Generative Flows with Matrix Exponential

Changyi Xiao [1]   **Ligang Liu** [1]

Abstract

Generative flows models enjoy the properties of tractable exact likelihood and efficient sampling, which are composed of a sequence of invertible functions. In this paper, we incorporate matrix exponential into generative flows. Matrix exponential is a map from matrices to invertible matrices, this property is suitable for generative flows. Based on matrix exponential, we propose matrix exponential coupling layers that are a general case of affine coupling layers and matrix exponential invertible 1×1 convolutions that do not collapse during training. And we modify the networks architecture to make training stable and significantly speed up the training process. Our experiments show that our model achieves great performance on density estimation amongst generative flows models.

1. Introduction

Generative models aim to learn a probability distribution given data sampled from that distribution, in contrast with discriminative models, which do not require a large amount of annotations. A number of models have been proposed including generative adversarial networks (GANs) (Goodfellow et al., 2014), variational autoencoders (VAEs) (Kingma & Welling, 2013; Rezende et al., 2014), autoregressive models (Van Oord et al., 2016), and generative flows models (Dinh et al., 2014; 2016; Rezende & Mohamed, 2015). Generative flows models transform a simple probability distribution into a complex probability distribution through a sequence of invertible functions. They gain popularity recently due to exact density estimation and efficient sampling. In applications, they have been used for density estimation (Dinh et al., 2016), data generation (Kingma & Dhariwal, 2018) and reinforcement learning (Ward et al., 2019).

How to design invertible functions is the core of generative

flows models. There are two principles that should be followed. First, the Jacobian determinant should be computed efficiently. Second, the inverse function should be tractable. Dinh et al. (2014) proposed coupling layers, they first applied generative flows models into density estimation. Dinh et al. (2016) extended the work with more expressive invertible functions and improved the architecture of generative flows models. Kingma & Dhariwal (2018) proposed Glow: generative flow with invertible 1×1 convolutions, which significantly improved the performance of generative flows models on density estimation and showed that generative flows models are capable of realistic synthesis. These flows all have easy Jacobian determinant and inverse.

However, generative flows models have not achieved the same performance on density estimation as state-of-the-art autoregressive models. In order to ensure that the function is invertible and effectively compute the Jacobian determinant, generative flows models suffer from two issues. First, due to the constraints of network, the network is not as expressive as that of GANs. Most of the flows constrain the Jacobian to a triangular matrix, which influences the effectiveness of the network. They apply element-wise transformation, parametrized by part of dimensions. Dinh et al. (2014) proposed additive transformations, then Dinh et al. (2016) presented affine transformations. These transformations are very simple invertible transformations. In (Huang et al., 2018; Ho et al., 2019), they used invertible networks instead of affine transformations, but which still are univariate invertible transformations. Using univariate transformations also hurts the performance of generative flows models. Designing univariate invertible transformations is relatively simple compared with multivariate invertible transformations. Second, the dimension of latent space of generative flows models is same as the input space, which makes the network pretty large for high-dimensional data. Dinh et al. (2016) proposed a multiscale architecture to alleviate this problem, which gradually factors out a part of the total dimensions at regular intervals.

In this paper, we combine matrix exponential with generative flows to propose a new flow called matrix exponential flows. Matrix exponential can be seen as a map from matrices to invertible matrices, this property is suitable for constructing invertible transformations. Meanwhile, matrix exponential has other properties that are also helpful for

generative flows. Based on matrix exponential, we propose matrix exponential coupling layers to enhance the expressiveness of networks, which can be seen as multivariate affine coupling layers. Training generative flows models often takes a long time to converge, especially for large-scale datasets. One reason is that training generative flows models is not stable, which prevents us from being able to use a larger learning rate. As standard convolutions may collapse during training, we propose a stable version of invertible 1×1 convolutions. And we also improve the coupling layers to make training stable and significantly accelerate the training process. The code for our model is available at https://github.com/changyi7231/MEF.

The main contributions of this paper are listed below:

1. We incorporate matrix exponential into neural networks

2. We propose matrix exponential coupling layers, which are a generalization of affine coupling layers.

3. We propose matrix exponential invertible 1×1 convolutions, which are more stable and efficient than standard convolutions.

4. We modify the networks architecture to make training stable and fast.

2. Background

2.1. Change of variables formula

Let $X \in R^n$ be a random variable with an unknown probability density function $p_X(x)$ and $Z \in R^n$ be a random variable with a known and tractable probability density function $p_Z(z)$, generative flows model is an unsupervised model for density estimation defined as an invertible function $z = f(x)$ transforms X into Z. The relationship between $p_X(x)$ and $p_Z(z)$ follows

$$\log p_X(x) = \log p_Z(z) + \log \left| \det(\frac{\partial f(x)}{\partial x}) \right| \quad (1)$$

where $\frac{\partial f(x)}{\partial x}$ is the Jacobian of f evaluated at x. Note that a composition of invertible function remains invertible, let the invertible functions f be composed of K invertible functions: $f_K \circ f_{K-1} \circ \cdots \circ f_1$. The log-likelihood of x can be written as

$$\log p_X(x) = \log p_Z(z) + \sum_{i=1}^{K} \log \left| \det(\frac{\partial h_i}{\partial h_{i-1}}) \right| \quad (2)$$

where $h_i = f_i \circ f_{i-1} \circ \cdots \circ f_1(h_0), h_0 = x$
The choice of f should satisfy two conditions in order to be practical. First, computing the Jacobian determinant should be efficient. In general, the time complexity of computing

Jacobian determinant is $\mathcal{O}(n^3)$. Many works design different invertible functions such that the Jacobian determinant is tractable. Most of them constrain the Jacobian to a triangular matrix, which reduces the computation from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$. Second, in order to draw samples from $p_X(x)$, the inverse function of $f : x = f^{-1}(z)$ should be tractable. Since the generative process is the reverse process of inference process, generative flows models only use a single network. Generative flows models attain the capabilities of both efficient density estimation and sampling.

