UNIVERSITY OF
CAMBRIDGE

DEPARTMENT OF
COMPUTER SCIENCE
AND TECHNOLOGY

Denoising Diffusion Probability Models for Image Inpainting

Dissertation

Pranav Talluri

2022 - 2023

Declaration of Originality

I, Pranav Talluri of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: 

Date: 12/05/2023

Proforma

Candidate Number:	2367B
Project Title:	Denoising Diffusion Probabilistic Models for Image Inpainting
Examination:	Computer Science Tripos - Part II
Year:	2023
Word Count:	11985
Code line Count:	8998
Project Originator:	Param Hanji
Project Supervisor:	Param Hanji
Original Aims:	This dissertation originally aimed to investigate the application of diffusion models in image inpainting. This would be achieved through first implementing unconditional generation using diffusion models, and then extending this implementation for conditional tasks. It aimed to then explore various improvements in the field and apply them to the task of image inpainting. Finally, it aimed to create a user facing web-application that allowed users to edit images using inpainting.
Work Completed:	This dissertation achieved all its original aims. In it, I implemented unconditional, and then conditional diffusion models. I experimented with colourisation and inpainting. I explored various improvements found in the latest literature, which lead to improvements in sample quality and sampling speed. I was able to achieve state of the art results for image inpainting. I then developed a web application that utilises this state-of-the-art model to allow users to edit images by drawing masks and then performing inpainting.
Special Difficulties:	None

Contents

Chapter 1: Introduction

This dissertation explores Denoising Diffusion Probabilistic Models and their application to image inpainting. It implements DDPMs from scratch to perform unconditional image generation, matching recent results. This implementation is extended to perform conditional tasks, including image inpainting. Various improvements are explored, with a detailed exploration of their suitability for conditional tasks. We shall see that the changes implemented aid sample quality, achieving state-of-the-art results for image inpainting, and sample speed, making the models practical for use in interactive applications. This dissertation includes the creation of a web-application - EDIFFUSER - that allows users to edit images using inpainting. This chapter introduces the motivation behind the project. It details the contributions made by this dissertation and how they relate to existing literature.

(a) Ground Truth (b) Masked image (c) Output 1 (d) Output 2 (e) Output 3

Fig. 1.1: Inpainting sample generated using EDIFFUSER

1.1 Motivation

Generative modelling uses Machine Learning (ML) to model data distributions and generate new data samples. This is in contrast to discriminative modelling, which models the decision boundary between data distributions. In recent years, deep generative models have gained prevalence following notable breakthroughs, such as Generative Adversarial Networks (GANs) [13], Variational Auto-Encoders (VAEs) [24] and Normalising Flows (NFs) [18].

These methods have opened the floodgates for the generation of creative works using Artificial Intelligence (AI), in a variety of media. For example, GPT4 [51] is a language model that can be used to generate text and WaveNet [17] is a waveform model that can produce speech. For image generation, GANs, VAEs and NFs each come with drawbacks that make them far from ideal. Image generation methods have 3 key aims:

Sample Quality: The extent to which the samples appear as if they could have been drawn from the training dataset.
Sample Diversity: The extent to which the range of generated samples matches the range of samples in the training dataset. Without this aim, we could encounter mode collapse, where models deterministically produce a single very high quality sample.
Sampling Speeds: How long it takes to generate a sample. The sample speed of a model has a large impact on its potential use cases.

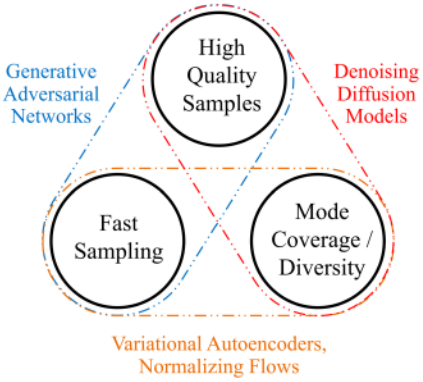


Fig. 1.2: Generative Learning Trilemma

Currently available methods have lead to an observation, the Generative Learning Trilemma, which identifies that methods are usually only able to excel in, at most, two of these aims [46]. GANs offer good sample quality and fast sampling but have poor sample diversity. VAEs/NFs are fast and produce diverse samples, but do not offer sufficient sample quality.

In my dissertation, I will focus on another generative technique - Denoising Diffusion Probabilistic Models (DDPMs), which have been shown to deliver impressive image generation results. They are a novel class of latent variable models which learn to denoise a Gaussian prior into a learned data distribution through an iterative sampling algorithm [15]. DDPMs are trained to learn the small changes at each iteration, rather than directly learning the data distribution. They have rapidly become the most prevalent method for image generation, being utilised in notable tools such as DALLE-2 [40], StableDiffusion [41] and Imagen [44].

As well as unconditionally generating images, DDPMs can be used for various image-to-image translation tasks. I will focusing on applying them to **image inpainting**, in which user-masked regions of an image are realistically filled in (see Figure 1.1). As indicated in Figure 1.2, DDPMs offer excellent sample quality and diversity but suffer from slow sampling. I will therefore explore techniques for **improving sampling speed** to address this trade-off. Taking advantage of faster sampling, I detail a **web-application that utilises inpainting** diffusion models to allow users to edit images.

1.2 Previous Works

Diffusion Models are relatively new in generative modelling literature. They iteratively destroy the structure of the data distribution by applying noise, and learn to denoise the intermediate samples in the process, yielding a tractable generative model of the data [15]. The idea is inspired by concepts in Non-equilibrium Statistical Physics [3], which explores the time evolution of physical systems subject to statistical descriptions. Early works were tested on toy problems and low resolution images, but were not feasible for larger images.

This was addressed in later literature, which reparameterised the learning objective, improving the training process, and showing that Diffusion Models are indeed able to generate large, high quality image samples [27]. The reparameterisation lead to an equivalence with Denoising Score Matching [31], hence the nomenclature - Denoising Diffusion Probabilistic Models (DDPMs). While delivering impressive results for image generation, these early DDPMs also raised further questions, such as usability for real world problems and whether it was possible to generate samples quickly.

To a large extent, real world problems involving image generation have requirements which need to be met and semantics which need to be considered. i.e., we are often working within a specific problem domain and hence need to be able to condition the generation process on additional inputs. When we condition on other images, we can perform image-to-image translation tasks [43]. These tasks include uncropping, which extends existing images, colourisation, which converts greyscale images to colour, and inpainting, which realistically fills in images.

Inpainting has existed with varying quality for over a decade. A notable consumer-facing example is Adobe's Content-Aware Fill, in which a masked region of an image is filled, based on approximate nearest-neighbour matches, using the PatchMatch algorithm [8]. These earlier methods worked well on textured regions but struggled with generating semantically consistent structure [5, 7, 12]. Newer, GAN-based methods are popular but require auxiliary objectives for context and hand-engineered features [28].

The methodology used by DDPMs to generate image samples is inherently sequential, and utilises many iterations. This results in slow sampling speeds, especially compared to more established techniques such as GANs, and prevents DDPMs from being utilised in real-time or

interactive applications. However, various techniques have since been developed to enable (10-50x) faster image generation. One method is to use an alternative sampling framework, known as Denoising Diffusion Implicit Models (DDIM), which incorporates a non-Markovian sampling procedure [30]. Another option is to utilise a modified training objective that encompasses more learnable parameters [33].

