A SurVAE

In this section we describe how the flowification procedure fits into the surVAE framework [15]. The surVAE framework can be used to compose bijective and surjective transformations such that the likelihood of the composition can be calculated exactly or bounded from below. To define a likelihood for surjections both a forward and an inverse transformation is defined, where one of the directions is necessarily stochastic.

A transformation that is stochastic in the forward direction is called a generative surjection, and the likelihood of such transformations can only be bounded from below. Flowified dimension increasing linear layers, residual connections, and coordinate repetition are examples of generative surjections. A transformation that is stochastic in the inverse direction is called an inference surjection, where the likelihood can be calculated exactly if the layer satisfies the right inverse condition [15]. A flowified dimension reducing linear layer is an inference surjection that meets the right inverse condition.

The augmentation gap Dimension increasing operations (generative surjections) introduce an augmentation gap in the likelihood calculations as derived by Huang et al. [13]. After this augmentation step the subsequent layers model the joint distribution $p(x, u)$ of the data x and the augmenting noise u. In this sense, the dimension increasing operations of this work are only normalizing flows to the degree to which augmented normalizing flows are normalizing flows. The way we perform convolutions (pad, unfold, apply kernel) requires a large number of dimension expanding steps (unfold) and it is inevitable to work with this (weakened) version of flows. If a convolutional layer is such that the dimensionality (channels times height times width) of its input is less than or equal than that of its output, the overall convolution operation is still in the dimension reducing regime, theoretically allowing to compute the exact likelihood. We investigated this approach by diagonalizing the convolution using the Fourier transform, which unfortunately turned out to be too expensive computationally to be used in practice. See Appendix H for details.

B Padding in dimension increasing operations

When flowifying a dimension increasing operation the choice of noise to use in the augmentation [13] plays an important role in the performance of the model. In this section we compare sampling from a uniform distribution to sampling from a normal distribution. In all cases we observe that sampling from a normal distribution is significantly better than sampling from a uniform distribution as seen in Table 3. This is likely due to the normal distribution inverse that is used as the inverse in the dimension reducing layers as well as the normal base distribution.

Table 3: Test-set bits per dimension (BPD) for MNIST and CIFAR-10 models, lower is better. Comparing noise sampled from a normal and a uniform distribution for FCONV1.

NOISE	MNIST	CIFAR-10
UNIFORM	3.41	6.24
NORMAL	3.11	4.91

Implementation details

The flowification of convolutional layers involves an unfolding step as explained in §3.2. This procedure entails the replication of all pixels values as many times as the number of patches the given pixel is contained in and the additional of orthogonal noise. Our implementation relies on the `Fold` and `Unfold` operations of PyTorch and its essence is summarised in the following piece of pseudocode,

```
#Let X be the input image
# unfolding is a diagonal embedding for each pixel p -> (p,p, ...,p).
patches = unfold(X)

# generate the orthogonal noise in two steps
### 1. Uniformly sample the direction from the  ###
### (N-1)-sphere  orthogonal to the diagonal    ###
u_dir = torch.randn(patches.shape)
u_dir -= unfold(fold(u_dir,aggr="mean")   # center per input pixel
u_dir /= unfold(fold(u_dir**2, aggr="sum")# normalize per input pixel

### 2. Sample the amplitude to make the  ###
### overall sample uniform on the N-ball ###
N = fold(unfold(torch.ones(X.shape))) # multiplicities
u_amp = torch.rand(X.shape)
u_amp = unfold(u_amp)** (1 / (N - 1)))

# the orthogonal noise is then the product
u = u_dir * u_amp
```

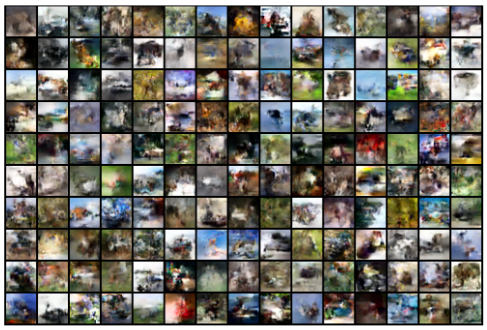
Figure 5: Pseudocode for the orthogonal noise generation in Convolutinal networks. To sample the $N-1$ dimensional orthogonal noise from a normal, the uniform sampling in line 15 needs to replaced with sampling from the χ^2 distribution with $(N-1)$-many degrees of freedom. The likelihood contribution is given by Eq. 18, which in this case is then calculated from the tensor N and the volumes of N dimensional balls.