2.2. Generative Flows

Generative flows models are constructed by a sequence of invertible functions, often parametrized by deep learning layers. Based on the two conditions mentioned in Section 2.1, many generative flows have been proposed. We list several of them that are related to our model. See Table 1 for an overview of these generative flows, and a description is as follows:

Affine coupling layers (Dinh et al., 2016) partition the input into two parts. The first part of dimensions remains unchanged, and the second part of dimensions is mapped with an affine transformation, parametrized by the first part.

Since coupling layers only change half of dimensions, we need to shuffle the dimensions after every coupling layer. Dinh et al. (2014) simply reversed the dimensions, Dinh et al. (2016) suggested randomly shuffling the dimensions. Since the operations are fixed during training, they may be limited in flexibility. Kingma & Dhariwal (2018) generalized the shuffle operations to invertible 1×1 convolutions, which are more flexible and can be learned during training.

Actnorm layers (Kingma & Dhariwal, 2018) are layers to improve training stability and performance. They perform an affine transformation of the activations using a scale and bias parameter per channel. They are data dependent, initialized such that the distribution of activations per-channel has zero mean and unit variance given an initial mini-batch of data.

3. Matrix Exponential

In generative flows models, the function f is implemented as a sequence of invertible functions, which can be parametrized by a neural network $f = W_m \phi(W_{m-1} \phi(W_{m-2} \cdots \phi(W_1 x)))$, where W_i, $1 \leq i \leq m$, is the weight matrix and ϕ is activation function. To ensure that f is invertible, we can let W_i be an invertible matrix and ϕ be a strictly monotone function. But it is difficult to ensure that W_i is invertible during training, and computing the determinant of W is $\mathcal{O}(n^3)$ in general. One method is to enforce W_i to be a triangular matrix. Its determinant is the product of its diagonal entries and it is invertible as

Table 1. The definition of several related generative flows and our generative flows. These flows all have easy Jacobian determinant and inverse. h, w, c denote the height, width and number of channels. The symbols $\odot, /$ denote element-wise multiplication and division. $\boldsymbol{x}, \boldsymbol{y}$ may denote the tensors with shape $h \times w \times c$.

Generative Flows	Function	Reverse Function	Log-determinant
Actnorm layers	$\forall i, j : \boldsymbol{y}_{i,j,:} = \boldsymbol{s} \odot \boldsymbol{x}_{i,j,:} + \boldsymbol{b}$	$\forall i, j : \boldsymbol{x}_{i,j,:} = (\boldsymbol{y}_{i,j,:} - \boldsymbol{b})/\boldsymbol{s}$	$h \cdot w \cdot \mathrm{sum}(\log|\boldsymbol{s}|)$
Affine coupling layers	$[\boldsymbol{x}_1, \boldsymbol{x}_2] = \boldsymbol{x}$ $\boldsymbol{y}_1 = \boldsymbol{x}_1$ $\boldsymbol{y}_2 = \exp(\boldsymbol{s}(\boldsymbol{x}_1)) \odot \boldsymbol{x}_2 + \boldsymbol{b}(\boldsymbol{x}_1)$ $\boldsymbol{y} = [\boldsymbol{y}_1, \boldsymbol{y}_2]$	$[\boldsymbol{y}_1, \boldsymbol{y}_2] = \boldsymbol{y}$ $\boldsymbol{x}_1 = \boldsymbol{y}_1$ $\boldsymbol{x}_2 = \exp(-\boldsymbol{s}(\boldsymbol{y}_1)) \odot (\boldsymbol{y}_2 - \boldsymbol{b}(\boldsymbol{y}_1))$ $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2]$	$\mathrm{sum}(\boldsymbol{s}(\boldsymbol{x}_1))$
Standard 1×1 convolutions	$\forall i, j : \boldsymbol{y}_{i,j,:} = \boldsymbol{W}\boldsymbol{x}_{i,j,:}$	$\forall i, j : \boldsymbol{x}_{i,j,:} = \boldsymbol{W}^{-1}\boldsymbol{y}_{i,j,:}$	$h \cdot w \cdot \log|\det(\boldsymbol{W})|$
Matrix exp coupling layers See Section 4.1	$[\boldsymbol{x}_1, \boldsymbol{x}_2] = \boldsymbol{x}$ $\boldsymbol{y}_1 = \boldsymbol{x}_1$ $\boldsymbol{y}_2 = e^{\boldsymbol{S}(\boldsymbol{x}_1)}\boldsymbol{x}_2 + \boldsymbol{b}(\boldsymbol{x}_1)$ $\boldsymbol{y} = [\boldsymbol{y}_1, \boldsymbol{y}_2]$	$[\boldsymbol{y}_1, \boldsymbol{y}_2] = \boldsymbol{y}$ $\boldsymbol{x}_1 = \boldsymbol{y}_1$ $\boldsymbol{x}_2 = e^{-\boldsymbol{S}(\boldsymbol{y}_1)}(\boldsymbol{y}_2 - \boldsymbol{b}(\boldsymbol{y}_1))$ $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2]$	$\mathrm{Tr}(\boldsymbol{s}(\boldsymbol{x}_1))$
Matrix exp 1×1 convolutions See Section 4.2	$\forall i, j : \boldsymbol{y}_{i,j,:} = e^{\boldsymbol{W}}\boldsymbol{x}_{i,j,:}$	$\forall i, j : \boldsymbol{x}_{i,j,:} = e^{-\boldsymbol{W}}\boldsymbol{y}_{i,j,:}$	$h \cdot w \cdot \mathrm{Tr}(\boldsymbol{W})$

long as its diagonal entries are non-zero. But triangular matrices are less expressive and computing the inverse of triangular matrices is sequential, not parallel. We propose to replace the weight matrix $\boldsymbol{W}_i$ by the matrix exponential of $\boldsymbol{W}_i$. It can make the networks invertible. And it is more expressive than triangular matrices. Moreover, computing the determinant needs only $\mathcal{O}(n)$ time and computing the inverse matrix is easy and parallel. Before introducing matrix exponential, we first set up some notations. Let $M_n(R)$ be the set of $n \times n$ real matrices, $GL_n(R)$ be the set of $n \times n$ real invertible matrices, $GL_n(R)^+$ be the set of $n \times n$ real invertible matrices with positive determinant, $GL_n(R)^-$ be the set of $n \times n$ real invertible matrices with negative determinant.

