Distilled Gaussianization

James Theiler and Christopher X. Ren

Space Data Science and Systems Group, Intelligence and Space Research Division
Los Alamos National Laboratory, Los Alamos, NM 87545

ABSTRACT

Gaussianization is a recently suggested approach for density estimation from data drawn from an unknown, presumably non-Gaussian, and possibly high dimensional, distribution. The key idea is to learn a transformation that, when applied to the data, leads to an approximately Gaussian distribution. The density, for any given point in the original distribution, is then given by the determinant of the transformation's Jacobian at that point, multiplied by the (analytically known) density of the Gaussian for the transformed data. In this work, we investigate variants of the rotation-based iterative Gaussianization (RBIG) algorithm that lead to simpler (but more) iterations; these include the use of a few-segment piecewise-linear squashing function and fractional iterations. We consider the use of distilled machine learning to provide a compact implementation of the many iterations of this Gaussianization transform, potentially enabling faster computation, better controlled regularization, and more direct estimation of the Jacobian.

Keywords: machine learning, Gaussianization, density estimation, neural netwok, cycle consistency

> *"Many insects have a larval form that is optimized for extracting energy and nutrients from the environment and a completely different adult form that is optimized for the very different requirements of traveling and reproduction. In large-scale machine learning, we typically use very similar models for the training stage and the deployment stage despite their very different requirements..."* —Hinton, Vinyals, and Dean [1]

1. INTRODUCTION

Much of data science boils down to the problem of estimating properties of an underlying probability distribution function, using a (typically large but) finite set of samples drawn from that distribution. That is, given a set $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, with $\mathbf{x}_n \in \mathbb{R}^d$ drawn from an unknown distribution $p_{\mathbf{x}}(\mathbf{x})$, what can these samples tell you about $p_{\mathbf{x}}(\mathbf{x})$? Note that there are many tasks for which an estimate of the distribution function is an intermediate step. Our motivating interest is in multispectral remote sensing, and characterization of the background distribution is crucial for many tasks in that field,[2] including:

- Anomaly detection: determine which samples $\mathbf{x}_n$ are the least consistent with the underlying distribution; that is, which $\mathbf{x}_n$ exhibit the largest values of $1/p_{\mathbf{x}}(\mathbf{x}_n)$.
- Anomalous change detection: from a set of pairs $(\mathbf{x}_n, \mathbf{y}_n)$, find the pairs for which $p_{\mathbf{x}}(\mathbf{x}_n)p_{\mathbf{y}}(\mathbf{y}_n)/p_{\mathbf{x},\mathbf{y}}(\mathbf{x}_n, \mathbf{y}_n)$ is largest.
- Target detection: Let $\boldsymbol{\xi}(\mathbf{x})$ be the effect of having a target present in a pixel $\mathbf{x}$. Then the optimal detector is given by the likelihood ratio:

$$\mathcal{L}(\mathbf{x}) = \frac{p_{\text{target}}(\mathbf{x})}{p_{\text{background}}(\mathbf{x})} = \frac{p_{\mathbf{x}}(\boldsymbol{\xi}^{-1}(\mathbf{x}))}{p_{\mathbf{x}}(\mathbf{x})} \left|\frac{d\boldsymbol{\xi}}{d\mathbf{x}}\right|. \tag{1}$$

  For an additive target, for example: $\boldsymbol{\xi}(\mathbf{x}) = \mathbf{x} + a\mathbf{t}$, where a is the strength of the target and $\mathbf{t}$ is the target signature. For this case, $|d\boldsymbol{\xi}/d\mathbf{x}| = 1$ and $\mathcal{L}(\mathbf{x}) = p_{\mathbf{x}}(\mathbf{x} - a\mathbf{t})/p_{\mathbf{x}}(\mathbf{x})$.

1.1 Gaussianization

Gaussianization[3] provides a way to recast the problem of estimating a distribution into the problem of estimating a function. The function we seek maps the data into a space where the distribution is Gaussian. That is, we seek an invertible function $T : \mathbb{R}^d \to \mathbb{R}^d$ so that if $\mathbf{x} \sim p$, and $\mathbf{y} = T(\mathbf{x})$, then $y \sim \mathcal{G}$, where $\mathcal{G}(\mathbf{y}) = (2\pi)^{-d/2} e^{-\|\mathbf{y}\|^2/2}$ is a Gaussian with zero mean and unit covariance. Given such a function T, we can explicitly estimate the original distribution by

$$p_T(\mathbf{x}) = \mathcal{G}(\mathbf{y}) \left| \frac{d\mathbf{y}}{d\mathbf{x}} \right| = \mathcal{G}(T(\mathbf{x})) \left| \frac{dT}{d\mathbf{x}} \right| \tag{2}$$

where $|dT/d\mathbf{x}|$ is the determinant of the $d \times d$ Jacobian matrix. The more accurately we estimate the Gaussianizing function, the better $p_T(\mathbf{x})$ will approximate $p(\mathbf{x})$.

Because T is invertible, we can also use this transform to generate new samples from the distribution p_T, by taking Gaussian samples $\mathbf{y}$ and applying the inverse function: $\mathbf{x} = T^{-1}(\mathbf{y})$.

1.1.1 The $d = 1$ case produces a "squashing function"

Obtaining a Gaussianizing transform in one dimension is relatively straightforward. If $f(x)$ is an estimate of the one-dimensional density, then $F(x) = \int_{-\infty}^{x} f(z)\, dz$ is the corresponding *cumulative* density function (CDF). If we write $\Phi(x)$ as the CDF for a Gaussian, then $\Phi^{-1}F$ will be a monotonically increasing (ergo invertible) function that transforms the data so that the set $\{\Phi^{-1}(F(x_1)), \ldots, \Phi^{-1}(F(x_N))\}$ exhibits a Gaussian distribution.

Fitting the CDF to data is relatively straightforward; given points $\{x_1, \ldots, x_N\}$, we can sort the points into a set $\{\tilde{x}_1, \ldots, \tilde{x}_N\}$, with $\tilde{x}_1 \leq ... \leq \tilde{x}_N$ and then associate $F(\tilde{x}_n) = \tilde{a}_n$, where $\tilde{a}_n = (n - \frac{1}{2})/N$ uniformly fills the interval $[0, 1]$. (We could also take $\tilde{a}_n$ to be the n'th value in a sorted list of N numbers that are randomly drawn from a uniform distribution on $[0, 1]$.)

This fit can be done in a nonparametric way that ensures $y_n = \Phi^{-1}(F(x_n))$, for all n; or in a parametric way the fits (x_n, y_n) to a simpler functional form. If the aim is to get the best possible Gaussian in this one single step, then a nonparametric approach may be called for. But if this is part of an iterative process, a simple form will still make the data incrementally more Gaussian, and further iterations will continue improving the Gaussianization. An advantage of the simpler form is that it is easier to to take inverses and derivatives, and there is less potential for overfitting the data.

