Relative gradient optimization of the Jacobian term in unsupervised deep learning

Gresele, Luigi

Neural Information Processing Systems Foundation
2020-12

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** [3,5]

[1]Max Planck Institute for Intelligent Systems, Tübingen, Germany
[2]Max Planck Institute for Biological Cybernetics, Tübingen, Germany
[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] Dept of Computer Science, University of Helsinki, Finland
luigi.gresele@tuebingen.mpg.de; giancarlo.fissore@inria.fr

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, which can typically be written as a product of its marginals — thus drawing a connection with the field of nonlinear independent component analysis. Deep density models have been widely used for this task, but their maximum 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. In this work, we propose a new approach for exact 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 naive approaches. This allows fast training with objective functions involving the log-determinant of the Jacobian, without imposing constraints on its structure, in stark contrast to autoregressive normalizing flows.

1 Introduction

Many problems of machine learning and statistics involve learning invertible transformations of complex, multimodal probability distributions into simple ones. One example is density estimation through latent variable models under a specified base distribution [51], which can also have applications in data generation [14, 33, 19] and variational inference [44]. Another example is nonlinear independent component analysis (nonlinear ICA), where we want to extract simple, disentangled features out of the observed data [27, 30].

One approach to learn such transformations, introduced in [50] in the context of density estimation, is to represent them as a composition of simple maps, the sequential application of which enables high expressivity and a large class of representable transformations. Deep neural networks parameterize functions of multivariate variables as modular sequences of linear transformations and component-wise activation functions, thus providing a natural framework for implementing that idea, as already proposed in [45].

[*] Equal contribution

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

Unfortunately, however, typical strategies employed in neural networks training do not scale well for objective functions like the aforementioned ones; in fact, through the change of variable formula, the logarithm of the absolute value of the determinant of the Jacobian appears in the objective. Its exact computation, let alone its optimization, quickly gets prohibitively computationally demanding as the data dimensionality grows.

A large part of the research on deep density estimation, generally referred to under the term *autoregressive normalizing flows*, has therefore been dedicated to considering a restricted class of transformations such that the computation of the Jacobian term is trivial [14, 44, 15, 34, 25, 12], thus imposing a tradeoff between computation and expressive power. While such models can approximate arbitrary probability distributions, the extracted features are strongly restricted based on the imposed triangular structure, which prevents the system from learning a properly disentangled representation. Other strategies involve the optimization of an approximation of the exact objective [5], and continuous-time analogs of normalizing flows for which the likelihood (or some approximation thereof) can be computed using relatively cheap operations [13, 19].

In this work, we provide an efficient way to optimize the exact maximum likelihood objective for deep density estimation as well as for learning disentangled representations by latent variable models. We consider a nonlinear, invertible transformation from the observed to the latent space which is parameterized through fully connected neural networks. The weight matrices are merely constrained to be invertible. 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, which was originally introduced as the natural gradient in the context of linear ICA [11, 2], and which can be feasibly computed. We show how this can be integrated with the usual backpropagation employed to compute gradients in neural network training, yielding an overall efficient way to optimize the Jacobian term in neural networks. This is a general optimization approach which is potentially useful for any objective involving such a Jacobian term, and is likely to find many applications in diverse areas of probabilistic modelling, for example in the context of Bayesian active learning for the computation of the information gain score [48], or for fitting the reverse Kullback-Leibler divergence in variational inference [54, 7].

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 [44] and thus on the class of functions they can represent. In contrast, our approach allows to efficiently optimize the exact objective for neural networks with arbitrary Jacobians.

In sections 2 and 3 we review maximum likelihood estimation for latent variable models, backpropagation and the Jacobian term for neural networks, and discuss the complexity of the naive approaches for optimizing the Jacobian term. Then in section 4 we discuss the relative gradient, and show how it can be integrated with backpropagation resulting in an efficient procedure. We verify empirically the computational speedup our method provides in section 5.

2 Background

2.1 Maximum likelihood for latent variable models

Consider a generative model of the form

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

where $\mathbf{s} \in \mathbb{R}^D$ is the latent variable, $\mathbf{x} \in \mathbb{R}^D$ represents the observed variable 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[2] of $\mathbf{f}$; $\mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^{D \times D}$ its Jacobian computed at the point $\mathbf{x}$, whose elements are the partial derivatives

[2] The forward transformation could also be parameterized, but here we only explicitly parameterize its inverse.

$[\mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})]_{ij} = \partial g^i_{\boldsymbol{\theta}}(\mathbf{x})/\partial x^j$; and $p_{\boldsymbol{\theta}}$ and p_s denote, respectively, the probability density functions of $\mathbf{x}$ and of the latent variable $\mathbf{s}$ under the specified model. In many cases, it is additionally assumed that the distribution of the latent variable is sufficiently simple; for example, that it factorizes in its components,

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_i \log p_i(\mathbf{g}^i_{\boldsymbol{\theta}}(\mathbf{x})) + \log|\det \mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})|\,. \tag{3}$$

In this case, the problem can be interpreted as nonlinear independent component analysis (nonlinear ICA), and the components of $\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})$ are estimates of the original sources $\mathbf{s}$.

Another variant of this framework can be developed to solve the problem that nonlinear ICA is, in general, not identifiable without additional assumptions [29]; that means, even if the data is generated according to the assumed model, there is no guarantee that the recovered sources bear any simple relationship to the true ones. In order to obtain identifiability, it is possible to consider models [27, 28, 30, 20] in which the latent variables are not *unconditionally* independent, but rather *conditionally* independent given an additional, observed variable $\mathbf{u} \in \mathbb{R}^d$,

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{u}) = \sum_i \log p_i(\mathbf{g}^i_{\boldsymbol{\theta}}(\mathbf{x})|\mathbf{u}) + \log|\det \mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})|\,, \tag{4}$$

where d can be equal to or different from D depending on the model.

Maximum likelihood estimation for the model parameters amounts to finding, through optimization, the parameters $\boldsymbol{\theta}^*$ such that the expectation of the likelihood given by the expression in equation (3) is maximized. For all practical purposes, the expectation will be substituted with the sample average. Specifically, for optimization purposes, we will be interested in the computation of a gradient of such term on mini-batches of one or few datapoints, such that stochastic gradient descent can be employed.

2.2 Neural networks and backpropagation

Neural networks provide a flexible parametric function class for representing $\mathbf{g}_{\boldsymbol{\theta}}$ through a sequential composition of transformations, $\mathbf{g}_{\boldsymbol{\theta}} = \mathbf{g}_L \circ \ldots \circ \mathbf{g}_2 \circ \mathbf{g}_1$, where L defines the number of layers of the network. When an input pattern $\mathbf{x}$ is presented to the network, it produces a final output $\mathbf{z}_L$ and a series of intermediate outputs. By defining $\mathbf{z}_0 = \mathbf{x}$ and $\mathbf{z}_L = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})$, we can write the forward evaluation as

$$\mathbf{z}_k = \mathbf{g}_k(\mathbf{z}_{k-1}) \;\text{ for } k = 1, \ldots, L\,. \tag{5}$$

Each module $\mathbf{g}_k$ of the network involves two transformations,

(a) a coupling layer $C_{\mathbf{W}_k}$, that couples the inputs to the layer with the parameters $\mathbf{W}_k$ to optimize;

(b) other arbitrary manipulations $\boldsymbol{\sigma}$ of inputs/outputs. Typically, these are element-wise nonlinear activation functions with fixed parameters; we can for simplicity think of them as operations of the form $\boldsymbol{\sigma}(\mathbf{x}) = (\sigma(x_1), \ldots, \sigma(x_n))$ applied to vector variables.

The resulting transformation can thus be written as $\mathbf{g}_k(\mathbf{z}_{k-1}) = \boldsymbol{\sigma}(C_{\mathbf{W}_k}(\mathbf{z}_{k-1}))$.

