FInC Flow: Fast and Invertible $k \times k$ Convolutions for Normalizing Flows

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Electronics and Communication Engineering by Research

by

ADITYA V KALLAPPA
2019702012
aditya.kallappa@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
November, 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled **"FInC Flow: Fast and Invertible $k \times k$ Convolutions for Normalizing Flows"** by **Aditya V Kallappa**, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Dr. Girish Varma

To my Parents

Acknowledgments

My salutations to *Mahābaleśvara* without Whose grace this work would not be possible. I would like express my unwavering gratitude to my parents for their constant support. I also would like to thank my advisor Dr. Girish Varma for taking me in as a research student under his wing. I thank him also for providing constant support and guidance not only to this work but also other projects as well. I would also thank Sandeep Nagar (co-author) for contributing to this work. My special thanks to friends of IIIT viz., Prateek Pani, Saransh Dave and many others for keeping me company and helping me in various tasks/projects. I would like to credit the institute - IIITH for providing me the resources throughout the journey. I would like to appreciate my friends and family who stood with me during difficult times and has contributed to this work behind the scenes.

Abstract

Invertible convolutions have been an essential element for building expressive normalizing flow-based generative models since their introduction in Glow. Several attempts have been made to design invertible $k \times k$ convolutions that are efficient in training and sampling passes. Though these attempts have improved the expressivity and sampling efficiency, they severely lagged behind Glow which used only 1×1 convolutions in terms of sampling time. Also, many of the approaches mask a large number of parameters of the underlying convolution, resulting in lower expressivity on a fixed run-time budget. We propose a $k \times k$ convolutional layer and Deep Normalizing Flow architecture which i.) has a fast parallel inversion algorithm with running time O(nk^2) (n is height and width of the input image and k is kernel size), ii.) masks the minimal amount of learnable parameters in a layer. iii.) gives better forward pass and sampling times comparable to other $k \times k$ convolution-based models on real-world benchmarks. We provide an implementation of the proposed parallel algorithm for sampling using our invertible convolutions on GPUs. Benchmarks on CIFAR-10, ImageNet, and CelebA datasets show comparable performance to previous works regarding bits per dimension while significantly improving the sampling time.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Generative models have been and is a dominant topic interest in Deep Learning Field. It is especially true in the advent of Text Based Generative Models like Dall-E [3], Imagen [4], Stable Diffusion [5], Midjourney [6] as well as text generative models like GPT [7]. In this work, we look at a type of generative models called Normalizing Flows which is entirely based on invertible and differentiable functions. We look at Normalizing Flows in detail in Chapter 3

1.1 Generative Adversarial Networks

GANs, or Generative Adversarial Networks, are a type of neural network architecture that can generate new data samples that are similar to a given training dataset. They consist of two main components: a generator network (G) and a discriminator network (D) as shown in Figure 1.1.

The generator network takes a random noise vector(z) as input and produces a new data sample. The discriminator network takes either a real data sample(x) from the training dataset or a generated sample from the generator network as input and outputs a probability value indicating whether the input is real or fake.

During training, the generator network tries to produce samples that can fool the discriminator network into thinking they are real, while the discriminator network tries to correctly distinguish between real and fake samples. The two networks are trained together in an adversarial manner, with the goal of both networks improving their performance over time.

Once trained, the generator network can be used to generate new data samples that are similar to the training dataset but not identical. GANs have been used for a variety of applications, including image and video generation, text generation, and data augmentation.

1.1.1 Training GANs

The goal of G is to generate fake data points that can fool D into thinking that they are real data points. The goal of D is to correctly distinguish between real and fake data points.

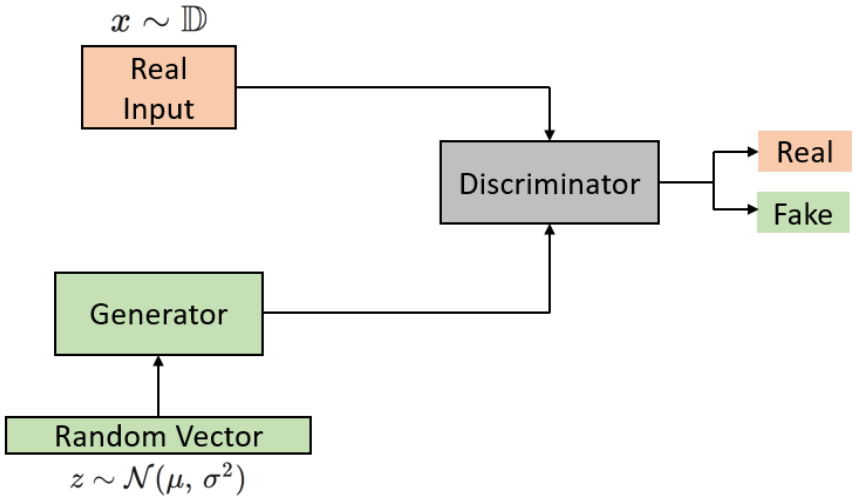


Figure 1.1: GANs: Generative Adversarial Networks.

To achieve this, we define a loss function $L(D, G)$ that measures how well D and G are doing in the game. The loss function consists of two parts: a) The discriminator loss L_D, which measures how well D is able to distinguish between real and fake data points, b) The generator loss L_G, which measures how well G is able to generate fake data points that can fool D

$$\min_G \max_D L(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

where $p_{data}(x)$ and $p_z(z)$ are the probability distributions of input and random noise respectively.

The first term in the loss function measures the average log probability that D assigns to real data points. The second term measures the average log probability that D assigns to fake data points generated by G. The generator loss measures the average log probability that D assigns to fake data points generated by G.

During training, we alternate between optimizing D to minimize L_D and optimizing G to minimize L_G. This creates a game-like scenario in which D and G are competing against each other, and they both get better over time as they try to outsmart each other.

The final result of training is a generator network G that can generate new data points that are similar to the real data, and a discriminator network D that can accurately distinguish between real and fake data points.

1.2 Autoencoders

Autoencoders are the generative models consisting of encoder-decoder architecture as shown in Figure 1.2. The encoder network compresses the input data into a lower-dimensional representation and a decoder network that reconstructs the original data from the compressed representation.

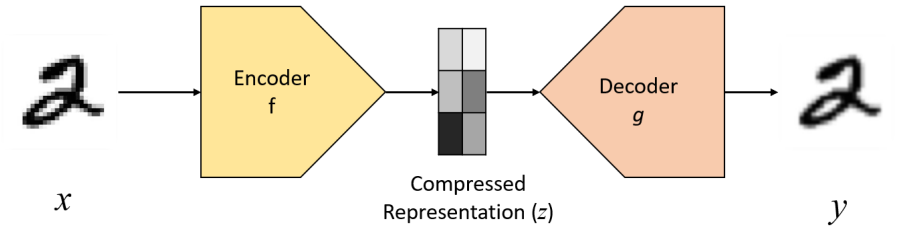


Figure 1.2: Autoencoders

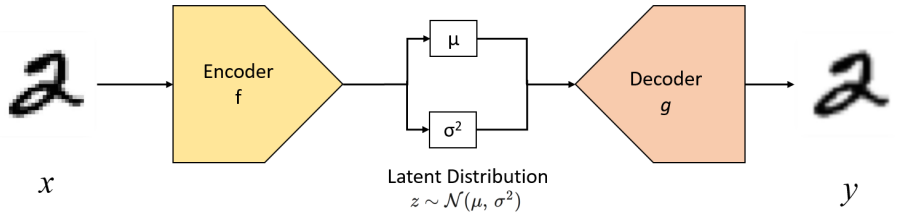


Figure 1.3: Variational Autoencoders

The basic idea behind an autoencoder is to learn a compressed representation of the input data that captures the most important features of the data. The encoder network reduces the dimensionality of the input data and creates a compressed representation, while the decoder network takes this compressed representation and reconstructs the original input data.

1.2.1 Training Autoencoders

Let x be the input data, and y be the output data. An autoencoder consists of two parts: an encoder function f and a decoder function g. The goal of training an autoencoder is to minimize the reconstruction error between the input data X and the output data Y. This can be done by minimizing the following loss function:

$$L(x, y) = ||x - y||^2$$

where $||.||$ represents Euclidean Norm. By minimizing this loss, autoencoders learn a compressed representation of the input data that captures the most important features of the data.

1.2.2 Variational Autoencoders (VAE)

While Autoencoders are good for data compression, a variation of autoencoders called Variational Autoencoders (VAEs) are more usefeul for data generation. VAEs are different from traditional autoencoders in that they learn a probabilistic distribution of the input data, rather than just learning a compressed representation.

VAEs consist of an encoder network that maps the input data to a probability distribution in a lower-dimensional latent space and a decoder network that maps the latent variable back to the output data (Figure 1.3). The key difference between VAEs and traditional autoencoders is that VAEs use a probabilistic approach to learn the distribution of the latent variable.

1.2.3 Training VAEs

Mathematically, let x be the input data, and let z be the latent variable in the lower-dimensional space. The encoder network maps x to the parameters of a probability distribution over z. This distribution is typically chosen to be a multivariate Gaussian distribution with a diagonal covariance matrix. The encoder network produces the mean and variance of the distribution, denoted as μ and σ^2, respectively. The decoder network then maps the latent variable z back to the output data y. The decoder network is typically a neural network that takes z as input and produces Y as output. Mathematically, let x be the input data, and let z be the latent variable in the lower-dimensional space. The encoder network maps x to the parameters of a probability distribution over z. This distribution is typically chosen to be a multivariate Gaussian distribution with a diagonal covariance matrix. The encoder network produces the mean and variance of the distribution, denoted as μ and σ^2, respectively. The decoder network then maps the latent variable z back to the output data y.

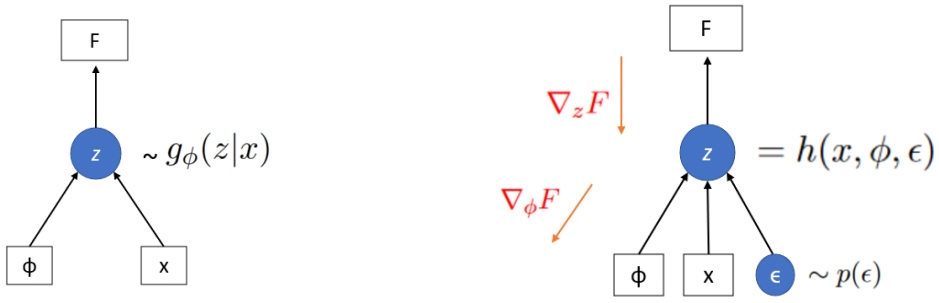
Let f_θ be the encoder network parameterized by θ, g_ϕ be the decoder network parameterized by ϕ. Now,

$$\begin{aligned}
\log f_\theta(x) &= \mathbb{E}_{g_\phi(z|x)}[\log f_\theta(x)] \\
&= \mathbb{E}_{g_\phi(z|x)}\left[\log\left[\frac{f_\theta(x,z)}{f_\theta(z|x)}\right]\right] \\
&= \mathbb{E}_{g_\phi(z|x)}\left[\log\left[\frac{f_\theta(x,z)g_\phi(z|x)}{f_\theta(z|x)g_\phi(z|x)}\right]\right] \\
&= \mathbb{E}_{g_\phi(z|x)}\left[\log\left[\frac{f_\theta(x,z)}{g_\phi(z|x)}\right]\right] + \mathbb{E}_{g_\phi(z|x)}\left[\log\left[\frac{g_\phi(z|x)}{f_\theta(z|x)}\right]\right] \qquad (1.1)
\end{aligned}$$

The training objective for VAEs is to maximize the evidence lower bound (ELBO) given in Equation (1.1), which is a lower bound on the log-likelihood of the data under the model. The ELBO is composed of two terms: the reconstruction loss and the KL divergence between the learned distribution and a prior distribution over the latent variable. The reconstruction loss measures the difference between the input data and the output data, while the KL divergence encourages the learned distribution to be close to the prior distribution over the latent variable.

KL (Kullback-Leibler) Divergence: KL Divergence is a mathematical measure used to quantify the difference between two probability distributions. It is used to assess how one probability distribution (lets say P) differs from another probability distribution (lets say Q).

