Parallel Backpropagation for Inverse of a Convolution with Application to Normalizing Flows

Sandeep Nagar
ML Lab, IIIT Hyderabad, India

Girish Varma
ML Lab, IIIT Hyderabad, India

Abstract

Inverse of an invertible convolution is an important operation that comes up in Normalizing Flows, Image Deblurring, etc. The naive algorithm for backpropagation of this operation using Gaussian elimination has running time $O(n^3)$ where n is the number of pixels in the image. We give a fast parallel backpropagation algorithm with running time $O(\sqrt{n})$ for a square image and provide a GPU implementation of the same. Inverse Convolutions are usually used in Normalizing Flows in the sampling pass, making them slow. We propose to use Inverse Convolutions in the forward (image to latent vector) pass of the Normalizing flow. Since the sampling pass is the inverse of the forward pass, it will use convolutions only, resulting in efficient sampling times. We use our parallel backpropagation algorithm for optimizing the inverse convolution layer resulting in fast training times also. We implement this approach in various Normalizing Flow backbones, resulting in our Inverse-Flow models. We benchmark Inverse-Flow on standard datasets and show significantly improved sampling times with similar bits per dimension compared to previous models.

1 Introduction

Large-scale neural network optimization using gradient descent is made possible due to efficient and parallel back-propagation algorithms [Bottou, 2010]. Large models could not be trained on large datasets without such fast back-propagation algorithms. All operations for building practical neural network models need efficient back-propagation algorithms [LeCun et al., 2002]. This has limited types of operations that can be used to build neural networks. Hence, it is important to design fast parallel backpropagation algorithms for novel operations that could make models more efficient and expressive.

Convolutional layers are very commonly used in Deep Neural Network models as they have fast parallel forward and backward pass algorithms [LeCun et al., 2002]. Inverse of a convolution is a closely related operation with use cases in Normalizing Flows [Karami et al., 2019], Image Deblurring [Eboli et al., 2020], Sparse Blind Deconvolutions [Xu et al., 2014], Segmentation, etc. However, Inverse of a Convolution is not used directly as a layer for these problems since straightforward algorithms for backpropagation of such layers are highly inefficient. Such algorithms involve computing inverse of a very large dimensional matrix.

Fast sampling is crucial for Normalizing flow models in various generative tasks due to its impact on practical applicability and real-time performance [Papamakarios et al., 2021]. Rapidly producing high-quality samples is essential for large-scale data generation and efficient model evaluation in fields such as image generation, molecular design [Zang and Wang, 2020], image deblurring, and deconvolution. Normalizing flows have demonstrated their capability in constructing high-quality images [Kingma and Dhariwal, 2018, Meng et al., 2022]. However, training and sampling process is computationally expensive due to repeated need for inverting functions (e.g., convolutions). Existing approaches rely on highly constrained architectures and often impose limitations like diagonal, triangular, or low-rank Jacobian matrices and approximate inversion methods [Hoogeboom et al., 2019, Keller et al., 2021]. These constraints restrict expressiveness and efficiency of normalizing flow models. To overcome these limitations, fast, efficient, and parallelizable algorithms are needed to compute inverse of convolutions and their backpropagation, along with GPU-optimized imple-

Preprint Proceedings of the 28th International Conference on Artificial Intelligence and Statistics (AISTATS) 2025, Mai Khao, Thailand. PMLR: Volume 258.

(a) Original (b) Reconstructed

Figure 1: a). Images from MNIST dataset. b). Reconstructed images using an Inverse-Flow model based on inv-conv layer for a forward pass.

mentations. Addressing these challenges would significantly enhance performance and scalability of Normalizing flow models.

In this work, we propose a fast, efficient, and parallelized backpropagation algorithm for inverse of convolution with running time $O(mk^2)$ on an $m \times m$ input image. We provide a parallel GPU implementation of proposed algorithm (together with baselines and experiments) in CUDA. Furthermore, we design *Inverse-Flow*, using an inverse of convolution (f^{-1}) in forward pass and convolution (f) for sampling. Inverse-flow models generate faster samples than standard Normalizing flow models.

In summary, our contribution includes:

1. We designed a fast and parallelized backpropagation algorithm for inverse of convolution operation.

2. Implementation of proposed backpropagation algorithm for inverse of convolution on GPU CUDA.

3. We propose a multi-scale flow architecture, *Inverse-Flow*, for fast training of inverse of convolution using our efficient backpropagation algorithm and faster sampling with $k \times k$ convolution.

4. Benchmarking of *Inverse-Flow* and a small linear, 9-layer flow model on image dataset (MNIST, CIFAR10).

2 Related work

Backpropagation for Inverse of Convolution Backpropagation algorithm performs stochastic gradient descent and effectively trains a feed-forward neural network to approximate a given continuous function over a compact domain. [Hoogeboom et al., 2019] proposed invertible convolution, Emerging, generalizing 1x1 convolution from Glow [Kingma and Dhariwal, 2018]. [Finzi et al., 2019] proposed periodic convolution with $k \times k$ kernels. Emerging convolution combines two autoregressive convolutions [Kingma et al., 2016], and parallelization is not possible for its inverse. MaCow [Ma et al., 2019] uses four masked convolutions in an autoregressive fashion to get a receptive field of 3×3 standard convolution, which leads to slow sampling and training. To best of our knowledge, this work is first to propose a backpropagation algorithm for inverse of convolution. Additionally, it is first to utilize an inverse Normalizing flow for training and a standard flow for sampling, marking a novel approach in field.

Normalizing flows (NF) NF traditionally relies on invertible specialized architectures with manageable Jacobian determinants [Keller et al., 2021]. One body of work builds invertible architectures by concatenating simple layers (coupling blocks), which are easy to invert and have a triangular Jacobian Nagar et al. [2021]. Many choices for coupling blocks have been proposed, such as MAF [Papamakarios et al., 2017], RealNVP [Dinh et al., 2016], Glow [Kingma and Dhariwal, 2018], Neural Spline Flows [Durkan et al., 2019]. Self Normalizing Flow (SNF) [Keller et al., 2021] is a flexible framework for training NF by replacing expensive terms in gradient by learning approximate inverses at each layer. Several types of invertible convolution emerged to enhance expressiveness of NF. Glow has stood out for its simplicity and effectiveness in density estimation and high-fidelity synthesis.