We will focus on fully connected modules, where the coupling $C_{\mathbf{W}}$ is simply a matrix-vector multiplication between the weights $\mathbf{W}_k$ and the input to the k-th layer; overall, the transformation operated by such a module can be expressed as $\boldsymbol{\sigma}(\mathbf{W}_k\mathbf{z}_{k-1})$. Another kind of coupling layer is given by convolutional layers, typically used in convolutional neural networks [36].

The parameters of the network are randomly initialized and then learned by gradient based optimization with an objective function $\mathcal{L}$, which is a scalar function of the final output of the network. At each learning step, updates for the weights are proportional to the partial derivative of the loss with respect to each weight.

The computation of these derivatives is typically performed by backpropagation [47], a specialized instance of automatic differentiation. Backpropagation involves a two-phase process. Firstly, during a *forward pass*, the intermediate and final outputs of the network $\mathbf{z}_1, \ldots, \mathbf{z}_L$ are evaluated and a value for the loss is returned. Then, in a second phase termed *backward pass*, derivatives of the loss with respect to each individual parameter of the network are computed by application of the chain rule. The gradients are computed one layer at a time, from the last layer to the first one; in the process, the intermediate outputs of the forward pass are reused, employing dynamic programming to avoid redundant calculations of intermediate, repeated terms.[3]

In matrix notation, the updates for the weights of the k-th fully connected layer $\mathbf{W}_k$ can then be written as

$$\Delta \mathbf{W}_k \propto \mathbf{z}_{k-1} \boldsymbol{\delta}_k^\top , \tag{6}$$

where $\boldsymbol{\delta}_k$ is the cumulative result of the backward computation in the backpropagation step up to the k-th layer, also called backpropagated error. We report the full derivation in appendix A. We adopt the convention of defining $\mathbf{x}$, $\mathbf{z}_k$ and $\boldsymbol{\delta}_k$ as column vectors.

2.3 Difficulty of optimizing the Jacobian term of neural networks

In the case of the objective function specified in Eq. (3), we have $\mathcal{L}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x})$. By defining

$$\mathcal{L}_p(\mathbf{x}) = \sum_i \log p_i(\mathbf{g}^i_{\boldsymbol{\theta}}(\mathbf{x})); \quad \mathcal{L}_J(\mathbf{x}) = \log |\det \mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})| \ , \tag{7}$$

the objective can be rewritten as $\mathcal{L}(\mathbf{x}) = \mathcal{L}_p(\mathbf{x}) + \mathcal{L}_J(\mathbf{x})$. The evaluation of the gradient of the first term $\mathcal{L}_p$ can be performed easily if a simple form for the latent density is chosen, as it only requires simple operations on top of a single forward pass of the neural network. Given that the loss is a scalar, as backpropagation is an instance of reverse mode differentiation [4], backpropagating the error relative to it in order to evaluate the gradients does not increase the overall complexity with respect to the forward pass alone.

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 again be performed through automatic differentiation. Overall, it can be shown [4] 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 special neural network architectures for which the evaluation of $\mathcal{L}_J$ is trivial. For example, in autoregressive normalizing flows [14, 15, 34, 25] 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 still quadratic in the input size [44].

Most critically, such architectures imply a strong restriction on the class of transformations that can be learned. While it can be shown, based on [29], that under certain conditions this class of functions has universal approximation capacity for *densities* [25], that is less general than other notions of universal approximation [23, 24]. In fact it is obvious that functions with such triangular Jacobians cannot be universal approximators 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 [27, 30], which would usually require unconstrained Jacobians. In other words, such restrictions might imply that the deployed networks are not general purpose: [5] showed that constrained designs typically used for density estimation can severely hurt discriminative performance. We further elaborate on this point in appendix E. Note that fully connected modules have elsewhere been termed *linear* flows [42], and are a strict generalization of autoregressive flows.[4]

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

[3] Note that invertible neural networks provide the possibility to not save, but rather recompute the intermediate activations during the backward pass, thus providing a memory efficient approach to backpropagation [18].

[4] Comprehensive reviews on normalizing flows can be found in [42, 35]. Other related methods are reviewed in appendix B.

differentiable transformations are *composable*; given any two such transformations, their composition is also invertible and differentiable. Furthermore, the determinant of the Jacobian of a composition of functions is given by the product of the determinants of the Jacobians of each function,

$$\det \mathbf{J}[\mathbf{g}_2 \circ \mathbf{g}_1](\mathbf{x}) = \det \mathbf{J}\mathbf{g}_2\left(\mathbf{g}_1(\mathbf{x})\right) \cdot \det \mathbf{J}\mathbf{g}_1(\mathbf{x})\,. \tag{8}$$

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 term relative to 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

$$\mathcal{L}_J(\mathbf{z}_{k-1}) = \sum_{i=1}^{D} \log\left|\sigma'(y_k^i)\right| + \log\left|\det \mathbf{W}_k\right| =: \mathcal{L}_J^1(\mathbf{y}_k) + \mathcal{L}_J^2(\mathbf{z}_{k-1})\,. \tag{9}$$

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 it can be evaluated easily with few additional operations on top of intermediate outputs of a forward pass; gradients with respect to it can be simply computed via backpropagation, not unlike the $\mathcal{L}_p$ term introduced in section 2.3.

The second term $\mathcal{L}_J^2$ however involves a nonlinear function of the determinant of the weight matrix. From matrix calculus, we know that the derivative is equal to

$$\frac{\partial \log|\det \mathbf{W}_k|}{\partial \mathbf{W}_k} = \left(\mathbf{W}_k^\top\right)^{-1}\,. \tag{10}$$

Therefore, the computation of the gradient relative to such term involves a matrix inversion, with cubic scaling in the input size.[5] For a fully connected neural network of L layers, given that we have one such operation to perform for each of the layers, the gradient computation for these terms alone would have a complexity of $\mathcal{O}(LD^3)$, thus matching the one which would be obtained if the Jacobian were to be computed via automatic differentiation as discussed in section 2.

It can therefore be seen that these inverses of the weight matrices are the problematic element in the gradient computation. In the next section, we show how this problem can be solved using relative gradients.

4 Relative gradient descent for neural networks

We now derive the basic form of the relative gradient, following the approach in [11].[6] The starting point is that the parameters in a neural networks are matrices, in particular invertible in our case. Thus, we can make use of the geometric properties of invertible matrices, while they are usually completely neglected in gradient optimization in neural networks.

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

$$\mathbf{W}_k \to (\mathbf{I} + \boldsymbol{\epsilon})\mathbf{W}_k \tag{11}$$

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

$$f((\mathbf{I}+\boldsymbol{\epsilon})\mathbf{W}_k) - f(\mathbf{W}) = \langle \nabla f(\mathbf{W}_k), \boldsymbol{\epsilon}\mathbf{W}_k\rangle + o(\mathbf{W}_k) = \langle \nabla f(\mathbf{W}_k)\mathbf{W}_k^\top, \boldsymbol{\epsilon}\rangle + o(\mathbf{W}_k) \tag{12}$$

which shows that the direction of steepest descent in this case is given by making $\boldsymbol{\epsilon} = \mu\nabla f(\mathbf{W}_k)\mathbf{W}_k^\top$ 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

$$\mathbf{W}_k \to \mathbf{W}_k + \mu\nabla f(\mathbf{W}_k)\mathbf{W}_k^\top\mathbf{W}_k \tag{13}$$

[5] Though slightly more favorable exponents can in principle be obtained, see appendix C.

[6] For linear blind source separation, this approach also corresponds to the natural gradient, which can be justified with an information-geometric approach [2].

