any autoregressive order and the second is the use of the weighted Jacobi method (Saad [2003]) for inversion instead of forward/backward substitution. As we will show, these boost performance considerably.

The least constrained kind of invertible convolution is the convolution exponential (Hoogeboom et al. [2020]). This method applies the matrix exponential of a convolution to inputs using the Taylor series expansion of the matrix exponential. Furthermore, the inverse can be found as quickly as the forward pass and log Jacobian determinant is trivial to compute. The main drawback of this method is that it requires computing multiple terms of the Taylor series expansion. Nevertheless, the authors report good performance in practice. Although PAC flows could be implemented with the convolution exponential instead of emerging convolutions, we choose to use emerging convolutions for simplicity.

3 Background

3.1 Normalizing Flows

Normalizing flows (Rezende and Mohamed [2015], Papamakarios et al. [2019]) use bijective functions to transform random variables from a simple base distribution to a learnable target distribution. The probability density of a data point under the model is known in closed form through the change of variables formula. Consider a data point $x \in \mathbb{R}^N$, a bijective function f and a base density $p_z(z)$. If we compute $z = f(x)$, then the log likelihood of x under this model is given by:

$$\log p_x(x) = \log p_z(z) + \log |\frac{dz}{dx}| \tag{1}$$

To sample from this model, we sample $z \sim p_z(z)$ and compute $x = f^{-1}(z)$. In order to use a normalizing flow for density estimation and generative modeling, f must be constructed so that $f^{-1}(z)$ and $\log |\frac{df(x)}{dx}|$ are easy to compute. A simple way to construct such an f is by using coupling.

3.2 Coupling based normalizing flows

Coupling splits an input into two parts and uses the first part to parametrize a simple bijective transformation of the second part. The bijection, τ, is called the transformer and the unconstrained network that generates its parameters, θ, is called the conditioner [2] (Papamakarios et al. [2019]). Given an input x, coupling splits x into $[x_1, x_2]$ and computes $z = [x_1, \tau(x_2; \theta(x_1))]$. This can be inverted by splitting z and computing $x = [z_1, \tau^{-1}(z_2; \theta(z_1))]$ and the Jacobian determinant of the full transformation is equal to $|\frac{d\tau(x_2;\theta(x_1))}{dx_2}|$.

The ability to condition τ on x_1 is what makes coupling based methods so powerful despite often only using elementwise transformations. Although invertible convolutions exist, they currently cannot be parameterized as effectively as elementwise transformations can.

4 Method

We introduce PAC flows, an invertible pixel adaptive convolution. In this section we introduce pixel adaptive convolutions, explain how to ensure invertibility through the PLU decomposition and give a globally convergent iterative method. In Appendix B we provide an implementation of PAC flows in NumPy (Harris et al. [2020]).

[2] Papamakarios et al. [2019] uses c to denote the conditioner.

4.1 Pixel adaptive convolutions

Pixel adaptive convolutions (PAC) (Su et al. [2019]) multiply a convolutional filter, $W \in \mathbb{R}^{K_x \times K_y \times C_{\text{in}} \times C_{\text{out}}}$, with a kernel, $k \in \mathbb{R}^{H \times W \times C_{\text{out}} \times K_x \times K_y}$. This composition is then multipled with input image x. Before introducing PAC, we first define $\Psi(x) \in \mathbb{R}^{H \times W \times C_{\text{in}} \times K_x \times K_y}$ to be the operation that extracts a (K_x, K_y) patch of x at every spatial location. Intuitively, $\Psi(x)$ extracts the patches that the convolutional filter will be applied to. Matrix multiplying $\Psi(x)$ and W is equivalent to the convolution $W * x$.

$$\Psi(x)_{hwiuv} \triangleq x_{(h+u-\text{pad}_x),(w+v-\text{pad}_y),i} \tag{2}$$

$$(W * x)_{hwo} = \sum_{i,u,v} \Psi(x)_{hwiuv} W_{uvio} \tag{3}$$

With this notation, a pixel adaptive convolution is computed as:

$$\text{PAC}(W, k, x)_{hwo} = \sum_{i,u,v} \Psi(x)_{hwiuv} k_{hwouv} W_{uvio} \tag{4}$$

We construct the kernel by evaluating a squared exponential kernel, whose parameters vary for every output dimension, over feature vectors within a spatially varying patch. The algorithm takes as input $f \in \mathbb{R}^{H \times W \times F}$, $\sigma^2 \in \mathbb{R}^{H \times W \times C}$ and $l \in \mathbb{R}^{H \times W \times C}$ and computes

$$\begin{aligned} k_{hwouv} &= K(f_{hw:}, \Psi(f_{hw:uv}); \sigma^2_{hwo}, l_{hwo}) & (5)\\ &= \sigma^2_{hwo} \exp\left(-\frac{\sum_{d=0}^{F}(f_{hwd} - \Psi(f)_{hwduv})^2}{2 l_{hwo}}\right) & (6) \end{aligned}$$

F is the feature dimension set by the user. Although the kernel will always weight the center pixel of the filter the most, the model can learn to compensate for this by weighting the center pixels of W less. When PAC is used in a coupling flow, f, σ^2 and l are provided as the output of our conditioner network $\theta : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}^{H \times W \times (F+2C)}$ and W is a learned parameter.

4.2 Emerging PAC

We make pixel adaptive convolutions tractable in normalizing flows by composing two autoregressive PACs. We chain together a PAC with upper triangular structure, U, with a lower triangular PAC with a unit diagonal, L, so that the result is equivalent to using a PLU decomposition (with a fixed permutation matrix P). The log Jacobian determinant is computed as $\log|PLU| = \log \sum_i |U_{ii}|$ and can be efficiently inverted because L and U are triangular. This makes our implementation of PAC an improvement on emerging convolutions (Hoogeboom et al. [2019]). Emerging convolutions chain together two regular autoregressive convolutions instead of pixel adaptive convolutions. We use a globally convergent iterative method for inversion instead of forward/backward substitution.

