Max McGuinness

Image Rescaling using Invertible Neural Networks, by Probabilistic Disentanglement

Computer Science Tripos – Part II
Queens' College

October 7, 2024

Declaration

I, Max McGuinness of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Max McGuinness of Queens' College, am content for my dissertation to be made available to the students and staff of the University.

Signed: Max McGuinness
Date: October 7, 2024

Proforma

Candidate number:	**2333C**
Project Title:	**Image Rescaling using Invertible Neural Networks, by Probabilistic Disentanglement**
Examination:	**Computer Science Tripos – Part II, 2022**
Word Count:	**11291** [1]
Code Line Count:	**3012** [2]
Project Originator:	**The dissertation author and Param Hanji**
Project Supervisor:	**Param Hanji**

Original Aims of the Project

The aim of this project was to tackle the computer vision problem of *image rescaling* by implementing a deep learning model known as an *Invertible Rescaling Network (IRN)*, based on work by Xiao et al. 2020. The core goal was to reproduce the image rescaling results reported in their paper by implementing *IRN* myself in PyTorch using an open-source framework known as FrEIA. As preparatory work, I planned to assess the suitability of FrEIA by using it to construct a smaller invertible architecture known as Glow. Extensions included assessing IRN's performance at adjacent tasks.

Work Completed

I exceeded my core success criteria and several extensions. My PyTorch-based *IRN* model successfully reproduced the performance described by its creators. I assessed the suitability of the FrEIA framework by training Glow to generate handwritten digits, and made a small contribution to FrEIA's codebase. I highlighted flaws with the field's current standard metrics for *image rescaling* by demonstrating how modifications to existing models can challenge state-of-the art performance. I proposed a new task - *compression-resistant image rescaling* - which I argue has higher real-world utility than *image rescaling*, and report initial results for solving the problem using *IRN*.

Special Difficulties

None.

[1] This word count was calculated using TeXcount (`https://app.uio.no/ifi/texcount/`).

[2] This line count was calculated using VsCodeCount (`https://github.com/uctakeoff/vscode-counter`).

Contents

List of Figures

Chapter 1

Introduction

Most challenging problems in science can be described as **inverse problems**. These are problems characterised by the need to deduce system parameters from observations - for example, deducing the 3D shape of an object from its shadow, or distinguishing tumours from the reflectance of biological tissue [1]. These problems are often considered *ill-posed* [2], in the sense that there is insufficient information in the observation to fully recover the parameters. In other words, information is "lost" in producing the observation.

Image rescaling is a long-standing inverse problem[1] in the field of computer vision. We can consider a high-resolution image to be the "parameter" of this system, and a low-resolution image the "observation", which has inherently lost some information from its high-resolution counterpart. Solving the inverse problem in this case is known as upscaling, and solving the forwards problem as downscaling. Because downscaling has long been a crucial technique for image compression [3] and compatibility across platforms - allowing faster image processing by reducing image dimensions - most images online are in fact a downscaled copy of some higher-resolution original[2]. This is typically performed using bicubic downscaling, and recovering a higher-resolution image from one of these results is notoriously difficult [4, 5].

I summarise the image rescaling problem as follows: **how can we downscale an image to low-resolution, then upscale to its original high-resolution, with a minimal loss in image quality at all stages[3]?**

Previous work in image rescaling typically considers upscaling and downscaling as separate problems. Super-resolution techniques such as the deep-learning-based SRGAN [6] and EDSR [7] focus on upscaling specifically from bicubic-downscaled images. Downscaling techniques such as Öztireli and Gross (2015) [8] focus on maximizing the perceived quality of downscaled images. More recently, however, there has been a resurgence of interest in developing a downscaling method that is conducive to high-quality upscaling. This approach has been found to greatly alleviate the ill-posed nature of image rescaling as an inverse problem [9]. In particular, in 2019, it was found that one can produce particularly effective and informative solutions to inverse problems by modelling them with a class of deep learning model known as **Invertible Neural Networks (INNs)** [10].

At a simple level, an Invertible Neural Network can be thought of as a function that is highly expressive and has an inverse. This invertibility means that we can train an INN to approximate $f(x) = y$, then invert the network to get $f^{-1}(y) = x$ for "free". This approach enables

[1] To be pendantic: it is an inverse problem specifically because image rescaling may involve *upscaling*.

[2] As a typical example, consider an image taken on a digital camera which downscales its raw 8000x5000px images to 4000x2500px, sent over an email service which downscales to 3000x1875px.

[3] In this case, by *image quality* I refer to perceived similarity to the original high-res image.

encapsulation of both the forward process (*parameters* → *observations*) and inverse process (*observations* → *parameters*) of an inverse problem in a single model, leading to highly effective solutions capable of understanding both domains.

Breakthrough image rescaling results were thus provided in 2020 by Xiao et al's **Invertible Rescaling Network (IRN)** [11], achieving scores at least 10% greater than previous methods on standard benchmarks[4]. This was closely followed by a small number of alternative invertible architectures for image rescaling, achieving state of the art results over IRN with some caveats [12, 13, 14] that I discuss in section 3.4 and appendix A.2.

However, despite recent successes, the realm of invertible networks for image rescaling remains under-explored. There are few open-source implementations available online and, I believe, a number of lingering assumptions remaining unchallenged, such as the efficacy of chosen test metrics and loss functions.

1.1 Project Summary

In this project, I implemented Xiao et al.'s Invertible Rescaling Network [11]. In the process, I evaluated different approaches to developing invertible architectures, assessed existing image rescaling solutions, and from these offered novel improvements to the IRN architecture that surpass the original.

Specifically, I will demonstrate the following:

- I implemented IRN using an open-source framework known as *FrEIA* in *PyTorch* (section 3.3). I provide evidence that my implementation achieves image rescaling results matching the original paper after training on a GPU server. These can be found in section 4.2.1.
- I extended the IRN architecture to support improvements suggested in recent papers. I also derive a method of reducing the *Lipschitz constant* of IRN's invertible layers (section 3.4.1), and show that applying it results in **performance surpassing the original paper** (section 4.2.2).
- I found that by altering the *loss function* used to learn image rescaling, I was able to produce a model that achieves **state of the art image-rescaling results on standard benchmarks** (*Y-channel PSNR/SSIM*) while *underperforming* in other tests, suggesting a need to reconsider the standard criteria used to assess image rescaling models (section 4.2.3). I suggest alternative benchmarks.
- I extended the IRN architecture to evaluate its aptitude at adjacent problems including *super-resolution* and *compression*, discussing properties of the network not noted in the original paper (section 4.3). I define a new problem - *compression-resistant image rescaling* - and present results for it in 4.3.1.
- As part of my preparation, and for the sake of comparison, I implemented two versions of an invertible architecture known as *Glow* (section 3.2.1) - one in *FrEIA* and one in base *PyTorch* (results in 4.1). I discuss the merits and shortcomings of FrEIA as a framework for invertible neural networks, and contribute to its open source codebase.

[4]Specifically its Y-channel PSNR score - for example achieving 44.32 on 2× rescaling in the DIV2K dataset compared to previous SOTA of 39.01. This metric is further explained in section 2.4.

Chapter 2

Preparation

2.1 Deep Learning

Machine learning can broadly be described as the process of solving a task by iteratively refining an algorithm (a *model*) using data. Deep learning is a subset of machine learning in which this model involves transforming data using a series of neural-inspired layers.

Deep learning models operate on multidimensional data known as tensors, and a typical **fully-connected neural layer** performs the affine operation:

$$y_i = w_i \cdot x + b_i \tag{2.1}$$

where y_i denotes the layer's output at index (i), x denotes the layer's input, and w and b are parameters of the layer known as weights and bias. Note that $\cdot$ denotes dot product, thus produces a scalar.

If each layer in a deep learning model is differentiable, then the model can be trained using a process known as gradient descent. At each sample in our training dataset, we quantify the success of the model using a loss function, then compute the gradient of each model parameter with respect to that loss function, using it to adjust the parameters so as to reduce the loss. We have:

$$w_{t+1} = w_t - lr\frac{dL(x, w_t)}{dw_t} \tag{2.2}$$

where w_t are the model parameters at time t, x is a sample from our training dataset, lr is the learning rate, and L is our loss function.

2.1.1 Loss Functions

Loss functions provide a means of measuring the incorrectness (termed *loss*) of a model at a single training sample or batch of samples. In this section, I outline several types of loss function.

Maximum Likelihood

Consider a dataset T of (x, y) observation-label pairs, and a network $N(x) = y$ that attempts to model it. If we have a way of expressing the likelihood of T given a model N (that is to

say, we have an expression for $\mathcal{L}(N|T) = p(T|N)$), then we can optimise N by maximising that likelihood. This means optimising for the following over the whole dataset [15]:

$$\begin{aligned} & \text{maximise } p(T|N) \\ \equiv\ & \text{maximise } \prod_{i=0}^{n-1} p(T_i|N) \\ \equiv\ & \text{maximise } \sum_{i=0}^{n-1} \ln p(T_i|N). \end{aligned} \tag{2.3}$$

This is known as **maximum likelihood estimation (MLE)**. If the expression is differentiable, then we can perform this optimisation via gradient descent, taking the loss function to be the negative log-likelihood:

$$loss = -\ln p(T_i|N). \tag{2.4}$$

One example of a type of model for which this likelihood is available is the **classification model**. Suppose $T : X \longrightarrow Y$ maps each element in X to a discrete label in Y, where $|Y| = n$. Our classification model N will estimate the probability of $(x, y) \in T$ for all Y for any given element in X, such that $N : X \longrightarrow \mathbb{R}^n$.

As a classification model, N is directly performing $N(X_i)_j = p((X_i, Y_j)|N)$, hence giving us a straightforward means of extracting likelihood. Without loss of generality, consider $T_i = (X_i, Y_j)$. This gives us likelihood:

$$p(T_i|N) = p((X_i, Y_j)|N) = N(X_i)_j. \tag{2.5}$$

Generic Loss

With most deep learning models, however, there does not exist an obvious expression for likelihood. Instead, we can take the loss function to be a direct comparison between model output and ground truth.

Suppose we want to optimise a model $N(x) = y$ and have a ground truth function T. I outline the two most common generic loss functions below, taking $x, N(x), T(x) \in \mathbb{R}^n$.

- **L1: Absolute Deviation**

$$loss = L_1(N(x) - T(x)) = \sum_{i=0}^{n-1} |N(x)_i - T(x)_i| \tag{2.6}$$

- **L2: Sum-of-Squares**

$$loss = L_2(N(x) - T(x)) = \sum_{i=0}^{n-1} (N(x)_i - T(x)_i)^2 \tag{2.7}$$

By being more sensitive to outliers (lacking a property known as *robustness*), sum-of-squares loss is more susceptible to instability during training - i.e. outliers are more influential. On the other hand, however, L2 loss is capable of producing a final model more ready to respond to outliers compared to a model trained by L1 loss - i.e. more appropriately encapsulating high-leverage data points [16].

It is useful to keep in mind that optimising for sum-of-squares loss is equivalent to performing maximum likelihood estimation when assuming that model error is gaussian-distributed [17, p. 197]. For completeness, I perform this derivation in appendix D.3.

Cross-Entropy Loss

Similar to maximum likelihood estimation is **cross-entropy** loss, which is a direct measurement of the difference between two distributions [15].

$$\begin{aligned} CE(f, g) &= \mathbb{E}_f(-\ln p_g(x)) \\ &= -\sum_x (p_f(x) \ln p_g(x)) \end{aligned} \tag{2.8}$$

Importantly, we can think of optimising a model through MLE (2.3) as equivalent to minimising the cross-entropy between that model's distribution and the empirical distribution of its training dataset. In other words, supposing $\hat{T}$ is the empirical distribution of training dataset T (where $|T| = n$), and $\hat{N}$ is the current distribution modelled by model N, we have:

$$\text{maximise } \mathcal{L}(N, T) = p(T|N) \quad \equiv \quad \text{minimise } CE(\hat{T}, \hat{N}) = p_{\hat{N}}(T) \tag{2.9}$$

We can derive this equivalence by breaking down the CE definition (2.8) and applying the empirical distribution property $p_{\hat{T}}(T_i) = \frac{1}{n}$, like so:

$$\begin{aligned} CE(\hat{T}, \hat{N}) &= \mathbb{E}_{\hat{T}}(-\ln p_{\hat{N}}(x)) \\ &= -\sum_{i=0}^{n-1} (p_{\hat{T}}(T_i) \ln p_{\hat{N}}(T_i)) \\ &= -\sum_{i=0}^{n-1} \frac{1}{n} \ln p_{\hat{N}}(T_i) \end{aligned} \tag{2.10}$$

Examining a single element and taking out constant factor $\frac{1}{n}$, we obtain a simple loss function for empirical cross-entropy:

$$loss = -\ln p_{\hat{N}}(T_i) \tag{2.11}$$

which is simply a formulation of our (2.4) MLE definition $loss = -\ln p(T_i|N)$.

2.1.2 Deep Learning for Image Processing

Deep learning for image processing largely revolves around the application of **convolutional layers**.

Convolutional Layers

A convolutional layer consists of a set of tensors referred to as "filters". Each filter slides over the layer's input tensor, computing each output element to be the dot product between the filter and a section of the input. Each filter's output is concatenated to form the total output

of the layer. More formally, for each filter, we have[1]:

$$y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} f_{ab} x_{(i+a)(j+b)} \tag{2.12}$$

where y_{ij} denotes the filter's output at index (i, j), $(m \times m)$ is the size of our 2D filter f, and x is the layer input. These convolutional layers are instrumental to image processing, as they allow a model to learn transformations that apply across images of any size, such as edge detection.

ReLU Activation

Being a linear operation, convolutional layers are typically followed by a non-linear transformation, most commonly Rectified Linear Unit (ReLU) [18]:

$$y = max(x, 0) \tag{2.13}$$

This non-linearity is important: without such a transformation, a convolutional neural network (a deep learning model consisting of multiple convolutional layers) would be a strictly linear transformation, and could effectively be compressed into a single linear layer. Introducing non-linearity is therefore essential in allowing deep neural networks the ability to approximate complex functions.

2.2 Invertible Neural Networks

As mentioned in the introduction, I define invertible neural networks very broadly as:

A neural network $N(x) = y$ which can be inverted to perform $N^{-1}(y) = x$.

Typical neural networks are not invertible. For a function to be invertible, it must be bijective, which is true of neither fully-connected layers (2.1), convolutional layers (2.12)[2], nor the ReLU function (2.13). Therefore substantial changes to the structure of a deep learning model are required in order to make it invertible.

The primary way of creating an invertible deep learning model is to construct it as a **normalizing flow**.

2.2.1 Normalizing Flow Networks

A normalizing flow is a machine learning model consisting of a series of invertible layers possessing certain properties. Specifically, each layer L_i, and thus the total flow itself $L_0(L_1(...,$ has the following properties:

- The layer's forward $(L_i(x))$ and inverse $(L_i^{-1}(y))$ are tractably computable. (2.14)
- The layer's Jacobian determinant (dL_i/dx) is tractably computable. (2.15)

[1]Note that the function described here, and typically implemented in practice, is technically *cross-correlation*, not *convolution*. A convolution is the same operation but with the filter flipped; this makes it commutative and thus easier to reason about in proofs.

[2]This is due to the dot product operation involved in both. To see that this is not bijective, consider how $(1, 0.5) \cdot (1, 0) = (1, 0.5) \cdot (0, 2) = 1$.

where by *tractably computable* I specifically mean *solvable in polynomial time* $O(n^k)$.

Together, these properties allow one to compute the likelihood of a normalizing flow producing a specific sample, as I will explain below. This enables normalizing flow models to be trained through Maximum Likelihood Estimation (MLE), in which we maximise the probability of a model producing results within a target distribution.

Normalizing flows receive their name from the concept of a probability distribution "flowing" through its layers. The flow is "normalizing" in that the invertibility of the layers results in a valid probability distribution on the other end.

Computing Likelihood

The key ability of normalizing flows is that, if the distribution on one end is known, then the distribution on the other can be derived through application of the **change of variables formula**. Where f is a differentiable bijective function (such as a normalizing flow) mapping from distribution A to B, the formula is:

$$p_A(x) = p_B(f(x)) * |\frac{df}{dx}| \tag{2.16}$$

Intuitively, we can think of $|\frac{df}{dx}|$ as a kind of "density" describing how compact A is in comparison to B. When $|\frac{df}{dx}|$ is high, smaller regions in A map to larger regions in B, hence $p(x|A)$ is higher. Extracting this probability allows us to train normalizing flow models through MLE - a walked-through example of this is given in **appendix C.2**.

2.2.2 Coupling layers

The backbone of a normalizing flow model is its choice of invertible layer. The most commonly used layers for this purpose are **coupling layers**, first described by Dinh et al. in 2014 [19].

This is a transformation that first requires us to split our input into two parts (which may be high-dimensional tensors - call these x_1 and x_2), and in the end produces two outputs (call these y_1 and y_2). For a coupling layer L:

$$L(x_1, x_2) = y_1, y_2$$

The choice of how to split some input x into x_1 and x_2 to prepare it for coupling varies depending on the structure of x and the wider task at hand, but is often performed by effectively taking slices at random along one axis.

Example

A simple example of such a layer is an *additive coupling layer*, which makes use of an arbitrary differentiable function m. Note that this inner function m does not need to be invertible, so could could itself be a non-invertible neural network - hence why it is sometimes referred to as a **subnet**.

Additive coupling layers perform the following forwards operation [19]:

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 + m(x_1) \end{aligned} \tag{2.17}$$

and can be inverted trivially to obtain the following backwards operation:

$$\begin{aligned} x_1 &= y_1 \\ x_2 &= y_2 - m(x_1) \end{aligned} \tag{2.18}$$

Moreover, the layer satisfies the requirement of having a tractably computable jacobian determinant, as it is simply a constant:

$$J_L(x) = \det \begin{pmatrix} dy_1/dx_1 & dy_1/dx_2 \\ dy_2/dx_1 & dy_2/dx_2 \end{pmatrix} = \det \begin{pmatrix} 1 & 0 \\ m'(x_1) & 1 \end{pmatrix} = 1$$

2.2.3 INNs for Inverse Problems

So far, I have described a type of invertible neural network known as normalizing flow, and given an example of a generative normalizing flow model $g : (Z \sim \mathcal{N}(0,1)) \longmapsto (IMAGE)$.

Now let us consider constructing an INN to solve an inverse problem. In our inverse problem, we have an "observation" from which we wish to derive the "parameter" that caused it. Let's specifically consider the task of image rescaling, in which our observation is a low-resolution image LR, and the parameter is its high-resolution counterpart HR.

Ideally, we would create an INN N that operates as follows:

$$\begin{aligned} N(HR) &= LR \\ N^{-1}(LR) &= HR \end{aligned}$$

However, inverse problems are unique in that there is often insufficient information in the observation to fully reconstruct the parameter: in this example, LR is smaller than HR, and as such there are many possible HR for a single LR.