Autoregressive [Kingma et al., 2016] propose an inverse autoregressive flow and scale well to high dimensions latent space, which is slow because of its autoregressive nature. Papamakarios et al. [2017] introduced NF for density estimation with masked autoregressive. Sample generation from autoregressive flows is inefficient since inverse must be computed by sequentially traversing through autoregressive order [Ma et al., 2019]

Invertible Neural Network [Dinh et al., 2016] proposed Real-NVP, which uses a restricted set of non-volume preserving but invertible transformations. [Kingma and Dhariwal, 2018] proposed Glow, which generalizes channel permutation in Real-NVP with 1×1 convolution. However, these NF-based generative models resulted in worse sample generation compared to state-of-the-art autoregressive models and are incapable of realistic synthesis of large images compared to GANs [Brock, 2018] and Diffusion Models. CInC Flow [Nagar et al., 2021] proposed a fast convolution

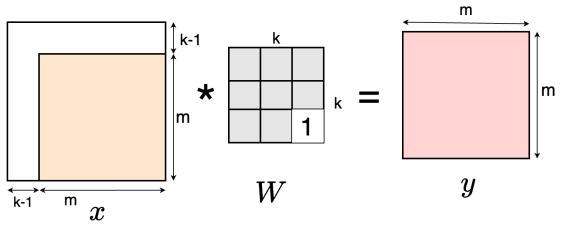

Figure 2: Invertible convolution with zero padding (top, left) on input x and masking of kernel $W_{k,k} = 1$

layer for NF. ButterflyFlow [Meng et al., 2022] leverage butterfly layers for NF models. FInC Flow [Kallappa. et al., 2023] leverages advantage of parallel computation for inverse of convolution and proposed efficient parallelized operations for finding inverse of convolution layers and achieving $O(n \times k^2)$. We designed a backpropagation algorithm for inverse of convolution layers. Then, multi-scale architecture, Inverse-Flow, is designed using an inverse of convolution for forward pass and convolution for sampling pass and backward pass.

Sampling Time NF requires large and deep architectures to approximate complex target distributions [Cornish et al., 2020] with arbitrary precision. Jung et al. [2024] present importance of fast sampling for Normalizing flow models. To model distribution using NF models requires inverse of a series of functions, backward pass, which is slow. This creates a limitation of slow sample generation. To address this, we propose Inverse-Flow, which uses convolution (fast parallel operation, $O(k^2)$, $k \times k$= kernel size) for a backward pass and inverse of convolution for a forward pass.

3 Fast Parallel Backpropagation for Inverse of a Convolution.

We assume that input/output of a convolution is an $m \times m$ image, with channel dimension assumed to be 1 for simplicity. The algorithm can naturally be extended to any number of channels. We also assume that input to convolution is padded on top and left sides with $k-1$ zeros, where k is a kernel size; see Figure 2. Furthermore, we assume that bottom right entry of convolution kernel is 1, which ensures that it is invertible.

The convolution operation is a Linear Operator (in Linear Algebraic terms; see Figure 2) on space of $m \times m$ matrices. Considering this space as column vectors of dimension m^2, this operation corresponds to multiplication by a $m^2 \times m^2$ dimensional matrix. Hence inverse of convolution is also a linear operator represented by a $m^2 \times m^2$ dimensional matrix. Suppose vectorization of $m \times m$ matrix to m^2 is done by row-major ordering; diagonal entries of Linear Operator matrix will be the bottom right entry of kernel, which we have assumed to be 1.

While convolution operation has fast parallel forward and backpropagation algorithms with running time $O(k^2)$ (assuming there are $O(m^2)$ parallel processors), a naive approach for inverse of convolution using Gaussian Elimination requires $O(m^6)$. [Kallappa. et al., 2023] gave a fast parallel algorithm for inverse of convolution with running time $O(mk^2)$. In this section, we give a fast parallel algorithm for backpropagation of inverse of convolution (inv-conv) with running time $O(mk^2)$ (see Table 1). Our backpropagation algorithm allows for efficient optimization of inverse of convolution layers using gradient descent.

Table 1: Running times of algorithms for Forward and Backward passes assuming there are enough parallel processors as needed. The forward pass algorithm for Inverse of Convolution was improved by [Kallappa. et al., 2023]. In this work, we give an efficient backward pass algorithm for Inverse of Convolution.

Layer	Forward	Backpropagation
Std. Conv.	$O(k^2)$	$O(k^2)$
inv-conv (naive)	$O((m^2)^3)$	$O((m^2)^3)$
inv-conv.	$O(mk^2)$	$\mathbf{O(mk^2)}$

Notation: We will denote input to inverse of convolution (inv-conv) by $y \in \mathbb{R}^{m^2}$ and output to be $x \in \mathbb{R}^{m^2}$. We will be indexing x, y using $p = (p_1, p_2) \in \{1, \cdots, n\} \times \{1, \cdots, n\}$. We define

$$\Delta(p) = \{(i,j) : 0 \leq p_1 - i, p_2 - j < k\} \setminus \{p\}.$$

$\Delta(p)$ informally is set of all pixels except p in the input which depend on p in the output, when convolution is applied with top, left padding. We also define a partial ordering $\leq$ on pixels as follows

$$p \leq q \quad \Leftrightarrow \quad p_1 \leq q_1 \text{ and } p_2 \leq q_2.$$

The kernel of $k \times k$ convolution is given by matrix $W \in \mathbb{R}^{k \times k}$. For backpropagation algorithm for inv-conv, input is

$$x \in \mathbb{R}^{m^2} \text{ and } \frac{\partial L}{\partial x} \in \mathbb{R}^{m^2},$$

where L is loss function. We can compute y on $O(m^2k^2)$ time using parallel forward pass algorithm of [Kallappa. et al., 2023]. The output of backpropagation algorithm is

$$\frac{\partial L}{\partial y} \in \mathbb{R}^{m^2} \text{ and } \frac{\partial L}{\partial W} \in \mathbb{R}^{k^2}$$

which we call input and weight gradient respectively. We provide the algorithm for computing these in the next 2 subsections.

3.1 Computing Input Gradients

Since y is input to inv-conv and x is output, $y = \text{conv}_W(x)$ and we get following m^2 equations by definition of convolution operation.

$$y_p = x_p + \sum_{q \in \Delta(p)} W_{(k,k)-p+q} \cdot x_q \tag{1}$$

Using chain rule of differentiation, we get that

$$\frac{\partial L}{\partial y_p} = \sum_q \frac{\partial L}{\partial x_q} \times \frac{\partial x_q}{\partial y_p}. \tag{2}$$

Hence if we find $\frac{\partial x_q}{\partial y_p}$ for every pixels p, q, we can compute $\frac{\partial L}{\partial y_p}$ for every pixel p.