Mathematically, $\text{KL}(P||Q) = \mathbb{E}_{x\ P(x)}\left[\log\left[\frac{P(x)}{Q(x)}\right]\right] = \sum_x P(x)\log\left[\frac{P(x)}{Q(x)}\right]$



(a) (b)

Figure 1.4: Reparameterization Trick: Black arrows represent the forward flow, red arrows represent the back propagation. Variables in the rectangular boxes are deterministic and the ones that are in the circles are the random variables. (a) Latent variables sampled from distribution (b) Latent variables sampling decoupled from the model parameters

1.2.4 Reparameterization Technique

In VAEs, the latent variables are typically sampled from a Gaussian distribution with a diagonal covariance matrix. The standard approach to sample from this distribution involves adding the mean and the standard deviation to a noise term sampled from a standard Gaussian distribution, and then multiplying the standard deviation with the noise term, and adding the mean. However, this approach is not differentiable, which makes it difficult to compute gradients with respect to the parameters of the network.

The reparameterization trick is a method to rewrite the sampling operation as a differentiable operation, which allows for backpropagation and the computation of gradients. By reparameterizing the sampling operation, we can decouple the randomness of the latent variables from the model parameters, which allows for efficient computation of gradients through the model parameters.

1.3 Overview

Normalizing flow is an important subclass of Deep Generative Models that offers distinctive benefits [8]. In comparison to GANs [9] and VAEs [10], they are trained using a very intuitive Maximum Likelihood loss function. Images and the *latent vector*, which is required to have a Gaussian distribution, correspond one-to-one in flow models. Despite these intriguing characteristics, GANs and VAEs are utilized more frequently. This is due to the need for the Normalizing Flows transformations to be invertible, which significantly restricts the neural network types employed. For deployment in a real-world scenario, the invertible transformations must be efficiently calculable in the forward and sample stages.

A significant breakthrough came with Glow [11] which used 1×1 invertible convolutions to design normalizing flows. If it exists, the inverse function for a 1×1 convolution also happens to be a 1×1 convolution. Since computing 1×1 convolution has fast parallel algorithms for which running time does not depend on the spatial dimensions, they are also highly efficient in forward pass (i.e. computing *latent vector* from an image) as well as the sampling passes (i.e. computing image from a sampled *latent vector*). Extending Glow [11] to use invertible $k \times k$ convolutions promises to improve the expressivity further, allowing it to model more complex datasets. However, this is a challenging problem since the inverse function for a $k \times k$ convolution, in general, is given by a $n^2 \times n^2$ matrix where $n = H = W$ (i.e. the spatial dimensions). Hence, while the forward pass can be fast, the trivial approach for the sampling pass will cost $O(n^4)$ operations per convolutional layer.

CInC Flow [12] introduced a padded 3×3 convolution layer design and gave it the necessary and sufficient conditions to make it invertible. They showed that the convolution matrix is lower triangular by ensuring padding in only two sides of the input. Furthermore, all the diagonal entries of the convolution matrix are equal to a single weight parameter. By setting this parameter to 1, they ensured that the convolutions are invertible, and Jacobian is always 1.

We build on their work by proposing a parallel inversion algorithm for their convolution design. The parallel algorithm only uses $\mathrm{O}(nk^2)$ sequential operations, unlike $\mathrm{O}(n^2k^2)$ operations used by most previous works. We also build a normalizing flow architecture, where channel-wise splitting is further used to parallelize operations.

1.4 Our Contributions

1. We design a $k \times k$ invertible convolutional layer with a fast and parallel invertible sampling algorithm (see Sections 5.1, 5.2).

2. We build a normalizing flow architecture based on the fast invertible convolution, which uses channel wise splitting to improve the parallelism further (see Sections 5.3, 5.4).

3. We provide a fast GPU implementation of our parallel inversion algorithm and benchmark the sampling times of the model (see Section 7). We show greatly improved sampling times due to our parallel inversion algorithm, while giving similar bits per dimensions as compared to other works.

1.5 Thesis Outline

Chapter 2 discusses related works, Chapter 3 presents a formal introduction to Normalizing Flows. Chapter 4 explores various methods for enhancing the inverse of convolution. Chapter 5 presents FInC Flow, our work. Chapter 7 presents comparative results of our work compared to others. We also provide the implementation of our inverse algorithm in Chapter 6.

Chapter 2

Related Works

In this section, we note down the related works in generative models.

2.1 Generative Models in General

A generative model in essence generates a signal on which it is trained to generate. For example, Waveglow [13], which is based on Normalizing Flows generates audio while Dall-E [3] generates images using text.

2.1.1 Generative Adversarial Networks

GANs are the type of Networks that uses Discriminator-Generator networks in an adversarial way. First introduced in [14], GANs have much evolved since then. DCGANs intorduced in [15] uses CNNs to generate images. Conditional GANs genearate images specific type of images based on input condition [16]. More diverse set of networks based on GANs have been introduced like [17, 18, 19] which have made significant changes in the architecture there by enchancing the usage of GANs. [20] proposes several improvements to training GANs. [21] introduces methods to visualize GANs.

2.1.2 Autoencoders

Autoencoders was first introduced in [22] as a neural network that reconstructs the input. A more formal way of representing the problem has been given by [23]. Different types of autoencoders have been introduced like Regularized Autoencoders [24] which provides regularization to train Autoencoders, Sparse Autoencoders [25], Denoising Autoencoders [26], Contrastive Autoencoders [27] etc. But the most popular type of Autoencoders are the Variational Autoencoders which have been used in various fields.

2.1.3 Variational Autoencoders(VAE)

[28] introduced VAEs to the world that can generate images through latent vector representation. Several changes to the original network have been made in the years since their introduction [29, 30, 31]. [32] introduces Gaussian Mixture based Encoder. [33] introduces a type of VAEs called Ladder VAE that recursively corrects the generative distribution.

2.1.4 Diffusion Models

Diffusion models have become really popular in the recent times with the advent of large text to image generative models. [34] introduces the concept of Diffusion in Deep Learning. However, things picked up with [35], [36] etc. Recently papers on Latent Diffusion like [5] have been in use to train and generate image to text models such as Imagen [4], Dall-E [3].

2.2 Normalizing Flows

Flows-based models construct complex distributions by transforming a probability density through a series of invertible mappings [37]. At the end of these invertible mapping, we obtain a valid distribution; hence, this type of flow is referred to as a Normalizing Flow model. Flow models apply the rule for change of variables; the initial density 'flows' through the sequence of invertible mappings [38]. Flow-based models generalize a dataset distribution into a latent space [8].

2.2.1 Convolution Based Models

An invertible neural network requires the inverse of the network with fast and efficient computation of the Jacobian determinant [39]. An invertible neural network can be used for generation and classification with more interpretability. [11] proposed an invertible 1×1 convolution building on top of NICE [40] and RealNVP [38] consisting a series of flow step combined in a multi-scale architecture. Each flow step consists of actnorm followed by an invertible 1×1 convolution, followed by a coupling layer (see Sec 5.3). Emerging [1] presented method to generalized 1×1 convolution to invertible $k \times k$ convolutions. Emerging chains two specific auto regressive convolutions [28] to form a single convolutional layer following the associativity of the convolution operation. Each of these autoregressive convolutions is chosen such that the resulting convolution matrix $\mathbf{M}$ is triangular with an inverse time of each of the convolutions is $\mathrm{O}(n \times n \times k^2)$. MintNet [39] presented a method for designing invertible neural networks by combining building blocks with a set of composition rules. The inversion of the proposed blocks necessitates a sequence of dependent computations that increase the network's sampling time. SNF [41] proposed a method to reduce the computation complexity of the Jacobian determinant by replacing the gradient term with a learned approximate inverse for each layer. This method avoids the determinant of Jacobian and makes it approximate, and requires an additional backward pass for

inversion of convolution. MaCow [2] while many other papers make use of the invertibility of triangular matrix to reduce inversion time, MaCow outperforms all of them by performing the inverse in O(nk^2) by carefully masking 4 kernels at the top, left, bottom, right to achieve a full convolution, but this flow model use four autoregressive convolutions to make an effective standard convolution. Woodbury [42] this paper employs the *Woodbury transformation* for invertible convolution, which is a generalized permutation layer that models dimension dependencies along the channel and spatial axes using the channel and spatial transformation. ButterflyFlow [43] introduced a new family of an invertible layer that works for special underlying structures and needs a sequence of layers for an effective invertible convolution.

2.2.2 Fast Algorithms for Invertible Convolutions

CInC Flow [12], derive necessary and sufficient conditions on a padded CNN for it to be invertible and require a single CNN layer for every effective invertible CNN layer. The padded CNN can leverage the advantage of parallel computation for inversion, resulting in faster and more efficient computation of Jacobian determinants.

The distinguishing feature of our invertible convolutions as compared to previous works is that we have a parallel inversion algorithm that does only $(2n-1)k^2$ operations where n is input size and k is kernel size. MaCow is the closest approach that takes twice the number of operations. Some of the approaches, like MintNet and SNF, do achieve a lesser number of operations. However, they are not proper normalizing flows as they compute only an approximate inverse. We use the convolution design from CInC Flow but give a parallel inversion algorithm for it. Furthermore, our FInC Flow *Unit* is designed to efficiently parallelize the operations by splitting the convolution operations channel-wise. In Table 2.1, we compare our proposed flow model with the existing model in terms of the receptive fields/number of learnable parameters, complexity of computing the inverse of convolution layer for sampling.

Method	# of ops	# params / CNN layer	Complexity of Jacobian	Inverse
FInC Flow (our)	$(2n-1)k^2$	k^2-1	1	exact
Woodbury [42]	cn^2	k^2	O$(d^2(c+n)+d^3)$	exact
MaCow [2]	$4nk^2$	$k(\lceil\frac{k}{2}\rceil-1)$	O(n^3)	exact
Emerging [1]	$2n^2k^2$	$k(\lceil\frac{k}{2}\rceil-1)$	O(n)	exact
CInC Flow [12]	n^2k^2	k^2-1	1	exact
MintNet [39]	$3n$	$\frac{k^2}{3}$	O(n)	approx
SNF [41]	k^2	k^2	approx	approx

Table 2.1: Comparison of the learnable parameters. where $n \times n$ is input size, $k \times k$ is filter size which is constant, c is number of input/output channels. d is the number of latent dimensions. # of ops: required number of operations for the inversion of convolutional layers. The complexity of Jacobian: Time complexity for calculating the Jacobian of a single convolution layer. For FInC Flow and CInC Flow, the Jacobian is 1, since the Convolution matrix is lower triangular with diagonal entries being 1.

Chapter 3

Preliminaries

In this chapter, we describe Normalizing Flows formally. We derive the Negative Log Likelihood (NLL) Loss.

3.1 Normalizing Flows

Formally, Normalizing Flows is a series of transformations of a known simple probability density into a much more complex probability density using invertible and differentiable functions. These invertible functions allow us to write the probability of the output as a differentiable function of the model parameters. As a result, the models can be trained using backpropagation with the negative log likelihood loss function.

Let $\mathbf{X} \in \mathbb{R}^d$ be a random variable with tractable density $p_{\mathbf{X}}$. Let $f : \mathbb{R}^d \to \mathbb{R}^d$ be a differentiable and invertible function. If $\mathbf{Y} = f(\mathbf{X})$ then the density of X can be calculated using variable change formula as

$$p_Y(y) = p_X(x) \left| \det \frac{\partial f^{-1}(y)}{\partial y} \right|$$

where $\frac{\partial f^{-1}(y)}{\partial y}$ is the Jacobian matrix of $f^{-1}(y)$ denoted by $J_{f^{-1}(y)}$.