The system $LUx = b$ can be solved quickly because L and U are triangular. Although we can solve $Ly = b$ and $Ux = y$ exactly using forward and backward substitution, this will take $O((HWC)^2)$ operations which can be expensive for large images. Instead, we use the weighted Jacobi method (Saad [2003]) to solve $Ly = b$ and $Ux = y$. Consider the linear system $Ax = b$ where A is triangular. The weighted Jacobi method iterates the following fixed point iteration until convergence:

$$x^{(t+1)} = x^{(t)} - \alpha \text{diag}(A)^{-1}(Ax^{(t)} - b) \tag{7}$$

For $\alpha \in (0, 2)$, Eq.7 will always converge to the solution $x = A^{-1}b$ when A is triangular. We provide a proof in Appendix A. Although the method can fail if A is ill conditioned (Huckle [2019]), we observe in our experiments that it seems to always converge after a bit of training or good initialization. The weighted Jacobi method for triangular matrices is a special the fixed point algorithm introduced by Song et al. [2019], however in the general case their algorithm is only locally convergent. In our experiments we use $\alpha = 1$.

4.3 Locally masked PAC

We ensure that a PAC is triangular using the masking scheme from Jain et al. [2020]. Locally masked convolutions provide a simple way to impose any autoregressive ordering on an input. Consider a matrix defining an order over locations, $\mathrm{O} \in \mathbb{N}_1^{H\times W}$, whose entries contain unique integers that are greater than or equal to 1. We can construct a per-location patch mask, $M \in [0,1]^{H\times W\times K_x\times K_y}$, that helps ensure upper triangular structure:

$$M_{hwuv} = \Psi(O)_{hwuv} \geq O_{hw} \tag{8}$$

M is multiplied into the patches form of a convolution in order to ensure that the convolutional filters are multiplied to every patch in an autoregressive manner. Using M and a simple upper triangular matrix $\widehat{M}_{io} = i \geq o$ over the channels, we can construct a PAC with upper triangular structure:

$$\text{Upper-PAC}(W,k,M,x)_{hwo} = \sum_{i,u,v} \underbrace{\Psi(x)_{hwiuv} k_{hwouv} W_{uvio}}_{\text{Original summand}} \; \underbrace{M_{hwuv}\widehat{M}_{io}}_{\text{Per-location patch mask}} \tag{9}$$

Upper-PAC(W,k,x) completes the $y = Ux$ multiplication in the PLU decomposition matrix-vector product. The remaining matrix-vector product $z = Ly$ requires that L is lower triangular with a unit diagonal. This is trivially achieved by logically negating M to ensure strictly lower triangular structure and adding the input for the unit diagonal:

$$\text{Lower-PAC}(W,k,M,y)_{hwo} = y + \sum_{i,u,v} \Psi(y)_{hwiuv} k_{hwouv} W_{uvio} \; \underbrace{\neg(M_{hwuv}\widehat{M}_{io})}_{\text{Logical negation of mask}} \tag{10}$$

4.4 PAC Flows

The crux of PAC flows is the introduction of invertible transformations that use emerging pixel adaptive convolutions. PAC flows use an arbitrary order over the input to construct an autoregressive per-location patch mask to impose upper or lower triangular structure. Fig.2 provides a visual summary of how PAC flows generate an output. The full algorithm is presented in Algorithm 1 and a NumPy Harris et al. [2020] implementation is given in Appendix B.

PAC flows have fewer parameters than popular elementwise coupling transformations such as neural splines Durkan et al. [2019] and logistic CDF mixtures Ho et al. [2019]. From section 4.1, we need the parameters f, σ^2 and l to contains the parameters that PAC flows need: $(f, \sigma^2, l) \in \mathbb{R}^{H\times W\times(2C+F)}$. Table 1 compares the number of parameters needed to parametrize a spatial location of PAC flows to the number needed by RealNVP, logistic CDF mixtures and neural splines. Our method uses far fewer parameters than logistic CDF mixtures and neural splines. In all of our experiments we set the kernel size to 5.

Algorithm 1 PAC Flows

Input $W, O, x, \theta(.)$

// Split the input
$(x_1, x_2) \leftarrow x$

// Compute the kernel parameters
$f, \sigma^2, l \leftarrow \theta(x_2)$

// Compute the kernel
$k \leftarrow K(f, \Psi(f); \sigma^2, l)$

// Get the mask
$M \leftarrow \Psi(O) \geq O$

// y=Ux
$y \leftarrow$ UpperPAC(W, k, M, x_1)

// z=LUx
$z_1 \leftarrow$ LowerPAC(W, k, M, y)

// Sum over the diagonal
logdet $= \sum_{hwc} \log |k_{h,w,c,p,p} W_{p,p,c,c}|$

// Combine the output
$z \leftarrow (z_1, x_2)$
return z, logdet

5 Experiments

Our experiments investigate how pixel adaptive convolutions fare against standard convolutions and linear layers. We evaluate three models on the CIFAR-10 Krizhevsky [2009] and downsampled ImageNet dataset Chrabaszcz et al. [2017] at 32x32 and 64x64 resolutions. All of our code is written using the JAX Bradbury et al. [2018] python library. Each model was trained on either one Nvidia 1080Ti or 2080Ti GPU.

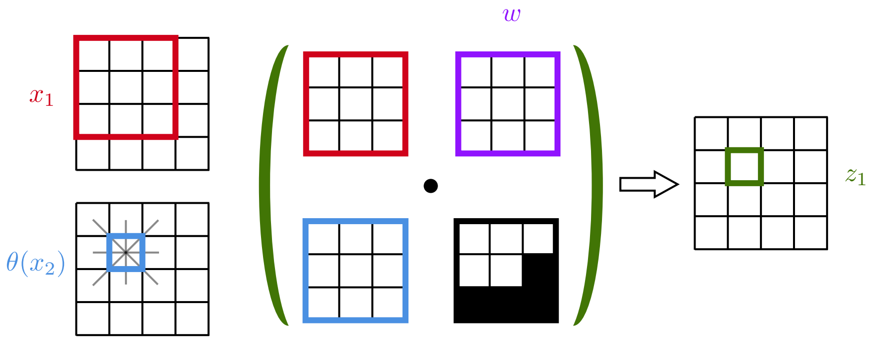


Figure 2: A visual summary of Upper-PAC from Eq.9. Each **output pixel** is computed as the dot product of an **input patch**, **location dependent filter**, **convolutional filter** and **autoregressive mask**. Lower-PAC is identical except that its mask is logically negated. PAC flows is the composition of Lower-PAC and Upper-PAC. See algorithm 1 for a summary of the algorithm and Appendix B for a Python implementation.

