# A. Appendix

## A.1. Self Normalizing Fully Connected Gradients

In this section we provide an extended derivation of Equations 10 and 11 including derivation of the reconstruction gradient. For the gradient with respect to $\boldsymbol{W}$, as given in Equation 10, we see that we only need to approximate the gradient of the Jacobian determinant term to achieve an efficient update, yielding:

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{W}}\mathcal{L}(\boldsymbol{x}) &\approx \frac{1}{2}\left(\boldsymbol{\delta}_{\boldsymbol{z}}^{f}\boldsymbol{x}^{T}+\boldsymbol{R}^{T}\right)-\lambda\frac{\partial}{\partial \boldsymbol{W}}||\boldsymbol{R}\boldsymbol{W}\boldsymbol{x}-\boldsymbol{x}||_{2}^{2}\\ &= \frac{1}{2}\left(\boldsymbol{\delta}_{\boldsymbol{z}}^{f}\boldsymbol{x}^{T}+\boldsymbol{R}^{T}\right)-2\lambda\boldsymbol{R}^{T}(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x}-\boldsymbol{x})\boldsymbol{x}^{T}\end{aligned} \tag{17}$$

For the gradient with respect to $\boldsymbol{R}$, we start with the exact gradient as given in Equation 9:

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{R}}\mathcal{L}(\boldsymbol{x}) = \frac{1}{2}\left(-\boldsymbol{R}^{-T}\boldsymbol{\delta}_{\boldsymbol{z}}^{g}\boldsymbol{x}^{T}\boldsymbol{R}^{-T}-\boldsymbol{R}^{-T}\right)\\ -\lambda\frac{\partial}{\partial \boldsymbol{R}}||\boldsymbol{R}\boldsymbol{W}\boldsymbol{x}-\boldsymbol{x}||_{2}^{2}\end{aligned} \tag{18}$$

To find an efficient approximation of this update without having to compute matrix inverses, we note that in addition to substituting $\boldsymbol{W}$ in the place of $\boldsymbol{R}^{-1}$, we also must approximate $\boldsymbol{\delta}_{\boldsymbol{z}}^{g}$ in order to avoid having to compute $\log p_{\boldsymbol{Z}}(\boldsymbol{R}^{-1}\boldsymbol{x})$. We propose this can similarly be approximated as $\boldsymbol{\delta}_{\boldsymbol{z}}^{g} \approx \boldsymbol{\delta}_{\boldsymbol{z}}^{f} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{W}\boldsymbol{x})}{\partial \boldsymbol{W}\boldsymbol{x}}$ under the same constraint that $\boldsymbol{R}^{-1} \approx \boldsymbol{W}$. Together this yields:

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{R}}\mathcal{L}(\boldsymbol{x}) \approx \frac{1}{2}\big(-\boldsymbol{W}^{T}\boldsymbol{\delta}_{\boldsymbol{z}}^{f}(\boldsymbol{W}\boldsymbol{x})^{T}-\boldsymbol{W}^{T}\big)\\ -2\lambda(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x}-\boldsymbol{x})(\boldsymbol{W}\boldsymbol{x})^{T}\end{aligned} \tag{19}$$

Conveniently, $\boldsymbol{W}^{T}\delta_{\boldsymbol{z}}^{f} = \frac{\partial \log p_{\boldsymbol{Z}}(\boldsymbol{W}\boldsymbol{x})}{\partial \boldsymbol{x}}$ is then the delta from the output backpropagated to the input of the layer, and is computed by standard backpropagation. We call this term $\boldsymbol{\delta}_{\boldsymbol{x}}^{f}$, giving a simplified gradient:

$$\frac{\partial}{\partial \boldsymbol{R}}\mathcal{L}(\boldsymbol{x}) \approx \frac{1}{2}\left(-\boldsymbol{\delta}_{\boldsymbol{x}}^{f}\boldsymbol{z}^{T}-\boldsymbol{W}^{T}\right)-2\lambda(\boldsymbol{R}\boldsymbol{W}\boldsymbol{x}-\boldsymbol{x})\boldsymbol{z}^{T} \tag{20}$$

We see then that both Equations 17 and 20 require no matrix inverses, and furthermore require no additional terms beyond those computed by standard backpropagation.

