ELF: Exact-Lipschitz Based Universal Density Approximator Flow

Achintya Gopal
Bloomberg Quant Research
achintyagopal@gmail.com

Wednesday 26th May, 2021

ABSTRACT

Normalizing flows have grown more popular over the last few years; however, they continue to be computationally expensive, making them difficult to be accepted into the broader machine learning community. In this paper, we introduce a simple one-dimensional one-layer network that has closed form Lipschitz constants; using this, we introduce a new Exact-Lipschitz Flow (ELF) that combines the ease of sampling from residual flows with the strong performance of autoregressive flows. Further, we show that ELF is provably a universal density approximator, more computationally and parameter efficient compared to a multitude of other flows, and achieves state-of-the-art performance on multiple large-scale datasets.

1 Introduction

Normalizing flows have become more popular within the last few years; however, they continue to have limitations compared to other generative models, more specifically that they are computationally expensive in terms of memory and time.

Early implementations of Normalizing Flows were coupling layers (Dinh et al., 2014, 2017; Kingma and Dhariwal, 2018) and autoregressive flows (Papamakarios et al., 2017; Kingma et al., 2016). These have easy to compute log-likelihoods; however, coupling layers tend to need quite a few parameters to achieve strong performance and autoregressive flows are extremely expensive to sample from. The newer technique of residual flows (Chen et al., 2019) allows for models that are built on standard components and have inductive biases that favor simpler functions (Gopal, 2020); however, these have the problem of being expensive in terms of time for computing log-likelihoods and training, as well as require quite a few layers for strong performance.

Since the introduction of these models, there have been many developments that have lead to improvement in parameter efficiency such as FFJORD (Grathwohl et al., 2019), a continuous normalizing flow, that has a dynamic number of layers. However, this too can have computational problems as having a few dynamics layers can lead to hundreds of implicit layers.

Among the flows introduced, the ones with provable universal approximation capability are Affine Coupling Layers (Dinh et al., 2014, 2017; Teshima et al., 2020), Neural Autoregressive Flows (NAF, Huang et al. (2018)), Block NAFs (BNAF, Cao et al. (2019)), Sum-of-Squares Polynomial Flow (Jaini et al., 2019), and Convex Potential Flows (CP-Flow, Huang et al. (2021)). Though these have been shown to be universal approximators, they do not necessarily translate into faster, more efficient training, and some of the flows listed require the expensive sampling routine of autoregressive flows.

In this paper, we introduce a new universal density approximator flow, Exact-Lipschitz Flows (ELF), based on residual flows and autoregressive flows, retaining the strong performance and closed-form log-likelihoods.
of autoregressive flows with the efficiency of sampling from residual flows. We achieve this by introducing a simple class of one-dimensional one-layer networks with closed form Lipschitz constants from which we generalize to higher dimensions using hypernetworks (Ha et al., 2016). Further, we show our flow achieves state of the art performance on many tabular and image datasets.

2 Related Work

2.1 Flows

Rezende and Mohamed (2015) introduced planar flows and radial flows to use in the context of variational inference. Sylvester flows created by van den Berg et al. (2018) enhanced planar flows by increasing the capacity of each individual transform. Both planar flows and Sylvester flows can be seen as special cases of residual flows. The capacity of these models were studied by Kong and Chaudhuri (2020).

Germain et al. (2015) introduced large autoregressive models which were then used in flows by Inverse Autoregressive Flows (IAF, Kingma et al. (2016)) and Masked Autoregressive flows (MAF, Papamakarios et al. (2017)). Larger autoregressive flows were created by Neural Autoregressive Flows (NAF, Huang et al. (2018)) and Block NAFs (BNAF, Cao et al. (2019)), both of which have better log-likelihoods but cannot be sampled from efficiently.

Early state of the art performance on images with flows was achieved by coupling layers such as NICE (Dinh et al., 2014), RealNVP (Dinh et al., 2017), Glow (Kingma and Dhariwal, 2018) and Flow++ (Ho et al., 2019). Affine coupling layers have been shown to be universal density approximators in the case of a sufficient number of layers (Teshima et al., 2020).

The likelihood performance of images was further improved by variational dequantization (Ho et al., 2019). Additional improvements were achieved by masked convolutions (Ma et al., 2019) and adding additional dimensions to the input (Huang et al., 2020; Chen et al., 2020).

Whereas autoregressive methods and coupling layers have block structures in their Jacobian, Residual Flows (Chen et al., 2019) have a dense Jacobian; FFJORD (Grathwohl et al., 2019) is a continuous normalizing flow based on Neural ODEs (Chen et al., 2018) and can be viewed as a continuous version of residual flows. Zhuang et al. (2021) improved the training of FFJORD achieving state of the art performance among uniformly dequantized flows.

2.2 Lipschitz Functions

To the best of our knowledge, the first upper-bound on the Lipschitz constant of a neural network was described by Szegedy et al. (2014) as the product of the spectral norms of linear layers. Regularizing for 1-Lipschitz functions has been used in the context of GANs (Arjovsky et al., 2017; Gulrajani et al., 2017); 1-Lipschitz was further enforced in GANs by using the power iteration method to approximate the Lipschitz constant of individual layers (Miyato et al., 2018). Residual Flows (Chen et al., 2019) also used the power iteration method to ensure 1-Lipschitz in its residual blocks.

In general, computing the Lipschitz constant even for two layer neural networks is NP-hard (Virmaux and Scaman, 2018). However, this theorem is for any arbitrary network, meaning any \( n \) dimensional input, any \( m \) dimensional output, and any non-linear activation. In this work, we focus on a specific subset of networks in which the Lipschitz constant can be computed efficiently. Anil et al. (2019) introduced a new activation function and methods to have 1-Lipschitz neural networks of arbitrary depth. Further approaches to estimate the Lipschitz constant have used semidefinite programs (Fazlyab et al., 2019).

3 Background

Suppose that we wish to formulate a joint distribution on an \( n \)-dimensional real vector \( x \). A flow-based approach treats \( x \) as the result of a transformation \( f^{-1} \) applied to an underlying vector \( z \) sampled from a base distribution \( p_z(z) \). The generative process for flows is defined as:

\[
\begin{align*}
z & \sim p_z(z) \\
x & = f^{-1}(z)
\end{align*}
\]

where \( p_z \) is often a Normal distribution and \( f \) is an invertible function. Using change of variables, the log likelihood of \( x \) is

\[
\log p_x(x) = \log p_z \left( f(x) \right) + \log \left| \det \left( \frac{\partial f(x)}{\partial x} \right) \right|
\]
Algorithm 1 Inverse of Residual Flow via Fixed Point Iteration

Input: data \( y \), residual block \( g \), number of iterations \( n \)

Initialize \( x_0 = y \).

for \( i = 1 \) to \( n \) do

\[ x_i = y - g(x_{i-1}) \]

end for

To train flows (i.e., maximize the log likelihood of data points), we need to be able to compute the logarithm of the absolute value of the determinant of the Jacobian of \( f \), also called the log-determinant. To construct large normalizing flows, we can compose smaller ones as this is still invertible and the log-determinant of this composition is the sum of the individual log-determinants.

Due to the required mathematical property of invertibility, multiple transformations can be composed, and the composition is guaranteed to be invertible. Thus, in theory, a potentially complex transformation can be built up from a series a smaller, simpler transformations with tractable log-determinants.

Constructing a Normalizing Flow model in this way provides two obvious applications: drawing samples using the generative process and evaluating the probability density of the modeled distribution by computing \( p_x(x) \). These require evaluating the inverse transformation \( f \), the log-determinant, and the density \( p_z(z) \).

In practice, if either \( f \) or \( f^{-1} \) turns out to be inefficient, then one or the other of these two applications can become intractable. For evaluating the density, in particular, computing the log-determinant can be an additional trouble spot. A determinant can be computed in \( O(n^3) \) time for an arbitrary \( n \)-dimensional data space. However, in many applications of flows, such as images, \( n \) is large, and a \( O(n^3) \) cost per evaluation is simply too high to be useful. Therefore, in flow-based modeling, there are recurring themes of imposing constraints on the model that guarantee invertible transformations and log-determinants that can be computed efficiently.

3.1 Autoregressive Flows

For a multivariate distribution, the probability density of a data point can be computed using the chain rule:

\[
p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i | x_{<i})
\]  

(3.1)

By using a conditional univariate normalizing flow \( f_\theta(x_i | x_{<i}) \) such as an affine transformation for each univariate density, we get an autoregressive flow (Papamakarios et al., 2017). Given that its Jacobian is triangular, the determinant is easy to compute as it is the product of the diagonal of the Jacobian. These models have a tradeoff where the log-likelihood is parallelizable but the sampling process is sequential, or vice versa depending on parametrization (Kingma et al., 2016). In this paper, due to the fact the Jacobian is triangular, we refer to \( f_\theta \) as a triangular function. Given a naive implementation of sampling from autoregressive models, the number of times an autoregressive function would need to be called to sample would be \( d \) times.

