Invertible Convolutional Networks

Marc Finzi [*1] **Pavel Izmailov** [*1] **Wesley Maddox** [*1] **Polina Kirichenko** [*1] **Andrew Gordon Wilson** [1]

Abstract

Recently, substantial work has been invested into developing neural network architectures that are invertible. In this work we relax some of the restrictions of past work and show how convolutional layers can be used directly. We show that standard convolutional layers of a neural network can be inverted exactly using the Fourier transform and we provide a method for computing tractable log determinants of this transformation using only matrix routines over real numbers. With the addition of bijective activation functions and downsampling, ordinary (non-residual) convolutional networks can be made invertible with minimal degradation of performance for the original classification task. We then apply these techniques to define a simple normalizing flow.

1. Introduction

Convolutional neural networks (ConvNets)(LeCun et al., 1990) have proved incredibly effective for problems with translational structure and spatial locality. It has recently been shown that networks can be designed that are invertible and yet can still be trained to be successful at discriminative tasks. These networks hold promise for more direct feature visualization, low memory training, and designing expressive normalizing flows.

We propose new building blocks for invertible neural networks that maintain the strong inductive biases that ConvNets possess for discriminative tasks. Specifically, we show how non-residual ConvNets can be converted into crude normalizing flow models with minimal modification. Because the procedure can be applied to highly performant existing CNN architectures, we can hope to design invertible neural networks useful for both generative and discriminative modelling. We show that our changes, required for invertibility, do not substantially impact discriminative capability (classification accuracy). Furthermore we show that with these adjustments, the direct convolutional network shows promise for normalizing flows and generative modelling.

[*]Equal contribution [1]Cornell University. Correspondence to: Andrew Gordon Wilson <andrew@cornell.edu>.

First workshop on *Invertible Neural Networks and Normalizing Flows* (ICML 2019), Long Beach, CA, USA

2. Related work

Normalizing Flows Dinh et al. (2016) proposed RealNVP, which uses a restricted set of non-volume preserving, but invertible transformations; an additional innovation was the usage of a multi-scale architecture to allow separating out information from different layers of the flow, incorporating checkerboard downsampling layers. Kingma & Dhariwal (2018) proposed Glow, which generalizes the channel permutation in RealNVP with 1×1 convolutions and scale up the flow to larger problems.

Autoregressive Flows Autoregressive flows (Kingma et al., 2016) utilize auto-regressive neural networks and a stacked composition of Gaussian sampling operations to produce invertible transformations for variational inference (and autoencoders), while Papamakarios et al. (2017) generalize the idea of using autoregressive networks to maximum likelihood density estimation.

Invertible Neural Networks Complementary to normalizing flows, there has been some work done designing more flexible invertible networks. Gomez et al. (2017) proposed reversible residual networks (RevNet) to limit the memory overhead of backpropagation, while Jacobsen et al. (2018) built modifications to allow an explicit form of the inverse, also studying the ill-conditioning of the local inverse. In FFJORD, Grathwohl et al. (2018) use a continuous version of neural network dynamics to create reversible generative models and flexible transformations, but at a high computational cost. Behrmann et al. (2018) explore a discretization of FFJORD requiring only a fixed computational budget by utilizing fixed point iteration to compute inverses and a Taylor expansion to compute the log determinant for spectrally constrained residual networks.

After this work was first submitted, an independent work (Hoogeboom et al., 2019) on using 3x3 convolutions in Glow-style normalizing flows was brought to our attention. Hoogeboom et al. (2019) use a similar approach to inverting the 3x3 convolutional layers, but they do not use it to construct invertible CNNs, which is the main focus of our

paper.

3. Method

3.1. Warmup: Inverses and Logdets for Fully Connected Layers

As a motivating example, consider a standard fully connected neural network defined recursively

$$h^{(i+1)} = f^{(i)}(h^{(i)}) = \sigma(W^{(i)}h^{(i)} + b^{(i)})$$

with $h^{(1)} = x \in \mathbb{R}^d$ and the network outputs $y = h^{(L)} = f(x)$. If the weight matrices are square and randomly initialized with uniform or normally distributed entries (or with scaling as in the Xavier and Glorot initializations), it is well known that these matrices will be invertible with probability 1. The determinant of each matrix is a polynomial of its entries, and so zero set of a polynomial has measure 0 (Caron & Traynor, 2005). This implies that, at least at initialization, fully connected linear layers can be inverted directly. This fact has been made use of in Kingma & Dhariwal (2018) in the form of 1×1 convolutions.