The latest research in the field approaches the concept of diffusion from a new perspective, framing the problem as one that can be modelled using Stochastic Differential Equations (SDEs) and reverse-time SDEs (or their corresponding Ordinary Differential Equations (ODEs)) [32]. This generalisation encompasses DDPMs and other generative modelling methods that evolve a distribution over time, acting as a framework which can accept various SDEs to express different learning characteristics. Utilising SDEs enables a continuous time generalisation of DDPMs, which can deliver better performance for unconditional generation. Sampling is performed by utilising black-box SDE/ODE solvers, which can take advantage of the specific structure of the SDEs utilised in this framework to deliver samples up to 100x faster than standard DDPMs [37].

1.3 My Contributions

- Implementation of DDPMs for unconditional image generation (from scratch), matching the results of recent works in the field in terms of sample quality, diversity and sampling speed. (cf. Section 4.2.2.1)
- Implementation of conditional image generation for colourisation and inpainting
- Implementation of techniques for improving sample quality (up to 29%) and sampling speed (up to 70x) for unconditional generation.
- **Experimentation** with techniques for improving sample quality for conditional generation, achieving **SOTA results for inpainting** on the CelebAHQ-256 dataset (cf. Section 4.2.5.2).
- Experimentation with techniques for improving sampling speed for conditional generation, achieving performance which enables the use of models in real-time/interactive applications.
- Implementation of the SDE generalisation of DDPMs, to achieve further performance improvements and adaptation of advanced ODE solvers to enable faster sample generation (up to 37x). Includes an **evaluation of performance of SDE framework for conditional generation**.
- **Creation of a web application**, called **EDIFFUSER**, that allows users to edit images utilising image inpainting (cf. Section 4.2.7).

Chapter 2: Preparation

This chapter explores the theoretical underpinnings of DDPMs, and the subsequent developments in the field (Section 2.1). It details the starting point for the project (Section 2.2), and discusses the planning undertaken before development, including the dataset choices (Section 2.3), project deliverables (Section 2.4), software development strategy (Section 2.6) and technology stack (Section 2.7).

2.1 Theory

2.1.1 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models (DDPMs) consist of a forward and reverse process. The forward process, which is fixed *a priori*, iteratively perturbs an image. The reverse process, which is learnt by a neural network, reverses these perturbations steps. The network is trained by applying the forward process to random samples from the data distribution (training dataset) and optimising a learning objective. Once the network is trained, samples are generated by applying the reverse process to a sample from the prior distribution. The prior is usually a Gaussian, as they are closed under the perturbations we perform, which enable us to model the intermediate distributions as Gaussians as well.

2.1.1.1 The Forward Process (Data → Prior)

Starting with a sample from the data distribution, $x_0 \in X_0$, the forward process adds varying amounts of noise over T timesteps, generating a sequence of intermediate samples $x_{1:T} = x_1, ..., x_T$, where (x_T) approaches pure noise. The process is a Markov chain, represented by q, where each intermediate term is assumed to be a Gaussian.

$$q(x_t|x_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbf{I}) \tag{2.1}$$

$$q(x_{1:T}|x_0) := q(x_1|x_0)q(x_2|x_1)...q(x_T|x_{T-1}) = \prod_{t=1}^{T} q(x_t|x_{t-1}) \tag{2.2}$$

where $\beta_t \in (0, 1)$ is the variance at time t. It is usually fixed to a predetermined variance schedule, which generally increases over time so that the mean of each Gaussian gets closer to 0. In the limit, $t \to \infty$, $q(x_T|x_0) \to \mathcal{N}(0, \mathbf{I})$, so all information about the original sample is lost. Early works show that a large, but finite, number of steps ($T \approx 1000$) is sufficient [15, 27]. We want a large number of steps as it allows us to set individual variances β_t to very small values, while still approximately maintaining the same limiting distribution. i.e., spreading the transformation from the data distribution to the prior over more, smaller steps makes learning to undo the steps of the forward process is feasible.

2.1.1.2 The Reverse Process (Prior → Data)

The reverse process is tasked with learning how to denoise samples from the prior distribution, x_T, to generate a sample, x_0, from the data distribution. This process is once again a Markov chain, represented by p_θ, where θ is a set of parameters learnt by our model. In addition to the intermediate sample, x_t, the model also takes the timestep t as an input to account for the forward process variance schedule. i.e., the reverse process accounts for the fact that different timesteps are associated with different noise levels, and learns to undo these individually.

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \tag{2.3}$$

$$p(x_T) = q(x_T) = \mathcal{N}(0, \mathbf{I}) \tag{2.4}$$

$p_\theta(x_{t-1}|x_t)$ can utilise the fact that, in the limit of infinitesimal step sizes, the true reverse process has the same functional form as the forward process [1]. We are therefore able to parameterise each learned reverse step, $p_\theta(x_{t-1}|x_t)$, as a Gaussian, like the forward process.

$$p_\theta(x_{t-1}, x_t) := \mathcal{N}(\mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{2.5}$$

where μ_θ and Σ_θ are learnable functions for the mean and variance of the Gaussian. We fix Σ_θ to constants *a priori*, as early works found that learning them led to unstable training and lower quality samples [27]. The causes and solutions to this are explored in Section 2.1.3.2. The neural network is then solely tasked with learning the means, μ_θ.

$$\Sigma_\theta(x_t, t) = \sigma_t^2 \mathbf{I} \qquad p_\theta(x_{t-1}, x_t) := \mathcal{N}(\mu_\theta(x_t, t), \sigma_t^2 \mathbf{I}) \tag{2.6}$$

2.1.1.3 Training Objective L

The reverse process neural network is tasked with denoising intermediate samples, x_t, to give x_{t-1}. The reverse process iteratively applies the network, starting from x_T, till we reach a sample x_0 (known as Ancestral Sampling). The naive optimisation approach would therefore be by Maximum Likelihood Estimation (MLE). However, this involves maximising the density assigned to x_0 by the model, which requires calculating $p_\theta(x_0) = \int p_\theta(x_{0:T}) dx_{1:T}$. This entails marginalising over all possible ways of arriving at x_0, starting from a noise sample, which is intractable.

However, it is tractable to maximise a lower bound on the likelihood. Casting x_0 as the observed variable and the intermediate samples, $x_1, ..., x_T$, as latent variables, frames DDPMs as latent variable generative models. We can then apply the training objective used in VAEs [24]. We use the Variational Lower Bound (VLB, aka ELBO) on the marginal log likelihood, which is given by a likelihood term subtracted by a KL divergence term. The likelihood term encourages the model to maximise the expected density assigned to the data. The KL-Divergence term encourages the approximate posterior, $q(z|x)$, to be similar to the prior on the latent variable, $p_\theta(z)$. Specifically, we consider the negative VLB as a bound on the negative log likelihood (NLL).

$$-\log p_\theta(x) \leq -\text{VLB}$$

$$\text{VLB} = \underbrace{\mathbb{E}_{q(x_{1:T}|x_0)}\left[\log p_\theta(x_0|x_{1:T})\right]}_{\text{Likelihood Term}} - \underbrace{D_{\text{KL}}(q(x_{1:T}|x_0) \| p_\theta(x_{1:T}))}_{\text{KL Divergence Term}}$$

We can simplify this to yield our training objective.[1] See Appendix A.1 for the derivation.

