Flowification: Everything is a Normalizing Flow

Anonymous Author(s)
Affiliation
Address
email

Abstract

We develop a method that can be used to turn any multi-layer perceptron or convolutional network into a normalizing flow. In some cases this requires the addition of uncorrelated noise to the model but in the simplest case no additional parameters. The techniques we develop can be applied to a broad range of transformations. Converting standard models to normalizing flows allows the same architectures to be used for a wide range of tasks. Our models also allow existing density estimation techniques to be combined with high performance feature extractors and for the exact likelihood to be calculated. In contrast to standard density estimation techniques that require specific architectures and specialized knowledge, our approach can leverage design knowledge from other domains and is a step closer to the realization of general purpose architectures. We investigate the efficacy of linear and convolutional layers for the task of density estimation on standard datasets. Our results suggest standard layers lack something fundamental that other normalizing flows do not.

1 Introduction

Density estimation techniques have proven effective on a wide variety of downstream tasks such as sample generation and anomaly detection [1–6]. Normalizing flows and autoregressive models perform very well at density estimation but do not easily scale to large dimensions [2, 7, 8] and have to satisfy strict design constraints to ensure efficient computation of their Jacobian and inverse. Advances in other areas of machine learning cannot be utilized as flow architectures because they are not typically seen as being invertible, this restricts the application of highly optimized architectures from many domains to density estimation and the use of the likelihood for diagnosing these architectures.

Methods using standard convolutions and residual layers for density estimation have been developed for architectures with specific properties [9–12]. These methods do not provide a recipe for converting general architectures into flows. There is no known correspondence between normalizing flows and the operations defined by linear and convolutional layers.

In this paper we show that a large proportion of machine learning models can be trained as normalizing flows. The forward pass of these models remains unchanged up to the addition of uncorrelated noise. To demonstrate our formulation works we consider fully connected layers, convolutions and residual connections as examples. The code used for these experiments will be made publicly available.

The contributions of this paper include:

- In §3.1 we show that linear layers induce densities as augmented normalizing flows [13] with the multi-scale architecture used in RealNVPs [3]. We also show how these layers can be viewed as funnels [14] to increase their expressivity. We term this process *flowification*.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

- In §3.2 we argue that most ML architectures can be decomposed into simple building blocks that are easy to flowify. As an example, we work out the details for 2-dimensional convolutional layers and residual blocks.
- In §4 we flowify convolutional networks and train them as normalizing flows using the likelihood. This demonstrates that models built from standard layers can be used for density estimation directly.

2 Background

Flows A complex probability distribution $\{p(x)|x \in X\}$ can be constructed from a simple base density[1] $\{p(z)|z \in Z\}$ and a diffeomorphism between X and Z, $f : X \to Z$. The likelihood of any given x is then given by $p(x) = p(f(x))|\det(J_x^f)|$, where J_x^f is the Jacobian of f evaluated at x. Thus, the log-likelihoods of the two densities are related by an additive term, which will be referred to as the *likelihood contribution* $\mathcal{V}(x, z)$ [15]. Normalizing flows [1] chain many invertible maps that are tuned to maximize the (log-)likelihood of the data. A lot of development has gone into constructing flexible invertible layers with easy to calculate Jacobians where both the forward and inverse passes are fast to compute [2–4, 16, 17].

As the function f must be invertible it is required to preserve the dimension of the data. This limits the expressivity of f and makes it expensive to model high dimensional data distributions. To reduce these issues several works have studied dimension altering variants of flows [13, 15, 18–21].

Reducing the dimensionality A simple method for altering the dimension of a flow is to take the output of an intermediate layer z' and split it into two pieces $z' = \{z'_1, z'_2\}$. Multi-scale architectures [3] match one of these pieces directly to the base density and apply further transformations to the other. Funnels [14] model z'_2 as being dependent on z'_1 and work with $p(z') = p(f'(z'_1))p(z'_2|f'(z_1))|\det J^{f'}_{z'_1}|$ where f' can be any computational graph for which the likelihood contribution can be calculated. The conditional distribution can be a trainable density and z'_1 is modelled with additional layers. It is useful to think of this factorization as a dimension reducing mechanism from $\dim(z')$ to $\dim(z'_1)$.

Increasing the dimensionality Dimension increasing flow layers can improve a models flexibility, as demonstrated by augmented normalizing flows [13]. To increase the dimensionality, x is embedded into a larger dimensional space and data independent noise u is added to the embedding to obtain a distribution with support of nonzero measure. This noise addition is similar to dequantization [3, 22], but is orthogonal to the distribution of x and increases its dimension. It is important to note that the augmentation $x \mapsto (x, u)$ increases the dimension from $\dim x$ to $\dim x + \dim u$. Under an augmentation the likelihood of x can be estimated using

$$
\begin{aligned}
\log p(x) &= \log \int du\, p(x, u) \\
&= \log \int du\, \frac{p(u)p(x, u)}{p(u)} \\
&\geq \int du\, p(u) \log \frac{p(x, u)}{p(u)} \\
&= \mathbb{E}_{u\sim p(u)}\big[\log \frac{p(x, u)}{p(u)}\big] \\
&= \mathbb{E}_{u\sim p(u)}\big[\log p(x, u) - \log p(u)\big].
\end{aligned}
\tag{1}
$$

In practice, we estimate this expectation value by sampling u everytime we pass a datapoint through the network. This means the integral is estimated with a single monte carlo sample as in surVAEs [15].

