Relative gradient optimization of the Jacobian term in unsupervised deep learning

Luigi Gresele [*1,2] **Giancarlo Fissore** [*3,4] **Adrián Javaloy** [1] **Bernhard Schölkopf** [1] **Aapo Hyvärinen** [4,5]

Abstract

Learning expressive probabilistic models correctly describing the data is a ubiquitous problem in machine learning. A popular approach for solving it is mapping the observations into a representation space with a simple joint distribution. Deep density models have been widely used for this task, but their likelihood-based training requires estimating the log-determinant of the Jacobian and is computationally expensive, thus imposing a trade-off between computation and expressive power. We propose a new approach for exact likelihood-based training of such neural networks. Based on relative gradients, we exploit the matrix structure of neural network parameters to compute updates efficiently even in high-dimensional spaces; the computational cost of the training is quadratic in the input size, in contrast with the cubic scaling of the naive approaches. This allows fast training with objective functions involving the log-determinant of the Jacobian without imposing constraints on its structure.

1. Introduction

Many problems of machine learning and statistics involve learning invertible transformations of complex, multimodal probability distributions into simple ones. Examples include density estimation by transforming simple base distributions (Tabak et al., 2010), which also has applications in data generation (Dinh et al., 2014; Kingma and Dhariwal, 2018; Grathwohl et al., 2018); variational inference (Rezende and Mohamed, 2015); and nonlinear independent component analysis (Hyvarinen and Morioka, 2016; Hyvärinen et al., 2018). One approach to learn such transformations is to represent them as compositions of simple maps (Tabak and Turner, 2013), and deep neural networks provide a natural framework for implementing this idea (Rippel and Adams, 2013). Unfortunately, typical strategies employed in neural networks training do not scale well for objective functions like the aforementioned ones; in fact, through a change of variable, the determinant of the Jacobian of the transformations appears in the objective. Its exact computation, let alone its deployment for optimization purposes, quickly gets prohibitively computationally demanding as the data dimensionality grows. A large part of the research on deep density estimation has therefore been dedicated to considering a restricted class of transformations such that the computation of the Jacobian term is trivial (Dinh et al., 2014; Rezende and Mohamed, 2015; Dinh et al., 2016; Kingma et al., 2016; Huang et al., 2018; Chen and Duvenaud, 2019), imposing a tradeoff between computation and expressive power.

In this work, we present an efficient way to optimize the exact maximum likelihood objective for deep density estimation as well as for learning disentangled features. The starting point is that the parameters of the linear transformations are matrices; this allows us to exploit properties of the Riemannian geometry of matrix spaces to derive parameter updates in terms of the relative gradient, a particular kind of natural gradient (Amari, 1998) which was originally introduced in the context of linear ICA (Cardoso and Laheld, 1996). We show how this can be integrated with the usual backpropagation employed in gradient based training, yielding an overall efficient way to optimize the Jacobian term in neural networks. The computational cost of our proposed optimization procedure is quadratic in the input size—essentially the same as ordinary backpropagation— which is in stark contrast with the cubic scaling of the naive way of optimizing via automatic differentiation. The joint asymptotic scaling of forward and backward pass as a function of the input size is therefore the same that aforementioned alternative methods achieve by imposing strong restrictions on the neural network structure (Rezende and Mohamed, 2015) and thus on the class of functions they can represent. In con-

[*]Equal contribution [1]MPI for Intelligent Systems, Tübingen [2]MPI for Biological Cybernetics, Tübingen [3]Université Paris-Saclay, Inria, Inria Saclay-Île-de-France, 91120, Palaiseau, France [4]Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France [5]Univ. Helsinki. Correspondence to: L. G. <luigi.gresele@tuebingen.mpg.de>, G. F. <giancarlo.fissore@inria.fr>.

Second workshop on *Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models* (ICML 2020), Virtual Conference

trast, our approach allows to efficiently optimize the exact objective for neural networks with arbitrary Jacobians.

2. Background

2.1. Maximum likelihood for density estimation

Consider a generative model of the form

$$\mathbf{x} = \mathbf{f}(\mathbf{s}) \tag{1}$$

$\mathbf{x} \in \mathbb{R}^D$ represents the observed variable, $\mathbf{s} \in \mathbb{R}^D$ is a real vector for which a simple *base distribution* p_s is assumed, and $\mathbf{f} : \mathbb{R}^D \to \mathbb{R}^D$ is a deterministic and invertible function, which we refer to as *forward* transformation. Under the model specified above, the log-likelihood of a single datapoint $\mathbf{x}$ can be written as

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_s(\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})) + \log |\det \mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})| , \tag{2}$$

where $\mathbf{g}_{\boldsymbol{\theta}}$ is some representation with parameters $\boldsymbol{\theta}$ of the *inverse* transformation of $\mathbf{f}$ (the forward transformation could also be parameterized, but here we only explicitly parameterize its inverse); $\mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^{D\times D}$ is the Jacobian of $\mathbf{g}_{\boldsymbol{\theta}}$ computed at the point $\mathbf{x}$, whose elements are the partial derivatives $[\mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})]_{ij} = \partial g^i_{\boldsymbol{\theta}}(\mathbf{x})/\partial x^j$; $p_{\boldsymbol{\theta}}$ denotes the probability density functions of $\mathbf{x}$. In many cases, it is additionally assumed that p_s can be factorized in its components, $\log p_s(\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})) = \sum_i \log p_i(\mathbf{g}^i_{\boldsymbol{\theta}}(\mathbf{x}))$.

Maximum likelihood estimation for the model parameters amounts to finding the parameters $\boldsymbol{\theta}^*$ such that the expectation of the likelihood in equation 2 is maximized; for all practical purposes, the expectation will be substituted with the sample average.

