ButterflyFlow: Building Invertible Layers with Butterfly Matrices

Chenlin Meng [* 1] **Linqi Zhou** [* 1] **Kristy Choi** [* 1] **Tri Dao** [1] **Stefano Ermon** [1]

Abstract

Normalizing flows model complex probability distributions using maps obtained by composing invertible layers. Special linear layers such as masked and 1×1 convolutions play a key role in existing architectures because they increase expressive power while having tractable Jacobians and inverses. We propose a new family of invertible linear layers based on butterfly layers, which are known to theoretically capture complex linear structures including permutations and periodicity, yet can be inverted efficiently. This representational power is a key advantage of our approach, as such structures are common in many real-world datasets. Based on our invertible butterfly layers, we construct a new class of normalizing flow models called ButterflyFlow. Empirically, we demonstrate that ButterflyFlows not only achieve strong density estimation results on natural images such as MNIST, CIFAR-10, and ImageNet-32×32, but also obtain significantly better log-likelihoods on structured datasets such as galaxy images and MIMIC-III patient cohorts—all while being more efficient in terms of memory and computation than relevant baselines.

1. Introduction

Generative models have achieved tremendous success in a wide range of domains, such as images (Brock et al., 2018; Karras et al., 2020; Vahdat & Kautz, 2020; Ho et al., 2020; Song et al., 2020), natural language (Brown et al., 2020; Chowdhery et al., 2022), video (Kumar et al., 2019; Ho et al., 2022), molecule synthesis (Kadurin et al., 2017; De Cao & Kipf, 2018; Gómez-Bombarelli et al., 2018), and speech (Oord et al., 2018; Kong et al., 2020). Normalizing

flows, in particular, have attracted significant attention since they allow *exact* likelihood evaluation of data rather than lower-bound approximations (Dinh et al., 2014; Kingma & Dhariwal, 2018).

To build such normalizing flows, one must design flexible families of functions that are both invertible and admit efficient computation of Jacobian determinants (Rezende & Mohamed, 2015; Papamakarios et al., 2019; Hoogeboom et al., 2020; Karami et al., 2019; Finzi et al., 2019; Hoogeboom et al., 2019; Chen et al., 2019; Ho et al., 2019; Grcić et al., 2021). While the development of non-linear coupling layers fueled early progress in the field (Dinh et al., 2014; 2016), recent advances have focused on the effectiveness of special linear layers such as masked, 1×1, and $d \times d$ convolutions as key architectural primitives, among others (Ma et al., 2019; Kingma & Dhariwal, 2018; Hoogeboom et al., 2019; 2020). In particular, most state-of-the-art flow models first preprocess the data with such linear layers while also leveraging non-linear layers for expressivity.

In this work, we draw inspiration from the literature on learning efficient, structured linear transformations and propose a new class of invertible linear layers based on *butterfly layers* (Dao et al., 2019). Our invertible butterfly layer satisfies the usual desiderata of a normalizing flow primitive. However, its key distinguishing feature lies in its representational power: in spite of its efficiency, it inherits desirable properties from Dao et al. 2019 in that it is theoretically guaranteed to capture complex structures in data such as permutations and periodicity. The expressivity of invertible butterfly layers gives it an advantage over existing methods when *modeling real-world datasets that exhibit such structures*. We then construct a new family of normalizing flow models called ButterflyFlow by combining our proposed invertible butterfly layers with coupling layers (Dinh et al., 2016) and a Glow-based model backbone (Kingma & Dhariwal, 2018).

Empirically, we demonstrate that ButterflyFlow is an effective generative model, performing favorably relative to existing methods on image datasets such as MNIST, CIFAR-10, and ImageNet-32×32. However, we highlight that ButterflyFlow shines when modeling real-world data with special underlying structures, such as periodicity and permutations. Our model outperforms relevant baselines on the MIMIC-III patient dataset by approximately 200% in negative log-

---

[*]Equal contribution [1]Computer Science Department, Stanford University. Correspondence to: Chenlin Meng <chenlin@cs.stanford.edu>, Linqi Zhou <linqizhou@stanford.edu>, Kristy Choi <kechoi@cs.stanford.edu>, Stefano Ermon <ermon@cs.stanford.edu>.

likelihoods per dimension while requiring *less than half the number of model parameters*. In this way, our invertible butterfly layer serves as a powerful architectural primitive for capturing global regularities present in the data.

The contributions of our work can be summarized as:

1. We introduce ButterflyFlow, a new class of flow-based generative models parameterized by butterfly matrices.

2. We provide theoretical guarantees that ButterflyFlow can efficiently capture common types of structures, such as permutations.

3. We show empirically that ButterflyFlow achieves strong performance on density estimation and image synthesis tasks, and is superior at modeling data with special structure (e.g. periodicity) in real-world settings relative to existing flow-based models.

2. Preliminaries

2.1. Flow-based Generative Models

Given a data distribution $p_X(\mathbf{x})$ and a base distribution $p_Z(\mathbf{z})$ (*e.g.*, a Gaussian distribution), a normalizing flow is an invertible transformation $f_\theta : \mathbf{x} \in \mathbb{R}^n \mapsto \mathbf{z} \in \mathbb{R}^n$ that approximates $p_X(\mathbf{x})$ via the change of variables formula:

$$p_\theta(\mathbf{x}) = p_Z(\mathbf{z})|\det J_{f_\theta}(f_\theta^{-1}(\mathbf{z}))|, \qquad (1)$$

where J_{f_θ} is the Jacobian of $f(\mathbf{x})$, and θ is the set of learnable parameters. In practice, the Jacobian determinant $\det J_{f_\theta}(f_\theta^{-1}(\mathbf{z}))$ must be tractable to compute. Coupled with a simple $p_Z(\mathbf{z})$, the change of variables formula allows for the exact likelihood evaluation of a complex $p_X(\mathbf{x})$ as well as maximum likelihood training of f_θ. To sample a new data point from the model, we first draw samples $\mathbf{z} \sim p_Z(\mathbf{z})$ from the prior distribution and then push it through the inverse flow transformation: $\mathbf{x} = f_\theta^{-1}(\mathbf{z})$.

Because the normalizing flow's ability to capture complex $p_X(\mathbf{x})$ hinges on the expressivity of the transformation f_θ, recent works have focused on developing more flexible parameterizations of f_θ. In particular, both non-linear and linear layers have demonstrated promise.

Non-linear coupling layers. Coupling layers (Dinh et al., 2014; 2016) are a powerful class of invertible non-linear layers. The coupling layer splits the input $\mathbf{x}$ into two components: $\mathbf{x}_a$ and $\mathbf{x}_b$. Then, it applies an identity map to $\mathbf{x}_a$ and transforms $\mathbf{x}_b$ using a learnable affine transform (with shift and scale parameters s_θ and b_θ) that depend on $\mathbf{x}_a$. The output of this layer $\mathbf{y}$ is obtained by concatenating these two

intermediate quantities:

$$\mathbf{z}_a = \mathbf{x}_a; \; \mathbf{z}_b = \mathbf{x}_b \odot s_\theta(\mathbf{x}_a) + b_\theta(\mathbf{x}_a)$$
$$\mathbf{y} = \text{concat}(\mathbf{z}_a, \mathbf{z}_b)$$

Due to its simplicity and efficiency, the coupling layer has become a fundamental building block for most state-of-the-art flow model architectures (Chen et al., 2020; Ma et al., 2019; Ho et al., 2019). However, their effectiveness depends heavily on the way in which the input $\mathbf{x}$ is partitioned. Recent works have shown that *linear layers* can learn an improved partitioning scheme, thereby boosting the performance of downstream coupling layers when used together (Kingma & Dhariwal, 2018).

Invertible linear layers. Linear layers, such as invertible 1×1 convolutions, were designed to increase the effectiveness of coupling layers when paired together. Specifically, they *learn* a more general partitioning of the input than the naive splitting as done in conventional coupling layers (Kingma & Dhariwal, 2018). Given an input with channel size c, we denote the learnable parameter (*i.e.*, the filter of the 1×1 convolution) as $\mathbf{W} \in \mathbb{R}^{c \times c}$. To compute the Jacobian determinant efficiently, Kingma & Dhariwal use LU decomposition and parameterize $\mathbf{W}$ as:

$$\mathbf{W} = \mathbf{P}\mathbf{L}(\mathbf{U} + \text{diag}(\mathbf{s})), \qquad (2)$$

where $\mathbf{P}$ is a pre-specified orthogonal matrix, $\mathbf{L}$ is a lower triangular matrix with ones on the diagonal, $\mathbf{U}$ is an upper triangular matrix with zeros on the diagonal, and $\mathbf{s}$ is a c-dimensional vector (Kingma & Dhariwal, 2018). This particular structure in the matrix decomposition allows for the Jacobian determinant to be computed in $\mathcal{O}(c)$, rather than $\mathcal{O}(c^3)$. Other invertible linear layers, such as the Emerging convolution and the Woodbury transformation (Hoogeboom et al., 2019; Lu & Huang, 2020), leverage similar types of matrix structures such as sparsity to improve the performance of coupling layers without sacrificing efficiency.

