Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Oskar Jørgensen

Improving Generative Image Models Using Hybrid Flow Matching and Autoregressive Models

Master's thesis in Computer Science
Supervisor: Helge Langseth
June 2025

NTNU
Norwegian University of Science and Technology

Oskar Jørgensen

Improving Generative Image Models Using Hybrid Flow Matching and Autoregressive Models

Master's thesis in Computer Science
Supervisor: Helge Langseth
June 2025

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

PREFACE

This Master's thesis was conducted while studying at the Norwegian University of Science and Technology (NTNU) in Trondheim. I would like to thank my supervisor, Helge Langseth, for valuable discussions and insights into the field of generative AI. I would also like to extend my gratitude to IDI for providing access to GPUs used to conduct the experiments in this thesis. Additionally, I would like thank my fellow students for informal discussions during much needed breaks, and the occasional round of Smash. Lastly, I want to thank my roommates at K33 for helping shape the best years of my life.

Parts of this thesis build on work done in the courses TDT70 and TDT4501. In particular, the theory related to flow matching is a revised and improved version of an initial draft submitted in TDT4501. However, all sections are rewritten to better fit this thesis and extended to be more thorough. If not stated otherwise, the figures and images in this thesis are original. AI has been used to help edit the text, particularly to detect grammatical error or typos. It has not been used to generate new text, but rather to restructure or review the flow of text I had already written. As for the code, AI has been used to explain existing code. I have also used intelligent autocomplete to speed up production, but this is limitied to small code snippets and trivial code. As for some of the illustrative plots, longer segments of the code are AI generated. However, all use of AI tools, be it for coding or writing, have been manually reviewed by me.

ABSTRACT

Motivated by the rising energy consumption of generative image models, we investigate the trade-off between efficiency and performance when generating images. Leveraging advances in autoregressive (AR) models for image generation, and flow matching (FM) models, we propose a novel hybrid architecture. We use a pre-trained AR model to generate low resolution images, which are then up-scaled and enhanced using a lightweight FM model trained on ImageNet. We compare our hybrid architecture against an AR baseline, and demonstrate an improved FID score, with the best model having a score of 1.87. The hybrid architecture has fewer total parameters and only a minimal increase in inference time when compared to the baseline. Empirically, we demonstrate that when pre-trained on large datasets for super-resolution, the vector field learned by the FM model approximates a linear vector field, where all paths are straight lines in the high-dimensional space. This insight allows us to use simple ODE solvers, enabling faster generation while maintaining a high perceptual quality. While the pre-training demonstrates promising results, finetuning the models on our custom-built dataset presented significant challenges. We highlight challenges related to dataset construction and the pairing of similar images to explain why this approach resulted in artifacts. Finally, we discuss the importance of sustainable development in AI, and the issues generative models raise related to copyright laws and intellectual property of the training data.

SAMMENDRAG

Motivert av det økenende energiforbruket til generative bildemodeller, undersøker vi balansen mellom effektivitet og bildekvalitet hos bildegeneringsmodeller. Ved å utnytte fremskritt innenfor autoregressive (AR) modeller og flow matching (FM) modeller, foreslår vi en ny, hybrid arkitektur. Vi bruker en forhåndstrent AR-modell for å generere bilder med lav oppløsning. Disse blir skalert opp og forbedret med en liten FM-modell trent på ImageNet. Vi sammenligner vår hybride arkitektur med en AR-referansemodell og viser en forbedring i FID-score, hvor den beste modellen får en score på 1,87. Hybridarkitekturen har færre parametere og er kun marginalt tregere under inferens sammenlignet med referansemodellen. Vi viser empirisk at når FM-modellen vår er trent på store datasett for oppskalering (super-resolution), er vektorfeltet som blir lært tilnærmet lik et lineært vektorfelt. Dette lar oss bruke en enkel ODE-løser, som gjør at vi kan gjøre inferens raskere samtidig som kvaliteten på de genererte bildene forblir høy. Til tross for at forhåndstrening ga gode resultater, bød fine-tuning på flere utfordringer. Vi fremhever utfordringer knyttet til hvordan datasettet ble laget (spesielt parringen av input og output), og hvordan dette resulterte i lav kvalitet på de genererte bildene. Avslutningsvis diskuterer vi viktigheten av bærekraftig utvikling innenfor KI og problemstillinger rundt generative modeller og opphavsrettslover knyttet til treningsdataen.

Contents

List of Figures

List of Tables

ABBREVIATIONS

List of all abbreviations by topic, in alphabetic order:

Math & Artificial Intelligence

AI	Artificial Intelligence
CDF	Cumulative Density Function
CFM	Conditional Flow Matching
CNF	Continuous Normalizing Flow
FM	Flow Matching
GenAI	Generative Artifical Intelligence
NF	Normalizing Flow
ODE	Ordinary Differential Equation
OT	Optimal Transport
PDF	Probability Density Function
SDE	Stochastic Differential Equation

Other

IDI	Institute of Data and Informatics at NTNU
IEA	International Energy Agency
NTNU	Norwegian University of Science and Technology
SDG	Sustainable Development Goals (proposed by the UN)
SOTA	State-of-the-Art
UN	United Nations

NOTATION

Notation is heavily inspired by [1]. However, if there are common conventions in related literature, the notation might deviate. This is based on what we deemed the least confusing option, and evaluated on a case to case basis. In the cases where a variable can be both univariate (scalar) or multivariate (vector or matrix), the scalar notation is used.

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor A
a_i	Element i of the random vector $\mathbf{a}$
$\dim(\boldsymbol{a})$	The dimension of the vector space of $\boldsymbol{a}$. i.e. $\boldsymbol{a} \in \mathbb{R}^n$, then $\dim(\boldsymbol{a}) = n$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Calculus

$\frac{\mathrm{d}x}{\mathrm{d}y}$	Derivative of y with respect to x
$\frac{\partial x}{\partial y}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\boldsymbol{J}_f$ or $\boldsymbol{J}(f)$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$

Probability and Information Theory

$\mathrm{a} \perp \mathrm{b}$	The random variables a and b are independent
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E}f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$D_{\mathrm{KL}}(P \Vert Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\ln x$	Natural logarithm of x
$\log x$	Base 10 logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$

Sometimes we use a function f whose argument is a scalar but apply it to a vector or matrix: $f(\boldsymbol{x})$ or $f(\boldsymbol{X})$. This denotes the application of f to the array element-wise. For example, if $\boldsymbol{C} = \sigma(\boldsymbol{X})$, then $C_{i,j,k} = \sigma(X_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\text{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$

CHAPTER

ONE

INTRODUCTION

This chapter provides an introduction to the thesis. We start with a motivation for conducting the thesis. Following that, we have the goals and research questions this thesis aims to answer. Then we describe the main contributions, before summarizing the thesis structure.

1.1 Motivation

Generative artificial intelligence (generative AI or GenAI) encompass all AI models that produce new data. This could be text, images, audio or something else entirely. In the pursuit of higher quality models, the model size, dataset size and computational resources have all grown over the years. Optimizing for quality often comes at the expense of less efficient models. This thesis aims to explore the efficiency-quality balance, with a focus on models that generate images. Deeply understanding this could provide insights into how one can better optimize for one of the two (that is, efficiency or performance), without compromising the other. While it is obvious why better model performance is beneficial, we will briefly touch upon why increasing the efficiency of models should be a goal of all AI researchers.

As the models grow, so does their electricity demands. While it is hard to quantify exactly how much electricity is used worldwide to power generative AI applications, the International Energy Agency (IEA) claim that roughly 2 % of the worlds electricity consumption in 2022 came from data centers,

cryptocurrency and AI. They also expected this sector to double in size by 2026. While a lot of this growth can be attributed to traditional data centers (used for non-AI purposes), they also estimate that electricity consumed by AI will increase at least tenfold from 2023 to 2026 [2]. This corresponds to an increase of 70 terawatt hours (TWh) per year, or roughly half of Norway annual electricity consumption [3].

This exponential increase in electricity consumption coincides with the popularization of generative models. IEA reports that a query to OpenAI's text generation model, ChatGPT, uses close to ten times the energy of using a search engine like Google. Depending on the model used, and the size of the image, generating one image is roughly equivalent to charging your smartphone halfway [4]. To ensure sustainable development of new generative AI models, it is important to research how one might make them more efficient. From the perspective of a company or an institution developing these models, the motivation would be lower costs, fewer resources needed and faster development time. And from a global perspective, it would result in a lower demand for electricity, which in turn would lead to less CO2-emissions.

1.2 Goals and Research Questions

Goal *Explore the balance between efficiency and quality in image generation models by combining different architectures*

The ultimate goal of this thesis is to improve on image generation models. For these models, there is typically a trade-off between efficiency and quality. By investigating different architectures, we can further explore this trade-off, and see if it is possible to improve the efficiency while maintaining a high quality, or improve the quality while maintaining the same efficiency. We can break this goal down into several smaller research questions (RQs):

RQ 1 *What are the State-of-the-Art architectures for image generation?*

RQ 2 *How can we measure the efficiency of models?*

RQ 3 *How can we measure the quality of generated images?*

RQ 4 *Is it possible to use traditional interpolation methods to up-scale generated images efficiently?*

RQ 5 *Is it possible to combine different architectures to improve either efficiency or quality, without compromising the other?*

Research question 1, 2, 3 are theoretical, and answers can be found in related research. Research question 4 and 5, on the other hand, requires experiments to empirically measure the efficiency/quality of different architectures, and combinations.

1.3 Contributions

The contributions of this thesis are as follows:

- **Extensive Background**: An extensive background chapter, including an introduction to key mathematical concepts, several important architectures in deep learning, and the theory behind flow matching and autoregressive image generation. In particular, the sections about flow matching and normalizing flows are rigorous, detailed and would serve as a great introduction to someone new to the field.
- **Codebase**: a code repository for training, sampling from and evaluating different hybrid architectures. All resource intensive code is written to be able to run on GPUs, and with scripts that allow for easily running them on GPU clusters, with multi-GPU support.
- **Trained models**: we also make available the pre-trained models used to achieve the results in this thesis. In particular, these models excel at performing super-resolution on images.

1.4 Thesis Structure

The thesis is split into 6 chapters.

This chapter, **Chapter 1**, provides a short introduction to the thesis. We motivate why we believe this research to be important, state concrete research goals and questions we aim to answer, and write the contributions of this thesis.

Chapter 2 introduces all theory necessary to understand the thesis. This is aimed at people who have completed an undergraduate degree in computer science, with a focus on artificial intelligence and deep learning. Section 2.1-2.3 cover necessary math knowledge, mainly vector calculus, probability and vector fields. Section 2.4 introduces theory related to deep learning and neural networks. Section 2.5-2.7 is an in-depth introduction to flow matching,

and how it has evolved starting from normalizing flows until where the field is today. Section 2.8 introduces the State-of-the-Art autoregressive models for image generation, while Section 2.9 is a comparison between flow matching and autoregressive models. Section 2.10 and 2.11 discuss how to measure the quality and efficiency of generative image models.

In **Chapter 3**, we outline our proposed method for answering our research questions. It includes a brief overview of the most relevant work in this field. As most of the research is very new and the field is moving fast, we opt to introduce a broad range of related work, rather than going in-depth. This chapter also discusses the datasets we will use throughout the thesis.

Chapter 4 details our experiments and results. We begin by outlining a plan for what to do, and describing the setup for our experiments. For each experiment, there follows a discussion of the results observed.

The next chapter, **Chapter 5**, contains longer, more holistic discussions regarding the observations we find most relevant in the experiments. Specifically, we discuss why finetuning the models prove to be a challenge, and how this can be improved. We discuss the open-ended question of how to measure the quality of images in an efficient and accurate way, without relying on human evaluators. Finally, we discuss ethical considerations when making generative image models related to copyright and intellectual property of images. We also discuss our research in the context of the Sustainable Development Goals proposed by the UN.

Finally, **Chapter 6** provides a summary of our findings in this thesis, before proposing what we deem to be promising directions of future research. We highlight the need to improve the approach for finetuning our models, why we believe it is interesting to benchmark them against super-resolution models and why it might be interesting to train the models in latent space, instead of pixel-space.

We also include 4 appendices. Appendix A contains a link to our codebase and a brief description. Appendix B is a proof of the continuity equation which we believe might help provide intuition for why it is a key part in flow matching. Appendix C contains generated images from all our models, while Appendix D details the hyperparameters for our models so that the results can be reproduced.

CHAPTER

TWO

THEORY

This chapter will introduce relevant theory on flow models and flow matching, as well as autoregressive models. Necessary preliminary knowledge, both from probability and vector calculus, will also be introduced. Work that this thesis builds upon is cited throughout. The math related to vector calculus in section 2.1 can be found in most introductory books to multivariate calculus, such as [5]. Similarly, the math related to probability in section 2.2, can be found in [6] or other undergraduate-level books about probability.

2.1 Vector Calculus

To understand the math presented, it is important to be familiar with vector calculus. This section briefly explains partial derivatives. Additionally, it introduces determinants, Jacobians and traces of matrices. There is also a section about the inverse function theorem, which will be used later.

2.1.1 Partial Derivative

The **partial derivative** of a multivariate function is the derivative with respect to one variable, while the others are held constant. Given a function $f(x, y, \dots)$ we denote the partial derivative of f with respect to x as

$$\frac{\partial f}{\partial x} \text{ or } \frac{\partial}{\partial x} f$$

For a function $f(x, y, z)$ evaluated at the point (u, v, w), we write

$$\left.\frac{\partial f(x, y, z)}{\partial x}\right|_{(x,y,z)=(u,v,w)}$$

2.1.2 Jacobian

Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, where the first order derivatives for f exist, we denote the **Jacobian J** or $\mathbf{J}(f)$ to specify that it is the Jacobian of f. It is given by an $m \times n$ matrix where the $(i, j)^{th}$ entry is $\frac{\partial f_i}{\partial x_j}$. Explicitly, we have:

$$\mathbf{J}(f) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

2.1.3 Gradient

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, where the first order derivatives for f exist, we denote the **gradient** ∇f. It is given by a column vector of length n

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

The gradient evaluated at a point is a vector that points in the direction of fastest increase at that point.

2.1.4 Determinant

The **determinant** exists only for square matrices and is a single scalar. For a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ it is denoted $\det \boldsymbol{A}$ or $\det(\boldsymbol{A})$. If the matrix entries are

real numbers, we can think of $\boldsymbol{A}$ as a linear transformation. Furthermore, the determinant of $\boldsymbol{A}$, $\det(\boldsymbol{A})$, can be thought of as the measure of volume change when applying the linear transformation $\boldsymbol{A}$. We will show this for $\mathbb{R}^2$, but the same logic can be applied to show this for any $\mathbb{R}^n$.

We consider the general matrix

$$\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

which is a linear transformation mapping $\mathbb{R}^2$ to $\mathbb{R}^2$. To see why the determinant is a measure of the volume change, we consider the two basis vectors of $\mathbb{R}^2$,

$$\boldsymbol{e}^{(1)} = \begin{bmatrix} 1 & 0 \end{bmatrix} \text{ and } \boldsymbol{e}^{(2)} = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

We can then calculate the new basis vectors $\boldsymbol{u}^{(1)}, \boldsymbol{u}^{(2)}$ for the new coordinate system:

$$\begin{aligned} \boldsymbol{u}^{(1)} = \boldsymbol{e}^{(1)}\boldsymbol{A} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a & b \end{bmatrix} \\ \boldsymbol{u}^{(2)} = \boldsymbol{e}^{(2)}\boldsymbol{A} &= \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} c & d \end{bmatrix} \end{aligned}$$

The parallelogram that $\boldsymbol{u}^{(1)}$ and $\boldsymbol{u}^{(2)}$ span is shown in Figure 2.1.1, along with its signed area, $ad - bc$. This is the same as $\det(\boldsymbol{A})$. By the definition of the initial basis vectors, this area was 1. We see that by multiplying any area by $\det(\boldsymbol{A})$, we get the volume of the new area after the linear transformation.

2.1.5 The Inverse Function Theorem

Given a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}^n$, the determinant of its Jacobian, $\det \mathbf{J}(f)$ can tell us information about f. If the Jacobian determinant at a point is non-zero, f is invertible near the point. This is known as the **inverse function theorem**. Furthermore, the absolute value of the determinant tells us the factor by which f expands or shrinks the volume near the point.

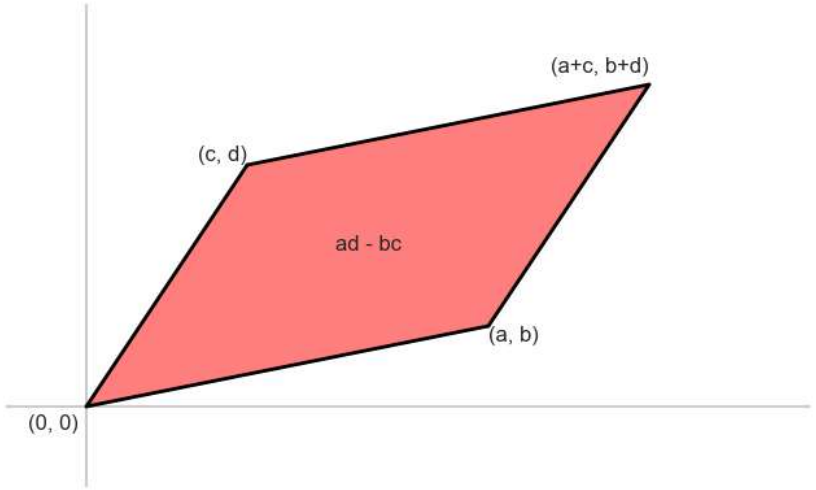


Figure 2.1.1: The area spanned by $\boldsymbol{u}^{(1)}$, $\boldsymbol{u}^{(2)}$ and $\boldsymbol{u}^{(1)}\boldsymbol{u}^{(2)}$ forms a parallelogram with area equal to $\det(\boldsymbol{A})$.

2.1.6 Trace

Given a square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, the **trace** is the sum of all elements along its diagonal. More formally, we have:

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_{i=1}^{n} A_{i,i} = A_{1,1} + A_{2,2} + \cdots + A_{n,n}$$

2.2 Probability

This section briefly introduces probability density functions and cumulative distributive functions. It derives the change of variables formula for scalar-to-scalar functions, and shows the formula for vector-to-vector functions.

2.2.1 Inverse Cumulative Density Function

Recall the definition of the **cumulative distribution function** (CDF):

$$F_X(x) = P(X \leq x) \tag{2.1}$$

Additionally, since $P(X \leq x) + P(x \geq X) = 1$, we can see that

$$P(X \geq x) = 1 - F_X(x) \tag{2.2}$$

2.2.2 Change of Variables

Let X be a random variable and its **probability density function** (PDF) given as $f_X(x)$. We wish to calculate the PDF of some variable $Y = g(X)$, $f_{g(X)} = f_Y$. This is called **change of variable**.

2.2.2.1 Scalar-to-scalar

Let $g : \mathbb{R} \to \mathbb{R}$ be a strictly monotonic (increasing or decreasing) continuously differentiable function with a continuously differentiable inverse $g^{-1} : \mathbb{R} \to \mathbb{R}$. Let us first look at the case where g is strictly increasing. Then we have that the PDF of Y is

$$\begin{aligned} P(Y \leq y) &= P(g(X) \leq y) \\ &= P(X \leq g^{-1}(y)) \end{aligned}$$

Using Equation 2.1, we see that

$$F_Y(y) = F_X(g^{-1}(y)) = F_X(x) \tag{2.3}$$

We can now differentiate the CDF w.r.t. to y. For the case where the function was strictly increasing, we use Equation 2.3 and the chain rule to show that

$$\begin{aligned} f_Y(y) = \frac{\mathrm{d}F_Y(y)}{\mathrm{d}y} &= \frac{\mathrm{d}F_X(x)}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}y} \\ &= f_X(x)\frac{\mathrm{d}x}{\mathrm{d}y} \end{aligned}$$

Similarly, if g is strictly decreasing, we have that

$$\begin{aligned} P(Y \leq y) &= P(g(x) \leq y) \\ &= P(X \geq g^{-1}(y)) \end{aligned}$$

Note that we switch the direction of the inequality sign. This is because if g is strictly decreasing, then so is g^{-1}. We can then use Equation 2.2 to show that

$$F_Y(y) = 1 - F_X(g^{-1}(y)) = 1 - F_X(x) \tag{2.4}$$

For the case where the function was strictly decreasing, we use Equation 2.4 and the chain rule

$$\begin{aligned} f_Y(y) = \frac{\mathrm{d}F_Y(y)}{\mathrm{d}y} &= \frac{\mathrm{d}(1 - F_X(x))}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}y} \\ &= -f_X(x)\frac{\mathrm{d}x}{\mathrm{d}y} \end{aligned}$$

Finally, since $\frac{\mathrm{d}x}{\mathrm{d}y} \geq 0$ if g is strictly increasing and $\frac{\mathrm{d}x}{\mathrm{d}y} \leq 0$ if g is strictly decreasing, we can combine the two equations.

$$\begin{aligned} f_Y(y) &= f_X(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right| \\ &= f_X(g^{-1}(y))\left|\frac{\mathrm{d}g^{-1}(y)}{\mathrm{d}y}\right| \end{aligned} \tag{2.5}$$

2.2.2.2 Vector-to-vector

Assume an n-dimensional random variable $\boldsymbol{x}$ with a joint density f. Let $G : \mathbb{R}^n \to \mathbb{R}^n$, where G is bijective and differentiable. If $\boldsymbol{y} = G(\boldsymbol{x})$, then $\boldsymbol{y}$ has a density:

$$f_{\boldsymbol{Y}}(\boldsymbol{y}) = f_{\boldsymbol{X}}\left(G^{-1}(\boldsymbol{y})\right)\left|\det\left[\frac{\partial}{\partial \boldsymbol{y}}G^{-1}(\boldsymbol{y})\right]\right| \tag{2.6}$$

$$= f_{\boldsymbol{X}}\left(G^{-1}(\boldsymbol{y})\right)\left|\det\left[\frac{\partial}{\partial \boldsymbol{y}}G(\boldsymbol{y})\right]\right|^{-1} \tag{2.7}$$

where the last equality is a result of the inverse function theorem. This is a generalization of Equation 2.5. For spaces in higher dimensions, the volume correction term corresponds to the absolute value determinant of the Jacobian.

2.3 Vector Fields, Divergence and the Continuity Equation

This section briefly introduces vector fields, and two key concepts related to them. The first is the divergence operator, which can be applied to a vector field to see how the quantity at any point changes (either increasing, decreasing or remaining the same), and by how much. Finally, it introduces the continuity equation.

2.3.1 Vector Fields

A **vector field** is a function in an n-dimensional space that assigns an n-dimensional vector to each point in that space. Formally, for $\mathbb{U} \subseteq \mathbb{R}^n$ we define the vector field $\vec{F} : \mathbb{U} \to \mathbb{R}^n$. We can also view this as a vector composed of scalar-valued functions. For some input $\boldsymbol{x} \in \mathbb{R}^n$ we have:

$$\vec{F}(\boldsymbol{x}) = \begin{pmatrix} F_1(\boldsymbol{x}) \\ \vdots \\ F_n(\boldsymbol{x}) \end{pmatrix}$$

Using the standard basis vectors in $\mathbb{R}^n$, $\boldsymbol{e}^{(1)}, \ldots, \boldsymbol{e}^{(n)}$ we can also write the vector field as:

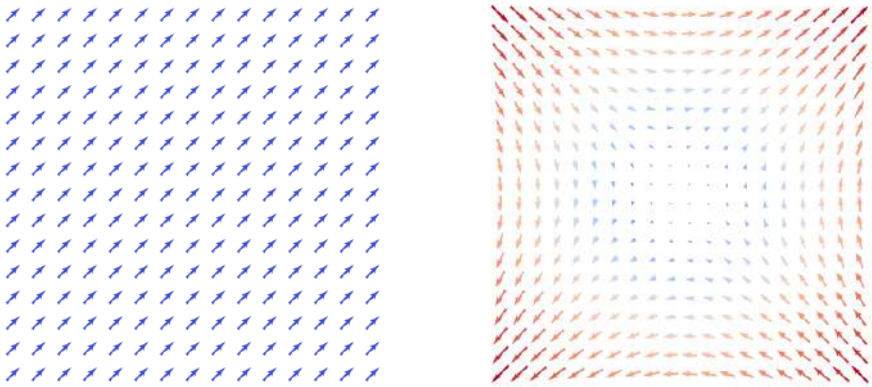
$$\vec{F}(\boldsymbol{x}) = F_1(\boldsymbol{x})\boldsymbol{e}^{(1)} + \cdots + F_n(\boldsymbol{x})\boldsymbol{e}^{(n)}$$

Since we are working in Euclidean space, vector fields and vector-valued functions can often be treated in the same way. We will therefore use the notation $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ to describe a vector-valued function, but it will behave like a vector field.[1]

2.3.2 Divergence Operator

Divergence, div for short, is an operator on a vector field that produces a scalar field that represents how much the density of some quantity changes at

[1] Importantly, vector fields map from a space to its tangential space. However, the tangential space to $\mathbb{R}^n$ is $\mathbb{R}^n$ itself, and we can therefore write it as a function to and from $\mathbb{R}^n$.