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

A further alternative can be obtained by perturbing the weight matrices from the right, as $\mathbf{W}_k \to \mathbf{W}_k(\mathbf{I}+\boldsymbol{\epsilon})$. A similar derivation shows that in this case, the optimal $\boldsymbol{\epsilon}$ is given by $\mathbf{W}_k\mathbf{W}_k^\top\nabla f(\mathbf{W}_k)$; we refer to this as *transposed relative gradient*. In the context of linear ICA, the properties of the relative and transposed relative gradient were discussed in [49]. This version of the relative gradient might be useful in some cases; for example, the transposed relative gradient can be implemented more straightforwardly in neural network packages where the convention is that vectors are represented as rows.

The relative gradient belongs to the more general class of gradient descent algorithms on Riemannian manifolds [1]. Specifically, relative gradient descent is a first order optimization algorithm on the manifold of invertible $D \times D$ matrices. Almost sure convergence of the parameters to a critical point of the gradient of the cost function can be derived even for its stochastic counterpart, with decreasing step size and under suitable assumptions (see e.g. [8]).

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 (10) by $\mathbf{W}_k^\top\mathbf{W}_k$ from the right we get

$$\left(\mathbf{W}_k^\top\right)^{-1}\mathbf{W}_k^\top\mathbf{W}_k = \mathbf{W}_k\,, \tag{14}$$

and similarly when multiplying by $\mathbf{W}_k\mathbf{W}_k^\top$ from the left. 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 $\mathbf{W}_k$ matrix 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 in equation (6), which still need to be multiplied by $\mathbf{W}_k^\top\mathbf{W}_k$ to turn it into a relative gradient. We will next see that we can do this avoiding matrix-matrix multiplications, which would be computationally expensive. Note that backpropagation necessarily computes the $\boldsymbol{\delta}_k$ vector in equation (6) and for our model, by applying the relative gradient carefully, we can avoid matrix-matrix multiplication altogether by computing

$$\left(\Delta\mathbf{W}_k\right)\mathbf{W}_k^\top\mathbf{W}_k \propto \mathbf{z}_{k-1}\left(\left(\boldsymbol{\delta}_k^\top\mathbf{W}_k^\top\right)\mathbf{W}_k\right)\,. \tag{15}$$

Thus, we have a cheap method for computing the gradient of the log-determinant of the Jacobian, and of our original objective function. In appendix D we provide an explanation of how our procedure can be implemented with relative ease on top of existing deep learning packages.

While we so far only discussed update rules for the weight matrices of the neural network, our approach can be extended to include biases. Including bias terms in our multilayer network endows it with stronger approximation capacity. We detail how to do this in appendix F.

Complexity Note that the parentheses in equation (15) stress the point that the relative gradient updates 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 do not 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 as in normalizing flow methods [44], but without imposing strong restrictions on the Jacobian of the transformation.

Extension to convolutional layers As we remarked in section 2.2, the formalism we introduced includes convolutional neural networks (CNNs) [36]. A natural question is therefore whether our approach can be extended to that case. The first natural question pertains the invertibility of convolutional neural networks; the convolution operation was shown [39] to be invertible under mild conditions (see appendix G), and the standard pooling operation can be by replaced an invertible operation [31]. We therefore believe that the general formalism can be applied to CNNs; this would require the derivation of the relative gradient for tensors. We believe that this should be possible but leave it for future work.

Invertibility and generation Given that invertible and differentiable transformations are composable, as discussed in section 3, invertibility of our learned transformation is guaranteed as long as the

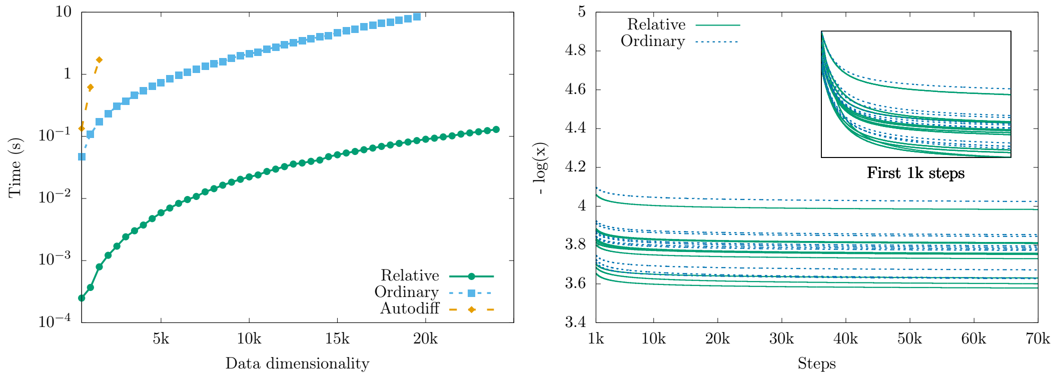

Figure 1: **Left:** Comparison of the average computation times of a single evaluation of the gradient of the log-likelihood; the standard error of the mean is not reported as it is orders of magnitude smaller then the scale of the plot. **Right:** Time-evolution of the negative log-likelihood for deterministic full-batch optimization for the two methods with the same initial points.

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 go to minus infinity for singular matrices (though high learning rates and numerical instabilities might compromise it in practice), as in estimation methods for linear ICA [6, 11, 26]. We additionally employ nonlinearities which are invertible by construction; we include more details about this in appendix H. 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 is available.

5 Experiments

In the following we experimentally verify the computational advantage of the relative gradient. The code used for our experiments can be found at `https://github.com/fissoreg/relative-gradient-jacobian`.

Computation of relative vs. ordinary gradient As a first step, we empirically verify that our proposed procedure using the formulas in section 4 leads to a significant speed-up in computation of the gradient of the Jacobian term. We compare the relative gradient against an explicit computation of the ordinary gradient, as described in section 3, and with a computation based on automatic differentiation, as discussed in section 2.3, where the Jacobian is computed with the JAX package [10]. While the output and asymptotic computational complexity of the ordinary gradient and automatic differentiation methods should be the same, a discrepancy is to be expected at finite dimensionality due to differences in how the computation is implemented. In the experiment, we generate 100 random normally distributed datapoints and vary the dimensionality of the data from 10 to beyond 20,000. We then define a two-layer neural network and evaluate the gradient of the Jacobian. The main comparison is run on a Tesla P100 Nvidia GPU. For the main plots, we deactivated garbage collection. Plots with CPU and further details on garbage collection can be found in appendix H.1. For each dimension we computed 10 iterations with a batch size of 100. Results are shown in figure 1, left. On the y-axis we report the average of the execution times of 100 successive gradient evaluations (forward plus backward pass in the automatic differentiation case). It can be clearly seen that *the relative gradient is much faster*, typically by two orders of magnitude. Autodiff computations could actually only be performed for the smallest dimension due to a memory problem. We report additional details on memory consumption in appendix H.1.

Optimization by relative vs. ordinary gradient Since our paper is, to the best of our knowledge, the first one proposing relative gradient optimization for neural networks (though other kinds of natural gradients have been studied [2]), we want to verify that the learning dynamics induced by the

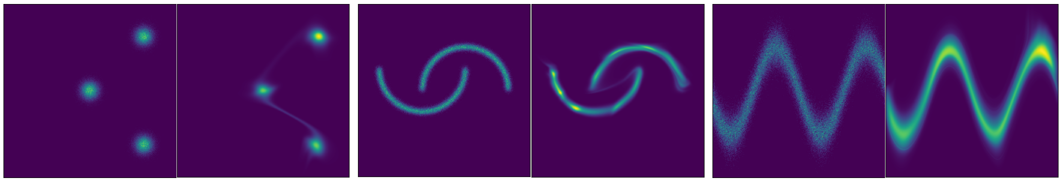

Figure 2: Illustrative examples of 2D density estimation. Samples from the true distribution and predicted densities are shown, in this order, side by side.

relative as opposed to the ordinary gradient do not bias the training procedure towards less optimal solutions or create other problems. We therefore perform a deterministic (full batch) gradient descent for both the relative and the ordinary gradient.[7] We employ 1,000 datapoints of dimensionality 2 and a two-layer neural network. We take 10 initial points and initialize both kinds of gradient descent at those same points. On the x-axis we plot the training epoch, while on the y-axis we plot the value of the loss. Figure 1, right 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 merely to show that the *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. Perhaps surprisingly, the results exhibit a rather constant speed-up factor of the order of 100 although the theory says it should be changing with the dimension D; in any case, the difference is very significant in practice.