1.1.2 The $d > 1$ case: multivariate Gaussianization

For d-dimensional Gaussianization, we can get a first approximation with a *marginal* Gaussianization; this is obtained by separately Gaussianizing each of the d components. If the initial distribution were equal to the product of its marginal distributions – *i.e.*, if $p_{\mathbf{x}}(\mathbf{x}) = p_1(x^{(1)}) \cdots p_d(x^{(d)})$, where $x^{(i)}$ denotes the ith component of $\mathbf{x}$ – then this first approximation would actually be a good d-dimensional Gaussianization.

In rotation-based iterative Gaussianization (RBIG), first proposed by Chen and Gopinath,[3] each iteration consists of a rotation of the data followed by marginal Gaussianization. It was further proposed that this choice of rotation be based on maximizing the non-Gaussian-ness of the projections. In Laparra *et al.*,[4] however, it was shown that random rotations could also be effective, and that this choice has a number of advantages: it is simple to implement, it avoids local minima, and it is found to converge nearly as fast as well-engineered rotations.

The basic structure of the RBIG algorithm is shown in Algorithm 1. Two unspecified aspects of the algorithm are the method for choosing rotation matrices (Line 3), and the method for obtaining the squashing functions (Line 7). For all of our experiments, we used random rotations for Line 3, but (as we will later describe) we considered several different design choices for the squashing in Line 7. The output of the RBIG algorithm is the collection of rotation matrices R_m and squashing functions H_m; from these we can define the full transform as the nested application of these functions. Let $T_m(\mathbf{x}) = H_m(R_m\mathbf{x})$; then

$$T(\mathbf{x}) = T_M(\cdots T_2(T_1(\mathbf{x}))\cdots). \tag{3}$$

The option to "rotate back" after each iteration (Line 12) does not improve the performance of the Gaussianization, but it makes visualizations of the iterative process easier to watch because there is less "spinning around." In this case, each step of the transformation is given by $T_m(\mathbf{x}) = R_m^{\mathsf{T}} H_m(R_m\mathbf{x})$.

Algorithm 1 Rotation-Based Iterative Gaussianization (standard implementation)

Require: $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, with $\mathbf{x}_n \in \mathbb{R}^d$
Initialize $\mathbf{y}_n \leftarrow \mathbf{x}_n$, for all $n = 1, \ldots, N$
for $m = 1, 2, \ldots, M$ **do** ▷ Iterate until $\mathbf{y}_n$'s are, by some measure, sufficiently Gaussian
  $R_m \leftarrow$ GetRotationMatrix$(\{\mathbf{y}_1, \ldots, \mathbf{y}_N\})$
  Rotate Data: $\mathbf{y}_n \leftarrow R_m \mathbf{y}_n$, for all n
  for $k = 1, \ldots, d$ **do** ▷ Each component k will be individually Gaussianized
    Let $z_n \leftarrow \mathbf{y}_n^{(k)}$ be kth component of $\mathbf{y}_n$, for all n
    $H_{mk} \leftarrow$ GetSquashingFunction$(\{z_1, \ldots, z_N\})$
    Let $z'_n \leftarrow H_{mk}(z_n)$, for all n
    Let $z'_n \leftarrow z'_n$, for all n ▷ Do nothing for now; later we will change this line
    Let $\mathbf{y}_n^{(k)} \leftarrow z'_n$ reset the k'th component of $\mathbf{y}_n$, for all n
  Define $H_m(\mathbf{y}) = [H_{m1}(\mathbf{y}^{(1)}), \ldots, H_{md}(\mathbf{y}^{(d)})]^\mathsf{T}$ ▷ Merge component-wise H_{mk} into a single function H_m
  (Optionally) Rotate Data Back: $\mathbf{y}_n \leftarrow R_m^\mathsf{T} \mathbf{y}_n$
Return: R_m, H_m for $m = 1, 2, \ldots, M$

function GetRotationMatrix$(\{\mathbf{y}_1, \ldots, \mathbf{y}_N\})$
  $R = \cdots$ ▷ Can use PCA, ICA, or just random rotation; note $R^\mathsf{T} R = R R^\mathsf{T} = I$
  return R

function GetSquashingFunction$(\{z_1, \ldots, z_N\})$
  Let $F(z)$ model the CDF of the set $\{z_1, \ldots, z_N\}$
  Define scalar function $h(z) = \Phi^{-1}(F(z))$
  return h

1.1.3 Flow loss

The quality of a Gaussianization function is usually measured in terms of the *negentropy*, which corresponds to the Kullback-Liebler (KL) divergence of the Gaussianized distribution to an actual Gaussian. Because the KL divergence is invariant to invertible transforms, this is also the KL divergence between the actual unknown distribution and the estimated distribution given by Eq. (2). Since this latter distribution is also what we most directly care about, we can write

$$D_{KL}(p \| p_T) = \int p(\mathbf{x}) \log \left(\frac{p(\mathbf{x})}{p_T(\mathbf{x})} \right) d\mathbf{x} \tag{4}$$

$$= \int p(\mathbf{x}) \log p(\mathbf{x}) \, d\mathbf{x} - \int p(\mathbf{x}) \log p_T(\mathbf{x}) \, d\mathbf{x} \tag{5}$$

$$= -H_p - \int p(\mathbf{x}) \left[\log \mathcal{G}(T(\mathbf{x})) + \log \left| \frac{dT}{d\mathbf{x}} \right| \right] d\mathbf{x} \tag{6}$$

$$= -H_p + (d/2) \log(2\pi) - \int p(\mathbf{x}) \left[-\|T(\mathbf{x})\|^2/2 + \log \left| \frac{dT}{d\mathbf{x}} \right| \right] d\mathbf{x}. \tag{7}$$

Here, $H_p = -\int p(\mathbf{x}) \log p(\mathbf{x}) \, d\mathbf{x}$ is the entropy of the unknown distribution. Although this value is unknown, it is constant. Since we have a finite sample of points $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ drawn from $p(\mathbf{x})$, we can approximate the

integral with a sum. Write

$$L_T = \int p(\mathbf{x}) \left[\|T(\mathbf{x})\|^2/2 - \log \left| \frac{dT}{d\mathbf{x}} \right| \right] d\mathbf{x} \tag{8}$$

$$\approx \frac{1}{N} \sum_n \left[\|T(\mathbf{x}_n)\|^2/2 - \log \left| \frac{dT}{d\mathbf{x}_n} \right| \right] \tag{9}$$

$$= \frac{1}{N} \sum_n \left[\|\mathbf{y}_n\|^2/2 - \log J(\mathbf{x}_n) \right] \tag{10}$$

as the "flow loss" associated with a Gaussianizing transform T. Here, $\mathbf{y}_n = T(\mathbf{x}_n)$ and $J(\mathbf{x}_n)$ is the determinant of the Jacobian of T at $\mathbf{x}_n$. Since our aim is to minimize the KL divergence in Eq. (4), that is achieved by minimizing the flow loss in Eq. (10). An appealing feature of this formulation is that each term in Eq. (10) depends on a single $(\mathbf{x}_n, \mathbf{y}_n)$ pair; this provides a way of evaluating the quality of T from even a single sample. In the Gaussian flows proposed by Meng *et al.*,[5] a parametric functional form for T is posited, and those parameters are then optimized to minimize the flow loss.