By the identity $\det(A)^{-1} = \det(A^{-1})$ (where A is an invertible matrix), we have:

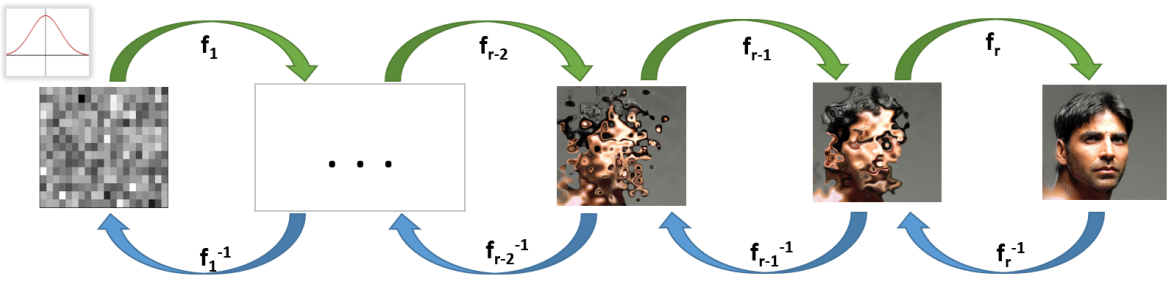


Figure 3.1: Normalizing Flows: Generate an image from random samples from a known distribution

$$p_Y(y) = p_X(x)|\det J_{f(x)}|$$

If X is transformed using a sequence of functions f_i's i.e., $f = f_1 \circ f_2 \circ f_3 \circ \cdots \circ f_r$, then probability density, $p_Y(y)$ can be expressed as

$$p_Y(y) = p_X(x) \cdot \prod_{i=r}^{1} |J_{f(y_i)}| \tag{3.1}$$

where $y_i = f_1 \circ \cdots \circ f_i(x)$. The log-probability of p_Y which will be used to model the complex image distribution is given by,

$$\log p_Y(y) = \log p_X(x) + \sum_{i=1}^{r} \log |\det J_{f(y_i)}| \tag{3.2}$$

The functions f_i will be given by neural network layers and the above function can be computed during the forward pass of the neural network. The negative of $\log p_Y(y)$ called the negative log likelihood loss (NLL) is minimized when the samples in the dataset are given highest probabilities. Hence it gives a simple, interpretable loss function for training the model.

3.2 ActNorm Layer

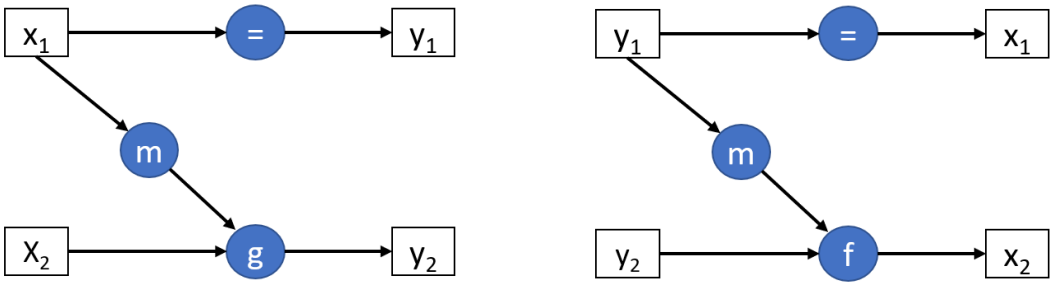
ActNorm is short for "Activation Normalization" and is a form of affine transformation that normalizes the activations in a neural network layer. It applies a separate scaling and shifting operation to each activation channel, so that the mean and standard deviation of the channels are normalized to have zero mean and unit variance, respectively. This allows the network to better model the data distribution and to learn more efficiently. ActNorm is similar to Batch Normalization (BN), but instead of normalizing over mini-batches of data, it normalizes over each individual data point. This makes ActNorm more suitable for smaller datasets and real-time applications, as it does not require batch statistics to be estimated and stored. Equations (3.3), (3.4) describe the forward and inverse flow operations of ActNorm layer respectively. s and b represent scale and bias parameters respectively.

$$y_{i,j} = s \odot x_{i,j} + b \tag{3.3}$$

$$x_{i,j} = \frac{y_{i,j} - b}{s} \tag{3.4}$$

3.3 Coupling Layer

A coupling layer is a type of invertible transformation that divides the input image into two parts and applies a different transformation to each part. One part is used to compute the scale and shift parameters that are applied to the other part. The transformed parts are then concatenated to produce



(a) Forward Pass

(b) Inverse Pass

Figure 3.2: Coupling Operation: Input x is split into x_1 and x_2, m and g are functions that transform x_1 while $f = g^{-1}$

the output. This allows the model to capture complex correlations between input variables and to learn a more complex data distribution. (See Figure 3.2)

Coupling layers can be either additive or affine. Additive coupling layers transform one subset of the input using an additive function, such as an element-wise sum. Affine coupling layers, on the other hand, apply an affine transformation to one subset of the input, which includes a scaling and a shift parameter. Coupling layer typically are made of a 3×3 convolution followed by a 1×1 and a modified 3×3 convolution used in Emerging. Equations (3.5) - (3.10) give the forward flow of the affine coupling layer. Equations (3.11) - (3.16) give the inverse flow of the affine coupling layer. NN in the equations refer to the set of convolution layers like the one described above.

$$x_a, x_b = \text{split}(x) \tag{3.5}$$
$$(\log s, t) = \text{NN}(x_b) \tag{3.6}$$
$$s = \exp(\log s) \tag{3.7}$$
$$y_a = s \odot x_a + t \tag{3.8}$$
$$y_b = x_b \tag{3.9}$$
$$y = \text{concat}(y_a, y_b) \tag{3.10}$$

$$y_a, y_b = \text{split}(y) \tag{3.11}$$
$$(\log s, t) = \text{NN}(y_b) \tag{3.12}$$
$$s = \exp(\log s) \tag{3.13}$$
$$x_a = \frac{y_a - t}{s} \tag{3.14}$$
$$x_b = y_b \tag{3.15}$$
$$x = \text{concat}(x_a, x_b) \tag{3.16}$$

3.4 Squeeze

The main purpose of a squeeze layer is to reduce the spatial size of the input tensor, while increasing the number of channels, which can help to extract more features from the input data. Squeeze layers are common in all Deep Networks for e.g., MaxPooling, AveragePooling Layers. However the commonly used squeeze layers are not invertible and as such they cannot be used in normalizing flows. So, in normalizing flows, we take features from spatial to channel dimension [44], i.e., we reduce the feature dimension by a factor of four, two across the height dimension and two across the width dimension resulting in increase of the channel dimension by four. As used by [38], we use squeeze layer to reshape the feature maps to have smaller resolution but more channels.

3.5 Split

Input is split into two halves across the channel dimension. We retain the first half, and a function parameterized by first half transform the second half. The transformed second half is modeled as Gaussian samples, are the *latent vectors*. Splitting of input can involve multiple strategies, for example, [38] uses checkerboard pattern while we keep it simple by splitting across height and width.

3.6 1×1 Convolutional Layer

A 1×1 convolutional layer is a convolutional layer with a kernel size of 1×1, which means that it applies a linear transformation to the input data without any spatial filtering. The purpose of using this layer is to increase the expressiveness of the flow, while also introducing non-linearity to the transformation. By applying a 1×1 convolution to the data, the flow is able to learn complex correlations between the input features, which can be useful for modeling complex data distributions. Additionally, the use of a 1×1 convolutional layer allows for efficient computation of Jacobian and parameter reduction, as it can be seen as a form of dimensionality reduction that reduces the number of channels in the input data. This layer was introduced in Glow. Equations (3.17) and (3.18) give the forward and inverse flow of this layer. Calculation of inverse is not a big issue as this is just a linear transformation.

$$y_{i,j} = W x_{i,j} \tag{3.17}$$

$$x_{i,j} = W^{-1} y_{i,j} \tag{3.18}$$

Chapter 4

Inverse Convolution

4.1 General Convolution

Definition 4.1.1 (Same Padding)**.** *Given an image X with shape $H \times W \times C$, the (t, b, l, r) padding of X is the image $\hat{X}$ of shape $(H + t + b) \times (W + l + r) \times C$ defined as*

$$\hat{X}_{i,j,c} = \begin{cases} X_{i-t,j-l,c} & i - t < H \ \wedge \ j - l > W \\ 0 & \textit{otherwise} \end{cases} \tag{4.1}$$

Definition 4.1.2 (Convolution Matrix)**.** *Let K be a kernel of shape $k \times k \times C \times C$. The Convolution Matrix(**M**) of kernel K with input X of size $H \times W \times C$ with padding (t, b, l, r) is a matrix describing the linear map $X \mapsto \hat{X} * K$*

A general $k \times k$ convolution can be thought of as a set of Linear operations on a different set of values at each step(stride). This also means that the convolution operation can be thought of as a linear transformation of input by the convolution matrix defined in 4.1.2. Figure 4.1 shows the convolution operation of a 3×3 input image with a 3×3 kernel as a linear transformation by **M**. Notice that input uses 'same' padding (defined in Equation (4.1)) to retain the original input size.

Figure 4.2 shows that **M** for a 3×3 image and 3×3 kernel results in a non triangular matrix. The inverse of the convolution is equivalent to finding out the inverse of **M**. For a general $N \times N$ square matrix, the inverse time is O(N^3) and thus the sampling time becomes very high when we increase image size. So, we make certain changes to input and kernel to make invertibility of **M** a little faster.

4.2 Emerging Convolutions

We will discuss a type of Convolution called Emerging Convolution which makes use of autoregressive convolutions in this section.

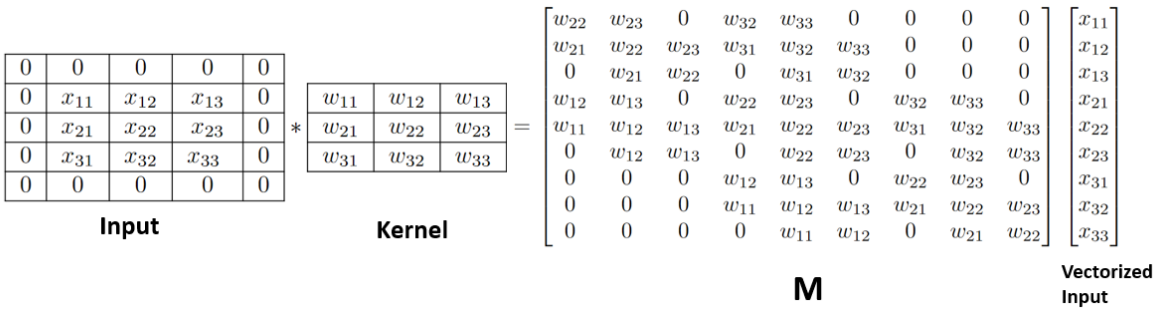


Figure 4.1: A general convolution as a linear transformation of input

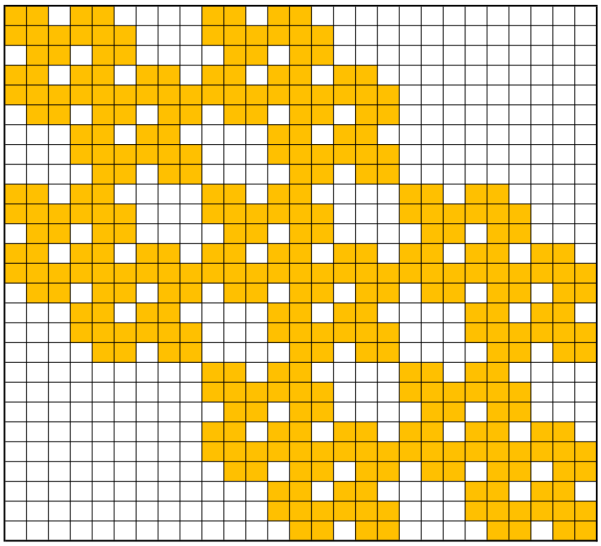


Figure 4.2: Convolution Matrix: **M** obtained from convolution of a 3×3 image with a 3×3 kernel with 'same' padding. White Colored pixels represent 0 and colored pixels represent non zero values.

4.2.1 Autoregressive Cnvolutions

In auto-regressive convolutions, the convolutional kernel is applied to the input data in a sequential manner, typically from left to right and top to bottom, rather than applying the kernel simultaneously across the entire input data as in traditional convolutions. At each step, the convolutional kernel takes into account the previously generated pixels or feature maps, conditioning the output on the context of the previous outputs. This allows the model to capture complex dependencies and patterns in the data in a sequential and autoregressive manner.