2.2. Butterfly Layers for Efficient Structured Transforms

The butterfly layer is a special family of linear layers that can be represented as a product of K sparse matrices called *butterfly factors* (Parker, 1995; Dao et al., 2019; 2020). The butterfly factor has a particular structure that requires the specification of two parameters: the level $i \in [K]$ and the factor dimension D. We assume that D is a power of 2 for ease of the technical exposition.

Level-one butterfly factor. A level-one D-dimensional butterfly factor $\mathbf{B}(1, D)$ is a $D \times D$ sparse matrix. Its only non-zero entries are the diagonals of the four $D/2 \times D/2$ sub-matrices obtained by partitioning the matrix in half (Dao et al., 2019), as shown in the left panel in Figure 1.

Level-i butterfly factor. More generally, a level-i D-dimensional butterfly factor $\mathbf{B}(i, D)$ is a $D \times D$ block diagonal matrix (*i.e.*, any off-diagonal block is a zero matrix) with block size $D/2^{i-1} \times D/2^{i-1}$. Each of the diagonal blocks is a $(D/2^{i-1})$-dimensional level-one butterfly factor. Therefore, a level-i D-dimensional butterfly factor has the form:

$$\mathbf{B}(i, D) = \begin{bmatrix} \mathbf{B}_1(1, D/2^{i-1}), & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2(1, D/2^{i-1}) & ... & \mathbf{0} \\ & & ... & \\ \mathbf{0} & \mathbf{0} & ... & \mathbf{B}_{2^{i-1}}(1, D/2^{i-1}) \end{bmatrix}$$

where $\mathbf{B}_j(1, D/2^{i-1})$ is a level-one $D/2^{i-1}$-dimensional butterfly factor on the jth sub-block of $\mathbf{B}(i, D)$, and $\mathbf{0}$ is the $D/2^{i-1} \times D/2^{i-1}$ zero matrix. We provide an illustrative example for $D = 16$ in Figure 1.
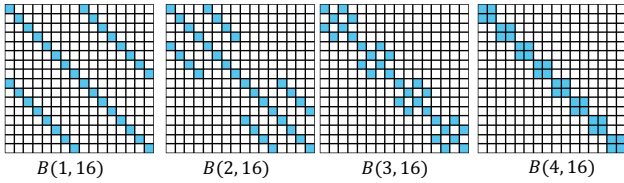

$$B(1,16) \qquad B(2,16) \qquad B(3,16) \qquad B(4,16)$$

Figure 1: Butterfly factors $B(i, D)$ ($D = 16$) at level $i = 1, 2, 3$, and 4. White entries denote zeros and blue entries represent non-zero values. Each butterfly factor is parameterized by the non-zero values in the blue entries.

We can now construct a *butterfly layer* by composing a sequence of butterfly factors as defined below.

Definition 2.1 (Butterfly layer). Given a D-dimensional input $\mathbf{x}$, a D-*dimensional butterfly layer* is a linear layer $b : \mathbf{x} \rightarrow \mathbf{B}(D)\mathbf{x}$, where $\mathbf{B}(D) = \mathbf{B}(a_1, D)\mathbf{B}(a_2, D)...\mathbf{B}(a_k, D)$ is a product of butterfly factors and $\{a_i\}_{i=1}^k$ is a sequence of integers such that $D \equiv 0 \pmod{2^{a_i}}$.

As an example, a commonly used butterfly layer in the literature (called the butterfly matrix) (Parker, 1995; Dao et al., 2019; 2020) can be constructed as $\mathbf{B}(D) = \mathbf{B}(1, D)\mathbf{B}(2, D)...\mathbf{B}(k, D)$, where k is the largest integer such that $D \equiv 0 \pmod{2^k}$.

2.3. Desirable Properties of Butterfly Layers

Although the construction of a butterfly layer involves a sequence of matrix multiplications, it can be efficiently computed on modern hardware by utilizing the sparsity of butterfly factors (Dao et al., 2019; 2020). The butterfly layers' efficiency belies their expressivity: they can represent a wide variety of structured linear maps including discrete Fourier transforms, permutations, and convolutions (Dao et al., 2020; 2019). In the following section, we port over such advantages into generative models by first proposing a new family of *invertible* butterfly linear layers, which serve as a useful architectural primitive for flow models.

3. Building Invertible Butterfly Layers

Recall that each transformation in a normalizing flow layer must be invertible and have an efficiently-computable Jacobian determinant. We describe how flow layers comprised of butterfly factors satisfy both desiderata.

3.1. Invertible Butterfly Factors for Normalizing Flows

We first demonstrate how to compute the Jacobian determinant of the butterfly factor. Given an input $\mathbf{x} \in \mathbb{R}^D$, we denote the parametrized level-i butterfly factor as $\mathbf{B}_\theta(i, D)$, where θ is the set of learnable parameters (values of the non-zero entries corresponding to the blue entries in Figure 1). Then given the linear transformation $b_i : \mathbb{R}^D \rightarrow \mathbb{R}^D : \mathbf{x} \rightarrow \mathbf{B}_\theta(i, D)\mathbf{x}$, we can see that the Jacobian $J_{b_i}(\mathbf{x}) = \mathbf{B}_\theta(i, D)$. Thus, computing the Jacobian determinant of the mapping b_i is equivalent to computing the Jacobian determinant of the butterfly factor $\mathbf{B}_\theta(i, D)$, which can be done efficiently as in Theorem 3.1.

Theorem 3.1. *The determinant of any D-dimensional butterfly factor can be computed in $\mathcal{O}(D)$.*

Proof sketch. We provide the full proof in Appendix A. Since we can decompose the matrix into diagonal matrices, computing the determinant only involves operations on the diagonal elements. $\square$

Next, we consider the invertibility of the butterfly layer. When $\mathbf{B}_\theta(i, D)$ is non-singular, the transformation $b_i(\mathbf{x})$ is invertible. More formally, we define an invertible butterfly factor as the following:

Definition 3.2 (Invertible butterfly factor). An *invertible* level-i D-dimensional butterfly factor $\mathbf{B}_\theta(i, D)$ is a $D \times D$ *non-singular* level-i D-dimensional butterfly factor.

Additionally, we can see that the inverse transformation of b_i is $b_i^{-1} : \mathbb{R}^D \rightarrow \mathbb{R}^D : \mathbf{x} \rightarrow \mathbf{B}_\theta^{-1}(i, D)\mathbf{x}$, where $\mathbf{B}_\theta^{-1}(i, D)$ is the matrix inverse of $\mathbf{B}_\theta(i, D)$. Thus computing $\mathbf{x} = b_i^{-1}(\mathbf{z})$ only requires the application of the following linear transformation to $\mathbf{z}$:

$$\mathbf{x} = b_i^{-1}(\mathbf{z}) = \mathbf{B}_\theta^{-1}(i, D)\mathbf{z}. \tag{3}$$

We note that although Equation (3) involves a potentially expensive matrix multiplication of a $D \times D$ matrix inverse with a D-dimensional vector, we can efficiently invert $\mathbf{B}_\theta(i, \theta)$ given the following proposition.

Proposition 3.3. *Assuming $\boldsymbol{B}_\theta(i, D)$ is non-singular, the matrix $\boldsymbol{B}_\theta^{-1}(i, D)$ is a D-dimensional level-i butterfly factor that can be computed in $\mathcal{O}(D)$. Given $\boldsymbol{B}_\theta^{-1}(i, D)$, the map $b_i^{-1} : \mathbf{z} \rightarrow \boldsymbol{B}_\theta^{-1}(i, D)\mathbf{z}$ can be computed in $\mathcal{O}(D)$.*

We provide the proof in Appendix A. Proposition 3.3 together with Theorem 3.1 show that butterfly factors can

be made efficiently invertible with tractable Jacobian determinants, making them suitable as building blocks for flow-based generative models.

3.2. Invertible Butterfly Layers

With our invertible butterfly factors in place, we introduce a way to compose them into a more powerful *invertible butterfly layer*. We make this precise in the following definition.

Definition 3.4 (Invertible butterfly layer). An *invertible* butterfly layer b is defined as

$$b = b_{a_1} \circ b_{a_2} \circ ... \circ b_{a_k}, \tag{4}$$

where $b_{a_i} : \mathbf{x} \to \mathbf{B}_\theta(a_i, D)\mathbf{x}$ are *invertible* butterfly factors and $\{a_i\}_{i=1}^k$ is a sequence of integers such that $D \equiv 0 \pmod{2^{a_i}}$.

Definition 3.4 suggests that by virtue of being a composition of invertible butterfly factors, the invertible butterfly layer b inherits some of their nice properties. Specifically, let us consider the Jacobian determinant of b in Equation (4). Using the chain rule:

$$\log |\det J_b(\mathbf{x})| = \sum_{i=1}^k \log |\det J_{b_{a_i}}(\mathbf{x})|. \tag{5}$$

Since each invertible butterfly factor b_{a_i} can be efficiently inverted with a Jacobian determinant that can be computed in $\mathcal{O}(D)$, their composition b is also efficiently invertible with a Jacobian determinant that can be computed in $\mathcal{O}(kD)$.