Log determinants of this network, useful for normalizing flows, can be decomposed into the sum over the individual log determinants of the Jacobian for each layer. These log determinants can be split into the sum of the log determinants of the weight matrices, computable in $O(d^3)$ time, and the activations computable in $O(d)$ time:

$$\begin{aligned}\log|\text{detDf}| &= \sum_{i=1}^{L} \log|\text{detDf}_i| \\ &= \sum_{i=1}^{L} \left(\log|\text{det}W^{(i)}| + \sum_{j=1}^{d} \log \sigma'(a_j^{(i)})\right).\end{aligned} \tag{1}$$

3.2. Analytic Inverse of Convolutional Layers

Convolutional (Conv) layers in neural networks usually refer to multi-channel discrete cross correlation, a linear operation. We show that if the convolution has the same number of input an output channels, it can be inverted directly without relying on complex and input dependent iterative methods like GMRES.

The main advance is to directly use the convolution theorem, where individual channel cross correlations can be expressed as element-wise multiplication in the Fourier domain. Suppose we have an input x of size $c \times h \times w$ and a convolutional weight matrix of size $c \times c \times r \times r$ where the r is the receptive field size, typically 3, and we take indices $i, j, k = 1, 2, ..., c$ to index along the channel dimension. The i^{th} channel of the convolutional layer output can be written as

$$\text{Conv}_W(x)_i = \sum_j W_{ij} \star x_j = \sum_j \mathcal{F}^{-1}\left(\mathcal{F}(W_{ij})^* \circ \mathcal{F}(x_j)\right),$$

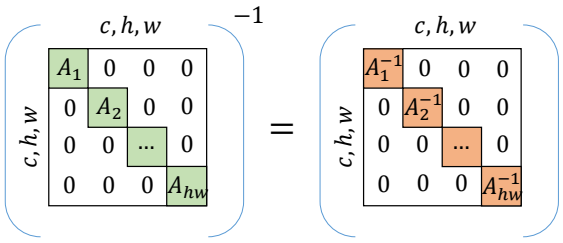

Figure 1. Inversion of convolution operation in the Fourier domain. The overall $chw \times chw$ linear operation performed by $(X_j)_{j=1}^c \to \left(\sum_j \left((\mathcal{F}(W)^*)_{ij} \circ X_j\right)\right)_{i=1}^c$ is block diagonal, with h, w many blocks of size $c \times c$. To invert this block diagonal matrix, we simply need to invert each of the blocks.

where $\star$ is the (circular) cross correlation operator, $\mathcal{F}$ denotes a Fourier transformation over the two spatial dimensions, $*$ is complex conjugate, $\circ$ is an elementwise product over the elements in an image, and W is interpreted as being padded with zeros so as to match the spatial dimension of x. This representation has been made use of in a number of papers (Vasilache et al., 2014; Rippel et al., 2015) for speeding up the operation when the filter size, k, is large. In the Fourier representation, the operation is diagonal over the spatial axes and altogether expressed as a matrix is a block diagonal matrix.

This means that in Fourier domain, the inverse can be expressed simply as the inverse of the channel blocks,

$$\text{Conv}_W^{-1}(y)_k = \sum_i \mathcal{F}^{-1}\left(\left(\mathcal{F}(W)^*\right)_{ki}^{-1} \circ \mathcal{F}(y_i)\right),$$

when circular padding (rather than zero padding) is used. Here $\left(\mathcal{F}(W)^*\right)^{-1}$ denotes the $h \times w$ inverses of the $c \times c \times h \times w$ tensor along the channel dimensions. This relationship is shown schematically in Figure 1.

Exploiting this representation reduces an (intractable) naive $O(h^3w^3c^3)$ computation time to $O(c^2hw\log(hw) + c^3hw))$. Another benefit is that only a single inversion needs to be performed and then many images can be sampled using Fourier transforms and standard Conv layers.