In this formulation, we do not need to know the entropy of the initial data distribution p, but a re-arrangement of the expressions above leads to

$$H_p = (d/2)\log(2\pi) + \min_T L_T \tag{11}$$

which illustrates how we can use the Gaussianization process to estimate the entropy of a distribution. In a recent pre-print,[6] Laparra *et al.* show how a number of information theoretic measures can be obtained using Gaussianization.

In our work, we will use the flow loss to compare different approaches to Gaussianization.

1.2 Distilled Machine Learning

One of the early ideas for distilling, initially called compressing,[7] was to replicate the behavior of large ensembles of learning algorithms with a single – and computationally cheaper – implementation. It was also found that the output of large neural networks could be replicated in smaller networks.[1] These smaller, distilled learners are not trained with the original training data, however, but with a modified training set that is based on the output of the original, larger learning algorithm. Their task is not to find the structure hidden in the data, but simply to duplicate the output of the algorithm that already did that. Evidently, more neurons are required for *learning* than for *remembering*.

The analogy is loose here, but our idea is to let RBIG use lots of iterations to learn the Gaussianizing transform $T(\mathbf{x})$, and then to train a small regressor (we'll end up using a neural network) to replicate this function.

2. ITERATIVE GAUSSIANIZATION WITH MANY ITERATIONS

Our implementation of RBIG has two innovations: 1/ a simpler marginal Gaussianizer – three segment piecewise linear estimation – that enables fast fitting and easy invertability; and 2/ fractional iterations, which enables a stabler convergence, and often leads to lower flow loss values. Both of these innovations lead to deliberately underfit models at each iteration, and a consequence is that more iterations are required to achieve good Gaussianization.

2.1 Piecewise linear squashing functions

The usual prescription for the one-dimensional function that Gaussianizes in a marginal direction is $h(z) = \Phi^{-1}(F(z))$ where F is the cumulative distribution function (CDF) of the estimated 1-d density. (See Line 16 in Algorithm 1.)

Our approach is to simplify this whole process and express $h(x)$ as a simple three-segment piecewise linear function. We constrain the slope to be positive for all three segments; this ensures that the function is monotonically increasing, and that it is invertible. The fit is a simple minimization of mean squared error, based on (x, y) pairs produced as described in Algorithm 2. The GetSquashingFunction function defined in Algorithm 2 can be used as a drop-in replacement for the function of the same name in the RBIG Algorithm 1.

Algorithm 2 Piecewise linear squashing functions

function GetSquashingFunction($\{z_1, \ldots, z_N\}$)
  Let $y_1, \ldots, y_N \leftarrow$ SquashData($z_1, \ldots, z_N$)
  Fit function $y = h(z)$ to data samples $(z_1, y_1), \ldots, (z_N, y_N)$ ▷ With $h(x)$ parameterized ▷ as a piecewise linear function
  return h

function SquashData($z_1, \ldots, z_N$)
  Let $r_1, \ldots, r_N \leftarrow$ ArgSort($\{z_1, \ldots, z_N\}$) ▷ So that: $z_{r_1} \leq z_{r_2} \leq \cdots \leq z_{r_N}$
  Let $y_{r_n} \leftarrow \Phi^{-1}\left(\frac{r_n - 1/2}{N}\right)$ ▷ Note y_n's are distributed as a Gaussian
  return $y_1, \ldots, y_n$

2.2 Fractional iterations

The motivation for fractional iterations is to slow down the convergence, and (we hope) thereby make it more stable. The fractional iteration approach is less dependent on the particular choice of the rotations $R_1, \ldots, R_M$ by making each individual rotation have a smaller effect. Algorithmically, this is achieved by replacing Line 9 in Algorithm 1 with a line of the form:

$$9: \quad \text{Let } z'_n \leftarrow (1-f)z_n + f z'_n, \text{ for all } n.$$

Equivalently, we can think of this as fractional squashing, with a line like $h(z) \leftarrow (1-f)z + f h(z)$ introduced into the GetSquashingFunction() call. Note that $f = 1$ reverts to the standard implementation, but we find that $f < 1$ often enables convergence to better solutions with less tendency to over-fit the dataset. (We will discuss this further in Section 4, and in particular in Fig. 3.) Note that in the limit as $f \to 0$ (and as $M \to \infty$), the discrete iterative map becomes a differential equation. Indeed, several authors[8,9] have investigated the idea of recasting the Gaussianization process as a continuous flow.

3. DISTILLED GAUSSIANIZATION

A downside to iterative Gaussianization in general, and to our many-iterates approach in particular, is that evaluation of T (which is a scalar) and especially of $dT/d\mathbf{x}$ (which is a $d \times d$ matrix) requires a lot of computation. A disadvantage of the piecewise linear aspect of our approach is that the estimated $p_T(\mathbf{x})$ often ends up having linear artifacts (a kind of "wrinkled" appearance). These are mitigated by the fractional iteration approach (many smaller wrinkles instead of a few larger ones), but smoother estimates of the underlying distributions will require smoother functions T. Distillation addresses both of these issues.

The similarity of RBIG to a deep neural network has been noted previously; for example, Laparra *et al.*[4] write that:

> "RBIG is essentially an iterated sequence of two operations, nonlinear dimension-wise squashing functions and linear transforms. Intuitively, these are the same processing blocks used in a feedforward neural network (linear transform plus sigmoid-shaped function in each hidden layer). Therefore, one could see each iteration as one hidden layer processing of the data, and thus argue that complex (highly non-Gaussian) tasks should require more hidden layers (iterations). This view is in line with the field of deep learning in neural networks..."

While this is an intriguing interpretation, we note that the RBIG-as-neural-network is a neural network with a peculiar structure. It is a deep network with many many layers, but one for which each layer has exactly d nodes, and one for which the weights between the layers are not trained. While most neural networks have an activation function that is specified in advance, the RBIG-as-network has a different specially fit activation (a.k.a. squashing) function at each node. With a distilled neural network, we can replace this architecture with one that is potentially more conducive to training. We can use a neural network with fewer layers, more nodes per layer, and standard activation functions.