(a) $(1, 1)$ **(b)** $((\sin(y), \sin(x))$

Figure 2.3.1: A portion of two vector fields. The arrows depict the fields at discrete points. However, the field is continuous and exists for all points in space. The varying color corresponds to the length of the vectors. **Left**: The constant vector field (1, 1). **Right**: The vector field $(\sin(y), \sin(x))$.

each point. A positive divergence represents "pushing" the quantity away, decreasing local density while increasing the density of surrounding areas. A negative divergence is the opposite, "pulling" in the quantity and increasing the local density. One such quantity is probability. Then, the divergence measures how the probability density at any point changes, and by how much.

For an n-dimensional coordinate system $\mathbb{R}^n$ with coordinates $(x_1, \ldots, x_n)$ and standard basis vectors $\boldsymbol{e}^{(1)}, \ldots, \boldsymbol{e}^{(n)}$ (where $\boldsymbol{e}^{(i)}$ is an n-dimensional vector with all zeros, except for a 1 at position i), we define a continuous, differentiable, n-dimensional vector field $\vec{F} = F_1\boldsymbol{e}^{(1)} + \cdots + F_n\boldsymbol{e}^{(n)}$. The divergence is defined as:

$$\begin{aligned} \operatorname{div} \vec{F} &= \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}\right) \cdot (F_1, \ldots, F_n) \\ &= \frac{\partial F_1}{\partial x_1} + \cdots + \frac{\partial F_n}{\partial x_n} \end{aligned}$$

We can see that this is the sum of all the elements along the diagonal of the Jacobian:

$$\operatorname{div} \vec{F} = \operatorname{Tr}(\mathbf{J}(\vec{F}))$$

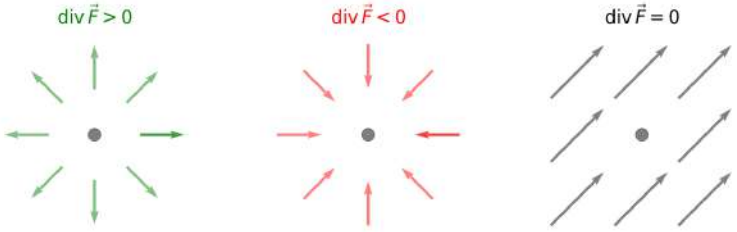


Figure 2.3.2: Left: When the divergence at a point is positive, the quantity is *increasing* there. **Center:** When the divergence at a point is negative, the quantity is *decreasing* there. **Right:** When the divergence at a point is zero, the quantity is *unchanged* there.

2.3.3 Continuity Equation

The **continuity equation**, also known as the **transport equation**, describes how a quantity, $\boldsymbol{x}_t \in \mathbb{R}^n$ moves in a space. We denote the density of the quantity $p_t(\boldsymbol{x}) : [0,1] \times \mathbb{R}^n \to \mathbb{R}$, and $\boldsymbol{u}_t(\boldsymbol{x}) : [0,1] \times \mathbb{R}^n \to \mathbb{R}^n$ is the vector field that describes the movement of $\boldsymbol{x}$. The continuity equation then states that:

$$\frac{\partial p_t(\boldsymbol{x})}{\partial t} = -\operatorname{div}(p_t(\boldsymbol{x})\boldsymbol{u}_t(\boldsymbol{x})) \tag{2.8}$$

A proof for the continuity equation can be found in Appendix B.1, which can help build an intuition for what this does and why it is useful.[2]

2.4 Deep Learning and Neural Networks

This section will give the reader a brief introduction to relevant concepts within deep learning. We will first cover what deep learning is, before diving into the inner workings of neural networks, a key algorithm used in deep learning.

[2]Since p_t is a scalar field and $\boldsymbol{u}_t$ is a vector field, the formula states that we take the divergence of the scalar multiplication between them, which is also a vector field with same dimensions as $\boldsymbol{u}_t$

2.4.1 Deep Learning

While many different definitions of **artificial intelligence** (AI) exist, one commonly used states that when computers perform tasks that are thought of to require intelligence to be solved by humans, they are exhibiting artificial intelligence. One of these tasks is learning, and this gives rise to a subfield of AI called **machine learning** (ML). The machine is tasked with learning from a set of data. They can use many different statistical models to achieve this goal, ranging from simple regression models to complex neural networks. If we are using neural networks, we call it **deep learning** (DL). Important concepts from deep learning will now be introduced, starting with neural networks.

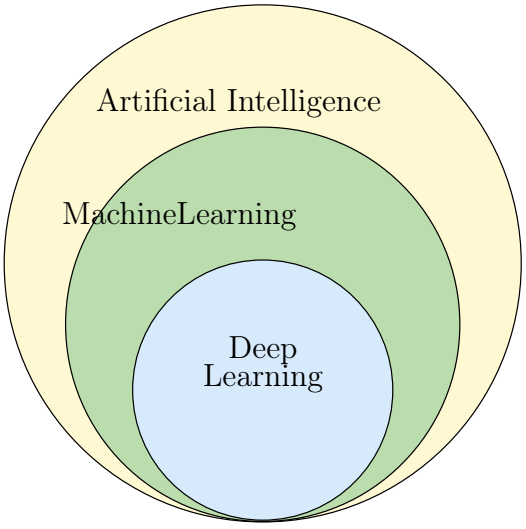


Figure 2.4.1: The relation between artificial intelligence, machine learning and deep learning. Figure taken from [7].

2.4.2 Neural Networks

A **neural network** (NN) consists of several neurons, arranged in layers, with edges between the neurons. A common type of neural networks are fully connected neural networks (FCNN). In them, each layer of neurons is fully connected to the subsequent layer.

The first layer is called the input layer, and the last is the output layer. It is important that they match the dimension of the input and output. In between these layer, we can have an arbitrary number of hidden layers. They

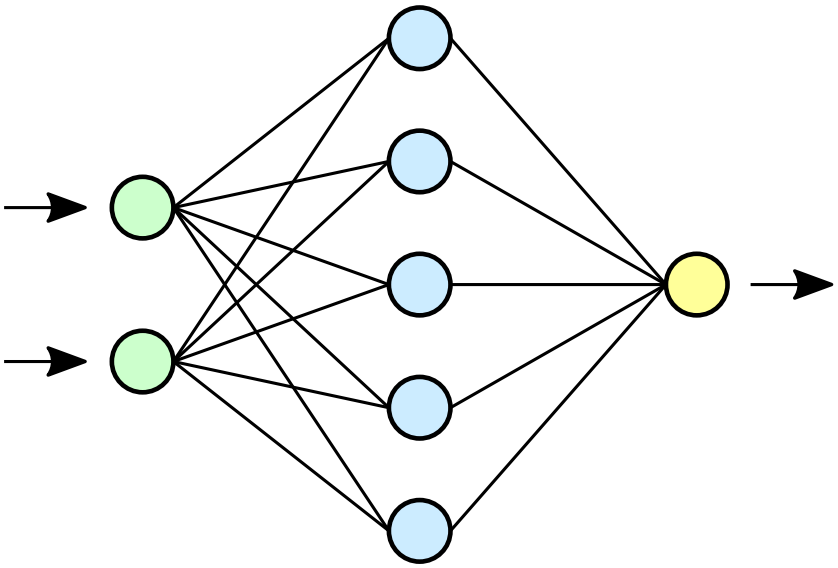


Figure 2.4.2: A fully-connected feed-forward neural network. The green neurons make up the input layer, the blue are the hidden layer and the yellow neuron makes up the output layer. Figure taken from [8].

can be of different dimensions. When some data (or input) is passed to the model, it is propagated through the model. Each neuron receives a signal, processes it, and propagates it forward. This is called the forward pass. Inside the neuron, each incoming signal is the product of the outgoing signal from the previous neuron, multiplied by the weight of the edge connecting them. The neuron calculates the sum of all incoming signals and adds its own bias to arrive at a value. This value is then passed through an activation function. A common choice is the rectified linear unit, or ReLU, which is given by $ReLU(x) = \max(0, x)$ [9]. This signal is then propagated forward in the network. We call the weights and biases the *parameters* of the network. It is the parameters we can change to make the network behave differently. However, networks can have billions of parameters, and so we need a smart way to update them.

For a given input-output pair x, y, we denote the networks output as $\hat{y} = f(x; \theta)$. We then define a loss function, $L(y, f(x; \theta))$, that we want to minimize. If we instead of a single input-output pair, have access to multiple samples from our data distribution, p_{data}, we can average over our losses using the cost-function

$$J(\theta) = \mathbb{E}_{x,y \sim p_{\text{data}}} L(y, f(x; \theta))$$

Using the cost function, we can measure the performance of the network given its current parameters. We can also calculate the gradient of J w.r.t. the network parameters. Recall from section 2.1.3 that the gradient points in the direction of greatest increase. Since we are looking to minimize J, we update our parameters using in the other direction

$$\theta_{\text{new}} = \theta - \alpha \cdot \nabla J(\theta)$$

where $\alpha \in \mathbb{R}$ is the *learning rate*, a hyperparameter we set. The algorithm used for efficiently calculating gradients in a neural network is called back-propagation [10].

2.4.3 Convolutional Neural Network

In fully connected networks, the number of parameters quickly grows, and especially if the dimensions of the input or the hidden layers are large. This is the case when working with images. A proposed solution to the problem is **convolution neural networks** (CNN). CNNs take advantage of the fact that spatially close pixels in an image, are often the most relevant when interpreting an image. Multiple filters are moved across the input to calculate the output. Therefore, the number of parameters is independent of the input size, determined only by the size and number of filters used.

2.4.4 Transformer

The **transformer** was introduced in 2017 [11]. Given an input sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, an encoder converts it to a latent representation $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$. This representation was then fed to a decoder generating the output sequence $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m$ one element at a time. While it was originally designed to be used in natural language tasks, such as translation, the architecture has proved to be very versatile and is commonly used in other domains as well. The transformer works well for all tasks where we the data can be ordered in a sequence.

2.4.4.1 Attention

The transformer uses scaled dot-product attention. Dot-product attention was originally introduced in [12]. The transformer uses both cross-attention

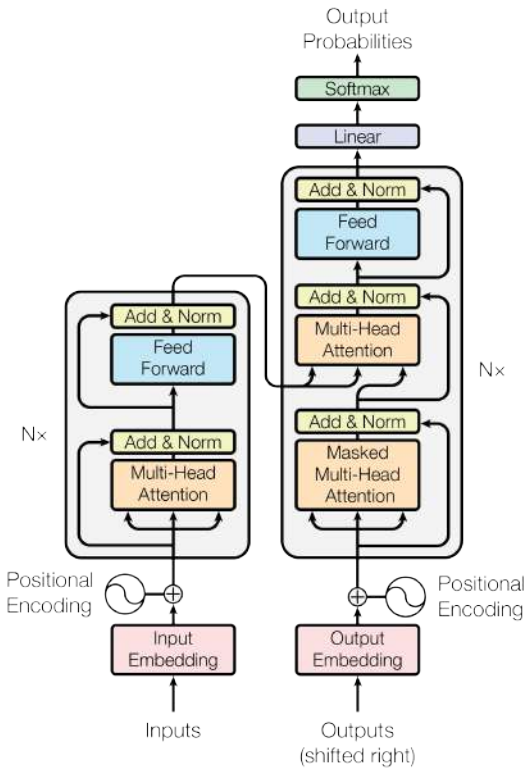


Figure 2.4.3: The transformer architecture as it was originally proposed. Figure from [11]

and self-attention. When generating the output sequence, the model can learn which parts of the latent representation are important for generating the next token. Since we are attending to a different sequence from the one we are generating, we call this cross-attention. In both the encoder and decoder, the transformer uses self-attention. Then, we are attending to the same sequence we are generating. When generating a sequence of words, the model can learn what previously generated words are important when generating the next one. If we consider a source sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ where $\forall i, \boldsymbol{x}_i \in \mathbb{R}^d$ and a target sequence $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m$, where $\forall i, \boldsymbol{y}_i \in \mathbb{R}^d$. By stacking the sequences vertically, we can represent them as two matrices

$$X = \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix}, \quad Y = \begin{bmatrix} \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix}$$

To calculate our attention, we first need to calculate the key, query and value

matrices. They are given by

$$Q = YW_Q$$
$$K = XW_K$$
$$V = XW_V$$

where W_Q, W_K, W_V are learnable parameters of our neural network, and $W_Q \in \mathbb{R}^{m \times d_k}$, $W_K \in \mathbb{R}^{n \times d_k}$ and $W_V \in \mathbb{R}^{n \times d_v}$. Both d_k, d_v are hyperparameters we can determine. The SoftMax function is applied to a vector $\boldsymbol{z} = (z_1, \ldots, z_K)$(if applied to a matrix, it is applied to each row individually) and is given by

$$\text{SoftMax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j = 1^K e^{z_j}}$$

The scaled dot-product attention is then given by

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{(QK^T)}{\sqrt{d_k}}\right) V$$

In the case of self-attention, we have $X = Y$.

One key factor of why attention works so well is that is is good at capturing long-range dependencies in sequences. Any two elements in the sequence can attend to each other, regardless of how far apart they are in the sequence. The SoftMax function also ensures that the sum of all attention weights must sum to 1. This means that the model has to deliberately choose where to put its attention, since attending more to one element in the sequence means that you have to attend less to the other elements. While the attention mechanism, like the transformer architecture, initially was only used in machine translation, it is now applied to many different subfields of deep learning.

2.4.5 UNet

The UNet architecture, proposed in [13], is a neural network originally designed for image segmentation of biomedical data. The original architecture, along with a short description, can be seen in Figure 2.4.4.

Since its inception, the UNet has evolved. Modern UNets often employ attention, as it has proven an efficient way to force models to concentrate on

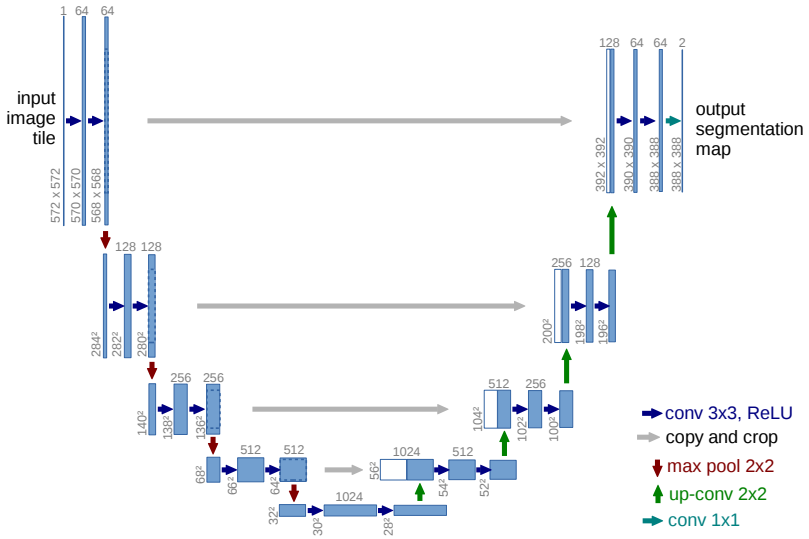


Figure 2.4.4: The original UNet. The left hand side consists of several down-sampling blocks, where the height and width of the input is decreased, while the depth is increased. The right hand side consists of up-sampling blocks, where the depth is decreased, while the height and width is increased. The gray arrows indicate skip connections, where the output of a down-sampling block is cropped and copied over to the corresponding up-sampling block and concatenated together with the output of the previous up-sampling block. Figure from [13]

important parts of the image, discarding details that are not important to the task. The number of down- and up-sampling blocks also vary, depending on the task. Finally, unlike the original architecture, it is also possible for the input and output dimensions to be equal.

2.4.6 Autoencoders

Autoencoders (AE) are another family of neural network architectures. The goal of AEs is to create low-dimensional representations of high dimensional data. Autoencoders consist of an encoder and a decoder. We call the space where the representations exist the latent space. The encoder is tasked with converting from input space to the latent space, while the decoder is trying to reconstruct the input from the latent space. By ensuring that the latent space is of a lower dimension than the input space, we force the model to compress the input signal into a meaningful representation. Given an input signal $\boldsymbol{x}$, an encoder $E_{\boldsymbol{\phi}} : \mathbb{R}^{\dim(\boldsymbol{x})} \to \mathbb{R}^{\dim(\boldsymbol{z})}$ and a decoder $D_{\boldsymbol{\theta}} : \mathbb{R}^{\dim(\boldsymbol{z})} \to \mathbb{R}^{\dim(\boldsymbol{x})}$, where

$\boldsymbol{\phi}, \boldsymbol{\theta}$ are the parameters of the encoder and decoder, we have that

$$E_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{z}$$
$$D_{\boldsymbol{\theta}}(\boldsymbol{z}) = D_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\boldsymbol{x})) = \hat{\boldsymbol{x}}$$

We tune the network parameters $\boldsymbol{\phi}, \boldsymbol{\theta}$ to minimize the reconstruction loss. For continuous data, one common choice is the mean-squared error (MSE) between the input $\boldsymbol{x}$ and the reconstructed input $\hat{\boldsymbol{x}}$. For a dataset $\boldsymbol{X} = \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$, it is given by

$$\text{MSE}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \mathbb{E}_{\boldsymbol{x} \sim \boldsymbol{X}}(||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2) = \frac{1}{n} \sum_{i=1}^{n} ||\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}^{(i)}||^2$$

Note that the dimension of the latent space representation is a hyperparameter. The smaller $\dim(\boldsymbol{z})$ is, the more the input is compressed.

$$\min_{\boldsymbol{\phi}, \boldsymbol{\theta}} \sum_{\boldsymbol{x} \in \boldsymbol{X}} ||\boldsymbol{x} - D_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\boldsymbol{x}))||^2$$

2.4.6.1 VAE

The **Variational Autoencoder** (VAE) is an improvement of the autoencoder proposed in [14]. Instead of the encoder output a vector $\boldsymbol{z}$, we output parameters that define a distribution. We denote the distribution modeled by the encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, and one common example is

$$E_{\boldsymbol{\phi}}(\boldsymbol{x}) = (\boldsymbol{\mu}, \log \boldsymbol{\sigma})$$
$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}, \operatorname{diag} \boldsymbol{\sigma})$$

This forces the latent variable $\boldsymbol{z}$ to follow a Gaussian distribution. The optimization objective in the VAE is therefore two-fold. Like with the autoencoder, we want to generate accurate samples. That is, we want the learned posterior $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ modeled by the decoder to approximate the true posterior $p(\boldsymbol{x})$. Additionally, we want $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ to approximate a distribution that we define, $p(\boldsymbol{z})$. $p(\boldsymbol{z})$ should be a distribution that we know the closed form of and can sample from efficiently. A common choice is the standard normal distribution $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$.

Kullback-Leibler (KL) divergence is a way to measure the distance between $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and $p(\boldsymbol{z})$. We write

$$D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})) = \mathbb{E}\left[\frac{\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{\log p(\boldsymbol{z}))}\right]$$

We would like to minimize this, as that means that the distributions are closer to each other. Furthermore, we also want to maximize the likelihood of the learned posterior $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$. This gives us the loss function

$$\mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}))$$

And finally, we wish to maximize this objective w.r.t. the parameters of the encoder and decoder

$$\max_{\boldsymbol{\phi},\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}}(\boldsymbol{x})$$

Since $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}$ is a random variable, it is not straightforward how we can calculate the gradients. The way this is done is by letting $\boldsymbol{z}$ be expressed as a differentiable and invertible function $f(\epsilon, \boldsymbol{\phi}, \boldsymbol{x})$, where ϵ is a random variable independent of $\boldsymbol{\phi}, \boldsymbol{x}$. When $\boldsymbol{z}$ follows a Gaussian, we can let $\epsilon \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. Instead of sampling $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}$, we have that

$$\begin{aligned} E_{\boldsymbol{\phi}}(\boldsymbol{x}) &= (\boldsymbol{\mu}, \log \boldsymbol{\sigma}) \\ \epsilon &\sim p(\epsilon) \\ \boldsymbol{z} &= \boldsymbol{\mu} + \boldsymbol{\sigma} * \epsilon \end{aligned}$$

We can now calculate the gradients through the encoder, as the only variable we are sampling is ϵ (see Figure 2.4.5). This is known as the **reparameterization trick**.

2.4.6.2 VQVAE

While the AE and VAE work with continuous latent variables, there are cases where a discrete variable would be preferable. For instance when working with autoregressive transformers, it has proven beneficial to use discrete tokens. In the natural language domain, this is easily done. We map each word

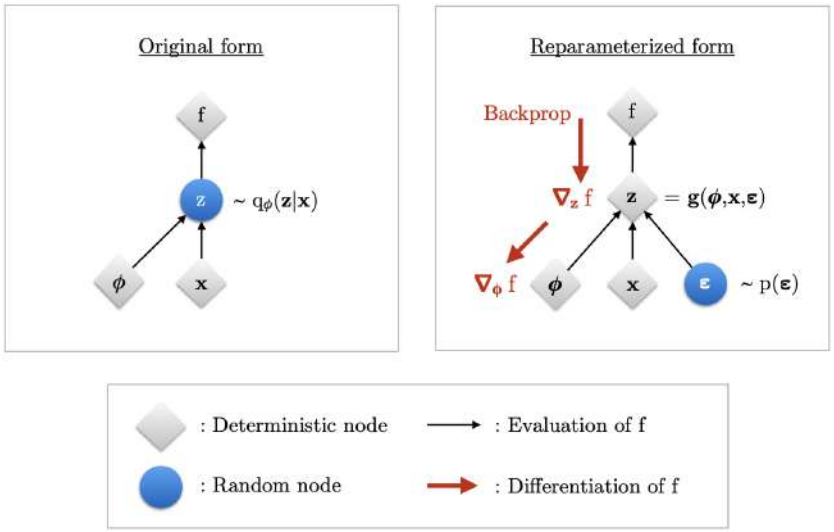


Figure 2.4.5: The reparameterization trick. By letting $\boldsymbol{z}$ not be sampled directly, and instead sampling ϵ, we are able to efficiently calculate the gradients of the encoder and optimize our loss function using backpropagation and gradient ascent. Figure taken form [15].

to a unique integer, to form a mapping from all possible words to tokens (the integers) and back. But what if the data is continuous (or of very high dimensionality), like in the case of images? How can we then map it to a finite set of values? A naive solution would be to map each pixel-value to an integer, similar to words, but this quickly grows out of proportions. While a typical mapping for language might have around 50,000 entries [16], doing this for images would lead to more than 16 million entries (assuming that each pixel consists of 3 values from 0 to 255). A better way to do it, would be to use a **vector quantized variational autoencoder** (VQVAE)[17].