3.3. Differentiable Log Dets for Convolutional Layers

The existence of the inverse and its conditioning depend on the singular values of the convolution operation. Making use of the work by Sedghi et al. (2018), we can compute the singular values, and log determinants of convolutional layers differentiably in an efficient manner using the Fourier domain. The $c \times h \times w$ singular values are just the concatenation of the c singular values of $\mathcal{F}(W)_{:,:,m,n}$ for each $m = 1, ..., h$ and $n = 1, ..., w$, and this holds exactly for convolutional layers with circular padding (Sedghi et al., 2018; Bibi et al., 2019).

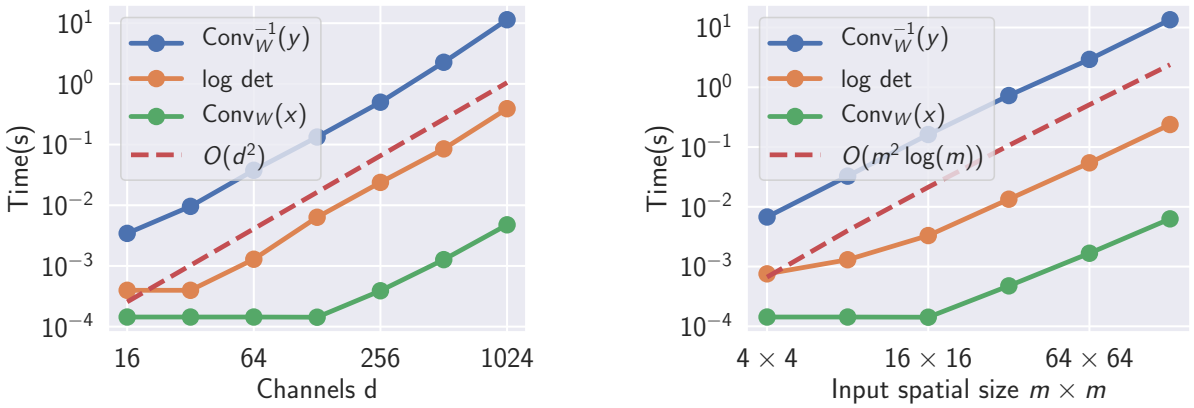

Figure 2. Runtime of Conv operations for $bs, c, h, w = 32 \times 64 \times 8 \times 8$ sized image with a 3×3 filter as the number of channels $d = c$ and the spatial size $h, w = m, m$ is varied. Log det and $\text{Conv}_W(\cdot)$ operations are accelerated on an Nvidia $1080ti$, and the $\text{Conv}_W^{-1}(\cdot)$ is computed on the CPU.

Since the discrete Fourier transform is a (linear) polynomial in the elements, the determinant of the convolutional layer is a polynomial in the entries of W and therefore its zero set has measure zero. This implies that Gaussian (or Uniform) sampled $c \times c \times r \times r$ parameter tensors will yield invertible convolutional layers with probability 1.

The log determinant decomposes into the sum over spatial locations,

$$\log|\det(\text{Conv}_W)| = \sum_{h,w} \log|\det(\mathcal{F}(W)^*_{:,:,h,w})|.$$

Since many deep learning frameworks[1] only support real valued arithmetic, we instead embed the complex valued matrices $(\mathcal{F}(W)^*_{:,:,h,w})$ into a higher dimensional real valued space, perform the computations there, and transform back. We can lift a complex valued matrix $C = A + iB \in \mathbb{C}^{n \times n}$ into the subring of real valued matrices in $\mathbb{R}^{2n \times 2n}$ with the invertible, structure preserving mapping,

$$\phi(A + iB) = I \otimes A + J \otimes B = \begin{pmatrix} A & -B \\ B & A \end{pmatrix}$$

where $\otimes$ denotes Kronecker product, I is the standard 2×2 identity and $J = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$, $J^2 = -I$. Note that the log determinant in this lifted space satisfies

$$\log|\det(C)| = \frac{1}{2}\log|\det(\phi(C))|,$$

(see e.g. Section 2.8 of Prasolov, 2007).