3.2 Residual Flows

A residual flow (Chen et al., 2019) is a residual network \( f(x) = x + g(x) \) where the Lipschitz constant (Definition A.1) of \( g \) is strictly less than one. This constraint on the Lipschitz constant ensures invertibility and is enforced using power iteration method to spectral normalize \( g \); the transform is invertible using Banach’s fixed point algorithm (Algorithm 1) where the convergence rate is exponential in the number of iterations and is faster for smaller Lipschitz constants (Behrmann et al., 2019):

\[
\|x - x_n\|_2 \leq \frac{\text{Lip}(g)^n}{1 - \text{Lip}(g)} \|x_1 - x_0\|_2
\]  

(3.2)

Evaluating the density given a residual flow is expensive as the log-determinant is computed by estimating the Taylor series:

\[
\ln|J_f(x)| = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\text{tr}(J_g^k)}{k}
\]  

(3.3)

where the Skilling-Hutchinson estimator (Skilling, 1989) is used to estimate the trace and the Russian Roulette estimator (Kahn, 1955) is used to estimate the infinite series.
Figure 1: Example of randomly initialized one layer networks with hidden size $H = 5$. Through the use of a quadratic piecewise activation function, computing the Lipschitz constant (the maximum absolute value of the gradient) requires simply evaluating the gradient at the ends of the pieces.

4 ELF (Exact-Lipschitz Flows)

One of the bottlenecks in the capacity of residual flows is, as shown by Behrmann et al. (2019), that a single flow can transform the log-determinant up to:

$$d \ln \left(1 - \text{Lip}(g)\right) \leq \ln|\det J_f(x)| \leq d \ln \left(1 + \text{Lip}(g)\right)$$

where $d$ is the dimensionality of the data, $g$ is the residual connection and $f = I + g$. Both the number of layers and Lipschitz constant of $g$ affect the expansion and contraction bounds of these flows; to make matters worse, since the spectral normalization of $g$ uses an upper bound of the Lipschitz constant that has been shown to be loose (Virmaux and Scaman, 2018), the number of layers required to achieve a target expansion or contraction increases. The main contribution of our residual flow is that the Lipschitz constant can be calculated efficiently and exactly.

4.1 Universal 1-D Lipschitz Function

We start with the one dimensional case and generalize to higher dimensions in Section 4.3. In Figure 1a, we show an example of a randomly initialized one layer network with ReLU, both the function and its first derivative. The key observation here is that for a network with hidden size $H$, there are $H + 1$ different gradients and so the Lipschitz can be computed with $H + 1$ gradient evaluations.

We can generalize this method of computing the Lipschitz constant by using any activation function that is piecewise linear and quadratic. Since ELU (Clevert et al., 2015) (a piecewise function with an exponential component) has been found to give strong performance in flows (Chen et al., 2019) and we were unable to successfully train a ReLU Residual flow (Appendix D), we created Fake ELU (FELU):

$$\text{FELU}(x) = \begin{cases} 
  x & x > 0 \\
  \frac{(x+1)^2}{2} - \frac{1}{2} & -1 \leq x \leq 0 \\
  -\frac{1}{2} & x < -1
\end{cases}$$

shown in Figure 2. Figure 1b shows the same network as in Figure 1a but with FELU. Though there are not a finite number of unique gradients as in ReLU, within each piecewise component, the maximum absolute gradient can be found by simply checking the two ends of the piece.

Though not explored in this paper, other quadratic piecewise activation functions could be used where an activation function with $N$ pieces would require $((N-1) \cdot H + 1)$ gradient evaluations to compute the Lipschitz constant. For the activation to be 1-Lipschitz, it needs to have linear pieces on the two ends.
Figure 3: Diagram of our one dimensional universal density approximator flow. ActNorm denotes an elementwise affine transformation with data dependent initialization (Kingma and Dhariwal, 2018).

We show that this one-dimensional architecture (a one-layer neural network with FELU) is a universal approximator of Lipschitz functions (Lemma A.5).

### 4.2 Universal 1-D Distributions

We can use the one-dimensional architecture from Section 4.1 to create a new one-dimensional distribution based on residual flows which we call Exact-Lipchitz Flows (ELF).

In residual flows \( f = I + g \), any neural network can be used for \( g \) where, if we write \( g \) as \( g_L \circ \cdots \circ g_1 \), an upper bound on \( \text{Lip}(g) \) is

\[
\text{Lip}(g) \leq \prod_{i=1}^{L} \text{Lip}(g_i)
\]  

and so to normalize \( g \), each function \( g_i \) is normalized independently.

For ELF, we use the following for \( g(x) \):

\[
b_2 + \sum_{i=1}^{H} w_{2,i} \text{FELU}(w_{1,i} x + b_{1,i})
\]  

where \( w_{1,i}, w_{2,i}, b_{1,i}, b_2, x \in \mathbb{R} \), or, in other words, a one layer network with FELU as the activation function.

Instead of normalizing each weight matrix independently, we normalize by the exact Lipschitz constant, as was shown in Section 4.1. More specifically, we use the maximum of:

\[
\max_{i \in [1, \ldots, H]} \left| \frac{\partial g}{\partial x} \right|_{x = -\frac{b_{1,i}}{w_{1,i}}} \quad \text{and} \quad \max_{i \in [1, \ldots, H]} \left| \frac{\partial g}{\partial x} \right|_{x = \frac{1+b_{1,i}}{w_{1,i}}}
\]  

By composing this variant of residual flows with an affine layer (Figure 3), we show in Lemma A that this is a universal density approximator of one dimensional distributions. The key idea of the proof is that since the composition of ELF and an affine transformation is a universal approximator of monotonic functions (Lemma A.2), this implies a universal density approximator of one dimensional distributions (Lemma A.1).

### 4.3 Universal Approximation of Higher Dimensional Distributions

To generalize to multivariate distributions, we use an autoregressive hypernetwork and refer to this as ELF-AR.

Similar to autoregressive flows (Oord et al., 2016; Papamakarios et al., 2017; Kingma et al., 2016; Huang et al., 2018; Cao et al., 2019), we use MADE (Germain et al., 2015) to create large autoregressive networks that ensures the ordering is preserved. The output of the MADE is then the parameters of ELF. More specifically, we can write the autoregressive hypernetwork as:

\[
g(x)_t = b_2(x_{<t}) + \sum_{i=1}^{H} w_{2,i}(x_{<t}) \text{FELU} \left( w_{1,i}(x_{<t}) \cdot x_t + b_{1,i}(x_{<t}) \right)
\]  

5
where \( x, g(x) \in \mathbb{R}^d \), \( t \leq d \), \( H \) is the hidden size of ELF, and \( b_2, w_{2, t}, w_{1, t}, \) and \( b_{1, t} \) are all hypernetworks that output a scalar. In terms of implementation, to maximize parallelism, we output all \( 3H + 1 \) parameters in one call. Importantly, the hypernetwork is only over ELF, not the elementwise affine transformation in Figure 3. Once the hypernetwork outputs the parameters of ELF, each output dimension \( g(x)_t \) is then normalized by the Lipschitz constant using Equation 4.3.

In Theorem A.1, we show that this architecture is a universal density approximator where the key idea of the proof is that the construction of one-dimensional 1-Lipschitz FELU Networks is continuous, hence we can use hypernetworks to create a universal approximator of triangular 1-Lipschitz functions. Thus creating a universal approximator of triangular functions by composing the ELF-AR with an elementwise affine transformation, Teshima et al. (2020) showed that this implies a universal density approximator.

In summary, the architecture introduced in this section uses a hypernetwork to parametrize ELF (Equation 4.4). After normalizing by the Lipschitz constant, we can evaluate the function \( f \) (Equation 4.4) as well as its log-determinant in closed form:

\[
\log \left| \frac{df(x)}{dx} \right| = \log \left| \sum_{t=1}^{d} \left[ 1 + \frac{g(x)_t}{dx_t} \right] \right| (4.5)
\]

Unlike Residual Flows, the log-determinant does not require estimating an infinite series (Equation 3.3). By normalizing by the Lipschitz constant, inverting ELF (sampling) is efficient (Section 5.4.1) since, instead of having to inverting each dimension sequentially as would be done for autoregressive flows, we apply the fixed point algorithm to the full input.

5 Experiments

In this section, we test the approximation capabilities of ELF on simulated data (Section 5.1, Section 5.2), and performance on density estimation on tabular data (Section 5.3) and image datasets (Section 5.4).

5.1 Approximation Power for Lipschitz Functions

For a simple use case comparing the power of exact Lipschitz computation versus using the product of the spectral norms of the linear layers, similar to Anil et al. (2019), we check how well absolute value can be learned since, as was proven by Huster et al. (2018), norm-constrained (such as spectral norm) ReLU networks are provably unable to approximate simple functions such as absolute value. In Figure 4, we can see that even if we use multiple layers to learn absolute value, spectral norm is too inaccurate to allow for learning absolute value, even though absolute value is a 1-Lipschitz function. On the other hand, by using an exact Lipschitz computation, our architecture is able to learn absolute value.

5.2 Synthetic Data

Though ELF-AR with FELU is provably a universal approximator, this might not translate into improved performance in practice when training with gradient descent. To test this out, we compare the performance of one large ELF-AR against other universal density approximators. In Figure 5, we can see, in comparison to autoregressive models, Glow, NAFs and CP Flows, ELF-AR is able to learn a mixture of Gaussians quite well with only one transformation. However, ELF-AR’s performance in sparse regions of the data suggests room for improvement in terms of extrapolation of the density function.

In Table 1, we compare the runtime between the different flows. Though Affine Coupling is significantly faster than the other methods, it also has significantly worse performance. Further, we can see that ELF-AR, though a bit slower than the other flows in terms of sampling and inference, has stronger performance.

