Self Normalizing Flows

T. Anderson Keller
UvA-Bosch Delta Lab
University of Amsterdam
Netherlands
t.a.keller@uva.nl

Jorn W.T. Peters
UvA-Bosch Delta Lab
University of Amsterdam
Netherlands

Priyank Jaini
UvA-Bosch Delta Lab
University of Amsterdam
Netherlands

Emiel Hoogeboom
UvA-Bosch Delta Lab
University of Amsterdam
Netherlands

Patrick Forré
University of Amsterdam
Netherlands

Max Welling
University of Amsterdam
Netherlands

Abstract

Efficient gradient computation of the Jacobian determinant term is a core problem of the normalizing flow framework. Thus, most proposed flow models either restrict to a function class with easy evaluation of the Jacobian determinant, or an efficient estimator thereof. However, these restrictions limit the performance of such density models, frequently requiring significant depth to reach desired performance levels. In this work, we propose *Self Normalizing Flows*, a flexible framework for training normalizing flows by replacing expensive terms in the gradient by learned approximate inverses at each layer. This reduces the computational complexity of each layer's exact update from $\mathcal{O}(D^3)$ to $\mathcal{O}(D^2)$, allowing for the training of flow architectures which were otherwise computationally infeasible, while also providing efficient sampling. We show experimentally that such models are remarkably stable and optimize to similar data likelihood values as their exact gradient counterparts, while surpassing the performance of their functionally constrained counterparts.

1 Introduction

The framework of normalizing flows [33] allows for powerful exact density estimation through the change of variables formula [30]. A significant challenge with this approach is the Jacobian determinant in the objective, which is generally expensive to compute. A significant body of work has therefore focused on methods to evaluate the Jacobian determinant efficiently by limiting the expressivity of the transformation. Two classes of functions have been proposed to achieve this: i) those with triangular Jacobians, such that the determinant only depends on the diagonal [7, 27, 18], and ii) those which are Lipschitz continuous such that Jacobian determinant can be approximated at each iteration through an infinite series [2, 12]. The drawback of both of these approaches is that they rely on strong functional constraints.

An insight which hints at a solution to this challenge is that the derivative of the log Jacobian determinant yields the inverse of the Jacobian itself. Recently, [13] leveraged this fact, and related work on Independant Component Analysis (ICA) [8, 1], to avoid computation of the inverse by using the natural gradient. Unfortunately, this method does not extend simply to convolutional layers.

In this work, we rely on this same insight and propose a new framework, called *self normalizing flows*, where flow components learn to approximate their own inverse through a self-supervised layer-wise reconstruction loss. We define a new density model as a mixture of the probability induced by both the forward and inverse transformations, and show how both transformations can be updated

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

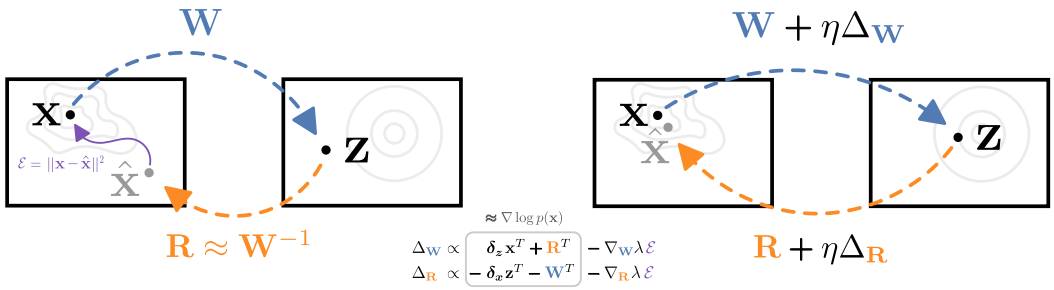

Figure 1: Overview of self normalizing flows. A matrix $\boldsymbol{W}$ transforms $\boldsymbol{x}$ to $\boldsymbol{z}$. The matrix $\boldsymbol{R}$ is constrained to approximate the inverse of $\boldsymbol{W}$ with a reconstruction loss $\mathcal{E}$. The likelihood is efficiently optimized by approximating the gradient of the log Jacobian determinant with the learned inverse.

symmetrically using their respective inverses directly in the gradient. Ultimately, this reduces the computational complexity of each training step from $\mathcal{O}(LD^3)$ to $\mathcal{O}(LD^2)$ where L and D are the numbers of layers and the dimensionality of each layer respectively.

The idea of greedy layer-wise learning has been explored in many forms for training neural networks [5, 14, 25]. One influential class of work uses stacked auto-encoders, or deep belief networks, for pre-training or representation learning [5, 14, 34, 21]. Our work leverages similar models into a modern flow framework by introducing the inverse weight matrix directly as an approximation of expensive terms in the gradient. Another class of work uses similar models to address the biological implausibility of backpropagation. Target propagation [22, 3, 23, 4] addresses the so-called weight transport problem by training auto-encoders at each layer of a network and using these learned feedback weights to propagate 'targets' to previous layers. Synthetic gradients [17] serve to alleviate the 'timing problem' of biological backpropagation by modelling error gradients at intermediate layers, and using these approximate gradients directly. Our method can also be seen in this light, and takes inspiration from all of these approaches. Specifically, our method can be viewed as a hybrid of target propagation and backpropagation [24, 35, 31] particularly suited to unsupervised density estimation in the normalizing flow framework. The novelty of our approach lies in the use of the inverse weights directly in the update, rather than in the backward propagation of updates.