Theorem 1.

$$\frac{\partial x_q}{\partial y_p} = \begin{cases} 1 - \sum_{q \in \Delta(p)} W_{(k,k)-p+q} \cdot \frac{\partial x_q}{\partial y_p} & \text{if } p = q \\ 0 & \text{if } q \not\leq p \\ -\sum_{r \in \Delta(p)} W_{(k,k)-r} \frac{\partial x_{p-r'}}{\partial y_p} & \text{otherwise.} \end{cases}$$

Formal proofs are deferred to the supplementary.

Informal proof: The theorem presents computing $\frac{\partial x_q}{\partial y_p}$, which represents how a change in input pixel y_p affects output pixel x_q in an inverse of convolution operation. Let's break down each case:

Case 1: When $p = q$, take partial derivative with respect to y_p on both sides of Equation 1 and rearranging.

$$\frac{\partial x_p}{\partial y_p} = 1 - \sum_{q \in \Delta(p)} W_{(k,k)-p+q} \cdot \frac{\partial x_q}{\partial y_p}$$

So if $\frac{\partial x_q}{\partial y_p}$ is known for all $q \leq p$, we can compute $\frac{\partial x_p}{\partial y_p}$. Since the off-diagonal entries are unrelated in the $\leq$ partial order, we can compute all of them in parallel, provided the previous off-diagonal entries are known.

Case 2: From Equation 1, when $q \not\leq p$, we have: $\frac{\partial x_q}{\partial y_p} = 0$. This case uses partial ordering defined earlier. If q is not less than or equal to p in this ordering, it means that output pixel x_q is not influenced by input pixel y_p in inverse of convolution operation 1. Therefore, derivative is 0.

Case 3: For all other cases:

$$\frac{\partial x_q}{\partial y_p} = -\sum_{r \in \Delta(p)} W_{(k,k)-r} \frac{\partial x_{p-r'}}{\partial y_p}$$

- $\Delta(p)$ is set of all pixels (except p) that depend on p in a regular convolution operation.
- $W_{(k,k)-r}$ represents weight in convolution kernel corresponding to relative position of r.
- $\frac{\partial x_{p-r'}}{\partial y_p}$ is a recursive term, representing how changes in y_p affect x at a different position.

The negative sign and summation in this formula account for inverse nature of operation and cumulative effects of convolution kernel.

3.2 Computing Weight Gradients

From Equation 1, we can say computing gradient of loss L with respect to weights W involves two key factors. Direct influence: how a specific weight W_a in convolution kernel directly affects output x pixels, and Recursive Influence: how neighboring pixels, weighted by kernel, indirectly influence output x during convolution operation. Similarly, to compute gradient of loss L w.r.t filter weights W, we apply chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial x} * \frac{\partial x}{\partial W} \tag{3}$$

where: $\frac{\partial L}{\partial x}$ is gradient of loss with respect to output x and convolution operation is applied between $\frac{\partial L}{\partial x}$ and output x. Computing gradient of loss L with respect to convolution filter weights W is important in backpropagation when updating convolution kernel during training. Similarly, $\partial L/\partial W$ can be calculated as 3 and $\partial x/\partial W$ can be calculated as (2) for each $k_{i,j}$ parameter by differentiating 1 w.r.t W:

$$\frac{\partial L}{\partial W_a} = \sum \frac{\partial L}{\partial x_q} * \frac{\partial x_q}{\partial W_a} \tag{4}$$

Equation 4 states that to compute gradient of loss with respect to each weight W_a, we need to:

- Compute how loss L changes with respect to each output pixel x_q (denoted by $\frac{\partial L}{\partial x_q}$).
- Multiply this by gradient of each output pixel x_q with respect to weight W_a (denoted by $\frac{\partial x_q}{\partial W_a}$)

We then sum over all output pixels x_q.

Theorem 2.

$$\frac{\partial x_q}{\partial W_a} = \begin{cases} 0 & \text{if } q \leq a \\ -\sum_{q' \in \Delta q(a)} W_{q'-a} \cdot \frac{\partial x_{q-q'}}{\partial W_a} - x_{q-a} & \text{if } q > a \end{cases}$$

Formal proofs are deferred to the supplementary.

Informal proof: Computation of $\partial x_q / \partial W_a$ depends on relative positions of pixel q and kernel weight index.

Case 1: When $a \leq q$, if index of weight matches index of output pixel 1, gradient is 0. This means that weight does not directly influence corresponding pixel in this case.
Case 2: When $q > a$, gradient is computed recursively by summing over neighboring pixel positions q' in convolution window. In this case, $q' \in \Delta q(a)$ represents pixels within the kernel's influence around pixel q that specifically correspond to weight W_a, meaning pixels whose relative position to q makes them affected by the particular weight W_a during the convolution operation. The convolution kernel weights $W_{q'-a}$ and shifted pixels value x_{q-a} are used to calculate gradients. See the Supplement section for more elaborated proof.

3.3 Backpropagation Algorithm for Inverse of Convolution

The backpropagation algorithm for inverse of convolution (inv-conv) computes gradients necessary for training models that use inv-conv operation for a forward pass. Our proposed algorithm 1 efficiently calculates gradients with respect to both input ($\frac{\partial L}{\partial Y}$) and convolution kernel ($\frac{\partial L}{\partial K}$) using a parallelized GPU approach.

Given gradient of loss L with respect to output ($\frac{\partial L}{\partial X}$), algorithm updates input gradient $\frac{\partial L}{\partial Y}$ by accumulating contributions from each pixel in output, weighted by corresponding kernel values. Simultaneously, kernel gradient $\frac{\partial L}{\partial K}$ is computed by accumulating contributions from spatial interactions between input and output. The process is parallelized across multiple threads, with each thread handling updates for different spatial and channel indices, ensuring efficient execution. This approach ensures that both input and kernel gradients are computed in a time-efficient manner, making it scalable for high-dimensional inputs and large kernels. A fast algorithm is key for enabling gradient-based optimization in models involving inverse of convolution.

Complexity of Algorithm 1: This computes $\frac{\partial L}{\partial y}$ and $\frac{\partial L}{\partial w}$ in $O(mk^2)$ utilizing independence of each diagonal of output x and sequencing of m diagonals. Diagonals are processed sequentially, but elements within each diagonal are processed in parallel. Each diagonal computation takes $O(k^2)$ time due to $k \times k$ kernel. This results in a time complexity of total $O(mk^2)$ and represents a substantial improvement over naive $O(m^6)$ approach. It makes algorithm highly efficient and practical for use in deep learning models with inverse of convolution layers, even for large input sizes or kernel sizes.