The distillation idea is to take the Gaussianization transform that was produced by RBIG and to replicate it in a more compact regressor. We initially tried several regression approaches (such as kernel ridge regression), but the examples reported here were obtained with a neural network. Using RBIG, we create data pairs $(\mathbf{x}, \mathbf{y})$ where $\mathbf{x}$ is the data drawn from the original distribution, and $\mathbf{y}$ is the Gaussianized data. To these data pairs, we fit a function $T(\mathbf{x})$ to minimize some measure of $\mathbf{y} - T(\mathbf{x})$ (we use mean squared error). Potential advantages of distillation include:

- $T(\mathbf{x})$ may be cheaper to evaluate than the iterative RBIG, which involves many iterations of rotation and squashing.
- $T(\mathbf{x})$ may have a functional form that enables direct differentiation, for computing the Jacobian $dT/d\mathbf{x}$.
- $T(\mathbf{x})$ may have a functional form that is smoother than the piecewise linear components of the RBIG solution.
- If the regression process includes some regularization, then the overfitting that might occur in the RBIG process can be ameliorated.
- $T^{-1}(\mathbf{x})$ can be directly estimated by doing a distillation with $(\mathbf{y}, \mathbf{x})$ data pairs.

3.1 Distilling "plain" RBIG

To be useful (*i.e.*, to be used either for density estimation or for sample generation), Gaussianization has to provide a functional form $T(\mathbf{x})$ that can be differentiated and that can be inverted. But if our aim is to use distilled machine learning to produce the actual functional form for $T(\mathbf{x})$, then all we need from iterative Gaussianization is a set of $(\mathbf{x}_n, \mathbf{y}_n)$ pairs, where the $\mathbf{x}_n$'s are from the original data set, and the $\mathbf{y}_n$'s are the associated Gaussian counterparts.

Given these more modest requirements, we can employ a more modest version of the Gaussianization algorithm. For this purpose, we propose a "plain RBIG" algorithm. In this plain variant, we do not need to keep track of all the rotations and squashing functions; we only need to follow the $\mathbf{y}_n$'s as they iterate towards a Gaussian distribution.

Because there is no need to keep track of all the rotations and squashing functions, *plain RBIG* is simple to code and fast to run. A second advantage is that the squashing function needn't be an actual function; it just needs to map the discrete set of points that is in the training set. Note that we can still employ fractional iterations with the plain method, in order to ameliorate overfitting.

Algorithm 3 Plain RBIG

Require: $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, with $\mathbf{x}_n \in \mathbb{R}^d$ ▷ Input data to be Gaussianized; drawn from $p(\mathbf{x})$
Require: $f \in [0, 1]$ ▷ Default $f = 1$; Use $f < 1$ for fractional iterations
Initialize $\mathbf{y}_n \leftarrow \mathbf{x}_n$, for all $n = 1, \ldots, N$
for $m = 1, 2, \ldots, M$ **do** ▷ Iterate until $\mathbf{y}_n$'s are, by some measure, sufficiently Gaussian
  $R \leftarrow$ GetRotationMatrix$(\{\mathbf{y}_1, \ldots, \mathbf{y}_N\})$ ▷ See Algorithm 1, Line 13
  Rotate Data: $\mathbf{y}_n \leftarrow R\mathbf{y}_n$, for all n
  for $k = 1, \ldots, d$ **do** ▷ Each component k will be individually Gaussianized
    Let $z_n \leftarrow \mathbf{y}_n^{(k)}$ be kth component of $\mathbf{y}_n$, for all n
    $z'_1, \ldots, z'_N \leftarrow$ SquashData$(z_1, \ldots, z_N)$ ▷ See Algorithm 2, Line 5
    $z'_n \leftarrow (1 - f)z_n + fz'_n$, for all n ▷ Fractional iteration; note that if $f = 1$, then this does nothing
    Let $\mathbf{y}_n^{(k)} \leftarrow z'_n$ reset the k'th component of $\mathbf{y}_n$, for all n
  (Optionally) Rotate Data Back: $\mathbf{y}_n \leftarrow R^{\mathsf{T}}\mathbf{y}_n$, for all n
Return: $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$

3.2 Cycle consistency

From the $(\mathbf{x}, \mathbf{y})$ pairs, we can separately train an approximator for $\mathbf{y} \approx T(\mathbf{x})$ and one for $\mathbf{x} \approx T^{-1}(\mathbf{y})$. Since we know that these are inverses of each other, however, we can incorporate that knowledge into the training process. We have implemented an optimization scheme that trains both the $T(\mathbf{x})$ (the Gaussianizer) and $T^{-1}(\mathbf{y})$ (the sampler) simultaneously, and includes an extra loss term to encourage cycle consistency.

3.2.1 Simple cycle consistency

In our first experiment, we employ a term of the form

$$(1/N) \sum_n \|\mathbf{x}_n - T^{-1}(T(\mathbf{x}_n))\|^2 \tag{12}$$

where the $\mathbf{x}_n$'s are from the training data. This term can be weighted relative to the $\|\mathbf{y}_n - T(\mathbf{x}_n)\|^2$ and $\|\mathbf{x}_n - T^{-1}(\mathbf{y}_n)\|^2$ terms, and for our experiments we just used a weight of one. (We found the the effect of the cycle consistency term did not depend strongly on the choice of this weight.)

3.2.2 Data-augmented cycle consistency

We don't have any "extra" training data from the unknown distribution, but we can generate as much Gaussian data as we like. To exploit this fact, our second experiment used a slightly different loss term

$$(1/K) \sum_k \|\mathbf{y}_k - T(T^{-1}(\mathbf{y}_k)\|^2 \tag{13}$$

where the $\mathbf{y}_k$ is Gaussian data that is randomly generated (indeed, re-generated at each training epoch). Because this extra training data exercises the functions T and T^{-1} at values outside the limited training data from the original Gaussianization, it has the potential to push back against the overfitting that might otherwise occur with small training sets.

4. NUMERICAL EXPERIMENTS

Although many of our motivating applications are in higher dimensions, our experiments in this study are two-dimensional.

Fig. 1 shows $N = 2000$ samples from a non-Gaussian two-dimensional distribution that looks like a fuzzy sine wave. Gaussianization provides a map $T : \mathbb{R}^2 \to \mathbb{R}^2$, which, when applied to the points in the initial distribution, produces points that are approximately Gaussian distributed. Based on the $T(\mathbf{x})$ function that was learned, we can create new samples, using $\mathbf{x} = T^{-1}(\mathbf{y})$ where $\mathbf{y}$ is drawn from a Gaussian distribution. This is shown in Fig. 2(a). Using the formula from Eq. (2), we can explicitly construct a density function, as shown in Fig. 2(b).

While Fig. 2 provides a kind of qualitative assessment of our estimation of the original distribution, Fig. 3 provides a more quantitative comparison, using the flow loss criterion from Eq. (10).