	RealNVP	Flow++	Neural Spline	**PAC Flow**
Parameters per pixel	2C	3KC	(3K-1)C	**2C + K**
Ratio with RealNVP (C=3,K=32)	1.0	48.0	47.5	**6.3**

Table 1: Comparison of the number of parameters used to transform feature vector at a spatial location of an image between popular elementwise flows and PAC Flow. RealNVP learns a shift/scale parameter for every element in the image, Flow++ learns K logistic cdf mixtures with 3 parameters each for every element and Neural Splines use K spline knots with up to 3 parameters each for every element. PAC flow learns a lengthscale and variance to parametrize the kernel function at every element and applies each kernel to a K dimensional feature vector for every spatial location (see section 4.1). As a result, PAC flows has minimal parameter overhead compared to lightweight flows such as RealNVP.

5.1 Architecture

Our architecture is similar to the one used in Chen et al. [2020] - we use a coupling network with variational dequantization Ho et al. [2019] and channel padding, 3 checkerboard and 3 channel coupling layers at the full spatial scale, then halve the spatial dimensions and quadruple the channel dimension and then another use another 3 checkerboard and 3 channel coupling layers. Our model differs from Chen et al. [2020] mainly in the transformer we use in the coupling layers and in our implementation of variational dequantization and padding. Below, we provide more details on each component of the network, in appendix C.1 we share the nonlinearities we used and how they helped initialization, and in appendix C.4 we have a full outline of the architecture.

Conditioner networks All of the conditioner networks are residual networks (He et al. [2016]) with 12 residual blocks that contain a gated convolution and layer normalization (Xu et al. [2019]), similar to the architecture of Ho et al. [2019]. Additionally, we increased the number of channels of the input to 32 with a 1x1 convolution before passing it to the resnet. We did this because we found that it worked well during our early experiments.

Transformer flows The transformer [3] flow that we use throughout our models is shown in Fig.3. It passes an input through a linear transformation (depending on the model type) with a learnable bias, a logistic CDF mixture (Ho et al. [2019]) nonlinearity and logit, another linear layer and then an S-Log gate (Karami et al. [2019]). We chose this architecture to resemble the architecture from Karami et al. [2019]. The addition of the S-Log gate in particular seemed to noticeably speed up training. The parameters of the transformer are generated using a conditioner network.

[3]This does not refer to the transformer model Vaswani et al. [2017] but instead the flow used in coupling (see Sec.3.2).

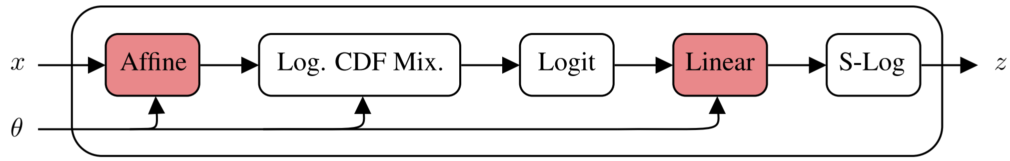


Figure 3: Architecture of the transformer flow for our experiments. The models we test in our experiments mainly differ in the affine and linear layer. The affine layer is the same as the linear layer, except with an extra elementwise addition. The RealNVP, emerging conv. and PAC flow models use elementwise multiplication, an emerging convolution + elementwise multiplication, and a pixel adaptive convolution respectively for their linear layers. We ensure each model has the same number of parameters only by altering the number of mixture components in the logistic cdf mixture layer.

Linear layers We compare three different linear transformations: PAC, a convolution followed by a elementwise multiplication, and elementwise multiplication. The experiments with elementwise multiplication are denoted with RealNVP because the RealNVP paper (Dinh et al. [2017]) used elementwise shift and scaling.

The convolutions without the pixel adaptive component are implemented exactly the same as PAC, but do not multiply in the kernel when evaluating Eq.9 and Eq.10. We refer to this model as emerging conv. because it is most similar to the method from the paper Hoogeboom et al. [2019]. We also add an elementwise multiplication after the convolution to compare against the combined convolutional flow from Sec.(3.3) of Karami et al. [2019] which claims that elementwise multiplication can be used to achieve a location dependent filtering scheme. We note that in this case the filters only differ by a scalar value where as our method yields more variation.

Finally, our method is PAC flows. The feature dimension for all of the experiments was set to 16. We test two different autoregressive orderings - a raster order and s-curve order. Jain et al. [2020] found that using multiple s-curve orders performed the best, but in our experiment we use just one.

To make the comparisons fair, we use the exact same conditioner architectures for all of the networks and vary the number of mixture components to ensure that the models have a similar parameter count. The models for over the CIFAR-10 and downsampled ImageNet at a 32x32 resolution have 4.6 million parameters and the models for the downsampled ImageNet dataset at 64x64 resolution have 5.2 million parameters. This was achieved by using 5, 10 and 16 mixture components for the PAC flow, Emerging Conv. and RealNVP models respectively.

Main flow layer Our main flow layer consists of a coupling layer that uses the conditioner and transformer networks described above, followed by a 1x1 convolution and act norm (Kingma and Dhariwal [2018]). Similar to Ho et al. [2019] and Chen et al. [2020], we use both checkerboard and channel splitting (Dinh et al. [2017]).