Algorithm 1: Backpropagation Algorithm for Inverse of Convolution (Input and Weight Gradients)

Input: K: Kernel of shape (C, C, k_H, k_W)
Y: output of conv of shape (C, H, W)
$\frac{\partial L}{\partial X}$: gradient of shape (C, H, W)
Output: $\frac{\partial L}{\partial Y}$: gradient of shape (C, H, W)
$\frac{\partial L}{\partial K}$: gradient of shape (C, C, k_H, k_W)

Initialization:
$\frac{\partial L}{\partial Y} \leftarrow 0$ (initialize input gradient to zero)
$\frac{\partial L}{\partial K} \leftarrow 0$ (initialize kernel gradient to zero)
for $d \leftarrow 0, H + W - 1$ **do**
  for $c \leftarrow 0, C - 1$ **do**
    `/* The below lines of code are executed parallelly on different threads on GPU for every index` (c, h, w) `on` d`th diagonal. */`
    for $k_h \leftarrow 0, k_H - 1$ **do**
      for $k_w \leftarrow 0, k_W - 1$ **do**
        for $k_c \leftarrow 0, C - 1$ **do**
          if *pixel* $(k_c, h - k_h, w - k_w)$ *not out of bounds* **then**
            `/* Compute input gradient for every pixel` (c, h, w)`: */`
            $\frac{\partial L}{\partial Y}[c, h, w] \leftarrow \frac{\partial L}{\partial Y}[c, h, w] + \frac{\partial L}{\partial X}[c, h, w] \cdot K[c, k_c, k_H - k_h - 1, k_W - k_w - 1]$
            `/* Compute kernel gradient: */`
            $\frac{\partial L}{\partial K}[c, k_c, k_h, k_w] \leftarrow \frac{\partial L}{\partial K}[c, k_c, k_h, k_w] + \frac{\partial L}{\partial X}[c, h, w] \cdot X[k_c, h - k_h, w - k_w]$
          end
        end
      end
    end
    `/* synchronize all threads */`
  end
end
return $\frac{\partial L}{\partial Y}, \frac{\partial L}{\partial K}$

4 Normalizing Flows

Normalizing flows are generative models that enable exact likelihood evaluation. They achieve this by transforming a base distribution into a target distribution using a series of invertible functions.

Let $\mathbf{z} \in \mathcal{Z}$ be a random variable with a simple base distribution $p_Z(\mathbf{z})$ (e.g., a standard Gaussian). A Normalizing flow transforms $\mathbf{z}$ into a random variable $\mathbf{y} \in \mathcal{Y}$ with a more complex distribution $p_Y(\mathbf{y})$ through

a series of invertible transformations: $\mathbf{y} = f(\mathbf{z}) = f_1(f_2(\cdots f_K(\mathbf{z})))$. Probability density of transformed variable $\mathbf{y}$ can be computed using change-of-variables formula:

$$p_Y(\mathbf{y}) = p_Z(\mathbf{z}) \left| \det \frac{\partial f^{-1}}{\partial \mathbf{y}} \right| = p_Z(f^{-1}(\mathbf{y})) \left| \det \frac{\partial f}{\partial \mathbf{z}} \right|^{-1}, \quad (5)$$

where $\left|\det \frac{\partial f}{\partial \mathbf{z}}\right|$ is absolute value of determinant of Jacobian of f.

This relationship (5) can be modeled as $y = f_\theta(z)$ called change of variable formula, where θ is a set of learnable parameters. This formula enables us to compute likelihood of y as:

$$\log p_Y(y) = \log p_Z(f_\theta(y)) + \log \left| \det \left(\frac{\partial f_\theta(y)}{\partial y} \right) \right|, \quad (6)$$

where second term, $\log \left|\det \left(\frac{\partial f_\theta(y)}{\partial y}\right)\right|$, is log-determinant of Jacobian matrix of transformation f_θ. This term ensures volume changes induced by transformation are properly accounted for in likelihood. For invertible convolutions, which are a popular choice for constructing flexible Normalizing flows, complexity of computing Jacobian determinant can be addressed by making it a triangular matrix with all diagonal entries as 1, and determination will always be one.

In this work, we leverage fast inverse of convolutions for a forward pass (inv-conv = f_θ) and convolution for a backward pass and designed *Inverse-Flow* model to generate fast samples. To train Inverse-Flow, we use our proposed fast and efficient backpropagation algorithm for inverse of convolution.

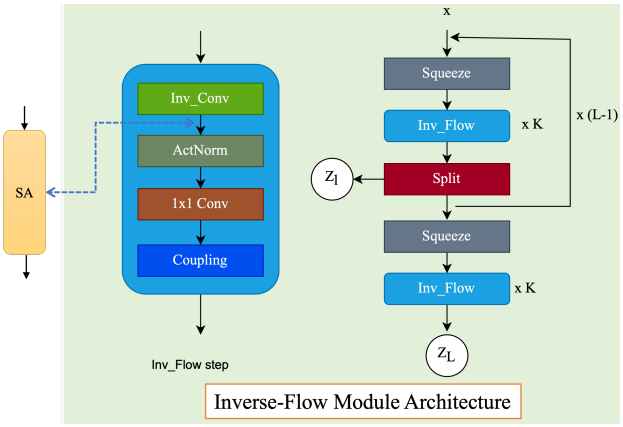

Figure 3: Multi-scale architecture of Inverse-Flow model and Inv Flow step.

4.1 Inverse-Flow Architecture

Figure 3 shows architecture of Inverse-Flow. Designing flow architecture is crucial to obtaining a family of bijections whose Jacobian determinant is tractable and computation is efficient for forward pass and backward pass. Our model architecture resembles architecture of Glow [Kingma and Dhariwal, 2018]. Multi-scale architecture involves a block of Squeeze, an *Inv_Flow* Step repeated K times, and a Split layer. The block is repeated $L-1$ a number of times. A Squeeze layer follows this, and finally, *Inv_Flow* Step is repeated K times. At end of each Split layer, half of channels are 'split' (taken away) and modeled as Gaussian distribution samples. These splits half channels are latent vectors. Same is done for output channels. These are denoted as Z_L in Figure 3. Each *Inv_Flow* Step consists of an $Inv - Conv$ layer, an Actnorm Layer, and a 1×1 Convolutional Layer, followed by a Coupling layer.

Inv_Flow **Step:** First we consider inverse of convolution and call it Inv_Conv layer. Figure 3 left visualizes inverse of $k \times k$ convolution (Inv-Conv) block followed by Spline Activation layer.

SplineActivation (SA): Bohra et al. [2020] introduced a free-form trainable activation function for deep neural networks. We use this layer to optimize Inverse-Flow model. Figure 3, left most: SA layer is added in Inv_Flow step after Inv_Conv block.