This is a problem; in order to model the problem with an INN, we would need to make the $HR \longrightarrow LR$ relation bijective instead.

Probabilistic Disentanglement

Ardizzone et al.'s [10] solution to this is to include some latent data LD in the output, which - continuing our image rescaling example - encodes the information that was lost from HR when downscaling to LR. In other words, we solve the inverse problem by constructing an INN that separates HR into LR and LD:

$$\begin{aligned} N(HR) &= (LR, LD) \\ N^{-1}(LR, LD) &= HR \end{aligned}$$

This is now a valid invertible neural network, as $HR \longmapsto (LR, LD)$ is a bijection. However, this separation has changed the nature of the image rescaling problem, as one would be required to predict a suitable LD before being able to upscale LR.

The authors propose that this requirement would be alleviated if, during training, we take LD to be a random sample from a known distribution (specifically, an isotropic multivariate Gaussian). This way, the model is forced to learn how to upscale LR to HR without relying on information in LD - as it doesn't convey useful information in training. Likewise, the downscaling operation is forced to reduce the amount of case-specific information it encodes into LD to assist this.

In other words, the model must learn to downscale in such a way that the distribution of LD resembles a Gaussian distribution, thus "disentangling" the relationship between LR and LD. With LD no longer encoding particularly useful information, the problem effectively resembles our $N^{-1}(LR) = HR$ process in which we are required only to provide LR, as it is acceptable to randomly guess LD.

So, suppose that in place of the latent data LD we use a sample from the latent variable $LD_{random} \sim \mathcal{N}(0, 1)$. We can now perform our rescaling as follows:

$$N(HR) = (LR, LD)$$
$$N^{-1}(LR, LD_{random}) \approx HR$$

The network's design means that although we can build it as a normalizing flow, it cannot be optimised with MLE. This is because we don't know the distribution on either side of the network (neither HR nor (LR, LD)), thus preventing application of the change of variables formula. Instead, the network can be trained through a task-appropriate loss function such as the L2 loss between the network's approximated HR and the ground truth (in the case of images, this is known as a "pixel loss").

Though it is not essential for the reader's understanding of this project specifically, I describe alternative workarounds (instead of probabilistic disentanglement) that would enable INN MLE for inverse problems in appendix C.1.

2.3 Invertible Rescaling Network

Xiao et al.'s Invertible Rescaling Network (IRN) is a normalizing flow model that makes use of probabilistic disentanglement, as described above. The network is applied like so:

$$\textbf{Downscaling} \qquad IRN(HR) = (LR', LD) \qquad \textit{(where } LR' \approx LR\textit{)} \tag{2.19}$$
$$\textbf{Upscaling} \qquad IRN^{-1}(LR', LD_{random}) = HR' \qquad \textit{(where } HR' \approx HR\textit{)} \tag{2.20}$$

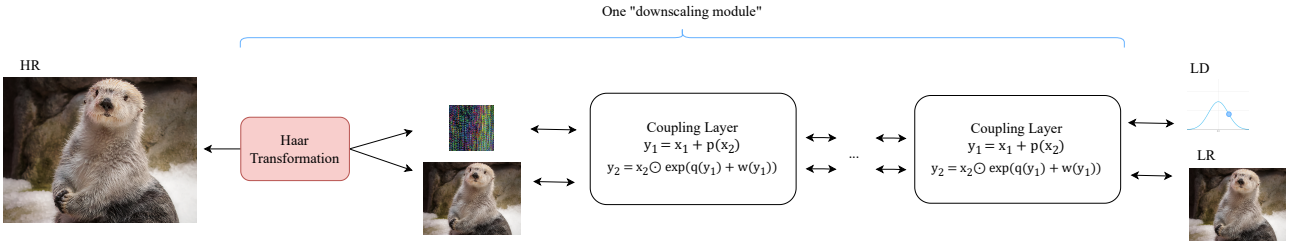
The core challenge in designing such a network is deciding how and when to split HR into LR' and LD. Xiao et al. make use of two types of invertible layers:

- **Haar transforms**. These primitively downscale an image (by $2\times$) while extracting from it the high-frequency information that was lost in downscaling, producing two outputs. In IRN, they perform $(HR) \longmapsto (LR, HF)$, with HF being said high-frequency information.
- **Coupling layers**. Specifically, a custom type of coupling layer that I call *enhanced affine coupling*. In IRN, the first of these layers takes its input from a Haar transform, performing $(LR, HF) \longmapsto (LR', LD)$.

The use of Haar transforms is key to IRN's design, as they provide an explicit mechanism for downscaling images while encoding the data that was lost in the downscaling into latent data. They furthermore provide a sensible method of splitting data to prepare it for transformation through coupling layers.

In the network, one Haar transformation is followed by about 8 coupling layers, which together form one "downscaling module". With a single downscaling module, one can produce an IRN that performs $2\times$ image rescaling like so:

Figure 2.1: An overview of IRN for 2× rescaling.



2.3.1 Haar Transform

A Haar Transform is a type of 2D discrete wavelet transform. It can be thought of as performing a dot product across segments of an input image, producing 4 output images (a, b, c, d) each of half the width and height of the input. The transform is invertible, producing the same amount of data as it takes in.

Consider an arbitrary 2×2 segment of input image x:

$$\begin{pmatrix} x_{ij} & x_{(i+1)j} \\ x_{i(j+1)} & x_{(i+1)(j+1)} \end{pmatrix}, \text{ which I shorthand to } \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

Although there exist a variety of conventions, in general a Haar transform computes a, b, c, d as:

$$\begin{aligned} a_{ij} &= p + q + r + s && \textit{downscaled image} \\ bij &= (p + r) - (q + s) && \textit{horizontal difference} \\ c_{ij} &= (p + q) - (r + s) && \textit{vertical difference} \\ d_{ij} &= (p + s) - (q + r) && \textit{diagonal difference} \end{aligned}$$

These four computations can be implemented as 2D convolutions with the following four filters:

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{2.21}$$

2.3.2 Enhanced Affine Coupling Layer

An *Enhanced Affine Coupling layer* is a coupling layer that performs an additive transformation for y_1, and an affine transformation for y_2. It makes use of three subnets (inner networks that can be complex differentiable functions).

Enhanced affine coupling layers perform the following forward operation:

$$\begin{aligned} y_1 &= x_1 + p(x_2) \\ y_2 &= x_2 \odot \exp(g(s(y_1))) + t(y_1) \end{aligned} \tag{2.22}$$

where p, s, t are arbitrary subnets, g is a clamp function, and $\odot$ indicates the Hadamard product operation[3]. In the case of IRN, the clamp function g is a sigmoid function with range $[-1, 1]$:

$$g(x) = \frac{1}{1 + e^{-x}} \cdot 2 - 1. \tag{2.23}$$

[3]The Hadamard product is an elementwise multiplication of two matrices of the same dimensions, i.e., $(A \odot B)_{ij} = A_{ij}B_{ij}$ for $A, B \in \mathbb{R}^{m \times n}$.

Motivation

This coupling layer is an extension of the *affine coupling layer* proposed by Dinh et al. [19], which performed only an identity transformation on y_1. The motivation for this extension is that it allows a network to subtly alter y_1 through coupling layers alone, without needing to swap it with y_2 between layers.

In IRN, we take x_1 to be the downscaled image produced by Haar downscaling, and x_2 to be the high-frequency information. In terms of output, we take y_1 to be LR and y_2 to be LD. The use of an additive transformation for y_1 enables the network to subtly alter LR, producing a downscaled image LR' that is conducive to high-quality upscaling. Without it, the network would effectively be constrained to performing super-resolution.

Implementation

In my implemetation of IRN, I defined Enhanced Affine Coupling as a subclass of FrEIA's `BaseCouplingBlock`, exposing parameters for the *clamp* function, *subnet* constructor, and the $x \to x_1/x_2$ *split* used. I also provided the ability to inject the layer with a *condition*, facilitating conditional INNs as described in C.1.

I submitted this new coupling class as a contribution to FrEIA's open source codebase, where the pull request is awaiting integration.

2.3.3 Loss Function

To train IRN, we optimise for the three criteria printed in table 2.1.

LR Guidance	The quality of our downsampled image.	$L_{guide} = L1(LR' - LR)$
HR Reconstruction	The quality of our high-res image reconstructed from LR.	$L_{recon} = L2(HR' - HR)$
LD Distribution Matching	How closely our latent data resembles a gaussian distribution.	$L_{distr} = \sum LD^2$

Table 2.1: IRN's loss functions

In practice, we weight the loss function with scalar coefficients $\lambda_{guide}, \lambda_{recon}, \lambda_{distr}$, giving us a total loss of $L_{total} = \lambda_{guide} \cdot L_{guide} + \lambda_{recon} \cdot L_{recon} + \lambda_{distr} \cdot L_{distr}$. These criteria are gained by performing a forward and then inverse application of the network, detailed in section 3.3.2.

Note that $\sum A$ indicates the sum of all elements in A, and A^2 indicates elementwise squaring of each element in A.

To further explain the motivation behind these criteria, it is worth commenting that L_{guide} is the cross entropy 2.1.1 between LD and $\mathcal{N}(0, 1)$ - hence reducing it is a form of MLE.

2.4 Perceptual Metrics

To evaluate the success of an image-processing model, it is important to measure the distance between ground truth and generated images not just as an absolute difference (e.g. $L1$, $L2$),

but also as a perceptual difference. In other words, it is important to approximately measure the similarity between two images from the perspective of the human visual system, which is not necessarily equivalent to, say, simple mean squared error.

I will briefly describe two types of perceptual metrics used for evaluation in image rescaling papers. Suppose X is our generated image and Y the ground-truth target:

- **PSNR (Peak Signal-to-Noise Ratio)** is a simple metric that measures the mean squared error between two images, but scaled by maximum value in the generated image X.

  $$PSNR(X,Y) = 20 \cdot log_{10}(MAX_X) - 10 \cdot log_{10}(MSE(X,Y))$$

  PSNR has been found to be particularly sensitive to noise, encouraging images that have a 'smooth' quality.

- **SSIM (Structural Similarity Index Measure)** is a newer and far more complex metric that measures distance using a sliding window, taking the mean of scores computed by the sliding window. For each pair of regions $(x, y) \in (X, Y)$ seen by the algorithm, the SSIM score is computed as the sum of three metrics, parameterised by scalars α, β, γ:

  $$SSIM(x,y) = luminance(x,y)^\alpha + contrast(x,y)^\beta + structure(x,y)^\gamma$$

  Without going into detail, it is enough to know that *luminance* compares the mean values across the two regions, *contrast* the standard deviation, and *structure* the correlation. Parameters α, β, γ are typically left at 1, as suggested by the authors [20].

  SSIM has been found to be closer to subjective quality compared to PSNR.

2.4.1 Y-Channel Testing

For the task of image rescaling, PSNR and SSIM typically computed on the **Y-channel** of images rather than RGB, computed by:

$$\mathbf{Y} = 16 + 65.481 \cdot \mathbf{R} + 128.553 \cdot \mathbf{G} + 24.966 \cdot \mathbf{B}. \tag{2.24}$$

Y refers to *luminance*[4], a component used in the YUV color model, and can be thought of as the intensity of light at a particular area. Human vision is far more sensitive to luminance than colour, and as such colour encoding standards such as PAL assign greater bandwidth to luminance than chrominance channels.

2.5 Requirements Analysis

Following a deeper dive into relevant papers in *image rescaling*, *machine learning*, and *invertible neural networks* - as well as gaining initial experience building toy examples in PyTorch and FrEIA - I revised my success criteria. In particular, I narrowed down on the difficulty of each task and considered it alongside that task's importance to the project. A summary of these requirements can be found in table 2.2. In particular, requirements critical to calling the project a "success" are given high priority; extensions (at the bottom) and preparatory tasks are given low priority; and difficulty is distributed roughly according to the amount of time I expect that requirement to consume during development.

[4]Technically, the equation I have provided here (2.24) is for *luma*, not *luminance*. Image rescaling papers do however use this expression to compute Y, and refer to it as *luminance*, and so I will stick with their convention.

Requirement	Priority	Difficulty
Construct a Glow architecture with FrEIA.	Low	Low
Construct a Glow architecture with base PyTorch.	Low	High
My Glow implementations achieve similar results on MNIST8.	Low	Medium
Implement IRN's coupling layers with FrEIA/PyTorch.	High	Medium
Construct the complete IRN model, including upscaling /downscaling functions.	High	High
Implement practical functions for training IRN, including computation of loss.	High	Medium
Provide functions and classes to aid the training of IRN, including checkpointing and remote reporting of results.	Medium	Low
Provide functions for evaluating IRN's performance at image rescaling, in particular in terms of PSNR / SSIM.	High	Medium
Train IRN to 10,000 epochs on the DIV2K dataset, achieving results matching or exceeding Xiao et al.'s implementation.	High	High
Devise and implement a Lipschitz-bounds-inspired modification to IRN based on the work of Behrmann et al.	Low	High
Adapt IRN to make its upscaling permissive of JPEG-compression.	Low	Medium
Adapt IRN to make it work as a super-resolution model.	Low	Medium
Demonstrate flaws with standard image rescaling test metrics by making simple modifications that achieve superior results to existing methods.	Low	Medium

Table 2.2: A summary of the success criteria for this project, with descriptions having an emphasis on implementation (over e.g. learning of theory).

2.6 Software Engineering Approach

I used a number of engineering tools and techniques to ensure that the development of the project was organised, free from unnecessary repetition, and safe from risks such as data loss. In particular, I utilised I type of *spiral model* for much of the project, building larger models and training on larger data over time (for example, initially working with a 2-layer IRN on grayscale images only). **Git** was used for version control, with backups of the project saved to Github, my own machine, as well as saved locally on the University's CSD3 servers.

As mentioned in the introduction, the project primarily involved the use of the **FrEIA** framework for invertible architectures within PyTorch. This architecture saved a significant amount of time in the project by providing features such as automatic propagation of log-Jacobian-determinants through layers, as well as providing definitions for some standard invertible operations such as Haar transforms.

Due to the significant computational cost of training a single version of IRN, I included in the training a number of features to track the progress of the model - including interface with the online model-tracking interface *Weights & Biases*. I give an overview of my engineering approach towards training below.

2.6.1 Practical Model Training

While algorithm 2 describes the functional aspects of training - i.e. the code which contributes to the final model - training in practice involves testing and checkpointing procedures to track a

model's progress over time. This is particularly critical when training a highly computationally expensive model such as IRN, whose original authors trained for 7 days on a P100 GPU.

My full training procedure thus produces various results beyond the model parameters for the same of progress tracking. I classify this output into 4 categories based on purpose:

- **Checkpoints** (local): Serialized instances of a model saved to disk. I generate checkpoints using Python's pickle utility. Routine saving was particularly necessitated by the 12h session limit on CSD3 servers.
- **Test results** (remote and local): Test metrics such as PSNR on validation data, and a running average of loss scores over recent samples in training, which is reported to provide rapid oversight on a model's performance.
- **Samples** (remote and local): Samples of LR' and HR' displayed beside LR and HR for qualitative evaluation.
- **Debugging information** (remote local): Particularly challenging samples (exceeding recent loss average by some factor), hardware statistics such as temperature, and measurements of the time taken by key functions.

The ability to measure the time taken by various functions during training was particularly crucial in planning long training sessions. It enabled the identification of bottlenecks in the code, and facilitated an estimate of how long it would take to train the model up to a particular number of epochs.

I plotted graphs estimating model training time across various hardware and parameters according to measured function times. In practice, I found the test function (called periodically on validation data during training) to be a significant bottleneck when its period was too low, taking several minutes at each call. With parameters adjusted, I identified on a configuration that would enable training a full IRN model in several days on an A100.

Remote tracking of models was primarily performed through *Weight & Biases (WandB)*. This is a suite of online tools for tracking the progress of machine learning models; one can submit updates during training, which WandB logs to their database, and these logs can be compared in the form of graphs. Long-running models such as IRN can be monitored less stringently by setting up alerts to trigger at certain loss thresholds, or when the model fails outright.

Model Interface

To enable a development cycle of proposing and training models concurrently and over multiple sessions, I encapsulated hyperparameters and other meta details into .YAML configuration files. In general, I created several config files per "experimental feature" such that I could easily return to any proposed model from eariler in the project.

Checkpoints are used in concert with .YAML config files to save and load models. I give each checkpoint a unique identifier of the form:

```
<timestamp>_<epoch number>_<recent loss>_<wandb id>.pth
```

where `wandb id` is a unique id corresponding to a training "run" in WandB. This allows a user to start and resume runs through a command-line interface (CLI) - as an example of typical usage, one can specify a WandB run to resume and the model interface will search for the most recent checkpoint ending with that id.

GPU Server

Due to the high computational demand of training IRN, this project makes use of the CSD3 (Cambridge Service for Data-Driven Discovery) service provided by UIS. In particular, I used the *Wilkes2* P100 GPU server for the first half of the project, after which I upgraded to *Wilkes3* running A100 GPUs, which saw a roughly 4× improvement in training times.

To remotely queue jobs (which were limited to 12h sessions at my account tier) on CSD3, I created Slurm[5] config files which I ran from an online dashboard or through SSH. The training of large models that produce regular output requires a degree of maintenance, and I found the main bottleneck in this regard to be a local cache stored by WandB agents. To avoid surpassing the CSD3 storage limit during runs, I created a script that regularly cleaned the WandB's *artefact* cache, keeping it below 1GB total.[6]

2.7 Starting Point

I had studied Machine Learning as part of the Computer Science Tripos - though not starting Neil Lawrence's "Deep Neural Networks" course until roughly mid-way through development - and had a basic understanding of deep learning models such as *convolutional neural networks* from my own study. However, I had never used a deep learning framework before (such as PyTorch, FrEIA), nor had I previously encountered the concept of invertible neural networks.

Gaining an understanding of invertible architectures such as *normalizing flows*, as well as nuances of the image rescaling task such as the *SSIM* metric proved to be a significant undertaking in the initial phases of the project. I found myself continuing to learn more about the relevant fields in reading dozens of papers during development.

2.8 Summary

Recap this chapter chronologically, make use of key words defined in the chapter. I could include a graph to make it more appealing; e.g. a histogram of my commits over time.

[5]Slurm Workload Manager is a job scheduler for Linux.

[6]By regularly calling `wandb artifact cache cleanup 1GB`.

Chapter 3

Implementation

3.1 Repository Overview

This section details the structure and content of my Git repository, with an overview provided in table 3.1. Lines were counted using VScodeCounter, and I have excluded from the count any code not written by myself (such as auto-generated tests), as well as any .YAML config files due to their repetitive nature (which otherwise amount to 1200 lines).