Performance on an additional synthetic dataset is in Appendix E. More details on the models trained is in Appendix F.1.
Table 1: Comparison of different flows on fitting to mixture of eight Gaussians (Figure 5a) where the architectures for each were chosen so that they had the same number of layers and approximately equivalent number of parameters (250K). †Implementation taken from https://github.com/CW-Huang/CP-Flow (MIT License).

| Model               | Log-Likelihood | Train Time (m) | Inference Time (s) | Sample Speed (s) |
|---------------------|----------------|----------------|--------------------|------------------|
| Affine Coupling     | -4.0           | 1.8            | 0.005              | 0.003            |
| NAF†                | -3.0           | 15.5           | 0.02               | N/A              |
| Residual Flow       | -3.5           | 10.1           | 0.06               | 0.23             |
| CP-Flow†            | **-2.8**       | 33.7           | 0.2                | 3.2              |
| ELF-AR (Ours)       | **-2.8**       | 4.2            | 0.04               | 0.27             |

5.3 Tabular Data
For our tabular experiments, we use the datasets preprocessed by Papamakarios et al. (2017)†. We compare ELF-AR against Real NVP (Dinh et al., 2017), Glow (Kingma and Dhariwal, 2018), MADE (Germain et al., 2015), MAF (Papamakarios et al., 2017), NAF (Huang et al., 2018), BNAF (Cao et al., 2019), FFJORD (Grathwohl et al., 2019), CP-Flow (Huang et al., 2021), and TAN (Oliva et al., 2018).

In Table 2, we report average log-likelihoods evaluated on held-out test sets, for the best hyperparameters found via grid search. The search was focused on weight decay and the width of the hypernetwork and of ELF. In all datasets, ELF-AR performs better than Real NVP, Glow, MADE, MAF, FFJORD, and CP-Flow, and it performs comparably or better to NAF, BNAF, and TAN. More details on architecture and training details are in Appendix F.2.

5.4 Image Data
We compare ELF-AR with variational dequantization against other state of the art flows. In Table 3, we can see that we are able to achieve state of the art performance on MNIST (LeCun and Cortes, 2010) and Imagenet 64 (Chrabaszcz et al., 2017) while having competitive performance on the other image datasets. One possible direction of improvement to close the gap between ELF-AR and other state of the art flows on CIFAR-10

---

†CC by 4.0
times is a lower bound of the true speedup. Though there are methods to use iterative methods for sampling
Section 4.2 does not have a closed form inverse (similar to NAFs). Thus, the speedup of approximately 37
evaluations than others to invert. The runtime for sampling 100 images is 70s and the runtime, if we were
however, this count is not representative of the fact that different layers in the network take more function
algorithm. For our best CIFAR-10 model, the average number of function evaluations required was 70;
Table 3: Bits-per-dim (Appendix G) estimates of standard image benchmarks (the lower the better). For vari-
ationally dequantized flows, we give the bits per dim for the uniformly dequantized versions in parenthesis.
Table 2: Log-likelihood on the test set (higher is better) for 4 datasets (Dua and Graff, 2017) from UCI
machine learning and BSDS300 (Martin et al., 2001), all preprocessed by Papamakarios et al. (2017). We
report average (±std) across 3 independently trained models.

| Model       | POWER | GAS   | HEPMASS | MINIBOONE | BSDS300 |
|-------------|-------|-------|---------|-----------|---------|
| Real NVP    | 0.17±0.01 | 8.33±14 | −18.71±0.02 | −13.55±0.49 | 153.28±1.78 |
| Glow        | 0.17±0.01 | 8.15±40 | −18.92±0.08 | −11.35±0.07 | 155.07±0.03 |
| MADE MoG    | 0.40±0.02 | 8.47±02 | −15.15±0.02 | −12.27±0.47 | 153.71±0.28 |
| MAF-affine  | 0.24±0.01 | 10.05±02 | −17.73±0.02 | −12.24±0.45 | 155.09±0.28 |
| MAF-affine MoG | 0.30±0.01 | 9.59±02 | −17.39±0.02 | −11.68±0.44 | 156.36±0.28 |
| FFJORD      | 0.46±0.01 | 8.59±12 | −14.92±0.08 | −10.43±0.04 | 157.40±0.19 |
| CP-Flow     | 0.52±0.01 | 10.36±03 | −16.93±0.08 | −10.58±0.07 | 154.99±0.08 |
| NAF         | 0.62±0.01 | 11.96±33 | −15.09±0.40 | −8.86±0.15 | 157.43±0.30 |
| BNAF        | 0.61±0.01 | 12.06±09 | −14.71±0.38 | −8.95±0.07 | 157.36±0.03 |
| TAN         | 0.60±0.01 | 12.06±02 | −13.78±0.02 | −11.01±0.48 | 159.80±0.07 |
| ELF-AR (Ours) | 0.62±0.01 | **12.55±0.03** | −14.05±0.03 | −9.60±0.04 | 157.81±0.09 |

(Krizhevsky and Hinton, 2009) and Imagenet 32 (Chrabaszcz et al., 2017) might be the use of attention
(Vaswani et al., 2017), as was done in Flow++ (Ho et al., 2019) and VFlow (Chen et al., 2020).
More details on architecture and training details are in Appendix F.

5.4.1 Sampling Efficiency

The main benefit of ELF-AR over other autoregressive flows is the ability to use a more efficient sampling
algorithm. For our best CIFAR-10 model, the average number of function evaluations required was 70;
however, this count is not representative of the fact that different layers in the network take more function
evaluations than others to invert. The runtime for sampling 100 images is 70s and the runtime, if we were
to run the hypernetwork 3072 times (the dimensionality of CIFAR-10), would be 2600s. However, 3072
function evaluations is still not fully representative of the sampling time since the function introduced in
Section 4.2 does not have a closed form inverse (similar to NAFs). Thus, the speedup of approximately 37
times is a lower bound of the true speedup. Though there are methods to use iterative methods for sampling
from an autoregressive model (Song et al., 2021; Wiggers and Hoogeboom, 2020), these methods do not
Table 4: Comparison of efficiency among uniform dequantized discrete flow-models specifically mentioning promoting fewer parameters. †Taken from Huang et al. (2021). ‡Obtained from official open source code.

|                              | MNIST     | CIFAR-10  |
|------------------------------|-----------|-----------|
|                              | Bits/dim | Param. Count | Bits/dim | Param. Count |
| Glow (Kingma and Dhariwal, 2018) | 1.06     | N/A       | 3.35    | 44.0M†         |
| RQ-NSF (Durkan et al., 2019)  | —        | N/A       | 3.38    | 11.8M†          |
| Residual Flow (Chen et al., 2019) | 0.97  | 16.6M‡   | 3.28    | 25.2M‡         |
| CP-Flow (Huang et al., 2021)  | 1.02     | 2.9M†    | 3.40    | 1.9M†          |
| ELF-AR (Ours)                | 0.92     | 1.9M     | 3.33    | 1.9M          |

easily lend themselves to models where the univariate function (e.g. ELF) does not have a closed-form inverse.

5.4.2 Parameter Efficiency
For evaluating the efficiency of ELF-AR, we measured the performance on MNIST and CIFAR-10 with approximately 2 million parameters without variational dequantization, with only 24 hours of training on a single Tesla V100-SXM2-32GB GPU. In Table 4, we have the performance of ELF-AR alongside other models with their corresponding parameter counts. With fewer parameters than any other model in the table, we were able to outperform all other flows except Residual Flows on CIFAR-10 and achieve state of the art among discrete flow models on MNIST with both a large margin in bits-per-dim and number of parameters. Results for variationally dequantized models can be found in Appendix C.

6 Limitations
Though we were able to achieve strong performance, the method does seem to find bad local minima when the data is highly multimodal. Specifically in the mixture of Eight Gaussians, when the clusters are smaller, there is a higher tendency to find bad solutions. We leave it to future work to find ways to improve this component which might help improve the overall performance in larger scale datasets. Another limitation is that, due to the use of hybernetworks, modeling higher dimensional tabular data can lead to large parameter counts. Further, from Figure 5b, it can be seen that ELF-AR does not seem to have the same inductive bias towards simple solutions as CP-Flow (Huang et al., 2021) and Residual Flows (Chen et al., 2019; Gopal, 2020).

7 Conclusion and Future Work
In this work, we have introduced a new normalizing flow that allows for more efficient training, evaluation, and sampling compared to previous flows while achieving state of the art on multiple large-scale datasets. Specifically, we introduce a simple one-dimensional one-layer network that has closed form Lipschitz constants and introduce a new Exact-Lipschitz Flow (ELF) that combines the ease of sampling from residual flows with the strong performance of autoregressive flows.

Though our model is provably a universal approximator, there is still a gap in performance between the state of the art flow for CIFAR-10 and our flow; future work entails exploring if using attention would further close the gap in performance.

As the flow we have introduced is a universal approximator that allows for more efficient sampling than autoregressive models, future research can further explore if the gap in performance of flows and autoregressive models such as PixelCNN++ (Salimans et al., 2017) is from dequantizing a discrete task, and what, if any, are the benefits of stacking autoregressive flows.