[1] One for which the likelihood function is easy to evaluate, usually a gaussian.

3 Flowification

Suppose $\mathcal{F}$ is a network architecture with parameter space Θ. Then for any choice of $\theta \in \Theta$ the network with parameters θ realizes a function $\mathcal{F}_\theta : \mathbb{R}^D \to \mathbb{R}^C$ for some D and C. Similarly, a normalizing flow model $\mathcal{P}$ is a parametric distribution on some $\mathbb{R}^A$ and for any choice of γ from their parameter space Γ they define a density function $\mathcal{P}_\gamma$ on $\mathbb{R}^A$. In this work we show that a large class of neural network architectures $\mathcal{F}$ can be thought of as a flow model by constructing a map

$$\left\{ \begin{array}{c} \text{network} \\ \text{architectures} \end{array} \right\} \to \left\{ \begin{array}{c} \text{flow} \\ \text{models} \end{array} \right\}. \tag{2}$$

This embedding of $\mathcal{F}_\theta$ to its flowification $\mathcal{F}^{\mathcal{P}}_{\theta'} : \mathbb{R}^{D+N} \to \mathbb{R}^{D+N}$ and a base distribution define a density on $\mathbb{R}^D$ for some $N \geq 0$. In the simplest case $\theta' = \theta$. For every θ the expectation value of the forward pass of any layer of $\mathcal{P}$ coincides with the output of the same layer of $\mathcal{F}^{\mathcal{P}}_\theta$ for all input vectors. It is in this sense that we claim that a large fraction of machine learning models are normalizing flows.

Terminology In what follows we will work with conditional distributions such as $p(z|x)$ and it will be practical to think of them as "stochastic functions" $p : x \mapsto z$, that take an input x and produce an output $z \sim p(z|x)$. Conversely, we can think of a function $f : x \mapsto z$ as the Dirac δ-distribution $f(z|x) = \delta(z - f(x))$. These definitions allow us to have a unified notation for deterministic and "stochastic functions" and we can talk about them in the same language. Consequently, when we say "stochastic function", it will include deterministic functions as a corner case. Depending on if f and f^{-1} are deterministic or stochastic, we can talk about left, right or two-sided inverses. We will be careful to always be precise about this. We recommend [15] for a more detailed discussion on this topic.

Method In the following we consider the standard building blocks of machine learning architectures and enrich them by defining (stochastic-)inverse functions and calculating the likelihood contribution of each layer. Treating each layer separately allows density estimation models to be built through composition [15]. This stochastic inverse can use the funnel approach, in which case additional parameters are required to model the inverse, or the multi-scale approach, in which case no additional parameters are required. For simplicity we will only consider conditional densities in the inverse as this is more general, but it is not necessary for the definitions to be correct. We will refer to this process as *flowification* and the enriched layers as *flowified*, non-flowified layers will be called *standard layers*. Flowified layers can then be seen as a being simultaneously

- **Flow** layers that are invertible, their likelihood contribution is known and therefore can be used to train the model to maximize the likelihood.
- **Standard** layers that can be trained with losses other than the likelihood, but for which the likelihood can be calculated after this training with fixed weights in the forward direction.

3.1 Linear Layers

Let $L_{W,b} : \mathbb{R}^n \to \mathbb{R}^m$ denote the the linear layer of a neural network with parameters defined by a weight matrix $W \in \mathbb{R}^{m \times n}$ and bias $b \in \mathbb{R}^m$. Formally, $L_{W,b}$ is defined as the affine function

$$x \mapsto L_{W,b}(x) \coloneqq Wx + b \qquad x \in \mathbb{R}^n. \tag{3}$$

Definition 1. *Let $\phi(z|x) : \mathbb{R}^n \to \mathbb{R}^m$ be a stochastic function. We say that ϕ is* ***linear in expectation*** *if there exists $W \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ such that for any $x \in \mathbb{R}^n$ the expected value of ϕ coincides with the application of $L_{W,b}$*

$$\mathbb{E}_{z \sim \phi(z|x)}[z] = L_{W,b}(x). \tag{4}$$

Similarly, we say that a stochastic function $\psi(z|x)$ is convolutional in expectation if the expectation $\mathbb{E}_{z \sim \psi(z|x)}[z]$ is given by a convolutional layer.

In this section we *flowify* linear layers, by which we mean we construct of a pair of stochastic functions, a forward $\mathcal{L}(z|x) : \mathbb{R}^n \to \mathbb{R}^m$ and an inverse $\mathcal{L}^{-1}(x|z) : \mathbb{R}^m \to \mathbb{R}^n$ such that the forward is linear in expectation and is compatible with the inverse in a way that will be made precise in the following paragraphs. Note that the inverse need not be conditioned on z and therefore does not necessarily increase the number of parameters in the model.

SVD parametrization

To build a flowified linear layer, the first step is to parametrize the weight matrices by the singular value decomposition (SVD). This involves writing $W \in \mathbb{R}^{m\times n}$ as a product $W = V\Sigma U$, where $U \in \mathbb{R}^{n\times n}$ is orthogonal, $\Sigma \in \mathbb{R}^{m\times n}$ is diagonal and $V \in \mathbb{R}^{m\times m}$ is orthogonal. This parametrization is particularly useful for our purposes because the orthogonal transformations are easily invertible and don't contribute to the likelihood and the non-invertible piece of the transformation is localized to Σ.