3.1. Properties of Matrix Exponential

Matrix exponential is a matrix function whose definition is similar to the exponential function. Matrix exponential has many applications. It can be used to solve systems of linear differential equations. And it plays an important role in the theory of Lie groups, which gives the connection between a matrix Lie algebra and the corresponding Lie group. Matrix exponential of $\boldsymbol{W} \in M_n(R)$ is defined as

$$e^{\boldsymbol{W}} = \sum_{i=0}^{\infty} \frac{\boldsymbol{W}^i}{i!} \tag{3}$$

Matrix exponential has four properties as follows:

1. For any matrix $\boldsymbol{W} \in M_n(R)$, $e^{\boldsymbol{W}}$ is converge, and $e^{\boldsymbol{W}} \in GL_n(R)$, meanwhile $(e^{\boldsymbol{W}})^{-1} = e^{-\boldsymbol{W}}$.

2. $\log \det(e^{\boldsymbol{W}}) = \mathrm{Tr}(\boldsymbol{W})$.

3. For any matrix $\boldsymbol{X} \in GL_n(R)^+$ and $\boldsymbol{X}$ satisfies each Jordan block of $\boldsymbol{X}$ corresponding to a negative eigen-

value occurs an even number of times, then there exists a matrix $\boldsymbol{W} \in M_n(R)$ such that $\boldsymbol{X} = e^{\boldsymbol{W}}$.

4. For any matrix $\boldsymbol{X} \in GL_n(R)^+$, there exists some matrices $\boldsymbol{W}_1, \boldsymbol{W}_2, \ldots, \boldsymbol{W}_n \in M_n(R)$ such that $\boldsymbol{X} = e^{\boldsymbol{W}_1} e^{\boldsymbol{W}_2} \cdots e^{\boldsymbol{W}_n}$.

See (Hall, 2015) for the proofs of property 1,2,4 and (Culver, 1966) for the proof of property 3. From property 1, we see that matrix exponential of $\boldsymbol{W}$ is always converge and invertible. Thus we implement the neural network as $f = e^{\boldsymbol{W}_m}\phi(e^{\boldsymbol{W}_{m-1}}\phi(e^{\boldsymbol{W}_{m-2}} \cdots \phi(e^{\boldsymbol{W}_1}\boldsymbol{x})))$, which makes the neural network invertible. Matrix exponential can be seen as a map from $M_n(R)$ to $GL_n(R)$, so we have no need to constrain the weight matrix $\boldsymbol{W}$. The inverse of matrix exponential is also matrix exponential, which can be computed in the same way. From property 2, computing the log-determinant of matrix exponential of $\boldsymbol{W}$ turns into computing the trace of $\boldsymbol{W}$. Computing the determinant of a $n \times n$ matrix is $\mathcal{O}(n^3)$ in general, while computing the trace is $\mathcal{O}(n)$. Property 3 demonstrates the image of matrix exponential. Matrix exponential is not a surjective map from $M_n(R)$ to $GL_n(R)$. Property 2 shows that the determinant of matrix exponential is always positive. $GL_n(R)$ has two connected components: $GL_n(R)^+$ and $GL_n(R)^-$. The image of matrix exponential is a subset of $GL_n(R)^+$. It is reasonable to ensure that the determinant is positive, because once the sign of the determinant changes during training, it may cause the matrix to be singular and numerically unstable. Although the matrix exponential is also not a surjective map to $GL_n(R)^+$, however, it only excludes a few matrices. Property 4 demonstrates any matrix in $GL_n(R)^+$ is a product of n matrix exponentials, so we can get a surjective map to $GL_n(R)^+$. In practice, using n matrix exponentials may be redundant, since the image of matrix exponential is

a rich class of invertible matrices, one or two is enough.

3.2. Compute Matrix Exponential

Matrix exponential is an infinite matrix series. Dozens of methods for computing matrix exponential have been proposed. Moler & Van Loan (2003) showed nineteen ways involving approximation theory, differential equations, the matrix eigenvalues. We propose two methods to incorporate matrix exponential into neural networks. The first method is for low-dimensional data, and the second method is for high-dimensional data.

The first method is to truncate the matrix series of Eq. (3) at index k to approximate the matrix series. Define the finite matrix series as

$$T_k(\boldsymbol{W}) = \sum_{i=0}^{k} \frac{\boldsymbol{W}^i}{i!} \tag{4}$$

There are several papers that study the truncation error of this series, Liou (1966) gave a bound of truncation error

$$\|T_k(\boldsymbol{W}) - e^{\boldsymbol{W}}\|_1 \leq (\frac{\|\boldsymbol{W}\|_1^{k+1}}{(k+1)!})(\frac{1}{1 - \|\boldsymbol{W}\|_1/(k+2)}) \tag{5}$$

where $\|\cdot\|_1$ is the matrix 1-norm. The error bound is affected by $\|\boldsymbol{W}\|_1$, which decreases as $\|\boldsymbol{W}\|_1$ decrease. When $\|\boldsymbol{W}\|_1$ is small, we can choose a small k and need less computation to approximate the infinite series. Fortunately, the value of weight matrices of neural networks is often small such that $\|\boldsymbol{W}\|_1$ is also small. This is a good property that makes incorporating matrix exponential into neural networks practical. Since $e^{\boldsymbol{W}} = (e^{\boldsymbol{W}/2^s})^{2^s}$, we first scale the weight matrix to a smaller value, then compute the matrix exponential and the matrix power. This further reduces the computation. Algorithm 1 shows the process of computing matrix exponential, which is mentioned in Moler & Van Loan (2003). This algorithm costs about $(s+k-1)n^3$ FLOPs, which makes it unable to scale to high-dimensional data.