The repository is divided into two projects: `mnist_generation/` and `image_rescaling/`. This split is due to the MNIST generation project being intended as a way of learning and exploring *FrEIA* in isolation - as well as tthe fact that the models in that project were to be trained in minutes on a local CPU, rather than in multiple sessions on a GPU server. Consequently, `mnist_generation/` did not require integration with the model saving and progress tracking frameworks of `image_rescaling/`.

The structure within the two projects is a modification of the *Cookiecutter Data Science* structure, which was chosen in part due to *Cookiecutter* being the most popular Python project structure by Github stars.[1] This makes the project structure more familiar to data scientists, and more easily comparable to existing work. Other structures considered include Neil Lawrence's *Fynnesse* template, which I found to be more oriented towards data analysis than constructing modular models.

Looking more closely, I have divided the `models/` folder into `layers/`, `test/`, and `train/` in order to make the project more modular - for example allowing one to train a model for one task and test it on another. This division makes the project more extensible and thus accommodating of any future rescaling tasks and models a person may choose to implement.

3.2 PyTorch/FrEIA Mini Networks

FrEIA (*Framework for Easily Invertible Architectures*) is an extensible framework for constructing invertible neural networks in PyTorch, first created by Ardizzone et al in 2018.[2] It allows one to define complex computation graphs and operators in a structure that keeps track of intermediate results by design (such as the *log Jacobian determinant*, necessary for application of

[1] Cookiecutter has the highest number of Github stars for a Python project template as of May 2022: **web archive**.

[2] FrEIA available at https://github.com/VLL-HD/FrEIA

Path	Purpose	Lines of Code
image_rescaling/		(2574 total)
configs/	Stores: .YAML files for model hyperparameters.	0
data/	Stores: Raw and processed image datasets.	0
output/	Stores: Images and metrics produced during training, testing, and debugging.	0
models/	Stores: Saved models (*checkpoints*).	0
remote/		(317 total)
slurm/	Stores: Slurm config files to run jobs on CSD3.	317
wandb/	Stores: Saved runs and sweep configurations for *Weights & Biases*.	0
src/		(2257 total)
data/	Code: Load and transform datasets.	264
tests/	Code: Unit tests to check functions using `pytest`.	336
utils/	Code: Common functions e.g. finding saved files.	33
visualisation/	Code: Visualise images and results.	94
models/		(1529 total)
layers/	Code: Model layer classes - e.g. coupling layers.	491
test/	Code: Test a model's performance at a task.	371
train/	Code: Train models.	626
model_loader.py	Code: Load model checkpoints and config files.	41
mnist_generation/		(438 total)
output/	Stores: Images and metrics produced during training, testing, and debugging.	0
src/		(438 total)
data/	Code: Load and transform MNIST8.	58
models/	Code: Glow implementations and their trainer.	325
visualisation/	Code: Visualise images and results.	55

Table 3.1: Repository overview.

the change of variables formula). It is also perhaps the most complete interoperable collection of INN layers, which contributes to the increasing reliance on FrEIA in INN research [21] [22].

Before implementing the IRN model, I created two implementations of a normalizing flow architecture known as **Glow** - one with and one without FrEIA. This architecture was chosen because it involves the most typical features of modern normalizing flows (*ActNorm, Permutation, Affine Coupling*), thus providing a solid baseline for comparing the usability and performance of FrEIA over raw PyTorch.

In particular, I trained a Glow model to generate handwritten digits from the *MNIST8* dataset of 8×8px handwritten digits, due to it being a computationally inexpensive task to solve. This made it feasible to implement small-sized models that one can debug weight-by-weight, and allowed more rapid testing and iteration of ideas before beginning work on IRN.

3.2.1 Glow Architecture

Glow is a generative normalizing flow model that combines the invertible layers of *affine coupling*, *permutation* (in the generalised form of 1×1 convolutions), and *activation normalization*. The full Glow model implements their flow in a multi-scale architecture borrowed from Dinh et al's RealNVP model [23]. I chose to omit the multiscale structure for this task, however, primarily due to low-dimensionality of MNIST8 making it superfluous - instead I focus on the three-layer Glow *flow-step* described below.

Activation Normalization (ActNorm)

ActNorm is an invertible affine transformation conceptualised by Kingma et al. [24]. Its aim it to be an invertible imitation of the *Batch Normalization* layer, which is commonly used to stabilise deep learning models by fixing intermediate results to have `standard_deviation=1, mean=0` [25].

ActNorm introduces invertibility by making its coefficients (bias, scale) parameters of the *model* rather than parameters of the *input data*. Through *data-dependent initialization* (Salimans et al. [26]), we can initialise the layer to give the input batch `standard_deviation=1, mean=0`, after which the parameters are changed through gradient descent alone. We have:

$$\mathbf{y} = (\mathbf{x} + \mathbf{bias}) \odot \mathbf{scale} \tag{3.1}$$

$$\log(\det(\frac{\mathbf{dy}}{\mathbf{dx}})) = h \cdot w \cdot \sum(\log|\mathbf{s}|) \tag{3.2}$$

where *bias*, *scale* are learnable parameters initialised to $bias = -mean(x)$, $scale = 1/std(x)$.

Affine Coupling

An *Affine Coupling layer* is a coupling layer that performs an affine transformation for y_2. To perform coupling, one must first split the layer's input into two parts if such parts are not already separate. Glow achieves this by splitting 50/50 along its input's channel dimension, and likewise repairing the two halves of output back into a single tensor at the end using `concat`:

$$\texttt{split}(\mathbf{x}) = (\mathbf{x}_1, \mathbf{x}_2) \quad \text{and} \quad \texttt{concat}(\mathbf{y}_1, \mathbf{y}_2) = \mathbf{y} \tag{3.3}$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{h \times w \times c}$ and $\mathbf{x}_1, \mathbf{x}_2, \mathbf{y}_1, \mathbf{y}_2 \in \mathrm{R}^{h \times w \times \frac{c}{2}}$. Examining affine coupling itself, the layer makes use of two *subnets*, performing the following forwards operation:

$$\mathbf{y}_1 = \mathbf{x}_1 \quad \text{and} \quad \mathbf{y}_2 = \mathbf{x}_2 \odot g(s(\mathbf{y}_1)) + t(\mathbf{y}_1) \tag{3.4}$$

where q, w are arbitrary subnets, and g is a clamp function. In the case of Glow, the clamp function is a sigmoid function with range $[0, 1]$:

$$g(x) = \frac{1}{1 + e^{-(x+2)}}. \tag{3.5}$$

Kingma et al. chose to simplify their affine coupling layers by making use of only one subnet instead of two, taking $q(\mathbf{y}_1)$ and $w(\mathbf{y}_1)$ to be two halves of that subnet's output. Calling this larger subnet p, we have:

$$s(\mathbf{y}_1), t(\mathbf{y}_1) = \text{split}(p(\mathbf{y}_1)), \tag{3.6}$$

and finally an expression for the log-Jacobian:

$$\log(\det(\frac{\mathbf{dy}}{\mathbf{dx}})) = \sum(s(\mathbf{y_1})). \tag{3.7}$$

The specific choice of subnet used for p may vary depending on application, and for my specific 8×8 MNIST task I tried both a shallow *convolutional neural network* (2.12) as used in RealNVP [23], and a simple two-layer *fully-connected network* (2.1) with ReLU activation (2.13). I found the fully-connected network to show more stability and consistency across training sessions, which is perhaps due to the unusually small size of images being learnt. It is for this reason that I primarily focus on Glow models utilising this fully-connected subnet, including for my evaluation in section 4.1.

Permutation

Because each affine coupling layer in Glow performs only an identity transformation on the first half of $\mathbf{x}$'s channels, we need an invertible function that can re-order $\mathbf{x}$'s channel dimension between coupling layers. Otherwise, only a subset of channels would be transformed by affine coupling. Glow achieves this re-ordering by taking a random permutation of the channels - and they generalise this permuation operation by implementing it as a 1×1 convolutional layer.

In other words, for a $h \times w \times c$ sized tensor, the application of $c \times c$ many 1×1 convolutional filters (one for each input and output channel) is a generalisation of permutation. This effectively forms a fully-connected structure between the channels of $\mathbf{x}$ and $\mathbf{y}$ over the tensor of filters:

$$y_{i,j,k} = \mathbf{W}_k \cdot \mathbf{x}_{i,j} \tag{3.8}$$

$$\log(\det(\frac{\mathbf{dy}}{\mathbf{dx}})) = h \cdot w \cdot \log|\det(\mathbf{W})| \tag{3.9}$$

where $\mathbf{W} \in \mathbb{R}^{c \times c}$ contains the layer's filters, and $x, y \in \mathbb{R}^{h \times w \times c}$ are the layer's input and output tensors.

Raw PyTorch Implementation

This implementation required manual provision of forward, inverse, log-determinant, and inverse log-determinant functions for each of Glow's layer types. I created five classes, which pass log-determinant information between layers in order to enable training using maximum likelihood estimation. A simple overview of this class structure is provided in figure 3.1.

FrEIA-PyTorch Implementation

This implementation was a simple matter of using FrEIA's built-in modules to construct a sequential INN supplied with `AllInOneBlock` objects, as seen in figure 3.2.

3.2.2 Training

To train Glow, I used *Adam optimization* with a loss function that combines MLE loss (2.4) given by change-of-variables, with a cross-entropy term encouraging the generative "seed" to conform to a normal distribution. I use the same training code for both FrEIA and raw PyTorch models. The 1797 images of the MNIST8 dataset were shuffled and partitioned into a 1617-image training set and 180-image test set, then trained according to algorithm 1.

Figure 3.1: The primary classes used in my implementation of Glow in PyTorch. `AllInOne` represents a single step of flow (*actnorm* → *permutation* → *coupling*), and the `nn.Module` class represents `AffineCoupling`'s subnet. Black classes are my own, blue are from `torch`.

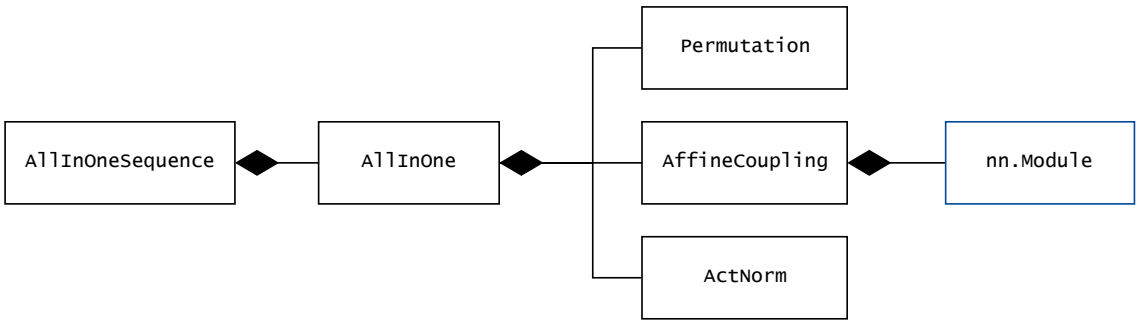
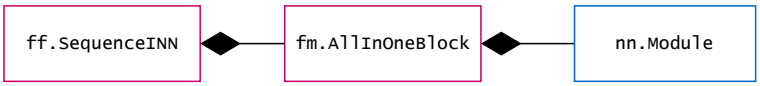


Figure 3.2: The primary classes used in my FrEIA-based implementation of Glow. In this case, the built-in `ff.AllInOneBlock` represents a single step of flow, and the `nn.Module` class represents the subnet we use for coupling. Pink classes are from `FrEIA`, blue are from `torch`.



3.3 Invertible Rescaling Network

My base implementation follows the IRN architecture described by Xiao et al. [11], and makes use of FrEIA to compose invertible layers. Being a common operation, the Haar transform was supplied largely by a built-in class in FrEIA, but more careful attention had to be paid to implementing custom coupling layers, subnets, separation of LR and LD, and correct testing and training for image-rescaling including quantisation.

Splitting of high-frequency data

A typical normalizing flow model is essentially trying to emulate a sequential architecture that processes each of its channels dependent on one another. For example, in our Glow implementation, we randomly permute channels before splitting them at a coupling layer in order to "hide" the fact that each coupling layer is only transforming half of its input. By contrast, IRN is specifically *exploiting* the separation of (downscaled image) LR against (high-frequency information) HF rather than trying to obscure it. It is therefore important that we

Algorithm 1: Training loop for Glow models on MNIST8

Input: inn_θ, D =MNIST8 up to 1617, $lr = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $max\_iter = 10000$.
Result: inn_θ having learned parameters θ.
for max_iter **do**
  $x \leftarrow \texttt{next}(D)$
  $y,\ j \leftarrow \texttt{inn}_\theta(x)$
  $loss \leftarrow \frac{1}{2}\sum_{i=0}^{n}(y_i^2) - j$
  $\Delta \leftarrow \texttt{backprop}(\theta, loss)$
  $\theta \leftarrow \texttt{Adam}(\theta, \Delta, lr, \beta_1, \beta_2)$
end

maintain this separation throughout the network, giving the model a kind of inductive bias towards producing disentangled LR' and LD.

One way to implement this seperation would be to construct a computation graph in which each coupling layer is a custom `FrEIA.framework.Node` that accepts two inputs and produces two outputs. This means that we only have to `split` and `concat` inputs once per Haar transform:

$$input \rightarrow haar \rightarrow split \rightarrow\rightarrow coupling \rightarrow\rightarrow ... \rightarrow\rightarrow coupling \rightarrow\rightarrow concat \rightarrow haar \rightarrow split \rightarrow\rightarrow ... \rightarrow\rightarrow output$$

However, as this structure has formed a simple *path multigraph*,[3] we can simplify it by instead constructing a sequential INN that passes input between layers in a line. We order the Haar transform's output by wavelet such that the first c channels of its $4c \times \frac{h}{2} \times \frac{h}{2}$ output contain LR, and make our coupling layers take the first c channels of their input to be their x_1, and likewise the first c of output to be y_1. Although this has simplified the structure of the network, there is the consequence of now performing a `split` and `concat` within each coupling layer rather than around each Haar transform:

$$input \rightarrow haar \rightarrow coupling \rightarrow ... \rightarrow coupling \rightarrow haar \rightarrow ... \rightarrow split \rightarrow\rightarrow output$$

Dense convolutional subnet

For the three subnetworks of IRN's enhanced affine coupling layers, we can use *densely connected convolutional blocks (DenseNets)* [27], which have been found to be particularly effective at image super-resolution tasks [28]. A DenseNet is a CNN in which each convolutional layer is connected to each previous convolutional layer with *skip connections* or *shortcut connections*. This encourages re-use of extracted features, ultimately reducing the number of parameters required, as the network avoids re-learning features. In contrast to *residual networks*, the features of previous layers are never combined through summation; rather they are just passed outright to subsequent layers through concatenation.

The specific architecutre of the DenseNets we use as subnets in IRN is shown in figure 3.3.

3.3.1 Data Processing

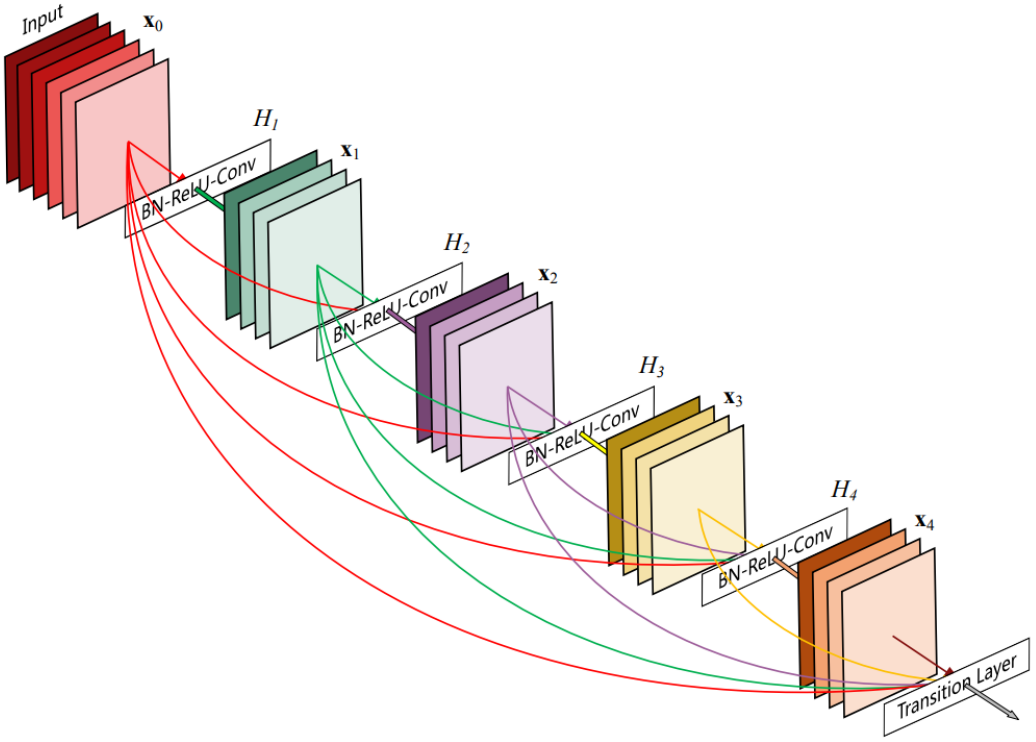
We require a way of loading individual samples from a dataset into GPU memory. Being a common problem, it is unnecessary to re-implement this data loading from scratch, and I was primarily able to make use of `torch`'s `DataLoader` class for batching, shuffling, and iteration with multiple worker processes.

To reduce overfitting and make our model more robust - in other words a form of regularization [29] - we introduce a number of **augmentations** into training. These randomly transform the data with four types of operations, supplied from `torchvision`:

- Random **cropping** taking a random 144×144×3 slice of input data.
- Random horizontal and vertical **flipping**.

[3] A graph is a *path graph* iff there exists an ordering of vertices $(v_1, v_2, ..., v_n)$ such that all $n-1$ edges are distinct and of the form $(v_i \rightarrow v_{i+1})$ where $i < n$. In other words, the graph forms a straight line. A *path multigraph* is a variant of this in which edges do not have to be distinct (one can have *parallel edges*).

Figure 3.3: A DenseNet of 5 layers. This diagram is sourced from Huang et al.'s original paper [27].



- Random 90-degree **rotation**.

We also perform a normalisation step on loaded data, restricting it from $[0, 255]$ to $[0, 1]$, which is standard for ensuring fast convergence in deep learning models.

Scaling and divisibility

A problem I discovered with data loading for image rescaling specifically is that of *how to rescale images by a factor by which they are not divisible*. By making use of Haar downscaling modules, IRN is restricted to rescaling by powers of 2 - and to compute our test metrics after upscaling, we require both the GT and reconstructed HR images to be of the same size.