The VQVAE shares a lot with the variational autoencoder. However, the latent variables passed into the decoder is now discrete. This is done through vector quantization. We still assume that we have an encoder $E_{\boldsymbol{\phi}}$ defined by the distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and have that

$$E_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{z}_e(\boldsymbol{x})$$

So far, $\boldsymbol{z}_e$ is still a continuous vector. The next step is vector quantization. We choose k, the size of our codebook, and define it as a matrix with dimension $K \times \dim(\boldsymbol{z}_e)$. We denote a row in the matrix $\boldsymbol{e}_k$ for all $k = 1, \ldots, K$, and initialize it randomly. Instead of passing the output of the encoder directly

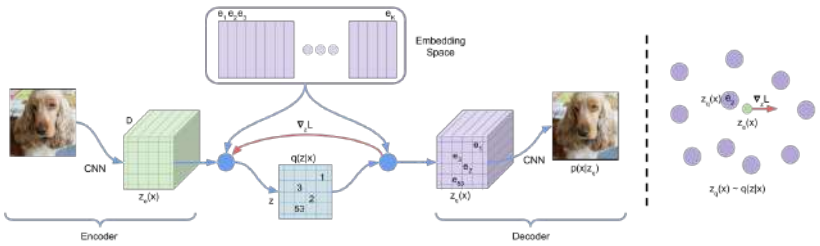


Figure 2.4.6: Left: The VQVAE architecture with encoder, decoder and codebook. The red line represents the straight-through estimator of the gradients that allow us to optimize the encoder with reconstruction loss. **Right:** The embedding space. The encoder output, $\boldsymbol{z}_e$, maps to the closest embedding in the codebook. Figure taken from [17].

into the decoder, like in the AE, we do a lookup in the codebook. The distribution of the latent variable k (which is now an integer, not a vector), is given by

$$q_{\boldsymbol{\phi}}(k|\boldsymbol{x}) = \begin{cases} 1, & \text{for } k = \text{argmin}_j \, ||\boldsymbol{z}_e - \boldsymbol{e}_j|| \\ 0, & \text{otherwise} \end{cases}$$

When decoding, we convert the index back to a vector using our codebook and the integer k

$$\boldsymbol{z}_q = \boldsymbol{e}_k \tag{2.9}$$

The loss function of the VQVAE consists of three parts. The first is the *reconstruction loss*

$$\mathcal{L}_{\text{rec}} = \mathbb{E}_{\boldsymbol{z}_q \sim q_{\boldsymbol{\phi}}} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_q) \right]$$

It optimizes both the encoder and decoder. Since the discretization operation in Equation 2.9 is non-differentiable, we simply pass the unaltered gradients from the decoder to the encoder. This is known as straight-through estimation.

The second part is the *codebook loss*. This loss aims to move the vectors in the codebook towards the output of the encoder. It uses the stopgradient operator

$$\begin{aligned} \text{sg}[\boldsymbol{x}] &= \boldsymbol{x}, \quad \text{during the forward pass} \\ \nabla \text{sg}[\boldsymbol{x}] &= 0, \quad \text{during the backward pass} \end{aligned}$$

The codebook loss is given by

$$\mathcal{L}_{\text{cb}} = ||\,\text{sg}[\boldsymbol{z}_e] - \boldsymbol{z}_q||^2$$

The stopgradient (sg) operator prevents the gradients from flowing back to the encoder, by setting $\nabla_{\boldsymbol{z}_e}\mathcal{L}_{\text{cb}} = 0$ during backpropagation. Therefore, this loss only serves to move to the codebook entries closer to the encoder output. The final loss is the commitment loss. It has a scaling factor β, which is a hyperparameter. The original paper uses $\beta = 0.25$. The commitment loss is given by

$$\mathcal{L}_{\text{cm}} = \beta||\boldsymbol{z}_e - \text{sg}[\boldsymbol{z}_q]||^2$$

This loss ensures that the encoder commits to an embedding. In other words, we force the output of the encoder to move towards the embeddings. The reason for having both $\mathcal{L}_{\text{cb}}$ and $\mathcal{L}_{\text{cm}}$, which are seemingly very similar, are crucial for stable training [17]. When $\beta < 1$, we move the codebook entries towards the encoder output faster than we move the encoder output towards the codebook entries. Conversely, if $\beta > 1$, the rate of change is fastest for the encoder outputs moving towards the codebook entries. The total loss is given by

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{rec}} + \mathcal{L}_{\text{cb}} + \mathcal{L}_{\text{cm}}$$

We want to minimize all these terms, by optimizing the parameters for the encoder and decoder, $\boldsymbol{\phi}, \boldsymbol{\theta}$ and the entries of the codebook $\boldsymbol{e}_i$.

2.5 Normalizing Flows

A **normalizing flow** describes a set of mappings applied to a probability density in succession and were first introduced in [18]. Given $\boldsymbol{x}_k \sim p_k$, we define $\boldsymbol{u}_{k+1}$ s.t. the relationship between $\boldsymbol{x}_k$ and $\boldsymbol{x}_{k+1}$ is given by

$$\boldsymbol{x}_{k+1} = \boldsymbol{u}_{k+1}(\boldsymbol{x}_k) \tag{2.10}$$

We require that a mapping $\boldsymbol{u}_k : \mathbb{R}^n \to \mathbb{R}^n$ is smooth and invertible. Given a random variable $\boldsymbol{x}_0 \sim p_0$, we can transform it into $\boldsymbol{x}_K \sim p_K$ by applying a chain of K transformations $\boldsymbol{u}_k$, which we can denote $\boldsymbol{\phi}_K$

$$\boldsymbol{x}_K = \boldsymbol{u}_K(\boldsymbol{u}_{K-1}(\ldots \boldsymbol{u}_1(\boldsymbol{x}_0))) = \boldsymbol{\phi}_K(\boldsymbol{x}_0) \tag{2.11}$$

Since all mappings are invertible, it follows that:

$$\boldsymbol{x}_k = \boldsymbol{u}_{k+1}^{-1}(\boldsymbol{x}_{k+1}) \tag{2.12}$$
$$\boldsymbol{x}_0 = \boldsymbol{u}_1^{-1}(\boldsymbol{u}_2^{-1}(\ldots \boldsymbol{u}_K^{-1}(\boldsymbol{x}_K))) \tag{2.13}$$

Using Equation 2.7 together with Equation 2.13 we get:

$$\log p_K(\boldsymbol{x}_K) = \log \left(p_0(\boldsymbol{u}_1^{-1}(\boldsymbol{u}_2^{-1}(\ldots \boldsymbol{u}_K^{-1}(\boldsymbol{x}_K)))) \prod_{k=1}^{K} \left| \det \frac{\partial \boldsymbol{u}_k}{\partial \boldsymbol{x}_{k-1}} \right|^{-1} \right) \tag{2.14}$$

$$= \log p_0(\boldsymbol{x}_0) + \sum_{k=1}^{K} \log \left| \det \frac{\partial \boldsymbol{u}_k}{\partial \boldsymbol{x}_{k-1}} \right|^{-1} \tag{2.15}$$

$$= \log p_0(\boldsymbol{x}_0) + \sum_{k=1}^{K} \log \left| \det \mathbf{J}(\boldsymbol{u}_k)(\boldsymbol{x}_{k-1}) \right|^{-1} \tag{2.16}$$

Given a prior $p_{\text{prior}} \sim \mathcal{N}(0, 1)$ and the distribution we want to learn, p_{data}, we set p_0 equal to p_{prior}, and want to learn transformations s.t. p_K is as close as possible to p_{data}. The idea behind normalizing flows is to learn these transformations with neural networks. We do this by transforming data samples $\boldsymbol{x}_K \sim p_{\text{data}}$ through the inverse mappings in Equation 2.13 to get $\boldsymbol{x}_0$, learned by the neural networks with parameters $\boldsymbol{\theta}_k$. The transformations take the previous input, and transforms it using the neural networks parameters, $\boldsymbol{x}_k = \boldsymbol{u}_k^{-1}(\boldsymbol{x}_{k+1}; \boldsymbol{\theta}_k)$. Finally, we maximize the log-likelihood of $\log p_K(\boldsymbol{x}_K)$ as defined in Equation 2.16.

In addition to the strict requirement that these mappings are invertible, we can see from Equation 2.16 that it is beneficial if the determinant of the

Jacobian is simple to compute. One way to do this is to impose some structure on the Jacobian. Previous work includes enforcing a low-rank Jacobian [18] or a lower triangular Jacobian [19]. However, this often comes at the cost of a reduced expressivity, as we no longer allow the transformations to have a full-rank Jacobian.

2.6 Continuous Normalizing Flows

While we have previously looked at discrete normalizing flows where we set a step size δ or some fixed amount of steps, K, it would be interesting to consider a **continuous normalizing flow** (CNF). CNFs were introduced in [20] and take their inspiration from Residual Networks (ResNets). For an input $\boldsymbol{x}_k \in \mathbb{R}^n$, and a step-dependent function $\boldsymbol{f} : \{0, 1, \ldots, K\} \times \mathbb{R}^n \to \mathbb{R}^n$, a ResNet layer calculates $\boldsymbol{x}_{k+1}$:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{f}(k, \boldsymbol{x}_k)$$

We can rewrite this as

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \boldsymbol{f}(k, \boldsymbol{x}_k)$$

Instead of using K discrete steps, we use a continuous variable $t \in [0, 1]$. Instead of a step-dependent function $\boldsymbol{f}$, we have a time dependent function $\boldsymbol{u} : [0, 1] \times \mathbb{R}^n \to \mathbb{R}^n$. We say that $\boldsymbol{u}$ is a time-dependent vector field. Note that the notation $\boldsymbol{u}(t, \boldsymbol{x})$ and $\boldsymbol{u}_t(\boldsymbol{x})$ will be used interchangeably going forward, following the notation in [21, 22]. Instead of looking at the difference between $\boldsymbol{x}_k$ and $\boldsymbol{x}_{k+1}$, we define $\delta > 0$ and look at

$$\frac{\boldsymbol{x}_{t+\delta} - \boldsymbol{x}_t}{\delta} = \boldsymbol{u}(t, \boldsymbol{x})$$

By letting $\delta \to 0$ we get the following ODE:

$$\mathrm{d}\boldsymbol{x}_t = \boldsymbol{u}(t, \boldsymbol{x})\mathrm{d}t \tag{2.17}$$

We denote the solution to the ODE $\boldsymbol{\phi}(t, \boldsymbol{x}_0)$, or $\boldsymbol{\phi}_t(\boldsymbol{x}_0)$, with initial condition $\boldsymbol{\phi}_0(\boldsymbol{x}_0) = \boldsymbol{x}_0$. As for discrete normalizing flows, we have that $\boldsymbol{x}_t = \boldsymbol{\phi}_t(\boldsymbol{x}_0)$. This means that we can write

$$\mathrm{d}\boldsymbol{\phi}_t(\boldsymbol{x}_0) = \boldsymbol{u}(t, \boldsymbol{x})\mathrm{d}t$$

or, by dividing both sides by $\mathrm{d}t$

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\phi}_t(\boldsymbol{x}_0) = \boldsymbol{u}(t, \boldsymbol{x})$$

In other words, $\boldsymbol{\phi}_t(\boldsymbol{x}_0)$ is the point $\boldsymbol{x}$ moved along the vector field $\boldsymbol{u}$ from time 0 to t. This corresponds to integrating w.r.t time on both sides

$$\begin{aligned}\int_0^t \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\phi}_t(\boldsymbol{x}_0) &= \int_0^t \boldsymbol{u}(t, \boldsymbol{x}) \\ \boldsymbol{\phi}_t(\boldsymbol{x}_0) &= \int_0^t \boldsymbol{u}(t, \boldsymbol{x})\end{aligned}$$

If we instead of a point $\boldsymbol{x}_0$, have a density p_0, we say that $\boldsymbol{\phi}_t$ induces a push-forward along $\boldsymbol{u}$ from 0 to t. The time-dependent PDF $p : [0, 1] \times \mathbb{R}^n \to \mathbb{R}$ is characterized by the continuity equation (see Equation 2.8):

$$\frac{\partial p_t(\boldsymbol{x})}{\partial t} = -\operatorname{div}(p_t(\boldsymbol{x})\boldsymbol{u}_t(\boldsymbol{x}))$$

with initial condition p_0. We say that $\boldsymbol{u}$ is the **probability flow ODE** for p, and that p is the **probability path** generated by $\boldsymbol{u}$. Note that, unlike discrete normalizing flows, CNFs allow us to use full-rank Jacobians. This means that the transformations are expressive and flexible. However, training CNFs is slow since we have to solve differential equations to move points along the vector field $\boldsymbol{u}$, which is costly.

2.7 Flow Matching

Given that there exists a vector field $\boldsymbol{u} : [0, 1] \times \mathbb{R}^n \to \mathbb{R}^n$ that allows us to transform $p_0 \approx p_{\text{prior}}$ into $p_1 \approx p_{\text{data}}$ from time 0 to 1, the goal of **flow matching** (FM) is to learn that approximation, the vector field $\boldsymbol{v}_{\boldsymbol{\theta}} : [0, 1] \times \mathbb{R}^n \to \mathbb{R}^n$, where $\boldsymbol{\theta}$ is a set of parameters that define the vector field. This vector field generates a time-dependent probability path $p : [0, 1] \times \mathbb{R}^n \to \mathbb{R}$, where p_t denotes the probability distribution at time $t \in [0, 1]$. The goal is that $p_0 \approx p_{\text{prior}}$ and $p_1 \approx p_{\text{data}}$. However, there exists an infinite number of

vector fields that can transform p_0 into p_1. If we want to establish a ground truth for all values of t, we will see that we must specify what the probability path p_t should look like[3].

2.7.1 Learning the Vector Field with Neural Networks

We assume that we know the probability path $p_t(\boldsymbol{x})$ and the associated vector field $\boldsymbol{u}_t(\boldsymbol{x})$, and that we can sample from $p_t(\boldsymbol{x})$. We then define a neural network with weights $\boldsymbol{\theta}$, and let $\boldsymbol{v}_{\boldsymbol{\theta}}(t, \boldsymbol{x}) : [0, 1] \times \mathbb{R}^n \to \mathbb{R}^n$ be a time-dependent vector field with parameters $\boldsymbol{\theta}$. We can then regress this towards $\boldsymbol{u}$ with the flow matching objective [21]:

$$\mathcal{L}_{\text{FM}}(\boldsymbol{\theta}) := \int_t \mathbb{E}_{\boldsymbol{x}\sim p_t(\boldsymbol{x})} ||\boldsymbol{v}_{\boldsymbol{\theta}}(t, \boldsymbol{x}) - \boldsymbol{u}_t(\boldsymbol{x})||^2 \mathrm{d}t$$

Alternatively, it can also be written as

$$\mathcal{L}_{\text{FM}}(\boldsymbol{\theta}) := \mathbb{E}_{t\sim\mathcal{U}(0,1),\boldsymbol{x}\sim p_t(\boldsymbol{x})} ||\boldsymbol{v}_{\boldsymbol{\theta}}(t, \boldsymbol{x}) - \boldsymbol{u}_t(\boldsymbol{x})||^2 \tag{2.18}$$

This is sometimes preferred as it signifies how we can use Monte Carlo sampling to efficiently approximate taking the integral over t. Nevertheless, it is rare that we have prior knowledge of what $\boldsymbol{u}_t$ and p_t should be, and thus have nothing to regress the neural network against. Instead, we will see that we must define conditional probability paths and a conditional vector field, and regress against that.

2.7.2 Conditional Flow Matching

The idea of **conditional flow matching** (CFM) is to define a distribution form which we can draw our **conditioning variable** $\boldsymbol{z} \sim q(\boldsymbol{z})$ independent of t. We then choose the **conditional probability paths** $p_t(\boldsymbol{x}|\boldsymbol{z})$. These must satisfy

$$\begin{aligned} \forall \boldsymbol{x}, \mathbb{E}_{\boldsymbol{z}} \left[p_0(\boldsymbol{x}|\boldsymbol{z})\right] &= p_0(\boldsymbol{x}) \\ \forall \boldsymbol{x}, \mathbb{E}_{\boldsymbol{z}} \left[p_1(\boldsymbol{x}|\boldsymbol{z})\right] &= p_1(\boldsymbol{x}) \end{aligned} \tag{2.19}$$

[3]Note that we write $\boldsymbol{u}_t(\boldsymbol{x})$ and $\boldsymbol{u}(t, \boldsymbol{x})$ interchangeably. Similarly, $\boldsymbol{v}_t(\boldsymbol{x}) = \boldsymbol{v}(t, \boldsymbol{x})$ and $p_t(\boldsymbol{x}) = p(t, \boldsymbol{x})$.

This ensures that the conditional probability path starts at p_0 and ends at p_1. We can then use the conditional probability path to determine what the conditional vector field, $\boldsymbol{u}_t^{\text{cond}}(\boldsymbol{x}, \boldsymbol{z})$ that generate these paths, looks like. The relationship between them is given by the continuity equation (see Equation 2.8).

Commonly, we set $q(\boldsymbol{z}) = p_{\text{data}}$, giving $\boldsymbol{z} = \boldsymbol{x}_1 \sim p_{\text{data}}$ or $q(\boldsymbol{z}) = p_{\text{prior}} \times p_{\text{data}}$ s.t. $\boldsymbol{z} = (\boldsymbol{x}_0, \boldsymbol{x}_1) \sim p_{\text{prior}} \times p_{\text{data}}$. However, we can also enforce stricter constraint on these pairings should we wish to do so.

One extension of conditional flow matching, that enforces even stricter constraints on the data pairs, is **optimal transport conditional flow matching** (OTCFM) introduced in [22]. We start by sampling $(\boldsymbol{x}_0, \boldsymbol{x}_1) \sim p_{\text{prior}}\times$ to construct n initial pairs $(\boldsymbol{x}_0^{(1)}, \boldsymbol{x}_1^{(1)}), \ldots, (\boldsymbol{x}_0^{(n)}, \boldsymbol{x}_1^{(n)})$. We then solve the optimal transport (OT) plan. In our case, with a set of input points and a set of target points, this equates to a mapping problem. If we let $\sigma(i)$ be a permutation of the sequence $1, 2, \ldots, n$. We let $c(\boldsymbol{x}_0^{(i)}, \boldsymbol{x}_1^{(j)})$ be the cost of transporting from $\boldsymbol{x}_0^{(i)}$ to $\boldsymbol{x}_1^{(j)}$. We denote the total cost

$$C(\sigma) = \sum_{i=1}^{n} c(\boldsymbol{x}_0^{(i)}, \boldsymbol{x}_1^{(\sigma(i))})$$

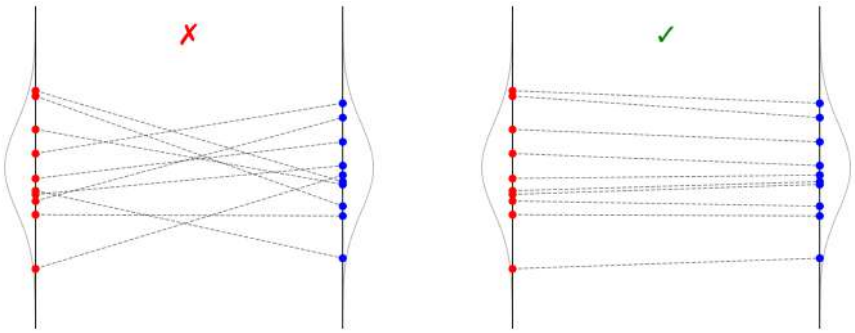
We wish to find σ^* s.t. C is minimized (or if an exact solution is not feasible, as small as possible). A common choice for c is the 2-Wasserstein distance, which in Euclidean space is the L2-norm

$$W_2(\boldsymbol{x}_0, \boldsymbol{x}_1) = ||\boldsymbol{x}_0 - \boldsymbol{x}_1||$$

If the dataset is small, we can solve the global OT plan. However, for large datasets, this is not feasible (the time is $\mathcal{O}(n^3)$ and the memory requirement is $\mathcal{O}(n^2)$). We therefore sample a subset of the data, and solve the minibatch OT. This has been shown empirically to improve the performance of flow matching model for image generation [22].

2.7.2.1 Gaussian Marginals

In general, determining a probability path does not mean that it is easy to find its corresponding conditional vector field. However, if we limit ourselves to probability paths that follow a Gaussian distribution, it becomes straightforward. We define $\boldsymbol{\mu}_t(\boldsymbol{z}) : [0, 1] \times \mathbb{R}^{|\boldsymbol{z}|} \to \mathbb{R}^n$ and $\sigma_t : [0, 1] \to \mathbb{R}$,



(a) Not optimal transport **(b)** Optimal transport

Figure 2.7.1: Two sets of points, red and blue, are drawn from the same 1D Gaussian distribution. Lines are drawn between them, representing the pairing of points. The cost can be thought of as the sum of the length of all lines. **Left**: The pairs do not follow optimal transport. **Right**: The pairs do follow optimal transport. In the case of 1-dimension data, this means that lines can never cross.

a function describing the mean and standard deviation over time. These functions depend on the conditioning variable $\boldsymbol{z}$. We then have the probability path

$$p_t(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_t(\boldsymbol{z}), \sigma_t(\boldsymbol{z}) \cdot \boldsymbol{I}_n)$$

We now need to find a solution to Equation 2.17, with Gaussian marginals (that is, p_t is Gaussian for all t). One such solution is given in [22] (from now on we will omit the argument of $\boldsymbol{\mu}_t(\boldsymbol{z}), \sigma_t(\boldsymbol{z})$ to make notation more compact)

$$\boldsymbol{x}_t = \boldsymbol{\phi}_t(\boldsymbol{x}_0) = \boldsymbol{\mu}_t + \sigma_t \cdot \boldsymbol{I}_n \left(\frac{\boldsymbol{x}_0 - \boldsymbol{\mu}_0}{\sigma_0 \cdot \boldsymbol{I}_n} \right) \tag{2.20}$$

We can find its corresponding vector field by using the same equation as Equation 2.17, only in the conditional case. For ease of notation, we define $\boldsymbol{w}_t(\boldsymbol{x}_t) = \boldsymbol{u}^{\text{cond}}(\boldsymbol{x}_t|\boldsymbol{z})$:

$$\frac{\mathrm{d}\boldsymbol{x}_t}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\phi}_t(\boldsymbol{x}_0) = \boldsymbol{w}_t(\boldsymbol{\phi}_t(\boldsymbol{x}_0))$$

As long as $\sigma_t > 0$, $\boldsymbol{\phi}_t$ is strictly monotone and therefore invertible. We let $\boldsymbol{x}_0 = \boldsymbol{\phi}_t^{-1}(\boldsymbol{x}_t)$ and get

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\phi}_t(\boldsymbol{\phi}_t^{-1}(\boldsymbol{x}_t)) = \boldsymbol{w}_t(\boldsymbol{x}_t) \tag{2.21}$$

We then invert $\boldsymbol{\phi}_t$ in Equation 2.20

$$\boldsymbol{x}_0 = \boldsymbol{\phi}_t^{-1}(\boldsymbol{x}_t) = \left(\frac{\boldsymbol{x}_t - \boldsymbol{\mu}_t}{\sigma_t \cdot \boldsymbol{I}_n}\right)\sigma_0 \cdot \boldsymbol{I}_n + \boldsymbol{\mu}_0$$

We differentiate Equation 2.20 with respect to t, and use the compact notation $\boldsymbol{\mu}_t', \sigma_t'$ to denote $\frac{\mathrm{d}\boldsymbol{\mu}_t}{\mathrm{d}t}, \frac{\mathrm{d}\sigma_t}{\mathrm{d}t}$

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\phi}_t(\boldsymbol{x}_0) = \mu_t' + \sigma_t' \cdot \boldsymbol{I}_n\left(\frac{\boldsymbol{x}_0 - \mu_0}{\sigma_0 \cdot \boldsymbol{I}_n}\right)$$

Combining the last two equations with Equation 2.21 we get

$$\begin{aligned}
\boldsymbol{w}_t(\boldsymbol{x}_t) &= \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\phi}_t(\boldsymbol{\phi}_t^{-1}(\boldsymbol{x}_t)) \\
&= \boldsymbol{\mu}_t' + \sigma_t' \cdot \boldsymbol{I}_n\left(\frac{\boldsymbol{\phi}_t^{-1}(\boldsymbol{x}_t) - \boldsymbol{\mu}_0}{\sigma_0 \cdot \boldsymbol{I}_n}\right) \\
&= \boldsymbol{\mu}_t' + \sigma_t' \cdot \boldsymbol{I}_n\left(\frac{\left(\frac{\boldsymbol{x}_t - \boldsymbol{\mu}_t}{\sigma_t \cdot \boldsymbol{I}_n}\right)\sigma_0 \cdot \boldsymbol{I}_n + \boldsymbol{\mu}_0 - \boldsymbol{\mu}_0}{\sigma_0 \cdot \boldsymbol{I}_n}\right) \\
&= \boldsymbol{\mu}_t' + \sigma_t' \cdot \boldsymbol{I}_n\left(\frac{\boldsymbol{x}_t - \boldsymbol{\mu}_t}{\sigma_t \cdot \boldsymbol{I}_n}\right)
\end{aligned}$$

Theorem 2.7.2.1. (Theorem 3 of [21]). *Given a probability path*

$$p_t(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_t(\boldsymbol{z}), \sigma_t^2(\boldsymbol{z}) \cdot \boldsymbol{I})$$

and its corresponding flow

$$\boldsymbol{x}_t = \boldsymbol{\phi}_t(\boldsymbol{x}_0) = \boldsymbol{\mu}_t + \sigma_t \cdot \boldsymbol{I}_n\left(\frac{\boldsymbol{x}_0 - \boldsymbol{\mu}_0}{\sigma_0 \boldsymbol{I}_n}\right)$$

The unique vector field that defines the flow $\boldsymbol{\phi}_t$ is given by

$$\boldsymbol{u}_t^{\mathrm{cond}}(\boldsymbol{x}_t|\boldsymbol{z}) = \boldsymbol{\mu}_t' + \sigma_t' \cdot \boldsymbol{I}_n\left(\frac{\boldsymbol{x}_t - \boldsymbol{\mu}_t}{\sigma_t \cdot \boldsymbol{I}_n}\right)$$

We now have a clear connection between the conditional vector field, $\boldsymbol{u}_t^{\text{cond}}(\boldsymbol{x}_t|\boldsymbol{z})$ and $p_t(\boldsymbol{x}|\boldsymbol{z})$. We are now able to go from conditional probability paths to conditional vector fields easily. The next step now is to go from conditional vector fields, to the unconditional vector field. But first, we will show an example where we define conditional probability paths, and find the corresponding conditional vector field.