## A.2. Self Normalizing Convolution Gradients

In this section we provide a detailed derivation of the approximate gradient of the log Jacobian determinant term for convolutional layers. From Equations 14 and 16, we see that after substitution of the approximate inverse we have:

$$\frac{\partial}{\partial \boldsymbol{w}}\log p_{\boldsymbol{X}}^{f}(\boldsymbol{x}) \approx \boldsymbol{\delta}_{\boldsymbol{z}}^{f} \star \boldsymbol{x} + \frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{w}))^{T}}{\partial \boldsymbol{w}}\left(\operatorname{vec}\mathcal{T}(\boldsymbol{r})^{T}\right) \tag{21}$$

$$\frac{\partial}{\partial \boldsymbol{r}}\log p_{\boldsymbol{X}}^{g}(\boldsymbol{x}) \approx -\boldsymbol{\delta}_{\boldsymbol{x}}^{f} \star \boldsymbol{z} - \frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{r}))^{T}}{\partial \boldsymbol{r}}\left(\operatorname{vec}\mathcal{T}(\boldsymbol{w})^{T}\right) \tag{22}$$

Focusing on the second term, we consider what $\frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{k}))^{T}}{\partial \boldsymbol{k}}\left(\operatorname{vec}\mathcal{T}(\boldsymbol{p})^{T}\right)$ is for arbitrary kernels $\boldsymbol{k}, \boldsymbol{p} \in \mathbb{R}^{m}$, and matrices $\mathcal{T}(\boldsymbol{k}), \mathcal{T}(\boldsymbol{p}) \in \mathbb{R}^{D\times D}$. First, we see $\frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{k}))^{T}}{\partial \boldsymbol{k}} \in \mathbb{R}^{m\times D^{2}}$, meaning that it is a rectangular matrix where each row corresponds to one dimension of the kernel, and each column corresponds to one element of the matrix $\mathcal{T}(\boldsymbol{k})$. Writing out the partial derivative elementwise, we get:

$$\left[\frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{k}))^{T}}{\partial \boldsymbol{k}}\right]_{i,j} = \frac{\partial\left[\operatorname{vec}\mathcal{T}(\boldsymbol{k})\right]_{j}}{\partial k_{i}} \tag{23}$$

From this it is clear that the $i^{th}$ row and $j^{th}$ column will be 1 if $\left[\operatorname{vec}\mathcal{T}(\boldsymbol{k})\right]_{j} = k_{i}$ and otherwise 0. Intuitively, given the definiton of $\mathcal{T}(\boldsymbol{k})$, this corresponds to an indicator matrix, indicating whether kernel element $i$ is shared at location $j$ of the vectorized convolution matrix $\mathcal{T}(\boldsymbol{k})$.

Given this intuition, we can see that the inner product of the $i^{th}$ row of this matrix and the same vectorized convolution matrix $\operatorname{vec}\mathcal{T}(\boldsymbol{k})$ will yield a sum over elements of $\mathcal{T}(\boldsymbol{k})$ where $\mathcal{T}(\boldsymbol{k}) = k_{i}$. Formally:

$$\begin{aligned}\frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{k}))^{T}}{\partial k_{i}}\operatorname{vec}\mathcal{T}(\boldsymbol{k}) &= \sum_{j}^{D\times D} k_{i} * \mathbb{1}\left[\left[\operatorname{vec}\mathcal{T}(\boldsymbol{k})\right]_{j} = k_{i}\right]\\ &= m_{i} * k_{i}\end{aligned} \tag{24}$$

where $m_{i}$ is the number of times the kernel element $k_{i}$ is shared across the matrix $\mathcal{T}(\boldsymbol{k})$.

Similarly, for the inner product between a row of this indicator matrix and a vectorized convolution matrix formed from a different kernel (i.e. $\mathcal{T}(\boldsymbol{p})$), we see that the result is again given by a multiple times the new kernel:

$$\begin{aligned}\frac{\partial(\operatorname{vec}\mathcal{T}(\boldsymbol{k}))^{T}}{\partial k_{i}}\operatorname{vec}\mathcal{T}(\boldsymbol{p}) &= \sum_{j}^{D\times D} p_{i} * \mathbb{1}\left[\left[\operatorname{vec}\mathcal{T}(\boldsymbol{k})\right]_{j} = k_{i}\right]\\ &= m_{i} * p_{i}\end{aligned} \tag{25}$$

This can be understood to be due to the fact that the mapping $\mathcal{T}(\cdot)$ which generates the weight sharing structure is the same for both the partial derivative matrix and the vectorized convolution matrix.