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 to highlight the potential of the relative gradient used in conjuction with the unconstrained factorial approximation in section 2.1. We use a fairly simple feedforward neural network with a smooth version of leaky-ReLU as activation function. Our empirical results show that this system, despite having quite *minimal fine-tuning* (details in appendix H.3), *achieves competitive results on all the considered datasets* compared with existing models—which are all tailored and fine-tuned for density estimation. First, we show in Figure 2 different toy examples that showcase the ability of our method to convincingly model arbitrarily complex densities. Second, in order to show the viability of our method in comparison with well-established methods we perform, as in [43], unconditional density estimation on four different UCI datasets [16] and a dataset of natural image patches (BSDS300) [41], as well as on MNIST [37]. The results are shown in Table 1. To achieve a fair comparison across models, the number of parameters was tuned so that the number of trainable parameters are as similar as possible. 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, the reported results make no use of batch normalization, dropout, or learning-rate scheduling. Therefore, it is sensible to expect even better results by including them in future work.

Table 1: Test log-likelihoods (higher is better) on unconditional density estimation for different datasets and models (same as in Table 1 of [43]). Models use a similar number of parameters; results show mean and two standard deviations. Best performing models are in bold. More details in appendix H.3

	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	$\mathbf{0.375 \pm 0.013}$	7.803 ± 0.022	$\mathbf{-18.368 \pm 0.019}$	-12.740 ± 0.439	150.84 ± 0.27	$\mathbf{-1038.5 \pm 1.8}$
Real NVP (10)	0.182 ± 0.014	$\mathbf{8.357 \pm 0.019}$	-18.938 ± 0.021	$\mathbf{-11.795 \pm 0.453}$	$\mathbf{153.28 \pm 1.78}$	-1370.7 ± 10.1
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
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

[7] Notice that there's no need to compare to autodiff in this case because the computed gradient should be exactly the same as the ordinary gradient with the formulas in section 3.

6 Conclusions

Using relative gradients, we proposed a new method for exact optimization of objective functions involving the log-determinant of the Jacobian of a neural network, as typically found in density estimation, nonlinear ICA, and related tasks. This allows for employing models which, unlike typical alternatives in the normalizing flows literature, have no strong limitation on the structure of the Jacobian. We use modules with fully connected layers, thus strictly generalizing normalizing flows with triangular Jacobians, while still supporting efficient combination of forward and backward pass. These neural networks can represent a larger function class than autoregressive flows, which, despite being universal approximators for density functions, can only represent transformations with triangular Jacobians. Our method can therefore provide an alternative in settings where more expressiveness is needed to learn a proper inverse transformation, such as in identifiable nonlinear ICA models.

The relative gradient approach proposed here is quite simple, yet rather powerful. 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.

Broader impact

As this paper presents novel theoretical results in unsupervised learning, the authors do not see any immediate ethical or societal concern. An important aspect of our paper is the improvement in computational efficiency with respect to naive methods. This can hopefully lead to reduced energy consumption to achieve comparable model performance.

Acknowledgments

A.H. was supported by a Fellowship from CIFAR, and by the DATAIA convergence institute as part of the "Programme d'Investissement d'Avenir", (ANR-17-CONV-0003) operated by Inria.
L.G. started working on this project while on an ELLIS exchange, hosted by the Parietal team at Inria, Saclay.
We thank Vincent Stimper, Patrick Putzky, Cyril Furtlehner and Ilyes Khemakhem for valuable comments on an earlier draft of this paper, and Isabel Valera and Guillaume Charpiat for helpful comments and tips. L.G. additionally thanks Roma Beaufret and Alexis Bozio for helpful support.

References

[1] P-A Absil, Robert Mahony, and Rodolphe Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009.

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

[3] Leemon Baird, David Smalenberger, and Shawn Ingkiriwang. One-step neural network inversion with pdf learning and emulation. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 966–971. IEEE, 2005.

[4] 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.

[5] Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *International Conference on Machine Learning*, pages 573–582, 2019.

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

[7] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

[8] Silvere Bonnabel. Stochastic gradient descent on riemannian manifolds. *IEEE Transactions on Automatic Control*, 58(9):2217–2229, 2013.

[9] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[10] 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.

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

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

[13] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6572–6583, 2018.

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

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

[16] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[17] Marc Finzi, Pavel Izmailov, Wesley Maddox, Polina Kirichenko, and Andrew Gordon Wilson. Invertible convolutional networks. 2019.

[18] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pages 2214–2224, 2017.

[19] 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.

[20] Luigi Gresele, Paul K. Rubenstein, Arash Mehrjou, Francesco Locatello, and Bernhard Schölkopf. The Incomplete Rosetta Stone problem: Identifiability results for multi-view nonlinear ICA. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, page 53. AUAI Press, 2019.

[21] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[22] Emiel Hoogeboom, Rianne van den Berg, and Max Welling. Emerging convolutions for generative normalizing flows. *arXiv preprint arXiv:1901.11137*, 2019.

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

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

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

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

[27] 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.

[28] Aapo Hyvärinen and Hiroshi Morioka. Nonlinear ICA of temporally dependent stationary sources. volume 54. Proceedings of Machine Learning Research, 2017.

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

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

[31] Jörn-Henrik Jacobsen, Arnold Smeulders, and Edouard Oyallon. i-RevNet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

[32] Mahdi Karami, Dale Schuurmans, Jascha Sohl-Dickstein, Laurent Dinh, and Daniel Duckworth. Invertible convolutional flow. In *Advances in Neural Information Processing Systems*, pages 5636–5646, 2019.

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

[34] 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.

[35] Ivan Kobyzev, Simon Prince, and Marcus A Brubaker. Normalizing flows: Introduction and ideas. *arXiv preprint arXiv:1908.09257*, 2019.

[36] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[37] 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.

[38] 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.

[39] Fangchang Ma, Ulas Ayaz, and Sertac Karaman. Invertibility of convolutional generative networks from partial measurements. In *Advances in Neural Information Processing Systems*, pages 9628–9637, 2018.

[40] Charles C Margossian. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):e1305, 2019.

[41] 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.

[42] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *arXiv preprint arXiv:1912.02762*, 2019.

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

[44] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538, 2015.

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

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

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

[48] Matthias W Seeger and Hannes Nickisch. Large scale variational inference and experimental design for sparse generalized linear models. *arXiv preprint arXiv:0810.0901*, 2008.

[49] Stefano Squartini, Francesco Piazza, and Ali Shawker. New Riemannian metrics for improvement of convergence speed in ICA based learning algorithms. In *2005 IEEE International Symposium on Circuits and Systems*, pages 3603–3606. IEEE, 2005.

[50] 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.

[51] 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.

[52] Jakub M Tomczak and Max Welling. Improving variational auto-encoders using Householder flow. *arXiv preprint arXiv:1611.09630*, 2016.

[53] Rianne Van Den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pages 393–402. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.

[54] Martin J Wainwright and Michael I Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305, 2008.

[55] Wikipedia. Computational complexity of mathematical operations — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Computational%20complexity%20of%20mathematical%20operations&oldid=958179308`, 2020. [Online; accessed 11-June-2020].

[56] Jianxin Wu. Introduction to convolutional neural networks. 2017.