One common approach to implement auto-regressive convolutions is to use masked convolutions. In masked convolutions, the convolutional kernel is masked or constrained such that it can only access and modify a subset of the input pixels or feature maps. The mask is designed to ensure that the convolutional kernel only depends on previously generated pixels or feature maps, following a specific

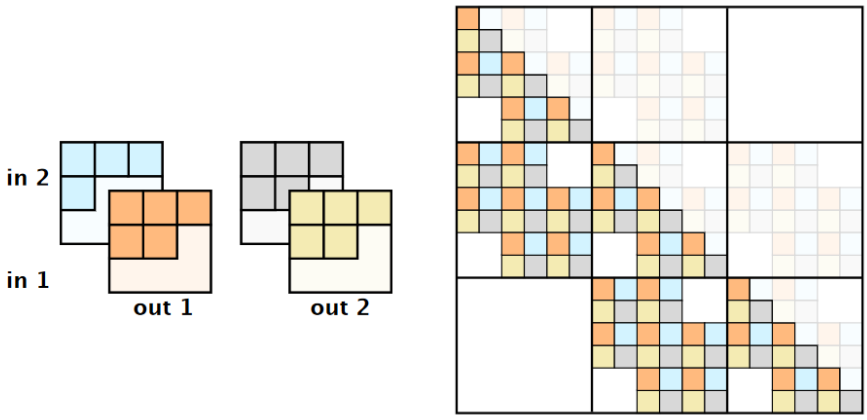


Figure 4.3: An autoregressive 3×3 convolution layer with two input and output channels. The convolution uses one-pixel-wide 'same' padding. Left: the autoregressive convolution filter K. Right: The convolutional matrix $\mathbf{M}$. Note that $\mathbf{M}$ is traingular. Image courtesy: [1]

autoregressive ordering. For example, in image generation tasks, the mask may be designed to allow the kernel to access only the pixels to the left and above the target pixel, which have already been generated in previous steps. Equation (4.2) gives the mathematical representation of autoregressive convolutions. Here, s and b are two autoregressive masks, $x_{<t}$ denotes the input variables in x positioned ahead of x_t in the autoregressive order.

$$y_t = s(x_{<t} \odot x_t + b(x_{<t})) \tag{4.2}$$

Auto-regressive convolutions have several advantages. They excel at modeling long-range dependencies and capturing fine-grained patterns in the data. They have been successfully employed in various generative models, such as PixelCNN [45], PixelRNN [46], and Masked Autoregressive Flow (MAF) [47], for tasks like image generation, image completion, and image synthesis. However, it's important to note that auto-regressive convolutions have inherent computational limitations. The generation process is sequential in nature, with each output relying on the context of previously generated outputs. As a result, auto-regressive models can be computationally expensive and are not fully parallelizable, which can affect their efficiency in certain applications.

4.2.2 Emerging Convolutions

Emerging Convolutions are obtained by chaining specific type of autoregressive convolutions sequentially to obtain a full convolution. Emerging convolutions are more fliexible than auto regressive convolutions and they are invertible. Fig 4.3 shows two specifically masked convolutional kernels produce a traingular $\mathbf{M}$. And since $\mathbf{M}$ is triangular, the inverse is easy to obtain.

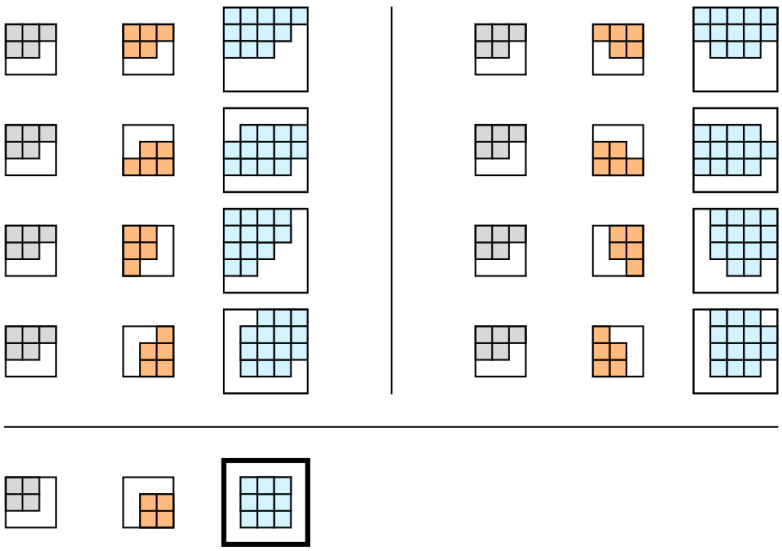


Figure 4.4: Achievable emerging receptive fields that consist of two distinct auto-regressive convolutions. Grey areas denote the first convolution filter and orange areas denote the second convolution filter. Blue areas denote the emerging receptive field, and white areas are masked. The convolution in the bottom row is a special case, which has a receptive field identical to a standard convolution. Image credits: [1]

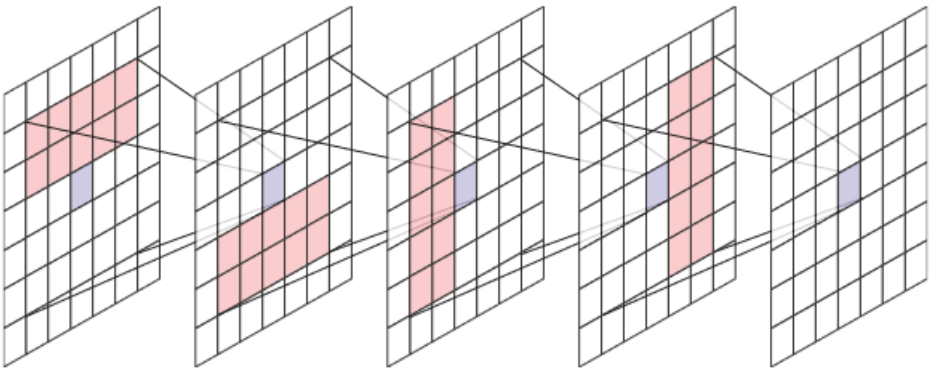


Figure 4.5: Visualization of the receptive field of four masked convolutions with rotational ordering. Image Credits: [2]

Figure 4.4 shows the different types of autoregressive convolutional filters resulting different receptive fields. Emerging flows works on the fact that the convolution operation is associative(See Equation (4.3)). To explain this, lets look at the bottom most row of Figure 4.4. It shows a special type of convolution called square emerging convolution which can be obtained by combining off center square convolutions. If K_1 and K_2 are two masked convolutional filters, then a) the general $k \times k$ convolution can be expressed as two consective $\frac{k+1}{2} \times \frac{k+1}{2}$ convolutions. Alternatively, b) We combine K_1 and K_2 to obtain a larger K and compute the convolution.

$$K_2(K_1 * X) = (K_2 * K_1) * X \tag{4.3}$$

4.3 Masked Convolutions (MACOW)

We will discuss another type of invertible convolution called - Masked Convolution in this section.

Since the autoregressive convolutions are very slow in genration of images owing to their inability to parallelize, [2] proposed a new type of masked convolution restrict the local connectivity in a small "masked" kernel. The two autoregressive neural networks, s and b are implemented with one-layer masked convolutional networks with small kernels (e.g. 2×5 in Figure 4.5 to ensure they only read contexts in a small neighborhood based on:

$$s(x_{<t}) = s(x_{t\star}), b(x_{<t}) = b(x_{t\star}) \tag{4.4}$$

where where $x_{t\star}$ denotes the input variables, restricted in a small kernel, on which x_t depends. By using masks in rotational ordering and stacking multiple layers of flows, the model captures a large receptive field (see Figure 4.5), and models dependencies in both the spatial and channel dimensions.

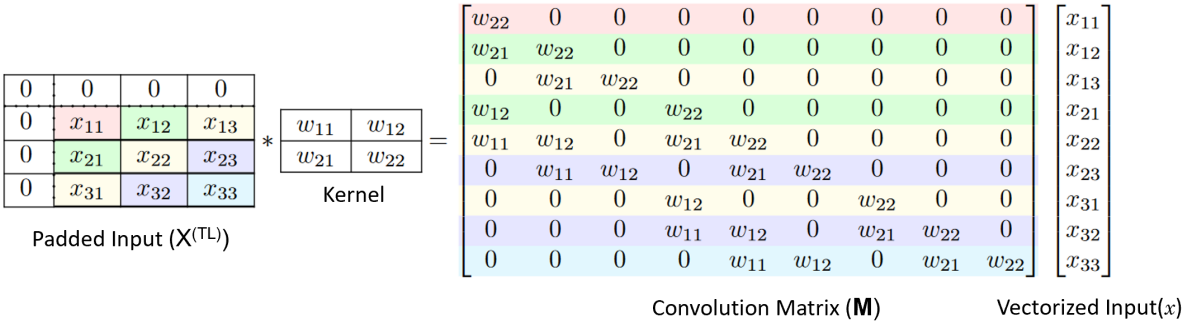


Figure 4.6: Convolution of a 3×3 Top Left (*TL*) padded image with a 2×2 kernel viewed as a linear transform of vectorized input(x) by the convolution matrix **M**. The *TL* padding on the input results in the making matrix **M** lower triangular, and all diagonal values correspond to $w_{k,k}$ of the filter. Each row of **M** can be used to find a pixel value. The rows or pixels with the same color can be inverted in parallel since all the other values required for computing them will already be available at a step of our inversion algorithm 1.

4.4 Top-Left Padded Convolutions

Definition 4.4.1 (Top-Left Padding)**.** *For an input image X with shape $H \times W \times C$, the top-left(*TL*) i.e., $(t, 0, l, 0)$ padding of X is the image $X^{(TL)}$ of shape $(H + t) \times (W + l) \times C$ is defined as*

$$X^{(TL)}_{i,j,c} = \begin{cases} X_{i-t,j-l,c} & i - t > 0 \ \wedge \ j - l > 0 \\ 0 & \textit{otherwise} \end{cases} \tag{4.5}$$

Equation (4.5) defines Top Left(TL) padding mathematically. What it means is that We pad the input only on the top and left side of the input. Interestingly, this turns **M** into a triangular matrix. Figure 4.6 shows the convolution of a 3×3 with a 2×2 filter with TL Padding.

We will discuss proofs about **M** for $C = 1$ i.e., number of channels = 1 and the same applies for arbitrary number of channels with a few modification in padding.

Lemma 1. ***M*** *is a lower triangular matrix with diagonal entries $K_{k,k}$ where K is the kernel of size $k \times k$*

Proof. Consider any entry in the upper right half of M. That is (i, j) such that $i < j$ according to the ordering given in the definition of M. $M_{i,j}$ is nothing but the scalar weight that needs to be multiplied to the j^{th} pixel of input when computing i^{th} pixel of the output. The linear equation relating these two

variable is as follows:

$$y_i = \sum_{l=0}^{k} \sum_{m=0}^{k} K_{k-l,k-m} x_{i_x-l,i_y-m} \tag{4.6}$$

From Equation (4.6) follows that if $j_x > i_x$ or $j_y > i_y$ then the i^{th} pixel of the output does not depend on the j^{th} pixel of the input and thus $\mathbf{M}_{i,j} = 0$. This also justifies that all diagonal coefficients of M are equal to $K_{k,k}$ □

Theorem 2. ***M*** *is invertible iff* $K_{k,k} \neq 0$

Proof. The proof of the theorem uses Lemma 1. Since **M** is lower triangular, the determinant is nothing but the product of diagonal entries, which is = $K_{k,k}^{HW}$ where H, W are the dimensions of the input/output image. □

Chapter 5

FInC Flow

We describe our approach including convolution layer design which has a fast parallel inversion algorithm with running time O(nk^2). For more clarity, we refer to the height of the image as H, width as W and channels as C in this section.

5.1 Convolution Design

We have discussed TL Padding and its related proof in the previous section. We define 3 more padded convolutions - Top-Right(TR), Bottom-Left(BL), Bottom-Rigth(BR) Padded Convolutions in 5.1.1, 5.1.2 and 5.1.3 respectively.