2.2. Difficulty of optimizing the Jacobian term of neural networks

Fully connected neural networks represent complex functions through a sequential composition of transformations, $\mathbf{z}_k = \boldsymbol{\sigma}(\mathbf{W}_k\mathbf{z}_{k-1})$ for $k = 1, \ldots, L$, where $\mathbf{z}_0 = \mathbf{x}$. The parameters are randomly initialized and learned by optimizing an objective function $\mathcal{L}(\mathbf{x})$. In the case of the objective specified in Eq. (2), and by defining

$$\mathcal{L}_p(\mathbf{x}) = \sum_i \log p_i(\mathbf{g}^i_{\boldsymbol{\theta}}(\mathbf{x})); \quad \mathcal{L}_J(\mathbf{x}) = \log |\det \mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})| ,$$

the objective can be rewritten as $\mathcal{L}(\mathbf{x}) = \mathcal{L}_p(\mathbf{x}) + \mathcal{L}_J(\mathbf{x})$. The evaluation of the first term $\mathcal{L}_p$ can be performed easily if a simple form for the base distribution p_s is chosen, and its gradient can be computed employing simple backpropagation (Rumelhart et al., 1986).

In contrast, the evaluation of the gradient of the second term, $\mathcal{L}_J$, is very problematic, and our main concern in this paper. The key computational bottleneck is in fact given by the evaluation of the Jacobian during the forward pass. Since the Jacobian involves derivatives of the function $\mathbf{g}_{\boldsymbol{\theta}}$ with respect to its inputs $\mathbf{x}$, this evaluation can be performed through automatic differentiation. Overall, it can be shown (Baydin et al., 2018) that both forward and backward mode automatic differentiation for a L-layer, fully connected neural network scale as $\mathcal{O}(LD^3)$, with L the number of layers. This is prohibitive in many practical applications with a large data dimension D.

Normalizing flows with simple Jacobians An approach to alleviate the computational cost of this operation is to deploy neural network architectures for which the evaluation of $\mathcal{L}_J$ is trivial. For example, in autoregressive normalizing flows (Dinh et al., 2014; 2016; Kingma et al., 2016; Huang et al., 2018) the Jacobian of the transformation is constrained to be lower triangular. In this case, its determinant can be trivially computed with a linear cost in D. Notice however that the computational cost of the forward pass still scales quadratically in D; the overall complexity of forward plus backward pass is therefore quadratic in the input size (Rezende and Mohamed, 2015).

Most critically, such architectures imply a strong restriction on the class of transformations that can be learned. While it can be shown, based on (Hyvärinen and Pajunen, 1999), that under certain conditions this class of functions has universal approximation capacity for *densities* (Huang et al., 2018), that is less general than other notions of universal approximation (Hornik et al., 1989; Hornik, 1991). In fact it is obvious that functions with such triangular Jacobians cannot have universal approximation of *functions*, since, for example, the first variable can only depend on the first variable. This is a severe problem in learning features for disentanglement, for example by nonlinear ICA (Hyvarinen and Morioka, 2016; Hyvärinen et al., 2018), which would usually require unconstrained Jacobians. In other words, such triangular restrictions might imply that the deployed networks are not general purpose: (Behrmann et al., 2018) showed that such designs, typically used for density estimation, can severely hurt discriminative performance.

3. Log-determinant of the Jacobian for fully connected neural networks

As a first step toward efficient optimization of the $\mathcal{L}_J$ term, we next provide the explicit form of the Jacobian for fully connected neural networks. As a starting point, notice that invertible and differentiable transformations are *composable*; given any two such transformations, their composition is also invertible and differentiable, and the determinant of the Jacobian of a composition of functions is given by the product of the determinants of the Jacobians of each function, $\det \mathbf{J}[\mathbf{g}_2 \circ \mathbf{g}_1](\mathbf{x}) = \det \mathbf{J}\mathbf{g}_2\,(\mathbf{g}_1(\mathbf{x})) \cdot \det \mathbf{J}\mathbf{g}_1(\mathbf{x})$.

The log-determinant of the full Jacobian for a neural network therefore simply decomposes in a sum of the log-determinants of the Jacobians of each module, $\mathcal{L}_J(\mathbf{x}) = \sum_{k=1}^{L} \log|\det \mathbf{J}\mathbf{g}_k(\mathbf{z}_{k-1})|$. We will focus on the Jacobian of a single submodule k with respect to its input $\mathbf{z}_{k-1}$; with a slight abuse of notation, we will call it $\mathcal{L}_J(\mathbf{z}_{k-1})$. As we remarked, fully connected $\mathbf{g}_k$ are themselves compositions of a linear operation and an element-wise invertible nonlinearity; applying the same reasoning, we then have

$$\begin{aligned}\mathcal{L}_J(\mathbf{z}_{k-1}) &= \sum_{i=1}^{D} \log\left|\sigma'(y_k^i)\right| + \log|\det \mathbf{W}_k| \\ &=: \mathcal{L}_J^1(\mathbf{y}_k) + \mathcal{L}_J^2(\mathbf{z}_{k-1})\,.\end{aligned}$$

where $\mathbf{y}_k = \mathbf{W}_k\mathbf{z}_{k-1}$. The first term $\mathcal{L}_J^1$ is a sum of univariate functions of single components of the output of the module, and its gradient is easy to compute.

The second term $\mathcal{L}_J^2$ involves a nonlinear function of the determinant of the weight matrix and its derivative is equal to

$$\frac{\partial \log|\det \mathbf{W}_k|}{\partial \mathbf{W}_k} = \left(\mathbf{W}_k^T\right)^{-1}\,. \tag{3}$$

Therefore, the computation of the gradients relative to such terms involves a matrix inversion, with cubic scaling in the input size. For a fully connected neural network of L layers, given that we have one such operation to perform for each layer, the gradient computation for these terms alone would have a complexity of $\mathcal{O}(LD^3)$. This would dominate the full gradient computation and match the cubic complexity of the automatic differentiation evaluation of the explicit Jacobian term discussed in Section 2.

4. Relative gradient descent for neural networks

We now derive the basic form of the relative gradient, following the approach in (Cardoso and Laheld, 1996). This approach can be seen as a special case of the more general framework of natural gradients in a Riemannian space, where an information-geometric approach is often used (Amari, 1998).