APPENDIX

A Backpropagation in neural networks

We will follow [46], Chapter 7, section 7.3.3 for the notation. Let us define a two-layer neural network

$$\mathbf{g}_\theta(\mathbf{x}) = \boldsymbol{\sigma}\left(\mathbf{W}_2\boldsymbol{\sigma}\left(\mathbf{W}_1\mathbf{x}\right)\right) \tag{16}$$

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 objective function due to the terms $\mathcal{L}_p$ and $\mathcal{L}_J^1$ (the contribution due to $\mathcal{L}_J^2$ 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

$$\begin{aligned}h(x) &= \frac{\partial}{\partial x}\log x'|_{x'=x} &(17)\\ &= \frac{1}{x} &(18)\end{aligned}$$

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 $\sigma''(x) = \sigma(x)(1-\sigma(x))\,(1-2\sigma(x))$. 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

$$\boldsymbol{\delta}_k = \mathbf{D}_k\mathbf{W}_{k+1}\boldsymbol{\delta}_{k+1} + \mathbf{G}_k\mathbf{h}_k \tag{19}$$

Which results in the update rule

$$\Delta\mathbf{W}_k = -\mu\mathbf{z}_{k-1}\boldsymbol{\delta}_k^\top\,,$$

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 $\mathbf{W}_k^\top\mathbf{W}_k$ from the right.

$$\Delta\mathbf{W}_k = -\mu\mathbf{z}_{k-1}\boldsymbol{\delta}_k^\top\mathbf{W}_k^\top\mathbf{W}_k\,.$$

Terms in $\mathcal{L}_J^2$ The terms in $\mathcal{L}_J^2$, consisting of $\log|\mathbf{W}_k|$ give as gradient $\left(\mathbf{W}_k^\top\right)^{-1}$. This requires a $D\times D$ matrix inversion for each of the matrices. Our strategy to avoid it is to substitute the ordinary gradient with a relative gradient, where we multiply the gradient (with respect to the whole objective but for each layer separately) by $\mathbf{W}_k^\top\mathbf{W}_k$ 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

$$\Delta\mathbf{W}_k = -\mu(\mathbf{z}_{k-1}\boldsymbol{\delta}_k^\top\mathbf{W}_k^\top\mathbf{W}_k + \mathbf{W}_k)\,. \tag{20}$$

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

$$\Delta\mathbf{W}_k = -\mu\left(\mathbf{z}_{k-1}\left(\left(\boldsymbol{\delta}_k^\top\mathbf{W}_k^\top\right)\mathbf{W}_k\right) + \mathbf{W}_k\right)\,. \tag{21}$$

B Related work

In the following, we present a review of related work in tractable deep density estimation and invertible neural networks.

Normalizing flows The modern conception of normalizing flows was introduced in [50], which discussed density estimation through the composition of simple maps. In [45], it was then proposed that deep density models implemented through neural networks could be used in order to construct bijective maps to a representation space and obtain normalized probability density estimates. Since then, the focus mainly shifted to scalability; [14, 15] introduced scalable architectures, further refined in [33] to make them more scalable and suitable for practical applications; [44] applied the results to variational inference. Comprehensive reviews on normalizing flows can be found in [42, 35].

Autoregressive flows Autoregressive flows are among the most used in practice. They involve maps which can be written as $\mathrm{z}_i' = \tau(\mathrm{z}_i; \boldsymbol{h}_i)$, with $\boldsymbol{h}_i = c_i\left(\mathbf{z}_{<i}\right)$. τ is termed the *transformer* and is a strictly monotonic function of z_i, and c_i is termed the i-th *conditioner*. Its constraint is that the $i-$th conditioner can only take variables with dimension indices less than i as an input. This results in an overall transformation with a triangular Jacobian; the determinant is therefore tractable and can be computed in $\mathcal{O}(D)$ time. Autoregressive flows differ in the way the transformer and conditioner are implemented; most commonly used are affine autoregressive flows [14, 15, 34, 43, 33] and non-affine neural transformers [25].

Linear flows A strict generalization of autoregressive flows, where the Jacobian is not constrained to be triangular, is given by linear flows, which are essentially transformations of the form $\mathbf{z}' = \mathbf{W}\mathbf{z}$, where $\mathbf{W}$ is a $D\times D$ invertible matrix. The Jacobian of the trasformation is simply $\mathbf{W}$ and both computing and optimizing its determinant takes time $O(D^3)$ in general. To obtain a better scaling behaviour, [14] and [22] proposed to parameterize the invertible $\mathbf{W}$ matrix via matrix decomposition. One possibility is to compute the $\mathbf{PLU}$ decomposition of $\mathbf{W}$ and optimize the $\mathbf{L}$ and $\mathbf{U}$ triangular transformations. The drawback in this approach is that the permutation matrix $\mathbf{P}$ cannot be learned. A more flexible alternative is to consider the $\mathbf{QR}$ decomposition of $\mathbf{W}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is upper triangular. However computing $\mathbf{Q}$ in full generality requires $O(D^3)$ operations, matching the complexity of the naive optimization of linear flows. [52] showed that we can apply the $\mathbf{Q}$ transformation as a sequence of at most D symmetry transformations each taking linear time, effectively making it possible to compute and optimize the $\mathbf{QR}$ parameterization of $\mathbf{W}$ in $O(D^2)$ time; note however that the sequential nature of the computation makes the method unsuitable for optimization on hardware accelerators. An experimental comparison of the performance of the $\mathbf{PLU}$ and $\mathbf{QR}$ decompositions against the direct optimization of $\mathbf{W}$ is found in [22].

Flows based on residual transformations Another class of normalizing flows is based on invertible transformations of the form $\mathbf{z}' = \mathbf{z} + g_\phi(\mathbf{z})$; this kind of flows are termed *residual flows*. Two main approaches can be applied to build invertible residual flows: the first exploits the matrix determinant lemma and also results in determinants with $\mathcal{O}(D)$ computation time; however, there is no analytical way of computing their inverse. Examples of these approaches are Sylvester flows [53], planar flows [44] and radial flows [50, 44]. The second approach is that of contractive flows [5]: in this case, the determinant can not be computed simply; likelihood-based training of these models therefore needs to rely on a Hutchkinson's trace based approximation to the exact log-likelihood.

Continuous time flows A separate line of work focuses on building *continuous flows*; in these approaches, the flow's infinitesimal dynamics is parametrized in continuous time, and the corresponding transformation is then found by integration [13, 19]; Hamiltonian Flows [44] can also be regarded as such kind of flows.

Other works Recently, many works have proposed ways of incorporating convolutional modules in normalizing flows, for example see [33, 22, 32]. In particular, [17] presents a formalization of the problem which bears some similarities to ours, while focusing on convolutional layers instead of fully connected ones. Other work has been dedicated to constructing invertible neural networks, see for example [3, 31, 18].

C Complexity of mathematical operations involved in gradient computation

We want to characterize the complexity of computing

$$\nabla_{\boldsymbol{\theta}} \log |\det \mathbf{J}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})| \,, \tag{22}$$

where $\mathbf{g}_{\boldsymbol{\theta}}$ is a neural network.

We will first recapitulate the computational complexity of the main mathematical operations we employ (see e.g. [55]). Then we'll recapitulate the complexity of forward evaluation and backpropagation in neural networks. Finally, we'll discuss the implications on the complexity of computing the term in equation (22) with the three methods discussed in the paper — namely, based on automatic differentiation, the standard computation described in section 3 and the relative gradient based computation.

C.1 Matrix operations

Matrix-vector and vector-vector multiplication The multiplication of a $D \times D$ matrix with a $D \times 1$ vector scales as $\mathcal{O}(D^2)$. Same for the outer product between two vectors of dimension $D \times 1$.

Matrix-matrix multiplication For the multiplication of two square matrices of size $D \times D$

- An implementation of the Bareiss algorithm would scale as $\mathcal{O}(D^3)$;
- An implementation of the Strassen algorithm would scale as$\mathcal{O}(D^{2.807\cdots})$;
- An implementation of the Coppersmith-Winograd algorithm would scale as$\mathcal{O}(D^{2.373\cdots})$.