2 A General Framework for Self Normalizing Flows

Given an observation $\boldsymbol{x} \in \mathbb{R}^D$, it is assumed that $\boldsymbol{x}$ is generated from an underlying real vector $\boldsymbol{z} \in \mathbb{R}^D$, through a series of invertible mappings $g_0 \circ g_1 \circ \dots g_K(\boldsymbol{z}) = \boldsymbol{x}$, where $\boldsymbol{z} \sim p_{\boldsymbol{Z}}(\boldsymbol{z})$, $\boldsymbol{z} = f_K \circ f_{K-1} \circ \dots f_0(\boldsymbol{x})$ and $f_k = g_k^{-1}$. We denote these compositions of functions as simply g and f respectively, i.e. $g \circ f(\boldsymbol{x}) = \boldsymbol{x}$. The base distribution $p_{\boldsymbol{Z}}$ is usually chosen to be simple to compute, such as a standard Gaussian. The probability density $p_{\boldsymbol{X}}$ can be computed using the change of variables formula:

$$p_{\boldsymbol{X}}(\boldsymbol{x}) = p_{\boldsymbol{Z}}(\boldsymbol{z}) \left| \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \right| = p_{\boldsymbol{Z}}\left(g^{-1}(\boldsymbol{x})\right) \left| \boldsymbol{J}_{g^{-1}} \right| = p_{\boldsymbol{Z}}\big(f(\boldsymbol{x})\big) \left| \boldsymbol{J}_f \right| \tag{1}$$

where the change of volume term $|\boldsymbol{J}_f| = \left| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right|$ is the determinant of the Jacobian of the transformation between $\boldsymbol{z}$ and $\boldsymbol{x}$, evaluated at $\boldsymbol{x}$. Typically, only the 'forward' functions f_k are defined and parameterized, and the inverses g_k are computed exactly when needed. The log-likelihood of the observations is then simultaneously maximized, with respect to a given f_k's vector of parameters $\boldsymbol{\theta}_k$, for all k, requiring the gradient. Using the identity $\frac{\partial}{\partial \boldsymbol{J}} \log |\boldsymbol{J}| = \boldsymbol{J}^{-T}$ yields:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} \log p^f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{\partial}{\partial \boldsymbol{\theta}_k} \log p_{\boldsymbol{Z}}(f(\boldsymbol{x})) + \frac{\partial \left(\operatorname{vec} \boldsymbol{J}_f\right)^T}{\partial \boldsymbol{\theta}_k} \left(\operatorname{vec} \boldsymbol{J}_f^{-T}\right) \tag{2}$$

where vec is vectorization through column stacking [26], and $\boldsymbol{\theta}_k$ is a column vector. In this work, in order to avoid the inverse Jacobian in the gradient, we instead propose to define and parameterize *both* the forward and inverse functions f_k and g_k with parameters $\boldsymbol{\theta}_k$ and $\boldsymbol{\gamma}_k$ respectively. We then propose to constrain the parameterized inverse g_k to be approximately equal to the true inverse f_k^{-1}

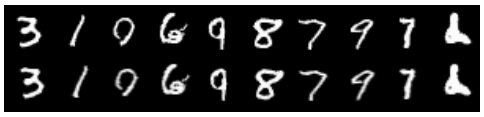

Figure 2: Samples from a self normalizing Glow-like model using the expensive exact inverse (top) vs. the fast approximate learned inverse (bottom).

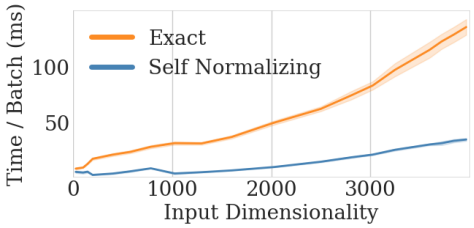

Figure 3: Time per batch vs. input dimensionality for a single fully connected layer, comparing self normalizing vs. exact gradient.

though a layer-wise reconstruction loss. We can thus define our maximization objective as the mixture of the log-likelihoods induced by both models minus the reconstruction penalty constraint, i.e.:

$$\mathcal{L}(\boldsymbol{x}) = \frac{1}{2}\log p^f_{\boldsymbol{X}}(\boldsymbol{x}) + \frac{1}{2}\log p^g_{\boldsymbol{X}}(\boldsymbol{x}) - \lambda \sum_{k=0}^{K} ||g_k\left(f_k(\boldsymbol{h}_k)\right) - \boldsymbol{h}_k||_2^2 \tag{3}$$

where $\boldsymbol{h}_k = \text{gradient\_stop}(f_{k-1} \circ ... f_0(\boldsymbol{x}))$ is the output of function f_{k-1} with the gradients blocked such that only g_k and f_k receive gradients from the reconstruction loss at layer k. We see that when $f = g^{-1}$ exactly, this is equivalent to the traditional normalizing flow framework. By the inverse function theorem, we know that the inverse of the Jacobian of an invertible function is given by the Jacobian of the inverse function, i.e. $\boldsymbol{J}_f^{-1}(\boldsymbol{x}) = \boldsymbol{J}_{f^{-1}}(\boldsymbol{z})$. Therefore, we see that with the above parameterization and constraint, we can approximate both the change of variables formula, and the gradients for both functions, in terms of the Jacobians of the respective inverse functions. Explicitly:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k}\log p^f_{\boldsymbol{X}}(\boldsymbol{x}) \approx \frac{\partial}{\partial \boldsymbol{\theta}_k}\log p_{\boldsymbol{Z}}(f(\boldsymbol{x})) + \frac{\partial\left(\text{vec}\,\boldsymbol{J}_f\right)^T}{\partial \boldsymbol{\theta}_k}\left(\text{vec}\,\boldsymbol{J}_g^T\right) \tag{4}$$

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}\log p^g_{\boldsymbol{X}}(\boldsymbol{x}) \approx \frac{\partial}{\partial \boldsymbol{\gamma}_k}\log p_{\boldsymbol{Z}}(g^{-1}(\boldsymbol{x})) - \frac{\partial\left(\text{vec}\,\boldsymbol{J}_g\right)^T}{\partial \boldsymbol{\gamma}_k}\left(\text{vec}\,\boldsymbol{J}_f^T\right) \tag{5}$$

where Equation 5 follows from the derivation of Equation 4 and the application of the derivative of the inverse. We note that the above approximation requires that the Jacobians of the functions are approximately inverses *in addition* to the functions themselves being approximate inverses. For the models presented in this work, this property is obtained for free since the Jacobian of a linear mapping is the matrix representation of the map itself.