2.7.2.2 Example: Linear Interpolation

We will now look at one way to construct the conditional probability paths, introduced in [23, 24]. We first choose the conditioning variable $\boldsymbol{z} \sim q(\boldsymbol{z}) = p_{\text{prior}} \times p_{\text{data}}$. Once we have sampled the two points $(\boldsymbol{x}_0, \boldsymbol{x}_1) \sim q$, we can define the probability path between them. We set this to be the linear interpolation between them. Since it has to be a probability distribution, we use the a Gaussian with a variance, $\sigma_t^2 = \epsilon$ for all t, where $\epsilon > 0$ is a small, positive number, and mean given by the linear interpolation $\boldsymbol{\mu}_t = (1-t) \cdot \boldsymbol{x}_0 + t \cdot \boldsymbol{x}_1$

$$\begin{aligned} p_t(\boldsymbol{x}, \boldsymbol{z} = (\boldsymbol{x}_0, \boldsymbol{x}_1)) &= \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_t, \sigma_t^2 \cdot \boldsymbol{I}_n) \\ &= \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_t = (1-t) \cdot \boldsymbol{x}_0 + t \cdot \boldsymbol{x}_1, \boldsymbol{\Sigma}^2 = \epsilon \cdot \boldsymbol{I}_n) \end{aligned}$$

It is obvious that $\boldsymbol{z}$ is independent of t. Furthermore, we need to verify the two boundary constraints, seen in Equation 2.19.

$$
\begin{aligned}
\int_{\boldsymbol{z}} p_0(\boldsymbol{x}|\boldsymbol{z}) q(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} &= \int_{\boldsymbol{z}} \mathcal{N}(\boldsymbol{x}|\boldsymbol{z}, \epsilon \cdot \boldsymbol{I}_n) q(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} \\
&= \int_{\boldsymbol{z}} \mathcal{N}(\boldsymbol{x}|(1-0) \cdot \boldsymbol{x}_0 + 0 \cdot \boldsymbol{x}_1, \epsilon \cdot \boldsymbol{I}_n) q(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} \\
&= \int_{\boldsymbol{z}} \mathcal{N}(\boldsymbol{x}|\boldsymbol{x}_0, \epsilon \cdot \boldsymbol{I}_n) p_0(\boldsymbol{x}_0) \mathrm{d}\boldsymbol{z} \\
&= \int_{\boldsymbol{x}_0} \mathcal{N}(\boldsymbol{x}|\boldsymbol{x}_0, \epsilon \cdot \boldsymbol{I}_n) p_0(\boldsymbol{x}_0) \mathrm{d}\boldsymbol{x}_0 \approx p_0(\boldsymbol{x}_0) \\
\int_{\boldsymbol{z}} p_1(\boldsymbol{x}|\boldsymbol{z}) q(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} &= \int_{\boldsymbol{z}} \mathcal{N}(\boldsymbol{x}|\boldsymbol{z}, \epsilon \cdot \boldsymbol{I}_n) q(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} \\
&= \int_{\boldsymbol{z}} \mathcal{N}(\boldsymbol{x}|(1-1) \cdot \boldsymbol{x}_0 + 1 \cdot \boldsymbol{x}_1, \epsilon \cdot \boldsymbol{I}_n) q(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} \\
&= \int_{\boldsymbol{z}} \mathcal{N}(\boldsymbol{x}|\boldsymbol{x}_1, \epsilon \cdot \boldsymbol{I}_n) p_1(\boldsymbol{x}_1) \mathrm{d}\boldsymbol{z} \\
&= \int_{\boldsymbol{x}_1} \mathcal{N}(\boldsymbol{x}|\boldsymbol{x}_1, \epsilon \cdot \boldsymbol{I}_n) p_1(\boldsymbol{x}_1) \mathrm{d}\boldsymbol{x}_1 \approx p_1(\boldsymbol{x}_1)
\end{aligned}
$$

We calculate $\boldsymbol{\mu}'_t$ and σ'_t, and then use Theorem 2.7.2.1 to find the corresponding conditional vector field

$$
\begin{aligned}
\boldsymbol{\mu}'_t &= \frac{\mathrm{d}}{\mathrm{d}t}((1-t) \cdot \boldsymbol{x}_0 + t \cdot \boldsymbol{x}_1) \\
&= \boldsymbol{x}_1 - \boldsymbol{x}_0 \\
\sigma'_t &= \frac{\mathrm{d}}{\mathrm{d}t}\epsilon = 0 \\
\boldsymbol{u}_t^{\mathrm{cond}}(\boldsymbol{x}_t) &= \boldsymbol{\mu}'_t + \sigma'_t \cdot \boldsymbol{I}_n \left(\frac{\boldsymbol{x}_t - \boldsymbol{\mu}_t}{\sigma_t \cdot \boldsymbol{I}_n} \right) \\
&= \boldsymbol{x}_1 - \boldsymbol{x}_0
\end{aligned}
$$

2.7.3 Regressing Against the Conditional Vector Field

In the previous section, we learned how we can define conditional probability paths with Gaussian marginals $p_t(\boldsymbol{x}|\boldsymbol{z})$ and find their associated conditional vector fields $\boldsymbol{u}^{\mathrm{cond}}(\boldsymbol{x}, \boldsymbol{z}, t)$. We will now introduce the Conditional Flow Matching objective:

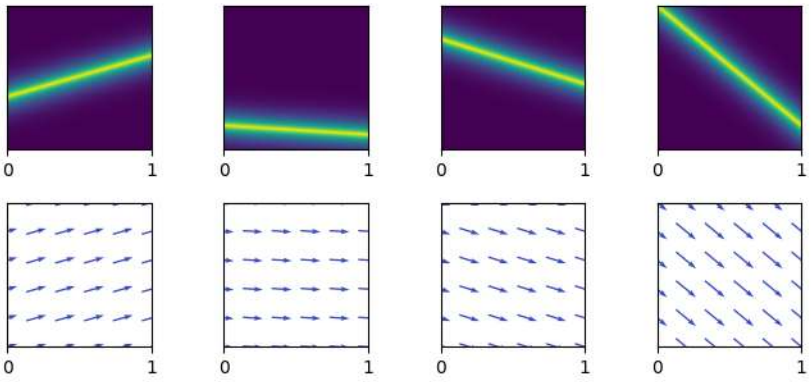


Figure 2.7.2: The conditional probability distribution when interpolating between $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, $p_t(\boldsymbol{x}|\boldsymbol{z} = (\boldsymbol{x}_0, \boldsymbol{x}_1)) = \mathcal{N}(\boldsymbol{x}|(1-t)\cdot\boldsymbol{x}_0 + t\cdot\boldsymbol{x}_1, \epsilon\cdot\boldsymbol{I}_n)$. Below is the corresponding conditional vector fields $\boldsymbol{u}_t^{\text{cond}}(\boldsymbol{x}) = \boldsymbol{x}_1 - \boldsymbol{x}_0$.

$$\mathcal{L}_{\text{CFM}}(\boldsymbol{\theta}) := \mathbb{E}_{t\sim\mathcal{U}(0,1),\boldsymbol{z}\sim q(\boldsymbol{z}),\boldsymbol{x}\sim p_t(\boldsymbol{x}|\boldsymbol{z})}||v_{\boldsymbol{\theta}}(\boldsymbol{x},t) - \boldsymbol{u}^{\text{cond}}(\boldsymbol{x},\boldsymbol{z},t)||^2 \tag{2.22}$$

As opposed to the flow matching objective seen in Equation 2.18, this one is easy to compute. By choice, we know and can sample from $\boldsymbol{z}\sim q(\boldsymbol{z})$ and $\boldsymbol{x}\sim p_t(\boldsymbol{x}|\boldsymbol{z})$. From Theorem 2.7.2.1, we also know the closed form of the conditional vector field $\boldsymbol{u}^{\text{cond}}$. The key finding in [21] is that optimizing the conditional flow matching objective, yields the same results as optimizing the flow matching objective (see Equation 2.18).

Theorem 2.7.3.1. (Theorem 2 of [21]). *Assume that* $\forall\boldsymbol{x}\in\mathbb{R}^n, p_t(\boldsymbol{x}) > 0$ *and* $t\in[0,1]$. *That is, no point has a zero probability. Then,* $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\text{FM}}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}\mathcal{L}_{\text{CFM}}(\boldsymbol{\theta})$.

2.7.3.1 Intuition for the Conditional Vector Field

While a rigorous proof for why Theorem 2.7.3.1 is true can be found in [21], we will now try to build some intuition for why it is true. Every single conditional probability path $p_t(\boldsymbol{x}|\boldsymbol{z})$ has a corresponding conditional vector field, as can be seen in Figure 2.7.3.

As we are taking the expectation over both $\boldsymbol{z}\sim q(\boldsymbol{z})$ and $\boldsymbol{x}\sim p_t(\boldsymbol{x}|\boldsymbol{z})$, the vector field we are regressing against is a weighted average of all these simple vector fields, weighted using each vector fields corresponding probability path (see Figure 2.7.4). The key insight is that we can model complex transforms between distributions by aggregating many simple point-to-point transforms.

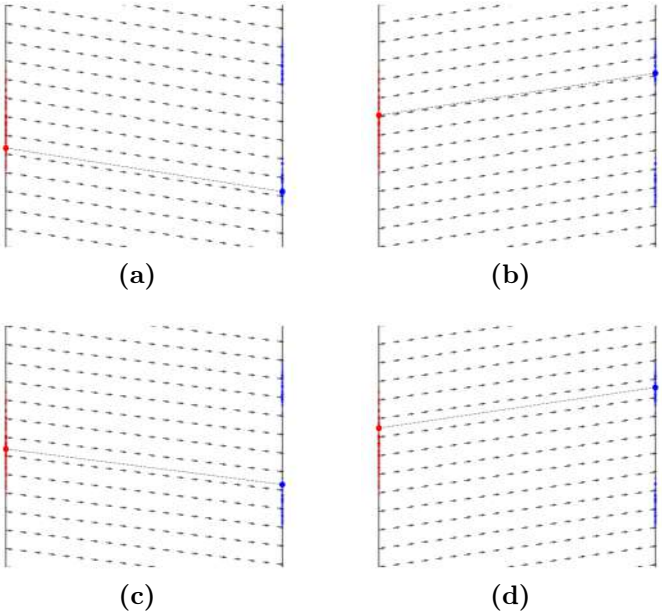


Figure 2.7.3: Four random pairs from the dataset. The corresponding uniform vector field is drawn in the back. The line draw between them can be thought of as the probability path over time. By sampling enough point-pairs, and aggregating the vector fields we can approximate the unconditional vector field. Note that we add the different vector fields weighted by their respective probabilities.

And when the distribution-to-distribution transformation is learned we can easily sample from it and transform the sample into the target distribution.

2.7.4 Sampling from the Data Distribution

To generate new samples from the approximated data distribution $\boldsymbol{x}_1 \sim p_1$, we first obtain a sample from $\boldsymbol{x}_0 \sim p_0$. We can then transform it using the Equation 2.17

$$\mathrm{d}\boldsymbol{x} = \boldsymbol{u}(t, \boldsymbol{x})\mathrm{d}t$$

Since we have access to $\boldsymbol{x}_0$ and $\boldsymbol{v}_{\boldsymbol{\theta}} \approx \boldsymbol{u}$, we solve it numerically, using some integration scheme to arrive at an approximate sample from p_1. One simple example is to use the Euler scheme, where $\Delta t > 0$ is some sufficiently small number

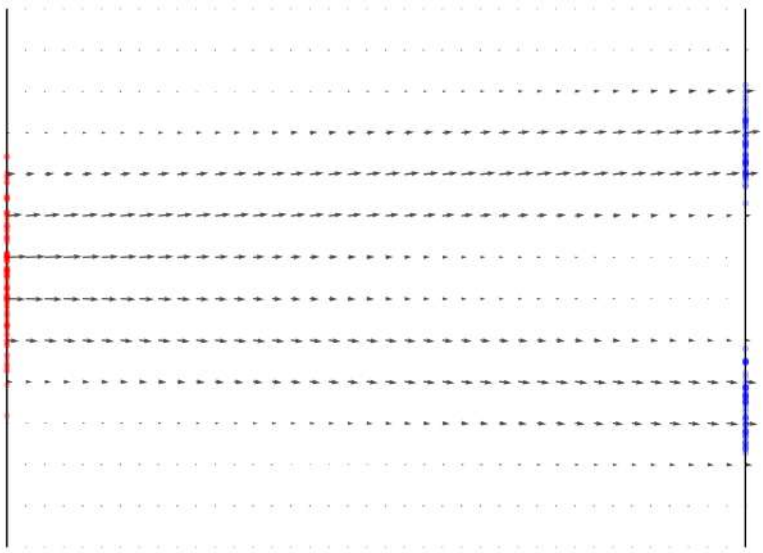


Figure 2.7.4: The aggregated vector field after sampling 100 random pairs.

$$\boldsymbol{x}_{t+\Delta t} = \boldsymbol{x}_t + \Delta t \cdot \boldsymbol{v}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$$

In reality, there exists more sophisticated integrations schemes, which are both more efficient and accurate. Roughly, we can divide them into fixed step size solvers and adaptive step size solver. Fixed step size solver include the Euler scheme and Runge-Kutta-4 (RK4). The 4 refers to the order of the RK method. For a step size h, a Runge Kutta-method of order n has an error of $\mathcal{O}(h^n)$. With an adaptive step size solver, we do not specify the step size. Instead, we define an upper bound for the absolute and relative error we tolerate at each step. The solvers then determines a step size as large as possible, where it can still ensure that the error is below the thresholds. A higher error tolerance leads to larger steps and less computation, while a smaller tolerance leads to smaller steps and more computation needed. DOPRI5 is a commonly used adaptive step size solver.

2.8 Autoregressive Models

Autoregressive models work by taking a sequence of information and predicting the next element in the sequence. A common use-case is for language generation, where we wish to predict the next word given the previous words. We often call the elements of the sequence tokens. They can

represent words, letters, patches of images or something else entirely. Given a sequence of tokens $t_1, \ldots, t_n$, the goal is to maximize the probability of predicting token t_i when observing the sequence of token $t_1, \ldots, t_{i-1}$. If p is a probability distribution over all possible token (the vocabulary), then the goal is:

$$\max \mathcal{L} = \sum_{i=1}^{n} \log p(t_i | t_1, \ldots, t_{i-1})$$

Given that we have access to $p(t_i|t_1, \ldots, t_{i-1})$, generating a new sequence is easy. For as many tokens as we need, we simply sample from the distribution, using in the previously generated tokens as the conditioning. The probability distribution is parametrized as a neural network with parameters $\boldsymbol{\theta}$. We then have that

$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(t_i | t_1, \ldots, t_{i-1})$$

2.8.1 Autoregressive Image Generation

While diffusion models and other flow-based models have been the State-of-the-Art for image generation since 2021 [25], there have also been attempts to train autoregressive models for image generation. An early example was the PixelRNN (and later PixelCNN) developed by researchers from Google Deepmind in 2016 [26, 27]. In fact, the first DALL-E model from OpenAI was also autoregressive [28], but as we know diffusion models would soon take over.

Since autoregressive models rely on generating a sequence, we need to determine the order we want to generate the image. PixelRNN does this by generating pixels left to right, top to bottom. When generating a pixel, they use the models hidden state, which is based on previously generated pixels. This hidden state is highly dependent on the most recently generated pixels, but information from pixels generated a long time ago might disappear. On the other hand, flow models can see the entire image at the previous timestep when generating a new image, allowing us to capture relationship between pixels that are both spatially close (local patterns), but also spatially far from each other (global patterns).

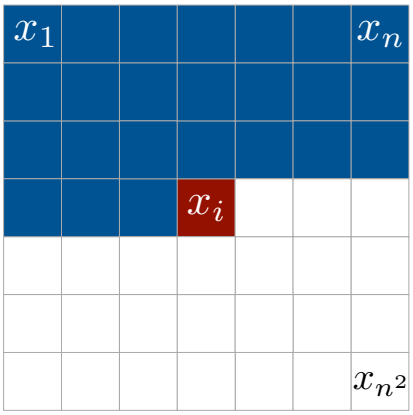


Figure 2.8.1: PixelRNN uses the previously generated pixels when generating a new one. Taken from Figure 2 in [26].

2.8.2 Visual Autoregressive Model (VAR)

The ability to capture both local and global patterns is something we would also want in autoregressive image generators. In 2024, the **Visual Autoregressive Model** (VAR) was introduced [29]. The authors claim that for the first time, autoregressive models have surpassed flow models [29]. While autoregressive image modeling has come a long way since PixelRNN, we will look at two key changes that make this model better suited for image generation.

2.8.2.1 Image Patches

While PixelRNN predicted individual pixels, this scales poorly to larger images. To increase efficiency, while maintaining performance, newer models use quantized autoencoders to create discrete values for patches, that are stored in a codebook [29, 30]. Instead of the autoregressive model predicting individual pixels, it predicts patches. When all patches are predicted, the decoder is used to convert the predicted patches back into an image.

2.8.2.2 Next-Scale Prediction

One of the key contributions in the VAR paper is order in which images are generated. Instead of going left to right, top to bottom, they perform what they name next-scale prediction [29]. They do this by predicting images of increasing scale. The first step is to train a multi-scale VQVAE. It learns to create accurate embeddings for images of different sizes. It is trained like a standard VQVAE, however, there are multiple encoders/decoders that work with images of different sizes. All these images use the same patch size, the number of pixels one embeddings represents. As such, the smallest encoder/decoder can work with images with 1×1 patches. The next scale is an image of 2×2 patches. We can progressively grow the image until we reach our final size. When we have a trained VQVAE, we can begin to train the transformer. The transformer learns to predict the next embedding given all previous embeddings. This allows it to increase the size of the generated image (represented by patch embeddings), while also being able to use the entire smaller-sized image as a reference when generating. In other words, the receptive field is the entire image during all stages of generation, mitigating a weakness in autoregressive models compared to flow models.

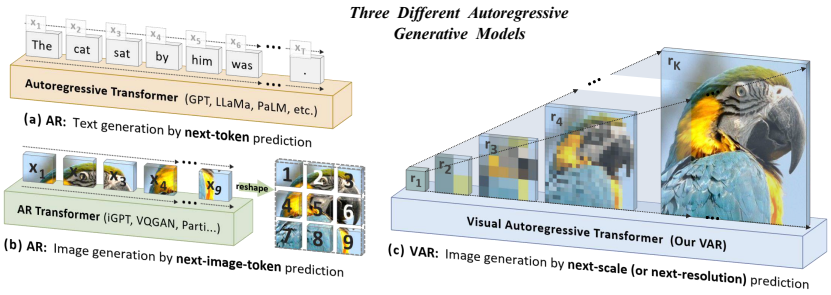


Figure 2.8.2: **(a)** AR model for language generation. **(b)** A traditional AR image generation, generating left to right, top to bottom. **(c)** Next-scale prediction, where the image is being predicted scale by scale. This figure is taken from Figure 2 in [29].

2.9 Comparing Flow Matching and AR Models

Autoregressive models are limited in two main areas: (1) the cost of producing images scales poorly. If doubling an image consisting of $n \times n$ patches, you need to generate another $4n^2$ patches, and (2) the autoregressive models require discrete tokens. When the image embeddings are discretised, information

Figure 2.9.1: This figure, taken from [33], demonstrate how using a VQ-VAE (discrete) encoder leads to poor reconstruction. The hybrid tokenizer referenced uses a VAE instead (continuous). We observe that the continuous autoencoder leads to better reconstruction. The ceiling for how good images you can generate, is therefore also higher.

is lost. This is due to the fact that the number of images possible is larger than the number of different embeddings. For images of size 512×512 pixels, and we represent each pixel by a red, blue and green integer ranging from 0 to 255, we have a total of $512 \cdot 512 \cdot 256 \cdot 3 \approx 2 \cdot 10^8$ possible images. The number of embeddings is given by the size of the codebook, and is usually on the order of 10^5 at most. The reason why good image generation models exist is that the subspace of *likely* images is way smaller than $2 \cdot 10^8$. This is known as the **manifold hypothesis**. It states that many high-dimensional datasets lie on a low-dimensional manifold. As such, we can describe the data distribution using fewer parameters than initally thought The manifold hypothesis predates deep learning [31], but has been used as an explanation to how models which learn data in very high-dimensional space are able to capture the distribution with relatively few parameters [32].

It is an inherent weakness that when we quantize embeddings we limit the expressivity of models (since all values are now discrete). Thus, when using a VQVAE and quantized embeddings, we set an upper limit on how good images the model can generate. This upper bound is the VQVAEs ability to reconstruct real images from their embeddings [33]. We show an example of this in Figure 2.9.1. FM models, on the other hand, operate with continuous values. While FM models are also bounded by the number of parameters and amount of data they have, they avoid the upper bound caused by discretization.

2.10 Measuring Generated Image Quality

In order to compare different image generation models, we need a way to measure the quality of the generated images. As the whole point of generative models is to (either explicitly or implicitly) model a data distribution, without knowing the true data distribution, it is not obvious how one would measure

the quality of generated images. Additionally, the quality of an image is not the only criteria we should measure when determining if we have learned a distribution. A variety of samples is also a key part of a good model. Arguably the most accurate way, is to provide humans with generated images and ask them to rate them (or to present two images, and ask which they prefer). However, this is resource intensive, and we want a way to measure this quantitatively. A widely adopted metric is the Fréchet Inception Distance (FID). Even though it is widely used, many papers criticize it. Since it builds on the Inception-V3 network from 2015 [34], critics claim that it is not able to richly represent the images generated by modern generative models, and also criticize it for only handling images of 299×299 pixels (images of other sizes need to be rescaled or cropped) [35].

2.10.1 Fréchet Inception Distance

Fréchet Inception Distance (FID) is a metric that builds upon the inception score [36]. We use a pre-trained image classifier, typically Inception-V3, to aid us. We use the output of the last pooling layer in Inception-V3 is used to extract features for each image, to get a 2048-dimensional feature vector. We assume that the feature vector belong to Gaussian distributions and calculate the mean and covariance matrices for the feature vectors of the real and generated images. We denote them $\boldsymbol{\mu}_r, \boldsymbol{\mu}_g$ and $\boldsymbol{\Sigma}_r, \boldsymbol{\Sigma}_g$. The FID is then given by

$$\text{FID}(\boldsymbol{\mu}_r, \boldsymbol{\mu}_g, \boldsymbol{\Sigma}_r, \boldsymbol{\Sigma}_g) = ||\boldsymbol{\mu}_r - \boldsymbol{\mu}_g||^2 + \text{Tr}\left(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g + 2\left(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g\right)^{1/2}\right) \quad (2.23)$$

A low FID is better, as it means the 2-Wasserstein distance between the two distributions $\mathcal{N}(\boldsymbol{\mu}_r, \boldsymbol{\Sigma}_r)$ and $\mathcal{N}(\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g)$ is small. While it might be hard to interpret what a score means, it has been shown that a better FID score often correlates with images that are preferred by humans [36]. As such, it is hard to compare models trained on different tasks (as different datasets might vary in what FID score is considered good), but the ranking it provides often coincides with the ranking made by human judges (although there are also examples of FID disagreeing with humans [35]).