In practice, what is usually implemented in linear algebra libraries is some flavor of the Strassen algorithm (this is because the Coppersmith-Winograd algorithm, while having a more favorable asymptotic behaviour, is effectively slower if D is not extremely high).

Matrix inversion To find the inverse of a matrix of size $D \times D$

- An implementation of Gauss-Jordan elimination would scale as $\mathcal{O}(D^3)$;
- An implementation of the Strassen algorithm would scale as $\mathcal{O}(D^{2.807\dots})$;
- An implementation of the Coppersmith-Winograd algorithm would scale as $\mathcal{O}(D^{2.373\dots})$.

Determinant To find the determinant of a matrix of size $D \times D$

- An implementation of the Bareiss algorithm would scale as $\mathcal{O}(D^3)$;
- Algorithms based on fast matrix multiplication scale as$\mathcal{O}(D^{2.373\dots})$.

For simplicity, in most of our considerations on complexity we assume that the computation of the determinant, the computation of the inverse and the multiplication of two square matrices have cubic cost. Notice that the cost of these operations always dominates over that of matrix-vector and vector-vector multiplication.

C.2 Other operations involved in the Jacobian term computation

Other operations turn out to be ininfluent on the overall computational complexity. Namely logarithms, absolute values, sums have no relevant effect in terms of asymptotic scaling, since their computational cost is dominated by that of the most expensive matrix operations listed above.

C.3 Complexity of neural network operations

Forward pass in a neural network The complexity of the forward pass in a neural network depends on the neural network structure. For simplicity, we will consider fully connected Neural Networks, which due to their dense structure will provide an upper bound for the complexity of most of the nets used in practice. Given an input vector, the forward pass is comprised of a sequential series of matrix-vector operations, plus elementwise operations on the resulting vector. The matrix-vector operations dominate the complexity; for an L layer neural network, there are L such operations. Therefore, for data of dimensionality D, the complexity of a forward pass in a Neural Network for a single data sample is $\mathcal{O}(LD^2)$.

Minibatching The objectives should, in principle, be optimized on the full batch. Stochastic optimization [9] relies on the idea that the update steps in the optimization process can be performed on subsets of the whole training data, called minibatches. In practice these objectives will always be computed on minibatches, so the expected value must be substituted with its empirical estimate over a single minibatch. The minibatch size should in principle be considered when considering how the algorithm scales. In the remainder, however, we will neglect this term, as minibatches used in practice are usually quite small.

Gradient computation On top of this, we also need to consider the gradient computation. Since the gradient is taken over the scalar loss function, this implies (through backpropagation or reverse mode differentiation) no increase in the asymptotic computational cost. We further elaborate on this in the next section.

C.4 Computing the Jacobian with automatic differentiation

Jacobian through automatic differentiation Automatic differentiation [4] includes two main operational modes: the forward mode and the backward mode. Consider the computation of the Jacobian of a function $\mathbf{g}_\theta) : \mathbb{R}^D \to \mathbb{R}^d$. The complexity of computing the Jacobian will depend on whether we use forward or reverse mode AD. This changes the complexity of the operation:

- forward mode requires $D\,c\,$ops$(\mathbf{g}_\theta)$ operations, where D is the dimensionality of the data and c is a constant, $c < 6$ and typically $c \in [2, 3]$ (see [21]);
- reverse mode requires $d\,c\,$ops$(\mathbf{g}_\theta)$ operations.

In the case of dimensionality reduction, reverse mode differentiation (of which backpropagation represents an instance) is clearly more efficient. This is the case when the output of the function is scalar ($d = 1$); thus, this explains our claim that gradients computation with backpropagation implies no increase in the asymptotic computational cost with respect to the forward pass alone.

For neural networks where all layers (including input and output) have the same size, both methods result in the same complexity. So in that case neither is better in terms of computational complexity — though in practice it is known that reverse mode performs better [40]. For such neural networks (including those we consider) therefore, given that $\text{ops}(\mathbf{g}_\theta)$ is $\mathcal{O}(LD^2)$, the overall complexity of the Jacobian computation via automatic differentiation is $\mathcal{O}(LD^3)$.

The gradient of the objective can then be computed via backpropagation; however, the forward evaluation is what dominates the overall complexity.

Standard and relative gradient computations The evaluation of the two terms $\mathcal{L}_p$ and $\mathcal{L}_J^1$ requires a forward pass of the neural networks, thus scaling as $\mathcal{O}(LD^2)$. As we discussed, backpropagation to compute the gradient does not increase the overall cost. For $\mathcal{L}_J^2$, as we have shown, the gradient can be computed without need to actually evaluate the corresponding term (that is, side-stepping the determinant computation). However, the standard computation of the gradient still requires computing inverses of all the weight matrices, resulting in a cubic cost operation for each layer — thus utimately in $\mathcal{O}(LD^3)$ cost.

When using the relative gradient, this inversion can be avoided, and computing the gradients of $\mathcal{L}_J^2$ implies *no additional costs*. The overall cost of the gradient computation is therefore simply $\mathcal{O}(LD^2)$.

D Implementation details

To efficiently optimize our objective (e.g. equation (3) in the main paper) we need to implement a variant of the backpropagation algorithm as detailed in appendix A. In particular, we need to compute the updates (equation (15) in the main paper) avoiding expensive matrix-matrix multiplications. This section is devoted to the description of an implementation strategy that takes advantage of Automatic Differentiation (AD), in order to have full flexibility in the definition of our model architectures and loss functions.

Although all modern deep learning frameworks include automatic differentiation libraries, they implement the standard backpropagation algorithm. To implement our variant, we have two straightforward alternatives:

- tweak some existing AD libraries to let us access the extra terms we need;
- implement our own AD library with the extra functionality we need.

The second alternative is easily excluded as we don't want to reinvent the wheel and the development effort would be too much. The first alternative is somewhat viable, but not future proof; we would be faced with the need to support our own modifications on top of the AD library we use.

We obviate to these problems with a little trick: we introduce in our architectures some dummy layers to accumulate the partial results that the standard backpropagation computes in the backward pass. This approach solves the previous problems by being:

- universal: it can be easily implemented on top of whatever AD library that computes reverse-mode AD, without tweaking the internals of the library;
- efficient: the dummy layer operations are $\mathcal{O}(1)$.

D.1 The Accumulator layer

To obtain the gradient updates (20) we need to compute the $\boldsymbol{\delta}$ terms (19). To better understand what these terms represent, we can consider a simple 2-layers "scalar" network, i.e. a network in which inputs, outputs and weights are scalar values:

$$
\begin{aligned}
f(x;\boldsymbol{w}) &= w_2\sigma(w_1 x) \\
&= w_2\sigma(y_1) \\
&= w_2 z_1 \\
&= y_2
\end{aligned}
\tag{23}
$$

where $\boldsymbol{w}$ is the vector of scalar parameters, σ is the activation function of choice and

$$
y_1 = w_1 x, \quad y_2 = w_2 z_1, \quad z_1 = \sigma(y_1)\,.
$$

Given a loss function $\mathcal{L}$, the gradient of $\mathcal{L}$ with respect to w_1 is easily computed with application of the chain rule

$$
\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial z_1}\frac{\partial z_1}{\partial y_1}\frac{\partial y_1}{\partial w_1} \tag{24}
$$

In this simple case, it is easy to isolate δ in the gradient equation:

$$
\frac{\partial \mathcal{L}}{\partial w_1} = \delta_1 \frac{\partial y_1}{\partial w_1} \tag{25}
$$

Reverse mode AD libraries necessarily compute all the partial derivatives in (24) and thus the δ_1 term we need. Unfortunately, the partial results are usually not accessible by the users. To access such terms without dealing with the internals of the AD libraries, we can introduce a parameterized function

$$
a(x;\lambda) = x + \lambda
$$

and redefine our scalar network as

$$
f(x;\boldsymbol{w}) = w_2\sigma(a(y_1)) \tag{26}
$$

The gradient with respect to w_1 becomes

$$
\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial z_1}\frac{\partial z_1}{\partial a}\frac{\partial a}{\partial y_1}\frac{\partial y_1}{\partial w_1} \tag{27}
$$

The introduction of a is only a trick; in order not to modify the gradients nor the behaviour of the scalar network, we require

$$
\begin{aligned}
a(y_1) &= y_1 \\
\frac{\partial z_1}{\partial a} &= \frac{\partial z_1}{\partial y_1} \\
\frac{\partial a}{\partial y_1} &= 1
\end{aligned}
\tag{28}
$$

which is easily achieved by setting $\lambda = 0$.