Relative gradient based on multiplicative perturbation In a classical gradient approach for optimization, we add a small vector $\boldsymbol{\epsilon}$ to a point $\mathbf{x}$ in a Euclidean space. However, with matrices, we are actually perturbing a matrix with another, and this can be done in different ways. In the relative gradient approach, we make a *multiplicative* perturbation of the form

$$\mathbf{W}_k \to (\mathbf{I} + \boldsymbol{\epsilon})\mathbf{W}_k \tag{4}$$

where $\boldsymbol{\epsilon}$ is an infinitesimal matrix. If we consider the effect of such a perturbation on a scalar-valued function $f(\mathbf{W}_k)$, we have

$$\begin{aligned}f((\mathbf{I}+\boldsymbol{\epsilon})\mathbf{W}_k) - f(\mathbf{W}) &= \langle \nabla f(\mathbf{W}_k), \boldsymbol{\epsilon}\mathbf{W}_k\rangle + o(\mathbf{W}_k) \\ &= \langle \nabla f(\mathbf{W}_k)\mathbf{W}_k^T, \boldsymbol{\epsilon}\rangle + o(\mathbf{W}_k)\end{aligned}$$

which shows that the direction of steepest descent in this case is given by making $\boldsymbol{\epsilon} = \mu \nabla f(\mathbf{W}_k)\mathbf{W}_k^T$ where μ is an infinitesimal step size. Furthermore, when we combine this $\boldsymbol{\epsilon}$ with the definition of a multiplicative update, we find that the best perturbation to $\mathbf{W}$ is actually given as

$$\mathbf{W}_k \to \mathbf{W}_k + \mu \nabla f(\mathbf{W}_k)\mathbf{W}_k^T\mathbf{W}_k \tag{5}$$

That is, the classical Euclidean gradient is replaced by $\nabla f(\mathbf{W})\mathbf{W}^T\mathbf{W}$, i.e. it is multiplied by $\mathbf{W}^T\mathbf{W}$ from the right. This is the relative gradient.

Jacobian term optimization through the relative gradient In section 3, we showed that the difficulty in computing the gradient of the log-determinant is in the terms $\mathcal{L}_J^2$, whose gradient involves a matrix inversion. Now we show that by exploiting the relative gradient, this matrix inversion vanishes. In fact, when multiplying the right hand side of equation 3 by $\mathbf{W}_k^T\mathbf{W}_k$ from the right we get

$$\left(\mathbf{W}_k^T\right)^{-1}\mathbf{W}_k^T\mathbf{W}_k = \mathbf{W}_k\,. \tag{6}$$

Most notably, we therefore have to perform *no additional operation* to get the relative gradient with respect to this term of the loss; it is, so to say, *implicitly* computed — as we know that the update for the parameters in $\mathbf{W}_k$ with respect to the error term $\mathcal{L}_J^2$ is proportional to $\mathbf{W}_k$ itself.

As for the remaining terms of the loss, $\mathcal{L}_p$ and $\mathcal{L}_J^1$, simple backpropagation allows us to compute the weight updates given by the ordinary gradient, which still need to be multiplied by $\mathbf{W}_k^T\mathbf{W}_k$ to turn it into a relative gradient. Remarkably, this can be done avoiding matrix-matrix multiplications, which would be computationally expensive. By applying the relative gradient carefully, we can avoid matrix-matrix multiplication altogether; overall, the resulting update rule will be equal to

$$(\Delta\mathbf{W}_k)\,\mathbf{W}_k^T\mathbf{W}_k \propto \left(\mathbf{z}_{k-1}\left(\boldsymbol{\delta}_k^T\mathbf{W}_k^T\right) + \mathbf{I}\right)\mathbf{W}_k\,, \tag{7}$$

where $\boldsymbol{\delta}_k$ is the backpropagated error up to layer k. We report the full derivation in appendix A.

Complexity and invertibility The relative gradient updates due to Eq. (7) only require matrix-vector or vector-vector multiplications, each of which scales as $\mathcal{O}(D^2)$, in a fixed number at each layer; that is, overall $\mathcal{O}(LD^2)$ operations. They therefore don't increase the complexity of a normal forward pass. Furthermore, the overall complexity with respect to the input size is quadratic, resulting in an overall quadratic scaling with the input size without imposing strong restrictions on the Jacobian of the transformation.

Our method is guaranteed to learn invertible transformations:

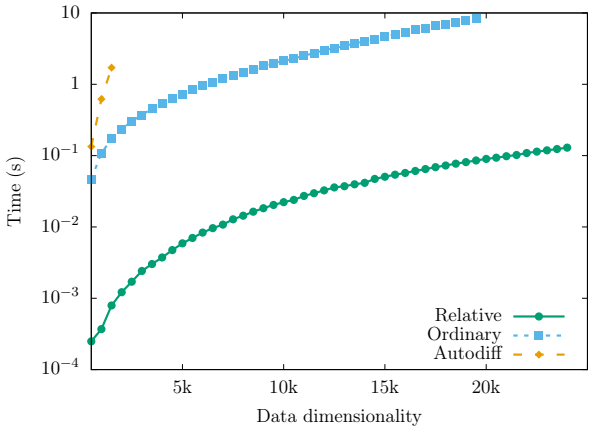

Figure 1: Comparison of the average computation times of a single evaluation of the gradient of the log-likelihood over a batch of size 100. Values are the mean over 100 steps, and the experiments have been run 10 times on a single GPU. The standard error of the mean is not reported as it is order of magnitude smaller then the scale of the plot.

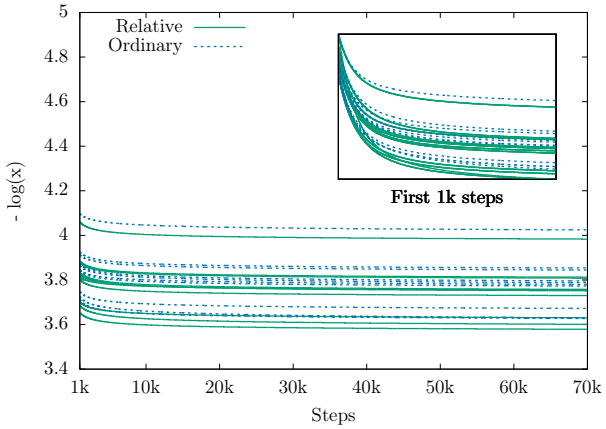

Figure 2: Time-evolution of the negative log-likelihood for deterministic full-batch optimization for 1000 samples of dimensionality 2. 10 couples of curves with the same initialization are shown.

in fact, the weight matrices are invertible at initialization, and the $\mathcal{L}_J^2$ term ensures that they don't diverge throughout the training[1]; furthermore, we employ invertible nonlinearities. The cost of computing the inverse transformation is cubic in D. We elaborate on this point in appendix B.