Parametrizing U and V We generate elements of the special orthogonal group $SO(d)$ by applying the matrix-exponential to elements of the Lie algebra $\mathfrak{so}(d)$ of skew-symmetric matrices. We parametrize $\mathfrak{so}(d)$ and perform gradient descent there. As the Lie-algebra is a vector space, this is significantly easier than working directly with $SO(d)$. See Appendix B for details.

Parametrizing Σ The matrix Σ is of shape $m \times n$ containing the singular values on the main diagonal. We ensure maximal rank of Σ, by parameterizing the logarithm of the main diagonal, this way all singular values are greater than 0.

It is important to note that this parametrization is not without loss of generality. In particular, it does not include matrices of non-maximal rank nor orientation reversing ones, where either $U \in O(n) \setminus SO(n)$ or $V \in O(m) \setminus SO(m)$. This implementation detail does not change the general perspective we provide of linear layers as normalizing flows, and instead simplifies the implementation of flowified layers.

Decreasing the dimension

Definition 2. *We call the tuple $(\mathcal{L}(z|x), \mathcal{L}^{-1}(x|z))$ a **dimension decreasing flowified linear layer** if $\mathcal{L}$ is dimension decreasing, linear in expectation and the following conditions are satisfied*

(i) The forward is deterministic, given by $\mathcal{L}(z|x) = L_{W,b}(x)$.

(ii) The network is right-invertible, $\mathcal{L} \circ \mathcal{L}^{-1} = id_z$.

(iii) The likelihood contribution of $\mathcal{L}$ can be exactly computed.

To flowify dimension decreasing linear layers, we define the forward function $\mathcal{L}$ as a standard linear layer with parameters W and b,

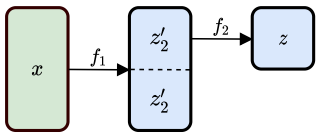
$$\mathcal{L}(z|x) = \delta(z - L_{W,b}(x)). \tag{5}$$

Since W is parametrized by the SVD decomposition, $W = V\Sigma U$, we need to invert V, U and Σ separately. As V and U are rotations, they are inverible in the usual sense. To construct a stochastic inverse to Σ, we think of it as a funnel[14] and use a neural network $p_{inv}((Ux)_{(m:)}|\Sigma Ux)$ that models the dropped $n - m$ coordinates as a function of the non-dropped m coordinates. Again, this is not required to calculate the likelihood under the model, even a fixed distribution could be used, but introducing some trainable parameters significantly improves the performance of the flow that is defined by the layer. We use Σ^{-1} to denote this stochastic inverse to Σ. The stochastic inverse function $\mathcal{L}^{-1}$ can then be written as

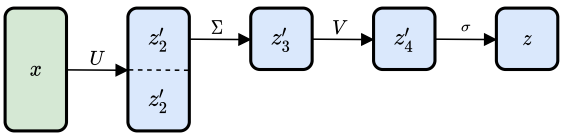
$$\mathcal{L}^{-1}(x|z) = U^T \circ \Sigma^{-1} \circ V^T. \tag{6}$$

Since the rotations don't contribute to the log-likelihood, the likelihood of data under a dimension decreasing flowified linear layer is

$$\log p(x) = p_{inv}((Ux)_{(m:)}|\Sigma Ux) + \log \Sigma + \log p(z), \tag{7}$$



(1a) Mutli-scale flow with a base density $p(z, z_2')$.



(1b) A dimension reducing linear layer with activation σ as a multi-scale flow with a base density $p(z, z_2')$.

where $\log \Sigma$ denotes the sum of the logarithms of the diagonal elements of Σ.

Theorem 3. *The above choices for $\mathcal{L}$ and $\mathcal{L}^{-1}$ define a dimension decreasing flowified linear layer.*

Sketch of proof. The definition of the forward pass (5) makes the forward pass linear in expectation and satisfies (i) by definition. Unpacking the definitions and decomposing W into its SVD form yields right-invertibility (ii) which in turn implies that the likelihood contribution can be exactly computed (iii). □

When the inverse density is not made to be conditional the above ideas can be visualized as a standard multi-scale flow architecture [3] as shown in Fig. 1b.

Increasing the dimension

Definition 4. *We* define *the **Moore-Penrose pseudoinverse** $L^+_{W,b}$ of a linear layer $L_{W,b} : \mathbb{R}^n \to \mathbb{R}^m$ as the affine transformation $\mathbb{R}^m \to \mathbb{R}^n$*

$$z \mapsto L^+_{W,b} := W^+(z - b) \qquad z \in \mathbb{R}^m \tag{8}$$

where W^+ denotes the Moore-Penrose pseudoinverse of the matrix W.

Definition 5. *We call the tuple $(\mathcal{L}(z|x), \mathcal{L}^{-1}(x|z))$ a **dimension increasing flowified linear layer** if $\mathcal{L}$ is dimension increasing, linear in expectation and the following conditions are satisfied*

(iv) The inverse $\mathcal{L}^{-1}$ is deterministic, given by $\mathcal{L}^{-1}(x|z) = L^+_{W,b}(z)$.