**Computing the Transposed Convolution Kernel** $\operatorname{flip}(\cdot)$ In Equations 21 and 22 we see the partial derivative matrix is multiplied with a transposed convolution matrix. For the convolution operations proposed here, the transpose convolution matrix can also be written as a standard convolution matrix with a transformed kernel. We denote this transformation as $\operatorname{flip}(\cdot)$ such that $\mathcal{T}(\operatorname{flip}(\boldsymbol{k})) = \mathcal{T}(\boldsymbol{k})^{T}$, or equivalently $\operatorname{flip}(\boldsymbol{k}) = \mathcal{T}^{-1}(\mathcal{T}(\boldsymbol{k})^{T})$. This transformation can

easily be implemented in deep learning frameworks through index manipulation of the kernel. Informally, this operation is achieved by swapping the input and output axes, and mirroring the spatial dimensions. Explicitly, given a four dimensional kernel $\boldsymbol{k} \in \mathbb{R}^{O\times I\times H\times W}$ where $O, I, H, W$ are the number of output channels, number of input channels, kernel height, and kernel width respectively, the flip operation can be defined as:

$$\text{flip}(\boldsymbol{k})_{o,i,h,w} = k_{i,o,H-h,W-w} \tag{26}$$

**Computing the Multiple $\boldsymbol{m}$**

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 $k_i$ of the kernel $\boldsymbol{k}$ is present in the matrix $\mathcal{T}(\boldsymbol{k})$. 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. Using syntax from the PyTorch framework, we can write this as:

$$\boldsymbol{m} = \text{ones\_like}(\boldsymbol{z}) \star \text{ones\_like}(\boldsymbol{x}) \tag{27}$$

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

Combining Equation 25 with the flip operation, for all kernel elements $i$, we see that we arrive at Equations 15 and 16.

### A.3. Experiment Details

All code for this paper can be found at the following repository: https://github.com/AKAndykeller/SelfNormalizingFlows

**Training & Evaluation Details**

In Tables 1 and 2, all log-likelihood (in nats) and bits per dimension values are computed using the exact log Jacobian determinant of the full transformation. They are reported on a held-out test set, using the saved model parameters from the best epoch as determined by performance on a separate validation set. Each value is given as a mean $\pm$ standard deviation as computed over 3 runs with different random initializations.

For MNIST, the first 50,000 training images were used for training, and the last 10,000 were used for validation. The plots in figure 5, show the negative log-likelihood on the MNIST validation set for the 2-layer fully connected (FC) models.

For CIFAR-10, the first 40,000 training images were used for training and the last 10,000 were used as a held out validation set. Data augmentation including random horizontal flipping with probability 0.5 and random jitter by 1 pixel was performed to prevent overfitting.

For ImageNet 32x32, a random subset of 20,000 images were used for validation, and the remaining 1,261,149 images were used for training. The dataset was constructed using the same methodology as Kingma & Dhariwal 2018, and can be downloaded from http://image-net.org/small/download.php. The values reported in Table 2 were computed on the provided 50,000 image test set. No data augmentation was performed.

**Optimization Details**

All fully connected (FC) models were trained for 6000 epochs using Adam optimizer (Kingma & Ba, 2014) with a batch size of 100, a learning rate of $1 \times 10^{-4}$, $\beta_1 = 0.9, \beta_2 = 0.999$, and reconstruction weight $\lambda = 1$. These parameters were chosen to match the training methodology of (Gresele et al., 2020). The loss was computed as the average over the batch. All convolutional (Conv.) models were trained for 1000 epochs with the same optimization parameters, but with a learning rate of $1 \times 10^{-3}$ due to observed faster convergence.

The MNIST Glow models were trained for 250 epochs using the same optimizer settings as the Conv. models. We experimented with values of $\lambda$ in the set $\{1, 10, 100, 1000\}$, and chose the $\lambda = 100$ based on the highest validation accuracy.

The CIFAR-10 Glow models were trained for 1000 epochs with the same optimizer settings, but with $\lambda = 1000$. Additionally, the norm of the gradients was clipped at $10,000$ for the self normalizing CIFAR-10 models for improved stability during training.