We propose the second method that combines matrix exponential with neural networks for high-dimensional data. Instead of directly parameterizing the weight matrix $\boldsymbol{W}$, we propose a low-rank parameterization method. Let $\boldsymbol{W} = \boldsymbol{A}_1 \boldsymbol{A}_2$, where $\boldsymbol{A}_1 \in R^{n \times t}$, $\boldsymbol{A}_2 \in R^{t \times n}$, and $\boldsymbol{A}_1, \boldsymbol{A}_2$ are the weight matrices. Substitute $\boldsymbol{W}$ into Eq. (3), then

$$e^{\boldsymbol{W}} = \sum_{i=0}^{\infty} \frac{(\boldsymbol{A}_1 \boldsymbol{A}_2)^i}{i!} \tag{6}$$

Let $\boldsymbol{V} = \boldsymbol{A}_2 \boldsymbol{A}_1$. Considering the associative law of matrix multiplication, we have

$$e^{\boldsymbol{W}} = \boldsymbol{I} + \boldsymbol{A}_1 \sum_{i=0}^{\infty} \frac{\boldsymbol{V}^i}{(i+1)!} \boldsymbol{A}_2 \tag{7}$$

Algorithm 1 Algorithm for computing matrix exponential
Input:
　　Weight matrix: $\boldsymbol{W}$
　　Tolerable error: ϵ
Output:
　　Matrix exponential of weight matrix: $e^{\boldsymbol{W}}$
 1: choose the smallest non-negative integer s such that $\|\boldsymbol{W}\|_1/(2^s) < \frac{1}{2}$
 2: $\boldsymbol{W} := \boldsymbol{W}/(2^s)$
 3: $\boldsymbol{X} := \boldsymbol{I}$
 4: $\boldsymbol{Y} := \boldsymbol{W}$
 5: $k := 2$
 6: **while** $\|\boldsymbol{Y}\|_1 > \epsilon$ **do**
 7: 　　$\boldsymbol{X} := \boldsymbol{X} + \boldsymbol{Y}$
 8: 　　$\boldsymbol{Y} := \boldsymbol{W} \cdot \boldsymbol{Y}/k$
 9: 　　$k := k + 1$
10: **end while**
11: **for** $i = 1$ to s **do**
12: 　　$\boldsymbol{Y} := \boldsymbol{Y} \cdot \boldsymbol{Y}$
13: **end for**
14: **return:** $\boldsymbol{Y}$

We truncate matrix series of $\boldsymbol{V}$ at index k to approximate $e^{\boldsymbol{W}}$. Similar to the truncation error bound of $e^{\boldsymbol{W}}$, the error bound of truncating matrix series of $\boldsymbol{V}$ is given by

$$\| \sum_{i=k+1}^{\infty} \frac{\boldsymbol{V}^i}{(i+1)!}\|_1 \leq (\frac{\|\boldsymbol{V}\|_1^{k+1}}{(k+2)!})(\frac{1}{1 - \|\boldsymbol{V}\|_1/(k+3)}) \tag{8}$$

Computing the matrix series of $\boldsymbol{W}$ turns into computing the matrix series of $\boldsymbol{V}$. Since $\boldsymbol{V} \in R^{t \times t}$, computing the matrix series of $\boldsymbol{V}$ costs $\mathcal{O}(t^3)$. The rank of matrix $\boldsymbol{W}$ is less than or equal to t. We can choose a small t to reduce the computation, but which will hurt the expressiveness. It is a balance between expressiveness and computation. Computing the matrix series of $\boldsymbol{V}$ is analogous to Algorithm 1, just set the scale coefficient $s := 0$ and modify the line 4 to $\boldsymbol{Y} := \boldsymbol{W}/2$ and line 5 to $k := 3$.

4. Matrix Exponential Flows

We utilize matrix exponential to propose a new flow called matrix exponential flows (MEF). In Section 4.1, we combine matrix exponential with coupling layers to present our matrix exponential coupling layers. In Section 4.2, we provide matrix exponential invertible 1×1 convolutions which are stable during training. Figure 1 illustrates a detailed overview of the architecture.

4.1. Coupling layers

Dinh et al. (2016) proposed affine coupling layers that split the n dimensional input $\boldsymbol{x}$ into two parts $(\boldsymbol{x}_{1:d}, \boldsymbol{x}_{d+1,n})$, the

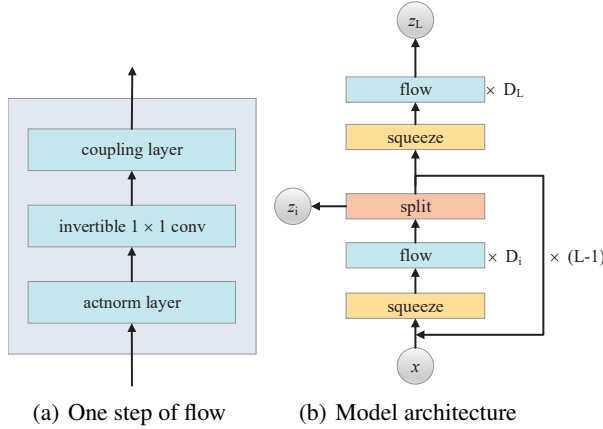(a) One step of flow   (b) Model architecture

Figure 1. Overview of the model architecture. Left describes each step of flow, which consists of an actnorm layer that normalizes all activations independently, followed by a matrix exponential invertible 1×1 convolution, followed by a matrix exponential coupling layer. Right shows the multiscale architecture. The squeeze operation rearranges the dimensions by reducing the spatial dimensions by a half and increasing the channel number by four. The split operation splits the dimensions into two parts along channel and outputs a part of dimensions. The architecture has L levels and D_i flows for one level.