(v) The network is left-invertible, $\mathcal{L}^{-1} \circ \mathcal{L} = id_x$

(vi) The likelihood contribution of $\mathcal{L}$ can be bounded from below.

To construct dimension increasing flowified linear layers, we rely again on the SVD parametrization where the only nontrivial component is Σ. In this case Σ is a dimension increasing operation and we think of it as an augmentation step [13] composed with diagonal scaling. To augment, we sample $m - n$ coordinates from a distribution U with zero mean and then apply a scaling in m dimensions. The likelihood contribution is then given by

$$\log p(x) = \log \Sigma + \log p(z), \tag{9}$$

where $\log \Sigma$ denotes the sum of the logarithms of the m scaling parameters. The inverse function $\mathcal{L}^{-1}$ is the composition of the inverse rotations, the inverse scaling and the dropping of the sampled coordinates.

Theorem 6. *The above choices for $\mathcal{L}$ and $\mathcal{L}^{-1}$ define a dimension increasing flowified linear layer.*

Sketch of proof. The augmentation step [13] results in a lower bound on the likelihood contribution, implying (vi). Since $\mathbb{E}_{u \sim U} = 0$, the augmentation does not influence the expected value of z, i.e. the forward pass is linear in expectation. Simple calculations using the SVD decomposition then imply both (iv) and (v). □

Preserving the dimension

Dimension preserving layers are a corner case of both of the above scenarios, where padding and sampling are not needed in either direction and the layer is non-stochastically invertible. All this implies (i),(ii),(iii),(iv) and (v).

3.2 Convolutional layers

Convolutions can be seen as a dimension increasing coordinate repetition followed by a matrix multiplication with weight sharing. In the previous section we worked out the details of matrix multiplication. We begin this section with the details of coordinate repetition after which we put the pieces together to build a flowified convolutional layer. In Appendix D we describe an alternative approach relying on the Fourier transform.

Repeating coordinates In this paragraph we focus on the N-fold repetition of a single scalar coordinate x. This significantly simplifies notation, but the technique generalizes in an obvious way. Intuitively, the idea is to expand the one dimensional volume to N dimensions by first embedding and then increasing the volume of the embedding such that we have control over the volume in the $N-1$ directions complementary to the embedding.

We have seen in §2 that the operation

$$x \mapsto (x, \underline{\mathbf{u}}) \qquad \underline{\mathbf{u}} = (u_1, ..., u_{N-1}), \tag{10}$$

has likelihood contribution $\mathbb{E}_{\underline{\mathbf{u}}}[-\log p(\underline{\mathbf{u}})]$. Now, we can apply any N-dimensional rotation R_N which maps $(1, \underline{\mathbf{0}})$ to $\frac{1}{\sqrt{N}}(1, \underline{\mathbf{1}})$ to obtain[2]

$$R_N(x, \underline{\mathbf{u}}) = R_N(x, \underline{\mathbf{0}}) + R_N(0, \underline{\mathbf{u}}) = \frac{1}{\sqrt{N}}(x, \underline{\mathbf{x}}) + R_N(0, \underline{\mathbf{u}}). \tag{11}$$

Note that this rotation does not contribute to the likelihood. Finally, we apply a diagonal scaling in N dimensions with factor $\sqrt{N}$ such that

$$x \mapsto (x, \underline{\mathbf{x}}) + R_N(0, \sqrt{N}\underline{\mathbf{u}}), \tag{12}$$

where the final scaling has likelihood contribution $N \log(\sqrt{N}) = (N/2) \log N$ and x is now repeated N times. The overall contribution to the likelihood of the embedding (12) is

$$\mathcal{V}(x, z) = \mathbb{E}_{\underline{\mathbf{u}}}[-\log p(\underline{\mathbf{u}})] + (N/2) \log N. \tag{13}$$

By construction, the padding distribution $R_N(0, \sqrt{N}\underline{\mathbf{u}})$ is orthogonal to the diagonal embedding $x \mapsto (x, ..., x)$ of the data distribution. The inverse function is given by the projection to the diagonal embedding,

$$(z_1, ..., z_N) \mapsto \tfrac{1}{N} \sum_i z_i. \tag{14}$$

General architectures Now that the likelihood contribution of arbitrary linear layers and coordinate repetition has been computed it is possible to flowify more general architectures such as convolutions and residual connections. It is important to add that just because an architecture works well for certain tasks, it is not clear if it's flowified version will perform well at density estimation.

Decomposing convolutional layers To flowify convolutional layers, we decompose it as a sequence of building blocks that are easy to flowify separately. A standard convolutional layer performs the following sequence of steps:

1. **Padding** of the input image with zeros to increase it's size.
2. **Unfolding** of the padded image into tiles. This step replicates the data according to the kernel size and stride.
3. **Applying a linear layer.** Finally, we apply the same linear layer to each of the tiles produced in the previous step. The outputs then correspond to the pixels of the output image.

[2] $\underline{\mathbf{0}}$ and $\underline{\mathbf{1}}$ denote the $(N-1)$-dimensional vectors $(0, ...0)$ and $(1, ..., 1)$, respectively. Similarly $\underline{\mathbf{x}}$ denotes the $(N-1)$-dimensional vector $(x, ..., x)$.