Although there are no known convergence guarantees for such a method, we observe in practice that, with sufficiently large values of λ, most models quickly converge to solutions which maintain the desired constraint. Figure 2 gives an example of samples from the base distribution $p_{\boldsymbol{Z}}$ passed through both the true inverse f^{-1} (top) and the learned approximate inverse g (bottom) to generate samples from $p_{\boldsymbol{X}}$. As can be seen, the approximate inverse appears to be a very close match to the true inverse. The details of the model which generated these samples are in Appendix A.3.3.

3 Self Normalizing Flows

3.1 Self Normalizing Fully Connected Layer

As a specific case of the above model, we consider a single fully connected layer, as exemplified in Figure 1. Let $f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} = \boldsymbol{z}$, and $g(\boldsymbol{z}) = \boldsymbol{R}\boldsymbol{z}$, such that $\boldsymbol{W}^{-1} \approx \boldsymbol{R}$. Taking the gradients of Equation 3 for this model, and applying Equations 4 and 5, we get the following approximate gradients (see A.1 for details):

$$\frac{\partial}{\partial \boldsymbol{W}}\mathcal{L}(\boldsymbol{x}) = \frac{1}{2}\underbrace{\left(\frac{\partial}{\partial \boldsymbol{W}}\log p_{\boldsymbol{Z}}(\boldsymbol{W}\boldsymbol{x}) + \boldsymbol{W}^{-T}\right)}_{\approx \boldsymbol{\delta}_{\boldsymbol{z}}\boldsymbol{x}^T + \boldsymbol{R}^T} - \frac{\partial}{\partial \boldsymbol{W}}\lambda\mathcal{E} \tag{6}$$

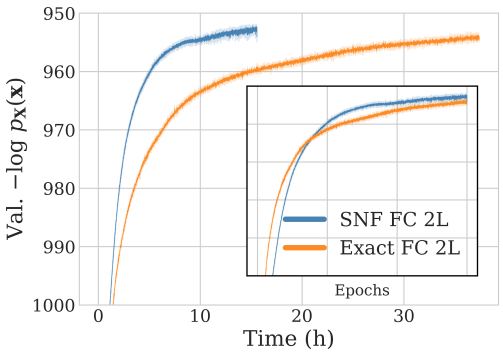

Figure 4: Neg. Log-likelihood on MNIST validation set for a 2-layer fully connected flow trained with exact vs. self normalizing gradient. Shown vs. training time, and epochs (inset).

Model	$-\log p_{\boldsymbol{X}}(\boldsymbol{x})$
Relative Grad. FC 2-Layer [13]	1096.5 ± 0.5
Exact Gradient FC 2-Layer	947.6 ± 0.2
SNF FC 2-Layer (ours)	947.1 ± 0.2
Emerging Conv. 9-Layer [16]	645.7 ± 3.6
SNF Conv. 9-Layer (ours)	638.6 ± 0.9
Conv. Exponential 9-Layer [15]	638.1 ± 1.0
Exact Gradient Conv. 9-Layer	637.4 ± 0.2
Glow-like 32-Layer [20]	575.7 ± 0.8
SNF Glow 32-Layer (ours)	575.4 ± 1.4

Table 1: Neg. Log-likelihood in nats on MNIST test set. Mean $\pm$ std. over 3 runs. Self normalizing flows (SNF) achieve comparable performance to their exact counterparts. See A.3 for details.

$$\frac{\partial}{\partial \boldsymbol{R}}\mathcal{L}(\boldsymbol{x}) = \frac{1}{2}\underbrace{\left(\frac{\partial}{\partial \boldsymbol{R}}\log p_{\boldsymbol{Z}}(\boldsymbol{R}^{-1}\boldsymbol{x}) - \boldsymbol{R}^{-T}\right)}_{\approx -\boldsymbol{\delta}_{\boldsymbol{x}}\boldsymbol{z}^T - \boldsymbol{W}^T} - \frac{\partial}{\partial \boldsymbol{R}}\lambda\mathcal{E} \tag{7}$$

where $\mathcal{E}$ denotes the reconstruction error, $\frac{\partial}{\partial \boldsymbol{W}}\lambda\mathcal{E} = 2\lambda\boldsymbol{R}^T(\hat{\boldsymbol{x}} - \boldsymbol{x})\boldsymbol{x}^T$, $\frac{\partial}{\partial \boldsymbol{R}}\lambda\mathcal{E} = 2\lambda(\hat{\boldsymbol{x}} - \boldsymbol{x})\boldsymbol{z}^T$, $\hat{\boldsymbol{x}} = \boldsymbol{R}\boldsymbol{W}\boldsymbol{x}$, $\boldsymbol{\delta}_{\boldsymbol{z}} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{z})}{\partial \boldsymbol{z}}$, and $\boldsymbol{\delta}_{\boldsymbol{x}} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{z})}{\partial \boldsymbol{x}}$ are ordinarily computed by backpropagation. We observe that by using such a self normalizing layer, the gradient of the log-determinant of the Jacobian term is approximately given by the weights of the inverse transformation, sidestepping computation of the Jacobian determinant and all matrix inverses.

3.2 Self Normalizing Convolutional Layer

To construct a self normalizing convolutional layer, let $f(\boldsymbol{x}) = \boldsymbol{w} \star \boldsymbol{x} = \boldsymbol{z}$, and $g(\boldsymbol{z}) = \boldsymbol{r} \star \boldsymbol{z}$, such that $f^{-1} \approx g$, where $\star$ is the convolution operation. We first note that the inverse of a convolution operation is not necessarily another convolution. However, for sufficiently large λ, we observe that f is simply restricted to the class of convolutions which is approximately invertible by a convolution. As we derive fully in Appendix A.2, the approximate self normalizing gradients with respect to a convolutional kernel $\boldsymbol{w}$, and corresponding inverse kernel $\boldsymbol{r}$, are given by:

$$\frac{\partial}{\partial \boldsymbol{w}}\log p^f_{\boldsymbol{X}}(\boldsymbol{x}) \approx \boldsymbol{\delta}_{\boldsymbol{z}} \star \boldsymbol{x} + \text{flip}(\boldsymbol{r}) \odot \boldsymbol{m} \tag{8}$$

$$\frac{\partial}{\partial \boldsymbol{r}}\log p^g_{\boldsymbol{X}}(\boldsymbol{x}) \approx -\boldsymbol{\delta}_{\boldsymbol{x}} \star \boldsymbol{z} - \text{flip}(\boldsymbol{w}) \odot \boldsymbol{m} \tag{9}$$

where $\text{flip}(\boldsymbol{r})$ corresponds to the kernel which achieves the transpose convolution and is given by swapping the input and output channels, and mirroring the spatial dimensions. The constant $\boldsymbol{m}$ is given by the number of times each element of the kernel $\boldsymbol{w}$ is present in the matrix form of convolution. The gradients for the reconstruction loss are then added to these.