output y of affine coupling layers follows the equations

$$
\begin{aligned}
\boldsymbol{y}_{1:d} &= \boldsymbol{x}_{1:d} \\
\boldsymbol{y}_{d+1:n} &= \exp(\boldsymbol{s}(\boldsymbol{x}_{1:d})) \odot \boldsymbol{x}_{d+1:n} + \boldsymbol{b}(\boldsymbol{x}_{1:d})
\end{aligned}
\tag{9}
$$

where s and t stand for scale and bias, are functions from $R^d \to R^{n-d}$, $\exp(\boldsymbol{s}(\boldsymbol{x}_{1:d}))$ is the element-wise exponential function of $\boldsymbol{s}(\boldsymbol{x}_{1:d})$, and $\odot$ is the Hadamard product. The first part remains unchanged and the second part is mapped with an element-wise exponential transformation, parametrized by the first part. Note that $\boldsymbol{y}_{d+i}$ is only the function of $\boldsymbol{x}_{1:d}$ and $\boldsymbol{x}_{d+i}$, not the function of $\boldsymbol{x}_{d+j}$, where $1 \le j \le n - d, j \ne i$. Rewrite the affine coupling layers as

$$
\begin{pmatrix} \boldsymbol{y}_{d+1} \\ \vdots \\ \boldsymbol{y}_n \end{pmatrix} = diag(\exp(\boldsymbol{s}(\boldsymbol{x}_{1:d}))) \begin{pmatrix} \boldsymbol{x}_{d+1} \\ \vdots \\ \boldsymbol{x}_n \end{pmatrix} + \boldsymbol{b}(\boldsymbol{x}_{1:d})
\tag{10}
$$

where $diag(\exp(\boldsymbol{s}(\boldsymbol{x}_{1:d})))$ is the diagonal matrix whose diagonal elements correspond to the vector $\exp(\boldsymbol{s}(\boldsymbol{x}_{1:d}))$. As a diagonal matrix is less expressive, we replace the $diag(exp(\boldsymbol{s}(\boldsymbol{x}_{1:d})))$ by matrix exponential $e^{\boldsymbol{S}(\boldsymbol{x}_{1:d})}$, thus

$$
\begin{pmatrix} \boldsymbol{y}_{d+1} \\ \vdots \\ \boldsymbol{y}_n \end{pmatrix} = e^{\boldsymbol{S}(\boldsymbol{x}_{1:d})} \begin{pmatrix} \boldsymbol{x}_{d+1} \\ \vdots \\ \boldsymbol{x}_n \end{pmatrix} + \boldsymbol{b}(\boldsymbol{x}_{1:d})
\tag{11}
$$

where $e^{\boldsymbol{S}(\boldsymbol{x}_{1:d})}$ is the matrix exponential of $\boldsymbol{S}(\boldsymbol{x}_{1:d}) \in R^{n-d \times n-d}$, each element of $\boldsymbol{S}(\boldsymbol{x}_{1:d})$ is a function of $\boldsymbol{x}_{1:d}$.

The first part is still unchanged. This form of layers is more expressive than former affine coupling layers. If $\boldsymbol{S}(\boldsymbol{x}_{1:d}) = diag(\boldsymbol{s}(\boldsymbol{x}_{1:d}))$, then $e^{\boldsymbol{S}(\boldsymbol{x}_{1:d})} = diag(\exp(\boldsymbol{s}(\boldsymbol{x}_{1:d})))$, thus our coupling layers turn into affine coupling layers when the matrix $\boldsymbol{S}(\boldsymbol{x}_{1:d})$ is a diagonal matrix. Affine coupling layers are a special kind of our coupling layers. Matrix exponential of a 1×1 matrix is equal to exponential function, thus our coupling layers can be seen as multivariate affine coupling layers. The Jacobian of our coupling layers is

$$
\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \boldsymbol{I}_d & 0 \\ \frac{\partial \boldsymbol{y}_{d+1:n}}{\partial \boldsymbol{x}_{1:d}} & e^{\boldsymbol{S}(\boldsymbol{x}_{1:d})} \end{pmatrix}
\tag{12}
$$

The Jacobian is a block triangular matrix. Compared with affine coupling layers whose Jacobian is a triangular matrix, our coupling layers extend the Jacobian from a triangular matrix to a block triangular matrix. Its log-determinant is $\log \det(e^{\boldsymbol{S}(\boldsymbol{x}_{1:d})}) = \mathrm{Tr}(\boldsymbol{S}(\boldsymbol{x}_{1:d}))$, which can be computed fast. The inverse function of our layers is:

$$
\begin{aligned}
\boldsymbol{x}_{1:d} &= \boldsymbol{y}_{1:d} \\
\boldsymbol{x}_{d+1:n} &= e^{-\boldsymbol{S}(\boldsymbol{y}_{1:d})}(\boldsymbol{y}_{d+1:n} - \boldsymbol{b}(\boldsymbol{y}_{1:d}))
\end{aligned}
\tag{13}
$$

For 2D image data $\boldsymbol{x}$ with shape $h \times w \times 2c$, where $h, w, 2c$ denote the height, width and number of channels, split $\boldsymbol{x}$ along channel into two parts $(\boldsymbol{x}^1, \boldsymbol{x}^2)$. The corresponding output is $(\boldsymbol{y}^1, \boldsymbol{y}^2)$. Eq. (11) shows that each element of $\boldsymbol{S}(\boldsymbol{x}^1)$ is a function of $\boldsymbol{x}^1$. So the output layer of $\boldsymbol{S}(\boldsymbol{x}^1)$ has $(h \cdot w \cdot c)^2$ units, which leads to too many units in the output layer of $\boldsymbol{S}(\boldsymbol{x}^1)$. In order to reduce the number of units, we propose a location-dependent type of coupling layers. The output $\boldsymbol{y}^2_{i,j,k}$ is not the function of all elements of $\boldsymbol{x}^2$, but the function of $\boldsymbol{x}^2_{i,j,l}$ with the same height and width index, where $1 \le l \le c$. Our coupling layers turn into