Actnorm: Next, Actnorm, introduced in [Kingma and Dhariwal, 2018], acts as an activation normalization layer similar to that of a batch normalization layer. Introduced in Glow, this layer performs affine transformation of input using scale and bias parameters per channel.

1×1 **Convolutional:** This layer introduced in Glow does a 1×1 convolution for a given input. Its log determinant and inverse are very easy to compute. It also improves effectiveness of coupling layers.

Coupling Layer: RealNVP [Dinh et al., 2016] introduced a layer in which input is split into two halves. First half remains unchanged, and second half is transformed and parameterized by first half. The output is concatenation of first half and affine transformation by functions parameterized by first of second half. Coupling layer consists of a 3×3 convolution followed by a 1×1 and a modified 3×3 convolution used in Emerging.

Squeeze: this layer takes features from spatial to channel dimension [Behrmann et al., 2019], i.e., it reduces feature dimension by a total of four, two across height dimension and two across width dimension, increasing channel dimension by four. As used by RealNVP, we use a squeeze layer to reshape feature maps to have smaller resolutions but more channels.

Split: input is split into two halves across channel dimensions. This retains first half, and a function parameterized by first half transforms second half. The transformed second half is modeled as Gaussian samples are latent vectors. We do not use checkerboard pattern used in RealNVP and many others to keep architecture simple.

4.2 Inverse-Flow Training

During training, we aim to learn parameters of invertible transformations (including invertible convolutions) by maximizing likelihood of data. Given input data y and a simple base distribution p_z (e.g., a standard Gaussian distribution), training process aims to find a sequence of invertible transformations such that: $z = \text{inv-conv}(y)$, where z is a latent vector from base distribution and θ represents model. The likelihood of data under model is computed using change of variables formula:

$$\log_{p_Y}(y) = \log_{p_z}(\text{inv-conv}(y)) + \log \left| \det\left(\frac{\partial \text{inv-conv}(y)}{\partial y}\right) \right|$$

Here $\det(\frac{\partial \text{inv-conv}(y)}{\partial y})$ represents a Jacobian matrix of transformation, which is easy to compute for *inv-conv*.

4.3 Sampling for Inverse-flow

To generate samples from model after training, we use reverse process: Sample from base distribution $z \sim p_z(z)$ from a Gaussian distribution. Apply inverse of learned transformation to get back data space: $y = \text{conv}_\theta(z)$ This process involves performing inverse of all transformations in flow, including *inv-conv*. This sampling procedure ensures that generated samples are drawn from distribution that model has learned during training, utilizing invertible nature of convolutional layers.

5 Results

In this section, we compare the performance of Inverse-Flow against other flow architectures. We present Inverse-Flow model results for bit per-dimension (log-likelihood), sampling time (ST), and forward pass time (FT) on two image datasets. To test modeling of Inverse-Flow, we compare bits-per-dimension (BPD). To compare ST, we generate 100 samples for each flow setting on single *NVIDIA GeForce RTX 2080 Ti GPU* and take an average of 5 runs after warm-up epochs. For comparing FT, we present forward pass time with a batch size of 100 averaging over 10 batch runs after warm-up epochs. Due to computation constraints, we train all models for 100 epochs, compare BPD with other state-of-the-art, and show that Inverse-Flow outperforms based on model size and sampling speed.

Table 2: Performance comparison for MNIST dataset with 4 block size and 2 blocks, small model size. ST = samling time, FT = Forward pass, NLL is negative-log-likelihood. All times are in milliseconds (ms) and parameters in millions (M).

Method	ST (ms)	FT (ms)	NLL	BPD	param (M)
Emerging	332.7 ±2.7	121.0 ±1.5	630	1.12	0.16
FIncFlow	47.3 ±2.3	95.1 ±2.5	411	0.73	5.16
SNF	33.5 ±2.2	212.5 ±7.3	557	1.03	1.2
Inverse-Flow	12.2 ±1.1	77.9 ±1.3	350	0.62	0.6

Table 3: Performance comparison for MNIST with block size ($K = 16$) and number of blocks ($L = 2$).

Method	ST	NLL	BPD	Param	Inverse
SNF	99 ±2.1	699	1.28	10.1	approx
FIncFlow	90 ±2.2	655	1.15	10.2	exact
MintNet	320±2.8	630	0.98	125.9	approx
Emerging	814±6.2	640	1.09	11.4	exact
Inverse-Flow	52 ±1.3	710	1.31	1.6	exact

5.1 Modeling and Sample time for MNIST

We compare sample time (ST) and number of parameters for small model architecture ($L = 2$, $K = 4$) on small image datasets, MNIST [LeCun et al., 1998] with image size $1 \times 28 \times 28$ in Table 2. It may not be feasible to run huge models in production because of large computations. Therefore, it is interesting to study behavior of models when they are constrained in size. So, We compare Inverse-Flow with other Normalizing flow models with same number of flows per level (K), for $K = 4, 16$, and $L = 2$. In Table 2, Inverse-Flow demonstrates fastest ST of 12.2, significantly outperforming other methods. This advantage is maintained in Table 3, where Inverse-Flow achieves the second-best ST of 52 ± 1.3, only behind SNF but with a much smaller parameter count. Inverse-Flow gives competitive forward time. Table 2 shows that Inverse-Flow has best forward time of 77.9, indicating efficient forward pass computations compared to other methods.

In Table 2, Inverse-Flow achieves lowest NLL (350) and BPD (0.62), suggesting superior density estimation and data compression capabilities for MNIST dataset with small model size. Inverse-Flow consistently maintains a low parameter count for all model sizes. Table 2 uses only 0.6M parameters, which is significantly less than FInc Flow (5.16M) while achieving better performance. In Table 3, Inverse-Flow has smallest model size among all methods, demonstrating its efficiency. Inverse-Flow consistently shows strong performance across multiple metrics (ST, FT, NLL, BPD) while maintaining a compact model size. Following observations highlight Inverse-Flow's efficiency in sampling, density estimation, and parameter usage,

making it a competitive method for generative modeling on MNIST dataset.

For small linear flow architecture, our Inv_Conv demonstrates the best sampling time of 19.7 ± 1.2, which is significantly faster than all other methods presented in Table 4. This indicates that Inv_Conv offers superior efficiency in generating samples from model, which is crucial for many practical applications of generative models. Inv_Conv achieves fastest forward time of 100, outperforming all other methods. Additionally, it has smallest parameter count of 0.096 million, making it most parameter-efficient approach. This combination of speed and compactness suggests that Inv_Conv offers an excellent balance between computational efficiency and model size, which is valuable for deployment in resource-constrained environments or applications requiring real-time performance.