In this figure, flow loss is plotted as a function of iteration for the RBIG process, and a distinction is made between in-sample and out-of-sample performance. It is the out-of-sample behavior that characterizes the true performance of the RBIG process, because its operational application (to density estimation and to sample generation) correspond to points not seen in the training data. In particular, Fig. 3 uses flow loss performance to evaluate the effect of the fractional iteration parameter f. In Fig. 3(a), standard full ($f = 1$) iterations are employed, and we see that Gaussianization (as measured by flow loss) continues to improve for the in-sample points. For the out-of-sample points, however, we see rapid improvement for the first few iterations, and much slower improvement until about the 50th iteration. After that there is a plateau, but after about 120 iterations, the out-of-sample performance becomes much worse, and appears to become unstable. This is the effect of overfitting. By contrast, in Fig. 3(b,c), we use $f = 0.5$ and the out-of-sample flow loss is not only more stable than the $f = 1$ case, it achieves a lower (better) value. Using $f = 0.2$, as seen in Fig. 3(d), we achieve a lower value still.

Finally, Fig. 4 shows that further improvements in out-of-sample flow-loss are possible by using *distilled* Gaussianization. In Fig. 4(a), we use as training $(\mathbf{x}, \mathbf{y})$ values from the Gaussianization shown in Fig. 3(d). In this figure, we plot flow loss against training epoch and we see that after a few hundred epochs, the out-of-sample performance is significantly better for the distilled Gaussianizer than it was for the RBIG solution that provided the training data for the neural net. Where the RBIG solution involved 250 iterations (the equivalent, by some reckoning,[4] to a 250-layer network), the distilled network had five fully-connected layers, with 2, 10, 20, 10, and 2 nodes in each respective layer. We used the "softplus" activation function (basically a smoothed version of the rectified linear unit) in our PyTorch[10] implementation.

Distillation also provides an opportunity to employ the much faster and cheaper "plain RBIG" (described in Algorithm 3) for the initial training, and to use the $(\mathbf{x}, \mathbf{y})$ pairs from that process to train the neural net. Since plain RBIG does not provide an explicit $T(\mathbf{x})$, and therefore no way to compute Jacobians and therefore no way to compute flow loss, we cannot evaluate the plain process itself. But Fig. 4(b) shows that the distilled $T(\mathbf{x})$ exhibits a flow-loss performance that is comparable to that obtained in Fig. 4(a) using the modified RBIG with piecewise-linear squashing and fractional iterations. This is a very promising result, but a potential difficulty in practice might be the identification of stopping condition in the plain RBIG.

Since the distilled $T(\mathbf{x})$ is based on training from $(\mathbf{x}, \mathbf{y})$ pairs, the same process can be used for obtaining T^{-1} directly, by training from $(\mathbf{y}, \mathbf{x})$ pairs. In Fig. 5, we use the distilled T^{-1} functions to generate new samples from the original distribution.

Finally, in Fig. 6, we looked at the effect of incorporating a cycle consistency term into the training of the distilled Gaussianizers. Our first experiments indicated little effect; we think that is because the existing training regime was already near the optimal fit. So we reduced the training set size down to $N = 200$ samples. With fewer samples the RBIG performance was also decreased so smaller datasets were not only smaller but also of lower quality. Even with this more limited input data, we found the simple cycle consistency had little effect on flow loss performance. The data-augmented cycle consistency, however, did reduce the effects of overfitting, and produce a more stable flow loss curve. Even here, however, we did not observe significant gain in the best flow loss performance – that is to say: the out-of-sample curve in Fig. 6(d) is "flatter" than the ordinary distilled curve in Fig. 6(c), but its minimum value is not any smaller.

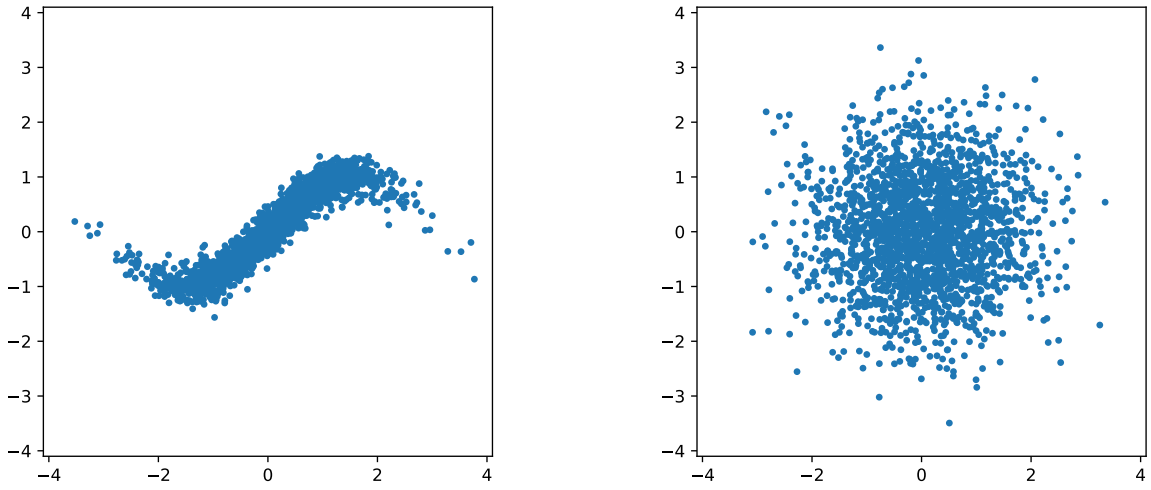
5. CONCLUSIONS

In traditional RBIG, a series of rotations, projections, and one-dimensional squashing functions are applied to a dataset in order to make its distribution more nearly Gaussian. This process is encapsulated in a transformation function $T(\mathbf{x})$, which can be used to infer the density function for the original data. In this work, we implemented a traditional RBIG algorithm, but with two modifications: one was to simplify the squashing function (using a three-segment piecewise linear function), and one was to "slow down" the process with fractional iterates.

One consequence of these modifications, however, is the need for more iterations, and that motivated the main idea here, which is: once RBIG has done its job, replicate the output of the learned $T(\mathbf{x})$ with a regression function. We tried several such regressors, and settled on a neural network. In addition to providing a more compact representation for $T(\mathbf{x})$, the network also provides some smoothing and a direct computation of the Jacobian.