$$L := -\mathbb{E}_q\left[\log p_\theta(x_0|x_{1:T})\right] + D_{\text{KL}}(q(x_{1:T}|x_0) \| p_\theta(x_{1:T})) \tag{2.7}$$

$$:= -\mathbb{E}_q\left[\log p(x_T) + \sum_{t \geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right] \tag{2.8}$$

2.1.1.4 Modified Training Objective $\mathcal{L}_{\text{vlb}}$

There are many possible sequences of intermediate samples x_{t-1} between x_0 and x_T. The objective in Equation 2.8 optimises over them all and can therefore have high variance, limiting

[1] $\mathbb{E}_q$ denotes $\mathbb{E}_{q(x_{1:T}|x_0)}$

training efficiency. This can be addressed by rearranging the objective. See Appendix A.2 for the full derivation.[2]

$$\mathcal{L}_{\text{vlb}} := \mathbb{E}_q \left[\underbrace{D_{\text{KL}}(q(x_T|x_0)\|p_\theta(x_T))}_{\mathcal{L}_T} + \sum_{t>1} \underbrace{D_{\text{KL}}(q(x_{t-1}|x_t, x_0)\|p_\theta(x_{t-1}|x_t))}_{\mathcal{L}_{t-1}} \underbrace{- \log p_\theta(x_0|x_1)}_{\mathcal{L}_0} \right]$$

$$\mathcal{L}_{\text{vlb}} := \mathbb{E}_q \left[\sum_{t>1} \mathcal{L}_{t-1} + \mathcal{L}_0 \right] \tag{2.9}$$

When the original sample x_0 is known, as it is during training, Bayes' rule shows that the $q(x_{t-1}|x_t, x_0)$ terms are just Gaussians.

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(\tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t \mathbf{I})$$

$$\tilde{\beta}_t := \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \beta_t$$

$$\tilde{\mu}_t(x_t, x_0) := \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1 - \bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t} x_t$$

Variance is also introduced in the training process as the $\mathcal{L}_{t-1}$ terms are given by the KL divergence between the $q(x_{t-1}|x_t, x_0)$ and $p_\theta(x_{t-1}|x_t)$. In the general case, where their densities are not known, we must perform a Monte Carlo estimation for $\mathcal{L}_{t-1}$ terms. However, recall from Equation 2.6 that the reverse steps, $p_\theta(x_{t-1}|x_t)$, are also parameterised as Gaussians, so each KL-divergence term is simply a comparison between two Gaussians. This allows us to evaluate these terms in closed form, reducing the variance. The derivation is given in Appendix A.3.

$$\mathcal{L}_{t-1} = \mathbb{E}_q \left[\frac{1}{2\sigma_t^2} \|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2 \right] \tag{2.10}$$

2.1.1.5 Reparameterising $\mathcal{L}_{t-1}$

Based on $\mathcal{L}_{t-1}$, we could learn to predict $\tilde{\mu}_t$ (the mean), or we could learn to predict x_0 (the denoised image) and then derive the mean using $q(x_{t-1}|x_t, x_0)$. We instead reparameterise $\mathcal{L}_{t-1}$ and learn the noise added rather than the means as early works found this to be the most effective [27]. We first note that the sum of independent Gaussian steps is also a Gaussian, so we can sample any arbitrary step of the forward process in closed form.

$$q(x_t|x_0) = (x_t; \sqrt{\bar{\alpha}_t}x_0, (1 - \bar{\alpha}_t)\mathbf{I}) \qquad \alpha_t := 1 - \beta_t \qquad \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$$

We introduce an auxiliary noise variable $\epsilon \sim \mathcal{N}(0, I)$ (a constant distribution independent of t) to give an expression for x_t, which can be rearranged to get an expression for x_0.

$$x_t(x_0, \epsilon) = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon \qquad x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t(x_0, \epsilon) - \sqrt{1 - \bar{\alpha}_t}) \tag{2.11}$$

This means that at training time, we can obtain any term of the objective without having to simulate the entire chain, which would be very slow. The objective can be optimised by randomly sampling pairs of x_{t-1} and x_t, and maximising the conditional density, $p_\theta(x_{t-1}|x_t)$.

We can plug these two definitions into $\tilde{\mu}_t(x_t, x_0)$, and then rewrite $\mathcal{L}_{t-1}$ from Equation 2.10 accordingly. x_0 is known during training, so the reverse step model just optimises to predict ϵ.

$$\mathcal{L}_{t-1} = \mathbb{E}_{x_0, \epsilon} \left[\frac{1}{2\sigma_t^2} \left\| \frac{1}{\sqrt{\alpha_t}} \left(x_t(x_0, \epsilon) - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon \right) - \mu_\theta(x_t(x_0, \epsilon), t) \right\|^2 \right] \tag{2.12}$$

[2]Note that $\mathcal{L}_T$ is a constant, independent of q, omitted from this point onwards for brevity.

$\mathcal{L}_{t-1}$ is minimised when μ_θ predicts $\tilde{\mu}_t$. μ_θ must therefore predict $\frac{1}{\sqrt{\alpha_t}}\left(x_t(x_0,\epsilon) - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)$ when given x_t. To this end, we introduce ϵ_θ, an approximator for ϵ given x_t. We can then use μ_θ to sample $x_{t-1} \sim p_\theta(x_{t-1}|x_t)$, according to Equation 2.6.

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) \tag{2.13}$$

$$x_{t-1} = \mu_\theta(x_t, t) + \sigma_t z \qquad z \sim \mathcal{N}(0, \mathbf{I}) \tag{2.14}$$

Our new parameterisation allows us to rewrite the $\mathcal{L}_{t-1}$ in terms of ϵ and x_0.

$$\mathcal{L}_{t-1} = \mathbb{E}_{x_0,\epsilon}\left[w_t \left\|\epsilon - \epsilon_\theta(x_t(x_0,\epsilon), t)\right\|^2\right] \tag{2.15}$$

$$w_t = \frac{\beta_t^2}{2\sigma_t^2\alpha_t(1-\bar{\alpha}_t)}$$

2.1.1.6 Simplified Training Objective $\mathcal{L}_{\text{simple}}$

An alternative, simplified objective discards the term weights used in the loss (Equation 2.9) (specifically w_t within from Equation 2.15), which effectively gives less weight to earlier timesteps where there is less noise [27]. Rather than summing over the timesteps, we introduce $t \sim U(1, T)$. The $t = 1$ case corresponds to $\mathcal{L}_0$ and the $t > 1$ case corresponds to an unweighted version of $\mathcal{L}_{t-1}$ (Equation 2.15). This allows training to focus on more challenging timesteps where there is more noise to remove, leading to better sample quality.

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{t,x_0,\epsilon}\left[\left\|\epsilon - \epsilon_\theta(x_t(x_0,\epsilon), t)\right\|^2\right] \tag{2.16}$$

$\mathcal{L}_{\text{simple}}$ provides no learning signal for $\Sigma_\theta(x_t, t)$, which is not an issue as we have fixed $\Sigma_\theta(x_t, t)$. This is discussed further in Section 2.1.3.2.

2.1.2 Conditioning DDPMs

DDPMs can be adapted to generate images conditionally. It is possible to condition on various data, such as text prompts or class labels. We focus on image conditioning, which enables DDPMs to perform image-to-image translation tasks. The objective is to model the conditional distribution $p(x|y)$ (rather than the marginal $p(x)$), where x is the image we are producing and y is the image we are conditioning on.