Table 4: Runtime comparison of small planer models with 9 layers with different invertible convolutional layers for MNIST.

Method	NLL	**ST**	**FT**	**Param**
Exact Conv.	637.4 ±0.2	36.5 ±4.1	294	0.103
Exponential Conv.	638.1 ±1.0	27.5 ±0.4	160	0.110
Emerging Conv.	645.7 ±3.6	26.1 ±0.4	143	0.103
SNF Conv.	638.6 ±0.9	61.3 ±0.3	255	0.364
Inv_Conv (our)	645.3 ±1.2	19.7 ±1.2	100	0.096

Table 5: Performance comparison for CIFAR10 dataset with $L = 2$ blocks and block size of $K = 4$.

Method	**BPD**	**ST**	**FT**	**Param**
SNF	3.47	199.0 ±2.2	81.8 ±3.6	0.446
Woodbury	3.55	2559.4 ±10.5	31.3 ±1.5	3.125
FIncFlow	3.52	47.3 ±2.3	125.5 ±4.2	0.589
Butterfly Flow	3.36	155 ±4.6	394.6 ±3.4	3.168
Inverse-Flow	3.56	23.2 ±1.3	250.2 ±2.9	0.466

5.2 Modeling and Sample time for CIFAR10

In Table 5, Inverse-Flow demonstrates the fastest sampling time of 23.2 ± 1.3, significantly outperforming other methods. This advantage is maintained in Table 6, where Inverse-Flow achieves the second-best sampling time of 91.6 ± 6.5 among methods with exact inverse computation, only behind SNF which uses an approximate inverse. While not the fastest in forward time, Inverse-Flow shows balanced performance. In Table 5, its forward time of 250.2 ± 2.9 is in the middle range. In Table 6, its forward time of 722 ± 7.0 is competitive with other exact inverse methods.

While not the best, Inverse-Flow maintains competitive BPD scores. In Table 5, it achieves 3.56 BPD, which is comparable to other methods. In Table 6, its BPD of 3.57 is close to the performance of other exact inverse methods. Inverse-Flow consistently maintains a low parameter count. In Table 5, it uses only 0.466M parameters, which is among the lowest. In Table 6, Inverse-Flow has the second-smallest model size (1.76M param) among methods with exact inverse computation, demonstrating its efficiency. Inverse-Flow demonstrates a good balance between sampling speed and BPD. Comparing Tables 5 and 6, we can see that Inverse-Flow scales well when increasing the block size from 4 to 16. It maintains competitive performance across different model sizes and complexities. Table 6 highlights that Inverse-Flow provides exact inverse computation, a desirable property shared with several other methods like MaCow, CInC Flow, Butterfly Flow, and FInc Flow.

Table 6: Performance comparison for CIFAR dataset with block size ($K = 16$) and number of blocks ($L = 2$). SNF uses approx for inverse, and MintNet uses autoregressive functions. *time in seconds.

Method	**BPD**	**ST**	**FT**	**Param**
SNF	3.52	16.8 ±2.7	609 ±5.4	1.682
MintNet	3.51	25.0* ±1.5	2458 ±6.2	12.466
Woodbury	3.48	7654.4 ±13.5	119 ±2.5	12.49
MaCow	3.40	790.8 ±4.3	1080 ±6.6	2.68
CInC Flow	3.46	1710.0 ±9.5	615 ±5.0	2.62
Butterfly Flow	3.39	311.8 ±4.0	1325 ±7.5	12.58
FInc Flow	3.59	194.8 ±2.5	548 ±6.2	2.72
Inverse-Flow	3.57	91.6 ±6.5	722 ±7.0	1.76

6 Conclusion

In this paper, we give a fast and efficient backpropagation algorithm for inverse of convolution. Also, we proposed a flow-based model, Inverse-Flow, that leverages convolutions for efficient sampling and inverse of convolution for learning. Our key contributions include a fast backpropagation algorithm for inverse of convolution, enabling efficient learning and sampling; a multi-scale architecture accelerating sampling in Normalizing flow models; a GPU implementation for high-performance computation; and extensive experiments demonstrating improved training and sampling timing. Inverse-Flow significantly reduces sampling time, making them competitive with other generative approaches. Our fast and efficient backpropagation opens new avenues for training more expressive and faster Normalizing flow models. Inverse-Flow represents a substantial advancement in efficient, expressive generative modeling, addressing key computational challenges and expanding the practical applicability of flow-based models. This work contributes to the ongoing development of generative models and their real-

world applications, positioning flow-based approaches as powerful tools in the machine-learning landscape.

Acknowledgment

This research was supported by iHub-IIITH PhD Fellowship 2023-24.

References

J. Behrmann, W. Grathwohl, R. T. Chen, D. Duvenaud, and J.-H. Jacobsen. Invertible residual networks. In *International conference on machine learning*, pages 573–582. PMLR, 2019.

P. Bohra, J. Campos, H. Gupta, S. Aziznejad, and M. Unser. Learning activation functions in deep (spline) neural networks. *IEEE Open Journal of Signal Processing*, 1:295–309, 2020.

L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

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

R. Cornish, A. Caterini, G. Deligiannidis, and A. Doucet. Relaxing bijectivity constraints with continuously indexed normalising flows. In *International conference on machine learning*, pages 2133–2143. PMLR, 2020.

L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

C. Durkan, A. Bekasov, I. Murray, and G. Papamakarios. Neural spline flows. *Advances in neural information processing systems*, 32, 2019.

T. Eboli, J. Sun, and J. Ponce. End-to-end interpretable learning of non-blind image deblurring. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVII 16*, pages 314–331. Springer, 2020.

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

E. Hoogeboom, R. Van Den Berg, and M. Welling. Emerging convolutions for generative normalizing flows. In *International conference on machine learning*, pages 2771–2780. PMLR, 2019.

G. Jung, G. Biroli, and L. Berthier. Normalizing flows as an enhanced sampling method for atomistic supercooled liquids. *Machine Learning: Science and Technology*, 5(3):035053, 2024.

A. Kallappa., S. Nagar., and G. Varma. Finc flow: Fast and invertible k × k convolutions for normalizing flows. *Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2023) - Volume 5: VISAPP*, pages 338–348, 2023. ISSN 2184-4321. doi: 10.5220/0011876600003417.

M. Karami, D. Schuurmans, J. Sohl-Dickstein, L. Dinh, and D. Duckworth. Invertible convolutional flow. *Advances in Neural Information Processing Systems*, 32, 2019.