Defining $D_{h,w} = \phi(\mathcal{F}(W)_{:,:,h,w})$, we can use the fact that $\log|D| = \frac{1}{2}\log|DD^T|$, and then apply standard methods (e.g. a Cholesky factorization in batch mode) for computing the determinants of real valued symmetric matrices.

[1] e.g. Pytorch, MXNet, Theano. Our implementation is in PyTorch.

The log determinant becomes

$$\log|\det(\text{Conv}_W)| = \frac{1}{4}\sum_{h,w} \log|\det(D_{h,w}D_{h,w}^T)|.$$

Surprisingly we find that runtimes for computing (Conv) log determinants and inverses grow quadratically in the number of channels over the range of d values $d = 16, .., 1024$ rather than the $O(d^3)$ that we would expect asymptotically, see Figure 2. This suggests perhaps that memory allocation and the 2d FFT are the bottleneck at this scale.

4. Additional Components

4.1. Smooth bijective activation function

In order to invert the composition of convolutional layers and nonlinearities, we need to use bijective nonlinearities.[2] While the LeakyReLU nonlinearity (Maas et al., 2013) is bijective, it is not twice differentiable, so derivatives of Jacobian log determinants are not well defined. We propose a smooth variation of the LeakyReLU function, SneakyReLU, defined by

$$\sigma(x) = \left(x + \alpha(\sqrt{1 + x^2} - 1)\right)/(1 + \alpha)$$

$\alpha = \frac{1-s}{1+s}$ where s is the slope as $x \to -\infty$. A visual comparison to the LeakyReLU function is shown in Figure 5. The function has a closed form inverse

$$\sigma^{-1}(y) = (\sqrt{\alpha^2 + \alpha^2 b^2 - \alpha^4} - b)/(\alpha^2 - 1)$$

[2] Here we need bijective rather than just injective functions so that roundoff error and numerical instabilities don't perturb an intermediate activation outside of the range of it's activation function. Also for the purposes of performing latent space interpolations, and feature visualization it is useful not to have a restricted image.

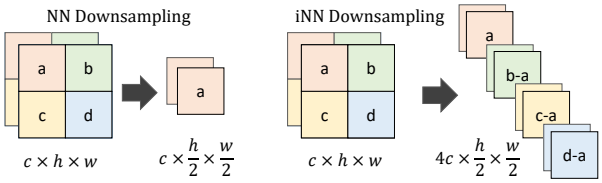

Figure 3. Invertible nearest neighbor downsampling

where $b = (1+\alpha)y + \alpha$. The log determinant also has a simple form: $\log \sigma'(x) = \log(1 + \alpha \frac{x}{\sqrt{1+x^2}}) - \log(1+\alpha)$. We set the negative slope s to 0.01. Although $\mathrm{SneakyReLU}$ closely approximates $\mathrm{LeakyReLU}$ in the positive and negative limits, we find that it still leads to substantially reduced classification performance, see Table 1. We hope to reduce this gap by learning s.

4.2. Improved invertible downsampling layers

We found that the commonly applied checkerboard invertible downsampling layer, (Dinh et al., 2016), substantially hinders classification performance, see Table 1. We propose two alternate invertible downsampling layers, iNN and $\mathrm{iAvgPool2d}$, that more closely mimic the up and downsampling layers typically used in the GAN community and for classification. In these layers, the standard output of a downsampling layer $\{\mathrm{NearestNeighbor}, \mathrm{AvgPool}\}$ passed on as one channel output and the *difference* in the upsampling of this output and the input is passed on in the remaining channels. This operation is explained in more detail in the appendix A.

4.3. Performance Degradation Path

We evaluate the effect of the proposed changes that are required to make the network invertible on the CIFAR10 dataset for image classification. For data augmentation, we use the standard set of random 2×2 translations and random flips.