I believe that the most sensible solution is to at some point crop the larger image to facilitate the comparison, as the crop would take off no more than $scale/2$ pixels from either dimension.[4] I implemented this as a data processing step, creating a custom `torchvision` transform that performs

$$x = \texttt{crop}(x, scale \cdot \lfloor h/scale \rfloor, scale \cdot \lfloor w/scale \rfloor) \tag{3.10}$$

making use of centre-cropping function `crop(img, new_h, new_w)`.

[4]To see this, consider that our worst-case cropping will be for a ground-truth image having dimensions of length $m \cdot scale + scale/2$ for some integer m. This is because $scale/2$ is of course the greatest a number can deviate from any factor of $scale$ in absolute terms.

3.3.2 Training

I trained IRN using the Adam optimizer augmented with gradient clipping, weight decay, and learning-rate scheduling. Due to its significant computational cost, I included in the training a number of features to track the progress of the model, including interface with the online model-tracking interface *Weights & Biases*.

Initialisation

The parameters of IRN's DenseNet subnets are initialised using *Xavier normal* initialisation [30] (3.11), while the Haar transform filters are simply initialised to the matrices defined in equation (2.21).

Considering a layer taking in a tensor of in_n elements, processing it with a weight tensor $\mathbf{W}$, and outputting a tensor of out_n elements, we have:

$$\texttt{xavier\_normn\_init}(\mathbf{W}) \sim \mathcal{N}\left(0, \frac{1}{in_n + out_n}\right)^{\texttt{size}(\mathbf{W})} \tag{3.11}$$

I also tested an alternative to *Xavier* initialisation, *Kaiming*, which is intended to avoid vanishing gradients in networks that use asymmetric non-linear activation functions such as ReLU. However, in practice I found Kaiming to have a slight negative effect on training performance in IRN, perhaps due to the vanishing gradients problem already being alleviated by DenseNet's skip connections.

Quantization

To extract the three terms used in IRN's loss function (table 2.1), we must utilise a training loop that first downscales a batch of images to LR', and then upscales them to the reconstructed HR'.

A subtle but important detail of training is the necessity of **quantization**. While the model may produce an LR' that contains values from throughout the $[0, 1]$ range and beyond it, in reality the intention with image rescaling is to store LR' as an image file. Hence, we should assume LR' is quantized to have 8-bit $[0, 1]$ values before upscaling, so that the model doesn't learn to depend on features it will not have access to in practice. Let us define our quantization function:

$$\texttt{quant}(\mathbf{x})_{i,j,k} = \frac{\lfloor 255 \cdot \min(\max(x_{i,j,k}, 0), 1) + \frac{1}{2} \rfloor}{255} \tag{3.12}$$

An immediate issue when using quantization in a gradient-descent-optimised model is the fact that `quant` is not differentiable. To solve this, I created a Straight-Through Estimator (STE) [31] class which treats `quant` as having an identity gradient for backpropagation. This means that, for a function f, the backpropagation algorithm treats STE as follows:

$$\texttt{backprop}(\theta, \texttt{STE}(f, x)) = \texttt{backprop}(\theta, x), \tag{3.13}$$

where $\texttt{STE}(f, x)$ performs $f(x)$.

Another important detail of quantization is that it is, slightly counter-intuitively, important to *not* give our loss function quantized data. i.e. we should perform $L1(LR' - LR)$ and $L2(HR' - HR)$, not $L1(\texttt{quant}(LR') - LR)$ and $L2(\texttt{quant}(HR') - HR)$. The reason for this is twofold:

- Because quantized data is clipped to $[0, 1]$, our gradient descent optimizer would be unable to detect exploding values.
- Quantization simply reduces the amount of information that the loss function obtains about a model's performance due to use of rounding. For a target value of 16.000/255, the same loss would be computed for LR's having value 15.999/255 and value 15.510/255.

Optimizations

It is worth mentioning a few smaller optimizations we use during the training procedure.

- **Weight Decay** adds an L2-loss (2.7) term of the network's weights to our loss function - this is known as *L2 Regularisation*, a classic technique for reducing overfitting by punishing variance amongst parameters. With weight decay `weight_decay` and weights w, we have:

$$new\_loss = loss + \texttt{weight\_decay} \cdot L2(w) \tag{3.14}$$

  In the original paper, IRN trains with $\texttt{weight\_decay}=1e-5$.

- **Gradient Clipping** is a means of preventing gradient explosion by forcing gradients below a maximum given by hyper-parameter `max_g`. I define two types of gradient clipping given gradients Δ:

$$\texttt{clip\_norm}(\Delta, \texttt{max\_g})_i = \Delta_i / \max(\boldsymbol{\Delta}) \ \textbf{if} \ \max(\boldsymbol{\Delta}) > \texttt{max\_g}, \ \Delta_i \ \textbf{otherwise}. \tag{3.15}$$

$$\texttt{clip\_value}(\Delta)_i = \texttt{max\_g} \ \textbf{if} \ \Delta_i > \texttt{max\_g}, \ \Delta_i \ \textbf{otherwise}. \tag{3.16}$$

  In the original paper, IRN trains with `clip_norm` at `max_g=10`.

- **Learning Rate Scheduling** we can improve convergence by reducing learning rate periodically, allowing the optimiser to "narrow in" on a local minima. The original IRN paper halves their learning rate every 100k batches.

IRN Training Loop

With quantization and the image-rescaling process taken into consideration, we obtain the core training procedure outlined in algorithm 2. The IRN paper initialises the λ_{guide} coefficient to $scale^2$ because LR' has $1/scale^2$ as many pixels as HR', hence the summation-based guidance loss would otherwise have $1/scale^2$ the impact of our reconstruction loss.

Hyperparameters

Though hyperparameter tuning is not the focus of this project, it is worth mentioning the model's main hyperparameters and their effects, which I provide an overview of in table 3.2. The hyperparameters for any of my own extensions that were not part of the IRN model described by Xiao et al. are provided in section 3.4.

One of the important tradeoffs to make in these models is the choice of **batch size** versus **learning rate**. In the broadest sense, both parameters trade off speed of learning in no. of epochs against stability, but increasing batch size comes with the added benefit of better utilising parallelizability in GPUs.[5] In particular, this tradeoff is constrained for IRN by the

[5]Specifically, for my hardware, utilising CUDA cores.

Algorithm 2: Core training procedure for IRN

Input: irn_θ, D, $scale$, $lr = 2\text{e}-4$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\lambda_{guide} = scale^2$, $\lambda_{recon} = 1$, $\lambda_{distr} = 1$, $weight\_decay = 1\text{e}-5$, $max\_g = 10$, $max\_iter = 500000$.

Result: irn_θ having learned parameters θ.

for *iter* ***in*** *max_iter* **do**

$x \leftarrow \texttt{next}(D)$

$y \leftarrow \texttt{bicubic}(x, 1/scale)$

$(y',\, z) \leftarrow \texttt{irn}_\theta(x)$ `// Downscale x to y'`

$y'_Q \leftarrow \texttt{STE}(\texttt{quant}, y)$ `// Quantize y' without altering backprop gradient`

$z' \leftarrow \mathcal{N}(0,1)^{\texttt{size(z)}}$ `// Generate latent data estimate z' from Normal`

$x' \leftarrow \texttt{irn}_\theta^{-1}((y'_Q, z'))$ `// Upscale y'_Q to x'`

$loss \leftarrow \lambda_{guide} \cdot L1(y' - y) + \lambda_{recon} \cdot L2(x' - x) + \lambda_{distr} \cdot \sum z^2$

$\Delta \leftarrow \texttt{backprop}(\theta, loss)$

$\Delta \leftarrow \texttt{clip\_norm}(\Delta, max\_g)$

$lr \leftarrow \texttt{scheduler}(lr, iter)$

$\theta \leftarrow \texttt{Adam}(\theta, \Delta, lr, \beta_1, \beta_2, weight\_decay)$

end

relatively large size of the model (4.4 million parameters) and the nature of its training set consisting of large (close to 4K) images. A batch size of 16 is chosen due to being roughly the maximum that is possible with the 16GB of GPU memory available on a P100 GPU.

The hyperparameters described in this project can be tuned through *Sweeps* organised by **Weights & Biases**. These are automated searches for optimal hyperparameters that train and compare a number of models in parallel; I experimented with both *grid-search-based* and *Bayesian-inference-based* sweeps. Unfortunately, however, the large size of IRN made it more difficult to accurately tune hyperparameters, as we cannot afford to fully train dozens of large-scale models - I describe a possible workaround to this in Future Work 5.3.

3.3.3 Model Testing

In order to test a variety of models (such as IRN vs bicubic) against a variety of rescaling-related tasks (such as super-resolution), I wrote a generic test routine that can be supplied with different *sample* and *metric* procedures as higher-order functions, as well as with different datasets. A simple overview is given in algorithm 3.

Details regarding *unit testing* of my code, in particular testing of the *rescaling test metrics* themselves, can be found in appendix B.

CPU-GPU Discrepancy

One hidden detail that caused a minor delay in the project came from the fact that `torchvision`'s SSIM implementation produces different results on GPU compared to CPU, leading to the impression that my models were slightly underperforming in terms of SSIM. Specifically, I found SSIM scores to decrease by around 1% on GPU, as shown in table 3.3.

A likely explanation for this discrepancy is rounding error. GPUs should perform their matrix multiplication as a *reduction* (e.g. $((a+b)+(c+d))$) to utilise parallelisation, while CPUs

Hyper-Parameter	IRN Default	Description
Of the model:		
`inv_per_ds`	8	No. of invertible blocks (enhanced affine coupling layers) per Haar transform.
`scale`	4	Rescaling factor. Downscaled images have $1/scale^2$ the number of pixels. The number of Haar transforms in an IRN model equals $\log_2(scale)$.
Of training:		
`batch_size`	16	No. of images included each single training sample.
`clip_norm_max_g`	10	`max_g` parameter used in `clip_norm` (3.15).
`img_size`	144	Width and height of crops used in training.
`initial_lr`	2e-4	Learning rate at epoch 0.
`lambda_recon`	1	Coefficient of reconstruction loss.
`lambda_guide`	16	Coefficient of guidance loss.
`lambda_distr`	1	Coefficient of distribution loss.
`lr_gamma`	0.5	Factor by which learning rate decreases.
`lr_milestones`	[100k,200k, 300k,400k]	Batch numbers at which learning rate is decreased by the scheduler according to `lr_gamma`.
`seed`	10	Manual seed controlling random number generation in `torch`, `python`, and `numpy`.
`weight_decay`	1e-5	Coefficient of L2 weight regularisation (3.14).

Table 3.2: Base hyperparameters of IRN.

Algorithm 3: Basic testing loop for rescaling-related tasks.

Input: $sample\_fun$, $metric\_fun$, D, $verbose = False$.
Result: Test metrics, reported at a level of detail determined by *verbose*.
$test\_scores \leftarrow$ `[]`
for *iter* ***in*** D **do**
  $x \leftarrow$ `next`(D)
  $(x, y, z, x', y') \leftarrow sample\_fun(x)$
  $t \leftarrow metric\_fun(x, y, z, x', y')$
  $test\_scores = test\_scores$ `@ [`t`]`
end
Report results aggregating *test_scores* - e.g. min, max, avg - according to *verbose*.

perform it sequentially (e.g. $(((a + b) + c) + d)$), which may give rise to different floating point errors - particularly when working with fewer bits.

CPU is the standard device used for testing in other image rescaling implementations, though it does unfortunately add a significant overhead - SSIM takes roughly 0.5s per image compared to a fraction of this on GPU. For this reason I added an alternate "**fast**" testing mode that runs on GPU instead. This presents an interesting tradeoff during training: one can sacrifice

Device	PSNR (RGB)	PSNR (Y)	SSIM (RGB)	SSIM (Y)
CPU	32.95	35.07	0.9133	0.9318
GPU	32.95	35.07	0.9061	0.9269

Table 3.3: Metrics obtained by testing Xiao et al.'s 4× IRN implementation on the DIV2K dataset. Underlined are the numbers reported by the authors (they did not record RGB-channel results). This shows that I need to compute SSIM on CPU to gather results comparable to the authors.

performance (running regular CPU tests) for accurate information.

3.4 Extensions

3.4.1 Modified Clamp for Lower Lipschitz Bounds

My most theoretically-involved extension is a modification to *Enhanced Affine Coupling*'s clamp function that reduces the layer's local Lipschitz bounds. I make use of a derivation of the layer's bounds in appendix D.2, which is my own extension of Behrmann et al.'s derivation of the Lipschitz bounds for *Affine Coupling* layers [32].

As the motivation behind these modifications is rather involved, and requires theory that is isolated from the rest of the project, I explain it more deeply in appendix D.2. However, it is enough to understand that I believe we can improve stability in *enhanced affine coupling layers* by discouraging small values of their clamp function g.

The simple modification that I propose is to *tighten* the clamp around $x = 0$, such that small values of x don't result in quite so small of a result in g. With our sigmoid function, the simplest way to achieve this effect is by compressing g along the x axis, shown in table 3.4.

Original IRN Clamp		Modified IRN Clamp (Proposal #2)
$g(x) = \frac{1}{1+e^{-x}} \cdot 2 - 1.$	$\implies$	$g_{\text{mod}}(x) = g(2x)$

Table 3.4: One possible Lipschitz-inspired modification to Glow's affine coupling clamp, discouraging smaller values. I call the IRN model trained with this modification **IRN-T**.

Although this modification only discourages, not cuts out entirely, smaller values, I found that it **does** have a positive effect on training, which I discuss in evaluation (4).

3.4.2 JPEG-Compression

I tested the model's ability to reconstruct JPEG-compressed low-res images by making use of a Straight-Through Estimator (3.13) during training. Specifically, in order to make IRN resilient to a variety of levels of compression (JPEG intensity being measured on a quality scale from 1 to 100), we can apply a random level of compression to LR' images, which I sample from a uniform distribution.

In other words, we can train IRN to be resiliant to compression by replacing the $x' \leftarrow \texttt{irn}_\theta^{-1}((y'_Q, z'))$ step with $x' \leftarrow \texttt{irn}_\theta^{-1}((\texttt{STE}(\texttt{random\_compress}, y'_Q), z'))$.

3.4.3 Loss "Flipping" Schedule

I experimented with an alternating loss schedule by which we rotate the guidance and reconstruction loss coefficients every few epochs. The loss coefficients move through three phases, parameterised by a `loss_flip_scale` factor as shown in table 3.5.

Phase 0	$\lambda_{guide} = \texttt{lambda\_guide}$	$\lambda_{recon} = \texttt{lambda\_recon}$
Phase 1	$\lambda_{guide} = \texttt{lambda\_guide}$	$\lambda_{recon} = \frac{\texttt{lambda\_recon}}{\texttt{loss\_flip\_scale}}$
Phase 2	$\lambda_{guide} = \frac{\texttt{lambda\_guide}}{\texttt{loss\_flip\_scale}}$	$\lambda_{recon} = \texttt{lambda\_recon}$

Table 3.5: Phases of loss scheduling. During training, the current phase is indexed by $\left\lfloor \mod (\frac{\texttt{total\_epochs}}{\texttt{loss\_flip\_period}}, 3) \right\rfloor$.

3.4.4 Y-Channel Loss Function

I facilitated varying emphasis on Y-channel reconstruction by introducing a `y_channel_usage` hyperparameter. Abbreviating to ych, this hyperparameter is utilised in the loss function like so:

$$\begin{aligned} L_{total} =& \lambda_{guide} \cdot ((1 - ych) \cdot L^{RGB}_{guide} + ych \cdot L^{Y}_{guide}) && (3.17)\\ &+ \lambda_{recon} \cdot ((1 - ych) \cdot L^{RGB}_{recon} + ych \cdot L^{Y}_{recon}) && (3.18)\\ &+ \lambda_{distr} \cdot L_{distr}, && (3.19) \end{aligned}$$

where L^Y denotes loss calculated using only the Y-channel. In 4.2.3, I evaluate an IRN model trained for 10k epochs on DIV2K with `y_channel_usage`=0.5.

3.5 Summary

This chapter elaborated on the core components of my FrEIA and PyTorch-based Glow implementations (section 3.2), as well as details of the IRN model (section 3.3), and lastly detailing through a few extensions (section 3.4).

As a further summary of my extensions, in appendix A.1 I provide an overview of all extensions (with hyperparameters) that I implemented which were not part of the original IRN described by Xiao et al. [11] I also provide a brief summary of each extension I made to IRN.

Chapter 4

Evaluation

4.1 Mini-MNIST Generation

I tested both FrEIA and raw PyTorch against the task of training a Glow-inspired normalizing flow model to generate 8×8 handwritten digits (MNIST8). The purpose of this preparatory section of the project was to determine FrEIA's suitability at constructing normalizing flow models, and to contrast its performance with my own PyTorch solution.

First, I compared the quality of the images produced by each network. Because we are comparing a distribution of generated images against a distribution of source images, rather than making a one-to-one comparison between the network's output and some ground truth, we cannot evaluate the networks using a metric like PSNR.

Instead, we must compare using either *no-reference metrics* on the network's generated images over many random draws, or by computing *log-likelihood* values on their inversely generated latent data over an MNIST8 test dataset. We can perform the latter using the change of variables formula (2.16).

I performed both forms of testing. For a no-reference metric, I used the *Frechet Inception Distance (FID)* score, which compares two distributions of images by running the images through a pre-trained convolutional neural network (*Inception v3*) and comparing the activation of its neural layers [33]. The results of these tests are shown in table 4.1.

	FID	Mean log-likelihood
FrEIA	**0.76**	**-104**
PyTorch	0.81	-102

Table 4.1: FrEIA vs PyTorch Glow models at MNIST8 digit generation, trained for 5 epochs, averaged over 5 runs of shuffled training data. FID is the *Frechet Inception Distance*. Superior results are denoted in **bold**.

These results firstly demonstrate that both models were able to learn the training distribution, with their FID after 5 epochs being significantly lower than the score of FID=∼5 for untrained models. The end results were relatively similar, but the FrEIA implementation performing better in terms of FID. This is most likely down to minor discrepancies in implementation, such as choice of clamp function or parameterisation of the 1×1 convolutional layer.

I also present a few qualitative samples of the two models' MNIST8 generation results in figure 4.1. As the generation process is random, it is not possible to directly contrast the outputs of the two models.