4 Experiments

We evaluate our model by constructing simple flow architectures and training them to maximize Equation 3 on the MNIST dataset. We constrain the networks to be small such that we can compare directly with the same architectures trained using the exact gradient. As can be seen in Table 1 and Figure 4, the models with self normalizing flow layers are nearly identical in performance to the exact gradient counterparts, while taking significantly less time to train. Additionally, we see that the self normalizing flow layer outperforms its constrained convolutional counterpart from [16], and the relative gradient method of [13]. We hypothesize that the convolutional self normalizing flow model slightly underperforms the exact gradient method due to the convolutional-inverse constraint. We propose this constraint can be relaxed by using a fully connected inverse g (see A.2.2), but leave this

to future work. Additionally, in Figure 3, we compare our self normalizing flow model to the exact gradient in terms of computational complexity. We see that the self normalizing layer scales much more favorably than the exact gradient. Further experiment details can be found Appendix A.3.

5 Discussion

We see that the above framework yields an efficient update rule for flow-based models which appears to perform similarly to the exact gradient. The limitations include that that evaluation of the exact log-likelihood is still expensive, however this cost can be amortized over many samples, and that there are no known optimization guarantees (see A.4). In future work we intend to see how this framework scales to larger models, and how it could be combined with methods such as target propagation.

References

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

[2] Jens Behrmann, Will Grathwohl, Ricky T. Q. Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 573–582. PMLR, 2019.

[3] Yoshua Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. *CoRR*, abs/1407.7906, 2014.

[4] Yoshua Bengio. Deriving differential target propagation from iterating approximate inverses, 2020.

[5] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19: Annual Conference on Neural Information Processing Systems 2006, NeurIPS 2006 Montréal, Canada*, pages 153–160, 2006.

[6] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[7] Vladimir Igorevich Bogachev, Aleksandr Viktorovich Kolesnikov, and Kirill Vladimirovich Medvedev. Triangular transformations of measures. *Sbornik: Mathematics*, 196(3):309, 2005.

[8] Jean-François Cardoso and Beate Hvam Laheld. Equivariant adaptive source separation. *IEEE Transactions on signal processing*, 44(21):3017–3030, 1996.

[9] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. *CoRR*, abs/1605.08803, 2016.

[10] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows. In *Advances in Neural Information Processing Systems 32*, pages 7511–7522. 2019.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, 2010.

[12] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: free-form continuous dynamics for scalable reversible generative models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[13] Luigi Gresele, Giancarlo Fissore, Adrián Javaloy, Bernhard Schölkopf, and Aapo Hyvärinen. Relative gradient optimization of the jacobian term in unsupervised deep learning. *CoRR*, abs/2006.15090, 2020.

[14] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[15] Emiel Hoogeboom, Victor Garcia Satorras, Jakub M. Tomczak, and Max Welling. The convolution exponential and generalized sylvester flows. *CoRR*, abs/2006.01910, 2020.

[16] Emiel Hoogeboom, Rianne van den Berg, and Max Welling. Emerging convolutions for generative normalizing flows. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2771–2780. PMLR, 2019.

[17] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. volume 70 of *Proceedings of Machine Learning Research*, pages 1627–1635, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[18] Priyank Jaini, Kira A Selby, and Yaoliang Yu. Sum-of-squares polynomial flow. *arXiv preprint arXiv:1905.02325*, 2019.

[19] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[20] Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 10236–10245, 2018.

[21] Adam Kosiorek, Sara Sabour, Yee Whye Teh, and Geoffrey E Hinton. Stacked capsule autoencoders. In *Advances in Neural Information Processing Systems 32*, pages 15512–15522. 2019.

[22] Yann Lecun. Learning processes in an asymmetric threshold network. In *Disordered systems and biological organization, Les Houches, France*, pages 233–240. Springer-Verlag, 1986.

[23] Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Proceedings of the 2015th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*, ECMLPKDD'15, page 498–515, 2015.

[24] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[25] Sindy Löwe, Peter O Connor, and Bastiaan Veeling. Putting an end to end-to-end: Gradient-isolated learning of representations. In *Advances in Neural Information Processing Systems 32*, pages 3039–3051. 2019.

[26] Jan R. Magnus. On the concept of matrix derivative. *Journal of Multivariate Analysis*, 101(9):2200–2206, 2010.

[27] Youssef Marzouk, Tarek Moselhy, Matthew Parno, and Alessio Spantini. An introduction to sampling via measure transport. *arXiv preprint arXiv:1602.05023*, 2016.

[28] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 2335–2344, Red Hook, NY, USA, 2017.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. 2019.

[30] Walter Rudin. *Real and Complex Analysis. 1987*, volume 156. McGraw-Hill Book Company, 1987.

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

[32] Jascha Sohl-Dickstein. Two equalities expressing the determinant of a matrix in terms of expectations over matrix-vector products. 2020.

[33] 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.

[34] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. pages 1096–1103, 2008.