The benefit of introducing this accumulator layer a is that now we can ask the AD library to compute the gradients with respect to the dummy parameter λ; it is easy to verify that

$$
\frac{\partial a}{\partial \lambda} = \delta_1 \tag{29}
$$

thus making it possible to obtain the δ terms we need to compute (20).

E Universal approximation capacity in normalizing flows

Universal approximation for densities is a property often discussed in the context of autoregressive normalizing flows. It can be shown, based on the proof of existence and non-uniqueness of solutions to the nonlinear ICA problem [29], that any distribution can be mapped onto a factorized base distribution by an invertible function with triangular Jacobian, provided that the function class used for this mapping is large enough. Normalizing flows with triangular Jacobians and a high number of parameters therefore have this approximation capacity (see e.g. [25]). However, they can obviously not represent all possible *functions* — but only those with triangular Jacobians. They can therefore not be used to learn proper inverse functions and perform useful feature extraction.

A more general notion of universal approximation is the one usually discussed in the neural network literature, that is — universal approximation for functions. It has been shown that standard multilayer feedforward networks can approximate any continuous function to any degree of accuracy. For example, [38] 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 proof, as universal approximation capacity wouldn't be possible without.

While the proof above does not directly apply to our case, since it requires hidden layers with arbitrary width, we discuss how to incorporate biases in our training procedure in appendix F, in order to increase the expressivity of our model. We describe the nonlinearities we employed in appendix H.

F Relative gradient for the augmented matrix

In order to allow for the training of neural networks with biases, we present a heuristic based on the fact that 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. [46]).

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

$$\begin{bmatrix} \mathbf{y} \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & \mathbf{W} & & \mathbf{b} \\ 0 & \dots & 0 & 1 \end{array}\right] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \overline{\mathbf{W}} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \tag{30}$$

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 $\overline{\mathbf{W}}_k^\top \overline{\mathbf{W}}_k$, 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 (30). 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

$$\left[\begin{array}{c|c} \mathbf{G} & \mathbf{g}_b \\ \hline \mathbf{g} & g \end{array}\right] \overline{\mathbf{W}}^\top \overline{\mathbf{W}} = \left[\begin{array}{c|c} \mathbf{G}\mathbf{W}^\top\mathbf{W} + \mathbf{g}_b\mathbf{b}^\top\mathbf{W} & \mathbf{G}\mathbf{W}^\top\mathbf{b} + \mathbf{g}_b\mathbf{b}^\top\mathbf{b} + \mathbf{g}_b \\ \hline \dots & \dots \end{array}\right] \tag{31}$$

The relative gradient updates we need are then given by

$$\Delta \mathbf{W} \to \mathbf{G}\mathbf{W}^\top\mathbf{W} + \mathbf{g}_b\left(\mathbf{b}^\top\mathbf{W}\right) \tag{32}$$

$$\Delta \mathbf{b} \to \mathbf{G}\left(\mathbf{W}^\top\mathbf{b}\right) + \mathbf{g}_b(1 + \mathbf{b}^\top\mathbf{b}) \tag{33}$$

Note that $\mathbf{G}$ is nothing more then the standard backpropagation update (6), thus we can efficiently compute $\Delta\mathbf{W}$ by avoiding matrix-matrix multiplications as in (15). For $\Delta\mathbf{b}$ we can directly avoid matrix-matrix multiplications by taking some care in the evaluation of (33).

G Convolutions

The convolutional neural network [56] is composed of modules whose main components are: (i) a convolution layer; (ii) a pooling layer; (iii) a nonlinearity.

The convolution operation We follow the same notation as in [56]. Typically, inputs to the convolution layers are order 3 tensors with size $H^l \times W^l \times D^l$. A convolution kernel is also an order 3 tensor with size $H \times W^l \times D^l$. If D convolutions are used, this results in a order 4 tensor $\mathbb{R}^{H \times W^l \times D^l \times D}$ of parameters. If the input is $H \times W^l \times D^l$ and the kernel size is $H \times W^l \times D^l \times D$, the convolution result has size $(H^l - H + 1) \times (W^l - W + 1) \times D$. In our setting, note that the number of channels which can be used in practice is constrained, due to the formula in equation (3), which requires the input and output dimensionalities to be equal.

Are convolutional neural networks invertible? The convolution operation was shown to be invertible under some mild conditions. See [39] and [17], section 3.3, describing how Gaussian (or Uniform) sampled $c \times c \times r \times r$ parameter tensors will yield invertible convolutional layers with probability 1.

The pooling layer can be substituted with an invertible counterpart (see [31], section 3; or [17], figure 3), which basically becomes a tensorial extension of the permutation operation. As usual, an invertible nonlinearity can be chosen.

Relative gradient for the convolution For a convolution layer that preserves the number of channels in the input, we can directly write the operation in the form of a square matrix. In this case we can compute the relative gradient as explained in section 4, and we can obtain the gradients with respect to the filter entries by careful application of the chain rule. We however leave the precise theoretical derivation and experiments for future work.

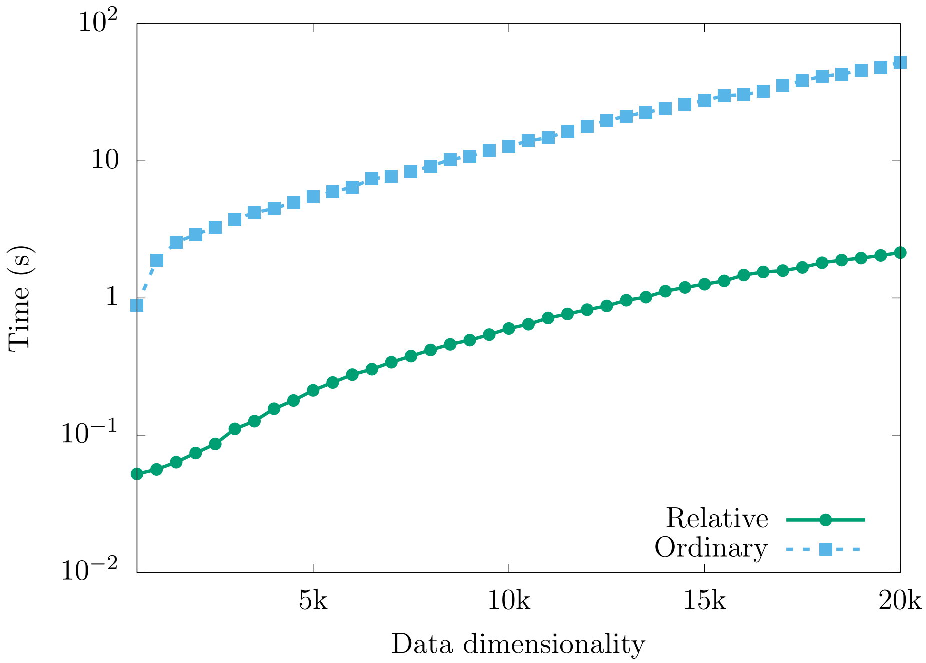

Figure 3: Comparison of the average computation times of a single evaluation of the gradient of the log-likelihood over a batch of size 100. Values are the mean over 5 steps, and the experiments have been run 5 times on a CPU cluster.