Fused variational dequantization and channel padding We implemented variational dequantization and channel padding together. Flow architectures typically begin with a dequantization SurVAE flow (Nielsen et al. [2020]) to map an image from $x \in [0, 255]^{H\times W\times C}$ to the reals using the stochastic right inverse of the floor function, $q(z|x)$. Additionally, it has been shown that increasing the dimensionality of an input or adding a stochastic component to a flow can help bypass some topological limitations that bijective functions suffer from and increase performance (Cornish et al. [2019], Huang et al. [2020], Chen et al. [2020], Dupont et al. [2019]), so we do this too. $q(z|x)$ is implemented using 4 of our main flow layers with checkerboard splits. The output $z \in \mathbb{R}^{H\times W\times 2C}$ is split for use in dequantization and padding tensor. First, dequantization is applied to the input image using the first half of z and then the result is rescaled and passed through a scaled logit as described by Dinh et al. [2017]. Then the result is concatenated with the second part of z.

Training We trained all of the models with a batch size of 8 with the AdaBelief (Zhuang et al. [2020]) optimizer with a learning rate of 10^{-3}. We took 2000 gradient steps to warm up the learning rate and used a cosine decay schedule over 100,000 gradient steps to drop the rate to 10^{-4}. The gradients were clipped to a max norm of 15.0. The models were trained for 600,000 gradient steps. We used data dependent initialization with an initial batch size of 128 and initialized our flow to a noisy identity where possible. In the appendix we describe some extra steps we took to avoid numerical instabilities.

5.2 Comparison with PAC Flows

	CIFAR-10	ImageNet 64x64	ImageNet 32x32
RealNVP	3.412870	3.652563	3.935588
Emerging Conv.	3.431914	3.625504	3.936859
PAC raster (ours)	**3.336659**	3.598069	3.888512
PAC s-curve (ours)	3.339258	**3.590702**	**3.877298**

Table 2: Comparison of PAC flow against comparable flows on various datasets, lower is better. RealNVP, Emerging Conv. and PAC share the same architecture but with some modifications to the linear layer. RealNVP uses an elementwise transformation instead of convolution, Emerging Conv. uses a convolution without pixel adaptation in addition to an elementwise multiplication and PAC flow is our full model. The RealNVP and Emerging Conv. models were given more mixture components in order to ensure the number of parameters each model used was the same (see Sec.5.1).

The results of our experiments show that PAC flows outperform the emerging convolution and RealNVP model by a noticeable margin on all three datasets. Table 2 contains the bits per dimension of the models on the test set of each dataset. The PAC models are able to achieve around 0.05 bits per dimension lower than the other models on average. Fig.4 shows the training and test losses during training and we see that PAC flows achieve a smaller loss consistently throughout training. Surprisingly, we find that the s-curve order does not perform any differently from the raster order and the emerging convolution performs only marginally better than the RealNVP model.

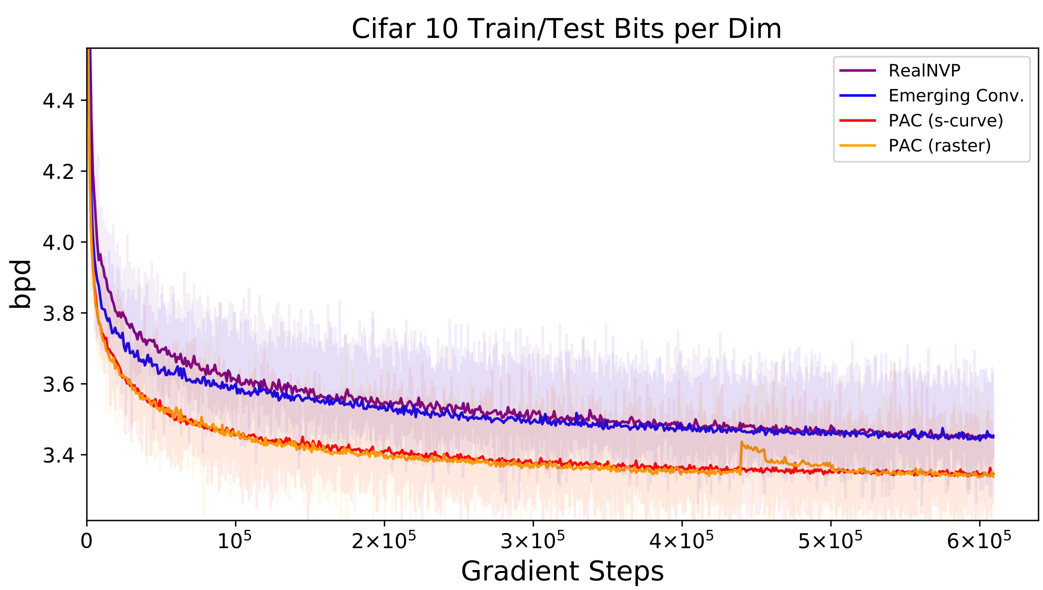


Figure 4: Test and train loss for PAC flows vs other methods. We observe that using convolutions improves on affine transformations and using pixel adaptive convolutions significantly improves on regular convolutions. Test set values are shown in bold colors for each method.

Our results support our hypothesis that the power of coupling comes from the ability to use a different function to transform every element of the input. We see this because even though the emerging conv model applies a strictly more expressive transformer than the RealNVP model, both only have a conditioned shift and scale parameter. Furthermore, when we add pixel adaptation to the emerging conv model (the PAC flow) we see a massive improvement in performance.

5.3 Inversion

We test the inversion speed of PAC flows by examining the maximum absolute difference between consective iterations during the reconstruction of 64 test set images. Fig.5 shows the results. We see that the Jacobi method (Eq.7) can efficiently invert autoregressive convolutions. The kernel and mask are computed on only the first iteration and every remaining iteration only requires evaluating the product in Eq.9 or Eq.10. In contrast to using forward/backward substitution which cost $O((HWC)^2)$, the Jacobi method is significantly more efficient.