5. Experiments

The code used for our experiments can be found at `https://github.com/fissoreg/relative-gradient-jacobian`.

[1]Though high learning rates and numerical instabilities might compromise it in practice.

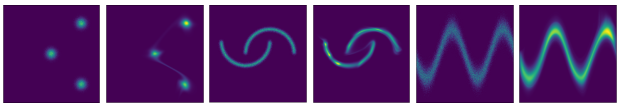

Figure 3: Illustrative examples of 2D density estimation. Samples from the true distribution and predicted densities are shown, in this order, side by side

Computation of relative vs. ordinary gradient In figure 1 we experimentally verify the computational efficiency of our proposed procedure implemented using the formulas in section 4. Comparing to an explicit computation of the ordinary gradient (section 3, eq. 3 in particular) and to a naive implementation that explicitly computes the full jacobian (using automatic differentiation through the JAX package (Bradbury et al., 2018)), we observe that *the relative gradient we propose is much faster*, typically by two orders of magnitude. Experimental details and further comparisons are reported in appendix E.

Optimization by relative vs. ordinary gradient To the best of our knowledge, this is the first work proposing relative gradient optimization for neural networks; as a sanity check, we want to verify that the learning dynamics induced by the relative as opposed to the ordinary gradient do not bias the training procedure towards less optimal solutions or create other problems. Figure 2 shows the results: there is no big difference between the two gradient methods. There may actually be a slight advantage for the relative gradient, but that is immaterial since our main point here is to show that *relative gradient does not need more iterations* to give the same performance.

Combining these two results, we see that the proposed relative gradient approach leads to a *much faster optimization* than the ordinary gradient.

Density estimation Although our main contribution is the computational speed-up of the gradient computation demonstrated above, we further show some simple results on density estimation as described in Section 2.1 to highlight the potential of the relative gradient. Figure 3 showcases the ability of our method to convincingly model arbitrarily complex densities for some toy examples. In appendix E we report more extensive experiments on real-world datasets.

6. Conclusions

We proposed a new method for exact optimization of objective functions involving the log-determinant of the Jacobian of a neural network. This allows for employing models which, unlike typical alternatives in the normalizing flows literature, have no strong limitations on the structure of the

Jacobian. These neural networks have a strong universal approximation capacity *for functions*, a more general notion than the one of universal approximation *for density functions* of normalizing flows with triangular Jacobians. The importance of the optimization of the log-determinant of the Jacobian is well-known, but it has not been previously shown that there is a way around its difficulty without restricting expressivity. Now that we have shown that the optimization of this term can be done quite cheaply, a substantial fraction of the research in the field can be reformulated in stronger terms and with more generality.

References

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153), 2018.

Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. *arXiv preprint arXiv:1811.00995*, 2018.

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

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

J-F Cardoso and Beate H Laheld. Equivariant adaptive source separation. *IEEE Transactions on signal processing*, 44(12):3017–3030, 1996.

Tian Qi Chen and David K Duvenaud. Neural networks with cheap differential operators. In *Advances in Neural Information Processing Systems*, pages 9961–9971, 2019.

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

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

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989. ISSN 0893-6080.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. *arXiv preprint arXiv:1804.00779*, 2018.

Aapo Hyvarinen. Fast and robust fixed-point algorithms for independent component analysis. *IEEE transactions on Neural Networks*, 10(3):626–634, 1999.

Aapo Hyvarinen and Hiroshi Morioka. Unsupervised feature extraction by time-contrastive learning and nonlinear ICA. In *Advances in Neural Information Processing Systems*, pages 3765–3773, 2016.

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

Aapo Hyvärinen, Hiroaki Sasaki, and Richard E Turner. Nonlinear ICA using auxiliary variables and generalized contrastive learning. *arXiv preprint arXiv:1805.08651*, 2018.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems*, pages 10215–10224, 2018.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pages 4743–4751, 2016.

Yann LeCun, Corinna Cortes, and Christopher JC Burges. The MNIST database of handwritten digits, 1998. *URL http://yann. lecun. com/exdb/mnist*, 10:34, 1998.

Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 416–423. IEEE, 2001.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347, 2017.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

Oren Rippel and Ryan Prescott Adams. High-dimensional probability estimation with deep density models. *arXiv preprint arXiv:1302.5125*, 2013.

Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

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.

Esteban G Tabak, Eric Vanden-Eijnden, et al. Density estimation by dual ascent of the log-likelihood. *Communications in Mathematical Sciences*, 8(1):217–233, 2010.

APPENDIX

A. Backpropagation in neural networks

We will follow (Rojas, 2013), Chapter 7, section 7.3.3 for the notation. Let us define a two-layer neural network

$$\mathbf{g}_\theta(\mathbf{x}) = \boldsymbol{\sigma}\left(\mathbf{W}_2\boldsymbol{\sigma}\left(\mathbf{W}_1\mathbf{x}\right)\right) \tag{8}$$

where we also define

$$\begin{aligned}\mathbf{z}_2 &= \boldsymbol{\sigma}\left(\mathbf{W}_2\mathbf{z}_1\right)\\ \mathbf{z}_1 &= \boldsymbol{\sigma}\left(\mathbf{W}_1\mathbf{x}\right)\ .\end{aligned}$$

and

$$\begin{aligned}\mathbf{u}_2 &= \boldsymbol{\sigma}'\left(\mathbf{W}_2\mathbf{z}_1\right)\\ \mathbf{u}_1 &= \boldsymbol{\sigma}'(\mathbf{W}_1\mathbf{x})\end{aligned}$$

and

$$\begin{aligned}\mathbf{y}_2 &= \mathbf{W}_2\mathbf{z}_1\\ \mathbf{y}_1 &= \mathbf{W}_1\mathbf{x}\end{aligned}$$