2.11 Measuring the Efficiency of a Model

We can measure the efficiency of models (model here refers to a neural network, that can be treated as a learnable function) in a number of ways. First, it is important to distinguish between resources used during training, C_T, and resources used during inference, C_I. The cost of training a model only needs to be paid once. On the other hand, the number of inferences can greatly vary between different models. If the model makes N number of inferences, the amortized inference cost is given by

$$C_A = C_I + \frac{C_T}{N}$$

As N grows, the cost of training becomes a smaller part of the amortized inference cost. However, as $C_T >> C_I$, both metrics are useful to measure.

There are several different metrics by which we can measure the resources used by a model. We will now list different metrics, explain how and what they measure as well as briefly touch upon what their strengths and weaknesses are.

of GPU hours: the total number of GPU hours used to train the model, or the number of inferences per GPU hour are both metrics which are commonly reported. They are easy to measure and give a rough indication of the resources needed. However, this metric also comes with weaknesses. Firstly, it is hard to compare models across different GPUs. As such, unless two different models were trained on the same GPU, it can be hard to interpret the result. Furthermore, they do not take into account how well written the underlying code is. Optimizing the code can be difficult and time consuming (and also hard to measure), and it would therefore be nice to have a metric without that bias. However, due to the simplicity of calculating this metric, it is a good baseline for rating the efficiency of a model.

of parameters: the number of parameters, like the GPU hours, is easy to calculate and therefore commonly reported. It gives an indication of how complex patterns the model can learn. This is often (but not necessarily) related to how much data it needs, and how expensive it is to train, as well as how expensive inference is [37]. It is also correlated to the memory requirements.

of FLOPS: the number of Floating Points Operations (FLOPS) is another way of measuring the complexity of a model. Examples of FLOPS include addition and multiplication of floating point numbers (numbers with decimal

points). The number of FLOPS allow for comparison between different hardware. However, accurately counting FLOPS for neural networks can be challenging.

CHAPTER

THREE

METHODS

This section will introduce the method, dataset and models used to conduct the experiments in this thesis. We begin by introducing our method, before giving a quick summary of the work most related to the work done in this thesis. We then introduce ImageNet, a large-scale dataset commonly used for image generation benchmarking. We then introduce our two models, the Visual Autoregressive Model and the flow matching model, which are the two element we of our hybrid architecture. Finally, we introduce the custom dataset built specifically for finetuning our flow matching models.

3.1 Method

We propose a novel architecture for generating and enhancing images efficiently. Given images with $N \times N$ pixels, we will use a flow matching (FM) model to convert them into $M \times M$ pixel images, where $M > N$. Note that while this approach is compatible any backbone (and also with both real and generated images as the input). The reason we decide to use flow matching is two-fold. Firstly, FM models allow us to further adjust the trade-off between quality and speed post-training. We do this by adjusting the parameters of the ODE solver we use when generating the images. Secondly, FM models, unlike other similar architectures like diffusion models, are able to learn a posterior distribution starting at an arbitrary prior distribution. As such, it is perfect for learning the transformation between two distributions that we believe to be quite close, namely the distribution of $N \times N$ pixel images and the

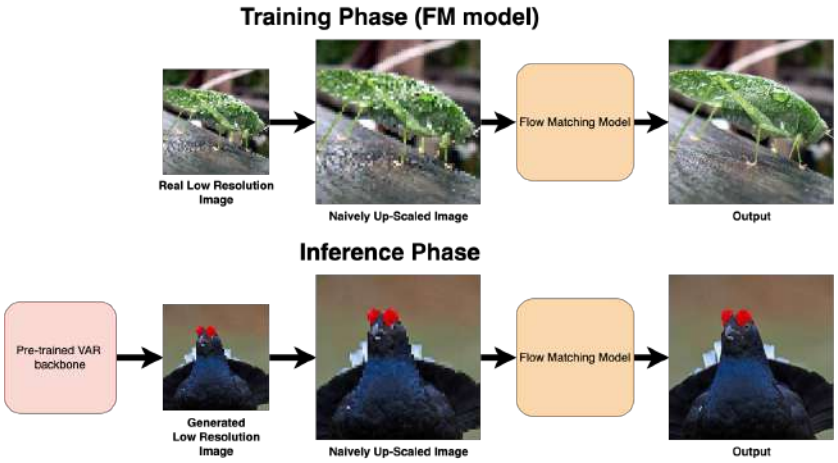


Figure 3.1.1: The proposed model during training and inference. **Top**: The training phase of the FM model. Real images in low resolution are fed as input, with their corresponding high resolution version being the target. **Bottom**: During inference, we use a pre-trained VAR model to generate a low resolution image. We naively up-scale it to our target resolution and use the trained FM model to get a high resolution image.

distribution of $M \times M$ pixel images.

More specifically, we will use 256×256 images generated by a VAR model and convert them into 512×512 images using the FM model. We will then compare the results against a larger VAR model that directly generates images of size 512×512. This task lies at the intersection between image generation and super-resolution. While the bulk of the "generation" is done by the VAR backbone, the FM model still needs to hallucinate likely details in the images, generating new information, while also performing more traditional super-resolution tasks working with the information already present in the image to convert it into a higher resolution. Picking the VAR as a backbone also allows us to investigate if the limitations introduced by quantized embeddings discussed in section 2.9 can be overcome using an FM model.

3.2 Related Work

While the previous chapter creates a thorough theoretical background to understand the work done in this thesis, this space is moving fast. This thesis builds on two key paper. They are *Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction* (VAR) from June 2024 [29]

and *Improving and generalizing flow-based generative models with minibatch optimal transport* (OTCFM) from February 2023 [22] . At the time of writing (May 2025), they have both been cited roughly 300 times each, with a lot of the work being very recent. We will therefore devote this section to briefly touch on the most relevant related work, some of which was done prior to this thesis and some of which has been done concurrently. This is meant as an overview of the landscape, rather than an in-depth review of the individual papers. Since most of the work is very recent, and the field is moving fast, it is likely not long before this is outdated. We therefore deemed it better to include this short summary, rather than try to write extensively about methods and papers which will soon be outdated.

Infinity, based on the VAR model, scales it further and adapts it to the text-to-image domain (where images are created not from class labels, but from natural language prompts) [38]. This outperformed SOTA diffusion models, while also generating images faster. We also find multiple other examples of scalable AR image generation models developed [30, 39]. There is also a lot of work related to bridging the gap between discrete and continuous embeddings, often by combining AR models with diffusion models [40, 41, 42, 33]. We find several approaches to combine traits from AR and flow models. ARFlow train a flow model to predict the latent representation of images with varying noise. It employs an autoregressive model to generate a noise of latents for images with varying degree of noise (starting from very noisy, and ending at completely denoised). During inference, the final latent representation is denoised using the flow model [43]. Somewhat related is [44], which proposes FlowAR. They use any pre-trained VAE (not VQVAE like in the VAR paper) to get a latent representation of an image. They then down-sample the image latents (either using a convolutional layer, or through some other non-learnable transformation, like taking the average) to the desired sizes. Like VAR, they learn to generate higher resolution latent representations using an autoregressive model. The latent representation for the highest resolution image is then used as conditioning for the FM model, which then learns a vector field transforming between pure noise and images. Li et al. train an autoregressive image generator without using vector quantization. Instead, they model the probability distribution of the continuous-valued latent variable $\boldsymbol{x}$, given the output of the AR model, $\boldsymbol{z}$, i.e. $p(\boldsymbol{x}|\boldsymbol{z})$. They do this using a diffusion model, and are able to sample $\boldsymbol{z}$, tokens for the AR model, through the reverse diffusion process [41]. Finally, HART introduces a hybrid tokenizer. They decompose the continuous latents from the autoencoder into a discrete part, modeled by an AR model, an a residual (or continuous part), modeled by a lightweight diffusion model [33].

For a large part of the related work, the motivation is often to mitigate the weakness of discrete tokens (and the information loss during quantization), while leveraging the power and scaling laws of autoregressive transformers (this was discussed in section 2.9). However, there are to our knowledge, no other methods that combine autoregressive models and flow matching models, *while* utilizing the fact that FM models can have an arbitrary prior distribution (i.e. not starting from noise). The idea is that this prior will mean that the transformation learned by the FM model is simple, requiring fewer parameters.

3.3 ImageNet

For this thesis, the dataset used is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012-2017 image classification and localization dataset. This datasets consists of more than 1.4 million images split among 1000 different classes. There are several reasons for using this dataset. Firstly, it is publicly available and easy to download. In using a public dataset, we hope to make the results presented verifiable. Secondly, this dataset is commonly used in image generation research. This means that much of the work that would be reasonable to compare to this work, already includes pre-trained models on the same dataset, easily allowing for a fair comparison. As this dataset human-annotated and quality-controlled, there is no need for further pre-processing of the data.

3.4 Visual Autoregressive Model (VAR)

We use the pre-trained VAR model from [29]. Both the code for running the model and model checkpoints are publicly available[1]. There exists different configurations for models that generate 256×256 pixel images, depending on how deep the neural network in the model is. The deeper a model is, the more parameters it has. As a result, they have the capacity to learn more complex transformations, and potentially generate images of higher quality, at the cost of being slower to run.

For the experiments, different VAR models will serve as the image generation backbone. They will generate images with 256×256 pixels.

[1]Both code for training and sampling from models, as well as model checkpoints are available at https://github.com/FoundationVision/VAR.

Table 3.4.1: The different models, their depth, the number of parameters they have and their FID score. The larger models have a lower FID score (lower is better), but at the expense of requiring more compute, both during training and inference. From [29].

Model Name	Depth	#param	FID ↓
VAR-D16	16	310M	3.55
VAR-D24	24	1.0B	2.33
VAR-D30	30	2.0B	1.97

3.5 Flow Matching Model for Image Enhancing

The flow matching models will take the output from the VAR, and convert it into 512×512 pixel images. It uses a UNet[2] and the conditional flow matching objective from Equation 2.22 to learn the vector field transforming our prior distribution into our posterior distribution. Ideally, the FM model needs to perform multiple tasks simultaneously. It would **(1)**: perform super resolution. The low-res image is naively up-scaled, and needs to be refined. **(2)**: perform generation. Going from a low-resolution to a high-resolution also entails generating likely details that would be present in a high-resolution image, but are not present in a low-resolution image. And finally, **(3)**: correct artifacts created by the VAR backbone. As the images coming out of the VAR model are not perfect (as reflected in the FID score), it is possible to get an even better FID score by correcting or reversing the artifacts created by the VAR backbone.

3.6 Building a Custom Dataset

Task (1) and (2) are learned using a dataset with low- and high-resolution image pairs. We use the ImageNet dataset, and scale the images to 256×256 and 512×512 pixels respectively, using LANCZOS resampling[3]. Since the FM model needs the input and output to be of the same dimension, we need to upscale the image. This is done either just by replicating each pixel 4 times, or by LANCZOS resampling. Therefore, when we say we use $256 - 512$ pixel image pairs, both images are in reality 512×512 pixels. However, 256 refers to the size the image was first down-sampled to, so information was

[2]The UNet implementation follows the implementation in [25]

[3]Lanczos resampling smootly interpolates between pixels using a custom kernel.

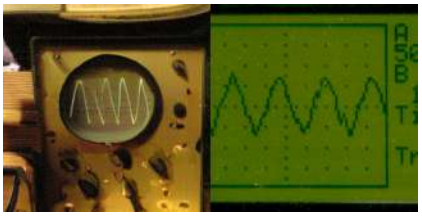
lost.

Figure 3.6.1: The image on the left is down-scaled to 64×64 pixels, and then up-scaled again. For the experiments, we use images down-scaled to 256×256 pixels and then up-scaled (middle) paired with images of 512×512 pixels (right). We include the 64×64 pixel image to better visually convey to the reader that information is indeed lost when doing this down-scale and up-scale operation.

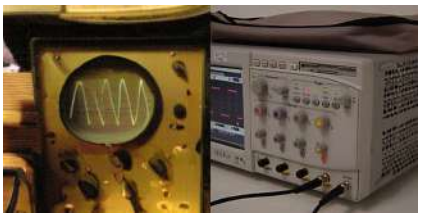
Task (3), however, cannot be solved using ImageNet images, as it relates to how we can correct for the errors made by the VAR model. To learn this, we need to pair output from the VAR model with images present in the ImageNet dataset. The way in which we do this is by sampling a batch of images (i.e. 32 or 64 images) from a given class using the VAR model. We then draw the same amount of images, from the same class, from the ImageNet dataset. These images are then mapped using optimal transport (minimizing the total distance between all pairs). The idea is that the images are similar enough to where learning the mapping is possible, while also being different, so that we can learn to correct the errors from the VAR model. We will pre-train the model on the ImageNet dataset, while finetuning on this dataset of VAR-ImageNet pairs. We create 200 images from each class for our custom dataset, resulting in 200k images. ImageNet has 1.2M images. Since we cannot solve the OT plan for this many images, and we randomly draw images from a randomly selected class, it might mean that some images appear more often than others, and that images can appear in pairs with a different partner.

(a)

(b)



(c)



(d)

Figure 3.6.2: Left column: The images-pairs before using optimal transport. **Right column**: The image-pairs after being paired using optimal transport. In some cases, like the pair in **b**, we see that the paired image resembles our starting image. In other cases, however, the results after optimal transport seem conceptually further apart from our starting image, like in case **d**. For each image pair, the left image is generated by VAR and the right is from ImageNet.

CHAPTER

FOUR

EXPERIMENTS & RESULTS

This section contains a plan for the experiments we conduct, with details for how the experimental setup looks. For each experiment, we provide a description of what was done, what the results are and discuss the results.

4.1 Experimental Plan

We will train several different models, and compare their quality and efficiency. We will systematically look at how changing different elements of the model impact performance. Unless stated otherwise, every model outputs images of 512×512 pixels. They consist of an autoregressive backbone from the VAR-family of models that outputs an image that is 256×256 pixels. This model is pre-trained, and not further trained during any experiments. The output of the VAR model is then converted to a 512×512 pixel image. In all cases, this starts by up-scaling it in a naive manner, since the FM model requires that the input and output are of the same dimensions. Naive up-scaling refers to up-scaling with no learnable parameters. In some experiments, we will further post-process this image using a flow matching (FM) model.

In Experiment 1, we will establish a baseline and answer **Research Question 4**. We will do this by naively up-scaling output from the VAR-D16 and VAR-D30 models. We will also look at whether using lossless (PNG) or lossy (JPEG) image compression plays a role. In Experiment 2, we will train and benchmark our FM models. They differ in what backbone they use, what

interpolation method is used to naively up-scale the images before passing them to the FM model, and what hyperparameters we use for the VAR model. Experiment 3 will benchmark models that are finetuned on our custom dataset detailed in section 3.6. Experiment 4 will investigate how the number of parameters of the FM model and the type of ODE solver we use impact the trade-off between efficiency and quality during inference. Experiment 2, 3 and 4 are all part of answering **Research Question 5**. For qualitative metrics, $\downarrow$ refers to metrics where a lower score is better. After each experiment, we include a short discussion of the results.

4.2 Experimental Setup

The models that use an autoregressive backbone with a flow matching model on top are denoted FM-X-DY, where the X denotes what naive up-scaling was used (LANCZOS, L, or NEAREST, N) and Y denotes the depth of the autoregressive backbone. They will also be compared to the VAR model that outputs 512×512 pixel images, VAR-D36. In all cases, we sample 50 000 images and calculate the FID by comparing our distribution against that of the ImageNet dataset[1]. For all metrics, the best result will be **bold** and the second best will be underlined. Hyperparameter for all models can be found in Appendix D. A complete implementation of the models can be found at the repository referenced in Appendix A. We train the FM models for n steps, where each step consists of training on one minibatch with b images. This results in the model training on a total of $n \cdot b$ images. When evaluating the model efficiency, we report three things. We report the number of parameters of the model, i.e. the number of tunable weights and biases in the underlying neural network. We also report the total number of GPU hours used to generate 50k images. Note that for our FM models, we report the total time spent including both the time the backbone spends generating images and the time spent by the FM model itself. We do the same when reporting the number of FLOPS.

4.3 Experiment 1 - Naive Baseline

In this experiment, we introduce a simple model that only uses naive up-scaling.

[1]Pre-computed statistics for ImageNet was downloaded from https://github.com/openai/guided-diffusion/blob/main/evaluations/README.md

Figure 4.3.1: Top Row: Interpolation using NEAREST. **Bottom row**: Interpolation using LANCZOS. The images are up-scaled to 256×256 pixels, from 8×8, 32×32 and 256×256 pixels respectively.

4.3.1 Experiment 1.1 - Naive Up-scaling

We will first establish a baseline. We do this by taking the output from the VAR-D30 model (which generates 256×256 pixels), and naively upscale it to 512×512 pixels. We then calculate the FID. We use two different schemes for naive up-scaling. NEAREST (N) simply repeats each pixel 4 times. LANCZOS (L) uses a kernel to smoothly interpolate values. An example can be seen in Figure 4.3.1.

Table 4.3.1: We establish a baseline by naively upscaling the images using both NEAREST (repeating pixels) and LANCZOS (interpolation with a kernel).

Model Name	Naive Upscaling	FID ↓
VAR-D30	NEAREST	60.79
VAR-D30	LANCZOS	<u>51.56</u>
VAR-D36	-	**2.63**

Regardless of the up-scaling used, it is clear that this is far from the performance of VAR-D36.

4.3.2 Experiment 1.2 - Image Format

As [29] reports that saving the images as the file format might impact the FID calculations, we calculate for both PNG and JPEG images.

Table 4.3.2: We differentiate the models by what method we use to naively upscale the images and the image format we use to save and benchmark our samples.

Model Name	Naive Upscaling	Image Format	FID ↓
VAR-D30	NEAREST	PNG	60.79
VAR-D30	NEAREST	JPEG	60.84
VAR-D30	LANCZOS	PNG	<u>51.50</u>
VAR-D30	LANCZOS	JPEG	51.56
VAR-D36	-	-	**2.63**

As there is a negligible difference between the two formats, we opt to use PNG for the remaining experiments, unless stated otherwise. This is because the authors of the VAR paper do this [29].

4.3.3 Discussion

In Experiment 1, we establish a baseline. We did this by naively up-scaling the images and calculating the FID. Since FID uses the Inception-V3 network, it can only handle images of 299×299 pixels. Since our generated images are larger, they are resized down to this size. It was therefore surprising to see that only using naive up-scaling produced very poor FID scores. It is evident that additional refinement of the images is required. While this approach evidently did not work, we believe that to a human evaluator, the difference between this approach and the later flow matching models would not be as stark as the high FID score indicates.

4.4 Experiment 2 - Base Flow Matching Model

In this experiment, we will look at implementing the base flow matching models and tuning the hyperparameters to maximize the performance. Since the models are resource intensive, we decide to focus on three main aspects. Experiment 2.1 looks at what backbone we use, and what way we upscale

the images. In Experiment 2.2, we look at the strength of the classifier free guidance for the VAR model, i.e. how strongly the generation process has to adhere to the class label.

4.4.1 Experiment 2.1 - Base Models

To investigate if the FM models benefit from a good backbone, we will compare models that us the output from the VAR-D16 and VAR-D30 [29]. We train a flow matching model with optimal transport following [22]. The model uses a UNet backbone, following the implementation used in [25]. For the flow matching model, we use a DOPRI-5 ODE solver with an absolute and relative tolerance of 1e-5. All models were trained with a batch size of 8 for 400 000 steps (meaning they trained on a total of $400\text{k} \cdot 8 = 3.2\text{M}$ images.

Table 4.4.1: The table describing our four different models

Model Name	Naive Upscaling	VAR Backbone	FID ↓	GPU Hours
FM-N-D16	NEAREST	VAR-D16	10.08	<u>6.7</u>
FM-L-D16	LANCZOS	VAR-D16	10.06	<u>6.7</u>
FM-N-D30	NEAREST	VAR-D30	**2.42**	7.3
FM-L-D30	LANCZOS	VAR-D30	<u>2.45</u>	7.3
VAR-D36	-	-	2.63	**3.0**

The flow matching model performs better than VAR-D36, but only if using a capable backbone. The method used for naive up-scaling does not seem particularly important. For the remaining experiments, we therefore use NEAREST as this is the simpler of the two. However, the VAR-D36 is still more than twice as fast.

4.4.2 Experiment 2.2 - Classifier Free Guidance

The authors of the VAR paper report different values for classifier free guidance (CFG). In this experiment, we vary the classifier free guidance to see if it is important.

It is clear that using CFG=1.5 improved the performance of the models. The smaller VAR-D16 based models still are not able to beat the large VAR-D36 model, but we are able to achieve a significantly lower FID score using a VAR-D30 model and a flow matching model on top. An interesting thing we

Table 4.4.2: The table describing our four different models

Model Name	Naive Upscaling	CFG	VAR Backbone	FID ↓	GPU Hours
FM-N-D16	NEAREST	4.0	VAR-D16	10.08	6.7
FM-N-D16-V2	NEAREST	1.5	VAR-D16	3.36	6.7
FM-N-D30	NEAREST	4.0	VAR-D30	2.42	7.3
FM-N-D30-V2	NEAREST	1.5	VAR-D30	**1.87**	7.3
VAR-D36	-	-	2.63	**3.0**	

observed during this experiment, is how sensitive the VAR backbones are to different hyperparameter. When trying to reproduce the results from [29], we observed that while both models benefited from saving the images as PNGs and tuning the classifier-free guidance (CFG) lower, the difference was more apparent in the smaller VAR-D16 model (see Table 4.4.3).

Table 4.4.3: The table shows the FID of our different backbones when trying to reproduce the original paper [29]. The models marked with '*' represent the models from the original paper. Using CFG=1.5 and saving the images as PNGs, we are able to reproduce the original FID scores.

Model Name	Save Format	CFG	FID ↓
VAR-D16	PNG	1.5	3.55
VAR-D16	JPEG	4.0	10.43
VAR-D16*	-	-	3.55
VAR-D30	PNG	1.5	1.99
VAR-D30	JPEG	4.0	2.49
VAR-D30*	-	-	1.97

4.4.3 Discussion

The flow matching models did outperform the larger VAR-D36 model, improving the FID score. While they did require more inference compute, they required very little compute during training. While this thesis was more concerned with the inference compute, it is worth noting that both the training and validation loss stabilized after training on roughly 160k images (20 000 steps with 8 images in each batch). On an NVIDIA A100 GPU, this required running for less than 3 hours.

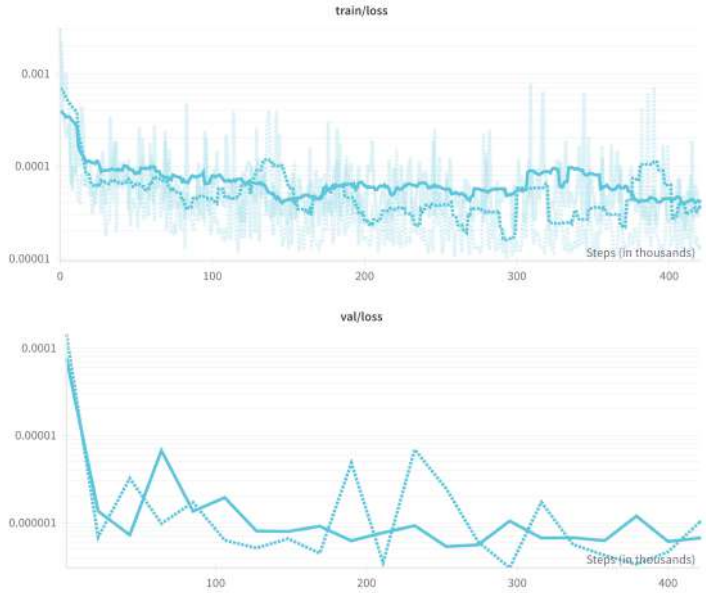


Figure 4.4.1: Top: The running average of the training loss for two training runs with the flow matching model. **Bottom**: The validation loss during the same runs. The x-axis shows the number of steps (in thousands), and the y-axis shows the logarithm of the loss. Both the training and validation loss stabilize after close to 20 000 steps.