The ImageNet 32x32 Glow models were trained for 15 epochs, with the same optimization parameters and again with $\lambda = 1000$. The norm of the gradients was clipped at $10,000$ for all ImageNet models.

All models except for the ImageNet models were 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.

**Discrepancies with Published Results**

We note that our value for the Relative Gradient model (Gresele et al., 2020) differs from the published result of $-1375.2 \pm 1.4$. We found experimentally that when using the same settings as published in (Gresele et al., 2020), 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 $272.7 \pm 0.3$. We discussed with the authors of (Gresele et al., 2020) and concluded they likely 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

(Gresele et al., 2020), such as $\alpha = 0.3$ in the activation function, a batch size of 100, and a significantly longer training duration.

We additionally see that our values in Table 2 are slightly worse than those reported in (Kingma & Dhariwal, 2018) (i.e. 3.36 vs. 3.35 on CIFAR-10, and 4.12 vs 4.09 on ImageNet 32x32). We hypothesize that our slightly worse performance is likely due to our use of an explicit validation set, reducing the effective size of the training set. To the best of our knowledge, this pre-processing step was not performed in (Kingma & Dhariwal, 2018). Additional factors could be a shorter training duration, no learning rate warmup (for ImageNet), or the imposed gradient clipping.

**Timing details**
Figure 4 was created by running a single fully connected layer (with no activation function) on a machine with an NVIDIA GeForce 1080Ti GPU and Intel Xeon E5-2630 v3 CPU. Each datapoint was computed by taking the mean and standard deviation of the time required per batch over 4,000 batches, with a batch size of 128, on synthetically generated random data at integer multiples of 96 dimensions starting at 32.

The values reported for MNIST in Table 3 were computed on the same machine with an NVIDIA GeForce 1080Ti GPU and Intel Xeon E5-2630 v3 CPU, with a batch size of 100. The values for CIFAR-10 and ImageNet were computed on a machine with an NVIDIA Titan X GPU and Intel Xeon E5-2640 v4 CPU, with batch sizes of 100 and 64 respectively. 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. Figure 5 was created using the time results from Table 3. For all times reported, the times of the first and last 100 batches per epoch were ignored to reduce variance.

**Architectures**
All models were trained using pre-processed data in the same as manner as (Gresele et al., 2020; Papamakarios et al., 2017; Dinh et al., 2016). 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 (Gresele et al., 2020)). Weights of the forward model ($\boldsymbol{W}$'s) are initialized to identity plus noise drawn from a Xavier Normal (Glorot & Bengio, 2010) 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 (Durkan et al., 2019) activations with individual parameters per pixel and 5-knots, kernels of size $(3 \times 3)$, 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 (Hoogeboom et al., 2019)). 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 (Glorot & Bengio, 2010) with gain $0.01$. Weights of the inverse model ($\boldsymbol{r}$'s) are initialized to $\text{flip}(\boldsymbol{w})$.

The Glow models for MNIST were constructed of $L = 2$ blocks of $K = 16$ steps each (as specified in (Kingma & Dhariwal, 2018)), 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 $(1 \times 1)$ convolution, and an affine coupling layer. The coupling layers are constructed as in (Kingma & Dhariwal, 2018). All convolutional weights were initialized to random orthogonal matrices. For CIFAR-10 and ImageNet 32x32, the Glow models were composed of $L = 3, K = 32$ and $L = 3, K = 48$ respectively, matching those in (Kingma & Dhariwal, 2018).

### A.4. Proposed Model Extensions

**Asymmetric Convolutions** As mentioned in Section 6, the inverse of the forward function may not always be given by a function of the same class (e.g. for convolutional layers). To partially alleviate this constraint, we propose that the forward and inverse functions may be asymmetric, and derive a simple case of this below for convolutions with different kernel sizes (but identical output sizes). Initial experiments with such an asymmetric model (i.e. 3×3 conv. $f$ with 7×7 conv. $g$) have shown promising results – improving over models with 3×3 convolutions in both directions.