We need to consider the contributions to the error due to formulas $\mathcal{L}_p$ and $\mathcal{L}_J^1$ (the contribution due to terms $\mathcal{L}_J^2$ will be dealt with separately). For $\mathcal{L}_p$, we define

$$e(x) = \frac{\partial}{\partial x}\log p(x')|_{x'=x}$$

and

$$\mathbf{e} = \begin{pmatrix} e(z_2^1)\\ e(z_2^2)\\ \vdots\\ e(z_2^D)\end{pmatrix}$$

To deal with the terms in $\mathcal{L}_J^1$, we define

$$h(x) = \frac{\partial}{\partial x}\log x'|_{x'=x} \tag{9}$$

$$= \frac{1}{x} \tag{10}$$

and

$$\mathbf{h}_k = \begin{pmatrix} h(u_k^1)\\ h(u_k^2)\\ \vdots\\ h(u_k^D)\end{pmatrix}$$

for $k = 1, 2$. During forward propagation, we store the $\mathbf{D}_k = \operatorname{diag}\left(\boldsymbol{\sigma}'\left(\mathbf{y}_k\right)\right)$ for $k = 1, 2$,

$$\mathbf{D}_k = \begin{pmatrix} \sigma'(y_k^1) & 0 & \cdots & 0\\ 0 & \sigma'(y_k^2) & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \sigma'(y_k^D)\end{pmatrix}$$

and the $\mathbf{G}_k = \operatorname{diag}\left(\boldsymbol{\sigma}''\left(\mathbf{y}_k\right)\right)$ for $k = 1, 2$,

$$\mathbf{G}_k = \begin{pmatrix} \sigma''(y_k^1) & 0 & \cdots & 0\\ 0 & \sigma''(y_k^2) & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \sigma''(y_k^D)\end{pmatrix}$$

for example, if the nonlinearity were a sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$, the second derivative would be $\sigma''(x) = \sigma(x)(1 - \sigma(x))\left(1 - 2\sigma(x)\right)$. Then

$$\boldsymbol{\delta}_2 = \mathbf{D}_2\mathbf{e} + \mathbf{G}_2\mathbf{h}_2$$

and

$$\boldsymbol{\delta}_1 = \mathbf{D}_1\mathbf{W}_2\boldsymbol{\delta}_2 + \mathbf{G}_1\mathbf{h}_1$$

In general, the following recursive relationship holds

$$\boldsymbol{\delta}_k = \mathbf{D}_k\mathbf{W}_{k+1}\boldsymbol{\delta}_{k+1} + \mathbf{G}_k\mathbf{h}_k \tag{11}$$

Which results in the update rule

$$\Delta\mathbf{W}_k = -\gamma\mathbf{z}_{k-1}\boldsymbol{\delta}_k^T\,, \tag{12}$$

where $\mathbf{z}_0 = \mathbf{x}$. Notice that the only necessary operations are vector-matrix, matrix-vector and vector-vector multiplications.

A.1. Relative gradient

Now if we want to use the relative/natural gradient trick each of these terms needs to be multiplied by $\mathbf{W}_k^T\mathbf{W}_k$ from the right.

$$\Delta\mathbf{W}_k = -\gamma\mathbf{z}_{k-1}\boldsymbol{\delta}_k^T\mathbf{W}_k^T\mathbf{W}_k$$

Terms in $\mathcal{L}_J^2$ The terms in $\mathcal{L}_J^2$, consisting of $\log|\mathbf{W}_k|$ give as gradient $\left(\mathbf{W}_k^T\right)^{-1}$. This requires a $D \times D$ matrix inversion for each of the matrices. A possible strategy to avoid it is to substitute the ordinary gradient with a relative gradient (Cardoso and Laheld, 1996) where we multiply the gradient (w.r.t. the whole objective but for each layer separately) by $\mathbf{W}_k^T\mathbf{W}_k$ from the right. In this case, the updates for the $\mathbf{W}_k$ terms simply become proportional to the $\mathbf{W}_k$ themselves. Therefore, the update rule becomes

$$\Delta\mathbf{W}_k = -\gamma(\mathbf{z}_{k-1}\boldsymbol{\delta}_k^T\mathbf{W}_k^T\mathbf{W}_k + \mathbf{W}_k) \tag{13}$$

As we already noted, the operations involved in these updates can be performed in a way such that no matrix-matrix multiplication needs to be performed – only matrix-vector and vector-vector multiplication. This is more apparent when the update rules are rewritten as below

$$\Delta\mathbf{W}_k = -\gamma\left(\mathbf{z}_{k-1}\left(\left(\boldsymbol{\delta}_k^T\mathbf{W}_k^T\right)\mathbf{W}_k\right) + \mathbf{W}_k\right) \tag{14}$$

B. Invertibility and generation

Since a composition of invertible functions is invertible, invertibility of our learned transformation is guaranteed as long as the weight matrices and the element-wise nonlinearities are invertible. Square and randomly initialized (e.g. with uniform or normally distributed entries) weight matrices are known to be invertible with probability one; invertibility of the weight matrices throughout the training is guaranteed by the fact that the $\mathcal{L}_J^2$ terms would diverge for singular matrices (though high learning rates and numerical instabilities might compromise it in practice), as in estimation methods for linear ICA (Bell and Sejnowski, 1995; Cardoso and Laheld, 1996; Hyvarinen, 1999). We additionally employ nonlinearities which are invertible by construction; we include more details about this in the next section. If we are interested in data generation, we also need to invert the learned function. In practice, the cost of inverting each of the matrices is $\mathcal{O}(D^3)$, but the operation needs to be performed only once. As for the nonlinear transformation, the inversion is cheap since we only need to numerically invert a scalar function, for which often a closed form for it is available; otherwise, Newton methods can be suitably used.

C. Universal approximation capacity in Normalizing Flows

It has been shown that standard multilayer feedforward network can approximate any continuous function to any degree of accuracy. (Leshno et al., 1993) proved that a standard multilayer feedforward network with a locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy if and only if the network's activation function is not a polynomial. Biases also play a crucial role in this, as universal approximation capacity wouldn't be possible without.