$$
\begin{aligned}
\boldsymbol{y}^1 &= \boldsymbol{x}^1 \\
\forall i, j, \boldsymbol{y}^2_{i,j,:} &= e^{\boldsymbol{S}(\boldsymbol{x}^1)_{i,j,:,:}} \boldsymbol{x}^2_{i,j,:} + \boldsymbol{b}(\boldsymbol{x}^1)_{i,j,:}
\end{aligned}
\tag{14}
$$

where $\boldsymbol{S}(\boldsymbol{x}^1) \in R^{h \times w \times c \times c}, \boldsymbol{b}(\boldsymbol{x}^1) \in R^{h \times w \times c}$. So the output layer of $\boldsymbol{S}(\boldsymbol{x}^1)$ only has $h \cdot w \cdot c \cdot c$ units. For very large c, the output layer may still have too many units. Thus for very large c, let the output layer of $\boldsymbol{S}(\boldsymbol{x}_1)$ has $h \times w \times c \times 2t$ units, where t can be chosen such that $t \ll c$. Split the output layer into two parts: $\boldsymbol{A}_1$ with shape $h \times w \times c \times t$ and $\boldsymbol{A}_2$ with shape $h \times w \times t \times c$. Our coupling layers follow

$$
\begin{aligned}
\boldsymbol{y}^1 &= \boldsymbol{x}^1 \\
\forall i, j, \boldsymbol{y}^2_{i,j,:} &= e^{\boldsymbol{A}_1(\boldsymbol{x}^1)_{i,j,:,:} \boldsymbol{A}_2(\boldsymbol{x}^1)_{i,j,:,:}} \boldsymbol{x}^2_{i,j,:} + \boldsymbol{b}(\boldsymbol{x}^1)_{i,j,:}
\end{aligned}
\tag{15}
$$

And use Eq. (7) to compute matrix exponential. This makes the model scalable, we can select a proper t to balance the model complexity and computation. Since $e^{\boldsymbol{0}} = \boldsymbol{I}$, we initialize the output layer of $\boldsymbol{S}(\boldsymbol{x}_1)$ with zeros such that each

Table 2. Density estimation performance on CIFAR-10 and ImageNet 32×32, ImageNet 64×64 datasets. Results are reported in bits/dim (negative $\log_2$ likelihood). In brackets are models that use variational dequantization (Ho et al., 2019).

Model	CIFAR10	ImageNet32 × 32	ImageNet64 × 64
RealNVP (Dinh et al., 2016)	3.49	4.28	3.98
Glow (Kingma & Dhariwal, 2018)	3.35	4.09	3.81
Emerging (Hoogeboom et al., 2019)	3.34	4.09	3.81
Flow++ (Ho et al., 2019)	3.29 (3.08)	— (3.86)	— (3.69)
MEF (Ours)	3.32	4.05	3.73

Table 3. Comparison of the number of parameters of Glow, Emerging, Flow++ and MEF

Model	CIFAR10	ImageNet32 × 32	ImageNet64 × 64
Glow (Kingma & Dhariwal, 2018)	44.0M	66.1M	111.1M
Emerging (Hoogeboom et al., 2019)	44.7M	67.1M	67.1M
Flow++ (Ho et al., 2019)	31.4M	169.0M	73.5M
MEF (Ours)	37.7M	37.7M	46.6M

coupling layer initially performs an identity function, this helps training deep networks and reduces the computation of matrix exponential.

4.2. Invertible 1× 1 convolutions

Standard 1×1 convolutions are flexible since the weight matrix $\boldsymbol{W}$ can become any matrix in $M_n(R)$. But they may be numerically unstable during training when the weight matrix is singular. Kingma & Dhariwal (2018) proposed to learn a $\boldsymbol{PLU}$ decomposition and constrained the diagonal element of $\boldsymbol{U}$ non-zero, which makes the convolutions more stable, but their flexibility is limited. In order to solve the stability issues and retain the flexibility of the convolutions, we propose to replace the weight matrix $\boldsymbol{W}$ by the matrix exponential of $\boldsymbol{W}$, the convolutions are implemented as :

$$\forall i, j : \boldsymbol{y}_{i,j,:} = e^{\boldsymbol{W}} \boldsymbol{x}_{i,j,:} \tag{16}$$

In Section 3.1, we demonstrate that $e^{\boldsymbol{W}} \in GL_n(R)^+$, which guarantees the determinant of $e^{\boldsymbol{W}}$ positive. So our matrix exponential convolutions are stable. The log-determinant of Jacobian is $h \cdot w \cdot \text{Tr}(\boldsymbol{W})$, where h, w are height and width. The inverse function is:

$$\forall i, j : \boldsymbol{x}_{i,j,:} = e^{-\boldsymbol{W}} \boldsymbol{y}_{i,j,:} \tag{17}$$

Suppose $\boldsymbol{W}$ is a skew-symmetric matrix, then

$$(e^{\boldsymbol{W}})(e^{\boldsymbol{W}})^T = e^{\boldsymbol{W}+\boldsymbol{W}^T} = e^{\boldsymbol{0}} = \boldsymbol{I} \tag{18}$$

thus matrix exponential of a skew-symmetric matrix is an orthogonal matrix. And the determinant of $e^{\boldsymbol{W}}$ is positive, so $e^{\boldsymbol{W}}$ is a rotation matrix. All $n \times n$ rotation matrices form a special orthogonal group. Special orthogonal group is in the image of matrix exponential (Hall, 2015). We initialize $\boldsymbol{W}$ as a skew-symmetric matrix such that $e^{\boldsymbol{W}}$ is a rotation matrix.

5. Related Work

This work mainly builds upon the ideas proposed in (Dinh et al., 2016; Kingma & Dhariwal, 2018). Generative flows models can roughly be divided into two categories according to the Jacobian. One is the models whose Jacobian is a triangular matrix, which are based on coupling layers proposed in (Dinh et al., 2014; 2016) or autoregressive flows proposed in (Kingma et al., 2016; Papamakarios et al., 2017). Ho et al. (2019); Hoogeboom et al. (2019); Durkan et al. (2019) extended the models with more expressive invertible functions. The other is the models with free-form Jacobian. Behrmann et al. (2019) proposed invertible residual networks and utilized it for density estimation. Chen et al. (2019) further improved the model with a unbiased estimate of the log density. Grathwohl et al. (2018) proposed a continuous-time generative flow with unbiased density estimation.