Definition 5.1.1 (Top-Right Padding)**.** *For an input image X with shape $H \times W \times C$, the top-right(*TR*) i.e., $(t, 0, 0, r)$ padding of X is the image $X^{(TR)}$ of shape $(H + t) \times (W + r) \times C$ is defined as*

$$X^{(TR)}_{i,j,c} = \begin{cases} X_{i-t,j,c} & i - t > 0 \ \wedge \ j - r < W \\ 0 & \textit{otherwise} \end{cases} \tag{5.1}$$

Definition 5.1.2 (Bottom-Left Padding)**.** *For an input image X with shape $H \times W \times C$, the bottom-left(*BL*) i.e., $(0, b, l, 0)$ padding of X is the image $X^{(BL)}$ of shape $(H + b) \times (W + l) \times C$ is defined as*

$$X^{(BL)}_{i,j,c} = \begin{cases} X_{i,j-l,c} & i - b < H \ \wedge \ j - l > 0 \\ 0 & \textit{otherwise} \end{cases} \tag{5.2}$$

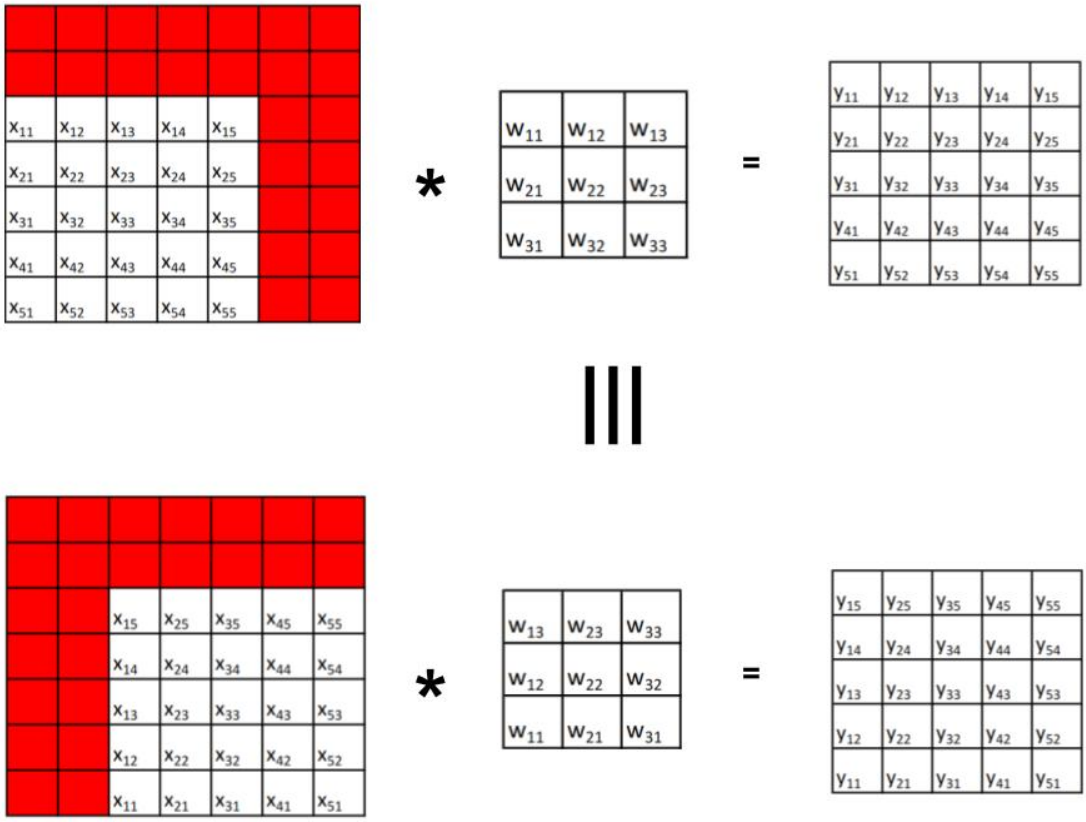


Figure 5.1: Top Right Padded Convolution is equivalent to Top Left Padded Convolution when input and kernel are flipped anti clockwise.

Definition 5.1.3 (Bottom-Right Padding)**.** *For an input image X with shape $H \times W \times C$, the bottom-right(*BR*) i.e., $(0, b, 0, r)$ padding of X is the image $X^{(BR)}$ of shape $(H + b) \times (W + r) \times C$ is defined as*

$$X^{(BR)}_{i,j,c} = \begin{cases} X_{i,j,c} & i - b < H \ \wedge \ j - r < W \\ 0 & \textit{otherwise} \end{cases} \tag{5.3}$$

Note that all the above padding techniques are equivalent. In fact, we use this fact to improve our algorithm which we will see in the later section. Figure 5.1 shows how appropriately flipping top right padded convolution results in the top left padded convolution.

Algorithm 1: Fast Parallel Inversion Algorithm of *TL* padded convolution block(PCB)

Input: K: Kernel of shape (C, C, k_H, k_W)

Y: output of the conv of shape (C, H, W)

Result: X: inverse of the conv. with shape (C, H, W).

1 **Initialization:** $X \leftarrow Y$;

2 **for** $d \leftarrow 0, H + W - 1$ **do**

3 **for** $c \leftarrow 0, C - 1$ **do**

/* The below lines of code executes parallelly on different threads on GPU for every index (c, h, w) of X on the dth diagonal. */

4 **for** $k_h \leftarrow 0, k_H - 1$ **do**

5 **for** $k_w \leftarrow 0, k_W - 1$ **do**

6 **for** $k_c \leftarrow 0, C - 1$ **do**

7 **if** *pixel* $(k_c, h - k_h, w - k_w)$ *not out of bounds* **then**

8 $X[c, h, w] \leftarrow$
$X[c, h, w] - X[k_c, h - k_h, w - k_w] * K[c, k_c, k_H - k_h - 1, k_W - k_w - 1];$

9 **end**

10 **end**

11 **end**

12 **end**

/* synchronize all threads */

13 **end**

14 **end**

5.2 Parallel Inversion Algorithm

We have presented our algorithm in Algorithm 1. The algorithm can be understood using Figure 4.6.

Definition 5.2.1 (Diagonal Elements)**.** *Two pixels $x_{i,j}$ and $x_{i',j'}$ are said to be secondary diagonal elements if $i + i' = j + j'$. For brevity, we refer to these elements from here on simply as Diagonal Elements.*

Theorem 3 proves that every element of on the diagonal can be computed parallelly and Line 2 of the algorithm takes care of that. We initialize X to Y in Line 1 and compute X in Line 8 which is given

in Equation 5.5. It is important that we wait for the threads to synchronize before we move to the next diagonal, as they are needed for computing the elements of the next diagonal. The *not out of bounds* in Line 7 means we are remaining in the $k \times k$ convolution window and also we are not including pixel (i, j) while computing $x_{i,j}$ as given in Equation 5.5

Theorem 3. *The inverse of the pixels on the diagonals of a TL padded convolution can be computed independently and parallelly.*

Proof. The $(i, j)^{th}$ pixel value of the output Y with shape $H \times W$ can be calculated as

$$y_{i,j} = (\mathbf{M}_{iW+j,:})^T \cdot x$$

which means $y_{i,j}$ is the dot product of $\mathbf{M}_{iW+j,:}$ i.e., the corresponding row of matrix $\mathbf{M}$ and the vectored input x. Because it is a *TL* padded convolution, $y_{i,j}$ depends only on the values of $k \times k$ window of $x_{\leq i,\leq j}$ pixels where $x_{\leq i,\leq j}$ are the pixels that are on the top and left side of the pixel $x_{i,j}$ including $x_{i,j}$. Because all the diagonal values are $w_{k,k}$, we have,

$$y_{i,j} = w_{k,k} x_{i,j} + f(x_{<i,<j})$$
$$x_{i,j} = \frac{y_{i,j} - f(x_{<i,<j})}{w_{k,k}}$$

where $x_{<i,<j}$ are the pixels which are strictly top and left side of (i, j). Following the masking pattern of CInC Flow, we have $w_{k,k} = 1$ and f is a linear function which is given by weighted sum of the given pixels weighed by the filter values. So,

$$x_{i,j} = y_{i,j} - f(x_{<i,<j}) \tag{5.4}$$

$$x_{i,j} = y_{i,j} - \sum_{p=0}^{k} \sum_{q=0}^{k} x_{i-p,j-q} K_{p,q} \text{ where } p = q \neq 0 \tag{5.5}$$

Let two pixels $x_{i,j}$ and $x_{i',j'}$ be on the same diagonal. This also means that only one of the following settings is true a) $i < i'$ and $j > j'$ or b) $i > i'$ or $j < j'$. Either way, we can conclude that computation of $x_{i,j}$ is not dependent on $x_{i',j'}$ and vice versa following the result in Equation 5.4. Hence they can be computed independently. Once $x_{i,j}$ is computed, following the Equation 5.4 and the above result, we can compute $x_{i+1,j}$ and $x_{i,j+1}$. Since, the sets of pixels $x_{<i+1,<j}$ and $x_{<i,<j+1}$ both include the

elements of $x_{<i,j}$ and also $x_{i,j}$, we can write

$$\begin{aligned} x_{i+1,j} &= y_{i+1,j} - f(x_{<i+1,<j}) \\ &= y_{i+1,j} - \alpha x_{i,j} - f_1(x_{<i,<j}) \qquad (5.6) \\ x_{i,j+1} &= y_{i,j+1} - f(x_{<i,<j+1}) \\ &= y_{i,j+1} - \beta x_{i,j} - f_2(x_{<i,<j}) \qquad (5.7) \end{aligned}$$

where α and β are kernel weights.

From Equations 5.6 and 5.7, we can conclude that $x_{i+1,j}$ and $x_{i,j+1}$ which are on the same diagonal can be calculated parallelly in a single step. □

Theorem 4. *Algorithm 1 uses only* $(H + W - 1)k^2$ *sequential operations.*

Proof. We have proved in Theorem 3 that the inverse pixels on a single diagonal can be computed parallelly in one iteration of Algorithm 1. Since there are $H + W - 1$ number of diagonals in a matrix and there are at maximum k^2 entries in a row of the convolutional matrix, the number of sequential operations needed will be $(H + W - 1)k^2$. □

Thus the running time of our algorithm is O(nk^2) where $n = H = W$

5.3 FInC Flow Unit

Figure 5.2 visualizes our $k \times k$ convolution block. We call this block as FInC Flow *Unit*. We use all the 4 padding techniques mentioned before to different channels of the image. For this purpose, we split the input into four equal parts along the channel axis. We do *TL* padding to the first part, *TR* to the second part, *BL* to the third part and *TR* to the fourth part. Then we use a masked filter on each of these parts to perform the convolution operation parallelly. We call each of this padded image along with it's corresponding kernel as Padded Convolution Block (PCB).

5.4 Architecture

Figure 5.3b shows the complete architecture of our model. Our model architecture resembles the architecture of Glow. The multi-scale architecture involves a block of a Squeeze layer, FInC Flow Step repeated K number of times and a Split layer. The whole block is repeated $L - 1$ number of times. A Squeeze layer follows this and finally FInC Flow Step repeated K times. At the end of each split layer, half of the channels are 'split' (taken away) and modeled as Gaussian distribution samples. These