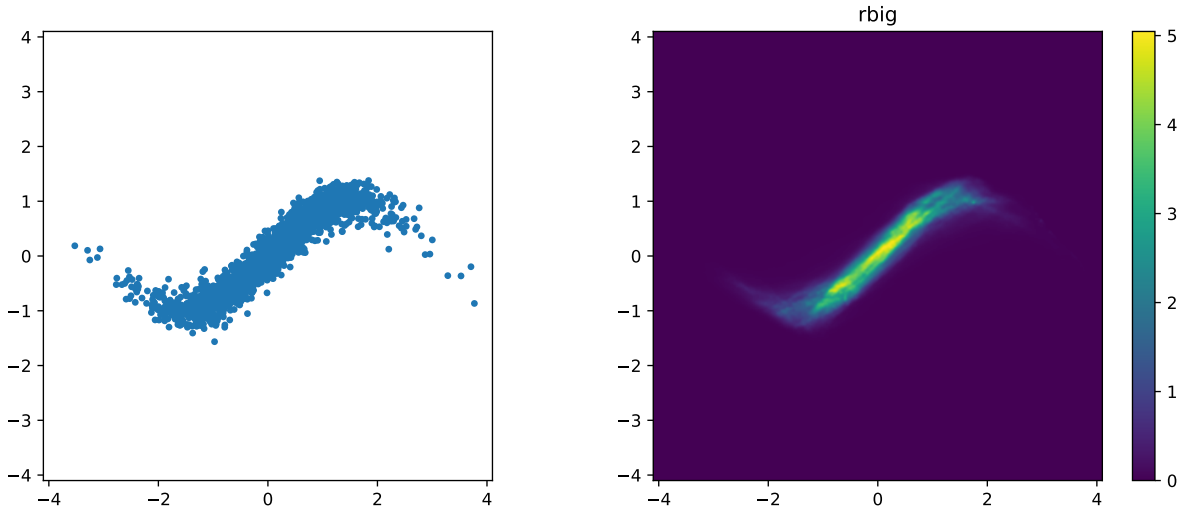
The examples shown in this paper were all two-dimensional. This obviously appealing, in terms of both visualization and general simplicity, but many of the real applications of interest, particularly multi- and even hyper-spectral remote sensing imagery, will require higher dimensional experiments. This will of course be challenging, but one unanswered question is whether even partial Gaussianization will be better than the usual approach, which is to simply *assume* that the distribution is Gaussian to begin with.

One very speculative idea is based on the observation that computing the Jacobian $dT/d\mathbf{x}$, and taking its determinant, is a process that becomes increasingly expensive as the dimension increases. But although the Jacobian is a $d \times d$ matrix, its determinant is a scalar, and that invites the possibility of training a separate network to estimate the determinant of Jacobian as a function of $\mathbf{x}$, rather than have to propagate the full Jacobian through the network for every data sample $\mathbf{x}$.



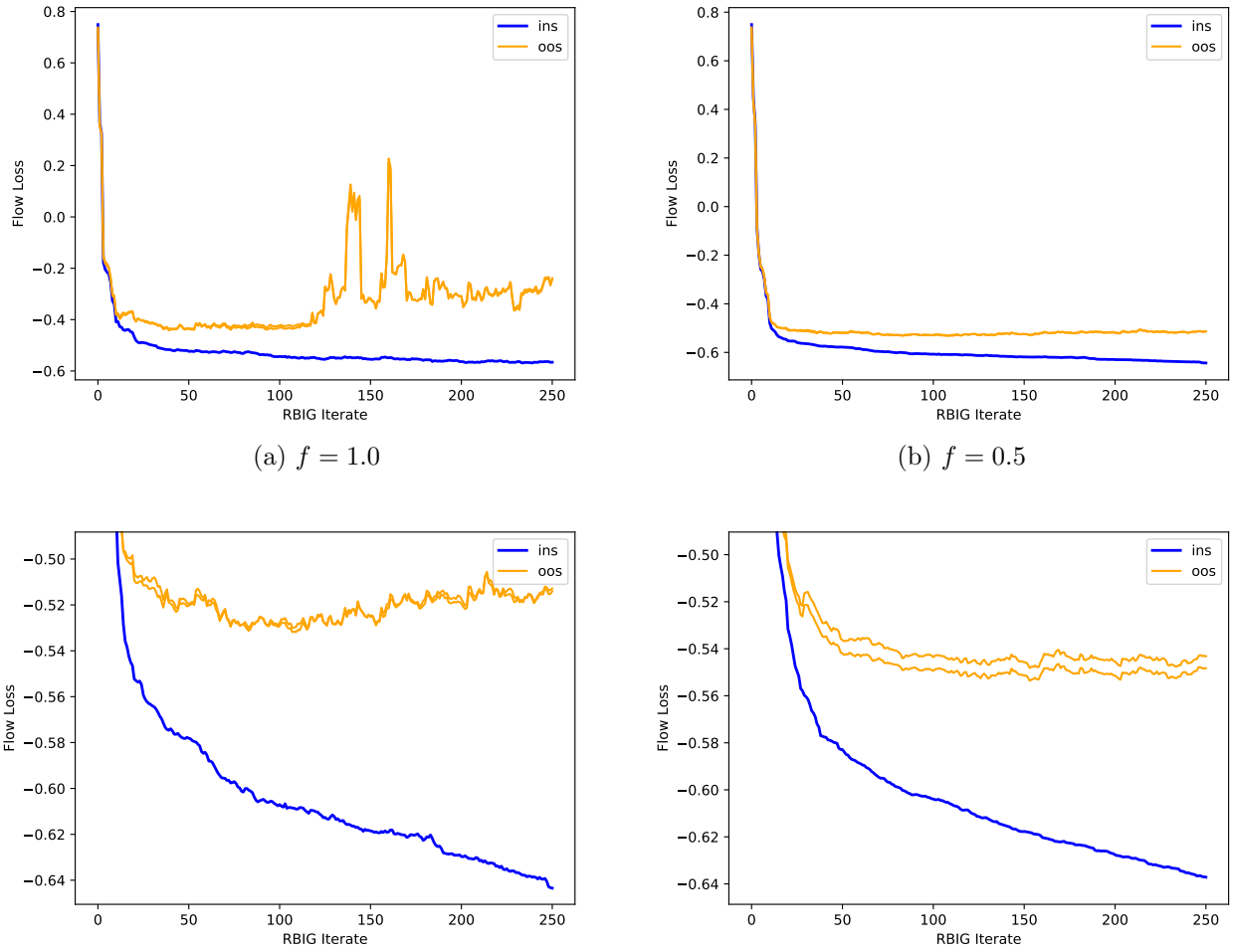
(a) (b)

Figure 1. (a) Our initial distribution is a fuzzy sine wave, from which we draw $N = 2000$ samples. The first component $\mathbf{x}^{(1)}$ of the fuzzy sine wave is Gaussian, and the second component is given by $\mathbf{x}^{(2)} = 0.2z + \sin(\mathbf{x}^{(1)})$, where z is Gaussian. (b) After $T = 250$ fractional iterations (with $f = 0.2$), we obtain a set of points with a more nearly Gaussian distribution.