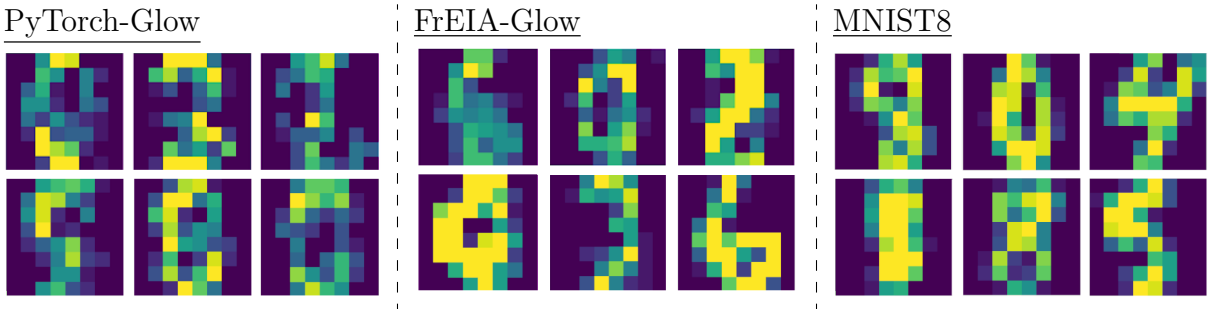


Figure 4.1: Samples of the Glow models' MNIST8 generation output (left and centre) after 5 epochs of training, with samples of the original dataset (right) provided for comparison.

Performance Comparison

In terms of performance, both models ran in similar time - taking around **0.0037s**, as shown in figure 4.2. This indicates that the overhead introduced by FrEIA is negligible. The timing in PyTorch was slightly more variable however, having a standard deviation of $\sigma = 1.21\text{e}-4s$ compared to FrEIA's $\sigma = 1.05\text{e}{-4}s$ over 5 training runs.

Lastly, on a subjective level, I found that the greatest advantage offered by building invertible architectures in *FrEIA* over raw *PyTorch* was the ability to automatically compute log Jacobian determinants - particularly as this determinant (and inverse determinant) is often not as well documented as the forward operation itself in invertible layers. The efficiency gained by making use of FrEIA's built-in library of invertible operators is partly illustrated in the difference it made to line count - with my FrEIA Glow definition constituting a mere ∼15 lines compared to raw PyTorch's near 200.

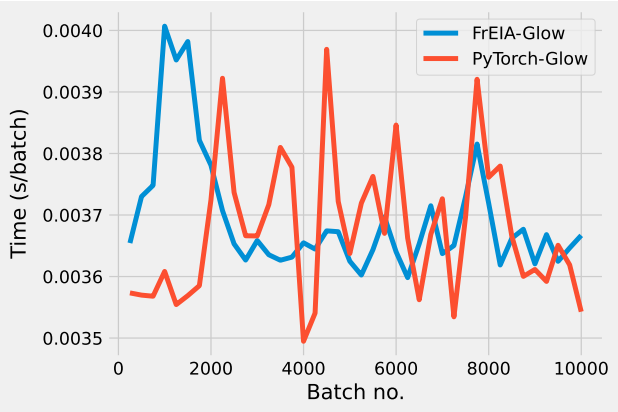


Figure 4.2: Performance in a single representative training run of the Glow architecture in *FrEIA* versus *PyTorch* - in terms of time taken to process a sample from MNIST8 on CPU. One can see that the overall differences are negligible, though my PyTorch implementation has slightly higher variance.

Summary

In conclusion, I determined that FrEIA was a suitable framework for implementing IRN, given it being both as *performant* as my own PyTorch implementation of Glow, and being as *successful* at the task of generating handwritten digits. This indicates that the framework has not sacrificed performance nor representational ability for its more flexible features such as *automatic propagation of log Jacobian determinants*.

4.2 IRN for Image Rescaling

In this section, I demonstrate that I met my success criteria of *achieving image rescaling results matching the original IRN paper after training on a GPU server*, and that my extensions can both improve on the original IRN paper and challenge state-of-the-art image rescaling results.

4.2.1 Matching the Original IRN

In table 4.2, I demonstrate the *image rescaling* performance of my IRN model, comparing its results against five standard image datasets and a number of recent super-resolution and downscaling models.

Included are the test scores of my FrEIA-based implementation of IRN (described in section 3.3), which was the original focus of the project. We can see that my model's PSNR/SSIM scores are within an acceptable error range of the original implementation described in Xiao et al.'s paper [11] - my original target being ±3dB PSNR / ± 0.1 SSIM on DIV2K.

Down & Upscaling	Scale	Param	Set5	Set14	BSD100	Urban100	DIV2K
Bicubic & Bicubic	2×	-	33.66 / 0.9299	30.24 / 0.8688	29.56 / 0.8431	26.88 / 0.8403	31.01 / 0.9393
Bicubic & EDSR	2×	40.7M	38.20 / 0.9606	34.02 / 0.9204	32.37 / 0.9018	33.10 / 0.9363	35.12 / 0.9699
TAD & TAU	2×	-	38.46 / -	35.52 / -	36.68 / -	35.03 / -	39.01 / -
CAR & EDSR	2×	51.1M	38.94 / 0.9658	35.61 / 0.9404	33.83 / 0.9262	35.24 / 0.9572	38.26 / 0.9599
IRN (Xiao et al.)	2×	1.66M	43.99 / 0.9871	40.79 / 0.9778	41.32 / 0.9876	39.92 / 0.9865	44.32 / 0.9908
IRN (Mine)	2×	1.66M	43.98 / 0.9859	40.69 / 0.9759	41.20 / 0.9850	39.93 / 0.9836	44.29 / 0.9898
Bicubic & Bicubic	4×	-	28.42 / 0.8104	26.00 / 0.7027	25.96 / 0.6675	23.14 / 0.6577	26.66 / 0.8521
Bicubic & EDSR	4×	43.1M	32.62 / 0.8984	28.94 / 0.7901	27.79 / 0.7437	26.86 / 0.8080	29.38 / 0.9032
Bicubic & ESRGAN	4×	16.3M	32.74 / 0.9012	29.00 / 0.7915	27.84 / 0.7455	27.03 / 0.8152	30.92 / 0.8486
TAD & TAU	4×	-	31.81 / -	28.63 / -	28.51 / -	26.63 / -	31.16 / -
CAR & EDSR	4×	52.8M	33.88 / 0.9174	30.31 / 0.8382	29.15 / 0.8001	29.28 / 0.8711	32.82 / 0.8837
IRN (Xiao et al.)	4×	4.35M	36.19 / 0.9451	32.67 / 0.9015	31.64 / 0.8826	31.41 / 0.9157	35.07 / 0.9318
IRN (Mine)	4×	4.35M	36.12 / 0.9439	32.51 / 0.8971	31.59 / 0.8779	31.61 / 0.9152	35.04 / 0.9295

Table 4.2: Image rescaling results measured in PSNR / SSIM scores in the Y Channel over a number of datasets. These results demonstrate that my implementation, IRN (Mine), is within an acceptable error range of the author's reported results, hence the implementation is faithful.

Importantly, it is clear that both implementations of IRN far exceed the image rescaling results of methods that do not model upscaling and downscaling jointly (e.g. CAR & EDSR) - an expectation which we described in the introduction.

In figure 4.4, I present a graph of IRN's training procedure. Interestingly, we can note that IRN was able to achieve scores surpassing 33dB of PSNR-Y in fewer than 70 epochs - meaning even within a few hours of training it had exceeded my original aims of being ±3dB from 35.07dB on DIV2K.

4.2.2 Exceeding the Original IRN

Next, I present results demonstrating that my extensions improve on the original IRN in several metrics. I also highlight fallacies of the standard way of evaluating image rescaling models.

Figure 4.3: 4× rescaling of img 1 of the B100 dataset, evaluated in Y-Channel PSNR / SSIM against the Ground Truth. We can observe that my IRN implementation achieves results almost identical to those reported by Xiao et al. (IRN), and both implementations significantly outperform super-resolution techniques such as ESRGAN [28].

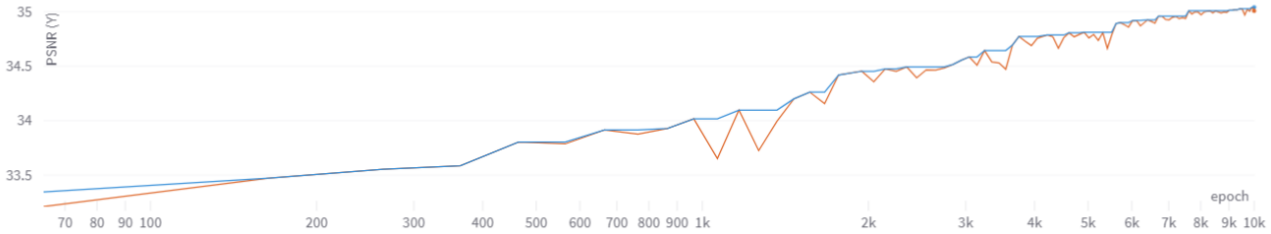


Figure 4.4: My IRN model was trained on DIV2K's training set for 10k epochs. In this graph, we can see IRN's Y-Channel PSNR score on the DIV2K test set increase at an exponential rate, appearing linear against the logarithmic epoch axis. This indicates that we may be able to see further improvements in image rescaling given an (exponentially) longer training time.

For the purposes of this project, we will focus on 4× rescaling - this is because it is a harder problem than 2×, and because I wish to avoid excessive retraining of expensive models. Training a 2× version of each proposed architecture would roughly double the cost of the project. Further, I focus specifically on the DIV2K dataset, as it is sufficiently diverse to communicate my arguments.

Table 4.3 presents the test scores of my IRN extensions mentioned in 1.1, as well as results on existing models not reported by previous papers. This is rather a dense table, but in particular I would like to highlight three important findings:

- My IRN implementation reproduces results comparable to the original authors' results, within a small error range.
- My **IRN-Y** model can challenge state-of-the-art results (HCFlow) while lagging behind in RGB-channel scores. In the table, we can see it achieves a state-of-the-art Y-channel PSNR score of 35.23, but performs below the level of IRN in RGB channels. This suggests that the current way in which image rescaling models are evaluated may be insufficient, as none of the four most high-scoring image-rescaling papers in the field (IRN [11], HCFlow [13], FGRN [12], AIDN [14]) currently report scores in the RGB-channel.

- My **IRN-2T** model offers a slight improvement on the results of my IRN implementation for high-res images (35.04 → 35.09 Y-channel PSNR), but seems to underperform when it comes to low-resolution PSNR scores (44.06 → 42.64 Y-Channel PSNR), again suggesting a failure of standard image rescaling metrics - as it is not typical for authors to compare scores for their downscaled images across models. In figure 4.5, I present more compelling results for IRN-2T, suggesting that tightening the clamp in enhanced affine coupling layers may still be a legitimate way of reducing their Lipschitz bounds.

			DIV2K Upscaling		DIV2K Downscaling	
Downscaling & Upscaling	Scale	Param	Y-Channel	RGB	Y-Channel	RGB
Bicubic & Bicubic	4×	-	28.11 / 0.7746*	26.69 / 0.7524	∞ / 1	∞ / 1
Bicubic & EDSR	4×	43.1M	29.38 / 0.9032	-	-	-
Bicubic & ESRGAN	4×	16.3M	30.92 / 0.8486	-	-	-
CAR & EDSR	4×	52.8M	32.82 / 0.8837	-	-	-
IRN (Xiao et al.)	4×	4.4M	35.07 / 0.9318	32.95 / 0.9133	**44.38** / *0.9933*	*37.40* / *0.9676*
HCFlow	4×	4.4M	*35.23* / **0.9346**	**33.06** / **0.9162**	36.19 / 0.9785	29.50 / 0.8819
IRN (Mine)	4×	4.4M	35.04 / 0.9295	32.96 / 0.9111	*44.06* / 0.9928	**37.60** / **0.9705**
IRN-Y (Mine)	4×	4.4M	**35.25** / *0.9337*	32.59 / 0.9098	44.02 / **0.9950**	34.62 / 0.9513
IRN-2T (Mine)	4×	4.4M	35.09 / 0.9320	*32.98* / *0.9143*	42.64 / 0.9917	36.85 / 0.9670

Table 4.3: Image rescaling results measured in PSNR / SSIM. Yellow cells are results not previously reported in other papers, **bold** denotes the best result in a column and *italics* the second-best. The upscaling scores are comparing the model's reconstructed high-res image (HR') to the ground truth (HR), while the downscaling scores compare the model's low-res image (LR') to bicubic downscaling (LR) as a test of quality.

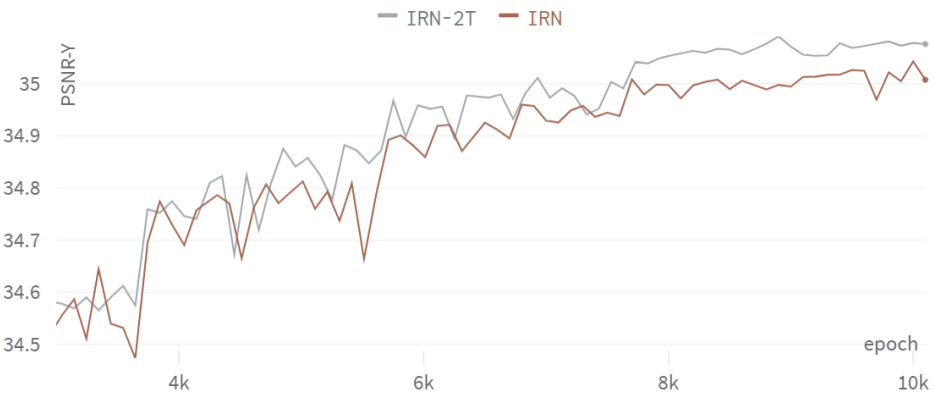


Figure 4.5: This graph compares the PSNR (Y) scores of the basic IRN model against my **IRN-2T** model, which makes use of a modification in its clamp function (table D.1) to reduce its Lipschitz bounds. Results are recorded on the DIV2K test set during training. We can see that while both models end up reaching a score just above 35dB PSNR, the IRN-2T model achieves this far sooner into training - managing 35.04dB after only 7900 epochs, compared to the basic IRN model taking around 10,000. Although further testing of my modification is required, this result does offer a preliminary indication that it can provide benefits in training.

Warning: In an ideal world, these models would be averaged across *multiple* training runs, not one. As a result, these results should be taken as indicative, not conclusive.

Downscaling Metrics

It is worth noting that the FGRN paper [12] is an example of an image rescaling paper which does in fact quantitatively compare their model's downscaled images against the quality of other models. However, the authors do so by computing no-reference scores NIQE and PIQE to evaluate LR image quality - these metrics do not take into account deviation from original image - and so we can "cheat" their metric by taking LR to be any quality image that does not match the original.

Ideally, one would use as a downscaling test metric some comparison against the *original* high-resolution image, but this task is an ongoing research problem [8]. For now, in this paper, we can check for deviation in our downscaled image by comparing against bicubic downscaling using PSNR and SSIM. As IRN was trained to replicate bicubic downscaling (2.1), comparing our low-resolution images to bicubic downcaling is also informative of the training process.

4.2.3 Criticism of Test Metrics with IRN-Y

In section 3.4.4, I described how I implemented an image rescaling model which *specifically* targets strong reconstruction and downscaling scores in the Y-channel of images - calling this model **IRN-Y**. In table 4.3, we can observe my IRN-Y model challenging state-of-the-art HCFlow in its Y-channel image rescaling scores, while lagging being in RGB. This is problematic, as none of the top recent image rescaling papers (IRN [11], HCFlow [13], FGRN [12], AIDN [14]) report their RGB channel scores.

Y-Channel Metrics Have Gaps

Let us examine why restricting ourselves to the Y-channel is unsuitable for the task of measuring perceptual similarity across potentially highly deviating images. It is intuitive that while Y-channel metrics have good ***recall*** (i.e. images that are perceptually similar are likely to be Y-channel similar), they have poor ***precision*** (i.e. images that are Y-channel similar are not necessarily perceptually similar). I summarise this in table 4.4.

	High score	Middling score	Low score
Great solutions	✓	×	×
Acceptable solutions	✓	✓	×
Unacceptable solutions	✓	✓	✓

Table 4.4: Properties of the PSNR-Y metric at evaluating solutions to the task of perceived image similarity. This represents the fact that one can find two images that are identical in the Y-channel ("High score") but highly opposed in RGB ("Unacceptable") - for example, a greyscale version of a coloured image.

It should also be noted that the primary image-rescaling models described in this paper - IRN and HCFlow - are technically optimized towards solving the task of *RGB* image reconstruction, not necessarily *perceptual* reconstruction, owing to the lack of perceptual metrics used in their loss functions. This makes it particularly strange to evaluate their performance using only Y-channel metrics, which are intended as rough approximations of perceptual difference.

	High score	Middling score	Low score
Great solutions	✓	×	×
Acceptable solutions	×	✓	×
Unacceptable solutions	×	×	✓

Table 4.5: Properties of a hypothetical "perfect" metric for evaluating solutions to the task of perceived image similarity. In this case, the score would more or less exactly correlate with human opinion.

Y-Channel Metrics Cause Accidents

While it is true that, for many tasks, no "perfect" metric exists, one thing that makes the image rescaling Y-channel PSNR/SSIM metrics dangerous is that they can be cheated *accidentally*. For example, while my `y_channel_usage=0.5` modification in IRN-Y is rather obviously targeting Y-channel metrics, one could make far more obscure decisions in the model for which the bias is harder to detect.

One notable instance is the choice of loss function for IRN's guidance and reconstruction terms (table 2.1). In Xiao et al's paper [11], they argue that L2 is more suitable for a guidance loss term due to it resulting in a higher PSNR-Y score - but I find it unclear whether this gain is actually indicative of an improvement in representational ability, or simply L2 causing a greater bias towards the Y-channel compared to L1. In fact, I believe that the L2 loss function is likely introducing a bias towards PSNR itself, due to PSNR being based on mean squared error (section 2.4).

Further IRN-Y Results

It is informative to examine IRN-Y from a qualitative point of view, which I present in figures 4.6 and 4.7.

IRN. RGB: 25.54 / 0.7806; Y: 30.17 / 0.8542.

IRN-Y. RGB: 24.91 / 0.7693; Y: 31.31 / 0.8891.

Ground Truth.

Figure 4.6: This figure examines the ability of my **IRN-Y** model at reconstructing an example image in the DIV2K dataset. This was one of the most clear examples of IRN-Y's superior ability at preserving (luminance) detail through the downscaling-upscaling process over the original IRN. For the original IRN model, this image was one of the most challenging examples in the 100-image DIV2K dataset.

4.3 IRN for Other Tasks

In this section, I describe adapting IRN to solve tasks other than *image rescaling*, including compression (4.3.1) and super-resolution (4.3.2).

IRN. RGB: 25.78 / 0.8744; Y: 28.05 / 0.9008.

IRN-Y. RGB: 25.35 / 0.8663; Y: 28.35 / 0.9101.

Ground Truth.

Figure 4.7: This figure points out the subtle flaws in IRN-Y's upscaling performance using an image from Set14. While the dragon is reconstructed in higher detail in IRN-Y, we can observe some colour distortion in the rings above it, and this is reflected in lower PSNR/SSIM scores in the RGB channel.