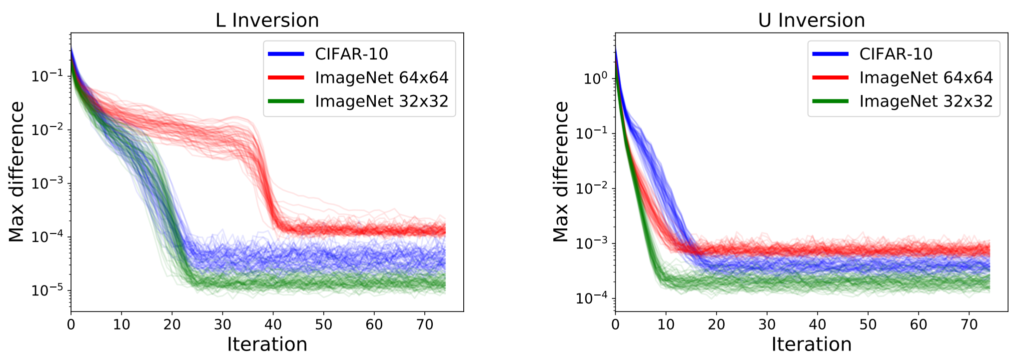


Figure 5: Maximum absolute difference between consecutive iterations of the Jacobi method during reconstructions for 64 test samples. The reported values are averaged over every flow layer in the s-curve PAC flow. PAC flow is inverted in fewer than 60 iterations total as opposed to forward/backward substitution which requires $O(HWC)$ iterations.

6 Conclusion

We introduced a new normalizing flow layer called PAC flow that uses invertible pixel adaptive convolutions. The method applies a different filter to different locations of an input image which makes it suited for normalizing flows where we have a restricted number of channel dimensions and for coupling, where we can parametrize the filters using an unconstrained neural network. PAC flows have a tractable log Jacobian determinant due to its implementation using the PLU decomposition and are efficiently inverted using the weighted Jacobi method. Our experiments indicated that PAC flow outperforms comparable invertible convolution models that do not use pixel adaptive convolutions and that the inversion algorithm converges quickly. A limitation of our model is that the filters it learns are not independent and are instead tied together through the feature parameter. This could potentially limit the flexibility of our model compared to standard convolutional neural networks that can use an arbitrary number of feature maps. Furthermore, by fixing the permutation matrix in our PLU decomposition to the identity matrix, we can only learn a subset of the space of linear transformations. In the future, we will investigate using the convolutional exponential Hoogeboom et al. [2020] in order to learn a wider class of transformations. Our method can potentially be used to improve deepfakes which could be used for nefarious purposes. Conversely, it could be used to to generate representations of health datasets such as brain and heart MRIs that are useful for downstream tasks.

References

Jon Barron. squareplus: a simple all-algebraic alternative to softplus, 2021. URL `https://twitter.com/jon_barron/status/1387167648669048833?lang=en`.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and

Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Jianfei Chen, Cheng Lu, Biqi Chenli, Jun Zhu, and Tian Tian. VFlow: More Expressive Generative Flows with Variational Data Augmentation. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1660–1669. PMLR, July 2020. URL http://proceedings.mlr.press/v119/chen20p.html.

Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017. URL http://arxiv.org/abs/1707.08819.

Rob Cornish, Anthony L. Caterini, George Deligiannidis, and Arnaud Doucet. Relaxing bijectivity constraints with continuously indexed normalising flows, 2019.

Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: non-linear independent components estimation. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*, 2015. URL http://arxiv.org/abs/1410.8516.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=HkpbnH9lx.

Justin Domke. Provable smoothness guarantees for black-box variational inference. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2587–2596. PMLR, 13–18 Jul 2020. URL http://proceedings.mlr.press/v119/domke20a.html.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented Neural ODEs. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/21be9a4bd4f81549a9d1d241981cec3c-Paper.pdf.

Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural Spline Flows. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/7ac71d433f282034e088473244df8c02-Paper.pdf.

Marc Finzi, Pavel Izmailov, Wesley Maddox, Polina Kirichenko, and Andrew Gordon Wilson. Invertible Convolutional Networks. page 6.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2. Publisher: Springer Science and Business Media LLC.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2722–2730. PMLR, June 2019. URL http://proceedings.mlr.press/v97/ho19a.html.

Emiel Hoogeboom, Rianne Van Den Berg, and Max Welling. Emerging Convolutions for Generative Normalizing Flows. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2771–2780. PMLR, June 2019. URL `http://proceedings.mlr.press/v97/hoogeboom19a.html`.

Emiel Hoogeboom, Victor Garcia Satorras, Jakub Tomczak, and Max Welling. The Convolution Exponential and Generalized Sylvester Flows. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 18249–18260. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/d3f06eef2ffac7faadbe3055a70682ac-Paper.pdf`.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural Autoregressive Flows. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2078–2087. PMLR, July 2018. URL `http://proceedings.mlr.press/v80/huang18d.html`.

Chin-Wei Huang, Laurent Dinh, and Aaron C. Courville. Augmented normalizing flows: Bridging the gap between generative flows and latent variable models. *CoRR*, abs/2002.07101, 2020. URL `https://arxiv.org/abs/2002.07101`.

T Huckle. Accelerated jacobi iterations for bidiagonal and sparse triangular matrices. *preprint*, 2019. URL `https://www5.in.tum.de/persons/huckle/it_triang.pdf`.

Ajay Jain, Pieter Abbeel, and Deepak Pathak. Locally Masked Convolution for Autoregressive Models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2020. URL `https://arxiv.org/pdf/2006.12486.pdf`.

Mahdi Karami, Dale Schuurmans, Jascha Sohl-Dickstein, Laurent Dinh, and Daniel Duckworth. Invertible Convolutional Flow. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/b1f62fa99de9f27a048344d55c5ef7a6-Paper.pdf`.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative Flow with Invertible 1x1 Convolutions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/file/d139db6a236200b21cc7f752979132d0-Paper.pdf`.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

Xuezhe Ma, Xiang Kong, Shanghang Zhang, and Eduard Hovy. MaCow: Masked Convolutional Generative Flow. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/20c86a628232a67e7bd46f76fba7ce12-Paper.pdf`.

Didrik Nielsen, Priyank Jaini, Emiel Hoogeboom, Ole Winther, and Max Welling. SurVAE Flows: Surjections to Bridge the Gap between VAEs and Flows. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12685–12696. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/9578a63fbe545bd82cc5bbe749636af1-Paper.pdf`.