In addition to their efficiency and ease of invertibility, invertible butterfly layers largely retain the expressiveness of the original butterfly layers (Dao et al., 2019; 2020). As a concrete example, they can represent any permutation matrix.

Proposition 3.5. *Any $D \times D$ permutation matrix (with $D = 2^k$ a power of 2) can be represented by an invertible butterfly layer.*

The proof of Proposition 3.5 follows Dao et al., and we provide more details in Appendix A. Proposition 3.5 shows that invertible butterfly layers can also act as *learnable* permutation layers. This is especially helpful for adding expressivity when our butterfly layers are paired with nonlinear coupling layers that use a fixed partitioning of the input.

3.3. Block-wise Invertible Butterfly Layers

We also introduce a new variant of our invertible butterfly factor called the block-wise butterfly factor. Specifically, given a D-dimensional input $\mathbf{x}$, we partition its entries into D/C groups where each group has C elements (see Figure 3). We assume that C divides D for simplicity.

Definition 3.6 (Block-wise invertible butterfly factor). A level-i, block-size-C, D-dimensional *block-wise invertible butterfly factor* $\mathbf{B}_\theta(i, D, C)$ is a $D \times D$ non-singular block matrix with block size $C \times C$ such that for any $j, \hat{j} \in \{1, ..., C\}$, the $D/C \times D/C$ sub-matrix of $\mathbf{B}_\theta(i, D, C)$ obtained by selecting the $C \cdot l + j$-th rows and the $C \cdot l + \hat{j}$-th columns for $l \in \{0, ..., D/C - 1\}$, is a level-$i$ D/C-dimensional butterfly factor.

Intuitively, the block-wise butterfly factor is a $D \times D$ block matrix whose $C \times C$ blocks satisfy the sparsity pattern of a $\mathbf{B}_\theta(i, D/C)$ butterfly layer. We provide an illustrative example in Figure 2. Unlike the naïve butterfly factor where only two entries per row are allowed to be non-zero (see Figure 1), this modification allows for at most $2C$ non-zero entries per row.
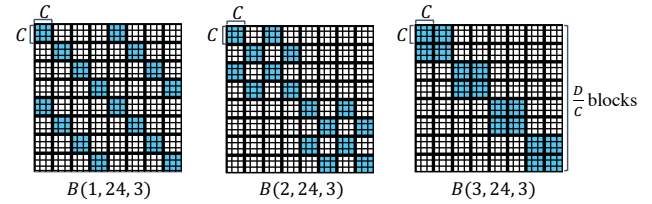

Figure 2: Block-wise invertible butterfly factors $B(i, D, C)$ ($D = 24, C = 3$) at levels $i = 1, 2, 3$. White entries denote zeros and blue entries denote non-zeros. A block-wise invertible butterfly factor can have $2C$ non-zero entries per row and is more expressive than the naïve butterfly factor (Section 3.2), which can only have 2 non-zero entries per row.

Constructing block-wise invertible layers. Similar to how an invertible butterfly layer is constructed using invertible butterfly factors, a block-wise invertible butterfly *layer* is constructed by composing a series of block-wise invertible butterfly *factors*. The block-wise invertible butterfly layer not only improves the flexibility of our invertible butterfly layers, but also reveals interesting connections to previous methods as in the following observations.

Observation 1. *When $C = 1$, the block-wise invertible butterfly layer reduces to the invertible butterfly layer discussed in Section 3.1.*

In fact, the block-wise butterfly layer generalizes commonly used invertible linear layers such as the 1x1 convolution (Kingma & Dhariwal, 2018).

Observation 2. *When C is set to the input's channel size, the block-wise invertible butterfly layer recovers the invertible 1x1 convolution by setting non-diagonal blocks to be zero and using tied weights for diagonal blocks. This is the byproduct of grouping the input entries by channels.*

Additionally, the following observation shows that allowing

the weights of the block-wise invertible butterfly layers to be complex numbers confers a significant boost to their representational power.

Observation 3. *The block-wise invertible butterfly layer with weights in $\mathbb{C}$ can be used to represent a subset of the invertible $d \times d$ convolution layer.*

Specifically, butterfly layers with weights in $\mathbb{C}$ can express any $d \times d$ convolution that can be decomposed into a channel-wise mixing (*e.g.* channel-wise matrix multiplication) and a channel-wise convolution (*i.e.* spatial convolution for each channel). This is an extension of a property of complex-valued naïve butterfly matrices, which can represent any 1D periodic convolution (Dao et al., 2019). We provide additional discussion on this point in Appendix A. Although butterfly layers can have weights in both $\mathbb{C}$ and $\mathbb{R}$, we empirically observe that restricting the butterfly layer weights to be in $\mathbb{R}$ yields good performance, and only consider real-valued weights in the rest of the paper.

Computational complexity. There exists a trade-off between flexibility and computational complexity in block-wise butterfly layers—larger values of C correspond to more powerful but (potentially) more computationally expensive models. To address this, we use a more efficient parameterization of the block-wise butterfly factor: each $C \times C$ block is implemented with LU decomposition (Kingma & Dhariwal, 2018), which reduces the complexity of computing each of the D/C Jacobian determinants of the $C \times C$ block from $\mathcal{O}(C^3)$ to $\mathcal{O}(C)$. Then, since the Jacobian determinant of a naïve butterfly layer can be computed in $\mathcal{O}(D)$, the Jacobian determinant of the block-wise butterfly layer can be evaluated in $\mathcal{O}(D)$. Similarly, with LU decomposition for each $C \times C$ block, we can show that the inverse of the block-wise invertible butterfly layer with k block-wise butterfly factors can be computed in $\mathcal{O}(kC^2D)$. This is because the desired computation reduces to inverting a sequence of block-wise butterfly factors.
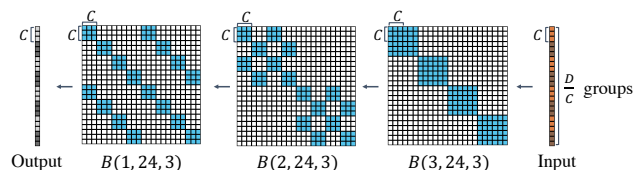

Figure 3: An example of a block-wise invertible butterfly *layer* ($D = 24, C = 3$), which is constructed by composing a sequence of block-wise invertible butterfly *factors*. White entries denote zeros and blue entries denote non-zero values. A 24-dimensional input is partitioned into 8 groups where each group has block-size $C = 3$ before feeding into the block-wise butterfly layer.

4. Generative Modeling with ButterflyFlow

4.1. Architectural Components

In this section, we introduce how to construct the ButterflyFlow model by leveraging our (block-wise) invertible butterfly layers from Section 3.3. We consider the following invertible layers as architectural building blocks that will be combined together for the final model.

Coupling layers. As discussed in Section 2, the coupling layer (Dinh et al., 2014; 2016) is a standard primitive in most state-of-the-art normalizing flow models. We similarly leverage such coupling layers to increase the expressivity of our ButterflyFlow model.

Split and squeeze layers. Dinh et al. split and reshuffle the input dimensions for better mixing. This allows for constructing deeper stacks of coupling layers within the same flow model, increasing its expressive power. We use them in combination with the above mentioned coupling layers to improve their performance, as done in prior works.

Actnorm layers. Actnorm layers are invertible normalization layers that have been developed as an alternative to batch normalization (Ioffe & Szegedy, 2015) in flow-based generative models (Kingma & Dhariwal, 2018). Their parameters are initialized in a data-dependent way (Hoogeboom et al., 2019; Ma et al., 2019). They linearly transform the activations of the input using a scale and translation parameter similar to affine coupling layers, and have been shown to improve training stability.

Invertible Butterfly layers. Given an input $\mathbf{x}$, we expand it into a D-dimensional vector before feeding it into the block-wise butterfly layer as in Figure 3. Each layer's block-size C and grouping mechanism are specific to each particular data type. In the case of RGB images, the block-wise butterfly layers use $C = 3$ and group together RGB values of the same pixels (*i.e.*, cells of the same colors in the input vector shown in Figure 3).

4.2. Building the ButterflyFlow Model

Following recent architectural advancements in flow-based models (Hoogeboom et al., 2019; Ma et al., 2019), ButterflyFlow stacks a series of squeeze, Flow, and split modules together. This results in an architecture of L levels and K Flow modules per level as shown in Figure 4. Within each Flow module, we combine our invertible butterfly layers with Actnorm layers and Coupling layers for added expressivity (Hoogeboom et al., 2019). We elaborate upon our design decisions as well as hyperparameter recommendations for ButterflyFlow in Appendix B.2.