We attribute this quick convergence to the fact that the starting distribution, naively up-scaled low-resolution images, and the target distribution, the corresponding high-resolution images, are not that far apart. The ability to model the transformation between any two distributions we have access to samples from, is one of the key advantages of flow matching. In our case, it is made even easier since we have access to perfectly paired samples from the two distributions. Recall from section 2.7.2 that empirical studies have shown that solving the minibatch optimal transport plan improves the performance of FM models. Since we essentially have solved the optimal transport plan for our whole dataset (of 1.2 million images), we can likely attribute the fast convergence and good performance to this. We believe that the reason what method we use to naively up-scale the images is insignificant, is due to the fact that the parameters used to naively up-scale images is way less than the parameters of the FM model, and thus insignificant (the exception being Experiment 1, where we did not have an FM model).

Table 4.4.4: The table shows the absolute and relative increase in FID score, compared to the backbone. Note that the backbone FID is calculated using a reference batch from ImageNet-256 while the FM models are compared to ImageNet-512.

Model Name	FID ↓	Backbone FID ↓	Abs. Increase	Rel. Increase
FM-N-D16	10.08	3.55	+6.53	+184%
FM-N-D16-V2	3.36	3.55	-0.19	-5%
FM-N-D30	2.42	1.99	+0.43	+22%
FM-N-D30-V2	1.87	1.99	-0.12	-6%
VAR-D36	2.63	-	-	-

It is important to keep in mind that generating higher resolution images is inherently a harder task, as the number of possible (and likely) images grows with the resolution. As such, one might think that our FM models generated higher quality images than their backbone (as they got a lower score on a harder task). However, it is not trivial to compare FID scores across different sizes (a more complete discussion of FID as a metric for image quality can be found in section 5.2). What we can say, however, is that our FM models using VAR-D30 as a backbone outperformed the VAR-D36 model.

These findings demonstrate empirically how the backbone directly influences how well our FM model performs. In both cases, the FID score was roughly 5% lower when using the optimal hyperparameters for the VAR model. However, it also demonstrates that given a good backbone, the model is able to outperform

the VAR-D36 model on ImageNet-512. Since the combination of using CFG=1.5 and saving images as PNGs was critical to reproduce the original results (likely due to the correct value for CFG), we will use this configuration for Experiment 4 (Experiment 3 was already underway, so it will use the old configuration of JPEG and CFG=4.0). One interesting thing to note is that FM-N-D16 has a significantly higher FID score than its backbone, while this is not the case for the other 3 models. We hypothesize that if the starting distribution of images is too far away from the distribution we have trained on, the distribution of real images, the FM model performs significantly worse. Conversely, we believe that the better the backbone we use is, the better our FM model will perform since the inference data distribution is closer to the training data distribution.

4.5 Experiment 3 - Finetuning

Regardless of which backbone we use, the VAR models produce some artifacts in the images. As the FID is larger than zero, there is a distance between the distributions of the generated and real images. In addition to learning super-resolution and generation, it would therefore be beneficial if the FM model could learn to reduce the inherent distance between distributions of our generated and real images. We therefore try to finetune our models trained in Experiment 2 by using our custom dataset detailed in section 3.6. The models were trained for 10 000 steps with a batch size of 16, meaning they saw a total of 160k images from out custom dataset (note that it might not be 160k unique images, as they are sampled randomly). We let the models run for up to 100k steps, but there was a visible degradation in the quality of the generated images as the models continued to train.

This did not lead to better FID scores. On the contrary, the models performed way worse. An example of the output images can be seen in Figure 4.5.1.

4.5.1 Discussion

In Experiment 3 we wanted to finetune the models pairing generated images from the backbones naively up-scaled with real images from ImageNet. We hoped that the FM models would become more robust when encountering images that were far from the distribution seen during pre-training, and learn to correct artifacts generated by the VAR backbone.

This experiment resulted in models that had a FID score close to 30 times

Table 4.5.1: This table shows the FID of the models we finetuned. We differentiate by how we up-scaled the output from the VAR backbone, and which backbone was used. We also report the difference between the finetuned models and their non-finetuned counterpart.

Model Name	Naive Upscaling	VAR Backbone	FID ↓	ΔFID ↓
FM-N-D16-FT	Nearest	VAR-D16	74.10	+Δ64.02
FM-L-D16-FT	LANCZOS	VAR-D16	<u>25.26</u>	+Δ15.20
FM-N-D30-FT	Nearest	VAR-D30	76.46	+Δ74.04
FM-L-D30-FT	LANCZOS	VAR-D30	71.72	+Δ69.27
VAR-D36	-	-	**2.63**	-

Figure 4.5.1: The output from FM-L-D30-FT. As reflected by the high FID score, the images seem to be of very poor quality.

as high as their corresponding base model. We believe that there are several ways to improve this experiment. Since we first sampled a random class index, and then a minibatch of both real and generated images from that class, the same generated image might be paired up with different real images. The model might then learn to output some average over all the images in the target class, and we observed in training that the longer we let the model train, the more blurry and unrecognizable the images became.

We attribute the (relatively) low FID for FM-L-D16-FT to the model being slower to fall into this pattern of producing the average output, but after training for more steps, we observed an identical effect. One possible way

!htb 

Figure 4.5.2: Top: Even after 1000 steps, the images look corrupted. **Middle**: After 10k steps, it is even harder to recognize what the images are meant to be. This is the time we let the model train before trying to sample images with it. **Bottom**: After 45k steps, the model output is very corrupted. All images are from FM-L-D30-FT, but the effect was observed for all models

to avoid this would be to create a static dataset. Another approach is to use the ImageNet pairs of low and high resolution, but distort the low-resolution images. However, one would need to analyze what artifacts the VAR backbones create and try to imitate them during training. For a more thorough discussion of how to improve this experiment, we refer to section 5.1

4.6 Experiment 4 - Improving Efficiency

There are two way to make the model more efficient. Either by using a model with fewer parameters or by relaxing the constraints of the ODE solver (reducing the number of steps, or increasing the accepted error tolerance). In this experiment, we will explore both options.

4.6.1 Experiment 4.1 - Lightweight UNet

In this experiment, we reduce the number of parameters of the UNet and see how it affect the quality of the generated images. We compare it against the

FM models with a full-sized UNet.

Table 4.6.1: We compare the full-sized models with their lightweight (LW) counterparts. We reduce the number of parameters by a factor of 5.2. We measure the models FID score and the GPU hours required on an A100 to generate 50k images.

Model Name	# of Parameters	Training Steps	FID ↓	GPU Hours ↓
FM-N-D16-V2	120M	400k	3.36	6.7
FM-N-D16-LW	23M	100k	10.00	<u>4.5</u>
FM-N-D30-V2	120M	400k	<u>1.87</u>	7.3
FM-N-D30-LW	23M	100k	**2.34**	5.1
VAR-D36	2.3B	-	2.63	**3.0**

The smaller sized model performed worse than the larger scale UNet. However, we are able to reduce the train time to only 100k steps. We are also able to reduce the inference time. The fastest FM model that outperforms VAR-D36 is now only using 70% more time to generate images.

4.6.2 Experiment 4.2 - Relaxing the ODE Solver

In this experiment, we use the same model (FM-N-D30-LW), but relax the constraints of the ODE solver progressively. This means that we allow a less accurate solution to the ODE, but that the calculations will be faster. We compare both ODE solver with adaptive step sizes and fixed step sizes. Solver with a fixed step size take exactly as many steps as we specify (and in our case, each step is the same length). Adaptive step size solver, however, are initialized with tolerance levels. They take the largest step they deem possible, while ensuring that the error does not exceed the tolerance level. In our implementation, we are able to differentiate between absolute and relative error. For all experiments, we use the FM-N-D30-LW model from last Experiment 4.1. The adaptive step size solvers are in Table 4.6.2 while the fixed step size solvers are in Table 4.6.3. Finally, a comparison between all solvers, looking at both quality and performance can be found in Table 4.6.4

4.6.3 Discussion

In this experiment, we were looking at ways to improve the efficiency of models. Like we initially thought, there is a trade-off between efficiency and

Table 4.6.2: We compare the different adaptive step size ODE solvers, changing their absolute and relative error tolerance. We write down the FID score and the GPU hours required on an A100 to generate 50k images. The tolerance level does not seem to matter.

Name	Abs. Tol	Rel. Tol	FID ↓	GPU Hours ↓
DOPRI-5-LOW	1e-5	1e-5	<u>2.34</u>	7.3
DOPRI-5-MEDIUM	1e-3	1e-3	**2.33**	5.1
DOPRI-5-HIGH	1e-1	1e-1	**2.33**	<u>4.8</u>
VAR-D36	-	-	2.63	**3.0**

Table 4.6.3: We compare the different fixed step size ODE solvers. Interestingly, the type of ODE solver and the precision does not seem to matter. For the fixed step size solver, we also write down the number of function evaluations (NFEs) and the GPU hours required on an A100 to generate 50k images.

Name	Step Size	Max NFEs	FID ↓	GPU Hours ↓
RUNGE-KUTTA-4	3	12	<u>2.33</u>	4.5
EULER	3	3	**2.32**	<u>3.3</u>
VAR-D36	-	-	2.63	**3.0**

performance. We found that the smaller models trained and generated images faster, but at a cost to the FID score.

With the ODE solvers, it does not seem to matter how complex the solver is. They all get essentially the same FID score (the small difference of 0.02 is not enough to justify any conclusions about improved performance). However, the fastest solver reduces the time spent generating images to 3.3 hours on an A100 for 50k images. Compared to the slowest solver, DOPRI5 with a low tolerance, it is more than twice as fast. We suspect that this is because the vector field we have learned approximates a linear vector field. In other words, a line between any corresponding points in our start and end distribution is straight. This is because using pairs the same image in low and high resolution, we essentially have already solved optimal transport for ImageNet.

While the FM model is still slower than the VAR-D36 model, we believe that this is not due to the architecture of the network or model, but rather due to other parts of the code like transferring data, saving data, and the

Table 4.6.4: We compare all ODE solvers looking at both the FID score and the GPU hours needed to sample 50k images (on an A100). While this is not necessarily a good way to compare across different architectures and hardware, it provides an accurate ranking (and relative difference).

Name	FID ↓	GPU Hours
DOPRI-5-LOW	2.34	7.3
DOPRI-5-MEDIUM	<u>2.33</u>	5.1
DOPRI-5-HIGH	<u>2.33</u>	4.8
RK-4	<u>2.33</u>	<u>4.5</u>
EULER	2.32	**3.3**

implementation of the ODE solver. In Table 4.6.5, you can see an estimate of how many floating point operations are required to generate one image. A lot more profiling and optimization of the code would be required to identify and remove the bottleneck, which is outside the scope of this thesis. However, the number of FLOPS does provide insight into the theoretical improvements in inference speed, given optimized code.

Table 4.6.5: We calcaulte the TFLOPS (Terra FLOPS) used to generate one image for our FM models (using the EULER solver with 3 steps) and VAR-D36. We do this by sampling 16 images and using a FLOPS profiler to count the average number of FLOPS per image. For the FM models, we add both the FLOPS for the baseline as well as the FLOPS for solving one step of the ODE multiplied by the number of steps.

Model Name	ODE Solver	TFLOPS
FM-N-D30-V2	EULER	<u>4.57</u>
FM-N-D30-LW	EULER	**2.74**
VAR-D36	-	11.11

CHAPTER

FIVE

DISCUSSION

While the previous chapter included concise discussions related to individual experiments, we will use this chapter to discuss broader observations from our experiments. We will begin by problematizing Experiment 3, and look at why the finetuning did not work and what we can do to improve it. We will then move on to talk about the weaknesses of the FID metric, and why even though it is the gold standard for generative image models, it is important to understand that it might differ from human preference. Finally, we will discuss questions related to the ethical considerations of these models, and discuss why AI can be both a hindrance and a solution as we continue to strive for sustainable development and a sustainable future.

5.1 Why Experiment 3 Failed & How to Finetune Better

The results from Experiment 3, finetuning, were bad, and so we devote this section to explaining *why* we think they ended up being so bad and *how* to improve the experiment. Due to time and resource constraints, we are not able to run and benchmark the improved experiments thoroughly. We will begin by talking about the way the dataset was created.

5.1.1 Issues with the Dataset

We believe that the results can be partially explained by looking at how the dataset these models were trained on was constructed (see section 3.6. We sampled 200k images from our VAR backbone and paired them up batch by batch, using optimal transport. Due to memory constraints, the model is only able to handle a batch size of 16 images. We therefore (naively) used the same batch size to perform optimal transport when pairing images. The small batch size, along with the completely random sampling from the dataset (they were not ordered, like datasets often are), meant that there was a high likelihood of the same generated image being paired with different real images (and vice versa). Instead of the very clear 1-to-1 relation between input and output in Experiment 2, we now have a many-to-many relation. This could be causing the model to learn an average of the possible output-mappings for each input-mapping. Since ImageNet has a high intra-class diversity (images from the same class can be very different). An example of a good 1-to-1 mapping can be seen in Figure 5.1.1. An illustrative example showing how the intra-class diversity might make it difficult for the FM model during the fine-tuning process is shown in Figure 5.1.2. When a generated image is paired up with multiple images, and especially if the images are not similar, the model might struggle to learn meaningful patterns.

However, we did run one experiment with a static dataset where we solve the optimal transport plan for 200 images from each class, one class at a time. Unlike the dataset used in Experiment 3, we solved the optimal transport plan prior to training the model. Images generated during different stages of training can be seen in Figure 5.1.3, and we see that the same problem arises. However, since we were not able to explore the effect of changing different hyperparameters and even larger batch sizes, we are hesitant to say that this was the sole, or even main, issue.

5.1.2 A Better Way to Find Similar Images

To create the optimal transport plan between the generated and real images, we use the 2-Wasserstein distance in pixel space. However, there are several weaknesses using this approach. Since we are operating in pixel space, the model is sensitive to small changes. For instance, the distance between an image, and the same image flipped or slightly rotated can be large. If the distance is large, transforming the flipped image back to the original is therefore also a difficult task for the FM model. However, human evaluators

Figure 5.1.1: The rows represent pairs of generated and real images. This is an example from one of the classes we have solved using a batch size of 200 (as opposed to 16 which we used in Experiment 3). Classes can have high intra-class diversity. By solving the optimal transport plan for a large batch size, we can reduce this effect.

would describe the images in a similar manner. A better way would therefore be to measure the distance between images using some latent representation. Here, there are many options. We could use the last pooling layer of the Inception-V3 network that we use to measure the FID. We also have access to the embeddings of images directly from the VQVAE of the VAR models. Finally, there are many other options that are commonly used to extract rich, semantic representations of images, like CLIP [45].

5.1.3 More Robust Pre-Training

A final improvement that could be made to help with finetuning, is making the pre-training more robust. During pre-training, we could apply augmentations to the images, like rotation, flipping and color shifting [46]. This is commonly done with image datasets to make them more diverse and the model more robust to seeing diverse images during the inference phase. However, this comes with some downsides. Since we are augmenting the images during training, the transformation we are learning is inherently more difficult. Additionally, we might lose some desirable qualities observed in our trained models. It is no longer necessarily true that we have solved the optimal

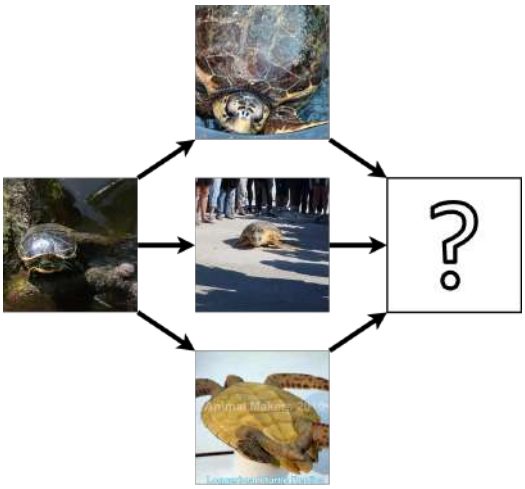


Figure 5.1.2: With a small batch size, there is no longer a 1-to-1 mapping between the VAR output and the real images from ImageNet. This might make it hard for the model to learn meaningful patterns. On the left we have a VAR generated image. In this scenario, the FM model sees this image appear with three different ImageNet images. They are all from the same class, but still quite different. The model therefore learns to output an average of the three images.

transport plan for the whole dataset and as such the vector field might not approximate a linear field (i.e. all paths are straight). This, in turn, would require using a more strict ODE solver and more compute during inference.

5.2 How to Evaluate Images

As the field of image generation is constantly evolving, it is necessary to have a way to compare different models efficiently. While the best way to do so is using human evaluators, this is both time consuming and expensive. The current standard for comparing models is the FID score (refer to section 2.10.1 to see how it works). While it often aligns with human preference, it is not always the case. We will highlight some potential issues with the FID score.

Firstly, FID is a biased estimate. It is dependent on the number of images you generate. This means that regardless of if you are modeling the distribution of horse images, or the distribution of all possible images, you need to generate the same number of samples as previous models have used. The standard is to generate 50k images. For simple tasks, this might feel exaggerated.

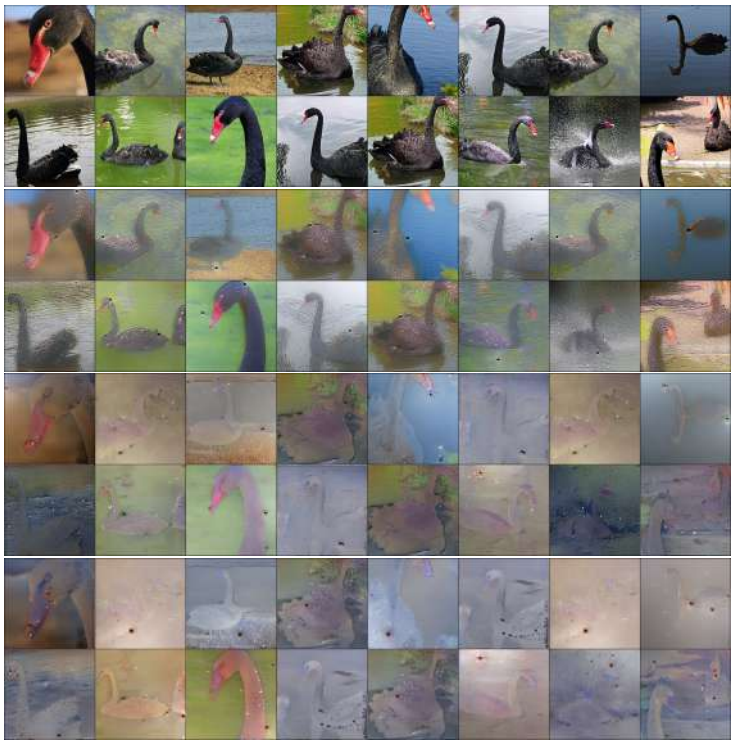


Figure 5.1.3: The same images after 5000, 30000, 55000 and 80000 steps of training. The output images grow progressively worse during the finetuning, even when using a static dataset.

For other tasks, like text-to-image generation, it might feel like you are not close to capturing the entire distribution. This can of course be mitigated by generating another set of images, evaluating them and averaging the result, but this introduces its own bias. FID assumes that the feature vectors of the images follow a Gaussian distribution. By generating several sets of samples and averaging the FID scores, we are allowing the distribution to be a mixture of Gaussians. It would also be beneficial during development to quickly get an indication of how well a model is doing, without needing to generate all 50k images (even if this estimate is less accurate).

FID also uses the Inception-V3 network to get the feature vectors for the images. This architecture has several issues. There exists larger models today, that are trained on more data that are likely able to capture more precise feature vectors. Additionally, the Inception-V3 can only handle images of 299×299 pixels. Larger images need to be resized or cropped. Today, where many models operate with images that are significantly larger, it might be difficult for the network to fully capture the features of the image as it is only

able to process the down-scaled version.

A final issue, which we also faced in this thesis, is how to compare the FID score of models producing images of different resolution. It is an open-ended question what a metric that is able to do this would look like. Should images that humans perceive as the same quality, across different resolutions get the same score? Or should an equal score between low and high resolution images indicate that the high resolution images are of a higher quality as the task of generating images is inherently harder the larger the image is?

5.3 Ethical Considerations & Sustainability

There are several ethical questions related to the research into and development of generative image models, and artificial intelligence in general. In this section we will touch upon how this model (and models like it) can be harmful, how one can mitigate the damage and how this is related to the Sustainable Development Goals (SDGs) presented by the UN. We will consider issues related to the theft of intellectual property, disinformation and the impact AI has had on global electricity consumption. But with so many downsides, we will start by reflecting on why we believe that AI research, done right, still has a net positive impact on the world. We will talk both about AI in the general sense, but also issues closely related to generative image models.

5.3.1 The Good: Why We Want to Research AI

When powerful AI is discussed in media, it is often in the context of the risks it might impose. And rightfully so. In the wrong hands, AI models can spread disinformation tailored to each individual, purposefully made to target the areas where they are the most susceptible. Combine that with the fact that the models we have now, are only going to get better and more powerful. We therefore want to take a step back and reflect on why we would even want to research AI when there are obvious and dangerous downsides. Since we only devote a short part of the thesis to this, we would like to state that for a well-written, in-depth essay on why the upsides of AI far outweigh the downsides, we will refer to Dario Amodei's *Machines of Loving Grace*[1].

The advances in AI has already had a tremendous positive impact on the world. In 2021, Google released AlphaFold and successfully determined the

[1]Available at https://www.darioamodei.com/essay/machines-of-loving-grace

structure of 100 000 proteins, a task that previously had taken human experts several years *per protein* [47]. It is not unreasonable to hope that advances in AI can aid discoveries in all scientific discipline, all the way from detecting and treating cancer to developing sustainable energy sources for the future. The spread of large language models like ChatGPT is a democratization of knowledge, distributing a powerful tool to people all over the world for free. And generative image models already impact the workday of designers and artists, making prototyping faster, allowing more people to be creative and express themselves.

5.3.2 The Bad: IP Theft and Disinformation

To train this relatively small-scale generative model, we needed a dataset consisting of 1.2 million images. While this might seem like a lot, it is easily dwarfed by the largest models available. A dataset commonly used is LAION-5B, a dataset containing more than 5 billion images. They are largely gathered by crawling the internet. There is an ongoing debate related to how the original owners of these images should be compensated, and if the way large models like DALL-E and Imagen are trained is even legal. Artists all over the worlds are upset seeing their artwork used to train a model that one day might take their jobs. This issue is related to SDG 13, which aims to "Promote peaceful and inclusive societies for sustainable development, provide access to justice for all and build effective, accountable and inclusive institutions at all levels". There is a clear need for strong, international institutions to combat potential theft of intellectual property. While there are already ongoing trials[2], there is yet to be a global cooperative effort to ensure that the companies behind these models develop them in a fair manner.

Another huge issue imposed by these models is disinformation. This is also related to SDG 16, and specifically target: "Ensure public access to information and protect fundamental freedoms, in accordance with national legislation and international agreements". It is important that content created by AI is somehow watermarked, ensuring that everyone can easily distinguish between what is real and what is generated. While there is an effort to ensure that AI content is watermarked, this technology is still not mature [48].

[2]See Andersen v. Stability AI et al

5.3.3 The Ugly: Energy Consumption and the Climate Crisis

As stated in the introduction, generating one image is roughly equivalent to charging your smartphone halfway [4]. The energy consumed by AI models already accounts for a sizable portion of the global energy consumption, and if current trends continue it will grow rapidly. While estimates vary, it is likely that data centers for AI consume as much energy as 22% of US households by 2028 [49]. However, companies are not keen to share details. Target 13.2 states that we need to "Integrate climate change measures into national policies, strategies and planning". This also applies to AI companies. To efficiently measure the impact this has on climate change, the first step is transparency. We also need more resources devoted to making both hardware and software more energy efficient. And finally, we need to start thinking of AI as a tool in the fight against climate change, and not an obstacle.