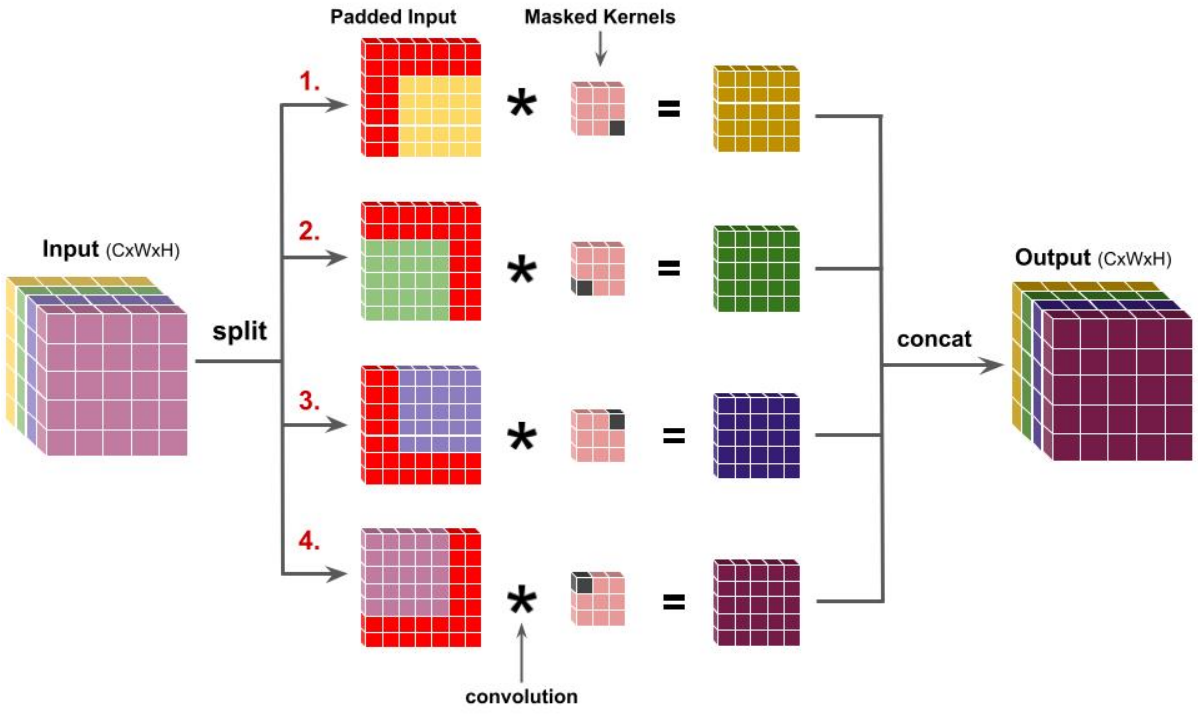
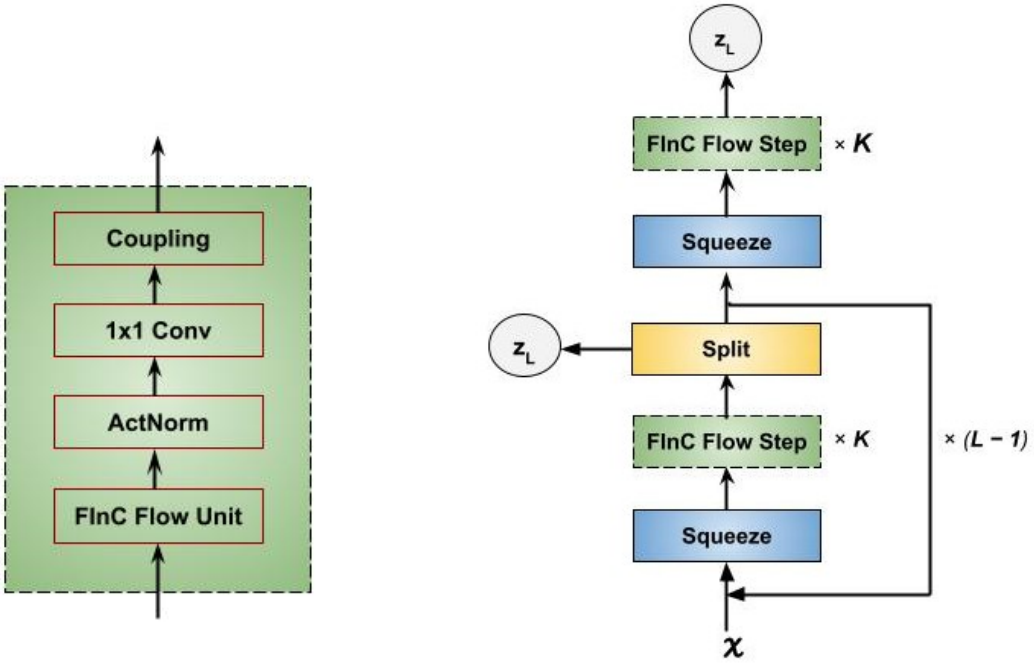


Figure 5.2: FInC Flow unit: to utilize the independence of convolution on channels the input channels are sliced into four equal parts and then padded (1. top-left, 2. top-right, 3. bottom-right, 4. bottom-left) to keep the input size and output size same. Next, parallelly convoluted each sliced channel with the corresponding masked kernel (masked corner of kernels: 1. bottom-right, 2. bottom-left, 3. top-left, 4. top-right). Finally, concatenate the output from each convolution.



(a) FInC Flow Step

(b) FInCFlow Multiscale architecture

Figure 5.3: FInC Flow Multiscale architecture: (a) shows a single FInC Flow Step. (b) shows the complete multiscale architecture. K and L represent the number of times corresponding block is repeated.

split half channels are *latent vectors*. The same is done for the output channels. These are denoted as z_L in Figure 5.3b. Each FInC Flow Step consists of a FInC Flow Unit, an Actnorm Layer, a 1×1 Convolutional Layer, followed by a coupling layer.

Chapter 6

Implementation of our Inverse Algorithm

In order to gain a comprehensive understanding of the implementation process, it is imperative to delve into a brief overview of the CUDA Architecture. Following that, we will proceed to discuss the implementation of the GPU Parallel Algorithm for Algorithm 1. Additionally, we will provide the implementation of Algorithm 2, which further enhances the speed of the inverse operation.

6.1 CUDA Architecture

To understand the implementation, we need to take a look at CUDA architecture. CUDA which stands for Compute Unified Device Architecture is a parallel computing platform and programming model created by NVIDIA Corporation. It is specifically designed for general-purpose computing on GPUs.

The architecture of CUDA is based on a hierarchy of parallel processing units, including threads, thread blocks, and grids. Threads are the smallest units of execution, and each thread is executed on a single core of the GPU. Multiple threads can be organized into a thread block, which can be thought of as a group of threads that can cooperate and communicate with each other. Multiple thread blocks can be organized into a grid, which is the largest unit of parallelism in CUDA. See Figure 6.1. In general, threads are organized into blocks, and blocks are organized into grids. The number of threads, blocks, and grids that can be used in a CUDA program is determined by the resources available on the GPU, such as the number of cores, amount of memory, and other hardware limitations.

The number of threads that can run parallelly is determined by the SMU - Streaming Multiprocessing Unit. An SMU is a physical processing unit within the GPU that is responsible for executing parallel computations. It consists of multiple processing cores, specialized units for arithmetic and logic operations, and a set of shared memory resources. Each SMU is capable of executing multiple threads in parallel, and can switch between different threads to maximize utilization of the processing resources. The number of SMUs in a GPU depends on the specific model and architecture of the GPU, and can range from a few to several dozen.

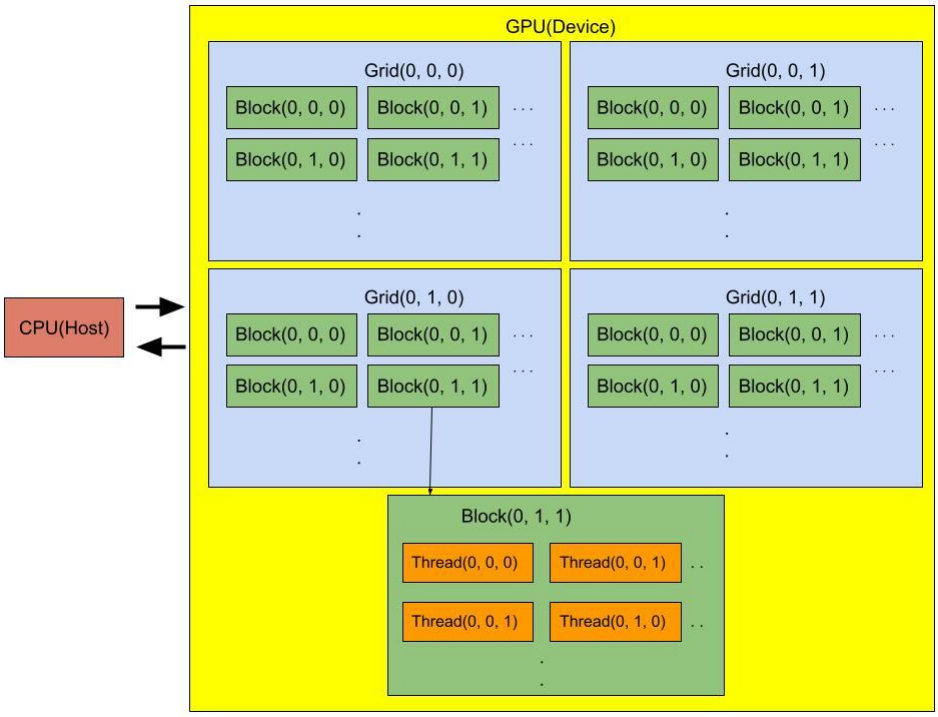


Figure 6.1: CUDA Architecture

6.2 CUDA Programming

CUDA programming is written in C/C++ language and compiled by NVCC - NVIDIA CUDA Compiler. A CUDA C/C++ code is executed on a GPU using a combination of the host CPU and the CUDA device. The CUDA programming model provides a set of APIs that allow the host CPU to launch parallel computations on the GPU, and to transfer data between the host and device memories.

A typical execution of a CUDA C/C++ code involves several steps like

1. Allocate memory on the host for input and output
2. Allocate memory on the device(GPU)
3. Copy data from host to device
4. Launch the kernel - CUDA function and execute it
5. Copy the results back to host
6. Free the memory on both the host and the device

```
int tid = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y * threadIdx.z);
```

Listing 1: Code to calculate global thread index within a block

```
int bid = blockIdx.x + gridDim.x * (blockIdx.y + gridDim.y * blockIdx.z);
```

Listing 2: Code to calculate global block index within a grid

6.3 Cuda code details

To run CUDA code, we use PyTorch-LTS 1.8.2 and cudatoolkit 10.2. Luckily for us, most of the points in the list 6 will be taken care of by PyTorch CUDA libraries. So, we only need 2 functions - 1) the function that we call from python code, 2) the CUDA kernel function.

We take advantage of the parallelization potential of multiple grids, blocks, and threads, noting that their indices can be organized in one-dimensional, two-dimensional, or three-dimensional arrays. In order to calculate the global index of a thread within a specific block, we need to implement the code provided in 1. Likewise, to obtain the global index of a block within a grid, the code in 2 needs to be implemented.

6.4 Implemetation of Algorithm 1

The main function is provided in Code 3, where we set block_x to be equal to the batch size and threads_x to 1024, as we found through empirical experimentation that using block_y or threads_y did not result in any noticeable speedup. However, not all 1024 threads are useful. The number of useful threads is determined by the number of pixels on the diagonal, which is calculated for each diagonal in lines 14-27 of Code 3. Then, the kernel function is called, and a portion of the kernel code is shown in Code 4, which implements Algorithm 1. Finally, before moving on to the next diagonal, we synchronize all the threads using the necessary CUDA function, which ensures that all threads have finished execution before proceeding to the next loop.

6.5 Faster Parallel Algorithm

While it is possible to implement Algorithm 2 to find the inverse of each padded block individually, an advantageous approach is to leverage the fact that all four Padded Conv Blocks are equivalent with proper flipping of padded inputs and kernels. See Figure 5.1. This can be achieved using Algorithm 2

```cpp
std::vector<torch::Tensor> cinc_cuda_inverse_level1(
    torch::Tensor input, // B, C, H, W
    torch::Tensor kernel, // C, C, K_H, K_W
    torch::Tensor output)  // B, C, H, W
{
    const auto B = output.size(0);
    const auto C = output.size(1);
    const auto H = output.size(2);
    const auto W = output.size(3);

    int n = H < W ? H : W; //samller dimension  -- min(H, W)
    int m = H > W ? H : W; //larger dimension   -- max(H, W)

    for (int d = 1; d <= H + W - 1; d++) { // Iterating over diagonal index
        for (int c = 0; c < C; c++) {
            // all elements of the dth diagonal computed in parallel
            int relevant_threads = d;          // For fixed number of threads, not
        ↪  all threads are going to be useful
            if (d > n) {
                if (d <= m) {
                    relevant_threads = n;
                } else {
                    relevant_threads = m + n - d; // equivalent to 2n -d if m == n
                }
            }

            dim3 threads(1024, 1, 1);          // Fix the number of threads
            dim3 blocks(B, 1, 1);              // Fix the number of blocks = Batch
        ↪  Size

            //Call Kernel Function
            AT_DISPATCH_FLOATING_TYPES(input.type(), "kernel_function", ([&] {
                cinc_cuda_inverse_kernel<scalar_t><<<blocks, threads>>>(
                    input, kernel, output, c, d, relevant_threads);
            }));

            // synchronize all threads
            cudaDeviceSynchronize();
        }
    }
  return {output};
}
```

Listing 3: Main function

```
// First we need to find the exact pixel if for which we are going
//to calculate the inverse using global thread id - tid as said in Code 1
if (d <= n) {
    h = d - 1 - tid;
    w = tid;
}
else {
    w = (d - n)  + tid;
    h = n - 1 - tid;
}
// compute entry of the output in the diagonal d assigned to this thread
output[b][c][h][w] = input[b][c][h][w]; // Initailization as shown in Algorithm 2


for (int k_h = 0; k_h < K_H; k_h++) {
    if (h - k_h < 0) break;
    for (int k_w = 0; k_w < K_W; k_w++) {
        if (w - k_w < 0) break;
        for (int k_c = 0; k_c < C; k_c++) {
            if (k_h == 0 && k_w == 0) {
                if (k_c == c) continue;
            }
            output[b][c][h][w] -= output[b][k_c][h - k_h][w - k_w] *
    ↪  kernel[c][k_c][K_H - k_h - 1][K_W - k_w - 1];
        }
    }
}

```

Listing 4: Kernel code - part of the function that calculates the output

on the GPU. The implementation of the algorithm involves several steps. First, Y is split into four parts along the channel dimension, and each split of Y and the corresponding kernels are flipped to match the TL-padding (Line 2). Next, the flipped Y and corresponding kernels are concatenated to obtain the final Y and K, respectively (Lines 3 and 4). Algorithm 1 is then applied to find the inverse. To obtain the correct X, the reverse process is followed (Lines 5, 6, 7, and 8).