Maximum likelihood training of the ButterflyFlow model proceeds in the same fashion as in conventional flow-based generative models via Equation (1). While maintaining the
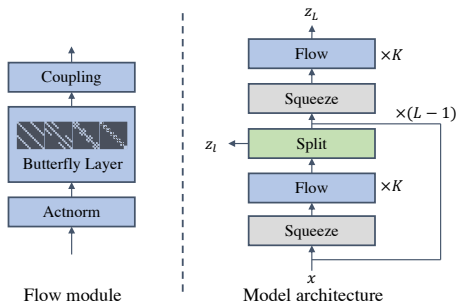
Figure 4: Architecture overview of ButterflyFlow, which shows the transformation of an input $\mathbf{x}$ to an intermediate output $\mathbf{z}_L$. The left diagram details a single Flow module on the right: the input is first passed through an Actnorm layer (Kingma & Dhariwal, 2018) then transformed by a block-wise invertible butterfly layer, whose output is then fed into a coupling layer (Dinh et al., 2016). Each Flow module is repeated K times. Similar to (Kingma & Dhariwal, 2018), we use a hierarchical prior using the split module. We also use the squeeze layer as in (Dinh et al., 2016) to split and reshuffle dimensions for better mixing.

invertibility of the (block-wise) invertible butterfly layers $\mathbf{B}_\theta(i, D, C)$ during training may present a concern, we note that we do not need to enforce any additional constraints to ensure that $\mathbf{B}_\theta(i, D, C)$ remains non-singular. This is because the training loss will become infinitely large when $\det(\mathbf{B}_\theta(i, D, C)) = 0$ (see Equation (1)). In particular, for a butterfly layer, a local non-zero Jacobian determinant (e.g. evaluated at a particular data point) implies a non-zero Jacobian determinant globally—this means that the layer will be invertible. This special property of the butterfly layer is not generally applicable to conventional model architectures.

5. Experiments

In this section, we are interested in investigating three broad questions empirically:

1. How effective is ButterflyFlow at density estimation tasks on standard natural image datasets?

2. How well can ButterflyFlow model datasets with special structures, such as permutation and periodicity?

3. Is ButterflyFlow indeed more efficient than relevant baselines in terms of wall-clock time and/or memory?

We evaluate ButterflyFlow on both synthetic and real datasets that have the corresponding structures of interest. We provide additional details on specific experimental settings and hyperparameter configurations in Appendix B.

5.1. Density estimation on images

We first benchmark our method on standard image datasets to ensure that ButterflyFlow still performs favorably on the usual generative modeling tasks.

Datasets. As in prior works (Hoogeboom et al., 2019; Lu & Huang, 2020), we evaluate our method's performance on MNIST (Deng, 2012), CIFAR-10 (Krizhevsky et al., 2009), and ImageNet-32 × 32 (Deng et al., 2009). We use uniform dequantization and standard data augmentation techniques for CIFAR-10 and ImageNet-32 × 32 during training.

Baselines. We compare ButterflyFlow against several of the most relevant baselines in terms of methods and model architectures: MAF (Papamakarios et al., 2017), Real NVP (Dinh et al., 2016), Glow (Kingma & Dhariwal, 2018), Emerging (Hoogeboom et al., 2019), Woodbury (Lu & Huang, 2020), and i-ResNet (Behrmann et al., 2019) We follow the standard experimental setups and architectural configurations as in prior works.

Results. Quantitative results are shown in Table 1, with visualizations of the generated samples in Figure 5. We find that ButterflyFlow either outperforms or is on par with all relevant baselines. It achieves some improvements on CIFAR-10 and, on ImageNet-32 × 32 and MNIST, ButterflyFlow performs comparably to Glow, Emerging, and Woodbury (with the same Glow backbone). This is possibly due to overparametrization of these large models over image datasets; it is unlikely that adding more linear layers will yield significant improvements. We further examine this claim in Section 5.3 by evaluating the performance of shallower variants of ButterflyFlow on smaller datasets.

Table 1: Density estimation on image datasets. Test set log-likelihood values are in bits per dimension. Lower is better. ButterflyFlow performs favorably relative to all baselines.

	MNIST	CIFAR-10	ImageNet 32×32
MAF (Papamakarios et al., 2017)	1.89	4.31	-
Real NVP (Dinh et al., 2016)	1.06	3.49	4.28
Glow (Kingma & Dhariwal, 2018)	1.05	3.35	4.09
Emerging (Hoogeboom et al., 2019)	1.05	3.34	4.09
Woodbury (Lu & Huang, 2020)	1.05	3.35	4.09
Residual Flows (Chen et al., 2019)	0.97	3.28	4.01
i-DenseNet (Perugachi-Diaz et al., 2021)	-	3.25	3.98
i-ResNet (Behrmann et al., 2019)	1.06	3.45	-
ButterflyFlow (Ours)	1.05	3.33	4.09

5.2. Density estimation on permuted image datasets

In contrast to many existing linear transformations such as 1×1 convolutions, invertible butterfly layers are theoretically guaranteed to be able to represent a large family of complex linear transformations (*e.g.*, any permutation matrix). In this section, we demonstrate empirically that ButterflyFlow is expressive enough to capture special structures in the data such as permutations (as in Proposition 3.5)
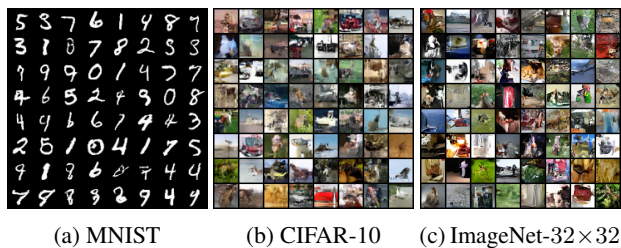
(a) MNIST   (b) CIFAR-10   (c) ImageNet-32×32

Figure 5: Uncurated samples from ButterflyFlow. From left to right: MNIST, CIFAR-10, ImageNet-32 × 32.

by testing our model on image datasets with built-in permutations.

Datasets. We experiment on a permuted version of MNIST, CIFAR-10, and ImageNet-32 × 32 and generate a dataset-wide random permutation matrix. The same permutation matrix is used to permute all images from the same dataset.

Baselines. We compare with Glow (Kingma & Dhariwal, 2018), Emerging (Hoogeboom et al., 2019), and Woodbury (Lu & Huang, 2020), which share the same architectural backbone as ButterflyFlow and similarly exploit spatial locality and permutation structures.

Results. We test the hypothesis that butterfly layers are more effective at capturing the structure in permuted images as compared to baselines. Intuitively, this is because our butterfly layer is a learnable permutation layer that can capture permutation structure present in the data (Proposition 3.5). The rest of the flow model can then learn the appropriate structure specific to the image dataset itself. As shown in Table 2, we find that ButterflyFlow outperforms all other methods. Specifically, our method achieves significantly lower likelihoods as computed by bits per dimension (BPD) on CIFAR-10 and ImageNet-32 × 32. The performance gap is noticeably closer for MNIST, and we show some visualizations of the generated images (permuted back) in Figure 6. All our baselines are able to reasonably model permuted MNIST, likely due to the large modeling capacity of the Glow-based architecture on lower-dimensional datasets such as MNIST. Thus adding butterfly layers to specifically model permutation in this setting only yields marginal improvements.

Table 2: Density estimation on image datasets with permutations. Test set log-likelihood values are reported in bits per dimension. Lower is better. ButterflyFlow outperforms all relevant baselines.

	MNIST	CIFAR-10	ImageNet 32×32
Glow (Kingma & Dhariwal, 2018)	1.44	5.48	6.29
Emerging (Hoogeboom et al., 2019)	1.43	5.41	6.25
Woodbury (Lu & Huang, 2020)	1.43	5.41	6.26
ButterflyFlow (Ours)	**1.42**	**5.11**	**6.18**

5.3. Density estimation on structured datasets

Many real-world datasets often exhibit (unknown) special types of structures such as permutation and periodicity. Therefore, in addition to modeling images with synthetic permutations, we also showcase a set of experiments where ButterflyFlow can be used to model real-world datasets with periodic structures. In particular, we experiment with galaxy images (Ackermann et al., 2018; Hoogeboom et al., 2019) and the MIMIC-III patient records dataset (Johnson et al., 2016) of intensive care units (ICU).

Galaxy images. The galaxy dataset is comprised of 5000 images for both train and test sets, and exhibits periodicity as the images are "continuous"—they represent snapshots of a continuum in space, rather than individual images. As shown in Table 3, we find that ButterflyFlow outperforms all relevant baselines, achieving a BPD improvement of up to 0.07. We also visualize 100 generated images with 100 test set examples in Figure 7. This result provides further evidence that our invertible butterfly layers excel at capturing naturally-occuring structure in real-world data.

Table 3: Comparison of 1 × 1 convolutions (Glow), Emerging convolution, Woodbury flows, and ButterflyFlow on the galaxy images dataset. Test set log-likelihood values are reported in bits per dimension. Lower is better. ButterflyFlow outperforms all relevant baselines.