CHAPTER

SIX

CONCLUSIONS

This thesis explored how we can combine different architectures for image generation. More specifically, we used an autoregressive VAR model as a backbone to generate low resolution images. These where then up-scaled using a flow matching model. We compare the models, looking at both the image quality, measured by the FID score, and the model efficiency measured by the time spent generating each image, the number of floating-point operations and the total number of model parameters.

6.1 Conclusion

This section will summarize the findings for each individual research question, based on what we have learned from reading related works in the field and our own experiments.

Research Question 1: What are the State-of-the-Art architectures for image generation?

While diffusion models were long the State-of-the-Art models for image generation, recent research has revealed that autoregressive models, using next-scale prediction, can be equally good or better. At the same time, a new family of models named flow matching models have been introduced. They are a generalization of diffusion models. Critically, they are able to learn the transformation between any prior and posterior distribution.

Research Question 2: How can we measure the efficiency of models?

Measuring the efficiency of a model is not trivial. When comparing different models against each other, the results are dependent on what hardware is used, the efficiency of the code, how the code is compiled or interpreted and more. However, there are several approaches that can provide insight into the efficiency of a model. We chose to report the number of parameters, the total number of hours spent during inference on a specific GPU and the number of floating-point operations (FLOPS). Together, they provide an estimate for how efficient our models are compared to other generative image models.

Research Question 3: How can we measure the quality of generated images?

It is our opinion that this is still an open-ended question. The gold standard for quantitative metrics is still the FID score, although there have been research criticizing its validity. In general, the goal should be for the metric to align with the opinion of human evaluators while being cheaper and faster to compute.

Research Question 4: Is it possible to use traditional interpolation methods to up-scale generated images efficiently? This hypothesis was tested in Experiment 1. While this is an efficient and simple way to up-scale images, the results are not good. These traditional methods are not able to infer new information, but rather take the information already present in images to create the up-scaled image.

Research Question 5: Is it possible to combine different architectures to improve either efficiency or quality, without compromising the other?

We combined an autoregressive VAR model with a flow matching model to test this hypothesis. In general, we found that FM models are excellent for performing super-resolution. We also demonstrated empirically that when trained on a super-resolution dataset, the vector field learned approximates a linear field. This allowed us to get accurate ODE solutions when using simple solvers. However, if the distribution of generated images is too far from the set of real images trained on, the results suffer. We tried to improve the model further by finetuning in, but ran into problems as the image quality degraded as the model was finetuned. We suspect that this might be due to the dataset we used during finetuning, but also believe that this task is in general way harder since we are no longer learning a linear field, but a complex, non-linear field. This means that we have to use more computationally intensive ODE solver during inference. However, we believe that understanding how to

finetune the models in an efficient and stable way is key to improve the results further.

Goal *Explore the balance between efficiency and quality in image generation models by combining different architectures*

Returning to our research goal, we have successfully combined different architectures for image generation. By varying the size of the UNet used, we were able to control the trade-off between efficiency and quality. While FM models also generally allow for a trade-off between efficiency and quality post-training, by changing the ODE solver, we found that it was strictly better to use a simple ODE solver. We hypothesize that this was because the vector field learned, generates straight paths between the input and output images. As such, we were able to keep the same high quality while improving efficiency when using simpler solvers.

6.2 Future Work

We see several promising avenues for future research. They are all related to challenges or observations from this thesis. Some are simple extensions of the work we have done, while others would require more time and effort to complete.

6.2.1 Finetuning

The most obvious continuation would be to further explore how to finetune the models. We highlighted potential issues in section 5.1. In general, we believe that using a static dataset where the optimal transport plan is solved for larger batches of data is a promising start. We also think that it would be beneficial to calculate the distance between images in the latent space, not in pixel space, as we believe that would be more robust against small augmentations such as rotation, color shift and artifacts created during generation. Additionally, we proposed augmenting the data during the pre-training phase to make the model more robust.

6.2.2 Super-Resolution

Perhaps the most promising finding, is how well suited flow matching models are for super-resolution. We believe that this is an interesting direction to explore further. It would require benchmarking the model against the State-of-the-Art super-resolution models available today. Related to this, we also believe that the results from using different ODE solvers is worth looking into. As we discussed, we believe that the vector field learned is almost straight and demonstrated this empirically, since the results were similar when using a sophisticated and a simple solver. To better understand it, we would suggest doing experiments with other datasets in low-dimensional space where the global optimal transport plan exists (or can be calculated efficiently before training the model) and see if the same effect can be observed. For low-dimensional space, it might be easier to visualize or quantify how straight the paths in the vector field are, to either confirm or disprove our hypothesis.

Additionally, it would be interesting to see if the model can be trained to perform super-resolution on arbitrary sized input images. To achieve this, we could train it on images of different resolution all up-scaled to the target resolution.

6.2.3 Flow Matching in the Latent Space

Inspired by the HART model, we think it would be interesting to train an FM model in latent space. More concretely, it would replace the residual diffusion model they propose in [33]. We would still use the same VAE/VQVAE hybrid encoder-decoder, but instead of learning the residual tokens using a diffusion model, we would learn to transform the quantized embeddings produced by the autoregressive model into the continuous embeddings using an FM model. These would then be passed to the decoder to produce the final image. We believe that this is a good fit for FM models. Instead of using a diffusion model starting from noise and conditioned on the quantized embeddings, an FM model can directly take the embeddings as input, instead of starting from noise. The distance between the input and output should be small. Additionally, for similar images, both the starting point (quantized embeddings) and ending point (continuous embeddings) should be close. We believe that the vector field we are approximating would be well-behaved, and that the training setup provides a clear pairing of data avoiding the issues we reported in the finetuning experiment. However, this would need to be thoroughly tested.

REFERENCES

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

[2] *Electricity 2024*. 2024. URL: https://www.iea.org/reports/electricity-2024.

[3] Thomas Aanensen. *Rekordhøyt strømforbruk i fjor*. 2022. URL: https://www.ssb.no/energi-og-industri/energi/statistikk/elektrisitet/artikler/rekordhoyt-stromforbruk-i-fjor.

[4] Sasha Luccioni, Yacine Jernite, and Emma Strubell. "Power Hungry Processing: Watts Driving the Cost of AI Deployment?" In: *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '24. Rio de Janeiro, Brazil: Association for Computing Machinery, 2024, pp. 85–99. ISBN: 9798400704505. DOI: 10.1145/3630106.3658542. URL: https://doi.org/10.1145/3630106.3658542.

[5] Don Shimamoto. *Multivariable calculus Don Shimamoto*. Don Shimamoto, 2019.

[6] Charles M. Grinstead and J. Laurie Snell. *Introduction to probability*. Amer Mathematical Soc, 2012.

[7] Wikimedia Commons. *File:AI hierarchy.svg — Wikimedia Commons, the free media repository*. [Online; accessed 19-May-2025]. 2025. URL: https://commons.wikimedia.org/w/index.php?title=File:AI_hierarchy.svg&oldid=1028102605.

[8] Wikimedia Commons. *File:Neural network.svg — Wikimedia Commons, the free media repository*. [Online; accessed 19-May-2025]. 2024. URL: https://commons.wikimedia.org/w/index.php?title=File:Neural_network.svg&oldid=907417425.

[9] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. 2019. arXiv: 1803.08375 [cs.NE]. URL: https://arxiv.org/abs/1803.08375.

[10] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[11] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[13] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

[14] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML]. URL: https://arxiv.org/abs/1312.6114.

[15] Diederik P. Kingma and Max Welling. "An Introduction to Variational Autoencoders". In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392. ISSN: 1935-8245. DOI: 10.1561/2200000056. URL: http://dx.doi.org/10.1561/2200000056.

[16] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019). URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[17] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. arXiv:1711.00937 [cs]. May 2018. DOI: 10.48550/arXiv.1711.00937. URL: http://arxiv.org/abs/1711.00937 (visited on 09/06/2024).

[18] Danilo Jimenez Rezende and Shakir Mohamed. *Variational Inference with Normalizing Flows*. 2016. arXiv: 1505.05770 [stat.ML]. URL: https://arxiv.org/abs/1505.05770.

[19] Diederik P. Kingma et al. *Improving Variational Inference with Inverse Autoregressive Flow*. 2017. arXiv: 1606.04934 [cs.LG]. URL: https://arxiv.org/abs/1606.04934.

[20] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG]. URL: https://arxiv.org/abs/1806.07366.

[21] Yaron Lipman et al. *Flow Matching for Generative Modeling*. 2023. arXiv: 2210.02747 [cs.LG]. URL: https://arxiv.org/abs/2210.02747.

[22] Alexander Tong et al. *Improving and generalizing flow-based generative models with minibatch optimal transport*. 2024. arXiv: 2302.00482 [cs.LG]. URL: https://arxiv.org/abs/2302.00482.

[23] Michael Samuel Albergo and Eric Vanden-Eijnden. "Building Normalizing Flows with Stochastic Interpolants". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=li7qeBbCR1t.

[24] Xingchao Liu, Chengyue Gong, and Qiang Liu. *Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow*. 2022. arXiv: 2209.03003 [cs.LG]. URL: https://arxiv.org/abs/2209.03003.

[25] Prafulla Dhariwal and Alex Nichol. *Diffusion Models Beat GANs on Image Synthesis*. 2021. arXiv: 2105.05233 [cs.LG]. URL: https://arxiv.org/abs/2105.05233.

[26] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. *Pixel Recurrent Neural Networks*. 2016. arXiv: 1601.06759 [cs.CV]. URL: https://arxiv.org/abs/1601.06759.

[27] Aaron van den Oord et al. *Conditional Image Generation with PixelCNN Decoders*. 2016. arXiv: 1606.05328 [cs.CV]. URL: https://arxiv.org/abs/1606.05328.

[28] Aditya Ramesh et al. *Zero-Shot Text-to-Image Generation*. 2021. arXiv: 2102.12092 [cs.CV]. URL: https://arxiv.org/abs/2102.12092.

[29] Keyu Tian et al. *Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction*. 2024. arXiv: 2404.02905 [cs.CV]. URL: https://arxiv.org/abs/2404.02905.

[30] Peize Sun et al. *Autoregressive Model Beats Diffusion: Llama for Scalable Image Generation*. 2024. arXiv: 2406.06525 [cs.CV]. URL: https://arxiv.org/abs/2406.06525.

[31] Lawrence Cayton. "Algorithms for Manifold Learning". In: *UC San Diego: Department of Computer Science & Engineering* (2008). Retrieved from eScholarship. URL: https://escholarship.org/uc/item/8969r8tc.

[32] Nick Whiteley, Annie Gray, and Patrick Rubin-Delanchy. *Statistical exploration of the Manifold Hypothesis*. 2025. arXiv: 2208.11665 [stat.ME]. URL: https://arxiv.org/abs/2208.11665.

[33] Haotian Tang et al. *HART: Efficient Visual Generation with Hybrid Autoregressive Transformer*. 2024. arXiv: 2410.10812 [cs.CV]. URL: https://arxiv.org/abs/2410.10812.

[34] Christian Szegedy et al. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV]. URL: https://arxiv.org/abs/1512.00567.

[35] Sadeep Jayasumana et al. *Rethinking FID: Towards a Better Evaluation Metric for Image Generation*. 2024. arXiv: 2401.09603 [cs.CV]. URL: https://arxiv.org/abs/2401.09603.

[36] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2018. arXiv: 1706.08500 [cs.LG]. URL: https://arxiv.org/abs/1706.08500.

[37] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[38] Jian Han et al. *Infinity: Scaling Bitwise AutoRegressive Modeling for High-Resolution Image Synthesis*. 2024. arXiv: 2412.04431 [cs.CV]. URL: https://arxiv.org/abs/2412.04431.

[39] Huiwen Chang et al. *MaskGIT: Masked Generative Image Transformer*. 2022. arXiv: 2202.04200 [cs.CV]. URL: https://arxiv.org/abs/2202.04200.

[40] Yuqing Wang et al. *Bridging Continuous and Discrete Tokens for Autoregressive Visual Generation*. 2025. arXiv: 2503.16430 [cs.CV]. URL: https://arxiv.org/abs/2503.16430.

[41] Tianhong Li et al. *Autoregressive Image Generation without Vector Quantization*. 2024. arXiv: 2406.11838 [cs.CV]. URL: https://arxiv.org/abs/2406.11838.

[42] Jiatao Gu et al. *DART: Denoising Autoregressive Transformer for Scalable Text-to-Image Generation*. 2025. arXiv: 2410.08159 [cs.CV]. URL: https://arxiv.org/abs/2410.08159.

[43] Mude Hui et al. *ARFlow: Autogressive Flow with Hybrid Linear Attention*. 2025. arXiv: 2501.16085 [cs.CV]. URL: https://arxiv.org/abs/2501.16085.

[44] Sucheng Ren et al. *FlowAR: Scale-wise Autoregressive Image Generation Meets Flow Matching*. 2024. arXiv: 2412.15205 [cs.CV]. URL: https://arxiv.org/abs/2412.15205.

[45] Alec Radford et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021. arXiv: 2103.00020 [cs.CV]. URL: https://arxiv.org/abs/2103.00020.

[46] Suorong Yang et al. *Image Data Augmentation for Deep Learning: A Survey*. 2023. arXiv: 2204.08610 [cs.CV]. URL: https://arxiv.org/abs/2204.08610.

[47] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[48] Niyar R Barman et al. *The Brittleness of AI-Generated Image Watermarking Techniques: Examining Their Robustness Against Visual Paraphrasing Attacks*. 2024. arXiv: 2408.10446 [cs.CV]. URL: https://arxiv.org/abs/2408.10446.

[49] James O'Donnell and Casey Crownhart. *We did the math on AI's energy footprint. Here's the story you haven't heard.* May 2025. URL: https://www.technologyreview.com/2025/05/20/1116327/ai-energy-usage-climate-footprint-big-tech/.

[50] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

INDEX

APPENDICES

APPENDIX

A

GITHUB REPOSITORY

There were two repositories used for this project. The main repository,

`https://github.com/oskarjor/FlowFormer`

contains code related to training models, experiments and more. There is also a secondary repository,

`https://github.com/oskarjor/master-thesis-plots`

which contains code used for generating some of the visualizations used throughout the thesis. The code is the second repository is not of high quality, as we do not expect people to use it further. We solely link to be transparent about how plots were generated.

APPENDIX

B

PROOFS

B.1 Proof of Continuity Equation

The following is a proof of the continuity equation in 3-dimensional space. Using the same logic, we can extend this proof to $\mathbb{R}^n$.

We consider a control volume with dimension $\mathrm{d}x, \mathrm{d}y, \mathrm{d}z$ (see Figure B.1.1), were we measure a mass. This could be physical mass, but also probability mass. We are interested in measuring how the density of our mass evolves over time. Our mass has a time-dependent density $p(x, y, z, t)$. We also have a velocity vector field given by $\vec{V}(x, y, z, t)$, describing how the mass moves. We can also decompose this vector field into three scalar fields, where each field describes the movement along one axis:

$$\vec{V}(x, y, z, t) = \begin{bmatrix} v_x(x, y, z, t) \\ v_y(x, y, z, t) \\ v_z(x, y, z, t) \end{bmatrix} \tag{B.1}$$

For any area $\mathbf{A}$, density p and velocity vector field $\vec{V}$, the **mass flow rate**, $\dot{m}$, is given by

$$\dot{m} = p \cdot \vec{V} \cdot \mathbf{A} \tag{B.2}$$

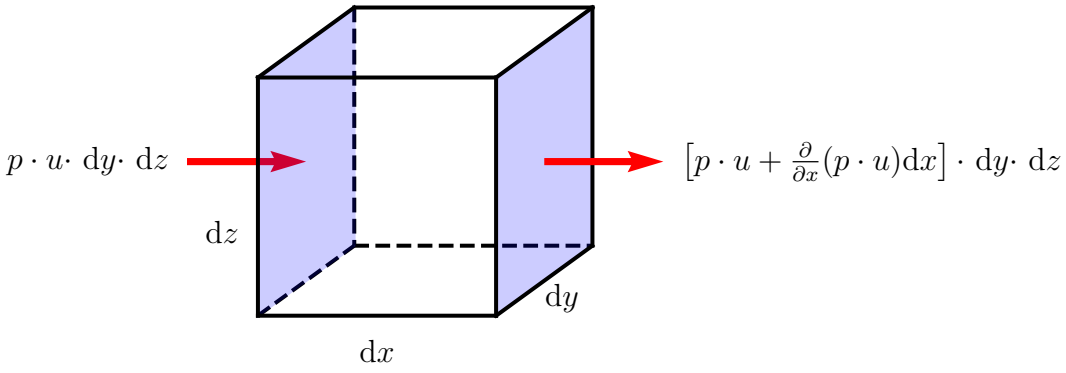


Figure B.1.1: A control volume. Highlighted are the two sides that lie along the x-axis. The left arrow denotes the flow in, while the right arrow shows the flow out.

Recall that $v_x(x, y, z, t)$ describes the velocity of the mass along the x-direction. For our control volume, it follows that the mass flow rate into the cube along the x-direction, $\dot{m}_x$, is given by

$$\dot{m}_x = p \cdot v_x \cdot \mathrm{d}y \cdot \mathrm{d}z \tag{B.3}$$

The outgoing flow, $\dot{n}_x$ is found by doing a Taylor expansion

$$\dot{n}_x = \left[p \cdot u + \frac{\partial}{\partial x}(p \cdot u)\mathrm{d}x + \mathcal{O}((\mathrm{d}x)^2)\right] \cdot \mathrm{d}y \cdot \mathrm{d}z \tag{B.4}$$

As $\mathrm{d}x \to 0$, we can drop the higher order terms, arriving at our final equation for outgoing flow

$$\dot{n}_x = \left[p \cdot u + \frac{\partial}{\partial x}(p \cdot u)\mathrm{d}x\right] \cdot \mathrm{d}y \cdot \mathrm{d}z \tag{B.5}$$

Similarly, we can arrive at equations describing the flow in and out along the other axis. Recall that v, w are the scalar fields describing the velocities along the y and z-direction respectively.

$$
\begin{aligned}
\dot{m}_x &= p \cdot v_x \cdot \mathrm{d}y \cdot \mathrm{d}z \\
\dot{m}_y &= p \cdot v_y \cdot \mathrm{d}x \cdot \mathrm{d}z \\
\dot{m}_z &= p \cdot v_z \cdot \mathrm{d}x \cdot \mathrm{d}y \\
\dot{n}_x &= \left[p \cdot v_x + \frac{\partial}{\partial x}(p \cdot v_x)\mathrm{d}x \right] \cdot \mathrm{d}y \cdot \mathrm{d}z \\
\dot{n}_y &= \left[p \cdot v_y + \frac{\partial}{\partial y}(p \cdot v_y)\mathrm{d}y \right] \cdot \mathrm{d}x \cdot \mathrm{d}z \\
\dot{n}_z &= \left[p \cdot v_z + \frac{\partial}{\partial z}(p \cdot v_z)\mathrm{d}z \right] \cdot \mathrm{d}x \cdot \mathrm{d}y
\end{aligned}
\tag{B.6}
$$

If we think about the density present in the differential volume, $\mathrm{d}B = \mathrm{d}x \cdot \mathrm{d}y \cdot \mathrm{d}z$, at any time, we can say that it is the rate of flow in subtracting the rate of flow out, along each dimension. Plugging in our inflows and outflows from Equation B.6, we get

$$
\begin{aligned}
\frac{\partial p}{\partial t}\mathrm{d}B &= \dot{m}_x + \dot{m}_y + \dot{m}_z - \dot{n}_x - \dot{n}_y - \dot{n}_z \\
&= (\dot{m}_x - \dot{n}_x) + (\dot{m}_y - \dot{n}_y) + (\dot{m}_z - \dot{n}_z) \\
&= (p \cdot v_x \cdot \mathrm{d}y \cdot \mathrm{d}z - \left[p \cdot v_x + \frac{\partial}{\partial x}(p \cdot v_x)\mathrm{d}x \right] \cdot \mathrm{d}y \cdot \mathrm{d}z) \\
&+ (p \cdot v_y \cdot \mathrm{d}x \cdot \mathrm{d}z - \left[p \cdot v_y + \frac{\partial}{\partial y}(p \cdot v_y)\mathrm{d}y \right] \cdot \mathrm{d}x \cdot \mathrm{d}z) \\
&+ (p \cdot v_z \cdot \mathrm{d}x \cdot \mathrm{d}y - \left[p \cdot v_z + \frac{\partial}{\partial z}(p \cdot v_z)\mathrm{d}z \right] \cdot \mathrm{d}x \cdot \mathrm{d}y) \\
&= -\frac{\partial}{\partial x}(p \cdot v_x)\mathrm{d}B - \frac{\partial}{\partial y}(p \cdot v_y)\mathrm{d}B - \frac{\partial}{\partial z}(p \cdot v_z)\mathrm{d}B
\end{aligned}
\tag{B.7}
$$

Dividing both sides by $\mathrm{d}B$, we end up with

$$
\frac{\partial p}{\partial t} = -\frac{\partial}{\partial x}(p \cdot v_x) - \frac{\partial}{\partial y}(p \cdot v_y) - \frac{\partial}{\partial z}(p \cdot v_z) \tag{B.8}
$$

We then see that

$$
\begin{aligned}
\frac{\partial p}{\partial t} &= -(\frac{\partial}{\partial x}(p \cdot v_x) + \frac{\partial}{\partial y}(p \cdot v_y) + \frac{\partial}{\partial z}(p \cdot v_z)) \\
&= -\left(\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) \cdot ((p \cdot v_x), (p \cdot v_y), (p \cdot v_z))\right) \\
&= -\left(\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) \cdot \left((p \cdot \vec{V})\right)\right) \\
&= -\operatorname{div}(p \cdot \vec{V})
\end{aligned}
$$

We have know shown that

$$
\frac{\partial p}{\partial t} = -\operatorname{div}(p \cdot \vec{V}) \tag{B.9}
$$

which is the formula known as the *general continuity equation*.

APPENDIX

C

OUTPUT FROM MODELS

C.1 VAR backbones

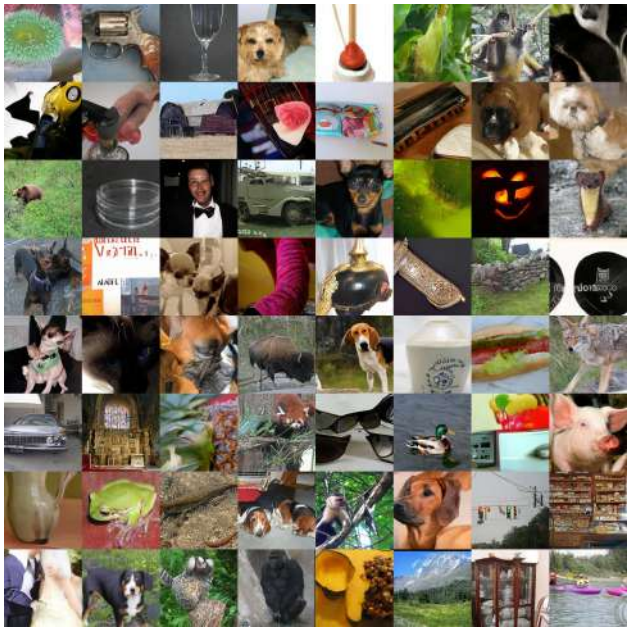


Figure C.1.1: 64 images from VAR-D16 (256×256 pixels).

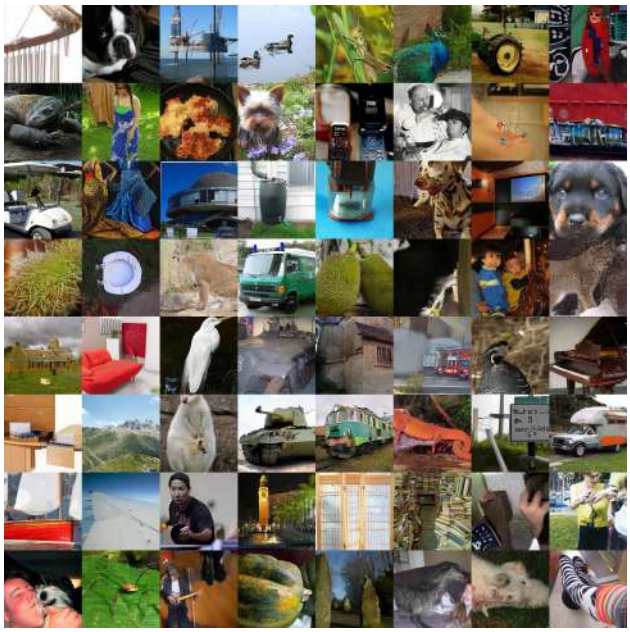


Figure C.1.2: 64 images from VAR-D30 (256 × 256 pixels).