Specifically, we are interested in learning $p_\theta(x_0|y)$, which we achieve by feeding the conditioning variable, y, as an additional input during training, so that our noise predictor is now $\epsilon_\theta(x_t, t, y)$. The trained model should then learn to use the additional information in y. We therefore have the following modified objective, based on $\mathcal{L}_{\text{simple}}$ from Equation 2.16.

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{t,x_0,y,\epsilon}\left[\left\|\epsilon - \epsilon_\theta(x_t(x_0,\epsilon), y, t)\right\|^2\right] \tag{2.17}$$

We can perform conditional sampling by adding the condition variable to the sampling processes, modifying Equations 2.13 and 2.14.

$$\mu_\theta(x_t, y, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, y, t)\right) \tag{2.18}$$

$$x_{t-1} = \mu_\theta(x_t, y, t) + \sigma_t z \qquad z \sim \mathcal{N}(0, \mathbf{I}) \tag{2.19}$$

The specific details for conditional DDPMs vary based on the specific task being performed.

2.1.2.1 Colourisation

One of the most common image-to-image translation tasks is colourisation, where, given a greyscale image as an input, a realistically colourised image is output. This is achieved by conditioning image generation on the greyscale image. i.e., y from Equation 2.17 is the greyscale image we wish to colourise. The training dataset provides colour images and their greyscale counterparts. Training is performed by perturbing the colour images according to the training procedure as before, while conditioning on the greyscale images. To sample, we provide a greyscale image as an input to get a colourised output.

2.1.2.2 Image Inpainting

For inpainting, the model outputs a complete image based on a partially masked image. y is therefore a partially obscured image, according to a binary mask, m. During training, we only compute the loss for the masked regions, as we do not need the model to learn to recreate the unmasked regions.[3]

$$\mathcal{L}_{\text{simple}} = m \odot \mathbb{E}_{t,x_0,y,\epsilon}\left[\|\epsilon - \epsilon_\theta(x_t(x_0,\epsilon), y, t)\|^2\right] \tag{2.20}$$

Similarly, during sampling, we do not need the model to change the unmasked regions of the image (which would also waste compute). As such, the initial input we pass to the model, x_T, is a copy of y, where the masked regions replaced with Gaussian noise.

$$x_T = m \odot g + (1-m) \odot y \qquad g \sim \mathcal{N}(0, \mathbf{I}) \tag{2.21}$$

Furthermore, after each sampling iteration, we apply a post-processing step to once again replace the unmasked regions of the image with the unmasked regions of y, in case the model has made any modifications. i.e., the model may attempt to denoise the unmasked regions of the image, which we can discard as we already have the ground truth in these regions.

$$x_{t-1} = m \odot (\mu_\theta(x_t, y, t) + \sigma_t z) + (1-m) \odot y \tag{2.22}$$

2.1.3 Improving DDPMs

2.1.3.1 Denoising Diffusion Implicit Models

DDPMs have slow sampling speeds, which is due to the (iterative) Markovian inference process, as steps have to be performed sequentially. DDIMs address this by utilising a non-iterative process, inspired by the fact that, in DDPMs, the loss ($\mathcal{L}_{\text{simple}}$) only depends on the marginals, $q(x_t|x_0)$, but not the joint probability $q(x_{1:T}|x_0)$ [30]. There are many inference distributions which have the same marginals, including non-Markovian alternatives. We use the distribution family, $q_\sigma \in Q$. Further details are given in Appendix A.4.

$$q_\sigma(x_{1:T}|x_0) = q_\sigma(x_T|x_0)\prod_{t=2}^{T} q_\sigma(x_{t-1}|x_t, x_0) \tag{2.23}$$

$$q_\sigma(x_T|x_0) = \mathcal{N}(\sqrt{\alpha_t}x_0, (1-\alpha_T)\mathbf{I}) \tag{2.24}$$

$$\forall t > 1.\ \ q_\sigma(x_{t-1}|x_t, x_0) = \mathcal{N}(\sqrt{\alpha_{t-1}}x_0 + \sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\frac{x_t - \sqrt{\alpha_t}x_0}{\sqrt{1-\alpha_t}}, \sigma_t^2\mathbf{I}) \tag{2.25}$$

When learned parameters, θ, are not shared across timesteps, we are able to use $\mathcal{L}_{\text{simple}}$ for DDIMs as well [30]. We can therefore apply DDIM sampling to a pre-trained DDPM model.

$$x_{t-1} = \sqrt{\alpha_{t-1}}\underbrace{\left(\frac{x_t - \sqrt{1-\alpha_t}\epsilon_\theta(x_t)}{\sqrt{\alpha_t}}\right)}_{\text{prediction for } x_0} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2\cdot\epsilon_\theta(x_t)}}_{\text{direction pointing to } x_t} + \underbrace{\sigma_t\epsilon_t}_{\text{noise}} \tag{2.26}$$

[3] $\odot$ denotes the Hadamard Product

where $\epsilon_t \sim \mathcal{N}(0, I)$ and $\alpha_0 := 1$. Setting $\sigma_t = 0$ results in a deterministic process, given x_{t-1} and x_0. As $\mathcal{L}_{\text{simple}}$ does not prescribe a specific forward process (with a specific number of forward steps, T) as long as $q_\sigma(x_t|x_0)$ is fixed, we can consider a process with less steps. We define our process on a subset of $x_{1:T}$, $x_{\tau_1:\tau_S}$, where τ is an increasing sub-sequence of $[1, ...T]$ of length S. The generative process produces intermediate samples according to reverse(τ).

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{i=1}^{S} p_\theta(x_{\tau_{i-1}}|x_{\tau_i}) \tag{2.27}$$

$$p_\theta(x_{\tau_{i-1}}|x_{\tau_i}, \tau_i) = q_{\sigma,\tau}(x_{\tau_{i-1}}|x_{\tau_i}, x_{0_\theta}(x_{\tau_{i-1}}, \tau_i)) \text{ if } i \in [S], i > 1 \tag{2.28}$$

$$p_\theta(x_0|x_t, t) = \mathcal{N}(x_{0_\theta}(x_t, t), \sigma_t^2 \mathbf{I}) \text{ otherwise} \tag{2.29}$$

2.1.3.2 Learning Reverse Process Variances

As previously discussed in Section 2.1.1.2, the reverse process variances, $\Sigma_\theta(x_t, t)$ can be learnt, or set to fixed constants, σ_t^2. We chose to use fixed constants as early works found that learning the variances lead to training instability and poorer sample quality [27]. Specifically, they considered setting $\Sigma_\theta(x_t, t)$ to the upper and lower bounds (β_t and $\tilde{\beta}_t$ respectively) on reverse cross-entropy for data with coordinate-wise unit variance. Notably, they found that both choices gave the same performance, which was unexpected given that they correspond to the two extreme choices. This occurs because $\beta_t \approx \tilde{\beta}_t$ except near $t = 0$, where the model is dealing with imperceptible changes [33].