George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing Flows for Probabilistic Modeling and Inference. *arXiv:1912.02762 [cs, stat]*, December 2019. URL `http://arxiv.org/abs/1912.02762`. arXiv: 1912.02762.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017. URL `http://arxiv.org/abs/1710.05941`.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1530–1538, Lille, France, 07–09 Jul 2015. PMLR. URL `http://proceedings.mlr.press/v37/rezende15.html`.

Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003. doi: 10.1137/1.9780898718003. URL `https://epubs.siam.org/doi/abs/10.1137/1.9780898718003`.

Yang Song, Chenlin Meng, and Stefano Ermon. MintNet: Building Invertible Neural Networks with Masked Convolutions. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/70a32110fff0f26d301e58ebbca9cb9f-Paper.pdf`.

Hang Su, Varun Jampani, Deqing Sun, Orazio Gallo, Erik Learned-Miller, and Jan Kautz. Pixel-Adaptive Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. URL `https://arxiv.org/pdf/1904.05373.pdf`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017. URL `https://arxiv.org/pdf/1706.03762.pdf`.

Jingjing Xu, Xu Sun, Zhiyuan Zhang, Guangxiang Zhao, and Junyang Lin. Understanding and Improving Layer Normalization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/2f4fe03d77724a7217006e5d16728874-Paper.pdf`.

Guoqing Zheng, Yiming Yang, and Jaime Carbonell. Convolutional normalizing flows, 2018. URL `https://openreview.net/forum?id=HkbJTYyAb`.

Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar C Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *Advances in Neural Information Processing Systems*, 33, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] In the abstract we claimed that PAC flows can be inverted efficiently and can improve performance for invertible convolutions. Sec.5.3 demonstrated quick inversion and sec.5.2 demonstrated the performance gain over invertible convolutions.
   (b) Did you describe the limitations of your work? [Yes] We discussed the limitations in the conclusion.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See the conclusion.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] Much like other generative models, PAC flows could be put towards societal good, though its use for disease diagnosis, for instance. On the other hand, it may be used for deceptive purposes such as the generation of deepfakes.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] We clearly stated the convergence of Eq.7 depends on the assumption that A is triangular and $\alpha \in (0, 2)$.
   (b) Did you include complete proofs of all theoretical results? [Yes] We included a proof of the convergence of Eq.7 in Appendix A.

3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We included code in appendix B and C.4 and will upload our actual implementation as supplementary material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Sec.5.1.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] We say this in the beginning of Sec.5.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See the beginning of Sec.5.
   (b) Did you mention the license of the assets? [N/A] The JAX is open source and the datasets we used are public.
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Weighted Jacobi Method for Triangular Matrices

Here we will show that the weighted Jacobi method converges globally to the solution $x = A^{-1}b$ when A is triangular.

$$x^{(t+1)} = x^{(t)} - \alpha \mathrm{diag}(A)^{-1}(Ax^{(t)} - b)$$

Proof. Let $x^* = A^{-1}b$ be the solution to $Ax = b$. Let $e^{(t)} = x^{(t)} - x^*$ denote the error at the tth iteration. By Theorem 4.1 of Saad [2003], if there is a matrix G s.t. $e^{(t+1)} = Ge^{(t)}$ and the spectral radius of G is less than 1, then the iterations will always converge. For the weighted Jacobi method, $G = I - \alpha \mathrm{diag}(A)^{-1}A$:

$$\begin{aligned}
e^{(t+1)} &= x^{(t+1)} - x^* && (11)\\
&= x^{(t)} - \alpha \mathrm{diag}(A)^{-1}(Ax^{(t)} - b) - x^* && (12)\\
&= x^{(t)} - \alpha \mathrm{diag}(A)^{-1}Ax^{(t)} + \alpha \mathrm{diag}(A)^{-1}A\underbrace{A^{-1}b}_{x^*} - x^* && (13)\\
&= (I - \alpha \mathrm{diag}(A)^{-1}A)x^{(t)} - (I - \alpha \mathrm{diag}(A)^{-1}A)x^* && (14)\\
&= (I - \alpha \mathrm{diag}(A)^{-1}A)(x^{(t)} - x^*) && (15)\\
&= (I - \alpha \mathrm{diag}(A)^{-1}A)e^{(t)} && (16)
\end{aligned}$$

Clearly G is triangular because A is triangular. Also, each of its diagonal entries will be equal to $1-\alpha$. The eigenvalues of triangular matrices are equal to the diagonal entries, so all of the eigenvalues of G are $1 - \alpha$. The spectral radius of G is equal to the maximum absolute value eigenvalue. Therefore, $\rho(G) = |1 - \alpha|$ and will always be less than 1 if $\alpha \in (0, 2)$. Under these conditions, $\rho(G) < 1$, so the weighted Jacobi method will always converge for triangular matrices when $\alpha \in (0, 2)$. $\square$