[35] Paul J. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kozin, editors, *System Modeling and Optimization*, pages 762–770, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

A Appendix

A.1 Derivation of Self Normalizing Fully Connected Gradients

In this section we derive Equations 6 and 7 in detail to make the proposed approximations explicit. First, as before, let $f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} = \boldsymbol{z}$, and $g(\boldsymbol{z}) = \boldsymbol{R}\boldsymbol{z}$, such that $\boldsymbol{W}^{-1} \approx \boldsymbol{R}$, with $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^D$. Taking the gradients of Equation 3 with respect to all parameters $\boldsymbol{W}$ and $\boldsymbol{R}$, we get the following exact gradients:

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{W}}\mathcal{L}(\boldsymbol{x}) &= \frac{1}{2}\frac{\partial}{\partial \boldsymbol{W}}\log p^f_{\boldsymbol{X}}(\boldsymbol{x}) + \frac{1}{2}\frac{\partial}{\partial \boldsymbol{W}}\log p^g_{\boldsymbol{X}}(\boldsymbol{x}) - \frac{\partial}{\partial \boldsymbol{W}}\lambda\mathcal{E} \\ &= \frac{1}{2}\left(\frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{W}\boldsymbol{x})}{\partial \boldsymbol{W}} + \frac{\partial \log|\boldsymbol{W}|}{\partial \boldsymbol{W}}\right) - \frac{\partial \lambda||\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x}||_2^2}{\partial \boldsymbol{W}} \\ &= \frac{1}{2}\left(\boldsymbol{\delta}^f_{\boldsymbol{z}}\boldsymbol{x}^T + \boldsymbol{W}^{-T}\right) - 2\lambda\boldsymbol{R}^T(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x})\boldsymbol{x}^T\end{aligned} \tag{10}$$

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{R}}\mathcal{L}(\boldsymbol{x}) &= \frac{1}{2}\frac{\partial}{\partial \boldsymbol{R}}\log p^f_{\boldsymbol{X}}(\boldsymbol{x}) + \frac{1}{2}\frac{\partial}{\partial \boldsymbol{R}}\log p^g_{\boldsymbol{X}}(\boldsymbol{x}) - \frac{\partial}{\partial \boldsymbol{R}}\lambda\mathcal{E} \\ &= \frac{1}{2}\left(\frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{R}^{-1}\boldsymbol{x})}{\partial \boldsymbol{R}} + \frac{\partial \log|\boldsymbol{R}^{-1}|}{\partial \boldsymbol{R}}\right) - \frac{\partial \lambda||\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x}||_2^2}{\partial \boldsymbol{R}} \\ &= \frac{1}{2}\left(-\boldsymbol{R}^{-T}\boldsymbol{\delta}^g_{\boldsymbol{z}}\boldsymbol{x}^T\boldsymbol{R}^{-T} - \boldsymbol{R}^{-T}\right) - 2\lambda(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x})(\boldsymbol{W}\boldsymbol{x})^T\end{aligned} \tag{11}$$

where $\boldsymbol{\delta}^f_{\boldsymbol{z}} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{W}\boldsymbol{x})}{\partial \boldsymbol{W}\boldsymbol{x}}$ and $\boldsymbol{\delta}^g_{\boldsymbol{z}} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{R}^{-1}\boldsymbol{x})}{\partial \boldsymbol{R}^{-1}\boldsymbol{x}}$ are ordinarily computed by backpropagation. To avoid computing any matrix inverse, we substitute $\boldsymbol{W}^{-1}$ with $\boldsymbol{R}$ in Equation 10, and symmetrically, all instances of $\boldsymbol{R}^{-1}$ with $\boldsymbol{W}$ in Equation 11. Additionally, we approximate $\boldsymbol{\delta}^f_{\boldsymbol{z}} \approx \boldsymbol{\delta}^g_{\boldsymbol{z}}$ in Equation 11 such that we do not need to 'forward propagate' through the inverse model, and can re-use the backpropagated error from the forward model. This corresponds to assuming that $\boldsymbol{R}^{-1}\boldsymbol{x} \approx \boldsymbol{W}\boldsymbol{x}$ and in practice we observe that this approximation is not more problematic than the original $\boldsymbol{R}^{-1} \approx \boldsymbol{W}$. Ultimately this results in the following approximations to gradient, as given in the main section:

$$\frac{\partial}{\partial \boldsymbol{W}}\mathcal{L}(\boldsymbol{x}) \approx \frac{1}{2}\left(\boldsymbol{\delta}^f_{\boldsymbol{z}}\boldsymbol{x}^T + \boldsymbol{R}^T\right) - 2\lambda\boldsymbol{R}^T(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x})\boldsymbol{x}^T \tag{12}$$

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{R}}\mathcal{L}(\boldsymbol{x}) &\approx \frac{1}{2}\left(-\boldsymbol{W}^T\boldsymbol{\delta}^f_{\boldsymbol{z}}\boldsymbol{x}^T\boldsymbol{W}^T - \boldsymbol{W}^T\right) - 2\lambda(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x})(\boldsymbol{W}\boldsymbol{x})^T \\ &= \frac{1}{2}\left(-\boldsymbol{\delta}^f_{\boldsymbol{x}}\boldsymbol{z}^T - \boldsymbol{W}^T\right) - 2\lambda(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x} - \boldsymbol{x})\boldsymbol{z}^T\end{aligned} \tag{13}$$

where $\boldsymbol{\delta}^f_{\boldsymbol{x}} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{W}\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{W}^T\delta^f_{\boldsymbol{z}}$. We see that by using such a self normalizing layer, the gradient of the log-determinant of the Jacobian term, which originally required an expensive matrix inversion at each iteration, is approximately given by the weights of the inverse transformation – sidestepping computation of both the Jacobian and the inverse. Additionally, we see many of the terms required for 13 are already computed by traditional backpropagation, making the update efficient. Finally, we note the above gradients can trivially be extended to compositions of such layers, combined with non-linearities, by substituting the appropriate deltas for each layer, and the corresponding layer inputs and outputs for $\boldsymbol{x}$ and $\boldsymbol{z}$ respectively.

A.2 Derivation of Convolution Gradients

To extend the self normalizing flow model from fully connected layers to convolutional layers, we first consider the setting where both the forward transformation f, and the inverse transformation g are convolutional with the same kernel size. We define the parameters of f to be the kernel $\boldsymbol{w}$ and similarly, the parameters of g to be the kernel $\boldsymbol{r}$. We then proceed as in section 3.2, letting $f(\boldsymbol{x}) = \boldsymbol{w} \star \boldsymbol{x} = \boldsymbol{z}$ and $g(\boldsymbol{z}) = \boldsymbol{r} \star \boldsymbol{z}$, such that $f^{-1} \approx g$ with $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^D$. As in the main section, we only derive the gradients of the log-likelihood for simplicity, ignoring the reconstruction loss term, but we note the gradient of the reconstruction loss can be derived in a similar manner to Equations 10 and 11.