C Flowification + NSF

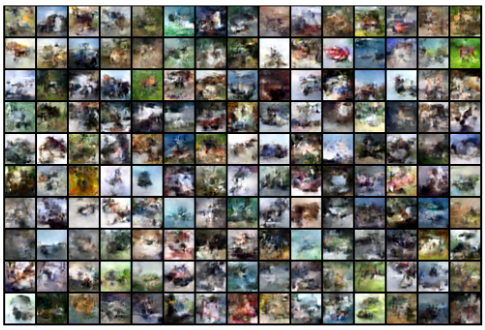
Below are samples from an architecture combining flowified standard layers and NSF layers.

(a) Flowified convnets with non-overlapping kernels combined with NSF layers on MNIST, BPD 1.35

(b) Flowified convnets with overlapping kernels combined with NSF layers on MNIST, BPD 2.70



(c) Flowified convnets with non-overlapping kernels combined with NSF layers on CIFAR-10, BPD 3.69



(d) Flowified convnets with overlapping kernels combined with NSF layers on CIFAR-10, BPD 3.93

D Complete proofs of Theorems 3 and 6

Proof of Theorem 3. The missing part of the proof is the calculation showing right invertibility. To prove this, first we let $z \in \mathbb{R}^m$ be arbitrary and then we calculate

$$\begin{aligned}
\left(\mathcal{L} \circ \mathcal{L}^{-1}\right)(z) &= (V\Sigma U) \circ \left(U^T \Sigma^{-1} V^T\right)(z-b) + b && (23)\\
&= \left(V\left(\Sigma\left(UU^T\right)\Sigma^{-1}\right)V^T\right)(z-b) + b && (24)\\
&= (z-b) + b && (25)\\
&= z. && (26)
\end{aligned}$$

□

Proof of Theorem 6. The missing part of the proof is showing that $\mathcal{L}^{-1} = L^+_{W,b}$ and that the resulting layer is left invertible. To prove $\mathcal{L}^{-1} = L^+_{W,b}$ we let $z \in \mathbb{R}^m$,

$$\mathcal{L}^{-1}(z) = U^T\Sigma^{-1}V^T(z-b) = W^+(z-b) = L^+_{W,b} \quad (27)$$