Acknowledgements
We thank Wenjing Ge (Bloomberg Quant Research) for comments on the paper and reviewing the proofs.
References

Anil, C., J. Lucas, and R. Grosse (2019, 09–15 Jun). Sorting out Lipschitz function approximation. In K. Chaudhuri and R. Salakhutdinov (Eds.), Proceedings of the 36th International Conference on Machine Learning, Volume 97 of Proceedings of Machine Learning Research, pp. 291–301. PMLR.

Arjovsky, M., S. Chintala, and L. Bottou (2017, 06–11 Aug). Wasserstein generative adversarial networks. In D. Precup and Y. W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, Volume 70 of Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia, pp. 214–223. PMLR.

Behrmann, J., W. Grathwohl, R. T. Q. Chen, D. Duvenaud, and J.-H. Jacobsen (2019, 09–15 Jun). Invertible residual networks. In K. Chaudhuri and R. Salakhutdinov (Eds.), Proceedings of the 36th International Conference on Machine Learning, Volume 97 of Proceedings of Machine Learning Research, Long Beach, California, USA, pp. 573–582. PMLR.

Cao, N. D., W. Aziz, and I. Titov (2019). Block neural autoregressive flow. In Conference on Uncertainty in Artificial Intelligence, Volume 3, pp. 1723–1735. UAI.

Chen, J., C. Lu, B. Chenli, J. Zhu, and T. Tian (2020, 13–18 Jul). VFlow: More expressive generative flows with variational data augmentation. In H. D. III and A. Singh (Eds.), Proceedings of the 37th International Conference on Machine Learning, Volume 119 of Proceedings of Machine Learning Research, pp. 1660–1669. PMLR.

Chen, R. T. Q., J. Behrmann, D. K. Duvenaud, and J.-H. Jacobsen (2019). Residual flows for invertible generative modeling. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Eds.), Advances in Neural Information Processing Systems 32, pp. 9916–9926. Curran Associates, Inc.

Chen, R. T. Q., Y. Rubanova, J. Bettencourt, and D. K. Duvenaud (2018). Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Advances in Neural Information Processing Systems 31, pp. 6571–6583. Curran Associates, Inc.

Chrabaszcz, P., I. Loshchilov, and F. Hutter (2017). A downsampled variant of imagenet as an alternative to the cifar datasets. arXiv preprint arXiv:1707.08819.

Clevert, D.-A., T. Unterthiner, and S. Hochreiter (2015). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In International Conference on Learning Representations.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems 2(4), 303–314.

Dinh, L., D. Krueger, and Y. Bengio (2014, October). NICE: Non-linear Independent Components Estimation. arXiv e-prints, arXiv:1410.8516.

Dinh, L., J. Sohl-Dickstein, and S. Bengio (2017). Density estimation using Real NVP. International Conference on Learning Representations.

Dua, D. and C. Graff (2017). UCI machine learning repository.

Durkan, C., A. Bekasov, I. Murray, and G. Papamakarios (2019). Neural spline flows. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Eds.), Advances in Neural Information Processing Systems, Volume 32. Curran Associates, Inc.

Fazlyab, M., A. Robey, H. Hassani, M. Morari, and G. Pappas (2019). Efficient and accurate estimation of lipschitz constants for deep neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Eds.), Advances in Neural Information Processing Systems, Volume 32. Curran Associates, Inc.

Germain, M., K. Gregor, I. Murray, and H. Larochelle (2015, 07–09 Jul). Made: Masked autoencoder for distribution estimation. In F. Bach and D. Blei (Eds.), Proceedings of the 32nd International Conference on Machine Learning, Volume 37 of Proceedings of Machine Learning Research, Lille, France, pp. 881–889. PMLR.

Gopal, A. (2020, September). Quasi-Autoregressive Residual (QuAR) Flows. arXiv e-prints, arXiv:2009.07419.
Grathwohl, W., R. T. Q. Chen, J. Bettencourt, and D. Duvenaud (2019). Scalable reversible generative models with free-form continuous dynamics. In International Conference on Learning Representations.

Gulrajani, I., F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville (2017). Improved training of wasserstein gans. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Advances in Neural Information Processing Systems, Volume 30. Curran Associates, Inc.

Ha, D., A. Dai, and Q. V. Le (2016). Hypernetworks. arXiv preprint arXiv:1609.09106.

Ho, J., X. Chen, A. Srinivas, Y. Duan, and P. Abbeel (2019, 09–15 Jun). Flow++: Improving flow-based generative models with variational dequantization and architecture design. In K. Chaudhuri and R. Salakhutdinov (Eds.), Proceedings of the 36th International Conference on Machine Learning, Volume 97 of Proceedings of Machine Learning Research, Long Beach, California, USA, pp. 2722–2730. PMLR.

Huang, C.-W., R. T. Q. Chen, C. Tsirigotis, and A. Courville (2021). Convex potential flows: Universal probability distributions with optimal transport and convex optimization. In International Conference on Learning Representations.

Huang, C.-W., L. Dinh, and A. Courville (2020, February). Augmented Normalizing Flows: Bridging the Gap Between Generative Flows and Latent Variable Models. arXiv e-prints, arXiv:2002.07101.

Huang, C.-W., D. Krueger, A. Lacoste, and A. Courville (2018, 10–15 Jul). Neural autoregressive flows. In J. Dy and A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, Volume 80 of Proceedings of Machine Learning Research, Stockholmsmässan, Stockholm Sweden, pp. 2078–2087. PMLR.

Huster, T., C.-Y. J. Chiang, and R. Chadha (2018). Limitations of the lipschitz constant as a defense against adversarial examples. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 16–29. Springer.

Jaini, P., K. A. Selby, and Y. Yu (2019, 09–15 Jun). Sum-of-squares polynomial flow. In K. Chaudhuri and R. Salakhutdinov (Eds.), Proceedings of the 36th International Conference on Machine Learning, Volume 97 of Proceedings of Machine Learning Research, pp. 3009–3018. PMLR.

Kahn, H. (1955). Use of Different Monte Carlo Sampling Techniques. RAND Corporation.

Kingma, D. P. and J. Ba (2015). Adam: A method for stochastic optimization. In International Conference on Machine Learning.

Kingma, D. P. and P. Dhariwal (2018). Glow: Generative flow with invertible 1x1 convolutions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Advances in Neural Information Processing Systems 31, pp. 10215–10224. Curran Associates, Inc.

Kingma, D. P., T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling (2016). Improved variational inference with inverse autoregressive flow. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.), Advances in Neural Information Processing Systems 29, pp. 4743–4751. Curran Associates, Inc.

Kong, Z. and K. Chaudhuri (2020, 26–28 Aug). The expressive power of a class of normalizing flow models. In S. Chiappa and R. Calandra (Eds.), Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics, Volume 108 of Proceedings of Machine Learning Research, pp. 3599–3609. PMLR.

Krizhevsky, A. and G. Hinton (2009). Learning multiple layers of features from tiny images. Master’s thesis, Department of Computer Science, University of Toronto.

LeCun, Y. and C. Cortes (2010). MNIST handwritten digit database.

Ma, X., X. Kong, S. Zhang, and E. Hovy (2019). Macow: Masked convolutional generative flow. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Eds.), Advances in Neural Information Processing Systems, Volume 32. Curran Associates, Inc.
Martin, D., C. Fowlkes, D. Tal, and J. Malik (2001, July). A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In Proc. 8th Int'l Conf. Computer Vision, Volume 2, pp. 416–423.

Miyato, T., T. Kataoka, M. Koyama, and Y. Yoshida (2018). Spectral normalization for generative adversarial networks. In International Conference on Learning Representations.

Oliva, J., A. Dubey, M. Zaheer, B. Poczos, R. Salakhutdinov, E. Xing, and J. Schneider (2018, 10–15 Jul). Transformation autoregressive networks. In J. Dy and A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, Volume 80 of Proceedings of Machine Learning Research, pp. 3898–3907. PMLR.

Oord, A. V., N. Kalchbrenner, and K. Kavukcuoglu (2016, 20–22 Jun). Pixel recurrent neural networks. In M. F. Balcan and K. Q. Weinberger (Eds.), Proceedings of The 33rd International Conference on Machine Learning, Volume 48 of Proceedings of Machine Learning Research, New York, New York, USA, pp. 1747–1756. PMLR.

Papamakarios, G., T. Pavlakou, and I. Murray (2017). Masked autoregressive flow for density estimation. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Advances in Neural Information Processing Systems 30, pp. 2338–2347. Curran Associates, Inc.

Polyak, B. and A. Juditsky (1992, 07). Acceleration of stochastic approximation by averaging. SIAM Journal on Control and Optimization 30, 838–855.

Rezende, D. and S. Mohamed (2015, 07–09 Jul). Variational inference with normalizing flows. In F. Bach and D. Blei (Eds.), Proceedings of the 32nd International Conference on Machine Learning, Volume 37 of Proceedings of Machine Learning Research, Lille, France, pp. 1530–1538. PMLR.

Salimans, T., A. Karpathy, X. Chen, and D. P. Kingma (2017). Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. In International Conference on Learning Representations.

Skilling, J. (1989). The eigenvalues of mega-dimensional matrices. In Maximum Entropy and Bayesian Methods, pp. 455–466. Springer.