To make the derivation easier, we note that the convolution operation is a linear operation, and can therefore be represented in matrix form. We define a transformation $\boldsymbol{W} = \mathcal{T}(\boldsymbol{w})$, which maps between the convolutional kernel $\boldsymbol{w}$ and the corresponding matrix form of the convolution:

$$\boldsymbol{z} = \mathcal{T}(\boldsymbol{w})\boldsymbol{x} = \boldsymbol{w} \star \boldsymbol{x} \tag{14}$$

In the following, we thus define $\boldsymbol{w}$ and $\boldsymbol{r}$ to be column vectors. Additionally, following the conventions of [26] for matrix derivatives, we make use of the vectorization operator vec which maps from $m \times n$ matricies to $mn \times 1$ column vectors by stacking the columns of the matrix sequentially.

Using this notation, we can use the chain-rule to compute the exact gradient of the log-likelihood with respect to the vectorized kernel weights $\boldsymbol{w}$ and $\boldsymbol{r}$:

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{w}} \log p^f_{\boldsymbol{X}}(\boldsymbol{x}) &= \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}} \left(\frac{\partial \log p_{\boldsymbol{Z}}\left(\mathcal{T}(\boldsymbol{w})\boldsymbol{x}\right)}{\partial \operatorname{vec} \mathcal{T}(\boldsymbol{w})} + \frac{\partial \log |\mathcal{T}(\boldsymbol{w})|}{\partial \operatorname{vec} \mathcal{T}(\boldsymbol{w})} \right) \\
&= \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}} \left(\operatorname{vec} \left[\boldsymbol{\delta}^f_{\boldsymbol{z}} \boldsymbol{x}^T \right] + \operatorname{vec} \mathcal{T}(\boldsymbol{w})^{-T} \right) \\
&= \boldsymbol{\delta}^f_{\boldsymbol{z}} \star \boldsymbol{x} + \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}} \left(\operatorname{vec} \mathcal{T}(\boldsymbol{w})^{-T} \right)
\end{aligned} \tag{15}$$

where we see the first term $\frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}} \left(\operatorname{vec} \left[\boldsymbol{\delta}^f_{\boldsymbol{z}} \boldsymbol{x}^T \right] \right)$ is given by the convolution $\boldsymbol{\delta}^f_{\boldsymbol{z}} \star \boldsymbol{x}^T$, as is usually done with backpropagation in convolutional neural networks. Then, given our soft constraint $f^{-1} \approx g$, we can approximate $\mathcal{T}(\boldsymbol{w})^{-T}$ with $\mathcal{T}(\boldsymbol{r})^T$ giving us:

$$\frac{\partial}{\partial \boldsymbol{w}} \log p^f_{\boldsymbol{X}}(\boldsymbol{x}) \approx \boldsymbol{\delta}^f_{\boldsymbol{z}} \star \boldsymbol{x} + \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}} \left(\operatorname{vec} \mathcal{T}(\boldsymbol{r})^T \right) \tag{16}$$

We then observe that the the matrix $\frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}}$ can be seen as an operator which, for each kernel element w_i, sums over all locations where the element w_i is present in the matrix $\mathcal{T}(\boldsymbol{w})$. Intuitively, since $\mathcal{T}(\boldsymbol{r})^T$ is another convolution matrix, the second term in Equation 16 simplifies to a constant multiple $\boldsymbol{m}$ (corresponding to the number of times w_i was present in the convolution matrix) multiplied by the kernel which achieves the transposed convolution:

$$\frac{\partial}{\partial \boldsymbol{w}} \log p^f_{\boldsymbol{X}}(\boldsymbol{x}) \approx \boldsymbol{\delta}^f_{\boldsymbol{z}} \star \boldsymbol{x} + \operatorname{flip}(\boldsymbol{r}) \odot \boldsymbol{m} \tag{17}$$

We detail how to compute this constant $\boldsymbol{m}$ in section A.2.1. The kernel which achieves the transposed convolution can be achieved by $\mathcal{T}^{-1}(\mathcal{T}(\boldsymbol{r})^T)$, or more simply by swapping the input and output axes, and mirroring the spatial (height and width) dimensions.

We see that the symmetric derivation can be obtained for the gradient with respect to the kernel $\boldsymbol{r}$. We outline this process briefly below:

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{r}} \log p^g_{\boldsymbol{X}}(\boldsymbol{x}) &= \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{r}))^T}{\partial \boldsymbol{r}} \left(\frac{\partial \log p_{\boldsymbol{Z}}\left(\mathcal{T}(\boldsymbol{r})^{-1}\boldsymbol{x}\right)}{\partial \operatorname{vec} \mathcal{T}(\boldsymbol{r})} + \frac{\partial \log |\mathcal{T}(\boldsymbol{r})^{-1}|}{\partial \operatorname{vec} \mathcal{T}(\boldsymbol{r})} \right) \\
&= \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{r}))^T}{\partial \boldsymbol{r}} \left(\operatorname{vec} \left[-\mathcal{T}(\boldsymbol{r})^{-T} \boldsymbol{\delta}^g_{\boldsymbol{z}} \boldsymbol{x}^T \mathcal{T}(\boldsymbol{r})^{-T} \right] - \operatorname{vec} \mathcal{T}(\boldsymbol{r})^{-T} \right) \\
&\approx \frac{\partial(\operatorname{vec} \mathcal{T}(\boldsymbol{r}))^T}{\partial \boldsymbol{r}} \left(\operatorname{vec} \left[-\mathcal{T}(\boldsymbol{w})^T \boldsymbol{\delta}^f_{\boldsymbol{z}} \boldsymbol{x}^T \mathcal{T}(\boldsymbol{w})^T \right] - \operatorname{vec} \mathcal{T}(\boldsymbol{w})^T \right) \\
&= -\boldsymbol{\delta}^f_{\boldsymbol{x}} \star \boldsymbol{z} - \operatorname{flip}(\boldsymbol{w}) \odot \boldsymbol{m}
\end{aligned} \tag{18}$$

A.2.1 Computing the Convolution Kernel Multiple

The constant $\boldsymbol{m}$ which is the same shape as the kernel, and is element-wise multiplied, is given by the number of times each element w_i of the kernel $\boldsymbol{w}$ is present in the matrix $\mathcal{T}(\boldsymbol{w})$. This can be easily computed as a convolution of two images filled entirely of $1's$, the first with the shape of the outputs, and the second with the shape of the inputs, i.e:

$$\boldsymbol{m} = \operatorname{ones\_like}(\boldsymbol{z}) \star \operatorname{ones\_like}(\boldsymbol{x}) \tag{19}$$

Note this convolution must be performed with the same parameters as the main convolution (e.g. padding, stride, grouping, dilation, etc.).