During the implementation of this algorithm, we set grid_x to be equal to the batch size, grid_y to be equal to the number of channels (denoted as c), and threads_x to 1024. It's important to note that c is not the same as C, which represents the total number of channels in the input. The Python code shown in Code 5 provides the implementation of Algorithm 2, while Code 6 shows a portion of the kernel function that implements the inverse operation.

Algorithm 2: Fast Parallel Inversion Algorithm for FInC Flow *Unit*

Input: K_1, K_2, K_3, K_4 - Convolution Kernels of different PCB, Y - Output of the FInC Flow *Unit*

Result: X - Input to the FInC Flow Unit / Inverse of the FInC Flow Unit

1. $Y_1, Y_2, Y_3, Y_4 \leftarrow split(Y)$
2. Flip $Y_2, Y_3, Y_4, K_2, K_3, K_4$ (inplace) appropriately to match *TL* padding
3. $Y \leftarrow concat(Y_1, Y_2, Y_3, Y_4)$
4. $K \leftarrow concat(K_1, K_2, K_3, K_4)$
5. Apply Algorithm 1 with input K, Y to get X
6. $X_1, X_2, X_3, X_4 \leftarrow split(X)$
7. Flip X_2, X_3, X_4 appropriately to get the correct output
8. $X \leftarrow concat(X_1, X_2, X_3, X_4)$

6.6 Implementation of Algorithm 2

The Python code for flipping the input and kernels is demonstrated in Code 3. The CUDA code, which represents the inverse implementation, is displayed in Code 4. Please note that the code snippets provided are partial representation of the kernel function and do not encompass the complete implementation, including the main function. For the comprehensive implementation, please refer to our GitHub

```python
def reverse(self, x):
    k_tl = self.conv_tl.conv.weight.data
    k_tr = torch.flip(self.conv_tr.conv.weight.data, [3])
    k_bl = torch.flip(self.conv_bl.conv.weight.data, [2])
    k_br = torch.flip(self.conv_br.conv.weight.data, [2, 3])

    kernel = torch.cat([k_tl, k_tr, k_bl, k_br], dim=0)
    out_tl, out_tr, out_bl, out_br = torch.chunk(x, 4, dim=1)
    out_tr = torch.flip(out_tr, [3])
    out_bl = torch.flip(out_bl, [2])
    out_br = torch.flip(out_br, [2, 3])

    x = torch.cat([out_tl, out_tr, out_bl, out_br], dim=1)
    y = torch.zeros_like(x).to(x.device)
    y = cinc_cuda_level2.inverse(x, kernel, y)[0]

    out_tl, out_tr, out_bl, out_br = torch.chunk(y, 4, dim=1)
    out_tr = torch.flip(out_tr, [3])
    out_bl = torch.flip(out_bl, [2])
    out_br = torch.flip(out_br, [2, 3])

    y = torch.cat([out_tl, out_tr, out_bl, out_br], dim=1)

    return y

```

Listing 5: Python Code for Algorithm 2

repository. Code 5 givels lines 1, 2, 3, 4, 6, 7, 8 lines of Algorithm 2 and the heart of the algorithm i.e., line 5 is given by 6.

```
// First we need to find the exact pixel if for which we are going
//to calculate the inverse using global thread id - tid as said in Code 1
if (d <= n) {
    h = d - 1 - tid;
    w = tid;
}
else {
    w = (d - n)  + tid;
    h = n - 1 - tid;
}
// compute entry of the output in the diagonal d assigned to this thread
output[b][c + order * order_stride][h][w] = input[b][c + order *
         ↪  order_stride][h][w]; // Initailization as shown in Algorithm 2
for (int k_h = 0; k_h < K_H; k_h++) {
    if (h - k_h < 0) break;
    for (int k_w = 0; k_w < K_W; k_w++) {
        if (w - k_w < 0) break;
        for (int k_c = 0; k_c < order_stride; k_c++) {
            if (k_h == 0 && k_w == 0) {
                if (k_c == c) continue;

            }
            output[b][c + order * order_stride][h][w] -= output[b][k_c + order *
          ↪  order_stride][h - k_h][w - k_w] * kernel[c + order *
          ↪  order_stride][k_c][K_H - k_h - 1][K_W - k_w - 1];
        }
    }
}

```

Listing 6: Kernel code - part of the function that calculates the output

Chapter 7

Results and Discussions

Table 7.1 shows the comparative results of our model with other models. For MaCow, we use the official code released by the authors. We use the code for Emerging, which was implemented in PyTorch by the authors of SNF.. We have implemented CInC Flow in PyTorch and used it to generate results. The FTs and STs are recorded by averaging ten runs on untrained models (including our model).

7.1 Datasets

We train our model on standard benchmark datasets MNIST [48], CIFAR-10 [49], and ImageNet [50] sampled down to 32×32 and 64×64. We also train our model on CelebA [51] sampled down to 64×64. Figure 7.2a present the CelebA-64×64 reconstructed samples and Figure 7.3b for the CIFAR-10 and ImageNet-64x64 generated samples.

7.2 Bits Per Dimension (BPD):

BPD is closely related to NLLLoss given in equation 3.2. BPD of $H \times W \times C$ image is given by

$$\text{bpd} = \frac{\text{NLLLoss} \times \log_2 e}{HWC} \tag{7.1}$$

Table 7.1 shows the BPD comparative results of various models with our model. We present the results of MaCow-var which uses Variational Dequantization which was introduced in Flow++ [52]. BPDs recorded are the reported numbers from the respective model papers.

7.3 Sampling Time

In Figure 7.1, we plot the relationship between the input image size and inverse sampling time. As the input image size increase, our *Parallel Inversion Algorithm* improve by utilizing the independence

Model	MNIST			CIFAR-10			Imagenet-32x32			Imagenet-64x64		
	BPD	FT	ST	BPD	FT	ST	BPD	FT	ST	BPD	FT	ST
Emerging	–	0.16	0.62	3.34	0.49	17.19	4.09	0.73	25.79	3.81	1.71	137.04
MaCow	–	–	–	3.16	1.49	3.23	–	–	–	3.69	2.91	8.05
CInC Flow	–	–	–	3.35	0.42	7.91	4.03	0.62	11.97	3.85	1.57	55.71
MintNet	0.98	0.16	17.29	3.32	2.09	230.17	4.06	2.08	230.44	–	–	–
FInC Flow (our)	1.05	0.14	0.09	3.39	0.37	0.41	4.13	0.48	0.52	3.88	1.43	2.11

Table 7.1: Comparison of the bits per dimension (BPD), forward pass time (FT) and sampling time (ST) on standard benchmark datasets of various $k \times k$ convolution based Normalizing Flow models. FT and ST are presented in seconds.

Models	Setting (K and L)	Learnable params (M = million)	FT(n=100)	ST(n=100)
MaCow-FG	[4, [12, 12], [12, 12], 12]	37.19M	0.88	2.64
MaCow-org	[4, [12, 12], [12, 12], 12], [4, 4]	38.4M	1.48	3.23
FInC Flow (our)	[28, 28, 28]	39.46M	**0.37**	**0.41**

Table 7.2: CIFAR-10: comparison of learnable parameters and the sampling time. FInC Flow has less number of learnable parameters with the same receptive field and fast layers (all the times are averaged over ten loops for n = 100 sample images in seconds). ST = Sample time, FT = Forward Time. MaCow-FG is the fine-grained MaCow model and MaCow-org stands for MaCow model utilizes the original multi-scale architecture which is the same as Glow. MaCow and our methods are closely similar in term of the convolutional design. So, here we show that our proposed method do fast sampling while maintaining the faster forward time .

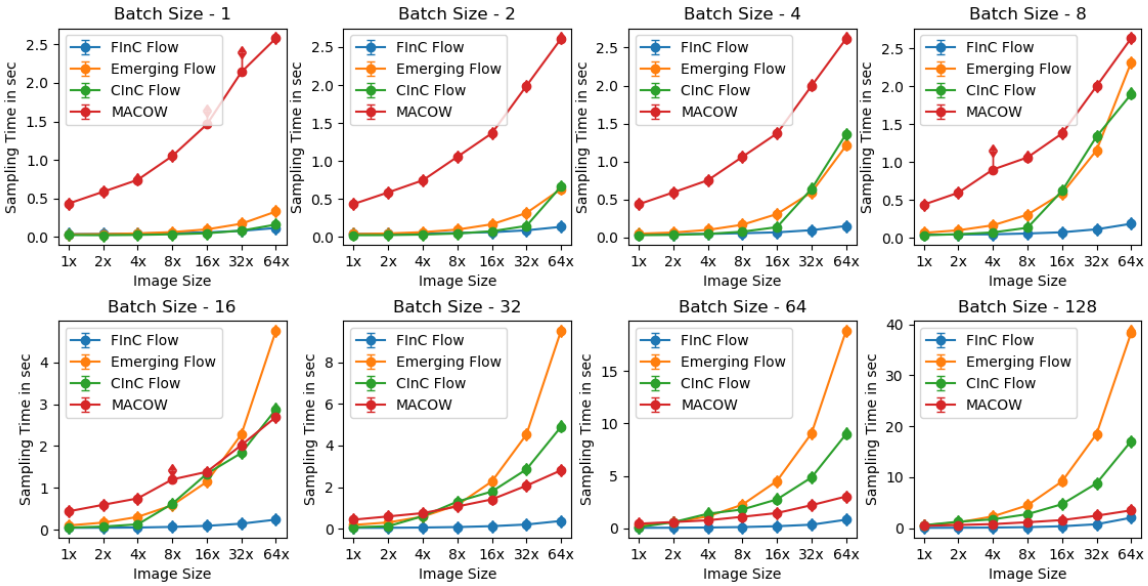
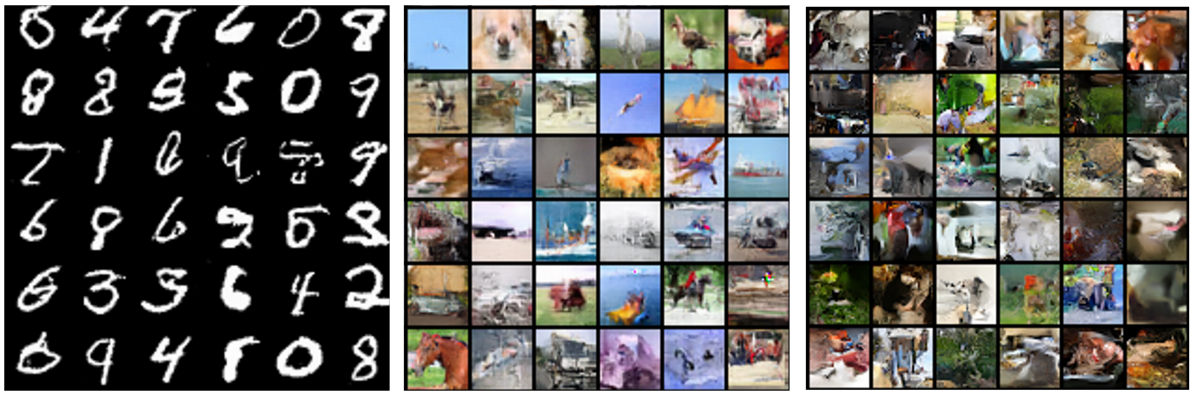


Figure 7.1: Sampling Times for four models - our, Emerging, CInC Flow, MaCow. Each plot gives the 95% Confidence Interval (CI) time of the ten runs to sample 100 images. X-axis represents the sizes of the image sampled starting from $16 \times 16 \times 2$ ($H \times W \times C$) all the way to $128 \times 128 \times 2$.