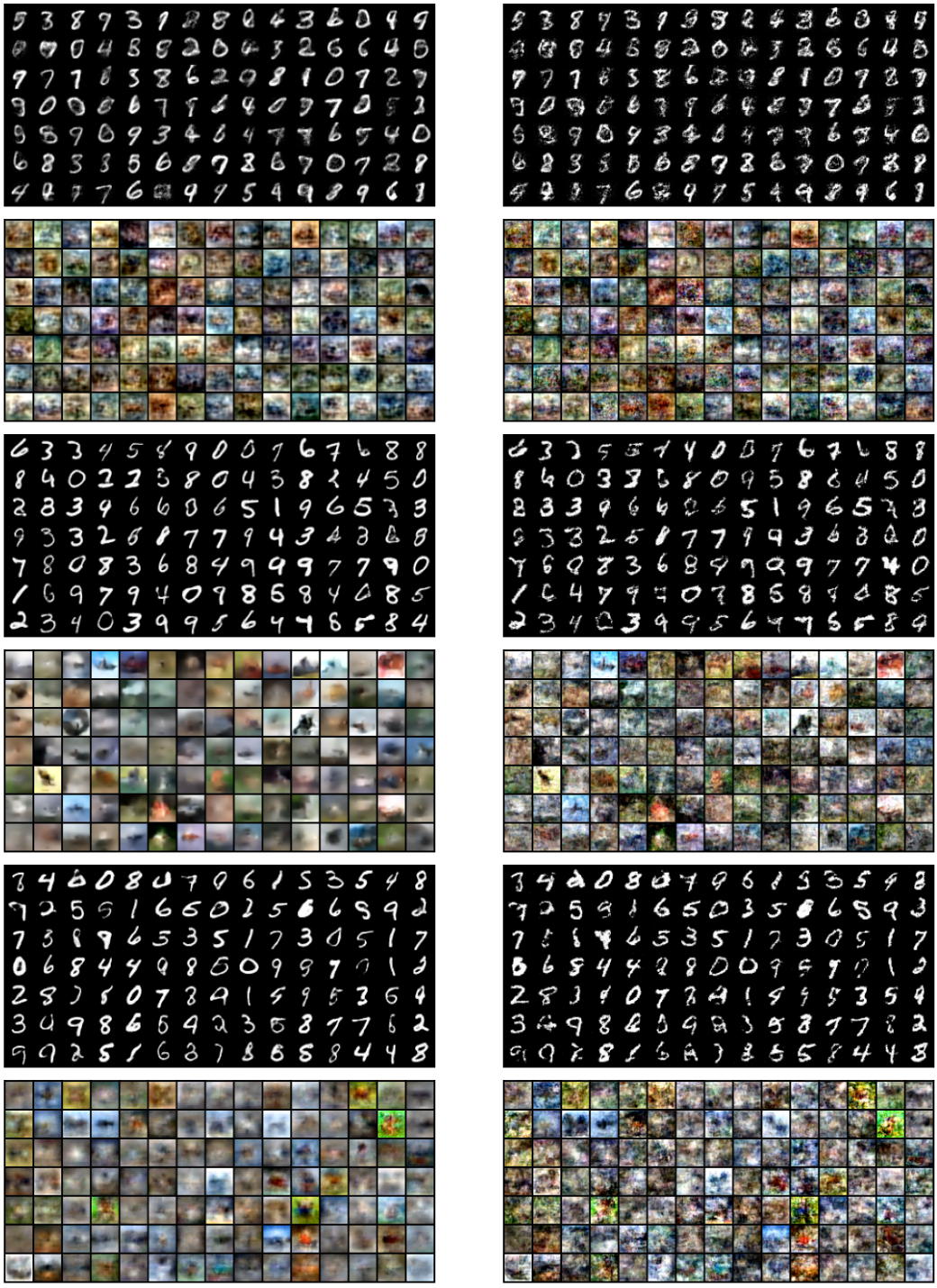
Finally, left invertibility is proven by

$$\begin{aligned}
\left(\mathcal{L}^{-1} \circ \mathcal{L}\right)(x) &= \left(U^T\Sigma^{-1}V^T\right)\left[(V\Sigma U)(x) + b - b\right] && (28)\\
&= \left(U^T\Sigma^{-1}V^T\right)\left[(V\Sigma U)(x)\right] && (29)\\
&= \left(U^T(\Sigma^{-1}(V^TV)\Sigma)U\right)(x) && (30)\\
&= x && (31)
\end{aligned}$$

□

E Samples from FMLP, FCONV1 and FCONV2

Below are samples from FMLP(top 2 rows), FCONV1(middle 2 rows) and FCONV2(bottom 2 rows) trained on MNIST and CIFAR-10 . Samples where the mean is used to invert the SVD in the inverse pass (left). Samples generated by drawing from the inverse density to invert the SVD in the inverse pass (right).



F Experimental details

Everything was implemented in PyTorch [37] and executed on a RTX 3090 GPU.

We also used weights and biases for experiment tracking [38] and the neural spline flows [39] package for the rational quadratic activation functions. Both of these tools are available under an MIT license.

Tabular data

All training hyper parameters were set to be the same as those used in the neural spline flow experiments [5]. All networks in this section are flowified MLPs. Several different architectures were tested with different design choices. Dimension preserving layers of different depth were considered with the number of layers set to $3, 5, 7, 9, 21$. Fixed depth layers were also considered with widths from $256, 128, 64, 32, 16$ and depths from $3, 5, 7$. We also tried hand designed widths with depths from $[128, 128, 128, 64, 32]$, $[128, 64, 32, 16, 8]$, $[8, 16, 32, 64, 128]$, $[32, 64, 128, 128, 128]$. In all cases the second architecture was used, except for BSDS300 where the first architecture was used. Leaky ReLU activations were used with a negative slope of 0.5.

Image data

The models on CIFAR-10 were trained for 1000 epochs with a batch size of 256 and lasted 36-48 hours. The models on MNIST were trained for 200 epochs with the same batch size and lasted approximately 8-12 hours. The learning rate in all experiments was initialized at 5×10^{-4} and annealed to 0 following a cosine schedule [40].