A.2.2 Convolutional f with Fully Connected g

As noted in the main section, constraining the forward transformation f to be approximately invertible by a convolution does restrict the class of possible functions f which can be learned. An alternative is to parameterize g as a fully connected layer, yielding the following approximate gradients:

$$\frac{\partial}{\partial \boldsymbol{w}} \log p^f_{\boldsymbol{X}}(\boldsymbol{x}) \approx \boldsymbol{\delta}^f_{\boldsymbol{z}} \star \boldsymbol{x} + \frac{\partial (\operatorname{vec} \mathcal{T}(\boldsymbol{w}))^T}{\partial \boldsymbol{w}} \left(\operatorname{vec} \boldsymbol{R}^T\right) \tag{20}$$

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{R}} \log p^g_{\boldsymbol{X}}(\boldsymbol{x}) &= -\boldsymbol{R}^{-T} \boldsymbol{\delta}^g_{\boldsymbol{z}} \boldsymbol{x}^T \boldsymbol{R}^{-T} - \boldsymbol{R}^{-T} \\ &\approx -\mathcal{T}(\boldsymbol{w})^T \boldsymbol{\delta}^f_{\boldsymbol{z}} \boldsymbol{x}^T \mathcal{T}(\boldsymbol{w})^T - \mathcal{T}(\boldsymbol{w})^T \\ &= -\boldsymbol{\delta}^f_{\boldsymbol{x}} \boldsymbol{z}^T - \mathcal{T}(\boldsymbol{w})^T \end{aligned} \tag{21}$$

We can see that the second terms of both Equations do not simplify nicely here, and are thus more computationally expensive than in the convolutional-g setting. We leave experiments with such a model to future work.

A.3 Experiment details

A.3.1 Training details

The negative log-likelihood values reported in Table 1 are computed on the 10,000 sample MNIST test set, using the saved model parameters from the best epoch as determined by performance on a 10,000 sample held-out validation set. The plots in figure 4, show the negative log-likelihood on this validation set for the 2-layer fully connected (FC) models. All values are reported, in nats, as mean $\pm$ standard deviation, as computed over 3 runs with different random initializations. All log-likelihood values are reported by computing the exact log Jacobian determinant of the full transformation (see A.4 for computational considerations).

All fully connected (FC) models were trained for 6000 epochs using Adam optimizer [19] with a batch size of 100, a learning rate of 1×10^{-4}, $\beta_1 = 0.9, \beta_2 = 0.999$, and reconstruction weight $\lambda = 1$. All convolutional (Conv.) models were trained for 1000 epochs with the same optimization parameters, but with a learning rate of 1×10^{-3}. The Glow-like models were trained for 250 epochs using the same optimizer settings as the Conv. models, but with a reconstruction weight of $\lambda = 100$.

All models are trained using a learning-rate warm-up schedule where the learning rate is linearly increased from 0 to its full value over the course of the first 10 epochs.

We note that our value for the Relative Gradient model [13] differs from the published result of -1375.2 ± 1.4. We found experimentally that when using the same setting as published in [13], our re-implementation achieved approximately -1102. We found the discrepancy to be due almost exactly to the log Jacobian determinant of the data-preprocessing steps (such as dequantization, normalization, and the logit transform), which we measure to sum to 272.7 ± 0.3. We thus assume the authors of [13] did not include the log Jacobian determinant of these steps in their reported values. We further note the numbers in Table 1 are using slightly different parameters than in [13], such as $\alpha = 0.3$ in the activation function, a batch size of 100, and a significantly longer training duration.

A.3.2 Timing details

Figure 3 was created by running a single fully connected layer (with no activation function) on a machine with an NVIDIA Titan X GPU and Intel Xeon E5-2640 v4 CPU. All data points were computed by taking the mean and standard deviation of the time required per batch over 10 epochs on the MNIST dataset. The times of the first and last 100 batches per epoch were ignored to reduce variance. The values reported in Table 2 were computed using the same protocol, but are shown specifically for MNIST-size inputs. The values reported in Table 3 again use the same protocol, but were computed on a machine with an NVIDIA GeForce 1080Ti GPU and Intel Xeon E5-2630 v3 CPU. Figure 4 and Figure 5 were created using the time results from Table 3. The discrepancy between training and sampling time for the FC models is due to the iterative optimization required to invert the Smooth Leaky ReLU activation. As can be seen, our proposed self normalizing layer yields both favorable training and sampling speed compared to existing methods.

A.3.3 Architectures

All models were trained using pre-processed data in the same as manner as [13, 28, 9]. This includes uniform dequantization, normalization, and logit-transformation. We additionally use a standard Gaussian as our base distribution $p_{\boldsymbol{Z}}$ for all models.

All 2-layer fully connected (FC) models use the Smooth Leaky ReLU (with $\alpha = 0.3$) activation (as in [13]). Weights of the forward model ($\boldsymbol{W}$'s) are initialized to identity plus noise drawn from a Xavier Normal [11] with gain 0.01. Weights of the inverse model ($\boldsymbol{R}$'s) are initialized to the transpose of the forward weights.