(a) (b)

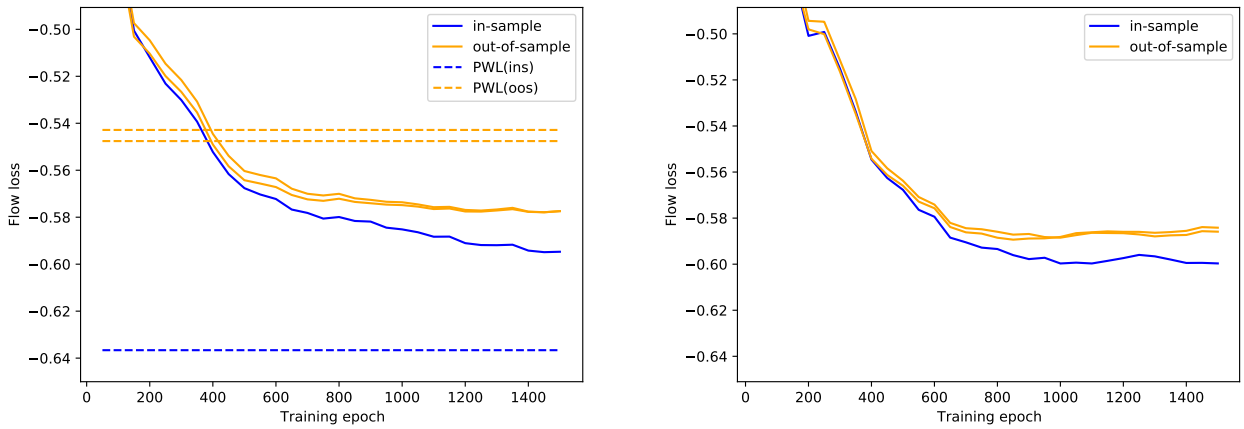
Figure 2. (a) Scattered plots generated by taking Gaussian variates and applying $T^{-1}(\mathbf{x})$ to them. (b) Probability density function, as estimated from Eq. (2) with $T(\mathbf{x})$ obtained from the Gaussianization process.



(a) $f = 1.0$ (b) $f = 0.5$

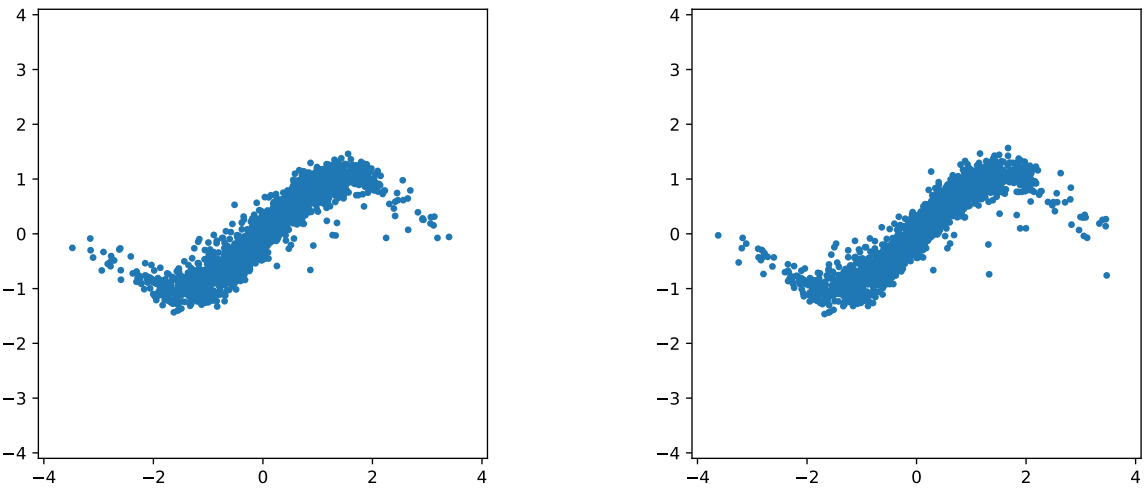
(c) $f = 0.5$ (d) $f = 0.2$

Figure 3. Flow loss vs iteration for RBIG with piecewise linear squashing function: (a) using full ($f = 1.0$) iterations; (b,c) using fractional ($f = 0.5$) iterations; and (d) using smaller fractional ($f = 0.2$) iterations. The bottom panels (c,d) zoom in on the flow loss range [-0.65,-0.51] in order to more clearly illustrate the flow loss values as the asymptotic performance is reached. Here heavy blue lines are the in-sample (ins) flow loss, and the lighter orange lines are out-of-sample (oos). These experiments correspond to training with $N = 2000$ samples from the fuzzy sine wave seen in Fig. 1. The out-of-sample curves are two independent samplings of $10 \times N$ samples each. We see in these examples the in-sample flow loss continues to decrease with more iterations; but the out-of-sample loss is erratic for $f = 1.0$, begins to increase after about 100 iterations for $f = 0.5$, and saturates for $f = 0.2$.



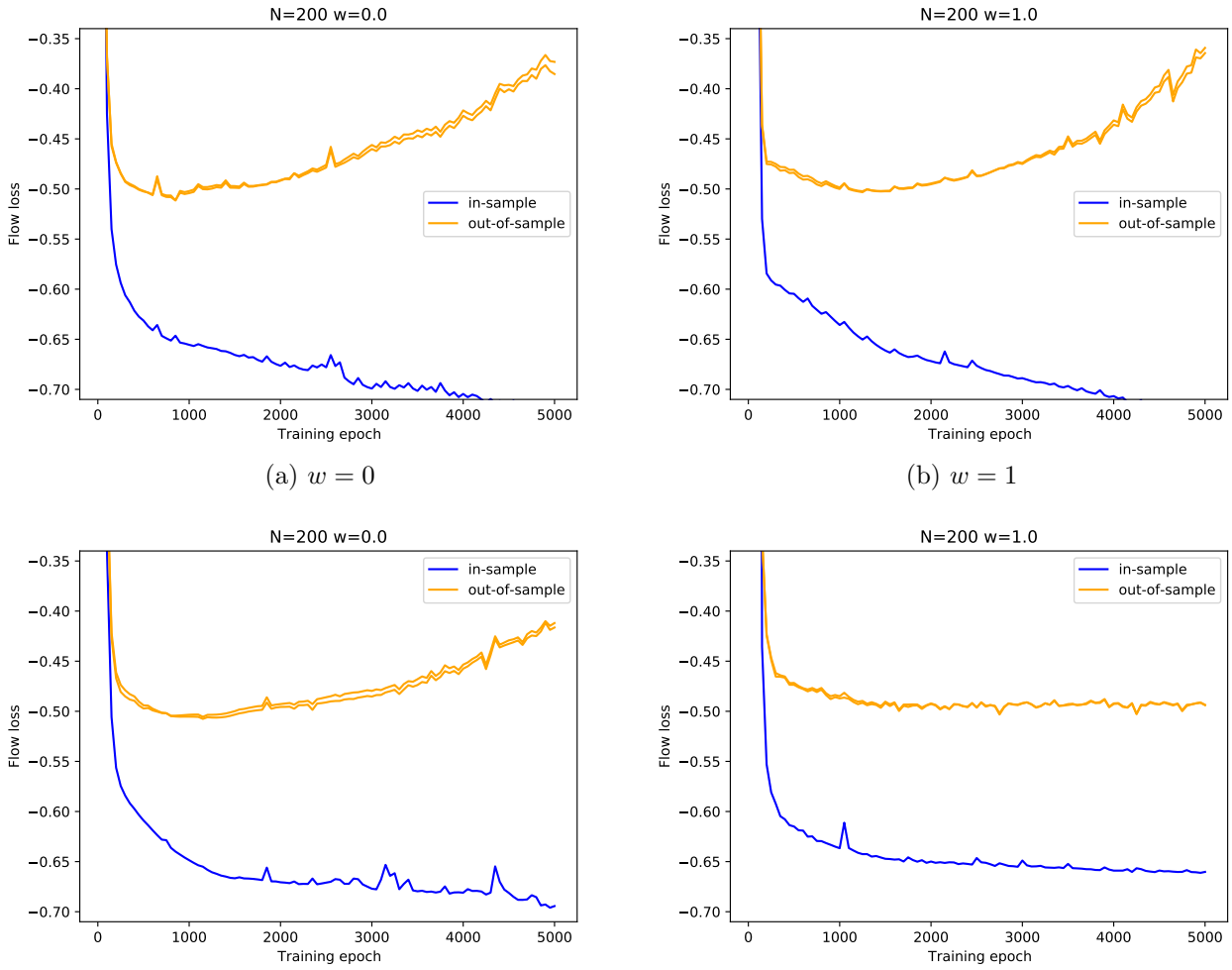
(a) (b)