F.1 Architectures

```
import torch.nn as nn
import flowification as ffn

model = nn.Sequential(
  ffn.Flatten(),
  ffn.Linear(784, 512), RqSpline(),
  ffn.Linear(512, 256), RqSpline(),
  ffn.Linear(256, 128), RqSpline(),
  ffn.Linear(128, 64), RqSpline(),
  ffn.Linear(64, 32), RqSpline(),
  ffn.Linear(32, 8)
    )

```

Figure 8: FMLP Architecture for MNIST.

```
import torch.nn as nn
import flowification as ffn

model = nn.Sequential(
  ffn.Flatten(),
  ffn.Linear(3072, 1024), RqSpline(),
  ffn.Linear(1024, 512), RqSpline(),
  ffn.Linear(512, 512), RqSpline(),
  ffn.Linear(512, 256), RqSpline(),
  ffn.Linear(256, 128)
    )

```

Figure 9: FMLP Architecture for CIFAR-10.

```
import torch.nn as nn
import flowification as ffn

model = nn.Sequential(
  ffn.Conv2d(3, 27, kernel_size=3, stride=2), RqSpline(),  # 16
  ffn.Conv2d(27, 64, kernel_size=3, stride=2), RqSpline(),  # 8
  ffn.Conv2d(64, 100, kernel_size=3, stride=2), RqSpline(),  # 4
  ffn.Conv2d(100, 112, kernel_size=3, stride=2), RqSpline(),  # 2
  ffn.Conv2d(112, 128, kernel_size=2, stride=2), RqSpline(),  # 1
  ffn.Flatten(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline(),
  ffn.Linear(128, 128), RqSpline()
    )

```

Figure 10: FCONV1 Architecture for CIFAR-10.

```
import torch.nn as nn
import flowification as ffn

model = nn.Sequential(
  ffn.Conv2d(1, 16, kernel_size=3, stride=2), RqSpline(),  # 14
  ffn.Conv2d(16, 24, kernel_size=2, stride=2), RqSpline(),  # 7
  ffn.Conv2d(24, 32, kernel_size=3, stride=2), RqSpline(),  # 3
  ffn.Conv2d(32, 48, kernel_size=2, stride=1), RqSpline(),  # 2
  ffn.Conv2d(48, 64, kernel_size=2, stride=1), RqSpline(),  # 1
  ffn.Flatten(),
  ffn.Linear(64, 64), RqSpline(),
  ffn.Linear(64, 64), RqSpline(),
  ffn.Linear(64, 64), RqSpline(),
  ffn.Linear(64, 64), RqSpline(),
  ffn.Linear(64, 64), RqSpline(),
  ffn.Linear(64, 64), RqSpline(),
  ffn.Linear(64, 32), RqSpline(),
  ffn.Linear(32, 32), RqSpline(),
  ffn.Linear(32, 32), RqSpline(),
  ffn.Linear(32, 32), RqSpline(),
  ffn.Linear(32, 32), RqSpline(),
  ffn.Linear(32, 32), RqSpline(),
  ffn.Linear(32, 8), RqSpline()
    )

```

Figure 11: FCONV1 Architecture for MNIST.

G Parametrizing rotations via the Lie algebra $\mathfrak{so}(d)$

The Lie algebra $\mathfrak{so}(d)$ of the rotation group $SO(d)$ is the space of $d \times d$ skew-symmetric matrices

$$\mathfrak{so}(d) = \{X \in M_d(\mathbb{R}) | X = -X^T\}$$

Moreover, since $SO(d)$ is connected and compact, the (matrix-)exponential map $\mathfrak{so}(d) \to SO(d)$ is surjective [41, Corollary 11.10], i.e. every rotation $U \in SO(d)$ can be written as the exponential of some skew-symmetric $X \in \mathfrak{so}(d)$.

Motivated by this result, we propose to parametrize the Lie algebra $\mathfrak{so}(d)$, and use the exponential map to obtain rotations. Since the exponential map is differentiable, we can propagate gradients all the way back to the Lie algebra and do gradient based optimization there.

Note that is significantly eases the optimization process. The reason for this is that the Lie group of unitary transformations has non-trivial geometry and doing gradient descent is cumbersome on such spaces. On the other hand, the Lie algebra is a vector space, where gradient descent can shine.