For a function $f$ with a (column vector) kernel $\boldsymbol{w} \in \mathbb{R}^m$, and an inverse $g$ with a larger kernel $\boldsymbol{r} \in \mathbb{R}^n$ $(n > m)$, we see that the approximate gradient with respect to $\boldsymbol{w}$ can be obtained by taking the internal central dimensions of $\boldsymbol{r}$, and similarly the gradient for $\boldsymbol{r}$ is given by taking a zero-padded version of $\boldsymbol{w}$. Formally we can write the central-indexing and padding operations as multiplication by the rectangular matrices $\boldsymbol{P}_{\boldsymbol{r}\to\boldsymbol{w}}$ and $\boldsymbol{P}_{\boldsymbol{w}\to\boldsymbol{r}}$ respectively.

$$\frac{\partial}{\partial \boldsymbol{w}} \log p^f_{\boldsymbol{X}}(\boldsymbol{x}) \approx \boldsymbol{\delta}^f_{\boldsymbol{z}} \star \boldsymbol{x} + \text{flip}(\boldsymbol{P}_{\boldsymbol{r}\to\boldsymbol{w}}\boldsymbol{r}) \odot \boldsymbol{m} \tag{28}$$

$$\frac{\partial}{\partial \boldsymbol{r}} \log p^g_{\boldsymbol{X}}(\boldsymbol{x}) \approx -\boldsymbol{\delta}^f_{\boldsymbol{x}} \star \boldsymbol{z} - \text{flip}(\boldsymbol{P}_{\boldsymbol{w}\to\boldsymbol{r}}\boldsymbol{w}) \odot \boldsymbol{m} \tag{29}$$

Where $\boldsymbol{P}_{\boldsymbol{r}\to\boldsymbol{w}}$ and $\boldsymbol{P}_{\boldsymbol{w}\to\boldsymbol{r}}$ are defined as:

$$\boldsymbol{P}_{\boldsymbol{r}\to\boldsymbol{w}} = \begin{bmatrix} \mathbf{0} \; \mathbf{I}_m \; \mathbf{0} \end{bmatrix} \quad \& \quad \boldsymbol{P}_{\boldsymbol{w}\to\boldsymbol{r}} = \begin{bmatrix} \mathbf{0} \\ \mathbf{I}_n \\ \mathbf{0} \end{bmatrix} \tag{30}$$

Where $\mathbf{I}_D$ refers to a $D \times D$ identity matrix, and $\mathbf{0}$ is a matrix of zeros such that the dimensions of $\boldsymbol{P}_{\boldsymbol{r}\to\boldsymbol{w}}$ are $m \times n$ and those of $\boldsymbol{P}_{\boldsymbol{w}\to\boldsymbol{r}}$ are $n \times m$.

**Jacobian Vector Product Inverse Constraint** As noted in Section 3, the approximations made assume that the Jacobians of the functions $f$ and $g$ are approximately inverses in addition to the functions themselves being approximate inverses. As stated, 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. However, since this property may not hold in general, we propose the following additional constraint could be added to the objective:

$$\mathcal{E}_{JVP}(f,g) = -\mathbb{E}_{\nu\sim\mathcal{U}(0,1)}\left[||\boldsymbol{J}_g\boldsymbol{J}_f\nu - \nu||_2^2\right] \tag{31}$$

where $\boldsymbol{J}_g, \boldsymbol{J}_f$ are the Jacobians of $g$ and $f$ respectively, evaluated at $\boldsymbol{z}$ and $\boldsymbol{x}$ respectively. The expectation can additionally be approximated by Monte Carlo methods through a finite number of samples.

We see that such a loss would reach a minimum when the Jacobians are exact inverses. However, it remains unclear from which distribution the points $\nu$ should be sampled to achieve the best approximate inverse. Since Jacobian vector products are efficiently implemented in most deep learning frameworks, (and are thus much faster than naive matrix-matrix multiplication) this loss could be added to the overall objective while still avoiding $\mathcal{O}(D^3)$ computational complexity.

**Variational Interpretation** One limitation of the self normalizing flow framework becomes apparent mainly in the presence of data-dependant transformations, as would be contained in the general framework outlined in Section 3. Specifically, in this setting, the invertibility of the transformation is more difficult to guarantee since the value of the Jacobian determinant is then a function of the input, and therefore ensuring invertibility is not as simple as computing non-zero determinants on the training data.