	Galaxy
1 × 1 (Glow) (Kingma & Dhariwal, 2018)	2.02
Emerging 3 × 3 (Hoogeboom et al., 2019)	1.98
Periodic (Hoogeboom et al., 2019)	1.98
Woodbury (Lu & Huang, 2020)	2.01
ButterflyFlow (Ours)	**1.95**

MIMIC-III waveform database. MIMIC-III is a large-scale dataset containing approximately 30,000 patients' ICU waveforms. For each patient's waveform, two features are recorded: Photoplethysmography (PPG) and Ambulatory Blood Pressure (ABP). Since each patient's recording is very long, we construct a per-patient dataset according to Appendix C.2 and randomly select 3 distinct patient records for our experiments. We illustrate some example ground-truth waveforms in Figure 8 and highlight its repetitive, periodic structure, which is difficult to capture faithfully with conventional flow-based generative models.

For modeling time series, we compare with Emerging and Periodic convolution baselines (Hoogeboom et al., 2019), as well as Woodbury (Lu & Huang, 2020). All methods use the same Glow-based backbone of the same depth and levels. As shown in Table 4, ButterflyFlow outperforms all baselines by a significant margin. In particular, our approach

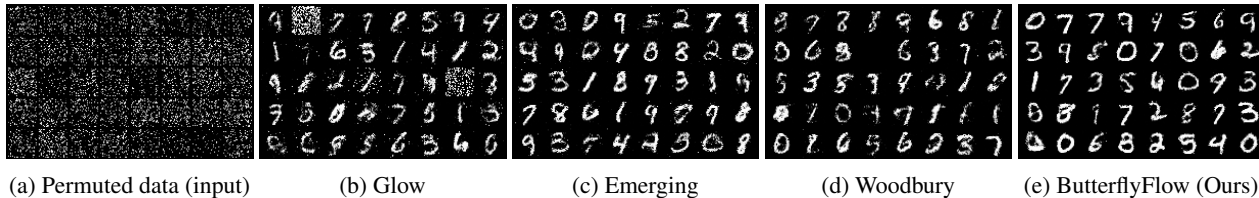| (a) Permuted data (input) | (b) Glow | (c) Emerging | (d) Woodbury | (e) ButterflyFlow (Ours) |

Figure 6: Comparison of (unscrambled) generated samples on permuted MNIST. We observe that the Glow, Emerging, and Woodbury transforms struggle to model the permuted data well (as shown by missing, corrupted, or extremely speckled samples), while ButterflyFlow's learnable permutation layer allows it to better capture the permuted structure.
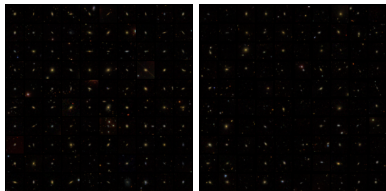

Figure 7: (Left) 10×10 examples from the galaxy images test set. (Right) 10×10 samples from the trained Butterfly-Flow model. Note that the samples look visually similar.
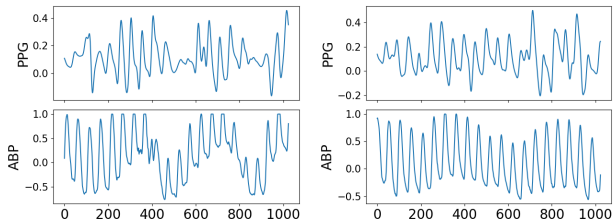

Figure 8: Two independent data points from the processed MIMIC-III patient waveforms. The x-axis indicates the 1024 intervals at which the signal was subsampled and the y-axis indicates the (normalized) recorded values for the PPG and ABP features.

Table 4: Density estimation results on the MIMIC-III task. We report the test set negative log-likelihood per dimension. Lower is better. ButterflyFlow outperforms all other baselines by a significant margin.

	Patient 1	Patient 2	Patient 3	Avg.
Glow (Kingma & Dhariwal, 2018)	-7.21	-5.59	-6.41	-6.40
Emerging (Hoogeboom et al., 2019)	-6.91	-8.48	-7.25	-7.55
Periodic (Hoogeboom et al., 2019)	-8.47	-9.623	-8.73	-8.94
Woodbury (Lu & Huang, 2020)	-11.68	-11.83	-10.91	-11.47
ButterflyFlow (Ours)	**-29.49**	**-27.07**	**-27.20**	**-27.92**

Table 5: Total number of parameters for each model trained on the MIMIC-III dataset. ButterflyFlow is the best performing model (as in Table 4) while using less than half the parameters as compared to baselines.

	# parameters
Glow (Kingma & Dhariwal, 2018)	36,032
Emerging (Hoogeboom et al., 2019)	42,576
Periodic (Hoogeboom et al., 2019)	39,312
Woodbury (Lu & Huang, 2020)	48,576
ButterflyFlow (Ours)	**15,280**

outperforms all competing methods while using *less than half the number of parameters* required by the second-best performing model, as shown in Table 5. Thus, our model is more efficient in terms of space while better modeling the patient data with periodic regularity.

Apart from natural image datasets, we find that our ButterflyFlow model shines when modeling real-world data with special underlying structures. Our empirical evaluations demonstrate that our invertible butterfly layers are able to better capture the global regularity than emerging or periodic convolutions, which rely on local spatial structures.

5.4. Running time

Finally, we benchmark the efficiency of ButterflyFlow, which exploits the sparsity structure of its underlying butterfly factors. We compare the forward and backward

pass through a single butterfly layer with those of Emerging and Periodic convolution layers across 4 settings: forward/inversion time vs. spatial dimension size and forward/inversion time vs. batch size. We present additional details and comparisons in Appendix B.2. As shown in Figure 9, our runtime stays consistently lower than baselines, indicating that our butterfly layer is more computationally efficient.

6. Conclusion

In this work we proposed ButterflyFlow, a novel class of flow-based generative models parameterized by invertible butterfly layers. Drawing inspiration from the literature on learning efficient structured linear transformations, we introduced how butterfly layers more generally can serve as powerful architectural primitives for flow models. We demonstrated that ButterflyFlow not only achieves strong performance on density estimation tasks for standard image datasets, but also better handles real-world data with naturally-occurring structures such as periodicity and per-
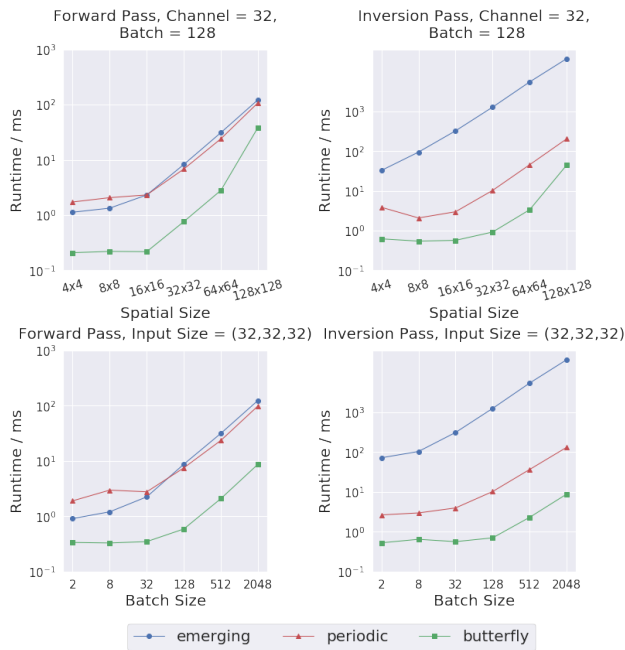
Figure 9: Run-time comparison. The y-axis shows run-time (**ms**) of each setting in log scale. Our run-time stays consistently lower.

mutations relative to existing baselines. A current limitation of our approach is that we must manually specify a particular partitioning of the input for cases where its dimension is not divisible by 2. It would be interesting to generalize the invertible butterfly layer to handle such cases automatically. Additionally, exploring further use cases of ButterflyFlow in applications beyond density estimation would be exciting future work.

7. Acknowledgement

References

Ackermann, S., Schawinski, K., Zhang, C., Weigel, A. K., and Turp, M. D. Using transfer learning to detect galaxy mergers. *Monthly Notices of the Royal Astronomical Society*, 479(1):415–425, 2018.

Behrmann, J., Grathwohl, W., Chen, R. T., Duvenaud, D., and Jacobsen, J.-H. Invertible residual networks. In *International Conference on Machine Learning*, pp. 573–582. PMLR, 2019.

Brock, A., Donahue, J., and Simonyan, K. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Chen, J., Lu, C., Chenli, B., Zhu, J., and Tian, T. Vflow: More expressive generative flows with variational data augmentation. In *International Conference on Machine Learning*, pp. 1660–1669. PMLR, 2020.