Figure 7.2: Comparison of (a) original and (b) reconstructed image samples for the 64×64 CelebA dataset after FInC Flow model for 100 epochs. From the images, we can conclude our model reconstructs the original image.

in the convolution matrix M. If we input a single image (batch size = 1), our model performs similarly to the CInC Flow and Emerging. MaCow is far slower because it does the masking of four kernels to maintain the receptive field. To do one convolution, it needs four convolutions to complete one standard convolution, making it slower. Emerging requires two consecutive autoregressive convolutions to have



(a) MNIST (b) CIFAR-10 (c) ImageNet-64x64

Figure 7.3: Uncurated generated samples images from our flow model.

the same receptive field as standard convolution and solver compared to FInC Flow. For batch size = 4 and larger, FInC Flow beats the Emerging, MaCow, and CInC Flow by a big difference (see Figure 7.1) while maintaining the same receptive field.

7.3.1 Scaling sampling time with spatial dimensions

Table 7.2 shows the comparison among the invertible convolution-based models. To keep it fair, we restrict the total parameters across all the models to be close to 5 M. We note down the average sampling time (ST) to generate 100 images over ten runs while doubling the size of the sampled image from 16×16 all the way to 128×128 and also doubling our batch size from 1 all the way to 128. Our model outperforms all the other models in most if not all, settings. All the models were untrained and run on a single NVIDIA GTX 1080Ti GPU.

7.3.2 ST Comparison of Algorithms

We compare the sampling times of different datasets while using Algorithm 1 and Algorithm 2. The results are shown in Table - 7.3. As with other cases, we record the timings by taking the average of 10 runs. The results clearly show that Algorithm 2 outperforms the Algorithm 1. It was expected as Algorith 2 makes use of parallel processing capabilities of modern GPUs.

7.4 Image reconstruction and generation

In Figure 7.2, we present the effectiveness of the FInC Flow model in the reconstruction (sampling) of the images. First, we feed the input image to forward flow and get the *latent vector* (z_L). To reconstruct the images from the *latent vector* (z_L), give the z as input to the inverse flow. Figure 7.2 present the reconstructed face images for the CelebA dataset after training our model for 100 epochs. The model

Dataset	(L, K)	#Params	#Samples	Algorithm 1 ST	Algorithm 2 ST
MNIST	(2, 16)	10M	1	0.09	0.07
MNIST	(2, 16)	10M	100	0.12	0.09
CIFAR10	(3, 32)	45M	1	0.73	0.30
CIFAR10	(3, 32)	45M	100	0.92	0.47
Imagenet32	(4, 48)	67M	1	1.01	0.44
Imagenet32	(4, 48)	67M	100	1.38	0.73
Imagenet64	(3, 32)	45M	1	1.42	0.50
Imagenet64	(3, 32)	45M	100	2.31	1.42

Table 7.3: Presents a comparison of the sampling times (STs) for various datasets using our Algorithms - 1, 2. The STs are given in seconds and represent the time taken to generate a certain number of samples, as indicated by the number of samples (#Samples) generated by the models with a specific number of parameters (#Params), which are defined by the parameters (L, K) for the respective dataset.

takes a random sample from the Gaussian distribution for the *latent vector* to generate sample images. This *latent vector* is used to generate images by going backward in the flow model. In Figure 7.3, we present generated sample by our model on the MNIST, CIFAR-10, and ImageNet-64x64 dataset.

7.5 Hyperparameters

To train our model, we use Adam optimizer [53] with learning rate of 0.001 with an exponential decay of 0.99997 per epoch. For training on Imagenet, we also make sure that the gradients stay between -1 and $+1$ by clipping them.

7.6 Masking

To make sure the masked values in the Padded Conv Blocks, we ensure they are not affected by back propagation. To achieve this, we reset the gradients of the masked values to zero after every training iteration.

7.7 Running MaCow, Emerging, CInC Flow, SNF

For MaCow and SNF, we use the official code released by the authors. Emerging was implemented in PyTorch by the authors of SNF. We make use of that. We have implemented CInC Flow on PyTorch to get the results.

7.8 Computing Run-time and Confidence Intervals

We run the model (both forward and sampling) 11 times and ignore the 1^{st} run as it includes the initialization time. We calculate the mean, standard deviation and 95% confidence interval and plot the numbers. To calculate forward time, we pass 100 images, as for sampling times, we sample 100 images. We present these numbers in Table-7.2. For sampling time comparison of different models shown in Figure 7.1, we set the total number of parameters for all the models to be close to $5\mathbf{M}$ to make it a fair comparison.

7.9 Hardware/Training Time

Our hardware setup consists of Intel Xeon E5-2640 v4 processor providing 40 cores, 80 GB of DDR4 RAM, 4 Nvidia GeForce GTX 1080 Ti GPUs each with 12 GB of VRAM. We train our model on all GPUs using PyTorch's Data Parallel class. We implement early stopping mechanism for smaller datasets like MNIST, CIFAR-10. For others we train the model for a maximum epochs. To evaluate Forward Time and Sampling Time, we use only one of the GPUs.We evaluate our FInC Flow model on MNIST, CIFAR-10, Imagenet-32x32, and Imagenet-64x64 datasets for three metrics - (a) Loss expressed in Bits per Dimension (BPD), (b) Forward Pass Time (FT): Time taken for 100 images to be passed through the model and (c) Sampling Time(ST): Time taken by the model to generate 100 images. To do this, we train our model with Adam Optimizer with a learning rate (lr) of 0.001 and exponentially reduce the lr by 0.99997 after each epoch. We have used 4 NVIDIA GTX 1080 Ti GPUs to train our model. Evaluation(FT/ST) is done on a single GPU.

Chapter 8

Conclusions

With a parallel inversion approach, we present a $k \times k$ invertible convolution for Normalizing flow models. We utilize it to develop a model with highly efficient sampling pass, normalizing flow architecture. We implement our parallel algorithm on GPU and presented benchmarking results, which show a significant enhancement in forward and sampling speeds when compared to alternative methods for $k \times k$ invertible convolution.

Related Publications

Kallappa, A.; Nagar, S. and Varma, G. (2023). FInC Flow: Fast and Invertible k × k Convolutions for Normalizing Flows. In Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 5: VISAPP, ISBN 978-989-758-634-7; ISSN 2184-4321, pages 338-348. DOI: 10.5220/0011876600003417

Bibliography

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

[2] X. Ma, X. Kong, S. Zhang, and E. Hovy, "Macow: Masked convolutional generative flow," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[3] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*. PMLR, 2021, pp. 8821–8831.

[4] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes *et al.*, "Photorealistic text-to-image diffusion models with deep language understanding," *arXiv preprint arXiv:2205.11487*, 2022.

[5] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 684–10 695.

[6] J. Oppenlaender, "The creativity of text-based generative art," *arXiv preprint arXiv:2206.02904*, 2022.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[8] I. Kobyzev, S. J. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, no. 11, pp. 3964–3979, 2020.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[10] D. P. Kingma, M. Welling *et al.*, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[11] D. P. Kingma and P. Dhariwal, "Glow: Generative flow with invertible 1x1 convolutions," *Advances in neural information processing systems*, vol. 31, 2018.

[12] S. Nagar, M. Dufraisse, and G. Varma, "CInc flow: Characterizable invertible 3$\times$3 convolution," in *The 4th Workshop on Tractable Probabilistic Modeling*, 2021. [Online]. Available: https://openreview.net/forum?id=kl1ds_AeLRM

[13] R. Prenger, R. Valle, and B. Catanzaro, "Waveglow: A flow-based generative network for speech synthesis," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 3617–3621.

[14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[15] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015. [Online]. Available: https://arxiv.org/abs/1511.06434

[16] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014. [Online]. Available: https://arxiv.org/abs/1411.1784

[17] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," 2017. [Online]. Available: https://arxiv.org/abs/1703.10593

[18] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," 2018. [Online]. Available: https://arxiv.org/abs/1809.11096

[19] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," 2018. [Online]. Available: https://arxiv.org/abs/1812.04948

[20] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *Advances in neural information processing systems*, vol. 29, 2016.

[21] D. Bau, J.-Y. Zhu, H. Strobelt, B. Zhou, J. B. Tenenbaum, W. T. Freeman, and A. Torralba, "Gan dissection: Visualizing and understanding generative adversarial networks," *arXiv preprint arXiv:1811.10597*, 2018.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[23] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proceedings of ICML workshop on unsupervised and transfer learning*. JMLR Workshop and Conference Proceedings, 2012, pp. 37–49.

[24] J. Zhao, Y. Kim, K. Zhang, A. Rush, and Y. LeCun, "Adversarially regularized autoencoders," in *International conference on machine learning*. PMLR, 2018, pp. 5902–5911.

[25] A. Makhzani and B. Frey, "K-sparse autoencoders," *arXiv preprint arXiv:1312.5663*, 2013.

[26] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1096–1103.

[27] Z. Huang, X. Jin, C. Lu, Q. Hou, M.-M. Cheng, D. Fu, X. Shen, and J. Feng, "Contrastive masked autoencoders are stronger vision learners," *arXiv preprint arXiv:2207.13532*, 2022.

[28] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[29] Y. Pu, Z. Gan, R. Henao, X. Yuan, C. Li, A. Stevens, and L. Carin, "Variational autoencoder for deep learning of images, labels and captions," *Advances in neural information processing systems*, vol. 29, 2016.

[30] O. Ivanov, M. Figurnov, and D. Vetrov, "Variational autoencoder with arbitrary conditioning," *arXiv preprint arXiv:1806.02382*, 2018.

[31] K. Ghasedi Dizaji, A. Herandi, C. Deng, W. Cai, and H. Huang, "Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5736–5745.

[32] B. Zong, Q. Song, M. R. Min, W. Cheng, C. Lumezanu, D. Cho, and H. Chen, "Deep autoencoding gaussian mixture model for unsupervised anomaly detection," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJJLHbb0-

[33] C. K. Sønderby, T. Raiko, L. Maaløe, S. K. Sønderby, and O. Winther, "Ladder variational autoencoders," *Advances in neural information processing systems*, vol. 29, 2016.

[34] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," in *International Conference on Machine Learning*. PMLR, 2015, pp. 2256–2265.

[35] Y. Song and S. Ermon, "Generative modeling by estimating gradients of the data distribution," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/3001ef257407d5a371a96dcd947c7d93-Paper.pdf

[36] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in Neural Information Processing Systems*, vol. 33, pp. 6840–6851, 2020.

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

[38] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," *In International Conference on Learned Representations*, 2017.

[39] Y. Song, C. Meng, and S. Ermon, "Mintnet: Building invertible neural networks with masked convolutions," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

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

[41] T. A. Keller, J. W. Peters, P. Jaini, E. Hoogeboom, P. Forré, and M. Welling, "Self normalizing flows," in *International Conference on Machine Learning*. PMLR, 2021, pp. 5378–5387.

[42] Y. Lu and B. Huang, "Woodbury transformations for deep generative flows," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5801–5811, 2020.

[43] C. Meng, L. Zhou, K. Choi, T. Dao, and S. Ermon, "Butterflyflow: Building invertible layers with butterfly matrices," in *International Conference on Machine Learning*. PMLR, 2022, pp. 15 360–15 375.

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

[45] A. Van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves *et al.*, "Conditional image generation with pixelcnn decoders," *Advances in neural information processing systems*, vol. 29, 2016.

[46] A. Van Den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," in *International conference on machine learning*. PMLR, 2016, pp. 1747–1756.

[47] G. Papamakarios, T. Pavlakou, and I. Murray, "Masked autoregressive flow for density estimation," *Advances in neural information processing systems*, vol. 30, 2017.

[48] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[49] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 2012.

[50] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[51] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 3730–3738.

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

[53] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980