Song, Y., C. Meng, R. Liao, and S. Ermon (2021, 18–24 Jul). Accelerating feedforward computation via parallel nonlinear equation solving. In M. Meila and T. Zhang (Eds.), Proceedings of the 38th International Conference on Machine Learning, Volume 139 of Proceedings of Machine Learning Research, pp. 9791–9800. PMLR.

Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus (2014). Intriguing properties of neural networks. In International Conference on Learning Representations.

Teshima, T., I. Ishikawa, K. Tojo, K. Oono, M. Ikeda, and M. Sugiyama (2020). Coupling-based invertible neural networks are universal diffeomorphism approximators. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Advances in Neural Information Processing Systems, Volume 33, pp. 3362–3373. Curran Associates, Inc.

van den Berg, R., L. Hasenclever, J. M. Tomczak, and M. Welling (2018). Sylvester normalizing flows for variational inference. In A. Globerson and R. Silva (Eds.), Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018, pp. 393–402. AUAI Press.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Advances in Neural Information Processing Systems, Volume 30, pp. 5998–6008. Curran Associates, Inc.

Virmaux, A. and K. Scaman (2018). Lipschitz regularity of deep neural networks: analysis and efficient estimation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Advances in Neural Information Processing Systems, Volume 31. Curran Associates, Inc.

Wiggers, A. and E. Hoogeboom (2020, 13–18 Jul). Predictive sampling with forecasting autoregressive models. In H. D. III and A. Singh (Eds.), Proceedings of the 37th International Conference on Machine Learning, Volume 119 of Proceedings of Machine Learning Research, pp. 10260–10269. PMLR.
A Universality

Definition A.1. A function $f : \mathbb{R}^d \to \mathbb{R}$ is called Lipschitz continuous if there exists a constant $L$, such that
\[ \|f(x_1) - f(x_2)\|_2 \leq L \|x_1 - x_2\|_2, \forall x_1, x_2 \in \mathbb{R}^d \]
$f$ is called an $L$-Lipschitz function. The smallest $L$ that satisfies the inequality is called the Lipschitz constant of $f$, denoted as $\text{Lip}(f)$.

Definition A.2. We define $\mathcal{F}_{\sigma, H}$ as the family of functions $f : \mathbb{R}^1 \to \mathbb{R}^1$ represented by a single layer neural network with a quadratic-piecewise 1-Lipschitz activation function $\sigma$ and hidden size $H$. $\mathcal{F}_{\text{RELU}, H}$ and $\mathcal{F}_{\text{ELU}, H}$ are where the activation function is FELU and ReLU, respectively.

Definition A.3. We define $\mathcal{L}_{\sigma, H}$ as the family of functions where $g \in \mathcal{L}_{\sigma, H}$ if there exists a function $f \in \mathcal{F}_{\sigma, H}$ such that $g(x) = f(x)$ for all $x \in \mathbb{R}$ if $\text{Lip}(f) \leq 1$ or $g(x) = f(x)/\text{Lip}(f)$ for all $x \in \mathbb{R}$.

Definition A.4. ($L^p$-sup-universality (Teshima et al., 2020)) Let $\mathcal{M}$ be a model which is a set of measurable mappings from $\mathbb{R}^m \to \mathbb{R}^n$. Let $p \in [1, \infty)$ and let $\mathcal{F}$ be a set of measurable mappings $f : U_f \to \mathbb{R}^n$ where $U_f$ is a measurable subset of $\mathbb{R}^m$ which may depend on $f$. We say that $\mathcal{M}$ is an $L^p$-approximator for $\mathcal{F}$ if for any $f \in \mathcal{F}$, any $\epsilon > 0$, and any compact subset $K \subset U_f$, there exists a $g \in \mathcal{M}$ such that
\[ \left( \int_K \|f(x) - g(x)\|_2^p \right)^{1/p} < \epsilon \]
We say that $\mathcal{M}$ is a sup-approximator for $\mathcal{F}$ if for any $f \in \mathcal{F}$, any $\epsilon > 0$, and any compact subset $K \subset U_f$, there exists a $g \in \mathcal{M}$ such that $\sup_{x \in K} \|f(x) - g(x)\|_2 < \epsilon$.

Definition A.5. (Elementwise affine transformation: $A_d$) We define $A_d$ as the set of all elementwise affine transforms $\{x \to Ax + b | A \in \mathcal{D}, b \in \mathbb{R}^d\}$ where $\mathcal{D}$ denotes the set of diagonal matrices $A \in \mathbb{R}^{d \times d}$ with $A_{ii} \in \mathbb{R}_+$.

Definition A.6. (Triangular transformation: $T^\infty$ (Teshima et al., 2020)) We define $T^\infty$ as the set of all $C^\infty$-increasing triangular mappings from $\mathbb{R}^d \to \mathbb{R}^d$. Here, a mapping $T = (T_1, \ldots, T_d) : \mathbb{R}^d \to \mathbb{R}^d$ is increasing triangular if for each $T_k(x) = T_k(x_k, x_{<k})$ is strictly increasing with respect to $x_k$ for a fixed $x_{<k}$ where $x \in \mathbb{R}^d$.

Lemma A.1. (Teshima et al., 2020, Lemma 1) An $L^p$-universal approximator for $T^\infty$ is a distributional universal approximator.

Using Lemma A.1 and that sup-universality implies $L^p$-universality, we simply need to prove that a composition of an elementwise affine layer ($A_d$) and ELF is a sup-universal approximator of $T^\infty$.

Definition A.7. ($1$-Lipschitz triangular transformation: $\mathcal{P}_{\text{tri}}^\infty$) We define $\mathcal{P}_{\text{tri}}^\infty$ as the set of all $C^\infty$-increasing $1$-Lipschitz triangular mappings from $\mathbb{R}^d \to \mathbb{R}^d$. Here, a mapping $P = (P_1, \ldots, P_d) : \mathbb{R}^d \to \mathbb{R}^d$ is increasing triangular if for each $P_k(x) = P_k(x_k, x_{<k})$ is strictly increasing with respect to $x_k$ for a fixed $x_{<k}$ where $x \in \mathbb{R}^d$ and $P_k$ is $1$-Lipschitz with respect to the first argument ($\sup_{x_k} \frac{\partial P_k}{\partial x_k} < 1$).

Lemma A.2. Given a set of triangular $1$-Lipschitz functions $\mathcal{M}$ that is a sup-universal approximator of $\mathcal{P}_{\text{tri}}^\infty$, then the set of functions $\{f \circ (I + g) | f \in A_d, g \in \mathcal{M}\}$ is a sup-universal approximator of $T^\infty$.

Proof. Given a multivariate continuously differentiable function $T(x)_t = T_t(x_t, x_{<t})$ for $t \in [1, m]$ that is strictly monotonic with respect to the first argument when the second argument in fixed (i.e. $T \in T^\infty$), we define $D_t$ as:
\[ \sup_{x_{<t}} \frac{1}{2} \left| \frac{\partial T_t(x_t, x_{<t})}{\partial x_t} \right| \]
If we divide $T(x)_t$ by $D_t$, then the resulting function is $2$-Lipschitz. Further, we can write $T(x)_t = D_t (x_t + \left( \frac{T(x)_t}{D_t} - x_t \right)) = D_t (x_t + \hat{g}(x_t, x_{<t}))$ where $\hat{g}(x_t, x_{<t})$ is a $1$-Lipschitz function with respect to the first argument. Say $\hat{g}(x_t, x_{<t}) \in \mathcal{M}$ is $\epsilon/D_t$ close to $g(x_t, x_{<t})$, then:
\[ \sup_{x_t} \left| D_t (x_t + \hat{g}(x_t, x_{<t})) - T(x)_t \right| \leq \epsilon \]
From here, we simply need to prove that ELF is a universal approximator of triangular 1-Lipschitz functions.

**Lemma A.3.** Given a compact set $K \subset [a, b]$, if $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ are $L$-Lipschitz and are at least $\epsilon$ close at the endpoint $a$ and $b$, then

$$
\sup_{x \in [a, b]} \left| f(x) - g(x) \right| \leq L(b - a) + \epsilon
$$

**Proof.**

$$
\begin{align*}
\sup_{x \in [a, b]} \left| f(x) - g(x) \right| & \leq \sup_{x \in [a, b]} \min \left( \left| g(x) - f(a) \right| + \left| g(a) - g(x) \right|, \left| g(x) - f(b) \right| + \left| g(b) - g(x) \right| \right) \\
& \leq \sup_{x \in [a, b]} \min \left( \left| g(x) - g(a) \right| + \epsilon + \left| g(a) - g(x) \right|, \left| g(x) - g(b) \right| + \epsilon + \left| g(b) - g(x) \right| \right) \\
& \leq \sup_{x \in [a, b]} \min \left( 2L|x - a| + \epsilon, 2L|x - b| + \epsilon \right) \\
& \leq L(b - a) + \epsilon
\end{align*}
$$

\[\square\]

### A.1 1-D Universality for ReLU

**Definition A.8.** We define $\mathcal{G}_{ReLU,H}$ as the set of functions $f \in \mathcal{F}_{ReLU,H}$ where the values of the weight matrix in the first layer are one.