Flowification Steps 1 and 3 are already flowified, i.e. their likelihood contribution is computed and an inverse is constructed, in §3.1. We denote their flowification with Pad and Linear, respectively. Step 2 fits into the discussion of the previous paragraph of repeating coordinates, where both its inverse (14) and its likelihood contribution (13) are given. We will denote this operation by Unfold.

Definition 7. *Let* $\mathrm{Linear}, \mathrm{Unfold}$ *and* Pad *be as above and define* $\mathcal{C}$ *and* $\mathcal{C}^{-1}$ *be the following stochastic functions*

$$\mathcal{C} = \mathrm{Linear} \circ \mathrm{Unfold} \circ \mathrm{Pad} \tag{15}$$

$$\mathcal{C}^{-1} = \mathrm{Pad}^{-1} \circ \mathrm{Unfold}^{-1} \circ \mathrm{Linear}^{-1} \tag{16}$$

call the resulting layer $(\mathcal{C}, \mathcal{C}^{-1})$ *a* ***flowified convolutional layer****.*

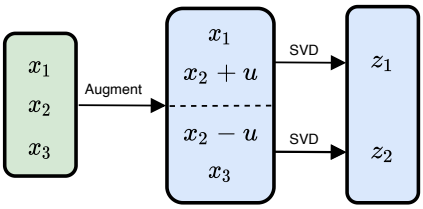


Figure 2: A flowified 1D-convolution with kernel size 2 applied to a vector with 3 features. The x_2 component appears in the operation twice, and so it is first duplicated so that the kernel can be applied to non-overlapping tiles.

A flowified convolutional layer $(\mathcal{C}, \mathcal{C}^{-1})$ is then *convolutional in expectation* (Definition 1), i.e. there exists a convolutional layer C_θ with parameters θ such that

$$\mathbb{E}_{z \sim \mathcal{C}(z|x)}[z] = C_\theta(x). \tag{17}$$

The flowification of a convolution without padding can be seen in Fig. 2. The Unfold operation is implemented as coordinate duplication and Linear is a flowified linear layer parameterized by the SVD.

Activation functions Functions that are surjective onto $\mathbb{R}$ and invertible fit well in our framework as they can be used out of the box without any modifications. In our experiments we use LeakyReLU and rational-quadratic splines [4] as activations functions. Non-invertible activations can also be used when equipped with additional densities [15].

Residual connections Residual connections can be seen as coordinate duplication followed by two separate computational graphs $\{f_1, f_2\}$ that are the recombined with a sum. The sum can be inverted by defining a density over one of the summands $p(f_1(x+u)|f_1(x+u), f_2(x_2))$ and sampling from this density, which will also define the likelihood contribution. Then if the likelihood contribution can be calculated for each individual computational graph the likelihood of the total operation can be calculated [15].

4 Experiments

In this section we flowify convolutional architectures and train them to maximize the likelihood of image data. We use the MNIST [23] and CIFAR10 [24] datasets in our experiments with the standard training and test splits. The data is uniformly dequantized as required to train on image data [25, 26]. The network for both datasets consists of convolutional layers followed by dense layers. The exact architectures can be found in Appendix C. The flowified layers sample from $\mathcal{U}(-a/2, a/2)$ for dimension increasing operations where a is a per-layer trainable parameter. We use RQ-splines [4] with 8 knots and a tail bound of 2 as activation functions, where the same function is applied per output node. Importantly, we don't use existing flow layers in the forward pass of our layers, nor any data-dependent operations such as couplings unless we explicitly say so.

The results of the density modelling can be seen in Table. 1. The samples in the left column of Fig. 3 are generated by sampling from a standard gaussian in the latent space and taking the expected output of the inverse in every layer. The images in the right column use the same latent samples as the left column but also sample from the distribution defined by the inverse pass of the layers.

To minimize the number of augmentation steps that occur in each model we define architectures with similar numbers of parameters but with non-overlapping kernels in the convolutional layers. As seen

Table 1: Test-set bits per dimension (BPD) for MNIST and CIFAR-10 models, lower is better. Results from several other works were included for comparison. Flowified models with overlapping kernels FCONV1 and non-overlapping kernels FCONV2 are shown, with a similar parameter budget to the neural spline flow [4].

MODEL	MNIST	CIFAR-10
GLOW [2]	1.05	3.35
REALNVP [3]	-	3.49
NSF [4]	-	3.38
I-RESNET [9]	1.06	3.45
I-CONVNET [12]		4.61
MAF* [17]	1.91	4.31
FCONV1	3.41	6.24
FCONV2	3.18	4.18

Figure 3: Samples from flowified convolutional networks with overlapping kernels trained on MNIST (top) and CIFAR-10 (bottom). Samples where the mean is used to invert the SVD in the inverse pass (left). Samples generated by drawing from the inverse density to invert the SVD in the inverse pass (right).

in Table. 1 these models perform significantly better. This suggests that the augmentation step costs more in terms of likelihood than it provides in terms of increased expressivity. The samples from this model are shown in Fig. 4 and are slightly improved in terms of visual quality which is a reflection of their improved modelling of the density.