To alleviate this difficulty, we believe a variational interpretation of the self normalizing framework as afforded by Gritsenko et al. (2019) would relax the global invertibility constraint to a local invertibility constraint. To achieve this, the encoder and decoder become stochastic with a very small noise level $\sigma^2$, and in the limit of $\sigma^2 \to 0$, the variational lower bound becomes equal to change of variables formula. In such a model, the inverse approximation error would contribute to the decoder likelihood directly, and the 'self normalizing' inverse approximation would become useful in computing the gradient of the entropy of the encoder. As the authors note, such an interpretation allows for non-invertible encoders, and we thus believe it is a fruitful direction for future research.

### A.5. Extended Results

**Choice of $\lambda$** As noted in the discussion, we observe that final likelihood values are only marginally impacted by the choice of $\lambda$. To quantitatively demonstrate this, we present the performance for different reconstruction weights $\lambda$ in Table 4 below. As can be seen, for values of $\lambda$ greater than a minimum threshold, the likelihood performance decreases slightly as $\lambda$ increases. For values of $\lambda$ which are too small, training is initially stable but eventually becomes unstable and diverges, resulting in numerical instability (denoted '-').

*Table 4.* NLL in nats on MNIST for SNF models with different values of $\lambda$. '-' implies the model training diverged.

| Model \ $\lambda$ | $\leq 0.1$ | 1.0 | 10.0 | 100.0 | 1000.0 |
|---|---|---|---|---|---|
| 2L FC | - | 946.3 | 954.6 | 1007.4 | 1039.8 |
| 9L Conv. | - | 639.6 | 642.9 | 646.3 | 660.6 |
| 2L-16K Glow | - | - | - | 574.1 | 574.8 |

**Improved Constrained Optimization** In this work, to enforce the approximate-inverse constraint $f^{-1} \approx g$, we make use of a penalty method with parameter $\lambda$ (the 'reconstruction weight'). The downsides of this method are that it requires manual tuning of $\lambda$ which can lead to sub-optimal local minima (as seen above). A powerful alternative to the basic penalty method is the method of Lagrange multipliers, whereby $\lambda$ is simultaneously optimized with the model parameters by a min-max optimization scheme. One related implementation of such a method is given by the GECO algorithm from Rezende & Viola (2018). Simply, the update equation of the algorithm is given by $\lambda^t \leftarrow \lambda^{t-1}\exp(\propto C^t)$ for each iteration $t$, where $C^t$ is derived from the exponential moving average of the constraint (i.e. reconstruction loss). In initial experiments, we have shown such a method works well when combined with the self normalizing framework, reducing the need for tuning, and enabling the stable training of more complex functions (see Table A.5 below). The implementation of this method is additionally provided in the code repository.

*Table 5.* NLL in nats on MNIST comparing the impact of larger kernel sizes incorporated into the Glow framework (2L-4K width=16). SNF models are trained with the method of Lagrange multipliers. Mean $\pm$ std over 3 random initalizations.

| Model | Glow 1x1 | SNF 1x1 | SNF 5x5 |
|---|---|---|---|
| $-\log p_{\mathbf{X}}(\mathbf{x})$ | $678.3 \pm 2.0$ | $678.3 \pm 9.7$ | $669.9 \pm 2.8$ |

### A.6. Novelty Comparison

We provide Table 6 (adopted from (Behrmann et al., 2019)), to facilitate comparison of the self normalizing framework with existing normalizing flow methods.

*Table 6.* High level comparison of Self Normalizing Flows (SNF) with existing normalizing flow frameworks.

| Method | NICE/ i-RevNet | Real-NVP | Glow | FFJORD | i-ResNet | SNF |
|---|---|---|---|---|---|---|
| Free-form | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Analytic Forward | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| Analytic Inverse | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Non-volume Preserving | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Exact Likelihood | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Unbiased Stochastic Log-Det Estimator | N/A | N/A | N/A | ✓ | ✗ | ✗ |
| Unconstrained Lipschitz Constant | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |

### A.7. Acknowledgements

We would like to thank the creators of Weight & Biases (Biewald, 2020) and PyTorch (Paszke et al., 2019). Without these tools our work would not have been possible. We thank the Bosch Center for Artificial Intelligence for funding, and Anna Khoreva and Karen Ullrich for guidance. Finally, we thank the reviewers for their proposed extensions and constructive comments.