H Experiments

H.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 3 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.

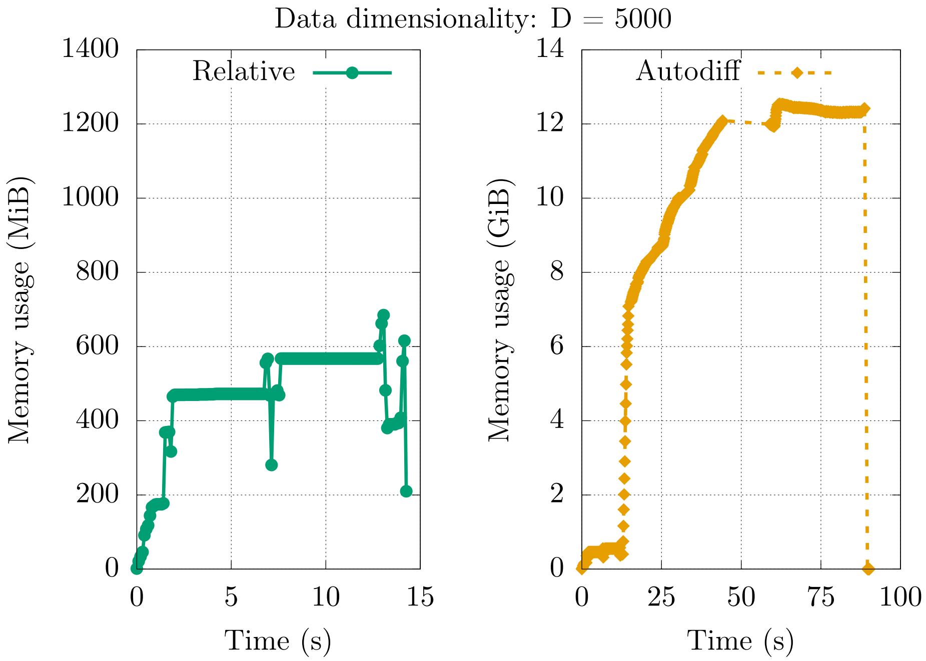

Figure 4: Comparison of the memory consumption for a single gradient evaluation. With D = 5000 our simplified analysis predicts a lower bound in the memory consumption of 400 MB for storing the parameters and the computed gradients; given that at startup time we observe a base memory consumption of almost 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.

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 4 we report a simple profiling of the memory consumption of the two methods, which shows how the difference is relevant in practice.

H.2 Relative gradient optimization behaviour with different optimizers

In this section we report some additional observations analyzing the relative gradient optimization behaviour with different optimizers.

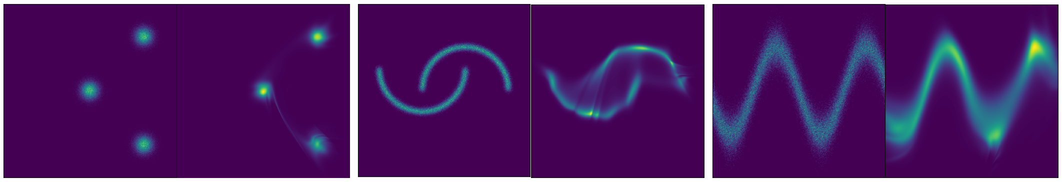

Figure 5: 2D toy examples trained with SGD. True distribution on the left, predicted densities on the right.

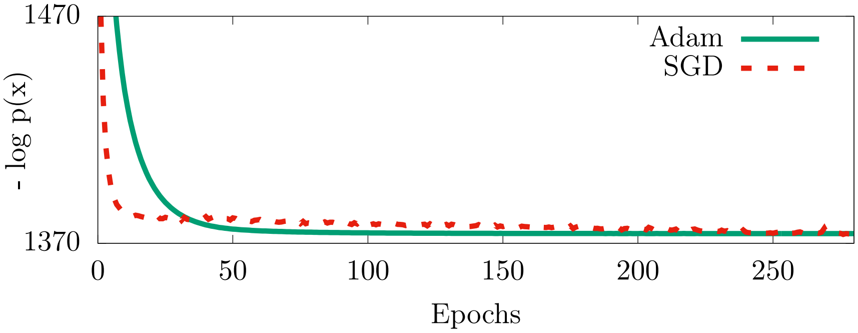

Figure 6: Log-likelihood evolution on MNIST validation set.

In figures 5 and 6 we compare the optimization behaviour using vanilla Stochastic Gradient Descent (SGD) and Adam. Results on toy datasets like those in figure 2 in the main paper are shown in figure 5. It can be seen that the data densities are modeled convincingly. We also report (figure 6) the evolution of the loss with SGD and Adam on density estimation on MNIST. The two methods seem to reach convergence at comparable speed: SGD is faster initially, but in the longer run Adam appears to achieve a better performance faster. Ultimately, both methods achieve a comparably good result.

H.3 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 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,

$$\mathrm{s_L}(x) = \alpha x + (1-\alpha)\log(1+e^x). \tag{34}$$

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

$$\mathrm{s_T}(x) = \tanh(\alpha x) + \beta x. \tag{35}$$

However, in our experiments, these two hyperparameters for the s_T nonlinearity are fixed to $\alpha = 1$ and $\beta = 0.1$ always. Both of these nonlinearities are relatively smooth, and while no closed form solution for their inverse is available they can be inverted easily with a Newton method; in practice, for our parameter choice, we use a fixed number of 100 iterations which seems to be (way) more than sufficient.

Toy examples For all the experiments shown in figure 2 of the main paper, 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 (that is, the distribution of the latent variables) 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 2.

Table 2: Hyperparameters used for figure 2 of the main paper.

	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 [43] has been applied. For the model architecture, we fixed the number of layers to 2, we used the smooth Leaky-ReLU (34) with $\alpha = 0.01$ and a standard normal distribution as a distribution for the latent variables. 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 did not 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 4.

Convergence time on MNIST To get an idea of the running time of our method in a real-world scenario, one epoch on MNIST ($D = 784$, 50k training samples) on a modern laptop CPU takes an order of tens of seconds, a $\sim 4.5\times$ speedup compared to “standard” optimization (which is roughly consistent with figure 3, which was obtained with a slightly different experimental setup) and $\sim 50\times$ speedup with respect to “autodiff”. Our convergence time is ~ 15 min. While the speed-up is already visible at this data dimensionality, the difference is expected to be larger at higher dimensionality.

Quantitative results First, we want to remark that the data used for the experiments shown in table 1 was pre-processed in the exact same way as described in [43].

For the results shown in such table (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 3, 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 4.

Table 3: Hyperparameters considered for the grid search.

	Option #1	Option #2	Option #3
activation	$s_{\mathrm{L}}, \alpha = 0.3$	$s_{\mathrm{L}}, \alpha = 0.01$	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	

Regarding the rest of the models shown in that table, we reproduce the exact same experiments as those described in [43]. 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 [43] and table 1 in our paper is the number of trainable parameters. As mentioned in section 5, 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

Table 4: Hyperparameters for the results in table 1 in the main paper.

	POWER	GAS	HEPMASS	MINIBOONE	BSDS300	MNIST
activation	$\mathrm{s}_\mathrm{L}, \alpha = 0.3$	$\mathrm{s}_\mathrm{L}, \alpha = 0.3$	$\mathrm{s}_\mathrm{L}, \alpha = 0.3$	s_T	s_T	$\mathrm{s}_\mathrm{L}, \alpha = 0.01$
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

in table 3 of [43] 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 [43]), finding afterwards the proper value of H to match the number of trainable parameters of our best model for that same dataset.