Although the first few steps of the sampling process have little effect on perceived sample quality, they make the greatest contribution to log likelihoods. Stably learning $\Sigma_\theta(x_t, t)$ should therefore give better likelihood performance. This is non-trivial, however, as the reasonable range for the the variances is very small, making it difficult for a neural network to predict, even in the log domain. We therefore parameterise the variance between β_t and $\tilde{\beta}_t$ in the log domain, where v is output by the model.

$$\Sigma_\theta(x_t, t) = \exp(v \log \beta_t + (1 - v) \log \tilde{\beta}_t) \tag{2.30}$$

The simplified training objective, $\mathcal{L}_{\text{simple}}$, is no longer valid as it does not account for $\Sigma_\theta(x_t, t)$. This is because we previously discarded $\mathcal{L}_T$ from the loss, as it was a constant when $\Sigma_\theta(x_t, t)$ was fixed. We must therefore formulate a new loss, where we utilise λ to create an appropriate ratio between $\mathcal{L}_{\text{vlb}}$ (Equation 2.15) and $\mathcal{L}_{\text{simple}}$ (Equation 2.16). We also apply a stop-gradient to $\mu_\theta(x_t, t)$ for the $\mathcal{L}_{\text{vlb}}$ term for the same reason. This leads to a loss where $\mathcal{L}_{\text{vlb}}$ guides $\Sigma_\theta(x_t, t)$, and $\mathcal{L}_{\text{simple}}$ is still the main source of influence over $\mu_\theta(x_t, t)$.

$$\mathcal{L}_{\text{hybrid}} = \mathcal{L}_{\text{simple}} + \lambda \mathcal{L}_{\text{vlb}} \tag{2.31}$$

We denote DDPMs that use this modification as Learned Variance DDPMs (LVDDPMs). LVDDPMs are able to produce high quality samples with 10-100 times fewer iterations at sampling time than they are trained with [33]. Details for fast sampling with LVDDPMS are given in Appendix A.5.

2.1.3.3 Score-Based Generative Modelling through SDEs

Both DDPMs, and an adjacently developed generative modelling technique, Score Matching with Langevin Dynamics (SMLDs) [31], can be generalised as Score-Based Generative Modelling through SDEs (SGMSDEs) [32]. The score is defined as the gradient of the log probability density with respect to data, given as $\nabla_x \log p(x)$.

DDPMs (Section 2.1.1) consist of a forward process which iteratively perturbs data samples over a series of timesteps, and a trained probabilistic model for each timestep to reverse the process. SMLDs estimate scores at each noise scale and then sample from decreasing noise scales using Langevin Dynamics.

SGMSDEs generalise both of these methods, by framing the model as a continuum of distributions that evolve over time according to a diffusion process, $\{x(t)\}^t$, where $t \in [0, T]$ is a continuous time variable. The evolution is given by an Itô SDE that progressively diffuses data points, $x(0) \sim p_0$, till they become random noise, $x(T) \sim p_T$, where p_T is a Gaussian.

$$dx = f(x, t)dt + g(t)dw \tag{2.32}$$

where w is Brownian motion, f models the drift of $x(t)$, and g models the diffusion of $x(t)$. The SDE does not depend on the data and has no trainable parameters. It has a unique (strong) solution when its coefficients are global Lipshitz in state and time [6]. DDPMs and SMLDs are two realisations of the SDE to perform $p_0 \to p_T$.

Reversing the diffusion process allows smooth transformation of random noise from the prior into a sample from the data distribution. The reverse process satisfies a reverse-time SDE [2], which can be derived given the score of the marginal probability densities as a function of time.

$$dx = [f(x, t) - g(t)^2 \nabla \log p_t(x)]dt + g(t)d\bar{w} \tag{2.33}$$

where $\bar{w}$ represents Brownian motion when time flows backwards, dt represents an infinitesimal negative timestep, and $p_t(x)$ denotes the probability density of $x(t)$.

We approximate the reverse time process by training a model, $s_\theta(x, t)$, to estimate marginal scores at each time step, $\nabla \log p_t(x)$. The model is trained on samples with score matching using an objective function that generalises the objectives of DDPMs and SMLDs.

$$\mathcal{L}_{\text{SDE}} := \mathbb{E}_{t,x(0),x(t)|x(0)} \left[\|s_\theta(x(t), t) - \nabla x(t) \log p_{0t}(x(t)|x(0))\|_2^2 \right] \tag{2.34}$$

Given this general framework, we can choose a specific SDE. The noise perturbations used in DDPMs and SMLDs are discretisations of two different SDEs. We consider the perturbation kernel, $\{p_{\alpha_i}(x|x_0)\}_{i=1}^N$, of DDPMs. The discrete Markov chain is given by $x_i = \sqrt{1 - \beta_i} x_{i=1} + \sqrt{\beta_i} z_{i-1}$ for $i = 1, ..., N$, where β_i are the noise scales at each timestep. As $N \to \infty$, this chain converges to an SDE, where $\beta(t)$ is a function indexed by $t \in [0, 1]$.

$$dx = -\frac{1}{2}\beta(t)xdt + \sqrt{\beta(t)}dw \tag{2.35}$$

This yields a process with a fixed variance (of 1) when the initial distribution has unit variance, so it is called a Variance Preserving SDE (VPSDE).

$$p_{0t}(x(t)|x(0)) = \mathcal{N}\left(x(0)e^{-\frac{1}{2}\int_0^t \beta(s)ds}, \mathbf{I} - \mathbf{I}e^{-\frac{1}{2}\int_0^t \beta(s)ds}\right) \tag{2.36}$$

Once the score of each marginal distribution is given by the score-based model, s_θ, for all t, we can derive the reverse-time SDE, and simulate it using numerical methods to sample from p_0. We can also use a Predictor-Corrector strategy instead of standard numerical solvers [32], which incorporates score-based MCMC approaches. i.e., a numerical solver (predictor) provides an estimate of the sample at the next time step, and then the score-based MCMC algorithm (corrector) corrects its marginal distribution. The sampling technique used by DDPMs is equivalent to using Ancestral Sampling as the predictor and the identity function as the corrector.

This SDE based approach also allows us to conditionally produce samples from $p_0(x(0)|y)$ without having to train a new model, if $p_t(y|x(t))$ is known (cf. 4.2.3.3). This allows us to use the same network for conditional and unconditional generation. Given a forward SDE, we can sample from $p_t(x(t)|y)$, starting at $t = T$, by solving the conditional reverse time SDE.

$$dx = [f(x, t) - g(t)^2[\nabla_x \log p_t(x) + \nabla_x \log p_t(y|x)]]dt + g(t)d\bar{w} \tag{2.37}$$

The only missing component is an estimate of the score of the forward process, $\nabla_x \log p_t(y|x(t))$. Rather than training a separate model for this task, the literature suggests that we can estimate it using heuristics and domain knowledge.

2.2 Starting Point

Following my literature review, I inspected the codebases associated with the literature, shown in Table 2.1. Of these codebases, a few of them could hypothetically act as starting points for my work. However, during my inspection, I found that they each have various deficiencies which make them unsuitable for extension.