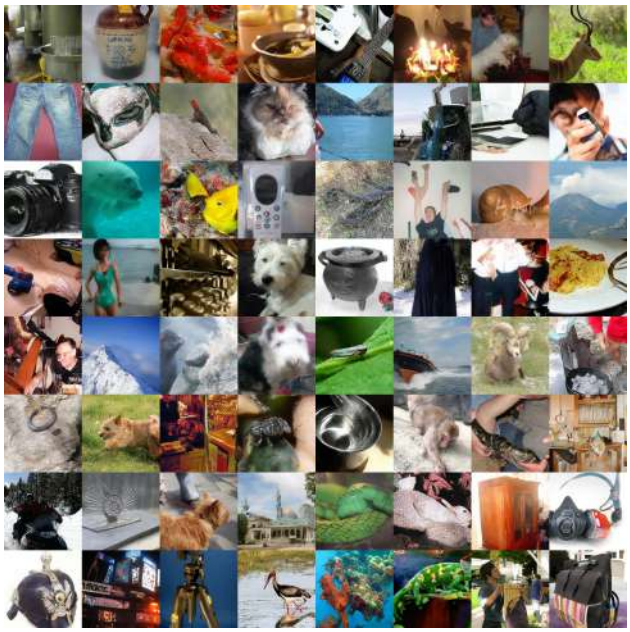


Figure C.1.3: 64 images from VAR-D36 (512 × 512 pixels).

C.2 Flow Matching Models

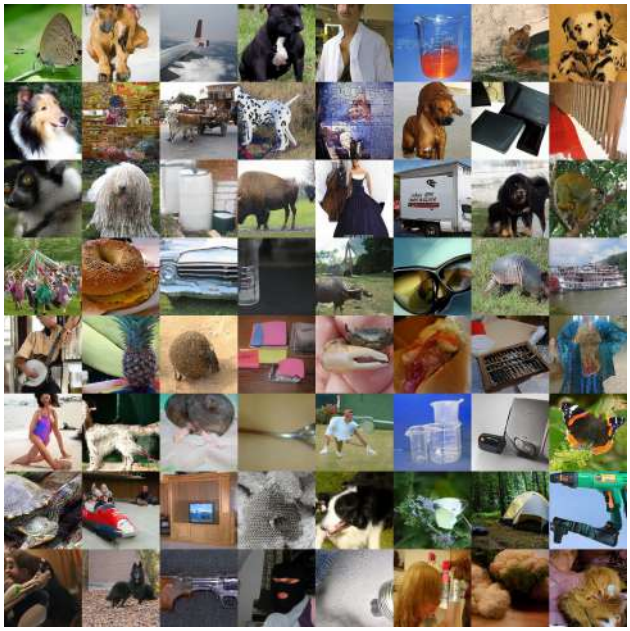


Figure C.2.1: 64 images from FM-N-D16 (512×512 pixels).

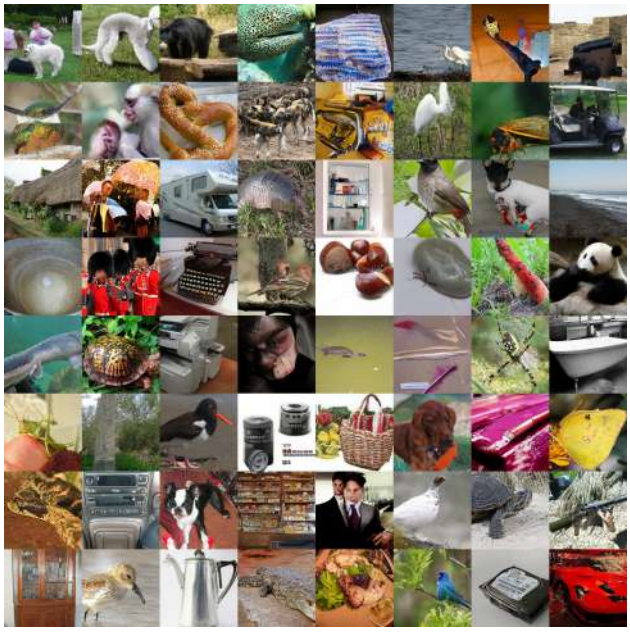


Figure C.2.2: 64 images from FM-L-D16 (512×512 pixels).

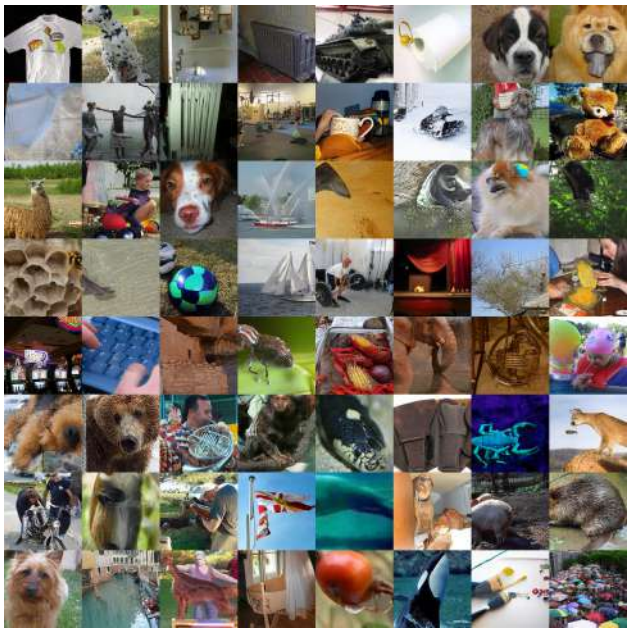


Figure C.2.3: 64 images from FM-N-D30 (512 × 512 pixels).

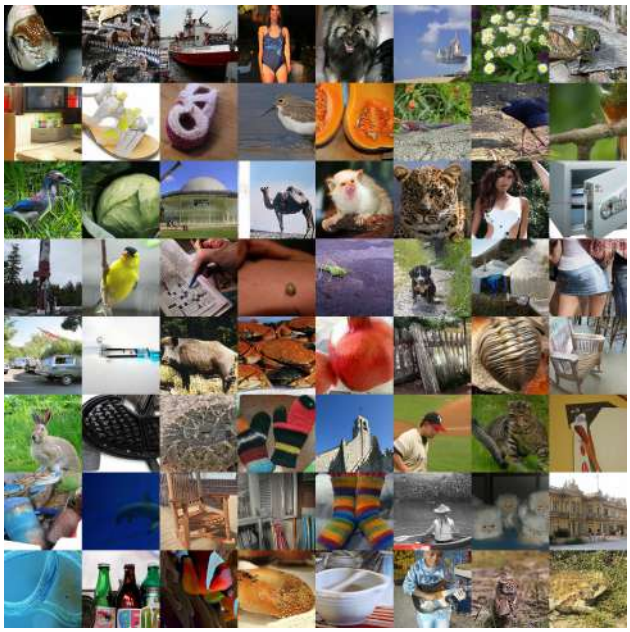


Figure C.2.4: 64 images from FM-L-D30 (512 × 512 pixels).

C.3 Finetuned Flow Matching Models

Figure C.3.1: 64 images from FM-N-D16-FT (512×512 pixels).

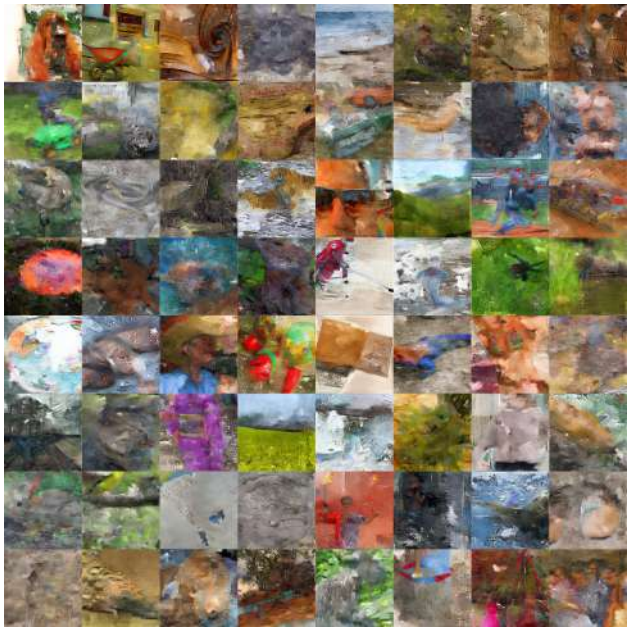


Figure C.3.2: 64 images from FM-L-D16-FT (512×512 pixels).

Figure C.3.3: 64 images from FM-N-D30-FT (512 × 512 pixels).

Figure C.3.4: 64 images from FM-L-D30-FT (512 × 512 pixels).

C.4 Lightweight Flow Matching Models

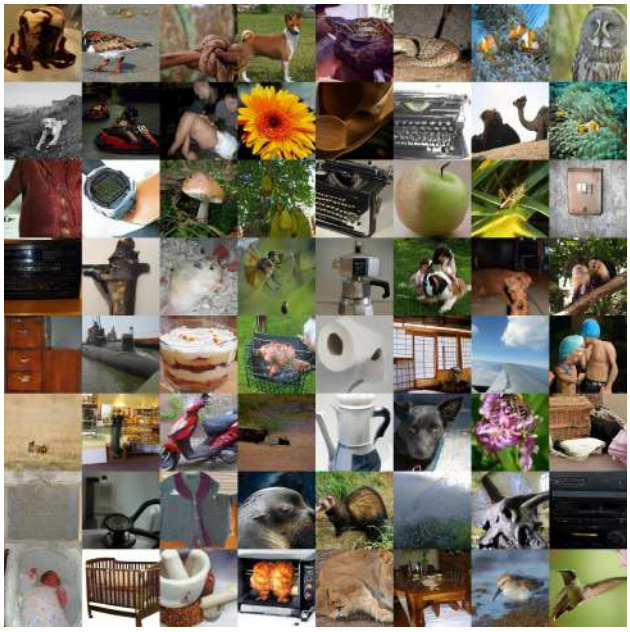


Figure C.4.1: 64 images from FM-N-D16-LW (512 × 512 pixels).

Figure C.4.2: 64 images from FM-N-D30-LW (512 × 512 pixels).

APPENDIX

D

HYPERPARAMETER

For the VAR model, we have the following hyperparameters during inference:

- **batch-size (int)**: the number of images sampled for each batch. Generally, we set this as high as possible without running out of memory on the GPU.
- **depth**: the model depth. A deeper model, means more parameters and a model with the capacity to learn more complex patterns. However, it also means that inference is more resource consuming.
- **cfg (float)**: classifier-free guidance. This determines how strongly the model has to adhere to classifier free guidance.
- **top-k (int)**: since VAR is sampling tokens from a probability distribution, we could in theory get very unlikely tokens. We therefore limit the model to only select from the top k most probable tokens.
- **top-p (float)**: similarly to top-k-sampling, we limit the model to only select from the most probable tokens that together account for p of the probability mass. For instance, if we have top-p $= 0.9$, we only select from the n most probable tokens that together account for 90% of the total probability mass.
- **save-format**: whether to save the images as PNG (lossless) or JPEG (lossy).

We do not report batch size during inference for the VAR models, but recommend setting it as high as your GPU memory allows. For the flow matching model we have the following hyperparameters during training:

- **total-steps (int)**: the total number of training steps. Each step consists of training on one batch.
- **batch-size (int)**: the number of images sampled for each batch. Generally, we set this as high as possible without running out of memory on the GPU. Note that if we want to train on the same amount of data while doubling the batch size, we would have to half the total-steps.
- **learning-rate (float)**: the learning rate (commonly abbreviated LR). We use the ADAM optimizer [50].
- **grad-clip (float)**: gradient clipping. This clips the gradients if their norm is larger than the specified value.
- **ema-decay (float)**: we train another model using the exponential moving average of the gradients. We need to specify the value by which the average decays over time. This is a trick from the ADAM paper [50].
- **warmup (int)**: the number of steps our learning rate warms up. The learning rate increases linearly until it reaches its final value. It then remains constant during training.
- **dataset**: what dataset to train on. We use both ImageNet and our custom datasets, denoted VAR-D16-ImageNet and VAR-D30-ImageNet.
- **naive-upscaling**: what method to use when naively up-scaling the images. Either NEAREST (repeating pixels) or LANCZOS (using the LANCZOS kernel to interpolate values).

We recommend setting the batch size to a power of 2, and as high as you GPU memory allows. We report it during training (as it relates to how much data the model sees), but not during inference. Furthermore, we always use grad-clip = 1, warmup = 5000 and ema-decay = 0.999.

For the flow matching model we have the following hyperparameters during inference:

- **batch-size (int)**: the number of images sampled for each batch. Generally, we set this as high as possible without running out of memory on the GPU. Note that if we want to train on the same amount of data while doubling the batch size, we would have to half the total-steps.
- **backbone**: what VAR model to use as the backbone during generation. We use pre-generated images (as opposed to sampling them right before passing them to the FM model).
- **naive-upscaling**: what method to use when naively up-scaling the output from the VAR model. Either NEAREST (repeating pixels) or LANCZOS (using the LANCZOS kernel to interpolate values).
- **ODE-solver**: what ODE solver we use. We can select from fixed step size solvers (EULER, RUNGE-KUTTA-4) where we need to specify the step size, or adaptive step size solver (DOPRI-5), where we need to specify the absolute and relative error tolerance.

Finally, we have hyperparameters for the UNet (which represent the vector fields the FM models are learning). The UNet remains the same during training and inference. It follows the implementation from [25]:

- **num-res-blocks (int)**: the number of residual block in each down-sampling step of the UNet. A higher value, means more parameters, more capacity to learn and slower training/inference.
- **num-channel (int)**: the number of base channels for the model. A higher value, means more parameters, more capacity to learn and slower training/inference.
- **channel-mult (int)**: the channel multiplier for each level of the UNet.
- **num-heads (int)**: the number of attention heads in each multi-head attention layer.
- **attention-resolutions**: at which down-sample rates attention is used.
- **dropout (float)**: the number of edges that are set to 0 during training.
- **class-conditional (bool)**: if the model is class conditional or not.
- **num-classes (int)**: the number of classes (only relevant if the model is class conditional).

In general, all UNets use the following value. num-res-blocks = 2, num-channel = 128, channel-mult = $(0.5, 1, 1, 2, 2, 4, 4)$, num-heads = 2, attention-resolutions = (16), dropout = 0.1, class-conditional = true and num-classes = 1000. The only exception is in experiment 5 where we use a lightweight UNet. It has num-res-blocks = 1, num-channel = 64, num-heads = 4, but otherwise remains the same. For an in-depth understanding of the UNet implementation, we suggest looking at the code from [25].

D.1 Experiment 1

Table D.1.1 describes the hyperparameters for the models in Experiment 1.1.

D.1.1 Experiment 1.1

Table D.1.1: The table describing the models for Experiment 1.1.

Model Name	Naive Upscaling	Depth	cfg	top-k	top-p	Save Format
VAR-D30	NEAREST	30	4.0	900	0.96	PNG
VAR-D30	LANCZOS	30	4.0	900	0.96	PNG
VAR-D36	-	36	1.5	900	0.96	PNG

The authors of the VAR paper report three different values for cfg. The paper reports 2.0, but the code logs report 1.5. Finally, in the sampling script provided, this value is by default set to 4.0. This led to some confusion. It is therefore unclear what the cfg value for VAR-D36 actually is. The same confusion arises for the value of top-k. The paper only reports this for models with a lower depth, and uses top-k = 600. However, the code repository uses 900 by default for all depths.

D.1.2 Experiment 1.2

Figure D.1.2 describes the hyperparameters for the models in Experiment 1.2.

Table D.1.2: The table describing the models for Experiment 1.2.

Model Name	Naive Upscaling	Depth	cfg	top-k	top-p	Save Format
VAR-D30	NEAREST	30	4.0	900	0.96	PNG
VAR-D30	NEAREST	30	4.0	900	0.96	JPEG
VAR-D30	LANCZOS	30	4.0	900	0.96	PNG
VAR-D30	LANCZOS	30	4.0	900	0.96	JPEG
VAR-D36	-	36	1.5	900	0.96	PNG

D.2 Experiment 2

Experiment 2 set out to find the best hyperparameters for our FM models. Experiment 2.1 looks at what naive up-scaling we use, and which backbone, while Experiment 2.2 looks at the classifier free guidance strength of the VAR backbone.

D.2.1 Experiment 2.1

Table D.2.1 describes the hyperparameters for the VAR backbones. Table D.2.2 describes the hyperparameters for the flow matching models. Since we have 2 backbones and 2 FM models, we can create a total of 4 models by combining them, see Table D.2.3.

Table D.2.1: The table describing our VAR backbones for Experiment 2

Model Name	Depth	cfg	top-k	top-p	Save Format
VAR-D16	16	4.0	900	0.96	PNG
VAR-D30	30	4.0	900	0.96	PNG
VAR-D36	36	1.5	900	0.96	PNG

Table D.2.2: The table describing our FM models for Experiment 2.

Model Name	Total Steps	Batch Size	LR	Dataset	Naive Upscaling
FM-N	400k	8	1e-4	ImageNet	NEAREST
FM-L	400k	8	1e-4	ImageNet	LANCZOS

Table D.2.3: All model combinations used in Experiment 2, with hyperparameters for inference.

Model Name	VAR backbone	FM Model	Naive Upscaling
FM-N-D16	VAR-D16	FM-N	NEAREST
FM-L-D16	VAR-D16	FM-L	LANCZOS
FM-N-D30	VAR-D30	FM-N	NEAREST
FM-L-D30	VAR-D30	FM-L	LANCZOS

D.2.2 Experiment 2.2

We change the strength of the classifier free guidance when generating images with the VAR backbone. We select this hyperparameter specifically because we found that it was unclear which value was used in the original paper and that it seems to impact performance a lot.

Table D.2.4: The table describing our VAR backbones for Experiment 2

Model Name	Depth	cfg	top-k	top-p	Save Format
VAR-D16	16	4.0	900	0.96	PNG
VAR-D16-V2	16	1.5	900	0.96	PNG
VAR-D30	30	4.0	900	0.96	PNG
VAR-D30-V2	30	1.5	900	0.96	PNG
VAR-D36	36	1.5	900	0.96	PNG

Table D.2.5: All model combinations used in Experiment 2, with hyperparameters for inference.

Model Name	VAR backbone
FM-N-D16	VAR-D16
FM-N-D16-V2	VAR-D16-V2
FM-N-D30	VAR-D30
FM-N-D30-V2	VAR-D30-V2

D.3 Experiment 3

In Experiment 3, we finetune the models we trained in Experiment 2 on custom datasets. The datasets are generated using the backbones VAR-D16 and VAR-D30 using the hyperparameters in Table D.2.1. The fine-tuning hyperparameters are detailed in Table D.3.1. The inference hyperparameters are the same as their base models, described in Table D.2.3.

Table D.3.1: Our finetuned models. They differ in what base model they use and what dataset we finetune them on.

Model Name	Base Model	FT Steps	Batch Size	Dataset
FM-N-D16-FT	FM-N-D16	10k	16	VAR-D16-ImageNet
FM-L-D16-FT	FM-L-D16	10k	16	VAR-D16-ImageNet
FM-N-D30-FT	FM-N-D30	10k	16	VAR-D30-ImageNet
FM-L-D30-FT	FM-L-D30	10k	16	VAR-D30-ImageNet

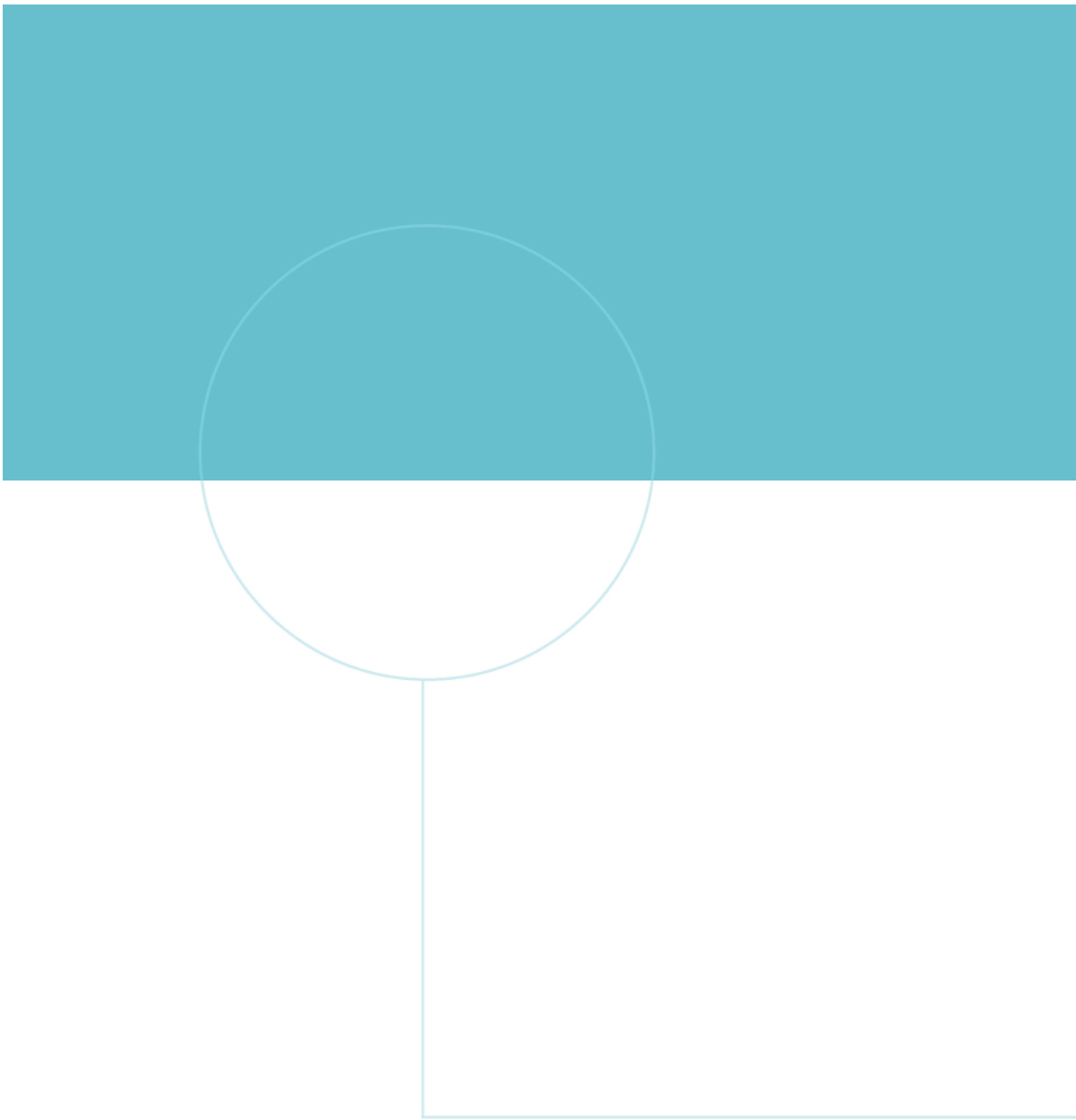
D.4 Experiment 4

In Experiment 4, we vary the size of the UNet and the precision of the ODE solver to explore the trade-offs between efficiency and quality. Table Table D.4.1 detail the hyperparameters of the smaller UNet. All other parameters are listed in the experiment details.

D.4.1 Experiment 4.1

Table D.4.1: The parameters of the standard UNet, used in all previous experiments, and the lightweight UNet introduced in Experiment 4.1

Model Name	# of Param.	# of Res. Blocks	# of Channels	# of Heads
Standard	120M	2	128	2
Lightweight	23M	1	64	4



NTNU

Norwegian University of
Science and Technology