We remark that this is a much stronger feature than that of universal approximation for densities based on (Hyvärinen and Pajunen, 1999) normalizing flows with triangular Jacobians have (see e.g. (Huang et al., 2018)), since those can obviously not represent all possible functions — but only those with triangular Jacobians.

We discuss how to incorporate biases in our training procedure in appendix D, and the nonlinearities we employed in appendix E.

D. Relative gradient for the augmented matrix

As already discussed, a highly desirable feature for a neural network approximating an identifiable function would be to be a universal function approximator. Biases play a crucial role in this, as universal approximation capacity would not be possible without them.

Fortunately, affine transformations involving vector-matrix products plus biases can be represented as a single matrix operation through the formalism of the augmented matrix (see e.g. (Rojas, 2013)).

Linear affine operations of the form $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ can be represented via an augmented matrix as follows

$$\begin{bmatrix} \mathbf{y} \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & \mathbf{W} & & \mathbf{b} \\ 0 & \dots & 0 & 1 \end{array}\right] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \overline{\mathbf{W}} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \tag{15}$$

where we refer to the matrix $\overline{\mathbf{W}}$ as *augmented matrix*.

The question is whether the relative gradient trick can be applied to the augmented matrix. The main issue is that we would like, throughout our optimization procedure, to remain on the manifold of augmented matrices; that is, we do not want to change the last row of $\overline{\mathbf{W}}_k$. Therefore, the problem becomes a constrained optimization problem.

The $\mathcal{L}_J^2$ term It is easy to see that $\det \overline{\mathbf{W}}_k = \det \mathbf{W}_k$. The ordinary gradient for all terms in the last column and row of $\overline{\mathbf{W}}_k$ will therefore be equal to zero, and this will not be changed by the relative gradient trick; therefore, the contribution of this term will not lead us away from the manifold of augmented matrices.

The $\mathcal{L}_p$ and $\mathcal{L}_J^1$ terms Both the $\mathbf{y}_k$ and $\mathbf{z}_k$ terms will however be influenced by the presence of biases, so the gradients on the first D elements of the last column (that is $\mathbf{b}_k$) will be nonzero. Through the multiplication with $\overline{\mathbf{W}}_k^T \overline{\mathbf{W}}_k$, the updates given by the relative gradient on the last row of $\overline{\mathbf{W}}_k$ will therefore in general be nonzero, thus implying moving outside of the manifold we are interested in.

To solve this issue, we use a projected gradient algorithm, enforcing that the update for the last row of $\overline{\mathbf{W}}_k$ at each step is equal to zero. We call this algorithm *projected relative gradient descent*.

In practice, we can use the augmented matrix formalism to apply the relative trick to the full parameters and then extract only the updates for the parameters of interest $\mathbf{W}$, $\mathbf{b}$ disregarding the dummy row in (15). Denoting by $\mathbf{G}$ the gradients of $\mathbf{W}$ and by $\mathbf{g}_b$ the gradients of $\mathbf{b}$, we can compute the relative gradients as

$$\left[\begin{array}{c|c} \mathbf{G} & \mathbf{g}_b \\ \hline \mathbf{g} & g \end{array}\right] \overline{\mathbf{W}}^T \overline{\mathbf{W}}$$

$$\downarrow$$

$$\left[\begin{array}{c|c} \mathbf{G}\mathbf{W}^T\mathbf{W} + \mathbf{g}_b\mathbf{b}^T\mathbf{W} & \mathbf{G}\mathbf{W}^T\mathbf{b} + \mathbf{g}_b\mathbf{b}^T\mathbf{b} + \mathbf{g}_b \\ \hline \dots & \dots \end{array}\right]$$

The relative gradient updates we need are then given by

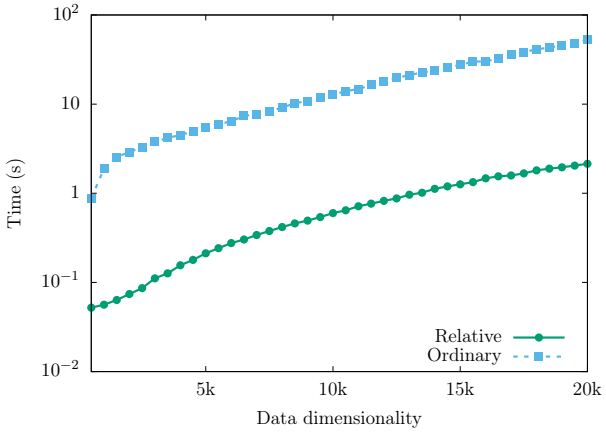

Figure 4: Comparison of the average computation times of a single evaluation of the gradient of the log-likelihood over a batch of size 100. Values are the mean over 5 steps, and the experiments have been run 5 times on a CPU cluster.

$$\Delta \mathbf{W} \to \mathbf{G}\mathbf{W}^T\mathbf{W} + \mathbf{g}_b \left(\mathbf{b}^T\mathbf{W}\right) \tag{16}$$

$$\Delta \mathbf{b} \to \mathbf{G}\left(\mathbf{W}^T\mathbf{b}\right) + \mathbf{g}_b(1 + \mathbf{b}^T\mathbf{b}) \tag{17}$$

Note that $\mathbf{G}$ is nothing more then the standard backpropagation update (12), thus we can efficiently compute $\Delta\mathbf{W}$ by avoiding matrix-matrix multiplications as in (7). For $\Delta\mathbf{b}$ we can directly avoid matrix-matrix multiplications by taking some care in the evaluation of (17).

E. Experiments

E.1. Computation of relative vs. ordinary gradient

Computational cost In section 5 and figure 1 we compared the computational cost of computing log-likelihood gradients with our newly proposed method and a naive backpropagation implementation when using hardware accelerators. Specifically, we used one Tesla P100 GPU card equipped with 16 GB of dedicated memory and circa 3500 computing cores. In figure 4 we show the same comparison for a computation platform comprising 48 cpu threads (Intel Xeon Processor E5-2650 v4 @ 2.20 GHz base frequency, 2.90 GHz max frequency) operating in parallel with about 250 GB of available RAM memory. It is hard to spot the expected theoretical improvement from $O(D^3)$ to $O(D^2)$, but a practical gain of about 2 orders of magnitude in computation time emerges in favor of the relative gradient computation.