Since PyTorch [37] has built-in support for the matrix exponential, random orthogonal transformations can be generated with just a few lines of Python code:

```
d  = 5                          # size of matrix
x  = torch.randn(d,d)           # random (d,d)  matrix
S  = x - x.T()                  # S is skew-symmetric
U  = torch.matrix_exp(S)        # U  is a rotation
```

Figure 12: Generating a random rotation in PyTorch

Comparison with Householder transformations The matrix exponential has the advantage of generating the whole rotation group, while many Householder transformations need to be composed to have the same level of expressivity. This, of course, comes with an increased ($\mathcal{O}(d^2)$) computational cost. We did not explore Householder transformations, but we expect that for linear layers with large input or output dimensionality, the matrix exponential will simply become too expensive and it will become necessary to rely on Householder transformations.

H Flowifying convolutions through the FFT

An alternative way of flowifing convolutional layers is through the convolution theorem which states that the Fourier-transformation diagonalizes convolutions. In this section we sketch the idea and discuss the issues with it.

Fig.13 displays the Pytorch code for performing convolutions through the convolution theorem. Line 12 of the code shows that in the Fourier basis, we can perform convolution as follows; at each frequency we form the vector of dimension (# of in_channels) and apply a weight matrix of size (# of in_channels) $\times$ (# of out_channels) to it. If this weight matrix is parametrized by the SVD, we can guarantee its invertibility. Since the FFT is also invertible, this construction should be a flowification of the convolutional layer if we can also solve the following two issues:

If the weight matrix is parametrized in frequency domain,

(1) it's inverse transform might not be real.

(2) it's inverse transform might not be of the appropriate size (i.e. nonzero outside of the first (kernel_size)-many pixels).

The first of these is easy to solve with the symmetric parametrization induced by

$$k \text{ is real} \Leftrightarrow FFT(k)_i^* = FFT(k)_{N-i}$$

where N is the length of the signal. To deal with the second condition, a auxiliary loss term is introduced that minimizes the terms that should be 0 . Although the idea of FFT seems promising, this approach turns out be slower (probably due to the FFT), worse performing and more cumbersome than the flowificaton presented in §3.2. Fig. 14 shows some samples from a Fourier-flowified convolutional network trained on MNIST.

```
        in_channels, out_channels = 4, 3
        signal_length, kernel_size = 32, 5
        batch_size = 24

        K = torch.randn(out_channels, in_channels, kernel_size)
        x = torch.randn(batch_size, in_channels, signal_length)
        z1 = torch.nn.functional.conv1d(input=x, weight=K)

        x_ = torch.fft.fft(x, norm='ortho')
        # pad kernel to match the length of x
        K_ = torch.fft.fft(K.flip(-1), n=x.size()[-1], norm='ortho')
        z_ = torch.einsum('bix,jix->bjx', (x_, K_))
        z2 = torch.fft.ifft(z_, norm='forward')
        z2 = z2[:, :, (kernel_size-1):]

        print((z1-z2).abs().max().item())
        # tensor(2.8610e-06)

```

Figure 13: 1D convolutions in PyTorch

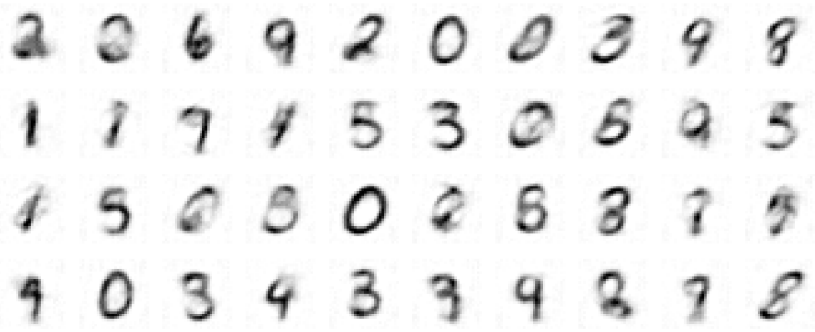


Figure 14: Samples from and Fourier-flowified convolutional network

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] **See 4 6**
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] **See 3 2**
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] **The script used to run the experiments has been included and the instructions can be found in the code.**
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] **This is not typically done on the benchmarks we considered.**
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] **See F.**
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] **See F.**
   (b) Did you mention the license of the assets? [Yes] **See F.**
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] **The tools we develop are contained in the paper, the code is included but is not a new library that should be widely adopted.**
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]