B NumPy Implementation of PAC Flows

```python
import numpy as np
import einops

def conditioner(x):
  """
  Neural network with learnable parameters.
  If x.shape == (H, W, C), then
  conditioner(x).shape == (H, W, 2*C + F)
  """
  raise NotImplementedError

def make_psi(filter_shape, pad, stride):
  """
  This will depend on your backend.
  JAX:        jax.lax.conv_general_dilated_patches
  PyTorch:    torch.nn.Unfold
  Tensorflow: tf.extract_image_patches

  Should return function "patches" so that
  patches(x).shape == (H, W, C, Kx, Ky)

  If x.shape == (H, W), should return shape (H, W, Kx, Ky)
  """
  raise NotImplementedError

def kernel(psi, f, s, l):
  """
  Compute the pixel adaptive kernel.
```

```
    f.shape == (H, W, F)
    s.shape == (H, W, C)
    l.shape == (H, W, C)
    """
    # Compute the difference between each feature vector
    # within a patch from the center feature.
    # f_diff.shape == (H, W, Kx, Ky)
    f_diff = np.sum((psi(f) - f[...,None,None])**2, axis=-3)

    # Broadcast s and l
    s, l = s[...,None,None,:], l[...,None,None,:]

    # Compute the kernel
    k_i2c = np.exp(-0.5*l*f_diff[...,None])*s

    # Rearrange the kernel so that it is consistent with psi
    k_i2c = einops.rearrange(k_i2c, "... h w u v c -> ... h w c u v")
    return k_i2c

def PAC(x, theta, kernel_size=5, order_type="s_curve"):
    """
    Compute an invertible pixel adaptive convolution.
    Asusmes that the unbatched input shapes are:
    x,shape     == (H, W, C)
    theta.shape == (H, W, 2*C + F)
    """
    H, W, C = x.shape[-3:]

    # Assume that the filter size is odd so that it is
    # easy to pad s.t. the center of the filter corresponds
    # to the diagonal of the Jacobian
    assert kernel_size%2 == 1
    Kx, Ky = kernel_size, kernel_size
    pad_x, pad_y = Kx//2, Ky//2
    c_x, c_y = Kx//2, Ky//2

    # Construct a raster or s_curve order
    order = np.arange(1, 1 + H*W).reshape((H, W, 1))
    if order_type == "s_curve":
      order[::2] = order[::2,::-1]

    # Split the parameters
    f, s, l = theta[...,:-2*C], theta[...,-2*C:-C], theta[...,-C:]

    # W is not learned with coupling
    W = get_parameter("W", shape=(Kx, Ky, C, C))

    # Construct the psi function.  Assume that
    # psi(x).shape == (H, W, C, Kx, Ky)
    psi = make_psi(filter_shape=(Kx, Ky),
                   pad=((pad_x, pad_x), (pad_y, pad_y)),
                   stride=(1, 1))

    # Extract the psi of the input and order
    x_i2c, order_i2c = psi(x), psi(order)

    # Get the autoregressive mask
    order = np.arange(1, 1 + util.list_prod(order_shape)).reshape(order_shape)
    mask = order[...,None,None] >= order_i2c
```

```
    # Compute the kernel
    k_i2c = kernel(psi, f, s, l)

    # Compute z = LUx
    pattern = "...hwiuv,...hwouv,uvio->...hwo"
    z = np.einsum(pattern, mask*x_i2c, k_i2c, np.triu(W))
    z = np.einsum(pattern, ~mask*psi(z), k_i2c, W) + z

    # Compute the diagonal of the transformation
    diag = k_i2c[...,c_x,c_y]*np.diag(W[c_x,c_y])

    # Compute the log Jacobian determinant
    log_det = np.log(np.abs(diag)).sum(axis=(-1,-2,-3))
    return z, log_det
```

C More Architecture Details

C.1 Nonlinearity

In our experiments we make use of a smooth approximation to the relu function with a non exponentially decaying tail:

$$\text{sp}(x;\gamma) = \frac{1}{2}(x + \sqrt{x^2 + 4\gamma})$$

The default value of γ that we use is 0.5. We were first made aware of this function from theorem 13 in Domke [2020] and found that it was a good fit for scaling parameters in normalizing flows, however it has recently been rediscovered by Barron [2021] who called it the "squareplus" function. We will denote it as $\text{sp}(x;\gamma)$. We also note that a similar kind of approximation for leaky relu, called "sneaky relu", was introduced in Finzi et al..

We can use squareplus to give us an approximation of the sigmoid function with the same nice tail properties by taking its derivative. We call this the "squaresigmoid" function

$$\begin{aligned} \text{ss}(x;\gamma) &= \frac{d}{dx}\text{sp}(x;\gamma) \\ &= \frac{1}{2}(x + \frac{x^2}{\sqrt{x^2 + 4\gamma}}) \end{aligned}$$

This immediately leads to an approximation of the swish Ramachandran et al. [2017] function, which we call "squareswish", as $x * \text{ss}(x;\gamma)$. We use this approximate swish as our neural network nonlinearities.

C.2 Numerical stability

Some of the parameters to flow layers must be positive. In order to have a neural network learn unconstrained parameters, we must pass the neural network outputs through a function, ϕ that ensures that the outputs are positive. The softplus and sigmoid function are common examples of this. However, one must be careful when using this to generate a parameter that is divided. For example, in RealNVP we learn a value to divided an input by:

$$\begin{aligned} \theta &= \text{NN}(x_2) \\ \widehat{s}, b &= \theta \\ z &= \frac{x - b}{\phi(\widehat{s})} \end{aligned}$$

If $\widehat{s}$ is too negative, then $\phi(\widehat{s})$ can get close to 0, causing numerical stability issues. This can be the case with softplus and sigmoid because they have exponentially decaying negative tail. So instead we use the squareplus function for this task and find that it works well in practice.

During data dependent initialization, if we want $\mathrm{sp}(\theta;\gamma)$ to be the standard deviation of a batch of inputs, we compute the standard deviation of our input, then set θ to be the inverse of the squareplus function:

$$\mathrm{sp}^{-1}(x;\gamma) = x - \frac{\gamma}{x} \tag{17}$$

If θ is the output of a neural network and we want to initialize $\mathrm{sp}(\theta;\gamma)$ to be 1, we simply use zero initialization for the neural network (so that θ is initialized to a value close to 0) and set the hyperparameter $\gamma = 1.0$ because $\mathrm{sp}(x;1.0)$ passes through the point (0,1).

C.3 Bounding the PAC kernel parameters

The kernel parameters of PAC, f, σ^2 and l only need to satisfy the constraint that $\sigma^2, l > 0$. However, we found that only satisfying this constraint led to poor test set performance. One solution to this was to force $f \in (-1,1)$ and $\sigma^2, l \in (0,1)$. We enforced this using the squaresigmoid function.