In order to directly compare the execution times disregarding bottlenecks due to memory operations, we performed all of the experiments with no garbage collection. Anyways, by using always the same batch we made our experiments not very memory intensive and repeating the experiments with garbage collection enabled didn't show any substantial difference; we therefore don't report the plot.

Memory consumption It is usual in deep learning to be constrained by the memory consumption of the models in use, as the available memory on hardware accelerators is typically scarce. To operate, a network needs to store the data, the intermediate activations (needed to compute gradients) and the parameters. For our simple architecture, the bottleneck is the storage of the parameters; this is because we don't employ very deep architectures, so the amount of intermediate activations to store is limited, and the size of the parameters grows quadratically with respect to the data size, meaning that parameters storage clearly dominate over data storage (this is assuming that data are loaded in small minibatches, which is the norm). This is certainly problematic for very high-dimensional datasets (i.e. high definition images) but even from this point of view we have a clear advantage over an explicit optimization of the Jacobian term with automatic differentiation. In this latter case, in fact, we need to compute the full Jacobian of the affine transformations for each individual data point; like for the weight matrices, the size of these terms grows quadratically with the input size, further increasing the memory footprint of the optimization procedure.

As a simple example, we can compare the approximate memory requirements of the two methods in the moderately high-dimensional case with $D = 20000$. For a modest 2-layers network and employing Float32 weights (each requiring 4 Bytes (B) for storage), the memory needed to store the parameters amounts to $D^2 \times 4B \times 2(\text{layers}) = 3.2GB$. Assuming a minibatch size of 100, data and activations require around 10-100 MB which is clearly negligible. The computed gradients will require the same space as the parameters, raising the memory footprint to over 6GB. For the gradient computations themselves, our method doesn't require additional memory (theoretically), while explicit automatic differentiation requires storing as many jacobian terms as the size of the minibatch, thus requiring over 300GB in our simple case. As this is clearly unfeasible on common hardware accelerators, we can drop the parallelization of the jacobian terms computation to considerably reduce memory consumption (bringing it down to over 9GB in our case), but this comes at the cost of further slowing down an already inefficient procedure.

While the simple analysis above shows a clear advantage for our proposed method, from the practical point of view many additional technical details might play a role in incrementing the memory requirements of both methods (e.g. loading of libraries and computing environment, just-in-time compilation steps, intermediate computations that can't be fused together...). In figure 5 we report a simple profiling of the

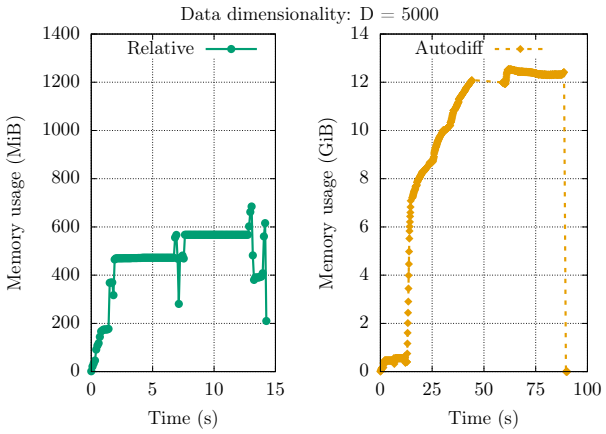

Figure 5: Comparison of the memory consumption for a single gradient evaluation. With D = 5000 our simplified analysis predicts a lower bound in the memory consumption of 400 MB for storing the parameters and the computed gradients; given that at startup time we observe a base memory consumption of almost 200 MB (computing environment + loaded libraries) we can see that our relative gradient implementation comes very close to the theoretical limit. For the naive autodiff implementation, instead, we compute a lower bound of 10.4 GB, which is approximately reflected in the empirical measurements (maximum consumption is almost 13 GB). Note: memory consumption for the autodiff case is reported in GiB, effectively making the scale of the plot one order of magnitude higher then in the relative gradient plot.

memory consumption of the two methods, which shows how the difference is relevant in practice.

E.2. Density estimation

Architecture Although mentioned all throughout the paper, let us recall the neural network used for these experiments. We here rely on the usual feedforward architecture, that is, a neural network for which the input is sequentially passed through an interleaving series of matrix multiplications and non-linear activation functions, being the last operation a matrix multiplication.

Nonlinearities Note that, since we make use of square weight matrices, the only two architectural hyperparameters left in our architecture are the number of layers in the network, L, and the non-linearity used. We consider two types of non-linearities. First, a smooth version of the leaky-ReLU activation function with a hyperparameter α,

$$\mathrm{s_L}(x) = \alpha x + (1-\alpha)\log(1+e^x). \tag{18}$$

Second, a weighted sum of the identity and hyperbolic tangent with two hyperparameters, α and β, controlling the steepness and "level of linearity" of the activation function,

$$\mathrm{s_T}(x) = \tanh(\alpha x) + \beta x. \tag{19}$$

However, in our experiments, these two hyperparameters for the s_T nonlinearity are fixed to $\alpha = 1$ and $\beta = 0.1$ always.

Toy examples For all the experiments shown in figure 3, we always use Adam as optimizer, fix the batch size and number of layers L to 100, use biases, and fix the activation function to $\mathrm{s_L}$ with $\alpha = 0.3$. We chose as base distribution the standard normal distribution. We plot, as in the quantitative experiments, the best model found during the training. Regarding the data, we sampled five-thousand samples for the training set and five-hundred points for the test set. The only changing hyperparameters across the figures is the learning rate and the number of epochs, which are summarised in table 1.

Table 1: Hyperparameters used for figure 3.