Codebase Name	Implements
Denoising-Diffusion [27]	DDPMs
Improved-Diffusion [33]	LVDDPMs
Palette [43]	Conditional DDPMs
Implicit-Diffusion [30]	DDIMs
Score-SDEs [32]	SGMSDEs
VPSDE-Solver [37]	DPMSolver

Table 2.1: Relevant Codebases

Denoising-Diffusion, which would be the ideal starting point for this dissertation, is written using TensorFlow, which has since been replaced by PyTorch as the standard ML library in the field. Simply translating the TensorFlow code would be a large enough task that rewriting from scratch is more useful, as it enables refactoring code into a more Pythonic style, with extensibility baked in. *Improved-Diffusion* uses a custom implementation for common evaluation metrics, such as Fréchet Inception Distance and the Inception Score. This introduces additional variability, making comparisons between literature less fair. *Palette*, which is the most relevant work on conditional image generation, has not been made publicly available by the authors. *Score-SDEs* is poorly structured and hence difficult to parse and navigate. Specifically, in an effort to be extensible, the codebase makes extensive use of function pointers and variables that override each other, which makes evaluation difficult to follow. Additionally, there is no support for distributed training, which is required given the computing constraints of my device.

To address these issues, I decided to build my codebase from scratch, rather than modifying an existing project. The key algorithms for the various methods discussed in this project are implemented from scratch. In some cases where entirely re-implementing a helper function or algorithm would not provide any benefit, I adapted the work of an official codebase, e.g., the fast ODE solver from *VPSDE-Solver*. Where this is the case, the code is labelled as such.

In terms of knowledge, I had no prior experience with Diffusion Models, or any deep generative modelling. I had briefly experimented with TensorFlow, but had never used PyTorch before. My relevant knowledge was therefore limited to the small amount I had seen up to Part IB, with IB Data Science and IA Probability proving to be the most applicable. Once the project had started, I picked up more useful skills during my Units of Assessment: Machine Visual Perception (MVP) and Mobile Health (MH). MVP touched on relevant topics, including briefly on Diffusion Methods. Both units allowed me to develop my ML skills in Python.

2.3 Dataset Choices

In this project I utilise two datasets. CIFAR10 (32x32) [10] contains 60,000 images from 10 categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). It is widely used in the generative modelling space, making comparing results with other literature easier. Additionally, training models for a 32x32 dataset is much faster than training for larger image datasets, which is vital given the slow training for DDPMs, and enables faster ablations.

CelebAHQ (256x256) [21] is a dataset of 30,000 images of celebrities' faces. Compared to CIFAR10 and other standard datasets in the generative modelling space, this dataset is high resolution. Compared to other high-res datasets (such as ImageNet [9] or Places2 [20]), this dataset is small, and the images are fairly homogeneous, i.e., they are all images of faces compared to, for example, the 400+ scene categories in Places2. This trade-off between resolution and number of images was key, as it made training on higher resolution images feasible within the time and resource constraints of this project.

2.4 Requirements Analysis and Deliverables

Upon gaining an understanding of the theory behind DDPMs, I devised a set of requirements and deliverables, dividing the project into 4 key tasks (two base tasks and two extensions).

2.4.1 Requirements

These requirements are largely unchanged from my proposal. I measured performance using a slightly different set of evaluation metrics, in order to compare to the latest literature. I also did not utilise semantic segmentation as part of the web-app as it would increase the scope of the project too much.

T1 (Base): Unconditional Generation
- Implement unconditional DDPM model for CIFAR10 (32x32)
- Implement unconditional DDPM model for CelebAHQ (256x256)

T2 (Base): Conditional Generation
- Implement conditional DDPM model to perform colourisation on CIFAR10
- Implement conditional DDPM model to perform inpainting on CelebAHQ

T3 (Extension): Improving Performance
- Implement neural network architecture improvements
- Implement learned variances (LVDDPM)
- Implement DDIM sampling
- Implement SDE generalisation (SGMSDE)
- Compare performance of all models
- Apply and evaluate improvements on conditional tasks

T4 (Extension): Web-Application
- Implement a client web-app to upload and edit images
- Implement a server to receive images and perform inpainting

2.4.2 Deliverables

Table 2.2 shows my project deliverables. These mostly manifest as diffusion models and their associated results, except for the web-app.

Name	Deliverable	Task	Priority	Difficulty
DDPM32	Base CIFAR10	T1	High	Medium
DDPM256	Base CelebAHQ	T1	High	Medium
Colourisation	Experiment with colourisation	T2	High	Medium
Inpainting	Inpainting on CelebAHQ	T2	High	Medium
FSDDPM	Fast Sampling Methods	T3	Medium	High
FSDDPMInpaint	Fast Inpainting	T3	Low	High
Web-App	Web-app for image editing	T4	Medium	Medium

Table 2.2: Project Deliverables

2.5 Redundancies and Backups

My project utilised Git commits for version control. These commits were pushed to GitHub for backup, but also so the code could be pulled on to the HPC (CSD3) and run. This also takes advantage of CSD3's extensive backups. Key training checkpoints of models were also backed up to Google Drive, enabling them to be tested on Google Colab as well.

2.6 Software Development Strategy

To a large extent, the deliverables in this project manifest as additional features and modifications to a base implementation of DDPMs (**T1**). I therefore decided on an Agile software development strategy. This involved weekly sprints with objectives and deliverables, concluded by a stand-up with my supervisor where I reflected on my progress in the week. I was also able to flag any blockers I may have faced. To this end, I utilised Kanban boards, as shown in Figure 2.1, to organise tasks into three categories: *To-do*, *In Progress*, and *Completed*. Each week's sprint aimed to complete the tasks in the *In Progress* section and further populate the *To-do* section. The *To-do* tasks were sorted by a priority, in accordance with Table 2.2.

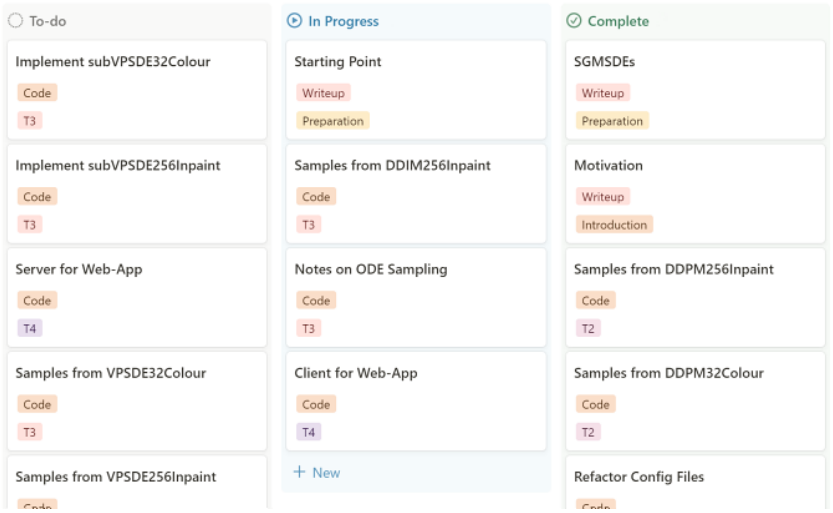


Fig. 2.1: Project Kanban Board

My overall division of time between tasks was based on my project timeline from my proposal. I made changes to the schedule for various reasons. Firstly, the extent of the project grew as I performed my literature review, which highlighted additional work in the field which would be relevant to the project. I also explored additional literature I was directed to by my supervisor. Rescheduling was necessary as model training was slower than expected.

2.7 Languages, Libraries, Tools

2.7.1 Languages

The majority of the code content of my project is for creating, training, and sampling the various models I experimented with. Python was the optimal choice for this, given the variety of deep learning frameworks available. Moreover, Python is the language used all of the papers I am referencing, so using the same language makes my results more comparable. Additionally, using the standard language in the field allows for my work to be extendable by other researchers.