6. Experiments

In this section, we run several experiments to demonstrate the performance of our model. In Section 6.1, we compare the performance on density estimation with other generative flows models. In Section 6.2, we study the training stability of generative flows models. In Section 6.3, we compare three 1×1 convolutions. In Section 6.4, we analyze the computation of matrix exponential. In Section 6.5 we show samples from our trained models.

6.1. Density Estimation

We evaluate our MEF model on CIFAR10 (Krizhevsky et al., 2009), ImageNet32 and ImageNet64 (Van Oord et al., 2016) datasets and compare log-likelihood with other generative flows models. See Figure 1 for a detailed overview of our architecture. We use a level $L = 3$ and depth

$D_1 = 8, D_2 = 4, D_3 = 2$. Each coupling layer is composed of 8 residual blocks (He et al., 2016) for CIFAR10 and ImageNet32 datasets and 10 residual blocks for ImageNet64 dataset. Each residual block has three convolution layers, where the first layer and the last layer are 3×3 convolution layers, the center layer is 1×1 convolution layer, all with 128 channels. The activation function is ELU (Clevert et al., 2015). The optimization method is Adamax (Kingma & Ba, 2014). All models are trained for 50 epochs with batch size 64. Table 2 shows MEF achieves great performance on the negative log-likelihood scores in bits/dim. Our model performs better than Glow (Kingma & Dhariwal, 2018) and Emerging (Hoogeboom et al., 2019), only worse than Flow++(Ho et al., 2019) with variational dequantization. Table 3 shows the comparison of the number of parameters of Glow, Emerging, Flow++ and MEF. Our model uses a relatively small number of parameters on ImageNet datasets.

6.2. Training Stability

Training generative flows models requires high computing infrastructure due to large computation, especially for large-scale datasets. One reason is that training generative flows models often take a long time to converge. Glow (Kingma & Dhariwal, 2018) was trained for 1800 epochs and Flow++ (Ho et al., 2019) had not fully converged after 400 epochs on CIFAR10 dataset. One reason is that training generative flows models is not stable, which prevents us from being able to use a larger learning rate. In Section 4.2, we explain why standard 1×1 convolutions may collapse during training. We replace the standard convolutions by our matrix exponential convolutions, which makes the convolutions stable. In our experiments, we find that training generative flows models often diverge when using coupling layers. The reason is that the output of model may be pretty large when using coupling layers. The log-likelihood in Eq. (2) is composed of two terms. $p_Z(z)$ often chooses Gaussian distribution. When the output is pretty large, $p_Z(z)$ tends to zero, then the log-likelihood tends to infinity. Coupling layers can be written as:

$$\begin{aligned} \boldsymbol{y_1} &= \boldsymbol{x_1} \\ \boldsymbol{y_2} &= g(\boldsymbol{x_2}; c(\boldsymbol{x_1})) \end{aligned} \quad (19)$$

where c is a function of $\boldsymbol{x_1}$ and g is an invertible function with respect to $\boldsymbol{x_2}$. Dinh et al. (2016) used hyperbolic function to make the output of c bounded. We further extend the idea to control the value of the output of c to prevent divergence during training. For matrix exponential coupling layers, we modify Eq.(11) to:

$$\boldsymbol{y_2} = e^{(u_1 \tanh(u_2 \boldsymbol{S} + v_2) + v_1)} \boldsymbol{x_2} + \boldsymbol{b} \quad (20)$$

where u_1, v_1, u_2, v_2 are scalar parameter, which can be learned during training, and $\tanh(\cdot)$ is hyperbolic function.
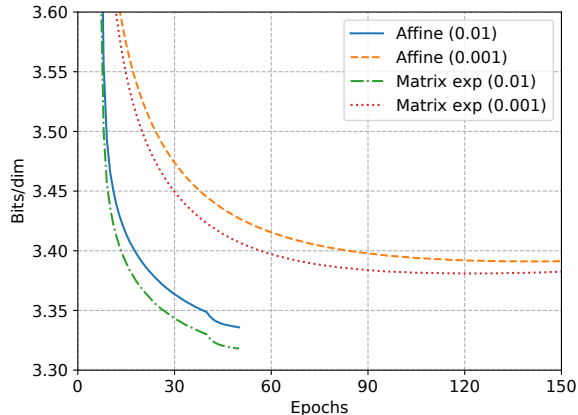

Figure 2. Bits per dimension curve on CIFAR10 test set with different coupling layers and learning rate.

Table 4. Comparison of models with different coupling layers and learning rate. Performance is measured in bits per dimension. In brackets are the learning rate. Results are obtained by running 3 times with different random seeds, $\pm$ reports standard deviation.

Model	CIFAR10
Affine (0.01)	3.336 ± 0.002
Affine (0.001)	3.391 ± 0.003
Matrix exp (0.01)	3.324 ± 0.004
Matrix exp (0.001)	3.381 ± 0.008

Initialize $v_1 = 0, v_2 = 0$. We first initialize $u_1 = 1$ and $u_2 = 1$. If the model diverges, then we initialize u_1 and u_2 with smaller values. We repeat this operation until convergence. Using this form of coupling layers can make training more stable and allows us to use a larger learning rate. Affine coupling layers have the similar form:

$$\boldsymbol{y_2} = \exp(u_1 \tanh(u_2 \boldsymbol{s} + v_2) + v_1) \odot \boldsymbol{x_2} + \boldsymbol{b} \quad (21)$$

We run models on CIFAR10 dataset with learning rate 0.01 and 0.001 to compare the convergence speed. Models with learning rate 0.01 are trained for 50 epochs, and models with learning rate 0.001 are trained for 150 epochs. We also compare our matrix exponential coupling layers with affine coupling layers. Figure 2 and table 4 show the results. Using a learning rate 0.01 achieves better performance and converges faster than using a learning rate 0.001. The results also show that matrix exponential coupling layers perform better than affine coupling layers.