Chen, R. T., Behrmann, J., Duvenaud, D. K., and Jacobsen, J.-H. Residual flows for invertible generative modeling. *Advances in Neural Information Processing Systems*, 32, 2019.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Dao, T., Gu, A., Eichhorn, M., Rudra, A., and Ré, C. Learning fast algorithms for linear transforms using butterfly factorizations. In *International conference on machine learning*, pp. 1517–1527. PMLR, 2019.

Dao, T., Sohoni, N. S., Gu, A., Eichhorn, M., Blonder, A., Leszczynski, M., Rudra, A., and Ré, C. Kaleidoscope: An efficient, learnable representation for all structured linear maps. *arXiv preprint arXiv:2012.14966*, 2020.

De Cao, N. and Kipf, T. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

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.

Finzi, M., Izmailov, P., Maddox, W., Kirichenko, P., and Wilson, A. G. Invertible convolutional networks. In *Workshop on Invertible Neural Nets and Normalizing Flows, International Conference on Machine Learning*, volume 2, 2019.

Gómez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

Grcić, M., Grubišić, I., and Šegvić, S. Densely connected normalizing flows. *Advances in Neural Information Processing Systems*, 34:23968–23982, 2021.

Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International Conference on Machine Learning*, pp. 2722–2730. PMLR, 2019.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

Ho, J., Salimans, T., Gritsenko, A., Chan, W., Norouzi, M., and Fleet, D. J. Video diffusion models. *arXiv preprint arXiv:2204.03458*, 2022.

Hoogeboom, E., Van Den Berg, R., and Welling, M. Emerging convolutions for generative normalizing flows. In *International Conference on Machine Learning*, pp. 2771–2780. PMLR, 2019.

Hoogeboom, E., Garcia Satorras, V., Tomczak, J., and Welling, M. The convolution exponential and generalized sylvester flows. *Advances in Neural Information Processing Systems*, 33:18249–18260, 2020.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Johnson, A. E., Pollard, T. J., Shen, L., Li-Wei, H. L., Feng, M., Ghassemi, M., Moody, B., Szolovits, P., Celi, L. A., and Mark, R. G. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3(1):1–9, 2016.

Kadurin, A., Nikolenko, S., Khrabrov, K., Aliper, A., and Zhavoronkov, A. drugan: an advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico. *Molecular pharmaceutics*, 14(9):3098–3104, 2017.

Karami, M., Schuurmans, D., Sohl-Dickstein, J., Dinh, L., and Duckworth, D. Invertible convolutional flow. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/b1f62fa99de9f27a048344d55c5ef7a6-Paper.pdf.

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8110–8119, 2020.

Kingma, D. P. and Dhariwal, P. Glow: Generative flow with invertible 1x1 convolutions. *arXiv preprint arXiv:1807.03039*, 2018.

Kong, Z., Ping, W., Huang, J., Zhao, K., and Catanzaro, B. Diffwave: A versatile diffusion model for audio synthesis. *arXiv preprint arXiv:2009.09761*, 2020.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Kumar, M., Babaeizadeh, M., Erhan, D., Finn, C., Levine, S., Dinh, L., and Kingma, D. Videoflow: A flow-based generative model for video. *arXiv preprint arXiv:1903.01434*, 2(5), 2019.

Lu, Y. and Huang, B. Woodbury transformations for deep generative flows. *arXiv preprint arXiv:2002.12229*, 2020.

Ma, X., Kong, X., Zhang, S., and Hovy, E. Macow: Masked convolutional generative flow. *arXiv preprint arXiv:1902.04208*, 2019.

Oord, A., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., Driessche, G., Lockhart, E., Cobo, L., Stimberg, F., et al. Parallel wavenet: Fast high-fidelity speech synthesis. In *International conference on machine learning*, pp. 3918–3926. PMLR, 2018.

Papamakarios, G., Pavlakou, T., and Murray, I. Masked autoregressive flow for density estimation. *arXiv preprint arXiv:1705.07057*, 2017.

Papamakarios, G., Nalisnick, E., Rezende, D. J., Mohamed, S., and Lakshminarayanan, B. Normalizing flows for probabilistic modeling and inference. *arXiv preprint arXiv:1912.02762*, 2019.

Parker, D. S. Random butterfly transformations with applications in computational linear algebra. 1995.

Perugachi-Diaz, Y., Tomczak, J., and Bhulai, S. Invertible densenets with concatenated lipswish. *Advances in Neural Information Processing Systems*, 34:17246–17257, 2021.

Rezende, D. and Mohamed, S. Variational inference with normalizing flows. In *International conference on machine learning*, pp. 1530–1538. PMLR, 2015.

Slapničar, G., Mlakar, N., and Luštrek, M. Blood pressure estimation from photoplethysmogram using a spectro-temporal deep neural network. *Sensors*, 19 (15), 2019. ISSN 1424-8220. doi: 10.3390/s19153420. URL https://www.mdpi.com/1424-8220/19/15/3420.

Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

Vahdat, A. and Kautz, J. Nvae: A deep hierarchical variational autoencoder. *Advances in Neural Information Processing Systems*, 33:19667–19679, 2020.

A. Proof

In this section, we provide proofs for the main paper.

Lemma A.1. *The determinant of any (invertible) level-one butterfly factor $\mathbf{B}_\theta(1, D)$ can be computed in $\mathcal{O}(D)$.*

Proof. According to the definition of butterfly factor, we can write $\mathbf{B}_\theta(1, D) = \begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$, where $\mathbf{D}_i$ is a $D/2 \times D/2$ diagonal matrix. It is easy to see that $\det(\mathbf{B}_\theta(1, D)) = \det(\mathbf{D}_1\mathbf{D}_4 - \mathbf{D}_2\mathbf{D}_3) = \prod_{j=1}^{D/2} (\mathbf{D}_1[j, j]\mathbf{D}_4[j, j] - \mathbf{D}_2[j, j]\mathbf{D}_3[j, j])$, where $\mathbf{D}_i[j, j]$ denotes the (j, j)-th entry for $\mathbf{D}_i$. The Jacobian determinant of $\mathbf{B}_\theta(1, D)$ can thus be computed in $\mathcal{O}(D)$. $\square$

Theorem 3.1. *The determinant of any D-dimensional butterfly factor can be computed in $\mathcal{O}(D)$.*

Proof. By definition, we have

$$
\mathbf{B}_\theta(i, D) = \begin{bmatrix} \mathbf{B}_1(1, D/2^{i-1}), & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2(1, D/2^{i-1}) & ... & \mathbf{0} \\ & & ... & \\ \mathbf{0} & \mathbf{0} & ... & \mathbf{B}_{2^{i-1}}(1, D/2^{i-1}) \end{bmatrix}
$$

where $\mathbf{B}_j(1, D/2^{i-1})$ is a level-one $D/2^{i-1}$-dimensional (invertible) butterfly factor and $\mathbf{0}$ is the $D/2^{i-1} \times D/2^{i-1}$ zero matrix. Using the property of diagonal block matrices, we have

$$
\det(\mathbf{B}_\theta(i, D)) = \prod_{j=1}^{2^{i-1}} \det(\mathbf{B}_j(1, D/2^{i-1})). \tag{6}
$$

From Lemma A.1, we know computing each $\det(\mathbf{B}_j(1, D/2^{i-1}))$ takes $\mathcal{O}(D/2^{i-1})$, computing $\det(\mathbf{B}_\theta(i, D))$ thus takes $2^{i-1}\mathcal{O}(D/2^{i-1}) = \mathcal{O}(D)$. $\square$

Proposition 3.3. *Assuming $\mathbf{B}_\theta(i, D)$ is non-singular, the matrix $\mathbf{B}_\theta^{-1}(i, D)$ is a D-dimensional level-i butterfly factor that can be computed in $\mathcal{O}(D)$. Given $\mathbf{B}_\theta^{-1}(i, D)$, the map $b_i^{-1} : \mathbf{z} \to \mathbf{B}_\theta^{-1}(i, D)\mathbf{z}$ can be computed in $\mathcal{O}(D)$.*

To prove Proposition 3.3, we first prove Lemma A.2.

Lemma A.2. *Assuming $\mathbf{B}_\theta(1, D)$ is non-singular, then its inverse $\mathbf{B}_\theta^{-1}(1, D)$ is a D-dimensional level-i butterfly factor that can be computed in $\mathcal{O}(D)$ given $\mathbf{B}_\theta(1, D)$.*

Proof. According to the definition of butterfly factor, we can write $\mathbf{B}_\theta(1, D) = \begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$, where $\mathbf{D}_i$ is a $D/2 \times D/2$ diagonal matrix. The inverse of $\mathbf{B}_\theta(1, D)$ can be computed as

$$
\mathbf{B}_\theta^{-1}(1, D) = \begin{bmatrix} -\mathbf{D}_4/(\mathbf{D}_3 \odot \mathbf{D}_2 - \mathbf{D}_4 \odot \mathbf{D}_1) & \mathbf{D}_2/(\mathbf{D}_3 \odot \mathbf{D}_2 - \mathbf{D}_4 \odot \mathbf{D}_1) \\ -\mathbf{D}_3/(\mathbf{D}_1 \odot \mathbf{D}_4 - \mathbf{D}_3 \odot \mathbf{D}_2) & \mathbf{D}_1/(\mathbf{D}_1 \odot \mathbf{D}_4 - \mathbf{D}_3 \odot \mathbf{D}_2) \end{bmatrix}, \tag{7}
$$

where $\mathbf{D}_i \odot \mathbf{D}_j$ are element-wise multiplication of diagonal matrices. Since $\mathbf{D}_i$ is a $D/2 \times D/2$ diagonal matrix, computing $\mathbf{D}_i \odot \mathbf{D}_j$ can be performed in $\mathcal{O}(D/2)$. Thus, evaluating $\mathbf{B}_\theta^{-1}(1, D)$ can be performed in $\mathcal{O}(D)$.