Change	Accuracy	Diff
Base Convnet (Tbl 4)	94.6	-
Zero Padding $\rightarrow$ Circular Padding	94.6	0
ReLU $\rightarrow$ SneakyReLU	93.0	-1.6
MaxPool2d $\rightarrow$ Checkerboard DS	92.0	-1.0
MaxPool2d $\rightarrow$ iNN	92.5	-0.5
MaxPool2d $\rightarrow$ iAvgPool2d	92.6	-0.4

Table 1: Path to Invertibility (CIFAR10 classification performance)

5. Fully Convolutional Normalizing Flow

Making use of the methods above to compute inverses and differentiable log determinants for convolutional layers, we construct a normalizing flow mapping the data space to the normal distribution $\mathcal{X} \rightarrow N(0, I)$. In general during the training process some of the singular values may converge arbitrarily close to 0, making the matrices difficult to invert numerically. However, the training objective of normalizing flows naturally remedies this potential problem. In a normalizing flow, the objective is the negative log likelihood $\mathrm{NLL}(x) = \frac{1}{2}\|f(x)\|^2 - \log|\mathrm{detD}f| + \frac{chw}{2}\log(2\pi)$. The $-\log|\mathrm{detDf}|$ term in the loss penalizes small singular values. The test NLL of the network trained on CIFAR10 is shown in Table 2. While the generated samples of the model are of low quality, the model admits plausible interpolations in the latent space, see Figure 4.

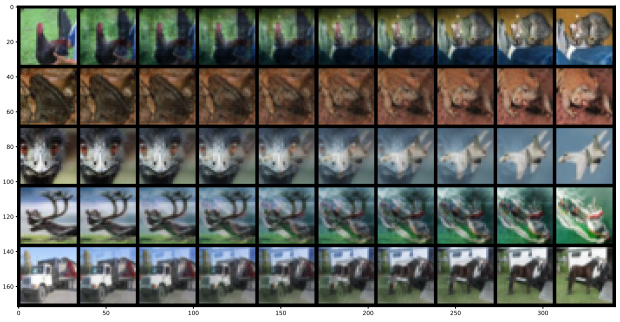

Figure 4. Latent space interpolations of test CIFAR10 images for the Convnet normalizing flow model.

Method	BPD
MADE (Germain et al., 2015)	5.67
MAF (Papamakarios et al., 2017)	4.31
RealNVP (Dinh et al., 2014)	3.49
Glow (Kingma & Dhariwal, 2018)	3.35
FFJORD (Grathwohl et al., 2018)	3.40
i-ResNet (Behrmann et al., 2018)	3.45
i-ConvNet	4.61

Table 2: Bits per dimension on CIFAR10.

6. Discussion and Future Work

In this preliminary work, we have introduced several building blocks for invertible convolutional networks that are less restrictive than other methods, showing promise for use in normalizing flows. We hope that these building blocks can be used to build models for joint classification and density modelling tasks. We plan on refine our normalizing flow model as well as targeting other applications of invertible conv layers. These building blocks could be incorporated into more established flow models like Glow, as well as for learning Gaussian random fields with convolutional covariance matrices, and perhaps for feature visualization.

References

Behrmann, J., Duvenaud, D., and Jacobsen, J.-H. Invertible residual networks. *arXiv preprint arXiv:1811.00995*, 2018.

Bibi, A., Ghanem, B., Koltun, V., and Ranftl, R. Deep layers as stochastic solvers. In *International Conference on Learning Representations*, 2019.

Caron, R. and Traynor, T. The zero set of a polynomial. *WSMR Report*, pp. 05–02, 2005.

Dinh, L., Krueger, D., and Bengio, Y. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*, pp. 881–889, 2015.

Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pp. 2214–2224, 2017.

Grathwohl, W., Chen, R. T., Betterncourt, J., Sutskever, I., and Duvenaud, D. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.

Hoogeboom, E., Berg, R. v. d., and Welling, M. Emerging convolutions for generative normalizing flows. *arXiv preprint arXiv:1901.11137*, 2019.

Jacobsen, J.-H., Smeulders, A., and Oyallon, E. i-revnet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

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

Kingma, D. P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., and Welling, M. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pp. 4743–4751, 2016.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pp. 396–404, 1990.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, pp. 3, 2013.

Papamakarios, G., Pavlakou, T., and Murray, I. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pp. 2338–2347, 2017.