C.4 Architecture code outline

The following code is an outline of the architecture we used for our experiments.

```
def gated_resblock(x, hidden_channel, aux=None):
  channel_in = x.shape[-1]
  gx = nonlinearity(x)
  gx = Conv(x, hidden_channel, kernel=3, stride=1, weight_norm=True)
  if aux is not None:
    # This is used during dequantization to condition on x
    aux = nonlinearity(aux)
    aux = Conv(aux, hidden_channel, kernel=1, stride=1, weight_norm=True)
    gx += aux
  gx = nonlinearity(gx)
  gx = dropout(gx, 0.2)
  gx = Conv(x, 2*channel_in, kernel=1, stride=1, weight_norm=True)
  a, b = split(gx, 2, axis=-1)
  gx = a*sigmoid(b)
  return gx

def conditioner(x, out_channel, n_res_blocks, hidden_channel, initial_channel, aux=None):
  x = Conv(x, initial_channel, kernel=1, stride=1)
  for i in range(n_blocks):
    gx = gated_resblock(x, hidden_channel, aux=aux)
    x += gx
    x = layer_norm(x)
  x = Conv(x, out_channel, kernel=1, stride=1)
  return x

def affine(x, theta, bias=None, kind="pac", order="s_curve"):

  if kind == "pac":
    x, log_det = PAC(x, theta, kernel=5, order=order)
  elif kind == "emerging":
    conv_params, scale = theta
    # Used same implementation as pac model, but without kernel
    x, log_det1 = PAC(x, theta, kernel=5, order=order, pixel_adaptive=False)
    x, log_det2 = ElementwiseScale(x, scale)
    log_det = log_det1 + log_det2
  elif kind == "realnvp":
    x, log_det = ElementwiseScale(x, scale)

  if bias is not None:
```

```
    # This differentiates the affine from linear layer
    x += bias
  return x, log_det

def transformer(x, theta, kind="pac", order="s_curve"):
  linear1, bias, logistic_cdf_mix_theta, linear2 = theta

  # Set to ensure models have same number of parameters
  if kind == "pac":
    n_mixtures = 5
  elif kind == "emerging":
    n_mixtures = 10
  elif kind == "realnvp":
    n_mixtures = 16

  x, log_det1 = affine(x, linear1, bias=bias, kind=kind, order=order)
  x, log_det2 = logistic_cdf_mixture_logit(x, logistic_cdf_mix_theta, n_mixtures=n_mixtures)
  x, log_det3 = affine(x, linear2, bias=None, kind=kind, order=order)
  x, log_det4 = SLogGate(x)
  log_det = log_det1 + log_det2 + log_det3 + log_det4
  return x, log_det

def main_flow(x, aux=None, checkerboard=True, kind="pac", order="s_curve", **cond_kwargs):
  if checkerboard:
    x = squeeze(x)

  x1, x2 = split(x, 2, axis=-1)
  theta = conditioner(x2, aux=aux, **cond_kwargs)
  z1, log_det1 = transformer(x1, theta, kind=kind, order=order)
  z = concatenate(z1, x2, axis=-1)

  z, log_det2 = OneByOneConv(z, weight_norm=True)
  z, log_det3 = ActNorm(z)
  log_det = log_det1 + log_det2 + log_det3

  if checkerboard:
    z = unsqueeze(z)
  return z, log_det

def fused_dequantization_padding(x, kind, order, padded_channel_size, feature_dim):

  # Extract some useful info about x first
  f_cond_kwargs = dict(out_channel=3,
                       hidden_channel=64,
                       initial_channel=32,
                       n_resnet_blocks=6,
                       aux=None)
  f = conditioner(x, **f_cond_kwargs)

  # Sample from q(z|x)
  q_cond_kwargs = dict(out_channel=2*padded_channel_size + feature_dim,
                       hidden_channel=64,
                       initial_channel=32,
                       n_resnet_blocks=3,
                       aux=f)
  noise = UnitGaussian.sample(shape=(H, W, padded_channel_size))
  log_qzgx = 0.0
  for i in range(4):
    # Sample from the main flow (invert not shown above)
```

```
    kwargs = dict(aux=f, checkerboard=True, kind=kind, order=order, invert=True)
    kwargs.update(q_cond_kwargs)
    noise, llc = main_flow(noise, **kwargs)
    log_qzgx += llc
  dequant_noise, padding_noise = noise.split(axis=-1)

  # Standard dequantization steps
  dequant_noise, log_det_sigmoid = sigmoid(dequant_noise)
  x += dequant_noise
  x, log_det_scale = Scale(x, 256)
  x, log_det_logit = logit(x)

  # Pad
  z = concatenate(x, padding_noise, axis=-1)
  llc = -log_qzgx + log_det_sigmoid + log_det_scale + log_det_logit
  return z, llc

def full_architecture(x, kind, order, padded_channel_size, feature_dim):
  x, elbo = fused_dequantization_padding(x, kind, order, padded_channel_size, feature_dim)

  condition_kwargs = dict(out_channel=2*padded_channel_size + feature_dim,
                          n_res_blocks=6,
                          hidden_channel=64,
                          initial_channel=32,
                          aux=None)
  kwargs = condition_kwargs
  kwargs.update(dict(kind=kind, order=order))

  for i in range(3):
    x, dlog_det = main_flow(x, checkerboard=True, **kwargs)
    elbo += dlog_det
  for i in range(3):
    x, dlog_det = main_flow(x, checkerboard=False, **kwargs)
    elbo += dlog_det

  x = squeeze(x)
  for i in range(3):
    x, dlog_det = main_flow(x, checkerboard=True, **kwargs)
    elbo += dlog_det
  for i in range(3):
    x, dlog_det = main_flow(x, checkerboard=False, **kwargs)
    elbo += dlog_det

  log_pz = UnitGaussianPrior(x)
  elbo += log_pz
  return x, elbo

def RealNVP(x):
  return full_architecture(x,
                           kind="realnvp",
                           order=None,
                           padded_channel_size=6,
                           feature_dim=16)

def EmergingConv(x):
  return full_architecture(x,
                           kind="emerging",
                           order="raster",
                           padded_channel_size=6,
```

```
                              feature_dim=16)

def PAC_raster(x):
  return full_architecture(x,
                              kind="pac",
                              order="raster",
                              padded_channel_size=6,
                              feature_dim=16)

def PAC_s_curve(x):
  return full_architecture(x,
                              kind="pac",
                              order="s_curve",
                              padded_channel_size=6,
                              feature_dim=16)
```

D Samples

Figure 6: Samples from PAC s-curve trained on CIFAR 10.

Figure 7: Samples from PAC s-curve trained on ImageNet 32x32.

Figure 8: Samples from PAC s-curve trained on ImageNet 64x64.