Since each $\mathbf{D}_i$ is a $D/2 \times D/2$ diagonal matrix, each of the block in Equation (7) are also diagonal. Thus, $\mathbf{B}_\theta^{-1}(1, D)$ is a level-one D-dimensional butterfly block by definition. $\square$

We now prove Proposition 3.3.

Proof of Proposition 3.3. According to the definition of $\mathbf{B}_\theta(i, D)$, we can write it as

$$
\mathbf{B}_\theta(i, D) = \begin{bmatrix} \mathbf{B}_1(1, D/2^{i-1}), & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2(1, D/2^{i-1}) & ... & \mathbf{0} \\ & & ... & \\ \mathbf{0} & \mathbf{0} & ... & \mathbf{B}_{2^{i-1}}(1, D/2^{i-1}) \end{bmatrix}
$$

where $\mathbf{B}_j(1, D/2^{i-1})$ is a level-one $D/2^{i-1}$-dimensional (invertible) butterfly factor and $\mathbf{0}$ is the $D/2^{i-1} \times D/2^{i-1}$ zero matrix. Using the properties of diagonal block matrices, it is easy to check

$$\mathbf{B}_\theta^{-1}(i, D) = \begin{bmatrix} \mathbf{B}_1^{-1}(1, D/2^{i-1}), & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2^{-1}(1, D/2^{i-1}) & ... & \mathbf{0} \\ & & ... & \\ \mathbf{0} & \mathbf{0} & ... & \mathbf{B}_{2^{i-1}}^{-1}(1, D/2^{i-1}) \end{bmatrix}.$$

According to Lemma A.2, we have each $\mathbf{B}_j^{-1}(1, D/2^{i-1})$ is a level-one $D/2^{i-1}$-dimensional butterfly factor that can be computed in $\mathcal{O}(D/2^{i-1})$ given $\mathbf{B}_j(1, D/2^{i-1})$. Thus, $\mathbf{B}_\theta^{-1}(i, D)$ is a level-i D-dimensional butterfly factor by definition. It can be computed in $2^{i-1}\mathcal{O}(D/2^{i-1}) = \mathcal{O}(D)$ given $\mathbf{B}_\theta(i, D)$. Since $\mathbf{B}_\theta^{-1}(i, D)$ is a $D \times D$ sparse matrix with only two non-zero entries each row, the map $b_i^{-1} : \mathbf{z} \to \mathbf{B}_\theta^{-1}(i, D)\mathbf{z}$ (*i.e.*, a matrix vector multiplication) can be computed in $2\mathcal{O}(D) = \mathcal{O}(D)$ given $\mathbf{B}_\theta^{-1}(i, D)$. $\square$

Proposition 3.5. *Any $D \times D$ permutation matrix (with $D = 2^k$ a power of 2) can be represented by an invertible butterfly layer.*

Proof. According to Theorem 2. in (Dao et al., 2020), any $D \times D$ permutation matrix $\mathbf{P} \in \mathbb{R}^{D \times D}$ (when $D = 2^k$) can be represented as

$$\mathbf{P} = b_1 \circ b_2 \circ ... \circ b_{k-1} \circ b_k \circ \hat{b}_k \circ \hat{b}_{k-1} \circ ... \circ \hat{b}_1(\mathbf{I}), \tag{8}$$

where $b_i : \mathbf{x} \to \mathbf{B}_\theta(i, D)\mathbf{x}$ and $\hat{b}_i : \mathbf{x} \to \mathbf{B}_{\hat{\theta}}(i, D)\mathbf{x}$ are linear layers obtained by multiplying a learnable level-i D-dimensional butterfly matrix with the input. Since $\mathbf{P}$ is a permutation matrix, it is non-singular, which implies that each b_i, $\bar{b}_i$ and $\hat{b}_i$ must be invertible. Thus, any $D \times D$ permutation matrix can be represented by an *invertible* butterfly layer. We also note that (Dao et al., 2020) does not consider settings where exponentiation of a linear transformation is also invertible (as in our invertible butterfly layers).

$\square$

Lemma A.3 ((Dao et al., 2020)). *Any $D \times D$ ($D = 2^k$) convolution matrix $\mathbf{C}_D$ can be represented as*

$$\mathbf{C}_D = b_1 \circ b_2 \circ ... \circ b_k \circ \hat{b}_{k-1} \circ \hat{b}_{k-2} \circ ... \circ \hat{b}_1, \tag{9}$$

where $b_i : \mathbf{x} \to \boldsymbol{B}_\theta(i, D)\mathbf{x}$ and $\hat{b}_i : \mathbf{x} \to \boldsymbol{B}_{\hat{\theta}}(i, D)\mathbf{x}$ are butterfly layers with weights in $\mathbb{C}$.

Proof. See Lemma J.5. in (Dao et al., 2020). $\square$

Proposition A.4. *Given a single channel 2D input $\mathbf{x} \in \mathbb{R}^{W \times H}$, any 2D convolution layer with kernel size $k \times k$, zero padding and output channel one, can be obtained by multiplying a circulant matrix with the input with padding expanded to a vector.*

Sketch of proof. Given the input $\mathbf{x}$, we apply the zero padding to $\mathbf{x}$ and obtain a padded input $\tilde{\mathbf{x}}$. We then expand $\tilde{\mathbf{x}}$ to a one-dimensional vector. It is easy to show that the 2D convolution can be represented as a circulant matrix multiplied by $\tilde{\mathbf{x}}$ with entries (of the output) corresponding to the paddings removed. $\square$

Observation 3. *The block-wise invertible butterfly layer with weights in $\mathbb{C}$ can be used to represent a subset of the invertible $d \times d$ convolution layer.*

Sketch of proof. Given an input $\mathbf{x} \in \mathbb{R}^{C, W, H}$, an invertible $d \times d$ convolution can be decomposed into two steps: (1) mix the channel information for each (w, h) pair, $w \in [W]$ and $h \in [H]$, by performing an invertible $C \times C$ matrix multiplication with a C-dimensional vector $\mathbf{x}[:, w, h]$, and (2) perform single channel $d \times d$ convolution for each of the C inputs $\mathbf{x}[i, :, :]$, $i = 1, ..., C$, independently. As we showed previously, each of the single channel $d \times d$ convolution can be performed by using circulant matrix, vector multiplication. For input whose size after padding is not a power of 2, we can always pad extra zeros so that the input after padding has size of power of 2. We can remove the entries corresponding to the paddings in the output to recover the correct output. Now, observe that each $C \times C$ matrix block in the block-wise butterfly matrix exactly corresponds to (1) and by Lemma A.3, any circulant matrix with size a power of 2 can be represented using naive butterfly layers. Then the $D/C \times D/C$ block matrix in block-wise butterfly factors (seeing each $C \times C$ as a whole) corresponds to (2). Thus block-wise invertible butterfly layer with weights in $\mathbb{C}$ can be used to represent a family of the invertible $d \times d$ convolution layer. $\square$

Table 6: Model architecture for various datasets.

	Levels (L)	Steps (K)	Coupling channels	Butterfly levels	Bi-direction?	EMA	Butterfly scheduler γ	Butterfly init
CIFAR-10	3	32	512	1	✗	none	N/A	id
ImageNet-32 × 32	3	32	512	1	✗	none	N/A	id
MNIST	2	20	512	1	✗	none	N/A	id
CIFAR-10,permuted	3	32	512	10	✗	separate	0.99	rot
ImageNet-32 × 32,permuted	3	32	512	10	✓	separate	0.99	rot
MNIST,permuted	2	20	512	[9,8,4]	✓	separate	0.996	id
Galaxy	2	8	512	2	✗	separate	0.996	id
MIMIC-III	2	2	16	10	✗	none	N/A	rot

B. Experiments

B.1. Training details

For all experiments, we use Adam optimizer with $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ for training. We warm up our learning by linearly increasing learning rate from 0 to initial learning rate for 10 iterations, and afterwards exponentially decaying with $\gamma = 0.999997$ per iteration. Training is done on TITAN RTX GPU machines. For some experiments we also employ exponential moving average (EMA) of either the entire model or only the butterfly layers, which we will specify in the next section.