Figure 4. Flow loss vs training epoch for distilled Gaussianization. (a) Training for this example is given by $(\mathbf{x}, \mathbf{y})$ pairs that are obtained at the end of the RBIG process depicted in Fig. 3(d). The dashed lines correspond to the flow loss exhibited by the $T(\mathbf{x})$ that was learned from the RBIG process. The solid lines correspond to the $T(\mathbf{x})$ that the neural net has learned from the $(\mathbf{x}, \mathbf{y})$ values. (b) Since we only need $(\mathbf{x}, \mathbf{y})$ values to train the neural network, we do not need to use the full RBIG process, but instead can use the "plain" process (described in Algorithm 3). We do not have dashed lines in this figure because there is no $T(\mathbf{x})$ produced by the plain process. But comparing (b) to (a), we see that the distilled Gaussianizer trained from the plain RBIG is comparable to the distilled Gaussianizer trained from the full RBIG, and that both distilled Gaussianizers are better than RBIG.



(a) (b)

Figure 5. Reconstruction of the original distribution by applying $T^{-1}(\mathbf{y})$ to points $\mathbf{y}$ drawn from a Gaussian distribution. Here the functiotn T^{-1} was trained directly from the same $(\mathbf{y}, \mathbf{x})$ pairs (but with $\mathbf{x}$ and $\mathbf{y}$ swapped) that were used for training in Fig. 4. Panels (a) and (b) here correspond to panels (a) and (b) in Fig. 4.



(a) $w = 0$ (b) $w = 1$

(c) $w = 0$ (d) $w = 1$

Figure 6. Flow loss vs training epoch for cycle-consistent distilled Gaussianization. Here $N = 200$ samples were created from RBIG and used as training for neural network. Top panels (a,b) correspond to the simple cycle consistency described in Section 3.2.1; bottom panels (c,d) correspond to the data-augmented cycle consistency described in Section 3.2.2. The left panels (a,c) use a weight of zero, so correspond to training without cycle consistency; thus, they correspond to two realizations of the same process; the right panels (b,d) correspond to training with weight $w = 1$. Note that $K = 1000$ auxiliary points were used in the data augmented approach.

An important caveat here is that although our ultimate goals (in anomaly, anomalous change, and target detection, for example) can be achieved by first estimating the probability distribution function (PDF), this approach violates Vapnik's famous dictum[11] that you should not try to solve a problem by solving a more difficult problem as an intermediate step. And estimating the PDF is a classic example of a more difficult intermediate problem. With this in mind, we are led to ask whether the Gaussianization process can be adapted to solving some of these other problems more directly.

6. ACKNOWLEDGMENTS

We are grateful to Gustau Camps-Valls, from whom both of us learned about Gaussianization during a talk at Descartes Labs. At that talk, Gustau described the process of iteratively rotating and squashing a distribution as akin to making meatballs. This work was supported in part by the United States Department of Energy (DOE) through the Laboratory Directed Research and Development (LDRD) program at Los Alamos National Laboratory.

REFERENCES

1. G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv 1503.02531v1* (stat.ML), 2015.
2. S. Matteoli, M. Diani, and J. Theiler, "An overview of background modeling for detection of targets and anomalies in hyperspectral remotely sensed imagery," *J. Selected Topics in Applied Earth Observations and Remote Sensing (JSTARS)* **7**, pp. 2317–2336, 2014.
3. S. S. Chen and R. A. Gopinath, "Gaussianization," *Advances in Neural Information Processing Systems (NIPS)* **13**, pp. 423–429, 2000.
4. V. Laparra, G. Camps-Valls, and J. Malo, "Iterative Gaussianization: from ICA to random rotations," *IEEE Trans. Neural Networks* **22**(4), pp. 537–549, 2011.
5. C. Meng, Y. Song, J. Song, and S. Ermon, "Gaussianization flows," *Proc. 23rd International Conference on Artificial Intelligence and Statistics* **108**, pp. 4336–4345, 2020.
6. V. Laparra, J. E. Johnson, G. Camps-Valls, R. Santos-Rodríguez, and J. Malo, "Information theory measures via multidimensional Gaussianization," *arXiv 2010.03807v2* (stat.ML), 2020.
7. C. Bucilă, R. Caruana, and A. Niculescu-Mizil, "Model compression," *Proc. 12th International Conference on Knowledge Discovery and Data Mining* (KDD), p. 535–541, 2006.
8. W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, "FFJORD: Free-form continuous dynamics for scalable reversible generative models," *arXiv 1810.01367v3* (cs.LG), 2018.
9. C. Finlay, J. Jacobsen, L. Nurbekyan, and A. M. Oberman, "How to train your neural ODE: the world of Jacobian and kinetic regularization," *arXiv 2002.02798v3* (stat.ML), 2020.
10. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems (NeurIPS) **32***, pp. 8024–8035, 2019.
11. V. Vapnik, *The Nature of Statistical Learning Theory*, Springer, New York, 2nd ed., 1999.