4.3.1 IRN for Compression

It occurred to me during the development of this project that, while the general real-world use-case for an *image rescaling* model is to offer high-quality upscaling for images downscaled when moving between devices, in reality such a downscaling operation is often accompanied by *lossy compression* - e.g. through the JPEG format. In other words, downscaling tends to be used for the purpose of reducing file size, and for that purpose it makes sense for a user to accompany the downscaling with lossy compression.

However, current image-rescaling models are designed with the idea in mind that their downscaled images will later be upscaled *unmodified*, aside from initial quantization. Hence I propose a new task, *compression-resistant image rescaling*, which acknowledges the reality that most downscaled images will be corrupted through lossy compression before an attempt at upscaling is made.

As far as I am aware, JPEG-reconstruction and image-rescaling have never been modelled *jointly* - but I believe that modelling the two tasks as one is more true to their real-world use cases, and allows one to achieve better results by designing a *downscaling* method that is conducive to recovering high-quality images from the compression.

I summarise the compression-resistant image rescaling problem (CRIR) as follows: **how can we downscale an image to low-resolution, then have it undergo JPEG-compression, and finally upscale to its original high-resolution, with a minimal loss in image quality at all stages?**

Assessment

I assessed both the standard IRN model and HCFlow at *compression-resistant image rescaling* by adding a JPEG-compression step into my testing code, configured at `quality=90`. I furthermore went on to train my own variant of IRN expressly for the task, **IRN-C** (section 3.4.2), gathering CRIR results for it too.

I present qualitative results for the CRIR task in figure 4.8. Notably, we can see on the right of the figure that IRN-C's CRIR training has allowed it to upscale the example image

			DIV2K Post-JPEG Upscaling		DIV2K Downscaling	
Downscaling & Upscaling	Scale	Param	Y-Channel	RGB	Y-Channel	RGB
HCFlow	4×	4.4M	27.19 / 0.7386	25.16 / 0.6929	*36.19/0.9785*	*29.50 / 0.8819*
IRN (Xiao et al.)	4×	4.4M	*28.40 / 0.7788*	*26.33 / 0.7348*	**44.38 / 0.9933**	**37.40 / 0.9676**
IRN-C (Mine)	4×	4.4M	**29.36 / 0.8004**	**27.46 / 0.7661**	29.24 / 0.8745	25.32 / 0.8202

Table 4.6: Compression-resistant image rescaling results measured in PSNR / SSIM. **Bold** denotes the best result in a column and *italics* the second-best. My IRN-C model outperforms both IRN and image-rescaling state-of-the-art HCFlow at reconstruction from compressed images, but inevitably performs worse at downscaling due to the compression-resistant artefacts it introduces.

without propagating JPEG artefacts, achieving a higher PSNR/SSIM than IRN. Examining the downscaled images to the left, it appears as though IRN-C has developed a strategy for reducing the impact of JPEG-compression by embedding the image with alternating green-red lines, in a pattern reminiscent of *chromatic aberration*. I have two theories as to how this behaviour may have appeared:

1. IRN-C is simply mimicking the artefacts produced in JPEG-compression, which it has learned to do due to the invertibility of the model. By learning the inverse process (compressed → high-res) during training, the model will naturally be inclined to produce (high-res → compressed) when inverted.

2. IRN-C is producing artefacts which minimize the difference between its downscaled images and their JPEG-compressed counterpart, in order to more reliably encode information that assists the upscaling process. This is the intended, ideal behaviour for the model.

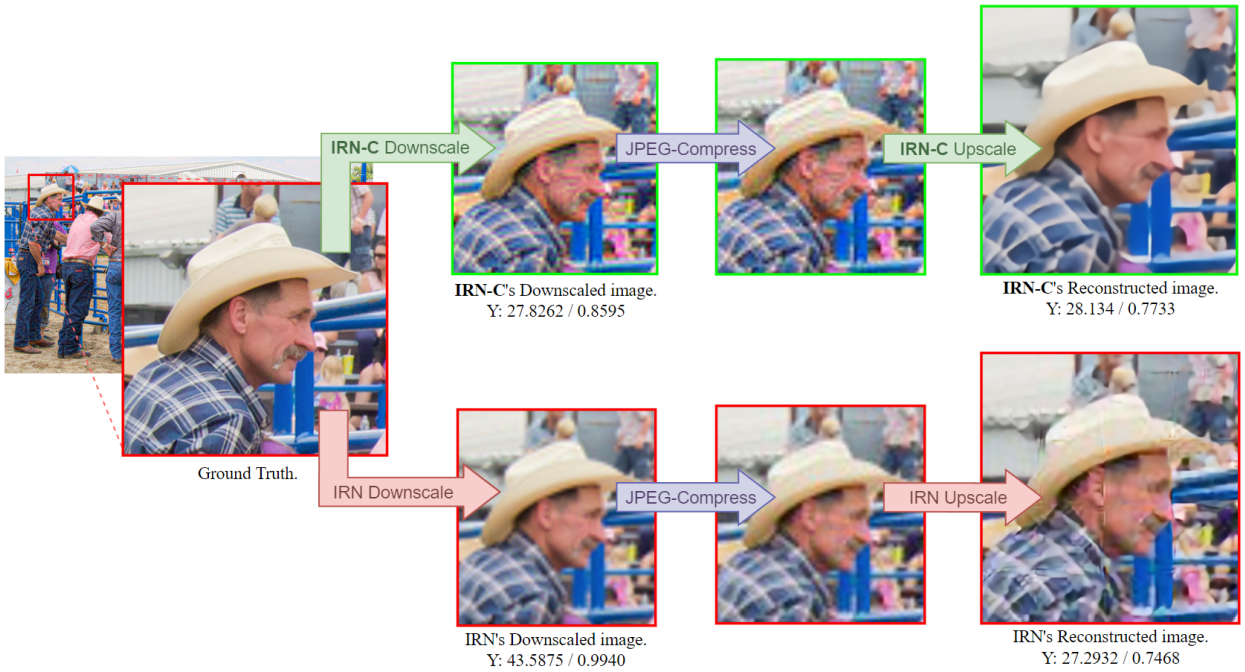


Figure 4.8: This figure provides an example of compression-resistant image rescaling on an image from the DIV2K dataset. On the top, we demonstrate the process of assessing **IRN-C** at CRIR by downscaling an image, compressing it with `quality=90` JPEG-compression, and finally upscaling the result. This is reproduced for the standard IRN model on the bottom.

4.3.2 IRN for Super-Resolution

I present preliminary results assessing the base IRN model at *super-resolution* (i.e. upscaling images which were downscaled by bicubic rescaling, rather than IRN-downscaling) and demonstrating the improvement of my **IRN-SR** model as described in section A.2. In this case, the IRN-SR model consists of Xiao et al.'s IRN combined with an "assistant" model trained on preparing low-resolution images for upscaling. I trained this assistant for 2.2k epochs on DIV2K, taking the HR-matching loss approach described in equation (A.3).

Given more time, I would evaluate the model much more thoroughly, computing additional scores against other datasets and contrasting with other models - as well as attempting longer training (i.e. >2.2k epochs). My results thus far are presented in table 4.7.

Upscaling	Scale	Param	PSNR/SSIM (RGB)
Bicubic	4×	-	26.70 / 0.77
EDSR	4×	43.1M	28.98 / 0.83
IRN (Xiao et al.)	4×	4.4M	25.88 / 0.75
IRN-SR (Mine)	4×	5M	28.67 / 0.81

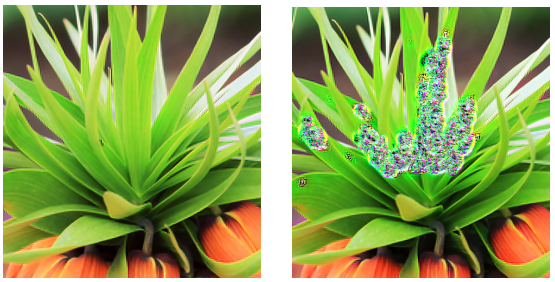


Table 4.7: Preliminary super-resolution (SR) results on the DIV2K validation set. To the right: a comparison of four techniques, demonstrating that my IRN-SR modification is able to assist IRN at the task to the extent of exceeding bicubic rescaling and reaching fairly close to established SR technique EDSR [7]. To the right: qualititative results from upscaling with IRN-SR and IRN respectively. One can see that my modification was primarily able to assist IRN by eliminating intense artefacts created in upscaling.

4.4 Summary

In this chapter, I presented results demonstrating that my IRN implementation matches the original *image rescaling* scores produced by Xiao et al. [11] (section 4.2.1). I furthermore highlighted several issues with the currently-standard test metrics used for comparing image rescaling models (PSNR-Y / SSIM-Y), by presenting models that were able to excel under these metrics while lacking in other areas (section 4.2.3). Additionally, I described a new task, *compression-resistant image rescaling*, and provided initial results in section 4.3.1.

Chapter 5

Conclusions

In conclusion, the project was a success: I exceeded all core success criteria and implemented several extensions, which I believe contribute to the field.

5.1 Project Achievements

I met my core success criteria by implementing a PyTorch-based Invertible Rescaling Network for *image rescaling* that successfully reproduces the performance described by its creators (Xiao et al. 2020). I utilised the FrEIA framework in my implementation after assessing its suitability by training an invertible architecture known as Glow, and making a small contribution to FrEIA's open-source codebase[1].

I met extension criteria by deriving a modification to IRN based on the concept of Lipschitz bounds (section 3.4.1), and found that it exceeds the *image rescaling* performance of Xiao et al.'s original model (see IRN-2T, table 4.3). I furthermore highlighted flaws with the field's current standard metrics for *image rescaling* by demonstrating how simple modifications to existing models can challenge state-of-the art performance (see IRN-Y, section 4.2.3).

I then went on to propose a new task - *compression-resistant image rescaling* - which I argue has higher real-world utility than *image rescaling* (see section 4.3.1), and I produced baseline results for the task (see IRN-C, table 4.6). Lastly, I briefly assessed the possibility of using existing *image rescaling* models for *super-resolution* by designing an invertible assistant network that enhances IRN's performance at the task (details in section 4.3.2).

5.2 Lessons Learnt

Having neither prior experience with PyTorch nor familiarity with invertible neural networks, I found this project to be an immensely educational experience. It provided me with a glimpse into the forefront of research - with regard to both image rescaling and deep invertible architectures more generally - and I learned how practical implementations like FrEIA can accelerate deep learning research.

[1]Specifically, I contributed my implementation of IRN's coupling layers (2.3.2). Note that as of May 2022, the contribution is in the form of a Git Pull Request that has not yet been integrated.

Retrospection on Development

In reflection, there were a few aspects of this project's development which I would approach differently were I able to restart. The PyTorch-based Glow implementation (section 3.2.1) - a relatively minor part of the project - took much longer than I anticipated. This was largely due to the complexity of combining different components, often having disparate descriptions online, without introducing unintended effects (such as one component producing *Jacobian determinants* where another produces *log Jacobian determinants*). In retrospect, I would have instead budgeted the time to further exploring state-of-the-art image rescaling model HCFlow, which my supervisor and I were not aware of at the project's inception.

The training of IRN models saw several delays as I realised flaws with my implementation that were not apparent until training was nearly complete. For example, realising that *loss* should be computed on *unquantized* model output, and seeing much higher instability otherwise. In part, these delays were a consequence of Xiao et al. not making certain aspects of IRN explicit in their original paper [11]. It may have been worth contacting the authors myself to acquire further metrics about the training of their model, such that I could spot issues in my own training more quickly. Instead, my workarounds to the issue of long training times included testing IRN on much smaller data initially, and testing my two preparatory Glow implementations against each other. These did help to an extent, allowing me to identify more general mistakes in testing/training code without computationally-expensive training, but nonetheless, I found the iterative process of training and testing IRN rather slow.

5.3 Future Development

This dissertation described issues with current image rescaling metrics. Given more time, I would aim to identify a single test metric or set of test metrics which are most suitable for assessing different image rescaling approaches. This may involve running a perceptual study to identify the correlation between metrics and subjective opinions.

I mentioned in section 3.3.2 that hyperparameter tuning on IRN is challenging due to the size of the model being computationally expensive to train on the hardware that was available to me[2]. One exciting recent development in this area is Microsoft's μ*Transfer* technique (March 2022) [34], which has been shown to allow rapid hyperparameter tuning on large models such as GPT3. It would be valuable to assess the suitability of this technique - which revolves around tuning far smaller but representative versions of a model - for invertible architectures like IRN.

I described in this project a Lipschitz-bounds-inspired modification to IRN's coupling layers which improves stability. If I had more time, I would aim to more rigorously evaluate the impact of this modification by applying the coupling layers to problems other than image rescaling, recreating the experiments from Behrmann et al.'s paper [32]. This would partly involve explicitly recording how the Jacobian of the coupling layer responds to different inputs compared to its unmodified counterpart.

Lastly, there still remain unexplored tasks adjacent to image rescaling, which I would like to pursue using an invertible model such as IRN. For example, one could perform rescaling of video, which comes with the additional constraint of needing to maintain temporal stability.

[2] 4.4 million parameters, taking 3-5 days to train to 10k epochs on an NVIDIA A100 GPU.

Bibliography

[1] Elisabeth J.M. Baltussen et al. "Using Diffuse Reflectance Spectroscopy to Distinguish Tumor Tissue From Fibrosis in Rectal Cancer Patients as a Guide to Surgery". English. In: *Lasers in surgery and medicine* 52.7 (Sept. 2020). Wiley deal, pp. 604–611. ISSN: 0196-8092. DOI: 10.1002/lsm.23196.

[2] S. Kabanikhin et al. "Definitions and examples of inverse and ill-posed problems". In: *Journal of Inverse and Ill-posed Problems* 16 (Jan. 2008), pp. 317–357.

[3] A.M. Bruckstein, M. Elad, and R. Kimmel. "Down-scaling for better transform compression". In: *IEEE Transactions on Image Processing* 12.9 (2003), pp. 1132–1144. DOI: 10.1109/TIP.2003.816023.

[4] Sina Farsiu et al. "Advances and challenges in Super-Resolution". In: *International Journal of Imaging Systems and Technology* 14 (Jan. 2004). DOI: 10.1002/ima.20007.

[5] Zhihao Wang, Jian Chen, and Steven C. H. Hoi. "Deep Learning for Image Super-Resolution: A Survey". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 43.10 (2021), pp. 3365–3387. DOI: 10.1109/TPAMI.2020.2982166. URL: https://doi.org/10.1109/TPAMI.2020.2982166.

[6] Christian Ledig et al. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 105–114. DOI: 10.1109/CVPR.2017.19. URL: https://doi.org/10.1109/CVPR.2017.19.

[7] Bee Lim et al. "Enhanced Deep Residual Networks for Single Image Super-Resolution". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 1132–1140. DOI: 10.1109/CVPRW.2017.151. URL: https://doi.org/10.1109/CVPRW.2017.151.

[8] A. Cengiz Öztireli and Markus H. Gross. "Perceptually based downscaling of images". In: *ACM Trans. Graph.* 34.4 (2015), 77:1–77:10. DOI: 10.1145/2766891. URL: https://doi.org/10.1145/2766891.

[9] Heewon Kim et al. "Task-Aware Image Downscaling". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part IV*. Ed. by Vittorio Ferrari et al. Vol. 11208. Lecture Notes in Computer Science. Springer, 2018, pp. 419–434. DOI: 10.1007/978-3-030-01225-0_25. URL: https://doi.org/10.1007/978-3-030-01225-0%5C_25.

[10] Lynton Ardizzone et al. "Analyzing Inverse Problems with Invertible Neural Networks". In: *CoRR* abs/1808.04730 (2018). arXiv: 1808.04730. URL: http://arxiv.org/abs/1808.04730.

[11] Mingqing Xiao et al. "Invertible Image Rescaling". In: *CoRR* abs/2005.05650 (2020). arXiv: 2005.05650. URL: https://arxiv.org/abs/2005.05650.

[12] Shang Li et al. “Approaching the Limit of Image Rescaling via Flow Guidance”. In: *CoRR* abs/2111.05133 (2021). arXiv: 2111.05133. URL: https://arxiv.org/abs/2111.05133.

[13] Jingyun Liang et al. “Hierarchical Conditional Flow: A Unified Framework for Image Super-Resolution and Image Rescaling”. In: *CoRR* abs/2108.05301 (2021). arXiv: 2108.05301. URL: https://arxiv.org/abs/2108.05301.

[14] Jinbo Xing, Wenbo Hu, and Tien-Tsin Wong. “Scale-arbitrary Invertible Image Downscaling”. In: *CoRR* abs/2201.12576 (2022). arXiv: 2201.12576. URL: https://arxiv.org/abs/2201.12576.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[16] Roger Koenker and Kevin F. Hallock. “Quantile Regression”. In: *Journal of Economic Perspectives* 15.4 (Dec. 2001), pp. 143–156. DOI: 10.1257/jep.15.4.143. URL: https://www.aeaweb.org/articles?id=10.1257/jep.15.4.143.

[17] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[18] Chigozie Nwankpa et al. “Activation Functions: Comparison of trends in Practice and Research for Deep Learning”. In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378.

[19] Laurent Dinh, David Krueger, and Yoshua Bengio. “NICE: Non-linear Independent Components Estimation”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1410.8516.

[20] Zhou Wang et al. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE Trans. Image Process.* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861. URL: https://doi.org/10.1109/TIP.2003.819861.

[21] Jonas Haldemann et al. “Exoplanet Characterization using Conditional Invertible Neural Networks”. In: *CoRR* abs/2202.00027 (2022). arXiv: 2202.00027. URL: https://arxiv.org/abs/2202.00027.

[22] Victor Fung et al. “Inverse design of two-dimensional materials with invertible neural networks”. In: *CoRR* abs/2106.03013 (2021). arXiv: 2106.03013. URL: https://arxiv.org/abs/2106.03013.

[23] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. “Density estimation using Real NVP”. In: *CoRR* abs/1605.08803 (2016). arXiv: 1605.08803. URL: http://arxiv.org/abs/1605.08803.

[24] Diederik P. Kingma and Prafulla Dhariwal. “Glow: Generative Flow with Invertible 1x1 Convolutions”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 10236–10245. URL: https://proceedings.neurips.cc/paper/2018/hash/d139db6a236200b21cc7f752979132d0-Abstract.html.

[25] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[26] Tim Salimans and Diederik P. Kingma. "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, p. 901. URL: https://proceedings.neurips.cc/paper/2016/hash/ed265bc903a5a097f61d3ec064d96d2e-Abstract.html.

[27] Gao Huang et al. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243. URL: https://doi.org/10.1109/CVPR.2017.243.

[28] Xintao Wang et al. "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks". In: *Computer Vision - ECCV 2018 Workshops - Munich, Germany, September 8-14, 2018, Proceedings, Part V*. Ed. by Laura Leal-Taixé and Stefan Roth. Vol. 11133. Lecture Notes in Computer Science. Springer, 2018, pp. 63–79. DOI: 10.1007/978-3-030-11021-5_5. URL: https://doi.org/10.1007/978-3-030-11021-5%5C_5.