B.2. Model architecture

We here define relevant model architecture hyperparameters. The backbone of the our network follows Glow (Kingma & Dhariwal, 2018) baseline as visualized in Figure 4. Our model uses L levels and K steps, and each butterfly layer is of maximum M levels. We by default choose a list of contiguous integers to parametrize our levels $\{a_i\}_{i=1}^k$, *i.e.*, for a butterfly layer of M levels, $\{a_i\}_{i=1}^k = \{1, 2, \ldots, M\}$. For our butterfly layers we also implement a version specified in Proposition 3.5, which stacks a level-inverted M-level butterfly layer on top of a regular butterfly layer. We indicate this version as "bi-direction" in Table 6. If it is set, our butterfly layer has $2M$ butterfly factors with selected integer set $\{a_i\}_{i=1}^k = \{1, 2, \ldots, M, \ldots, 2, 1\}$. For our models, we also implement different types of parameter EMA for training. When EMA is "none", we use a single Adam optimizer for all parameters. When EMA is indicated as "all", we employ EMA on all model parameters. When EMA is indicated as "separate", we employ EMA only for all of our butterfly layers. During training, we use a separate Adam optimizer of the same hyperparameters and exponential decay scheduler of different γ for butterfly layers than the Glow backbone, and we optimize the Glow backbone based on the EMA output of butterfly layers.

We also explore different initialization types for our butterfly layers. If it is "id", we initialize all our butterfly factors to identity matrix. If it is "rot", we initialize our butterfly factors such that the 4 diagonal matrices are element-wise orthogonal. That is, if a butterfly factor is $\begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$ with each sub-matrix being a diagonal matrix, each 2×2 matrix $\begin{bmatrix} \mathbf{D}_1[k,k] & \mathbf{D}_2[k,k] \\ \mathbf{D}_3[k,k] & \mathbf{D}_4[k,k] \end{bmatrix}$ is initialized to a rotation matrix.

Image datasets. For MNIST datasets specifically, we use logit transform with $\lambda = 10^{-6}$ for data preprocessing. For CIFAR-10 and ImageNet-32 × 32, we follow (Hoogeboom et al., 2019) for data preprocessing.

Permuted image datasets. For ImageNet-32 × 32 and CIFAR-10 in particular, we use level-10 butterfly layers and decrease the level by 1 after each Squeeze layer. Since MNIST's image size is $28 \times 28 = 784$, it is not divisible by 2 as required by butterfly layers. Therefore, we choose to partition the space into a concatenation of $512, 256, 16$-dimensional spaces where each can be fed into a $9, 8, 4$-level butterfly factor respectively. Each separate butterfly matrix's level decreases by 1 after each Squeeze layer.

Galaxy images. Model architecture is as shown in Table 6 and we empirically find the using batch size 64 results in better performance.

MIMIC-III waveform database. Since the data has shape (1024, 2), we treat each data point as a 1D image of size 1024 and 2 channels. We then straight-forwardly adapt the Glow backbone for 2D image to process 1D data. For our Emerging and Periodic baselines, we use filter size of 51 since we empirically found that using the default value filter size of 3 fails in learning a reasonable density estimator. For all our model we also use learning rate of 0.0001 because we observed that

higher learning rate results in unstable loss curves. For our butterfly matrix, to reduce the number of learnable parameters, we also share parameters for each diagonal matrix, *i.e.*, if a butterfly factor is $\begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$ with each sub-matrix being a diagonal matrix, the diagonal elements in each $\mathbf{D}_k$ are the same. This sharing holds for each primitive diagonal matrix in a butterfly factor.

Running time. We define specifically what is forward pass and inversion pass for each layer tested. In PyTorch's language, by forward pass we mean applying the tested layer *and* computing the log determinant of its Jacobian under `requires_grad` mode. By inversion pass we mean applying the inverse of the tested layer under `no_grad` mode.

All testing is done on a TITAN XP GPU. For each tensor tested, e.g. of size $3 \times 32 \times 32$, we flatten it into a vector before applying butterfly matrix in our CUDA implementation. We use level-10 butterfly layer by default, and for tensors of smaller sizes, we use the maximum possible level to construct our butterfly layer. For example, a tensor of size $1 \times 4 \times 4$ allows for a level-4 butterfly layer. For tensors of large sizes, e.g. $3 \times 128 \times 128$, which allows for butterfly layers with more than 10 levels, we stop at level 10 because it is the maximum number we use in all our other experiments.
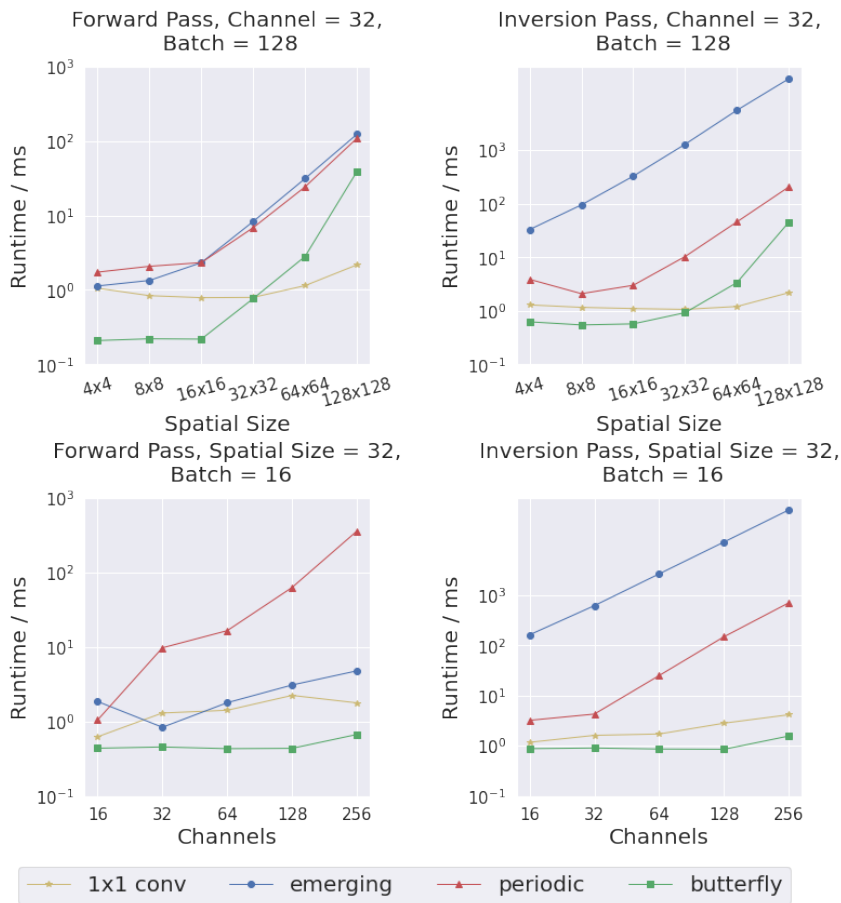
Here we also present additional comparisons with 1×1 convolution.

Figure 10: Comparisons with 1×1 convolution added.

Shown in Figure 10, 1×1 convolution scales better than our model on large images because the operation can be parallelized across all spatial locations. We note that our model is faster on smaller images (first row of Figure 10) and performs comparably with 1×1 convolution at spatial size 32×32. Nevertheless, 1×1 convolution does not scale well with increasing channel size primarily because calculation of its determinant is cubic with respect to channel size. For fair comparison, with fix spatial size at 32×32 and vary channel size (second row of Figure 10). We find that our model outperforms all baselines for runtime vs. channel size.

C. Datasets

C.1. Permuted image datasets

For each of CIFAR-10, ImageNet-32 × 32, and MNIST, we generate a random permutation matrix and preprocess the images in each dataset using the same dataset-wise permutation matrix. visualizations are done by first generating from the model and permute back using the ground-truth matrix.

C.2. MIMIC-III waveform database

MIMIC-III is a large-scale dataset containing approximately 30,000 patients' ICU waveforms. Each patient's record contains a time series of periodic measurements, which is a quasi-continuous recording of the patient's vital signals over their entire stay at the hospital (sometimes days and usually weeks). For this dataset in particular, two feature waveforms are recorded by bedside monitors: Photoplethysmography (PPG) and Ambulatory Blood Pressure (ABP) waveforms.

Due to the extremely long samples per patient, we built per-patient datasets by cutting each waveform sequence into chunks of length 1024. As a concrete example, we can build a dataset of 10,000 data points for a patient with 10.24M sampled intervals. Within this patient's recording, we then have 10,000 data points of dimension (1024, 2) where each dimension corresponds to PPG and ABP features in time. The data points are additionally normalized to $[-1, 1]$ before training. Patient 1, 2, 3 corresponds to patient ID 3000063, 3000393, 3000397, respectively. More details about the dataset is available at https://physionet.org/content/mimic3wdb/1.0/. We also preprocess our data according to (Slapničar et al., 2019) with this Github page https://github.com/gslapnicar/bp-estimation-mimic3, which performs necessary filtering for noise removal and anomaly removal.