All 9-layer convolutional models are trained with spline [10] activations with individual parameters per pixel and 5-knots, kernels of size (3×3), and zero-padding of 1 on all sides. The convolutional models are additionally divided into three blocks, each of 3 layers, with 2 'squeeze' layers in-between the blocks. The squeeze layers move feature map activations from the spatial dimensions into the channel dimension, reducing spatial dimensions by a half and increasing the number of channels by 4 (as in [16]). Weights of the forward model ($\boldsymbol{w}$'s) are initialized with the dirac delta function (preserving the identity of the inputs) plus noise drawn from a Xavier Normal [11] with gain 0.01. Weights of the inverse model ($\boldsymbol{r}$'s) are initialized to $\mathrm{flip}(\boldsymbol{w})$.

The Glow-like models were constructed of $L = 2$ blocks of $K = 16$ steps each (as specified in [20]), where each block is composed of a squeeze layer and K-steps of flow. A split layer is placed between the two blocks. Each step of flow is composed of an act-norm layer, a (1×1) convolution, and an affine coupling layer. The coupling layers are constructed as in [20]. All convolutional weights were initialized to random orthogonal matrices.

A.4 Limitations

A.4.1 Evaluating the Log-Likelihood

Computing the exact log-likelihood of observations is notably still slow in the proposed self normalizing flow framework since it still requires computation of the exact log Jacobian determinant of the full transformation. However, as we describe next, once a model is trained, the Jacobian determinants of the layer transformations only need to be computed once, and can then be re-used for all future likelihood evaluation. Thus, the cost of the expensive Jacobian determinant computation is effectively amortized over all future samples.

The amortization described above is possible because, for square dimensionality preserving transformations, the log Jacobian determinant factorizes into a sum of the log determinants of the square components, i.e.:

$$\log |\boldsymbol{J}_f| = \sum_k \log |\boldsymbol{J}_{f_k}| \tag{22}$$

In this way, we can see that the expensive part of the computation of the log-likelihood (the Jacobian determinant of the layer transformations $\boldsymbol{W}$) is no longer data-dependant, and thus only needs to be computed once after the model is trained. The data-dependant terms (the Jacobian determinants of the activations, data pre-processing, etc.) are usually much faster and therefore not a limitation. We further propose, in the case when even a single evaluation of the Jacobian determinant is infeasible, estimators such as [32] could be used to yield approximate values of the log-likelihood.

A.4.2 Optimization Guarantees

We note that, to the best of our knowledge, there are no known optimization guarantees for our proposed model. Therefore, in theory, the model could end up deviating from the flow objective (maximizing the data likelihood) and instead minimize reconstruction error. In practice, we observe that this is rarely the case, and that reconstruction error stays extremely low for most models when initialized properly. In future work, we intend to explore the possibility of augmented Lagrangian methods, and other constrained optimization techniques, which could provide better convergence guarantees.

A.5 Acknowledgements

We would like to thank the creators of Weight & Biases [6] and PyTorch [29]. Without these tools our work would not have been possible.

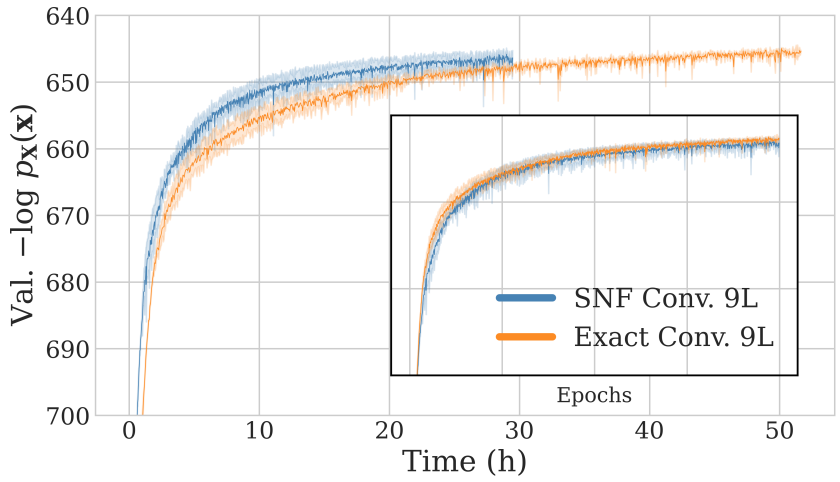

Figure 5: Log-likelihood on MNIST validation set for 9-layer convolutional models trained with exact gradient vs. self normalizing flow (SNF) gradient for 1000 epochs. Inset plot shows performance vs. training epochs, while the main plot shows performance vs. training time. We see the SNF model optimizes more quickly in terms of training time while still reaching comparable final performance. The difference in training time is not as significant as for the fully connected setting due to the computational complexity of the spline activation functions [10] which overshadow the log Jacobian determinant calculation for small networks.

Model	Time / batch (ms)	Time / sample (ms)
Exact FC	30.2 ± 1.5	15.8 ± 1.8
Self Normalizing FC	9.9 ± 0.2	1.0 ± 0.2
Exact Conv.	31.7 ± 3.1	15.1 ± 2.2
Self Normalizing Conv.	6.3 ± 0.6	1.0 ± 0.2

Table 2: Time comparison for a single layer training and sampling a batch of MNIST data.

Model	Time / batch (ms)	Time / sample (ms)
Exact FC 2-Layer	44.9 ± 4.4	61.5 ± 5.8
Relative Gradient FC 2-Layer [13]	7.0 ± 0.4	69.2 ± 5.6
Self Normalizing FC 2-Layer (ours)	18.7 ± 0.8	38.6 ± 3.1
Exact Conv. 9-Layer	372.2 ± 24.5	241.6 ± 12.9
Emerging Conv. 9-Layer [16]	305.0 ± 14.5	71.7 ± 8.8
Conv. Exponential 9-Layer [15]	304.4 ± 11.9	84.2 ± 9.2
Self Normalizing Conv. 9-Layer (ours)	212.5 ± 37.3	29.9 ± 6.3
Glow-like 32-Layer [20]	583.4 ± 21.2	163.1 ± 21.8
Self Normalizing Glow-like 32-Layer (ours)	476.4 ± 16.7	30.6 ± 2.2

Table 3: Runtime comparison for the models presented in Table 1. The discrepancy between training and sampling time for the FC models is due to the iterative Newton-Raphson optimization required to invert the Smooth Leaky ReLU activation. Details in A.3.2