**Lemma A.4.** $\mathcal{G}_{ReLU,H}$ is a sup-approximator of $\mathcal{P}_1^\infty$.

**Proof.** Given some $\epsilon > 0$ and a compact set $K \subset [a, b]$, we choose $H = 2\left\lceil \frac{b-a}{\epsilon} \right\rceil$ and divide the range $[a, b]$ into evenly spaced intervals $n = H/2$ intervals: $(x_0 = a, x_1, x_2, \ldots, x_{n-1}, x_n = b)$. We denote the target function as $f$ and the function we are learning:

$$
\hat{f}(x) = b_2 + \sum_{i=1}^{H} w_i \text{ReLU}(x + b_{1,i})
$$

For each interval $(x_i, x_{i+1})$, we set $b_{1,2i}, b_{1,2i+1}, w_{2i}$, and $w_{2i+1}$ such that $\hat{f}(x_i) = f(x_i)$ and $\hat{f}(x_{i+1}) = f(x_{i+1})$.

Using induction over the intervals, we want $\hat{f} = f$ at each of the end points of the evenly spaced intervals and $\frac{\partial \hat{f}}{\partial x} = 0$ for $x < a$ and $\frac{\partial \hat{f}}{\partial x} = 0$ for $x > b$ (points to the right of the rightmost interval). We first set $b_2 = f(a)$ and all other parameters to zero.

For the interval $(x_i, x_{i+1})$, we have $\hat{f}(x_i) = f(x_i)$. We set:

$$
\begin{align*}
 w_{2i} &= \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} & b_{1,2i} &= -x_i \\
 w_{2i+1} &= -\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} & b_{1,2i+1} &= -x_{i+1}
\end{align*}
$$

From the definition of ReLU, all points $x \leq x_i$ are unaffected by the change in parameters. Further, $\hat{f}(x_i) = f(x_i), \hat{f}(x_{i+1}) = f(x_{i+1})$, and $\frac{\partial \hat{f}}{\partial x} = 0$ for $x > x_{i+1}$.

By construction and from the fact that $f$ is 1-Lipschitz, $\hat{f}$ is 1-Lipschitz. Using Lemma A.3, within any interval:

$$
\sup_{x \in [x_i, x_{i+1}]} \left| f(x) - \hat{f}(x) \right| \leq \epsilon
$$

\[\square\]
A.2 1-D Universality for FELU

Lemma A.5. $F_{\text{FELU}, H}$ is a sup-approximator of $P_1^\infty$.

Proof. Given some $\epsilon > 0$ and a compact set $K \subset [a, b]$, we choose $H = 2 \left \lfloor \frac{b-a}{\epsilon n} \right \rfloor$ and divide the range $[a, b]$ into evenly spaced intervals $n = H/2$ intervals: $(x_0 = a, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n = b)$. We denote the target function as $f$ and the function we are learning:

$$\hat{f}(x) = b_2 + \sum_{i=1}^{H} w_{2,i} \text{FELU}(w_{1,i} x + b_{1,i})$$

(A.1)

Unlike for ReLU, we cannot set the parameters such that $\hat{f}(x_i) = f(x_i)$ and $\hat{f}(x_{i+1}) = f(x_{i+1})$ for all intervals.

Instead, inductively over the intervals, we want to set the parameters such that $|\hat{f}(x_i) - f(x_i)| < \epsilon_1$, then $|\hat{f}(x_{i+1}) - f(x_{i+1})| < \epsilon_1 + \frac{\epsilon}{2n}$ at each of the end points of the evenly spaced intervals. Further, we want $\frac{\partial \hat{f}}{\partial x} = 0$ for $x < a$ and $\frac{\partial \hat{f}}{\partial x} = 0$ for $x > b$ (points to the right of the rightmost interval). We first set $b_2 = f(a)$ and all other parameters to zero.

The intuition of the following construction comes from:

$$\lim_{w_1 \to \infty} w_2/w_1 \text{FELU}(w_1(x + b)) = w_2 \text{ReLU}(x + b)$$

For the interval $(x_i, x_{i+1})$, we have $|\hat{f}(x_i) - f(x_i)| < \epsilon_1$. We denote $L_i = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$.

$$w_{1,2i} = w_{1,2i+1} = 2n/\epsilon,$$

$$w_{2,2i} = L_i/w_{1,2i} \quad b_{1,2i} = -1 - x_i w_{1,2i}$$

$$w_{2,2i+1} = L_i/w_{1,2i+1} \quad b_{1,2i+1} = -x_{i+1} w_{1,2i}$$

$$b_2 = b_2 + \frac{(w_{2,2i} + w_{2,2i+1})/2}{2n/\epsilon}$$

where we update the value for $b_2$ since FELU$(x) = -1/2$ for $x < -1$.

From construction, all values $x \leq x_i$ are unaffected and $\frac{\partial \hat{f}}{\partial x} = 0$ for $x > x_{i+1}$. Further, by construction, $\hat{f}$ is 1-Lipschitz and $|\hat{f}(x_{i+1}) - f(x_{i+1})| < \epsilon_1 + 1/w_{1,2i} = \epsilon_1 + \frac{\epsilon}{2n}$.

Using Lemma A.3:

$$\sup_{x \in [a, b]} |f(x) - \hat{f}(x)| \leq \sup_{i \in [0, \ldots, n-1]} \sup_{x \in [x_i, x_{i+1}]} |f(x) - \hat{f}(x)|$$

$$\leq \sup_{i \in [0, \ldots, n-1]} (i + 1) \frac{\epsilon}{2n} + \epsilon/2$$

$$\leq n \frac{\epsilon}{2n} + \epsilon/2$$

$$\leq \epsilon$$

\[ \square \]

A.3 Universality of Higher Dimensional Distribution

Theorem A.1. Let $x \in [a, b]^d$ where $a, b \in \mathbb{R}$. Given any $0 < \epsilon < 1$ and any multivariate continuously differentiable function $P(x) = P_t(x, x_{<t})$ for $t \in [1, d]$ that is strictly monotonic and 1-Lipschitz with respect to the first argument when the second argument is fixed (i.e. $P(x) \in \mathcal{P}^\infty_d$), then there exists a multivariate function $\hat{P} \in \mathcal{P}^\infty_d$, such that $\|\hat{P}(x) - P(x)\| < \epsilon$ for all $x$, of the following form:

$$\hat{P}(x)_t = \frac{M_{x_{<t}}(x_t)}{\max(1, \text{Lip}(M_{x_{<t}}))}$$
where:
\[
M_{x,c}(x) = b_2 + \sum_{i=1}^{H} w_i \text{ReLU}(x + b_1) \tag{A.2}
\]

where \(b_2, w_i\) and \(b_1\) may depend on \(x_{<t}\), and \(\text{Lip}(M_{x,c})\) is the Lipschitz constant of Equation A.2.

**Proof.** We denote \(C(x_{<t})\) as the function that maps from \(x_{<t}\) to the parameters of Equation A.2 based on the construction in Lemma A.4 for the univariate function \(P_1(\cdot, x_{<t})\).

For shorthand, we will use \(c\) to denote \(x_{<t}\), use \(M(c)\) to denote a neural (hyper)network that outputs the parameters of Equation A.2, use \(C\) and \(M\) to denote the univariate function created by the outputs of \(C(c)\) and \(M(c)\) respectively, and use \(L_M\) to denote \(\max(1, \text{Lip}(M_c))\).

The crux of the proof is that from Lemma A.4, we can construct an \(\epsilon_1\)-close approximation \(C_c\) to \(P_1(\cdot, x_{<t})\) \(\mathbb{R} \to \mathbb{R}\) when the second argument is fixed, and we can then approximate \(C(c)\) with a neural network to get an \(\delta\)-close approximation. Since the specification in Lemma A.4 depends only on the function value at specific pre-defined points, the parameters of Equation A.2 (the output of the target function \(C(c)\)) is continuous with respect to \(x_{<t}\). Using the fact that \(C(c)\) is continuous, we can apply the classic results of Cybenko (1989) which states that a multilayer perceptron can approximate any continuous function on a compact subset of \(\mathbb{R}^d\), giving us \(M(c)\) as a \(\delta\)-close approximation to \(C(c)\).

More specifically, given a fixed \(c\):
\[
\sup_{x \in [a,b]} \left| \frac{M_c(x)}{L_M} - P(x) \right| \leq \sup_{x \in [a,b]} \left| \frac{M_c(x)}{L_M} - M_c(x) \right| + \sup_{x \in [a,b]} \left| M_c(x) - P(x) \right| \tag{A.3}
\]
\[
\leq \sup_{x \in [a,b]} \left| \frac{M_c(x)}{L_M} - M_c(x) \right| + \epsilon_1 \tag{A.4}
\]

where \(C_c\) is from Lemma A.4.

Simplifying the first term:
\[
\sup_{x \in [a,b]} \left| \frac{M_c(x)}{L_M} - M_c(x) \right| \leq \sup_{x \in [a,b]} \left| \frac{M_c(x)}{L_M} - M_c(x) \right| + \sup_{x \in [a,b]} \left| M_c(x) - C_c(x) \right| \tag{A.5}
\]
\[
\leq \frac{L_M - 1}{L_M} \sup_{x \in [a,b]} \left| M_c(x) \right| + \sup_{x \in [a,b]} \left| M_c(x) - C_c(x) \right| \tag{A.6}
\]
\[
\leq (L_M - 1) \sup_{x \in [a,b]} \left| M_c(x) \right| + \sup_{x \in [a,b]} \left| M_c(x) - C_c(x) \right| \tag{A.7}
\]

To bound the effect of having a \(\delta\)-close approximation to \(C(c)\), we need to bound \(L_M\) and \(\sup_{x \in [a,b]} |M_c(x)|\).