	MoG	half moons	sine
learning rate	0.001	0.001	0.005
no. of epochs	2000	1300	4000

Quantitative results on MNIST To obtain the density results on the MNIST dataset, the same preprocessing as in (Papamakarios et al., 2017) has been applied. For the model architecture, we fixed the number of layers to 2, we used the smooth Leaky-ReLU (18) with $\alpha = 0.01$ and a standard normal distribution as the base distribution. The optimization has been performed with Adam with default parameters. The hyperparameters search has been performed over learning rate values of $0.001, 0.0005, 0.0001$ and batch sizes of $10, 100$. For each run, we selected the model whose performance didn't improve in the successive 30 epochs of training (i.e. we chose the model at epoch 10 if all the values of the loss for epochs 11 to 40 were higher then the value after 10 epochs). The best hyperparameters selection is shown in table 3.

Results on real data In order to show the viability of our method in comparison with well-established methods we perform, as in (Papamakarios et al., 2017), unconditional density estimation on four different UCI datasets (Dua and Graff, 2017) and a dataset of natural image patches (BSDS300) (Martin et al., 2001), as well as on MNIST (LeCun et al., 1998). We use a fairly simple feedforward neural network with a smooth version of leaky-ReLU as activation function. Note that, as we can perform every computation efficiently, all the experiments are suitable to run on usual hardware, thus avoiding the need of hardware accelerators such as GPUs. As a final remark, no fine-tuning in the form of, for example, batch normalization, dropout, or learning-rate scheduling, was considered on any dataset.

First, we want to remark that the data used for the experi-

ments shown in table 4 was preprocessed in the exact same way as described in (Papamakarios et al., 2017).

For the results shown in table 4 (MNIST excluded) a more exhaustive hyperparameter search has been performed. Particularly, for each dataset a grid-search was run with the options shown in table 2, taking for each experiment the model with best validation log-likelihood obtained during training and, across experiments, getting the one with best test log-likelihood. Experiments were again trained using Adam and, instead of fixing the number of epochs, training was finished with an early-stopping criteria that evaluates the validation set every 25 epochs and has a patience of 5 trials. The best hyperparameters selection is shown in table 3.

Regarding the rest of the models shown in table 4, we reproduce the exact same experiments as described in (Papamakarios et al., 2017). Therefore, the considered models have the same architecture and stopping criteria as the ones shown in table 1 of the aforementioned paper. The only difference with respect to the results shown in table 1 of (Papamakarios et al., 2017) and table 4 of the main paper is the number of trainable parameters. As mentioned in section 2.1, in order to perform a fair comparison, we tweaked the hyperparameters of each architecture so they have approximately the same number of parameters.

Specifically, we first trained our model as described above and, once we knew the number of parameters of the best-performing model (which is approximately LD^2) we used the formulae shown in table 3 of (Papamakarios et al., 2017) to find to which values we should fix the hyperparameters L and H of their models so that they have the same number of parameters.

As a final remark, note that there is one degree-of-freedom in those equations (for every L there is a value of H solving the given equation). Therefore, for each of the considered models and datasets, we run two different experiments, one with $L = 1$ and another with $L = 2$ (as similarly done in (Papamakarios et al., 2017)), finding afterwards the proper value of H to match the number of trainable parameters of our best model for that same dataset. The results in Table 4 show that this system, despite having quite *minimal fine-tuning*, *achieves competitive results on all the considered datasets* compared with existing models—which are all tailored and fine-tuned for density estimation.

Table 2: Hyperparameters considered for the grid search.

	Option #1	Option #2	Option #3
activation	$\mathrm{s_L}, \alpha = 0.3$	$\mathrm{s_L}, \alpha = 0.01$	$\mathrm{s_T}$
no. layers	25	50	100
learning rate	0.001	0.0005	0.0001
batch size	10	50	100
base distribution	standard normal	hyperbolic secant	
bias	Yes	No	

Table 3: Hyperparameters for the results in table 4.

	POWER	GAS	HEPMASS	MINIBOONE	BSDS300	MNIST
activation	$\mathrm{s_L}, \alpha = 0.3$	$\mathrm{s_L}, \alpha = 0.3$	$\mathrm{s_L}, \alpha = 0.3$	$\mathrm{s_T}$	$\mathrm{s_T}$	$\mathrm{s_L}, \alpha = 0.01$
no. layers	50	100	50	25	25	2
learning rate	0.001	0.001	0.001	0.0001	0.0001	0.0001
batch size	100	100	50	100	100	10
base dist.	std normal	std normal	hyper. secant	std normal	hyper. secant	std normal
bias	Yes	Yes	No	Yes	No	Yes

Table 4: Results on unconditional density estimation for different datasets and models. Models use a similar number of parameters and results show mean and two standard deviations.

	POWER	GAS	HEPMASS	MINIBOONE	BSDS300	MNIST
Ours	0.065 ± 0.013	6.978 ± 0.020	-21.958 ± 0.019	-13.372 ± 0.450	151.12 ± 0.28	-1375.2 ± 1.4
MADE	-3.097 ± 0.030	3.306 ± 0.039	-21.804 ± 0.020	-15.635 ± 0.498	146.37 ± 0.28	-1380.8 ± 4.8
MADE MoG	0.375 ± 0.013	7.803 ± 0.022	-18.368 ± 0.019	-12.740 ± 0.439	150.84 ± 0.27	-1038.5 ± 1.8
Real NVP (5)	-0.459 ± 0.010	6.656 ± 0.020	-20.037 ± 0.020	-12.418 ± 0.456	151.76 ± 0.27	-1323.2 ± 6.6
Real NVP (10)	0.182 ± 0.014	8.357 ± 0.019	-18.938 ± 0.021	-11.795 ± 0.453	153.28 ± 1.78	-1370.7 ± 10.1
MAF (5)	-0.458 ± 0.016	7.042 ± 0.024	-19.400 ± 0.020	-11.816 ± 0.444	149.22 ± 0.28	-1300.5 ± 1.7
MAF (10)	-0.376 ± 0.017	7.549 ± 0.020	-25.701 ± 0.025	-11.892 ± 0.459	150.46 ± 0.28	-1313.1 ± 2.0
MAF MoG (5)	0.192 ± 0.014	7.183 ± 0.020	-22.747 ± 0.017	-11.995 ± 0.462	152.58 ± 0.66	-1100.3 ± 1.6