T. A. Keller, J. W. Peters, P. Jaini, E. Hoogeboom, P. Forré, and M. Welling. Self normalizing flows. In *International Conference on Machine Learning*, pages 5378–5387. PMLR, 2021.

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

D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improved variational inference with inverse autoregressive flow. *Advances in neural information processing systems*, 29, 2016.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.

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

C. Meng, L. Zhou, K. Choi, T. Dao, and S. Ermon. Butterflyflow: Building invertible layers with butterfly matrices. In *International Conference on Machine Learning*, pages 15360–15375. PMLR, 2022.

S. Nagar, M. Dufraisse, and G. Varma. CInc flow: Characterizable invertible 3×3 convolution. In *The 4th Workshop on Tractable Probabilistic Modeling, Uncertainty in Artificial Intelligence (UAI)*, 2021.

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

G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan. Normalizing

flows for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021.

L. Xu, J. S. Ren, C. Liu, and J. Jia. Deep convolutional neural network for image deconvolution. *Advances in neural information processing systems*, 27, 2014.

C. Zang and F. Wang. Moflow: an invertible flow model for generating molecular graphs. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 617–626, 2020.

Parallel Backpropagation for Inverse of a Convolution with Application to Normalizing Flows: Supplementary Materials

We provide a comprehensive extension to the main paper, offering in-depth insights into the experimental setup, additional experimental results, and rigorous mathematical proofs. The Supplementary begins with experimental specifications Section 7, including information about model architecture, training parameters, and hardware used. In next Section 8, we present an interesting application of inverse convolution layers in image classification, demonstrating high accuracy on MNIST dataset with a remarkably small model. Section 9 presents thorough proofs of two theorems related to backpropagation algorithm for inverse of convolution layers. These proofs, presented with clear mathematical notation and step-by-step derivations, establish a theoretical foundation for computing input gradients and weight gradients in context of inverse of convolution operations.

7 Experimental Details

The architecture of SNF is the starting point for Inverse-Flow architecture and all our experiments. All models are trained using the Adam optimizer. We evaluate our Inverse-Flow model for density estimation (BPD, NLL), Sampling time (ST), and Forward time (FT) with a batch size of 100 for all experiments. For MNIST, we use an initial learning rate of $1e-3$, scheduled to decrease by one order of magnitude after 50 epochs for all datasets but CIFAR10, which is decreased every 25 epochs. All the experiments were run on NVIDIA GeForce RTX 2080 Ti GPU. For MaCow, SNF, 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.

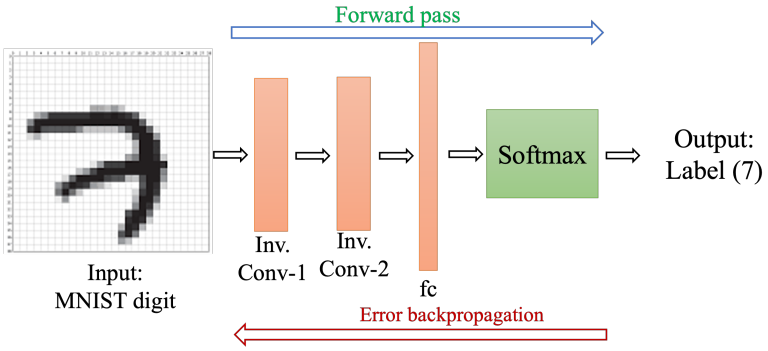

Figure 4: Overview of a small image classification model with two inverse of convolution (3×3 inv-con) layers with 97.6% accuracy on MNIST dataset.

8 Image Classification using Inverse of Convolution Layers:

For MNIST digits image classification using the inverse of convolution (inv-conv) layers and proposed its back-propagation algorithm, we trained a two inv-conv layer and one fully connected layer model with 16 learnable parameters for inv-conv layers. See Figure 4, this simple and small two inv-conv and one fully connected (fc) layers model gives 97.6% classification accuracy after training for 50 epochs.

9 Detailed Proofs

In this section, we provide the proofs relating to the proposed backpropagation algorithm for the inverse of the convolution layer. First, we provide the following notation and the equation for the gradients.

Notation: We will follow the notation used in the main paper.
We will denote input to the inverse of convolution (inv-conv) by $y \in \mathbb{R}^{m^2}$ and output to be $x \in \mathbb{R}^{m^2}$. We will be indexing x, y using $p = (p_1, p_2) \in \{1, \cdots, n\} \times \{1, \cdots, n\}$. We define

$$\Delta(p) = \{(i, j) : 0 \leq p_1 - i, p_2 - j < k\} \setminus \{p\}.$$

$\Delta(p)$ informally is set of all pixels except p which depend on p, when convolution is applied with top, left padding. We also define a partial ordering $\leq$ on pixels as follows

$$p \leq q \quad \Leftrightarrow \quad p_1 \leq q_1 \text{ and } p_2 \leq q_2.$$

The kernel of $k \times k$ convolution is given by matrix $W \in \mathbb{R}^{k \times k}$. For backpropagation algorithm for inv-conv, input is

$$x \in \mathbb{R}^{m^2} \text{ and } \frac{\partial L}{\partial x} \in \mathbb{R}^{m^2},$$

where L is loss function. We can compute y on $O(mk^2)$ time using parallel forward pass algorithm Aaditya et. al. The output of backpropagation algorithm is

$$\frac{\partial L}{\partial y} \in \mathbb{R}^{m^2} \text{ and } \frac{\partial L}{\partial W} \in \mathbb{R}^{k^2}$$

which we call input and weight gradient, respectively. We provide the algorithm for computing these in the next 2 subsections.

9.1 Proof of Theorem 1

Computing Input Gradients Since y is input to inv-conv and x is output, $y = \text{conv}_W(x)$ and we get following m^2 equations by definition of convolution operation.

$$y_p = xp + \sum_{q \in \Delta(p)} W_{(k,k)-p+q} \cdot x_q \tag{7}$$

Using chain rule of differentiation, we get that

$$\frac{\partial L}{\partial y_p} = \sum_q \frac{\partial L}{\partial x_q} \times \frac{\partial x_q}{\partial y_p}. \tag{8}$$

Hence if we find $\frac{\partial x_q}{\partial y_p}$ for every pixels p, q, we can compute $\frac{\partial L}{\partial y_p}$ for every pixel p.