Prasolov, V. V. *Elements of homology theory*, volume 81. American Mathematical Soc., 2007.

Rippel, O., Snoek, J., and Adams, R. P. Spectral representations for convolutional neural networks. In *Advances in neural information processing systems*, pp. 2449–2457, 2015.

Sedghi, H., Gupta, V., and Long, P. M. The singular values of convolutional layers. *CoRR*, abs/1805.10408, 2018. URL http://arxiv.org/abs/1805.10408.

Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

Classification Architecture
$3 \times 32 \times 32$ RGB image
ChannelPadding($3 \rightarrow 128$)
Conv(128,128), BN, SneakyReLU
Conv(128,128), BN, SneakyReLU
Conv(128,128), BN, SneakyReLU
InvertibleDownsample(NN)
KeepChannels(128)
Conv(128,128), BN, SneakyReLU
Conv(128,128), BN, SneakyReLU
Conv(128,128), BN, SneakyReLU
InvertibleDownsample(NN)
KeepChannels(256)
Conv(256,256), BN, SneakyReLU
Conv(256,256), BN, SneakyReLU
Conv(256,256), BN, SneakyReLU
GlobalAveragePool2d
Linear(256,10)
Softmax

Table 3: Invertible CNN (4M params). The keepChannels(k) layer throws away all but the first k channels, this analogous to the multiscale elements for normalizing flows.

Flow Architecture
$3 \times 32 \times 32$ RGB image
Sigmoid Dequantization Layer
Conv(3,3), SneakyReLU
Conv(3,3), SneakyReLU
Conv(3,2), SneakyReLU
InvertibleDownsample(NN)
Conv(12,12), SneakyReLU
Conv(12,12), SneakyReLU
Conv(12,12), SneakyReLU
InvertibleDownsample(NN)
Conv(48,48), SneakyReLU
Conv(48,48), SneakyReLU
Conv(48,48), SneakyReLU
InvertibleDownsample(NN)
Conv(192,192), SneakyReLU
Conv(192,192), SneakyReLU
Conv(192,192), SneakyReLU
Conv(192,192)

Table 4: Invertible CNN (1.4M params). The sigmoid dequantization layer is identical to that used in Dinh et al. (2016) which converts the discrete $0-255$ valued distribution over intensities into continuous valued logits, so that the inverse transform is squashed by a sigmoid to lie between 0 and 1.

A. iNN and iAvgPool2d*downsampling*

In checkerboard downsampling squares of 4 spatial locations are separated into their own channels, and for the inverse this means that the corresponding channels are highly correlated because of the similar values. We hypothesize that this correlation makes it more difficult to synthesize plausible activation maps in the reverse direction. To get around this difficulty, we propose to separate the channels of a given downsampling method (such as nearest neighbors or average pooling) from the difference between the upsampled downsampled output. For nearest neighbors, the operation in the downsampling direction is shown in Figure 3, and the log determinant is 0. For iAvgPool2d, the output of the first component is replaced by the average over the 2×2 spatial neighborhood, and $\log\det(\text{iAvgPool2d}) = -c \times h \times w \log(\sqrt{2})$.

B. Normalizing Flows Details

Unlike in the classification architecture, we do not apply the injective zero padding of channels so as not to change the dimension of the data. We also found it useful to remove batch normalization, as we sometimes found that the learned channel wise scaling made inversion of the entire network impossible even when the individual convolutional layers are invertible.

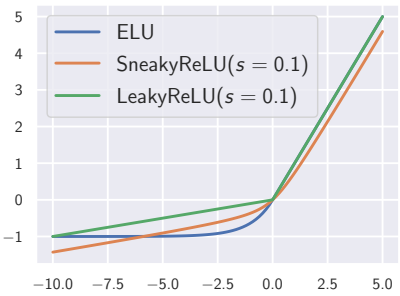

Figure 5. Comparison of proposed activation function SneakyReLU (Smooth Leaky ReLU) to the twice differentiable but not bijective ELU and the bijective but not twice differentiable LeakyReLU. Notice that SneakyReLU and LeakyReLU have the same asymptotic behaviour as $x \rightarrow \pm\infty$.