The training and inference for my models was almost entirely performed on CSD3, which uses the SLURM job manager for scheduling. I therefore needed to write scripts to submit jobs. The only option for this tasks was to utilise the SLURM shell script format. These scripts are responsible for requesting hardware resources for an allotted time, establishing the environment, and then calling the code.

The remainder of my code output is written to create a user-facing web-app. This task offers the most flexibility in language choice, with a plethora of options available in web-development.

I used JavaScript on the front-end and wrote the back-end server in PHP. My choice of development stack was primarily dictated by my pre-existing knowledge, which coincided with some of the most commonly used languages for web-development. This means there is an abundance of documentation and support available. I settled on using these languages as a large proportion of my time in this project had already been spent in learning to perform deep learning in Python, and web-development methodologies were not the key focus, so minimising time spent learning for this task seemed sensible.

2.7.2 Libraries

Table 2.3 details the libraries utilised in this project and their associated licenses. I chose to use the PyTorch deep learning library for a variety of reasons, despite early works utilising TensorFlow. More recently, the research community, and especially the generative modelling field, has mostly shifted to using PyTorch, making it the more suitable choice today. Additionally, debugging is more straightforward in PyTorch due to its dynamic computational graphs and general flexibility (compared to TensorFlow and it's static graphs).

TorchVision was utilised for dataset transformations, as it integrates well with PyTorch. I utilised TorchMetrics for computing some of the evaluation metrics, rather than utilising a custom implementation for the reasons outlined in Section 2.2. Mainly, as a standardised implementation enables fairer comparisons. Similarly, I utilise TensorBoard for logging training and evaluation metrics rather than any custom implementation.

The choice of a front-end library for web-development was the only other major library decision. I chose to React for the same reasons I chose to use JavaScript and PHP - it is the front-end library I am most familiar with. Although I have previously used Create React App to create single page web-apps, it is now deprecated, so I used Vite in this project.

Library	**Purpose**	**License**
PyTorch	Deep learning	BSD
TorchVision	Dataset transformations	BSD
TorchMetrics	Evaluation metrics	Apache 2.0
TensorBoard	Logging	Apache 2.0
tqdm	Progress visualisation	MIT
NumPy	Array manipulation	BSD
Pillow (PIL)	Image manipulation	HPND
OpenCV (cv2)	Image manipulation	Apache 2.0
React	Web-app front-end	MIT
Vite	Web development server	MIT

Table 2.3: Metrics Utilised

Tool	**Purpose**
VS Code	Project IDE
CSD3	Training and testing
Colab	Development
GitHub	Git commit backups
Termius	File transfer and ssh
Notion	Project planning
draw.io	Creating diagrams

Table 2.4: Tools Utilised

2.7.3 Tools

Table 2.4 shows all of the tools utilised in this project. VSCode was used to write Python code and SLURM scripts, as well as JavaScript and PHP code for the web-app. I utilised various extensions within the IDE, including for LaTeXcompilation and some version control. Notion was used extensively for note-taking during my literature review, as well for project planning (cf. Figure 2.1). Google Colab was useful for iterating on code faster than possible on CSD3, due to its job queues. All models were trained and evaluated on CSD3. The Research Data Store (RDS) on CSD3 was use to store logging outputs, sample images and model checkpoints.

Chapter 3: Implementation

This chapter explores the algorithms behind DDPMs (Section 3.1). It details the neural network architectures and training processes utilised (Section 3.2). The architecture and implementation of the web-application are covered in Section 3.3. Finally, it provides a codebase listing and the purpose of the included files (Section 3.4). The codebase is made publicly available ***here***

3.1 DDPM Algorithms

3.1.1 Unconditional Image Generation

The algorithms for training and sampling unconditional DDPMs are fundamental as they form the basis later models and their associated algorithms. The training algorithm (Algorithm 1) takes samples from the data distribution (the dataset), samples a random timestep and noise value, then computes a perturbed version of the sample. We then perform gradient descent and backpropagation to learn the noise added. The sampling algorithm (Algorithm 2) takes random noise, and then iterates over the timesteps. On each iteration, it uses the closed form for the reverse process to denoise the sample, utilising the network to predict the noise to remove.

Algorithm 1 Training DDPMs

q is the data distribution
$\beta_t = \text{RANGE}(\beta_{\text{start}}, \beta_{\text{end}}, N), \alpha_t = 1 - \beta_t$
while training **do**
    for x_0 in q **do**
        $t \sim U(1, T), \epsilon \sim \mathcal{N}(0, \mathbf{I})$
        $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$
        $L = ||\epsilon - \epsilon_\theta(x_t, t)||$
        BACKPROP_GRADDESC(L)

Algorithm 2 Sampling from DDPMs

$x_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = 1$ to T **do**
    if $t > 1$ **then**
        $z \sim \mathcal{N}(0, \mathbf{I})$
    else
        $z = 0$
    $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sigma_t z$
return x_0

3.1.2 Conditional Image Generation

Conditional generation requires modifications to Algorithms 1 and 2. We perform extra processing on samples and include conditions as baggage in our computations.

Algorithm 3 Generating Greyscale Images

procedure GREY(x_0)
    for pixel i, j **do**
        $g_R = \frac{299}{1000} x_{0_R} + \frac{587}{1000} x_{0_G} + \frac{114}{1000} x_{0_B}$
        $g_G = g_R$
        $g_B = g_B$
    return g

Algorithm 4 Generating Masked Images

procedure MASKER(x_0)
    $p \sim U(0, 1), noise \sim \mathcal{N}(0, \mathbf{I})$
    if $p >$ threshold **then**
        $mask =$ GETFREEFORMMASK()
    else
        $mask =$ GETREGULARMASK()
    $x_{masked} = mask(x_0) + (1 - mask)(noise)$
    return $x_{masked}, mask$

3.1.2.1 Colourisation

All the datasets utilised in this project provide colour images. In order to train while conditioning on greyscale images, we need to additionally generate the greyscale images, which is handled in Algorithm 3. The algorithm takes colour images as input and generates greyscale images according to the ITU-R 601-2 luma transform [52]. Before outputting, we replicate the greyscale values across three colour channels so that the greyscale images have the same shape as the colour images. This is not mandatory, i.e., we could have a single channel image, and then pass create a network that takes a four channel input. However, this methodology is more transferable to tasks where we condition on colour images (e.g., inpainting).

We modify the unconditional training algorithm (Algorithm 1) according to 2.1.2. We first generate a greyscale version of the sample image by adding $x_{grey} = \textsc{Grey}(x_0)$ before Line 5. We concatenate the sample image and its greyscale version to use as a combined input for noise estimation by adding $x_{comb} = cat(x_t, x_{\text{grey}})$ before Line 7. We then use x_{comb} as to compute the loss instead of x_t on Line 7. This training process allows the estimator to take advantage of the greyscale information.

In order to sample, we modify the unconditional sampling algorithm (Algorithm 2). We have an additional input x_{grey}. After Line 6, we concatenate the colour and greyscale samples using $x_{comb} = cat(x_t, x_{\text{grey}})$. We then use x_{comb} instead of x_t as the input to the noise predictor on Line 7.

3.1.2.2 Inpainting

As with colourisation, we have to define a method for generating conditional information. We want models to be able to perform inpainting on both regular and freeform masks. We therefore set up the masked image generator to randomly choose between applying freeform and regular masks according to some pre-determined hyperparameter. We then generate the masked image by replacing the masked regions of the image with Gaussian noise.