5 Related Work

Several works have developed methods that allow standard layers to be made invertible, but these approaches restrict the space of the models, whereas we consider networks in their full generality. Invertible ResNets [9, 10] require that each residual block has a Lipschitz constant less than one, but even with this restriction they attain competitive results on both classification and density estimation. The same Lipschitz constraint can also be applied to other networks [11]. In these architectures the multi-scale architecture used in RealNVPs [3] is not leveraged, and so no information is discarded by the model. It is unclear why this approach outperforms flowified layers, as shown in Table. 1, but it could be due to the preservation of information through the model, the very large number

Figure 4: Samples from flowified convolutional networks with non-overlapping kernels trained on MNIST and CIFAR-10. Samples where the mean is used to invert the SVD in the inverse pass (left). Samples generated by drawing from the inverse density to invert the SVD in the inverse pass (right).

of parameters that are used in these approaches, the restricted subspace of the models, or some combination of these three.

It can also be shown that convolutions with the same number of input and output channels can be made invertible [12]. These layers perform poorly at the task of density estimation and are outperformed by flowified layers. This is likely due to the increased expressivity that comes from considering a larger space of architectures. There have been several works that develop convolution inspired invertible transformations [27–29], but these architectures consider restricted transformations to maintain invertibility.

6 Future Work and Conlusion

Our experiments suggest that flowified convolutional networks do not match the density estimation performance of similarly sized normalizing flows. A possible explanation is that the dimension reducing steps discard information that is still structured and more expressive layers are necessary to transform the distributions before reducing the dimensionality. Possibly the main limitation is the fact that unlike flow layers the forward passes of standard linear and convolutional layers are not data-dependent. This is reinforced by the fact that the entanglement capability typically used in flows also appears in attention mechanisms, which have been shown to excel at capturing complex statistical structures [30–32].

With further development – such as increased capacity given to the inverse density, or data dependent parameters in the forward pass – standard architectures could become good density estimators in their own right and allow for general purpose models to be developed. The focus of this work was on using standard layers for density estimation, but it is possible that designing data dependent variants of standard layers that are more flow-like could improve their performance on tasks such as classification and regression. The flowification procedure provides a useful means for designing such models, and demonstrates that standard architectures can be considered a subset of normalizing flows, a correspondence that has not previously been demonstrated.

References

[1] Esteban G Tabak and Cristina V Turner. A family of nonparametric density estimation algorithms. *Communications on Pure and Applied Mathematics*, 66(2):145–164, 2013.

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

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

[4] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows, 2019.

[5] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. *arXiv preprint arXiv:1705.07057*, 2017.

[6] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference, 2021.

[7] Eric Nalisnick, Akihiro Matsukawa, Yee Whye Teh, Dilan Gorur, and Balaji Lakshminarayanan. Do deep generative models know what they don't know?, 2019.

[8] Claudius Krause and David Shih. Caloflow: Fast and accurate generation of calorimeter showers with normalizing flows, 2021.

[9] Jens Behrmann, Will Grathwohl, Ricky T. Q. Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. 2018. doi: 10.48550/ARXIV.1811.00995. URL `https://arxiv.org/abs/1811.00995`.

[10] Ricky T. Q. Chen, Jens Behrmann, David Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling, 2019. URL `https://arxiv.org/abs/1906.02735`.

[11] Yura Perugachi-Diaz, Jakub M. Tomczak, and Sandjai Bhulai. Invertible densenets, 2020. URL `https://arxiv.org/abs/2010.02125`.

[12] Marc Finzi, Pavel Izmailov, Wesley Maddox, Polina Kirichenko, and Andrew Gordon Wilson. Invertible convolutional networks. In *Workshop on Invertible Neural Nets and Normalizing Flows, International Conference on Machine Learning*, 2019.

[13] Chin-Wei Huang, Laurent Dinh, and Aaron Courville. Augmented normalizing flows: Bridging the gap between generative flows and latent variable models, 2020.

[14] Samuel Klein, John A Raine, Sebastian Pina-Otey, Slava Voloshynovskiy, and Tobias Golling. Funnels: Exact maximum likelihood with dimensionality reduction. *arXiv preprint arXiv:2112.08069*, 2021.

[15] Didrik Nielsen, Priyank Jaini, Emiel Hoogeboom, Ole Winther, and Max Welling. Survae flows: Surjections to bridge the gap between vaes and flows. *Advances in Neural Information Processing Systems*, 33:12685–12696, 2020.

[16] Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improving variational inference with inverse autoregressive flow, 2016. URL `https://arxiv.org/abs/1606.04934`.

[17] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation, 2017. URL `https://arxiv.org/abs/1705.07057`.

[18] Johann Brehmer and Kyle Cranmer. Flows for simultaneous manifold learning and density estimation, 2020. URL `https://arxiv.org/abs/2003.13913`.

[19] Edmond Cunningham and Madalina Fiterau. A change of variables method for rectangular matrix-vector products. In Arindam Banerjee and Kenji Fukumizu, editors, *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 2755–2763. PMLR, 13–15 Apr 2021. URL https://proceedings.mlr.press/v130/cunningham21a.html.

[20] Anthony L. Caterini, Gabriel Loaiza-Ganem, Geoff Pleiss, and John P. Cunningham. Rectangular flows for manifold learning, 2021. URL https://arxiv.org/abs/2106.01413.