[29] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*. IEEE Computer Society, 2012, pp. 3642–3649. DOI: 10.1109/CVPR.2012.6248110. URL: https://doi.org/10.1109/CVPR.2012.6248110.

[30] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*. Ed. by Yee Whye Teh and D. Mike Titterington. Vol. 9. JMLR Proceedings. JMLR.org, 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[31] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation". In: *CoRR* abs/1308.3432 (2013). arXiv: 1308.3432. URL: http://arxiv.org/abs/1308.3432.

[32] Jens Behrmann et al. "Understanding and mitigating exploding inverses in invertible neural networks". In: *CoRR* abs/2006.09347 (2020). arXiv: 2006.09347. URL: https://arxiv.org/abs/2006.09347.

[33] Martin Heusel et al. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 6626–6637. URL: https://proceedings.neurips.cc/paper/2017/hash/8a1d694707eb0fefe65871369074926d-Abstract.html.

[34] Greg Yang et al. "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer". In: *NeurIPS 2021*. Mar. 2022. URL: https://www.microsoft.com/en-us/research/publication/tuning-large-neural-networks-via-zero-shot-hyperparameter-transfer/.

[35] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. by Zoubin Ghahramani et al. 2014, pp. 2672–2680. URL: https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

[36] Tianci Liu and Jeffrey Regier. "Flows Succeed Where GANs Fail: Lessons from Low-Dimensional Data". In: *CoRR* abs/2006.10175 (2020). arXiv: 2006.10175. URL: https://arxiv.org/abs/2006.10175.

[37] Takeru Miyato et al. "Spectral Normalization for Generative Adversarial Networks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=B1QRgziT-.

Appendix A

Extension Details

A.1 Extension Hyperparameters

Hyper-Parameter	IRN Default	Description
Of the model:		
`actnorm`	False	Applies ActNorm (3.1) after each coupling.
`clamp_tightness`	1.0	Tightens the clamp function to discourage small values.
`compression_mode`	False	JPEG-compress training images.
`sr_mode`	False	Super-resolution mode: upscale LR rather than LR' during training .
`zerosample`	False	Upscale using LD=0 rather than sampling from a normal distribution during training.
Of training:		
`clip_value_max_g`	∞	`max_g` parameter used in `clip_value` (3.16).
`loss_flip_period`	∞	Epochs between flipping loss coefficients.
`loss_flip_scale`	1	Factor by which we flip loss coefficients .
`quantize_guide_loss`	False	Quantize LR' before computing guidance loss.
`quantize_recon_loss`	False	Quantize HR' before computing recon loss.
`y_channel_usage`	0	Coefficient of y-channel included in loss.

Table A.1: Extension hyperparameters introduced in this project.

A.2 Super-Resolution

Though IRN's invertibility makes it particularly well-suited for the task of *image rescaling*, I also explored its potential at straightforward *super-resolution*. I explored this task in three different ways:

SR Approach #1: Assisted IRN

I designed an assistant invertible network to convert between LR and the LR' of an existing IRN model, thus facilitating super-resolution by running the assistant on IRN's output. Let us call this assistant *AN*, and the operation of IRN and AN together *IRN-SR*.

This assistant network AN is based on the **Glow** architecture described in section 3.2.1, but makes use of **DenseNet** subnetworks as described in section 3.3, which I chose due to their aptitude at super-resolution tasks [28].

One can see several sensible ways to train this assistant network. We could optimize it to match the LR' produced by IRN,

$$loss = L1(LR' - AN(LR)), \tag{A.1}$$

in the inverse direction to reconstruct ground-truth LR images,

$$loss = L1(LR - AN^{-1}(LR')), \tag{A.2}$$

or simply train it as a single super-resolution unit with IRN and optimize to match HR images for a given LR,

$$loss = L1(HR - IRN(AN^{-1}(LR), LD_{random})), \tag{A.3}$$

or a combination of all three. The first two approaches have the advantage of not requiring us to run IRN during the training of AN; we can simply run IRN beforehand and create a dataset of LR' images with with to train. This frees up memory as well as reducing processing time.

However, the third approach (A.3) should ultimately land on a better solution, as it can exploit features of IRN to produce high-quality images even in cases where it struggles to recreate LR'. I evaluate this SR approach in 4.3.2.

SR Approach #2: Re-Trained IRN

I trained a standard IRN model to perform super-resolution by altering its training loop (3.3.2) to discard LR', instead making the model learn to upscale from LR alone. In other words, we can simply replace the $x' \leftarrow \texttt{irn}_\theta^{-1}((y'_Q, z'))$ step with $x' \leftarrow \texttt{irn}_\theta^{-1}((y, z'))$.

Intuitively, the guidance loss (2.1) function of IRN is less important when training for super-resolution, because we are now discarding the network's LR' low-resolution output, and only care about its capacity for upscaling. However, including this term can still be beneficial in encouraging a strong solution - due to the invertibility of the network, naturally once we have $IRN(HR) = LR, z$, we will have $IRN^{-1}(LR, z) = HR$ - meaning the distribution-matching term optimizing z remains all that needs to be trained.

SR Approach #3: Non-Re-Trained IRN

I tested a standard IRN model, originally trained on image-rescaling, at super-resolution without further training.

Appendix B

Testing Details

B.1 Unit Testing; Confirmation of Old Results

I used *PyTest* to perform unit tests across my codebase, particularly paying attention to whether my data transformations (such as Y-channel conversion) and test metrics returned the same values as Xiao et al.'s implementation.

It is important to ensure that any results I obtain with my implementation are comparable against existing image rescaling results, and as such I computed my own rescaling PSNR/SSIM scores for existing *bicubic*, *IRN*, and *HcFlow* rescaling models across popular image datasets - for the sake of verification against numbers reported in other papers.

These tests all returned expected results within an acceptable floating point error (∼1e-7 to account for e.g. nondeterminism in CUDA convolutions), with the exception of **bicubic rescaling on DIV2K**, which I found to produce lower SSIM and higher PSNR than reported in Xiao et al.'s paper [11]. This difference is recorded in table B.1.

Originator	PSNR (Y) (2×)	SSIM (Y) (2×)	PSNR (Y) (4×)	SSIM (Y) (4×)
Myself	32.45	0.9042	28.11	0.7746
Xiao et al.	<u>32.95</u>	<u>35.07</u>	<u>0.9061</u>	<u>0.9269</u>

Table B.1: Metrics obtained by testing Bicubic rescaling on DIV2K. <u>Underlined</u> are the numbers reported by the authors.

I have several reasons to believe that the bicubic DIV2K results reported by Xiao et al. (table B.1) may be incorrect:

- All of my other test metrics match those reported by Xiao et al. This includes B100, DIV2K, Set5, Set14, and Urban100 test datasets against bicubic rescaling, Xiao et al.'s IRN rescaling model, and HCFlow rescaling.
- Each component of my test function has been verified, achieving the same results as the bicubic-rescaling, Y-channel conversion, and PSNR/SSIM functions of Xiao et al.'s paper.
- It is surprising that the DIV2K SSIM score of bicubic rescaling should be greater than that of sophisticated super-resolution models such as ESRGAN, particularly given that its PSNR and SSIM scores on every other dataset are significantly lower (e.g. Bicubic achieving 0.6577 on Urban100 4× vs ESRGAN's 0.8152).

- If anything, we would expect it to be the PSNR metric that Bicubic rescaling achieves better results in than SSIM, due to PSNR being more noise-sensitive and bicubic rescaling naturally blurring out noise.

Digging deeper into the issue, I discovered that the authors' reported Bicubic DIV2K results were in fact quoted from a previous paper, which itself quotes a previous paper, etc. The earliest source I could find for these results were Lim et al.'s EDSR paper [7], which unfortunately only states the following:

"Note that DIV2K validation results are acquired from published demo codes."

I reached out to the authors for clarification on how exactly the DIV2K Bicubic rescaling results were acquired, but unfortunately did not receive a response in time.

Appendix C

Invertible Architecture Examples

C.1 Alternatives to Probabilistic Disentanglement

Although they aren't used in this project, it may be worth mentioning two workarounds I am aware of that would enable INN MLE in inverse problems:

- Use a conditional INN. This is an INN that takes two inputs: one to be fed through the network as normal, and one to be supplied directly into each coupling layer as a *condition* (e.g. concatenated onto the input to each inner network m in our additive coupling example (2.17)).

  One could perform image super-resolution by training a generative model using MLE and supplying it with LR as a condition during training. However, learning image rescaling would be a little more complex - we would need to train a second network N' to downscale HR to LR, as our first network would only be capable of taking LR as a condition.

  Let N_C denote network N supplied with condition C. For the conditional image rescaling described above, where $LR' \approx LR$ and $HR' \approx HR$, we have:

$$N'_{HR}(LD_{random}) = LR'$$
$$N_{LR'}(LD_{random}) = HR'$$

- Use what I term as *assisted probabilistic disentanglement*. Instead of making N constrain LD to a normal distribution, add an assistant INN N' that converts between LD and $\mathcal{N}(0,1)$, and train this assistant using MLE. This is roughly the approach taken by Liang et al.'s HCFlow (2021) [13], which I denote as:

$$N(HR) = (LR', LD)$$
$$N^{-1}(LR', N'(LD_{random})) = HR'$$

C.2 Example: Change of Variables Formula

Consider a typical convolutional neural network that generates pictures by converting samples of a gaussian distribution into images. This is known as a **generative** model. More formally:

$$N : X \longrightarrow Y \text{ where } X \sim \mathcal{N}(0,1) \text{ and } Y \sim IMG.$$

Our optimisation goal is to bring the IMG distribution modelled by N closer to the distribution $\hat{T}$ that generated some target dataset T. A neat way to achieve this would be to minimise the cross-entropy $CE(IMG, \hat{T})$, equivalent to minimising $p_{IMG}(T)$ (2.9).

However, for typical deep generative models, there is no obvious way of extracting this probability: the density of the IMG distribution modelled by N is unknown, and because convolutional layers are not bijective, we cannot apply the change of variables formula to extract it. One workaround would be to estimate this likelihood using a critic network, forming a *Generative Adversarial Network (GAN)* [35]. However, producing accurate likelihood estimates has been found challenging for GANs [36], and can become computationally expensive for higher dimensional data.

Normalizing flows offer a much cleaner solution. Suppose N is a normalizing flow model. Because normalizing flows are invertible and designed to have a tractably computable determinant, we can perform $N^{-1}(x)$, and then apply the **change of variables formula** (2.16) to compute $p(T(x)|N)$:

$$p(T(x)|N) = p_{IMG}(T(x)) = p_{\mathcal{N}(0,1)}(N^{-1}(T(x))) * |\frac{dN^{-1}}{dT(x)}|$$

thus allowing normalizing flow models to be trained through maximum likelihood estimation.

Appendix D

Derivations

D.1 Clamp Proposal #1 Derivative

We have:

$$\begin{aligned}\frac{d}{dx}j(x) &= \frac{d}{dx}(\text{sign}(x)\cdot(|x|+\frac{1}{2})+\frac{1}{2}) \\ &= \left(\frac{d}{dx}\text{sign}(x)\right)\cdot(|x|+\frac{1}{2})+\text{sign}(x)\cdot(\frac{d}{dx}(|x|+\frac{1}{2})) \\ &= 0\cdot(|x|+\frac{1}{2})+\text{sign}(x)\cdot\text{sign}(x) \\ &= \text{sign}(x)\cdot\text{sign}(x), \\ &= 1 \text{ for } x\neq 0,\end{aligned}$$

hence:

$$\frac{d}{dx}(g_{\text{mod}}(x)) = g'(j(x))\cdot\frac{d}{dx}j(x) = g'(j(x)) \text{ for } x\neq 0.$$

D.2 Lipschitz Bounds

This section walks through the processing of deriving Lipschitz-inspired modifications for coupling layers, beginning with a theoretic explanation of Lipschitz Bounds.

Lipschitz Bounds

A function $f : X \longmapsto Y$ is **Lipschitz Continuous** iff there exists some constant ("**Lipschitz constant**") L such that:

$$|f(x_1)-f(x_2)| \leq L|x_1-x_2| \quad \text{for all } x_1, x_2 \in X, \tag{D.1}$$

and this function can be called **bi-Lipschitz continuous** if its inverse f^{-1} is also Lipschitz continuous.

Suppose the Lipschitz constant L is given by some function Lip, such that if $\text{Lip}(f) = L$ is constant with respect to f's input x, then f is Lipschitz continuous. If $\text{Lip}(f)$ is **not** constant with respect to choice of input x, then we can say it is only locally Lipschitz continuous.

Another formulation of this is to state that f is **locally Lipschitz continuous** iff it is only Lipschitz continuous for limited regions of its domain X (there exists some $[a,b]^d \in X$):

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2| \quad \text{for all } x_1, x_2 \in [a,b]^d. \tag{D.2}$$

In many cases, we will be unable to derive an exact Lipschitz constant (for full neural networks, this is NP-hard [37]), but rather a **Lipschitz bound** $\text{Lip}(f) \leq B$ which may be either global (independent on x) or local (dependent on x).

Spectral Normalisation

Controlling the Lipschitz bounds of a neural layer is useful because it allows us to **stabilise the training of that layer**, providing a constraint on its gradient - which, for example, can prevent gradient explosion. This relationship between Lipschitz bounds and stability exists because $\text{Lip}(f)$ can be formulated in terms of f's Jacobian, like so:

$$\text{Lip}(f) = \sup_x(\|J(f(x))\|_2), \tag{D.3}$$

where $\|A\|_2$ denotes the spectral norm of A,

$$\|A\|_2 = \sigma_{\max}(A) = \text{the largest singular value of } A, \tag{D.4}$$

and $\sup_x(F(x))$ denotes the smallest upper bound of F,

$$\forall x \in X : \left(\sup_x(F(x)) \geq F(x)\right) \wedge \left(y \geq F(x) \iff y \geq \sup_x(F(x))\right). \tag{D.5}$$

Layers possessing a global Lipschitz bound - such as linear (fully-connected) and convolutional layers, which in fact have Lipschitz constant $Lip(f) = \|W\|_2$ for weights tensor W - can have the bound directly controlled by *spectral normalisation* [37].

Spectral normalisation involves reducing the spectral norm of some parameters of the layer to lower their Lipschitz bound. In the case of linear and convolutional layers, we can constrain the global bounds by setting the spectral norm of the weight tensor W to 1, performing the following during training:

$$W \leftarrow W/\|W\|_2 \tag{D.6}$$

Lipschitz Bounds of Coupling Layers

Behrmann et al. [32] derived global bi-Lipschitz bounds for additive coupling layers, and local bi-Lipschitz bounds for affine coupling. This allows us to directly control the bound on additive coupling's Lipschitz constant via spectral normalisation, but only influence that of affine coupling more indirectly (unless we have a constraint on the layer's input).

For an **additive coupling layer** (2.17) f with subnet m, we have global bounds:

$$\text{Lip}(f) \leq 1 + \text{Lip}(m) \tag{D.7}$$
$$\text{Lip}(f^{-1}) \leq 1 + \text{Lip}(m). \tag{D.8}$$

→ Therefore, if we assume m is a Lipschitz continuous function such as a neural network, we can reduce the additive coupling layer's bound simply by spectral normalisation in m.

For an **affine coupling layer** (3.4) f with clamp g and subnets s, t, we have local bounds for $x \in [a, b]^d$:

$$\text{Lip}(f) \leq \max(1, c_g) + M \tag{D.9}$$

where $M = \max(|a|, |b|) \cdot c_{g'} \cdot \text{Lip}(s) + Lip(t)$, and c_g is an upper bound on g for $x \in [a, b]^d$. For the inverse we have bounds over $y \in [a^*, b^*]^d$:

$$\text{Lip}(f^{-1}) \leq \max(1, c_{\frac{1}{g}}) + M^*, \tag{D.10}$$

where $M^* = \max(|a^*|, |b^*|) \cdot c_{\frac{1}{g}'} \cdot \text{Lip}(s) \cdot c_t + c_{\frac{1}{g}} \cdot Lip(t)$.

→ One can see that the inverse Lipschitz bound (D.10) is heavily influenced by small values of g. Hence, Behrmann et all. suggest that an appropriate way to encourage affine coupling's $\text{Lip}(f^{-1})$ downwards would be to **modify the clamp function** g such that it avoids smaller values, preventing explosion of $c_{\frac{1}{g}}$.

Earlier I mentioned that, for the affine coupling layer, Berhmann et al. suggest lowering Lipschitz bounds by modifying the clamp function g to avoid smaller values. They described how one would make this modification for the clamp used in **Glow**, which is a sigmoid function having range $[0, 1]$, by performing an affine transformation that fixes its range to $[0.5, 1]$ instead. I illustrate this modification in table D.1.

Original Glow Clamp		Modified Glow Clamp
$g(x) = \frac{1}{1+e^{-(x+2)}}$	$\implies$	$g_{\text{mod}}(x) = g(x) \cdot \frac{1}{2} + \frac{1}{2}$

Table D.1: The Lipschitz-inspired modification to Glow's affine coupling clamp, suggested by Behrmann et al.

Now let us consider making a comparable change to IRN's *Enhanced Affine Coupling Layers*, which use a transformed sigmoid function having range $[-1, 1]$. The most directly comparable modification we can make, without sacrificing the clamp's ablility to produce negative values, would be to also cut out the $[0, 0.5]$ range of the function, shown in table D.2.

Original IRN Clamp		Modified IRN Clamp (Proposal #1)
$g(x) = \frac{1}{1+e^{-x}} \cdot 2 - 1.$	$\implies$	$g_{\text{mod}}(x) = g(\text{sign}(x) \cdot (\lvert x \rvert + \frac{1}{2}) + \frac{1}{2})$

Table D.2: One possible Lipschitz-inspired modification to Glow's affine coupling clamp, cutting out the $[0, 0.5]$ range of g's output.

However, I found in practice that using a clamp function that has such a large *jump discontinuity* (jumping from $g_{\text{mod}} = 0$ to $g_{\text{mod}} = 0.5$ around $x = 0$) has the potential to introduce greater instability in the network over time. This makes sense as the jump cannot be communicated in g_{mod}'s derivative - because the derivative of $j(x) = (\text{sign}(x) \cdot (|x| + \frac{1}{2}) + \frac{1}{2})$ is 1 everywhere except for at the discontinuity, for which it is undefined. Instead, I propse a simpler modification in section 3.4.1.

D.3 L2 and MLE Loss

Optimizing for sum-of-squares loss is equivalent to performing maximum likelihood estimation when assuming that model error is gaussian-distributed.