Theorem 1: Input y gradients

$$\frac{\partial x_q}{\partial y_p} = \begin{cases} 1 & \text{if } p = 1 \\ 0 & \text{if } q \not\leq p \\ -\sum_{r \in \Delta(p)} W_{(k,k)-r} \frac{\partial x_{p-r'}}{\partial y_p} & \text{otherwise.} \end{cases} \tag{9}$$

Proof. We will prove Theorem 1 by induction on the partial ordering of pixels.

Base Case: For $p = (1, 1)$, which is the smallest element in our partial ordering:

From Equation (7), we have: $y_{(1,1)} = x_{(1,1)}$. This implies: $\frac{\partial x_{(1,1)}}{\partial y_{(1,1)}} = 1$ and for any $q \neq (1, 1)$: $\frac{\partial x_q}{\partial y_{(1,1)}} = 0$. This satisfies the theorem for the base case.

Inductive Step: Assume the theorem holds for all pixels less than p in the partial ordering. We will prove it holds for p.

1. For $q \not\leq p$, x_q does not depend on y_p due to the structure of the convolution operation. Therefore, $\frac{\partial x_q}{\partial y_p} = 0$.

2. For $q \leq p$, we differentiate Equation (7) with respect to y_p:

$$\frac{\partial x_p}{\partial y_p} = \frac{\partial x_p}{\partial y_p} + \sum_{r \in \Delta(p)} W_{(k,k)-p+r} \cdot \frac{\partial x_r}{\partial y_p}$$

$$1 = \frac{\partial x_p}{\partial y_p} + \sum_{r \in \Delta(p)} W_{(k,k)-p+r} \cdot \frac{\partial x_r}{\partial y_p} \tag{10}$$

Rearranging 10:

$$\frac{\partial x_p}{\partial y_p} = 1 - \sum_{r \in \Delta(p)} W_{(k,k)-p+r} \cdot \frac{\partial x_r}{\partial y_p} \tag{11}$$

This is equivalent to the third case in the theorem, with $q = p$.

3. For $q < p$, we can write:

$$x_q = y_q - \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot x_r$$

Differentiating with respect to y_p:

$$\frac{\partial x_q}{\partial y_p} = \frac{\partial y_q}{\partial y_p} - \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot \frac{\partial x_r}{\partial y_p}$$

Since $q < p$, $\frac{\partial y_q}{\partial y_p} = 0$. Therefore:

$$\frac{\partial x_q}{\partial y_p} = - \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot \frac{\partial x_r}{\partial y_p} \tag{12}$$

This is equivalent to the third case in the theorem.

Thus, by induction, the theorem holds for all pixels p. □

9.2 Proof of Theorem 2

Computing Weight Gradients From Equation 7, we can say computing gradient of loss L with respect to weights W involves two key factors. Direct influence: how a specific weight W in convolution kernel directly affects output x pixels, and Recursive Influence: how neighboring pixels, weighted by kernel, indirectly influence output x during inverse of convolution operation. Similarly, to compute gradient of loss L w.r.t filter weights W, we apply chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial x} \times \frac{\partial x}{\partial W} \tag{13}$$

where: $\frac{\partial L}{\partial x}$ is gradient of loss with respect to output x and inverse of convolution operation is applied between $\frac{\partial L}{\partial x}$ and output x. Computing gradient of loss L with respect to convolution filter weights W is important in backpropagation when updating convolution kernel during training. Similarly, $\partial L/\partial W$ can be calculated as Equation 13 and $\partial x/\partial W$ can be calculated as (Equation 13) for each $k_{i,j}$ parameter by differentiating Equation 7 w.r.t W:

$$\frac{\partial L}{\partial W_a} = \sum \frac{\partial L}{\partial x_q} \times \frac{\partial x_q}{\partial W_a} \tag{14}$$

Equation 14 states that to compute gradient of loss with respect to each weight W_a, we need to:

- Compute how loss L changes with respect to each output pixel x_q (denoted by $\frac{\partial L}{\partial x_q}$).
- Multiply this by gradient of each output pixel x_q with respect to weight W_a (denoted by $\frac{\partial x_q}{\partial W_a}$)

We then sum over all output pixels x_q.

Theorem 2: Weights W gradients

$$\frac{\partial x_q}{\partial W_a} = \begin{cases} 0 & \text{if } q \leq a \\ -\sum_{q' \in \Delta q(a)} W_{q'-a} \times \frac{\partial x_{q-q'}}{\partial W_a} - x_{q-a} & \text{if } q > a \end{cases} \tag{15}$$

Proof. We will prove Theorem 2 by induction on the partial ordering of pixels.

Base Case:

For $q \leq a$, we have $\frac{\partial x_q}{\partial W_a} = 0$.

This is because in the inverse of convolution operation, x_q does not directly depend on W_a. The weight W_a only affects pixels that come after q in the partial ordering.

Inductive Step: Assume the theorem holds for all pixels less than q in the partial ordering. We will prove it holds for $q > a$.

From Equation (7), we have:

$$y_q = x_q + \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot x_r \tag{16}$$

Rearranging this equation 16:

$$x_q = y_q - \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot x_r \tag{17}$$

Now, let's differentiate both sides of 17with respect to W_a:

$$\frac{\partial x_q}{\partial W_a} = \frac{\partial y_q}{\partial W_a} - \sum_{r \in \Delta(q)} \left(\frac{\partial W_{(k,k)-q+r}}{\partial W_a} \cdot x_r + W_{(k,k)-q+r} \cdot \frac{\partial x_r}{\partial W_a} \right) \tag{18}$$

Note that $\frac{\partial y_q}{\partial W_a} = 0$ because y is the input to the inverse convolution and doesn't depend on W.

Also, $\frac{\partial W_{(k,k)-q+r}}{\partial W_a} = 1$ if $(k,k) - q + r = a$, and 0 otherwise.

Let $\Delta q(a) = \{r \in \Delta(q) : (k,k) - q + r = a\}$. Then we can rewrite the equation 18 as 19:

$$\frac{\partial x_q}{\partial W_a} = -\sum_{r \in \Delta q(a)} x_r - \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot \frac{\partial x_r}{\partial W_a} \tag{19}$$

The first sum simplifies to $-x_{q-a}$ because $r = q - (k,k) + a$ for $r \in \Delta q(a)$.

In the second sum, we can use the inductive hypothesis for $\frac{\partial x_r}{\partial W_a}$ because $r < q$.

Therefore:

$$\frac{\partial x_q}{\partial W_a} = -x_{q-a} - \sum_{r \in \Delta(q)} W_{(k,k)-q+r} \cdot \frac{\partial x_r}{\partial W_a} \tag{20}$$

The right side of 20 is equivalent to the second case in the theorem.

Thus, by induction, the theorem holds for all pixels q. □