[21] Brendan Leigh Ross and Jesse C. Cresswell. Tractable density estimation on learned manifolds with conformal embedding flows, 2021. URL https://arxiv.org/abs/2106.05275.

[22] Benigno Uria, Iain Murray, and Hugo Larochelle. Rnade: The real-valued neural autoregressive density-estimator. 2013. doi: 10.48550/ARXIV.1306.0186. URL https://arxiv.org/abs/1306.0186.

[23] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[24] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto, 2009.

[25] Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design, 2019. URL https://arxiv.org/abs/1902.00275.

[26] Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models, 2015. URL https://arxiv.org/abs/1511.01844.

[27] Mahdi Karami, Dale Schuurmans, Jascha Sohl-Dickstein, Laurent Dinh, and Daniel Duckworth. Invertible convolutional flow. *Advances in Neural Information Processing Systems*, 32, 2019.

[28] Emiel Hoogeboom, Rianne van den Berg, and Max Welling. Emerging convolutions for generative normalizing flows, 2019.

[29] Emiel Hoogeboom, Victor Garcia Satorras, Jakub M. Tomczak, and Max Welling. The convolution exponential and generalized sylvester flows, 2020. URL https://arxiv.org/abs/2006.01910.

[30] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[31] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

[32] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020. URL https://arxiv.org/abs/2010.11929.

[33] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017. URL https://arxiv.org/abs/1912.01703.

[34] Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

[35] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. nflows: normalizing flows in PyTorch, November 2020. URL `https://doi.org/10.5281/zenodo.4296287`.

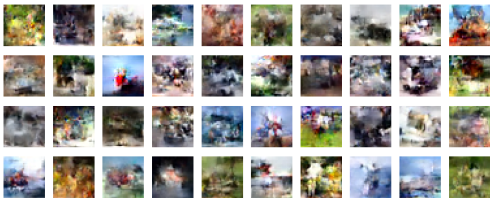
[36] B. Hall. *Lie Groups, Lie Algebras, and Representations: An Elementary Introduction*. Graduate Texts in Mathematics. Springer International Publishing, 2015.

Checklist

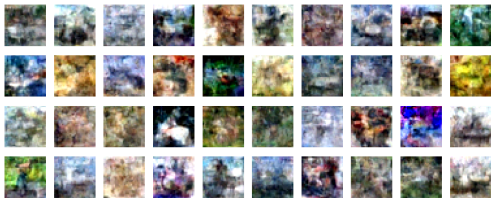
1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] **See 4 6**
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] **See 3 2**
   (b) Did you include complete proofs of all theoretical results? [No] **We provided sketches of the proof, and as the results we present depend on well known results we think further details are unnecessary and don't help with the clarity of the paper.**
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] **The script used to run the experiments has been included and the instructions can be found in the code.**
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] **This is not typically done on the benchmarks we considered.**
   (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] **See A.**
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 A.**
   (b) Did you mention the license of the assets? [Yes] **See A.**
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] **The tools we develop are contained in the paper, the code is included but is not a new library that should be widely adopted.**
   (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 Flowification + NSF

Below are samples from an architecture combining flowified standard layers and NSF layers.



(a) Combination of flowified convolutions with non-overlapping convolutions and NSF layers, BPD 3.65



(b) Combination of flowified convolutions with overlapping convolutions and NSF layers, BPD 6.12

Experimental details

The models on CIFAR-10 were trained for 600 epochs with a batch size of 256 and lasted 16-20 hours. The models on MNIST were trained for 200 epochs with the same batch size and lasted approximately 2 hours.

Everything was implemented in PyTorch [33] and executed on a RTX 3090 GPU.

We also used weights and biases for experiment tracking [34] and the neural spline flows [35] package for the rational quadratic activation functions. Both of these tools are available under an MIT license.

B Parametrizing rotations via the Lie algebra $\mathfrak{so}(d)$

The Lie algebra $\mathfrak{so}(d)$ of the rotation group $SO(d)$ is the space of $d \times d$ skew-symmetric matrices

$$\mathfrak{so}(d) = \{X \in M_d(\mathbb{R}) | X = -X^T\}$$

Moreover, since $SO(d)$ is connected and compact, the (matrix-)exponential map $\mathfrak{so}(d) \to SO(d)$ is surjective [36, Corollary 11.10], i.e. every rotation $U \in SO(d)$ can be written as the exponential of some skew-symmetric $X \in \mathfrak{so}(d)$.

Motivated by this result, we propose to parametrize the Lie algebra $\mathfrak{so}(d)$, and use the exponential map to obtain rotations. Since the exponential map is differentiable, we can propagate gradients all the way back to the Lie algebra and do gradient based optimization there.

Note that is significantly eases the optimization process. The reason for this is that the Lie group of unitary transformations has non-trivial geometry and doing gradient descent is cumbersome on such spaces. On the other hand, the Lie algebra is a vector space, where gradient descent can shine.