6.3. 1×1 Convolutions

We run models on CIFAR10 dataset to compare the performance of standard 1×1 convolutions, $\boldsymbol{PLU}$ decomposition

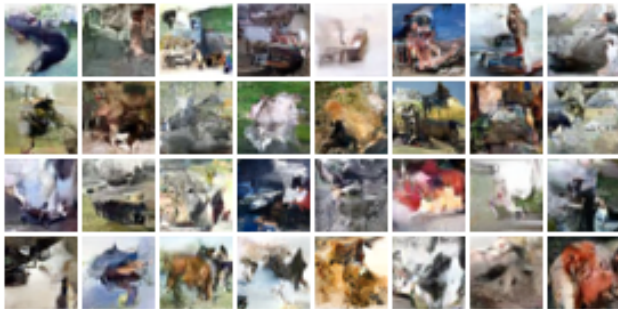*Figure 3.* Samples from our trained ImageNet64 model
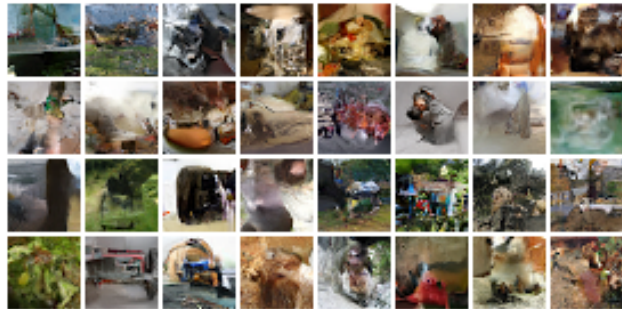

Figure 4. Samples from our trained CIFAR10 model

Figure 5. Samples from our trained ImageNet32 model

1×1 convolutions and matrix exponential 1×1 convolutions. All models have the same parameter settings expect the 1×1 convolutions. We also record the running time per epoch to compare the computation of convolutions. All models are trained on one TITAN Xp GPU. Table 5 shows the result. Our matrix exponential 1×1 convolutions achieve nearly same performance on density estimation as standard convolutions and have nearly the same computation compared with $\boldsymbol{PLU}$ decomposition 1×1 convolutions.

Table 5. Comparison of standard, $\boldsymbol{PLU}$ decomposition and matrix exponential convolutions. Performance is measured in bits per dimension. Computation is measured in running time per epoch. Results are obtained by running 3 times with different random seeds, $\pm$ reports standard deviation.

Convolutions	CIFAR10	Time
Standard	3.324± 0.001	844.3± 2.6s
Decomposition	3.330± 0.007	668.3± 19.6s
Matrix exp	3.324± 0.004	669.5± 10.1s

6.4. Truncate Matrix Exponential

Matrix exponential is an infinite matrix series. We need to truncate it at a finite term to approximate it. We use Algorithm 1 to approximate it, which costs about $(s + k - 1)n^3$ FLOPs. In this section, we present the coefficient $m = s + k - 1$ during our training. We set the tolerable error of Algorithm 1 $\epsilon = 10^{-8}$. We count 1 million times of coefficient m when computing matrix exponential. In Table 6, we show the mean, standard deviation, maximum and minimum of coefficient m. The coefficient m is no more than 11 and is about 9 in average. Experiments show that matrix exponential can converge fast.

Table 6. Mean, standard deviation, maximum and minimum of the coefficient m.

	Mean	Std	Max	Min
Coefficient m	9.28	0.94	11	2

6.5. Samples

We show the samples from our trained models on CIFAR10, ImageNet32 and ImageNet64 datasets in Figure 3 to 5. Our

CIFAR10 model takes 1.67 seconds to generate a batch of 64 samples on one NVIDIA 1080 Ti GPU.

7. Conclusion

In this paper, we propose a new type of generative flows, called matrix exponential flows, which utilizes the properties of matrix exponential. We incorporate matrix exponential into neural networks and combine it with generative flows. We propose matrix exponential coupling layers which are a generalization of affine coupling layers. In order to solve the stability problem, we propose matrix exponential 1×1 convolutions and improve the coupling layers. Our model significantly speeds up the training process. Based on matrix exponential, we hope that more layers can be proposed or incorporate it into other layers.

Acknowledgements

References

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, 2019.

Chen, R. T., Behrmann, J., Duvenaud, D. K., and Jacobsen, J.-H. Residual flows for invertible generative modeling. In *Advances in Neural Information Processing Systems*, pp. 9916–9926, 2019.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

Culver, W. J. On the existence and uniqueness of the real logarithm of a matrix. *Proceedings of the American Mathematical Society*, 17(5):1146–1151, 1966.

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.

Durkan, C., Bekasov, A., Murray, I., and Papamakarios, G. Neural spline flows. In *Advances in Neural Information Processing Systems*, pp. 7511–7522, 2019.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Grathwohl, W., Chen, R. T., Bettencourt, J., Sutskever, I., and Duvenaud, D. Ffjord: Free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations*, 2018.

Hall, B. *Lie groups, Lie algebras, and representations: an elementary introduction*, volume 222, pp. 31–71. 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

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, 2019.

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

Huang, C.-W., Krueger, D., Lacoste, A., and Courville, A. Neural autoregressive flows. In *International Conference on Machine Learning*, pp. 2083–2092, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

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

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

Liou, M. A novel method of evaluating transient response. *Proceedings of the IEEE*, 54(1):20–23, 1966.

Moler, C. and Van Loan, C. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM review*, 45(1):3–49, 2003.

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

Rezende, D. and Mohamed, S. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pp. 1530–1538, 2015.

Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pp. 1278–1286, 2014.

Van Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. Pixel recurrent neural networks. In *International Conference on Machine Learning*, pp. 1747–1756, 2016.

Ward, P. N., Smofsky, A., and Bose, A. J. Improving exploration in soft-actor-critic with normalizing flows policies. *arXiv preprint arXiv:1906.02771*, 2019.