Since Equation A.2 and its Lipschitz constant are both continuous with respect to the parameters and on a compact set, the two are uniformly continuous. By uniform continuity, for some \(\epsilon_2\), there exists a \(\delta_1\) such that when \(\|M(c) - C(c)\|_2 < \delta_1, |L_M - L_c| < \epsilon_2\). Thus,
\[
L_M \leq \max(1, \text{Lip}(M_c)) \tag{A.8}
\]
\[
\leq \max(1, \text{Lip}(C_c) + \epsilon_2) \tag{A.9}
\]
\[
\leq 1 + \epsilon_2 \tag{A.10}
\]

Similarly, for some \(\epsilon_3\), there exists a \(\delta_2\) such that when \(\|M(c) - C(c)\|_2 < \delta_2, |M_c(x) - C_c(x)| < \epsilon_3\). Thus,
\[
\sup_{x \in [a,b]} |M_c(x)| \leq \sup_{x \in [a,b]} |M_c(x) - C_c(x)| + \sup_{x \in [a,b]} C_c(x) \tag{A.11}
\]
\[
\leq \epsilon_3 + \|C_c\|_{\infty} \tag{A.12}
\]

where \(\|C_c\|_{\infty}\) is finite due to the compactness of the domain.

Using \(\delta = \min(\delta_1, \delta_2)\) and that \(M(c)\) is \(\delta\)-close \(C(c)\), plugging all these into Equation A.7,
\[
\sup_{x \in [a,b]} \left| \frac{M_c(x)}{L_M} - C_c(x) \right| \leq \epsilon_2(\epsilon_3 + \|C_c\|_{\infty}) + \epsilon_3 \tag{A.13}
\]
Finally, using Equation A.4, \( \epsilon_1 < \frac{1}{2} \), and \( \epsilon_2 = \epsilon_3 < \min(\frac{1}{2}, \frac{1}{2 \max_{c \in [c]\|_{\infty}}} ) \), we have

\[
\sup_{x,c} \left| \frac{M_c(x)}{L_M} - P(x) \right| < \epsilon
\]

Having proved the univariate case, from the above, we know that given any \( \epsilon > 0 \) for each \( t \), there exists \( \delta_t \) such that \( \sup_{x \in [a,b]} \| P(x) - \hat{P}(x) \| < \epsilon \) for all \( x < t \). Choosing \( \delta_{full} = \max_{t \in [1,d]} \delta_t \), we have \( \| \hat{P}(x) - P(x) \| < \epsilon \) for all \( x \).

Though we proved universality for ReLU, due to the construction of Lemma A.5, the proof of universality is a straightforward generalization of the above proof since Equation A.1 and its Lipschitz constant are both continuous with respect to the parameters.

## B Complexity Analysis of Lipschitz Constant Computation

In Section 4.2, we introduce Equation 4.3 which shows the computation we perform to compute the Lipschitz constant of ELF. In this section, we perform a more thorough analysis of the runtime complexity of this computation. For optimization, we implemented the computation using CUDA to fully utilize the parallel capabilities of GPUs (Appendix I).

As was discussed in Section 4.1, for a quadratic piecewise activation function with \( N \) pieces for a one-layer network with hidden size \( H \), the number of gradient evaluations required to compute the Lipschitz constant is \( (N - 1) \cdot H + 1 \). In the case of FELU (with three pieces), Equation 4.3 requires only \( 2 \cdot H \) evaluations. The reason for the removal of the +1 is that the gradient is continuous for FELU and the two ends of the activation function is linear (i.e. constant gradient).

If the gradient of the activation were not continuous (e.g. ReLU), evaluation at the points where the gradient changes (e.g. \( w_{1,i} x_i + b_{1,i} = 0 \) for ReLU) could still be used; however, a convention must be chosen for the gradient at that point. The choice of convention would inform which of the \( N \cdot H + 1 \) gradients has not been calculated.

Focusing on FELU, we can see that the runtime complexity of \( \frac{\partial g}{\partial x} \) is \( O(H) \) (compute the gradient coming from each neuron). Further, since the number of gradient evaluations we need to do is \( 2H = O(H) \), the full runtime complexity is \( O(H^2) \).

Empirically, we evaluate the runtime as a function of hidden size which we expect to be quadratic (Figure 6a) and as a function of batch size which we expect to be linear (Figure 6b). Further, we verify the assumption by fitting a polynomial curve and checking the \( R^2 \) of the fit (0.998 and 0.993 respectively). The batch size in Figure 6b indicates the number of one-dimensional Lipschitz constants computed.

To put into context the importance of the batch size, given the description of the model in Section 4.3, the number of Lipschitz constants to be computed for a batch size \( B \) for \( D \)-dimensional input would be \( D \cdot B \) (this number can be thought of as the batch size in Figure 6b). For example, for our MNIST model, for a batch size of 64, the number of Lipschitz constants computed per flow is \( 784 \cdot 64 = 50176 \).
Figure 6: Empirical runtime analysis of Lipschitz computation on a GPU. As a function of hidden size, the runtime is quadratic; as a function of batch size, the runtime is linear. The batch size used for Figure 6a is $64 \cdot 784$.

Table 5: Comparison of efficiency among variationally dequantized discrete flow-models on CIFAR-10. Flow++ and VFlow results taken from Chen et al. (2020). The validation set used by Chen et al. (2020) is different from the set we evaluated our model on; specifically, Chen et al. (2020) used a random subset of the training set whereas we used CIFAR-10’s official test set.

| Flow Model        | Bits/dim | Param. Count |
|-------------------|----------|--------------|
| 3-channel Flow++  | 3.23     | 4.02M        |
| 4-channel VFlow   | 3.16     | 4.03M        |
| 6-channel VFlow   | 3.13     | 4.01M        |
| ELF-AR (Ours)     | 3.18     | 4.1M         |

C Parameter Efficiency for Variationally Dequantized Flows

Most models shown for parameter efficiency have yet to be combined with variational dequantization, thus not allowing for a fair comparison. However, in terms of parameter efficiency, Chen et al. (2020) performed an ablation study under a fixed parameter budget, specifically with approximately 4 million parameters. The results on a validation set were reported (shown in Table 5); however, the validation set used by Chen et al. (2020) was a random subset of 10,000 images from the training set. We performed a similar experiment with approximately four million parameters; however, we use the original test set of CIFAR-10 and report the performance on this set.

D ReLU Flow

To show that residual flows and ELF with ReLU does not learn, we trained a flow on a uniform distribution. In Figure 7, we can see that both ELF and residual flows with ReLU are unable to learn anything meaningful, even though we gave a hidden size of 2048. On the other hand, simply replacing ReLU with ELU improved performance for the residual flow.

All the flows here are placed in between two affine flows. The only difference between ELF and residual flow is that ELF uses the exact Lipschitz computation whereas residual flow uses spectral norm.

Speculatively, we believe that the fact ReLU does not work comes from a combination of the fact that we optimize flows using gradient based methods, the loss function of flows requires first derivatives (meaning gradient based methods require second derivatives of the flow) and that ReLU has zero second derivative everywhere.
Figure 7: Comparison of using ReLU vs ELU in flows

Figure 8: Training loss curves for mixture of eight gaussians

Figure 9: Training loss curves for checkerboard data

E More Synthetic Data Results

Similar to the experiment in Section 5.2, we tested out numerous flows on checkerboard data (Figure 10). However, one difference is that we stacked five flows instead of just one. In Table 6, we see that ELF-AR and NAF are both able to perform strongly; however, a key difference being that we can sample from ELF-AR.
Figure 10: The contour plots in log space of mixture of Gaussians. The levels shown in each subfigure are the same.

Table 6: Comparison of different flows on fitting to checkerboard data where the architectures for each were chosen so that they had the same number of layers and approximately equivalent number of parameters (500K). Implementation taken from https://github.com/CW-Huang/CP-Flow.

| Checkerboard | Log-Likelihood | Train Time (m) | Inference Time (s) | Sample Speed (s) |
|--------------|----------------|----------------|--------------------|------------------|
| Affine Coupling | -3.60         | 6.4            | 0.04               | 0.004            |
| NAF          | **-3.56**     | 20.7           | 0.06               | N/A              |
| Residual Flow | -3.90         | 21.1           | 0.10               | 0.31             |
| CP-Flow      | -3.60         | 143            | 0.69               | 3.5              |
| ELF-AR (Ours) | **-3.56**     | 12.6           | 0.20               | 0.54             |

We further show the training NLL as a function of time and updates in Figure 8 and Figure 9. We do not show CP-Flow as the loss used during training is a surrogate loss.
F Experimental Setup

For all experiments, we trained on a single Tesla V100-SXM2-32GB GPU, except for the Imagenet experiments where we used four GPUs for Imagenet 32 and eight GPUs for Imagenet 64. All image experiments were run for up to 2 weeks.

F.1 Synthetic Data

The mixture of Eight Gaussians was created using https://github.com/CW-Huang/CP-Flow. The checkerboard dataset was created using https://github.com/rtqichen/residual-flows.

For each method in our synthetic data, we used four hidden layers within each flow. For the results in Table 1, we used one flow; for the results in Table 6, we used five flows.