Since PyTorch [33] has built-in support for the matrix exponential, random orthogonal transformations can be generated with just a few lines of Python code:

```
d  = 5                          # size of matrix
x  = torch.randn(d,d)           # random (d,d)  matrix
S  = x - x.T()                  # S is skew-symmetric
U  = torch.matrix_exp(S)        # U  is a rotation
```

Figure 6: Generating a random rotation in PyTorch

C Architectures

```
import torch.nn as nn
import flowification as ffn

model = nn.Sequential(
  ffn.Conv2D(in_c = 3, out_c = 27, kernel_size = 3, stride = 2), # 16
  ffn.ffn.RqSpline(),
  ffn.Conv2D(in_c = 27, out_c = 48, kernel_size = 3, stride = 2), # 8
  ffn.RqSpline(),
  ffn.Conv2D(in_c = 48, out_c = 72, kernel_size = 3, stride = 2), # 4
  ffn.RqSpline(),
  ffn.Conv2D(in_c = 72, out_c = 96, kernel_size = 3, stride = 2), # 2
  ffn.RqSpline(),
  ffn.Conv2D(in_c = 96, out_c = 128, kernel_size = 2, stride = 2), # 1
  ffn.RqSpline(),
  ffn.Flatten(),
  ffn.Linear(128, 128), ffn.RqSpline(),
  ffn.Linear(128, 128), ffn.RqSpline(),
  ffn.Linear(128, 128), ffn.RqSpline(),
  ffn.Linear(128, 128), ffn.RqSpline(),
  ffn.Linear(128, 128), ffn.RqSpline(),
  ffn.Linear(128, 128), ffn.RqSpline(),
    )

```

Figure 7: Architecture for CIFAR10

```
import torch.nn as nn
import flowification as ffn

model = nn.Sequential(
  ffn.Conv2D(in_c = 1, out_c = 4, kernel_size = 3, stride = 2),  # 14
  ffn.RqSpline(),
  ffn.Conv2D(in_c = 4, out_c = 8, kernel_size = 3, stride = 2),  # 7
  ffn.RqSpline(),
  ffn.Conv2D(in_c = 8, out_c = 12, kernel_size = 3, stride = 2), # 3
  ffn.RqSpline(),
  ffn.Flatten(),
  ffn.Linear(108, 64),
  ffn.Linear(64, 32),
  ffn.Linear(32, 16),
  ffn.Linear(16, 8),
    )

```

Figure 8: Architecture for MNIST

D Flowifying convolutions through the FFT

An alternative way of flowifing convolutional layers is through the convolution theorem which states that the Fourier-transformation diagonalizes convolutions. In this section we sketch the idea and discuss the issues with it.

Fig.9 displays the Pytorch code for performing convolutions through the convolution theorem. Line 12 of the code shows that in the Fourier basis, we can perform convolution as follows; at each frequency we form the vector of dimension (# of in_channels) and apply a weight matrix of size (# of in_channels) $\times$ (# of out_channels) to it. If this weight matrix is parametrized by the SVD, we can

guarantee its invertibility. Since the FFT is also invertible, this construction should be a flowification of the convolutional layer if we can also solve the following two issues:

If the weight matrix is parametrized in frequency domain,

(1) it's inverse transform might not be real.

(2) it's inverse transform might not be of the appropriate size (i.e. nonzero outside of the first (kernel_size)-many pixels).

The first of these is easy to solve with the symmetric parametrization induced by

$$k \text{ is real} \Leftrightarrow FFT(k)_i^* = FFT(k)_{N-i}$$

where N is the length of the signal. To deal with the second condition, a auxiliary loss term is introduced that minimizes the terms that should be 0 . Although the idea of FFT seems promising, this approach turns out be slower (probably due to the FFT), worse performing and more cumbersome than the flowificaton presented in §3.2. Fig. 10 shows some samples from a Fourier-flowified convolutional network trained on MNIST.

```
in_channels, out_channels = 4, 3
signal_length, kernel_size = 32, 5
batch_size = 24

K = torch.randn(out_channels, in_channels, kernel_size)
x = torch.randn(batch_size, in_channels, signal_length)
z1 = torch.nn.functional.conv1d(input=x, weight=K)

x_ = torch.fft.fft(x, norm='ortho')
# pad kernel to match the length of x
K_ = torch.fft.fft(K.flip(-1), n=x.size()[-1], norm='ortho')
z_ = torch.einsum('bix,jix->bjx', (x_, K_))
z2 = torch.fft.ifft(z_, norm='forward')
z2 = z2[:, :, (kernel_size-1):]

print((z1-z2).abs().max().item())
# tensor(2.8610e-06)

```

Figure 9: 1D convolutions in PyTorch

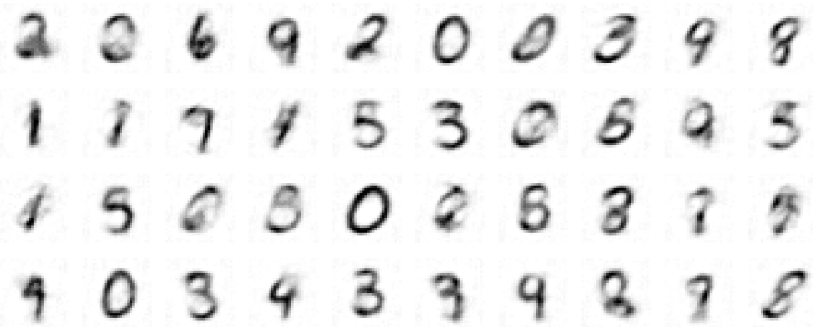


Figure 10: Samples from and Fourier-flowified convolutional network