Assume gaussian-distributed error such that $N(x) + \epsilon = T(x)$ where ϵ is taken from normal distribution $\mathcal{N}(0, \sigma^2)^n$. This gives us an expression for the likelihood $p(T(x)|N)$ in terms of error ϵ:

$$\begin{aligned} & p(T(x)|N) \\ & = p(\mathcal{N}(0, \sigma^2)^n = \epsilon) \\ & = p(\mathcal{N}(0, \sigma^2)^n = T(x) - N(x)) \end{aligned}$$

Note that by definition of $\mathcal{N}(0, \sigma^2)^n$, the elements of ϵ are independently distributed, allowing us to examine them in isolation:

$$= \prod_i p(\mathcal{N}(0, \sigma^2) = T(x)_i - N(x)_i)$$

Applying the gaussian PDF formula to one of these elements, we get:

$$\begin{aligned} p\left(\mathcal{N}(0, \sigma^2) = T(x)_i - N(x)_i\right) &= \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{(T(x)_i - N(x)_i) - \mu}{\sigma}\right)^2\right) \\ &= \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{T(x)_i - N(x)_i}{\sigma}\right)^2\right) \end{aligned}$$

Now inserting into the MLE loss formula (2.4):

$$\begin{aligned} -\ln p(T(x)|N) &= -\sum_{i=0}^{n-1} \ln p\left(\mathcal{N}(0, \sigma^2) = T(x)_i - N(x)_i\right) \\ &= -\sum_{i=0}^{n-1} \ln \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{T_i - N(x)_i}{\sigma}\right)^2\right) \\ &= -\sum_{i=0}^{n-1} \left(\left(\ln \frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2}\left(\frac{T_i - N(x)_i}{\sigma}\right)^2\right) \\ &= -\sum_{i=0}^{n-1} \left(\ln \frac{1}{\sigma\sqrt{2\pi}}\right) + \frac{1}{2\sigma^2} \sum_{i=0}^{n-1} \left((T_i - N(x)_i)^2\right) \end{aligned}$$

As we are deriving a formula for minimizing N's loss, we can take out any factors that do not depend on x and N. This includes taking out σ, leaving us with simply the sum-of-squares loss (2.7):

$$\sum_{i=0}^{n-1} (T_i - N(x)_i)^2$$

Appendix E

Project proposal

Introduction and description

In this project, I use the idea of invertible neural networks to reduce the degradation in image quality incurred by image downscaling and upscaling, re-implementing a recent paper[1].

Background

Invertible neural networks

Invertible neural networks can be thought of as functions that have an inverse.

Suppose a neural network N can be used on some input data x to compute $N(x)=y$. If that network is invertible, then it is possible to also perform the computation $N^{-1}(y)=x$. I call these the *forward* and *backward* directions of the network respectively.

This invertibility means that we can train an INN to solve $N(x)=y$, then invert the network to get $N^{-1}(y)=x$. This training method has been found to be particularly effective for a class of problems known as *inverse problems*, as explained further below.

Naturally, to be invertible, N must be bijective.

Inverse problems

In science, an *inverse problem* is any problem characterised by needing to deduce system parameters from observations - for example, deducing the 3D shape of an object from its shadow.

These problems are sometimes considered ill-posed, as there is often insufficient information in the observation to fully recover the parameters. In other words, there is usually a one-to-many relation between observation and parameters, as some information gets "lost" in producing the observation. (In literature, this unobserved information is often called "latent" data.)

The image rescaling problem

One such inverse problem is the problem of *image rescaling*. We can consider a high-resolution image to be the "parameter" of this system, and a low-resolution image the "observation", which has inherently lost some information from its high-res counterpart. Solving the inverse problem in this case is known as *upscaling*, and solving the forwards problem is *downscaling*.

In most cases, images on the internet will have been at some point downscaled before reaching your monitor. For example, the image may have been taken on a digital camera which downscales its raw 8000x8000px images to 4000x4000px, sent over an email service which downscales to 3000x3000px, and uploaded on an image hosting site that imposes a 2000x2000px limit. This is typically performed using bicubic downscaling, and recovering a higher-resolution image from one of these results would be difficult.

I summarise the **image rescaling problem** as this: *after a high-resolution image has been downscaled, how can we upscale it back to high-resolution with minimal loss in image quality?*

---

[1] Xiao, Mingqing, et al. "Invertible image rescaling." European Conference on Computer Vision. Springer, Cham, 2020

Invertible Image Rescaling

Introduction

Historically, upscaling and downscaling have been considered separate problems. More recently however, efforts have been made to design a downscaling method that is conducive to high-quality upscaling. One such effort is the *Invertible Image Rescaling* paper that I intend to re-implement.

The paper proposes that we perform rescaling using an invertible neural network. Ideally, we would train a network to downscale high-resolution (HR) to low-resolution (LR) in one direction:

$$N(HR) = LR$$

And upscale LR to HR in the other:

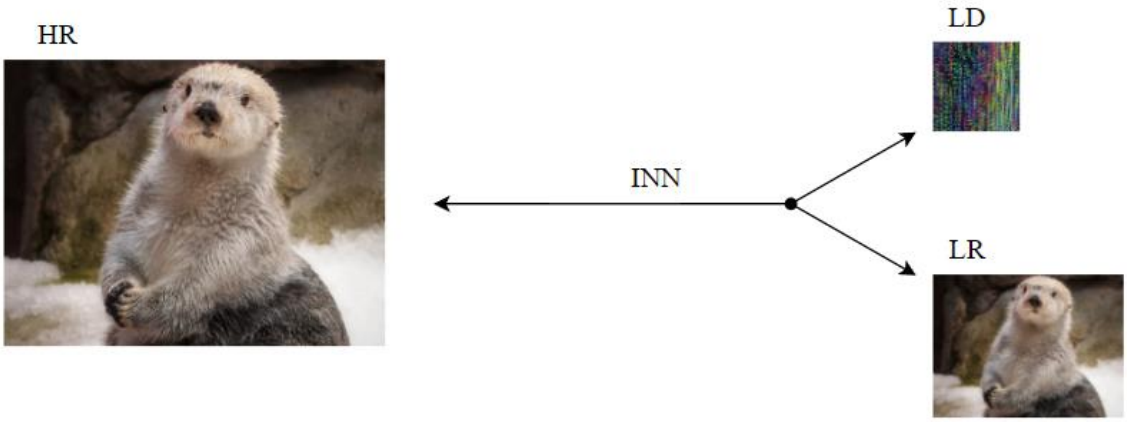
$$N^{-1}(LR) = HR$$

Latent variables

However, this upscaling is a one-to-many operation - and in order to model the problem with an INN, we need to make the operation bijective instead. The paper's solution to this is to include some latent data LD in the output, which encodes the information that was lost from HR when downscaling to LR ("disentangling" HR into LR and LD).

$$N(HR) = (LR, LD)$$

$$N^{-1}(LR, LD) = HR$$



Probabilistic disentanglement

However, this disentanglement has changed the nature of the image rescaling problem, as we would now be required to predict a suitable LD before being able to upscale LR. The paper proposes that this requirement would be alleviated if, when training the network to upscale LR to HR, we take LD to be a random sample from a simple distribution (specifically, an isotropic multivariate Gaussian).

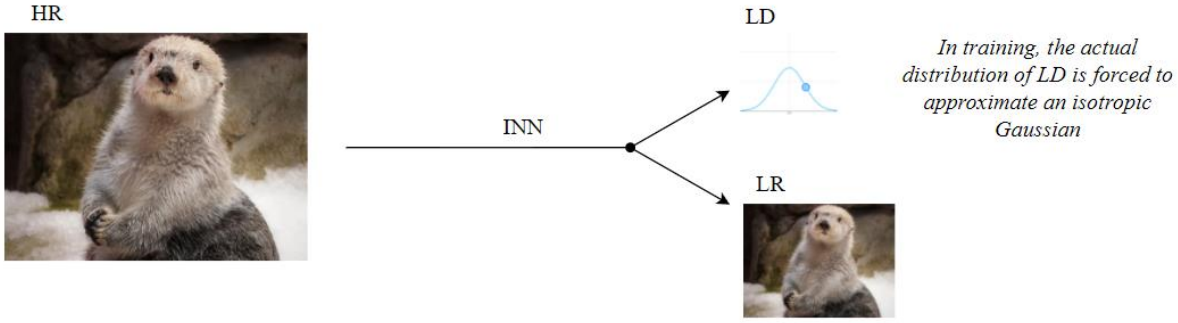
This way, the model is forced to learn how to upscale without relying on LD (as it doesn't convey useful information in training), and the downscaling operation is forced to become more predictable - reducing the amount of case-specific information encoded in LD - to assist this. In other words, the model must learn to disentangle in such a way that the distribution of LD resembles a Gaussian distribution and is somewhat case-agnostic to LR.

With LD no longer encoding particularly useful information, the problem effectively resembles our $N^{-1}(LR)=HR$ problem in which we are required only to provide LR, as it is acceptable to randomly guess LD.

So, suppose that in place of the latent data LD we use a sample from the latent *variable* $LD_{random} \sim \mathcal{N}(\mu,\Sigma)$. We can now perform our rescaling like so:

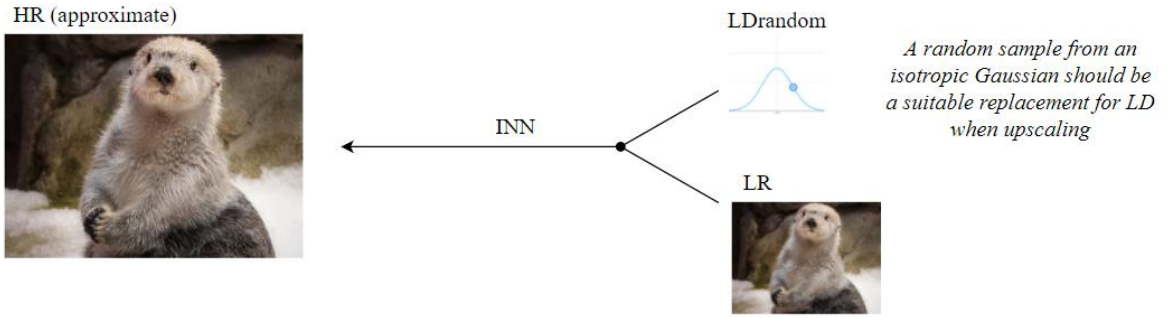
Downscaling

$$N(HR) = (LR, LD)$$



Upscaling

$$N^{-1}(LR, LD_{random}) \approx HR$$



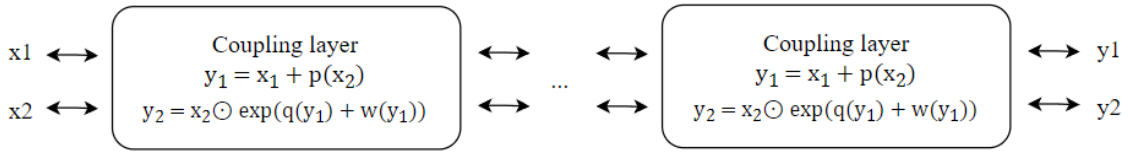
Network Architecture

To finish my explanation of the paper that I plan to reimplement, I will give an overview of how the invertible neural network in the paper actually works internally.

The architecture the paper uses for the INN consists of a series of invertible, differentiable transformations. The idea is that with enough chained transformations, the network becomes sufficiently complex that it can be trained to approximate complicated functions.

The main invertible transformation used in the network is a *coupling layer*. Without going into technical detail (though I leave the formula for the transformation in the diagram below), this is a transformation that first requires us to split our input into two parts (which may be high-dimensional tensors - call these x_1 and x_2), and in the end produces two outputs (call these y_1 and y_2).

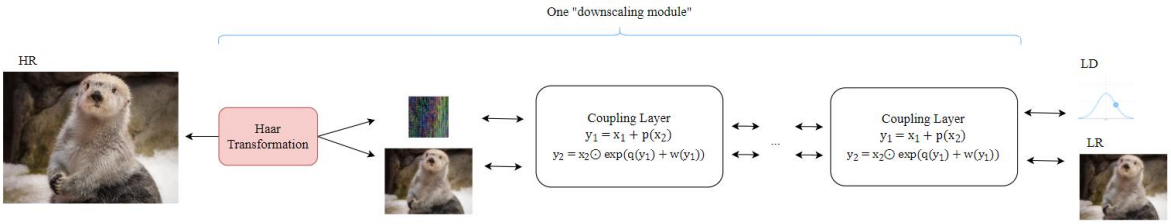
These layers are highly expressive and easily invertible, as they can themselves contain arbitrary functions such as convolutional neural networks.



One other invertible transformation the paper uses is a *Haar transformation*, which primitively downscales an image (by 2x) and disentangles from it the high-frequency information that was lost in downscaling. Essentially, this is a simplistic version of the image rescaling operation that we intend the network to perform.

This Haar transformation is used to divide the input image into two parts, which can be given as x_1 and x_2 to the coupling layers. It also assists the INN in producing aesthetically-pleasing downscaled images.

One Haar transformation is followed by about 8 coupling layers, which together form one "downscaling module".



My (re-)implementation

An implementation of the original paper by its authors exists in Pytorch. I plan to reimplement this using an open source framework known as FrEIA, which is intended specifically for constructing and training INNs.

As part a more minor of the work, I will begin by comparing FrEIA to another popular ML framework (Pytorch) and assess its merits.

Work to be undertaken

Demonstrate the utility of FrEIA over a more general framework, such as Pytorch, by writing and comparing solutions to some smaller deep generative problem, such as unconditional MNIST generation.

- Work complete when I have a working implementation of this smaller problem in FrEIA and one other framework, and I have extracted no-reference metrics evaluating the implementations.

- This is a minor unit of work.

Re-implement the *Invertible Image Rescaling* paper, primarily relying on FrEIA in my implementation.

- Work complete when I have a working implementation and have tested it on at least one benchmark dataset, achieving results within 3dB PSNR and 0.1 SSIM compared to the results found in the paper on the 2x resolution image downscaling/upscaling task.

- This is the most significant piece of work.

Challenges and risks

Risk: the paper I plan to reimplement makes use of some unusual mechanisms (such as a custom type of coupling layer and a custom loss function that scores the INN on its forwards and backwards performance). These may prove tricky to implement in FrEIA, or FrEIA may prove to have bugs which make my implementation faulty.

Plan: if FrEIA seems too limiting (though currently we don't believe it is), we have identified other frameworks that I could use for my re-implementation - such as SurVAE. Additionally, if we come into bugs or limitations with FrEIA that don't seem too difficult to fix, I could extend the open source code and fix them myself.

Risk: I may be unable to find a suitable GPU server to train my model within a reasonable training time. This would prevent me from comparing my results directly to that of the paper. The paper I plan to reimplement spent 6 days training their model using a Tesla P100 - I should find a similarly powerful server.

Plan: I can gather some results evaluating my model by training it on much smaller images (64x64 greyscale) on my home GPU (an RTX 3080), and potentially I could adapt the author's implementation to train it in the same way and compare results directly.

Starting point

I will be using the open source FrEIA framework, but am not planning to extend any existing code.

I will also use open source image datasets for training and testing purposes at some point, including DIV2K.

I may run the author's open source implementation to compare its results with my own model if I am unable to acquire the resources necessary to run my own model on the exact dataset used in the paper.

Success criteria

Core criteria

I consider this project to be successful when I have:

1. Compared my choice of ML framework to one other framework by creating implementations of a small deep generative model.
2. Written a re-implementation of the model described in *Invertible Image Rescaling* and tested it on at least one benchmark dataset, achieving results within 3dB PSNR and 0.1 SSIM compared to the results found in the paper on the 2x resolution image downscaling/upscaling task.

Possible extensions

1. Evaluating the rescaling model under more extreme cases than were demonstrated in the paper - e.g. 16x downscaling and upscaling.
2. Altering an existing open-source application to make use of the model for rescaling or compression.
3. Altering the INN to optimise it for different problems, such as compression without rescaling, or for image colourisation.
4. Extend the open source code for FrEIA.
5. Compare FrEIA with another INN framework.

Work packages and milestones

Package no.	**Date range**	**Work to be completed**
1 *(weeks 1-2)*	18 Oct - 1 Nov	- Make notes on the FrEIA framework - Make notes on Pytorch - Initial experiments with FrEIA - Plan a simple deep learning problem to try implementing in FrEIA and PyTorch - Read around details from the *Invertible Image Rescaling* paper
2 *(weeks 3-4)*	1 Nov - 15 Nov	- Implement an INN for a simple deep learning problem in FrEIA - Progress on implementing the same model in Pytorch
3 *(weeks 5-6)*	15 Nov - 3 Dec	- Finish implementation of an INN for a simple deep learning problem in Pytorch - Begin writing about my work thus far
Milestone 1	**By Dec 3**	**End of Michaelmas term** **- Achieved success criteria 1 (created implementations of simple ML model)** **- Began writing justification of my choice of ML framework**
4 *(weeks 7-8)*	3 Dec - 13 Dec	- Begin writing structure for code

		- Identity or create a low-resolution image dataset I can use for "rapid" testing of the model - Ensure I have a working GPU server that I could use to train my model if one has not been found already
5 *(weeks 9-10)*	13 Dec - 27 Dec	- Holiday & revision - May begin building out main INN model
6 *(weeks 11-12)*	27 Dec - 10 Jan	- Build out main INN model
7 *(weeks 13-14)*	10 Jan - 24 Jan	- Continue iterating on main INN model - Build out code for assisting with the model training process (i.e. training optimizations described in the paper, such as an adaptive learning rate) - Test my implementation on increasingly high-resolution datasets
8 *(weeks 15-16)*	24 Jan - 4 Feb	- Evaluate the model by running full-length training on the DIV2K dataset using a GPU server, then running benchmark tests - Final changes and polish to the model code, planning write-up
Milestone 2	**By Feb 4 (*noon)***	**Progress Report Deadline** **- Success criterion passed by now** **- Progress report submitted**
9 *(weeks 17-18)*	4 Feb - 21 Feb	Finish intro and preparation chapters
10 *(weeks 19-20)*	21 Feb - 7 Mar	Work on implementation chapter
11 *(weeks 21-22)*	7 Mar - 21 Mar	Finish implementation, evaluation, conclusions chapters; work on possible extension / collect further evaluation data
12 *(weeks 23-24)*	21 Mar - 1 Apr	Work on possible extension/collect further evaluation data
Milestone 3	**By Apr 1**	**Pass full draft dissertation to DofS and supervisor**
13 *(weeks 25-26)*	1 Apr - 13 May	Respond to DofS and supervisor feedback
Milestone 4	**By May 13 *(noon)***	**Final dissertation and source code deadline**

Resource declaration

I will need GPU server to use for training to evaluate my model against the paper - ideally, something of similar power to a Tesla P100, which the paper's authors used to train their model (for 6 days). Most likely I will use a departmental GPU server or CSD3.

I may use my own laptop for training the network in small cases (using a 3080 mobile GPU). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.