For mixture of eight gaussians, we used a hidden size of 256 for Affine Coupling and Residual Flow, 192 for ELF-AR, 384 for CP-Flow, and 256 for NAFs. For checkerboard data, we used a hidden size of 196 for Affine Coupling and Residual Flow, 128 for ELF-AR, 256 for CP-Flow, and 160 for NAFs.

For each method, we optimized using Adam (Kingma and Ba, 2015), starting with a learning rate of 2e-3 or 5e-3 (depending on whichever gave stronger performance). We trained for 10K steps with a batch size of 128, halving the learning rate every 2.5K steps.

F.2 Tabular Data

For all five datasets, we used five flows with five hidden layers. We hyperparameter tuned using a hidden size of 8d, 16d, 32d or 64d where d is the dimensionality of the dataset. We further explored a weight decay of 1e-4, 1e-5, and 1e-6. For each dataset, we trained for up to 1000 epochs, early stopping on the validation set. We used a batch size of 1024 for every dataset except MINIBOONE where we used a batch size of 128.

We optimized using Adam (Kingma and Ba, 2015) starting with a learning rate of 1e-3. Further, we clipped the norm of gradients to the range $[-1, 1]$ and reduced the learning rate by half (to no lower than 1e-4) whenever the validation loss did not improve by more than 1e-3, using patience of five epochs.

Of the five datasets, MINIBOONE and BSDS300 eventually started to overfit whereas the other three did not. We expect larger parameterizations with stronger regularization might further improve performance on these two datasets.

F.3 Image Data

For the image models, we used a multiscale architecture. For all four datasets, we used three scales, four flows per scale for MNIST and CIFAR-10 and eight flows per scale for the two Imagenet datasets. We used a hidden size of 256 for ELF. For the hypernetwork, we used the convolutional residual blocks from Oord et al. (2016); the residual block is:

$$\text{ReLU} \rightarrow 3x3xHxH\ Conv \rightarrow \text{ReLU} \rightarrow 1x1xHxH\ Conv \rightarrow \text{ReLU} \rightarrow 3x3xHx2H\ Conv$$ (F.1)

where the notation used for the convolution is the first two are the kernel sizes in the height and width direction, and the last two numbers are the input and output size respectively. We used $H = 192$ for all our full scale image experiments. Further, we used five residual blocks per flow.

For variational dequantization, we used a similar architecture for each flow except we condition on the image we are dequantizing. We used only one scale for dequantization instead of multiscale. To condition on an image, we first passed it through 4 residual blocks with hidden size 32:

$$\text{ReLU} \rightarrow 3x3x32x32\ Conv \rightarrow \text{ReLU} \rightarrow 3x3x32x32\ Conv$$

To condition each flow on this representation, we concatenated it to the beginning of each residual blocks (Equation F.1).

For training, we optimized using Adam (Kingma and Ba, 2015) with a learning rate of 1e-3 with a batch size of 64. We clipped the norm of gradients to the range $[-1, 1]$. We warmed up the learning rate over 100 steps; after 100K updates, we start to exponentially decay the learning rate by 0.99999 at every step until a learning rate of 2e-4, similar to the schedule used by Chen et al. (2020).

For our small architectures (Section 5.4.2), we used three residual blocks in each flow with a hidden size of 80 for CIFAR-10 and 84 for MNIST. For the experiment in Section C, we used a hidden size of 88 and the same dequantization model as was used for our full image model.
To handle the range of the data being $[0, 1]$, we used a logit transformation as the first layer in the network (Dinh et al., 2014). However, for MNIST, we went one step further. Since majority of the pixels values are 0, before the logit transformation, we first transformed the data using the CDF of a mixture of Uniform(0, 1/256) and Uniform(1/256, 1), weighed by the probability of a pixel value being in the corresponding range in the training set. This trick helps to make the pixel values more uniform. We found this improved the ease of learning.

For all image models, we used Polyak averaging (Polyak and Juditsky, 1992) with decay 0.999.

**G Bits Per Dimension**

The performance of log-likelihood models for images is often defined using bits per dimension. Given a dequantization distribution $q(x)$ for $x \in \mathbb{R}^d$, the bits per dimension is defined as

$$\frac{\log p(x) - \log q(x)}{d \log 2}$$

**H Image Generations**

We generated samples from our model with the best bits per dimensions for MNIST (Figure 11) and CIFAR-10 (Figure 12).

Figure 11: Random samples from MNIST
I Lipschitz Computation Code

In this section, we give our custom CUDA code for Lipschitz computation.
/**
 * Inspired by https://pytorch.org/tutorials/advanced/cpp_extension.html
 * Implements our algorithm in CUDA/C++.
 * Copyright 2021 Achintya Gopal. Distributed under the terms of the Apache 2.0 license.
 */
#include <ATen/ATen.h>
#include <torch/extension.h>

#include <cuda.h>
#include <cuda_runtime.h>

#include <vector>
#include <stdio.h>
#include <math.h>

// DIFFERENT CONSANTS.
// OPTIMIZING THESE MIGHT IMPROVE PERFORMANCE FURTHER.
#define TILE_WIDTH_1 4

__global__
void cuda_elf_lip_op(
    at::PackedTensorAccessor32<float, 3, torch::RestrictPtrTraits> block_diag1,
    at::PackedTensorAccessor32<float, 3, torch::RestrictPtrTraits> block_diag2,
    at::PackedTensorAccessor32<float, 3, torch::RestrictPtrTraits> test_vals,
    at::PackedTensorAccessor32<float, 3, torch::RestrictPtrTraits> test_vals1,
    at::PackedTensorAccessor32<float, 3, torch::RestrictPtrTraits> test_vals2,
    at::PackedTensorAccessor32<float, 3, torch::RestrictPtrTraits> res,
    int B, int I, int S
) {
    int b = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int s = blockIdx.z * blockDim.z + threadIdx.z;
    if ((b < B) && (i < I) && (s < (2 * S + 1))) {
        float sum_res = 0.0;
        float test_val = test_vals[b][i][s];
        int diff1_pos;
        int diff1_neg;
        int diff2_pos;
        int diff2_neg;
        int w1_pos;
        int w1_neg;
        float bd_prod;
        float denom;
        float diff2;
        for (int h = 0; h < S; h++) {
            diff1_pos = (test_val > test_vals1[b][i][h]);
            diff1_neg = 1 - diff1_pos;
            diff2_pos = (test_val > test_vals2[b][i][h]);
            diff2_neg = 1 - diff2_pos;
            w1_pos = (block_diag1[b][i][h] > 0);
            w1_neg = 1 - w1_pos;
            bd_prod = block_diag1[b][i][h] * block_diag2[b][i][h];
            diff2 = test_val - test_vals2[b][i][h];
            denom = test_vals2[b][i][h] - test_vals1[b][i][h];
            if (denom == 0) {
                denom = 1;
            }
        }
    }
}
void cuda_elf_lip(
    at::Tensor block_diag1, // B x I x S
    at::Tensor block_diag2, // B x I x S
    at::Tensor test_vals, // B x I x (2 * S + 1)
    at::Tensor test_vals1, // B x I x S
    at::Tensor test_vals2, // B x I x S
    at::Tensor res // B x I x (2 * S + 1)
) {
    int B = test_vals1.size(0);
    int I = test_vals1.size(1);
    int S = test_vals1.size(2);
    int H = (2 * S + 1);
    auto block_diag1A = block_diag1.packed_accessor32<float, 3, at::RestrictPtrTraits>();
    auto block_diag2A = block_diag2.packed_accessor32<float, 3, at::RestrictPtrTraits>();
    auto test_valsA = test_vals.packed_accessor32<float, 3, at::RestrictPtrTraits>();
    auto test_vals1A = test_vals1.packed_accessor32<float, 3, at::RestrictPtrTraits>();
    auto test_vals2A = test_vals2.packed_accessor32<float, 3, at::RestrictPtrTraits>();
    auto resA = res.packed_accessor32<float, 3, at::RestrictPtrTraits>();

    int TILE_WIDTH_2 = 1;
    int TILE_WIDTH_3 = 1;
    if (H > I) {
        TILE_WIDTH_2 = 1;
        TILE_WIDTH_3 = 32;
    } else {
        TILE_WIDTH_2 = 32;
        TILE_WIDTH_3 = 1;
    }

    unsigned int grid_x = (B + TILE_WIDTH_1 - 1) / TILE_WIDTH_1;
    unsigned int grid_y = (I + TILE_WIDTH_2 - 1) / TILE_WIDTH_2;
    unsigned int grid_z = (H + TILE_WIDTH_3 - 1) / TILE_WIDTH_3;
    dim3 grid_1(grid_x, grid_y, grid_z);
    dim3 block_1(TILE_WIDTH_1, TILE_WIDTH_2, TILE_WIDTH_3);
    cuda_elf_lip_op << < grid_1, block_1 >> > (
        block_diag1A,
        block_diag2A,
        test_valsA,
        test_vals1A,
        test_vals2A,
        resA, B, I, S
    );
}

sum_res += (
    diff1_pos * diff2_pos * w1_pos +
    diff1_neg * diff2_neg * w1_neg +
    (diff1_pos * diff2_neg - diff1_neg * diff2_pos)
    * (w1_pos - w1_neg)
    * diff2
    / denom
)
) * bd_prod;
}
res[b][i][s] = sum_res;
}