To train, we once again modify Algorithm 1, according to 2.1.2.2. We generate a masked version of the input image and also keep the mask itself by inserting $x_{masked}, mask = \textsc{Masker}(x_0)$ before Line 5. We generate x_t as before, but before combining it with the masked version of x_0, we first replace the unmasked region with the original image by inserting $x_{comb} = cat(x_t * \text{mask} + (1 - mask) * x_0, x_{masked})$ after Line 6. This is because we are only trying to learn to fill the masked region. We then compute the loss, but only consider the masked regions by replacing Line 7 with $L = \|mask * \epsilon - mask * \epsilon_\theta(x_{comb}, t)\|$.

Sampling is performed using a modified version of Algorithm 2. We input a masked image, x_{masked}, and its mask. We combine the masked image and the noisy image to input into the noise predictor by inserting $x_{\text{comb}} = cat(x_t, x_{\text{masked}})$ before Line 7. We only use the prediction for the masked region, replacing the unmasked regions with the input image using the mask by inserting $x_{t-1} = (x_{\text{masked}} * (1 - \text{mask})) + (x_{t-1} * \text{mask})$ after Line 7.

3.1.3 DDIM

To perform DDIM sampling, as per Section 2.1.3.1, we can use a model trained using Algorithm 1. We use the modified sampling algorithm shown in Algorithm 5. If we set $\eta = 1$, we have standard sampling. We instead set $\eta = 0$, for deterministic DDIM sampling.

Algorithm 5 DDIM Sampling

1: $x_T \sim \mathcal{N}(0, \mathbf{I})$
2: $S <= T$
3: τ is an increasing subsequence of $[1...T]$ of length S
4: $\eta = 0$ for DDPM sampling
5: **for** i in $range(1, S)$ **do**
6: $\quad z \sim \mathcal{N}(0, \mathbf{I})$ if $i > 1$ else $z = 0$
7: $\quad \sigma_{\tau_i}(\eta) = \eta\sqrt{\frac{1-\alpha_{\tau_{i-1}}}{1-\alpha_{\tau_i}}}\sqrt{1 - \frac{\alpha_{\tau_i}}{\alpha_{\tau_{i-1}}}}$
8: $\quad x_{t-1}(\eta) = \sqrt{\alpha_{\tau_{i-1}}}\left(\frac{x_{\tau_i} - \sqrt{1-\alpha_{\tau_i}}\epsilon_\theta(x_{\tau_i}, \tau_i)}{\sqrt{\alpha_{\tau_i}}}\right) + \sqrt{1 - \alpha_{\tau_{i-1}} - \sigma_{\tau_i}(\eta)^2} \cdot \epsilon_\theta(x_{\tau_i}, \tau_i) + \sigma_{\tau_i}(\eta)z$
9: return x_0

Algorithm 6 Training for LVDDPMs

1: q is the data distribution
2: $\beta_t = \text{RANGE}(\beta_{\text{start}}, \beta_{\text{end}}, N), \alpha_t = 1 - \beta_t$
3: **while** training **do**
4: $\quad$ **for** x_0 in q **do**
5: $\quad\quad t \sim U(1, T), \epsilon \sim \mathcal{N}(0, \mathbf{I})$
6: $\quad\quad x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$
7: $\quad\quad \hat{\epsilon}, v = \epsilon_\theta(x_t, t)$
8: $\quad\quad \Sigma_\theta = \exp(v \log \beta_t + (1 - v) \log \tilde{\beta}_t)$
9: $\quad\quad \mathcal{L}_{\text{simple}} = \|\epsilon - \hat{\epsilon}\|$
10: $\quad\quad \mathcal{L}_{\text{vlb}} = D_{\text{KL}}[p_\theta(x_{t-1}|x_t)\|q(x_{t-1}|x_t, x_0)]$
11: $\quad\quad$ BACKPROP_GRADDESC($\mathcal{L}_{\text{simple}} + \lambda\mathcal{L}_{\text{vlb}}$)

Algorithm 7 Sampling for SDEs using PC

1: T discretisation steps
2: C corrector steps
3: $x_N \sim p_T(x)$
4: **for** t in $range(T - 1, 0)$ **do**
5: $\quad x_t = \text{PREDICTOR}(x_{t+1})$
6: $\quad$ **for** s in $range(1, C)$ **do**
7: $\quad\quad x_t = \text{CORRECTOR}(x_t, s, t)$
8: return x_0

3.1.4 LVDDPMs

When we want to learn reverse process variances, as per 2.1.3.2, we first modify our network and train it to output a vector, v, which we use to compute variances. We then modify our objective to also be reliant on these variances, by using a weighted sum of $\mathcal{L}_{\text{simple}}$ and $\mathcal{L}_{\text{vlb}}$. We perform gradient descent over this objective. The modified training procedure is shown in Algorithm 6. We can use the same sampling procedure as for standard DDPMs (Algorithm 2).

3.1.5 SGMSDEs

In order to train VPSDEs, as per Section 2.1.3.3, we use a modified version of Algorithm 1. We use a different perturbation kernel, replacing Line 6 with $x_t = x_0 e^{-\frac{1}{2}\int_0^t \beta(s)ds} + z(e^{-\frac{1}{2}\int_0^t \beta(s)ds} + 1)$. Additionally, we use a different learning objective, replacing Line 7 with $\mathcal{L} = \lambda(t)\|s_\theta - \nabla_x(t) \log q_t\|^2$. We no longer need to compete α_t and β_t on Line 2. Sampling can be performed using various techniques. To utilise Predictor-Corrector methods, we use Algorithm 7. For each discretised timestep, we generate a prediction for the next time-step's sample, and then perform some number of corrector steps.

3.2 Neural Networks

3.2.1 General UNet Architecture

The architecture used to denoise images at each timestep is based on the original UNet architecture [14] with modifications made for image generation and diffusion models [22] [27]. In my implementation of the neural network, I employed a modular approach to enable extensions and modifications to the architecture. This also maximised code re-use, which is useful for UNets, as they apply the same layers repeatedly at different resolutions. There are two key modules in the architecture. The first is the Residual Block (Figure 3.1), which performs convolutions and combines inputs. A key component is the skip connection, which allows the time embedding input to bypass the layers within the block and directly contribute to the output. This connection can either be taken as is (using an identity layer) or after applying a convolution.

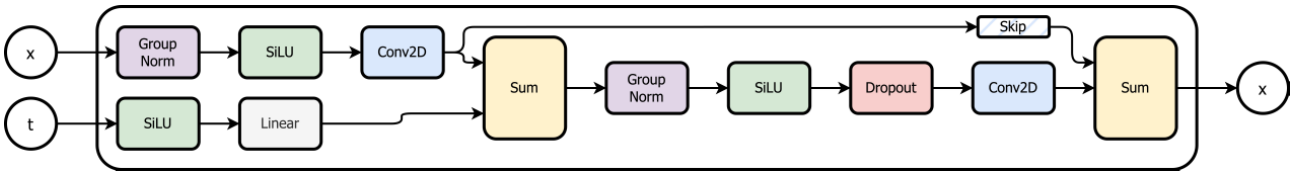


Fig. 3.1: ResBlock

The second key module is the Attention Block shown in Figure 3.2, which employs spatial self-attention. It once again utilises a skip connection.

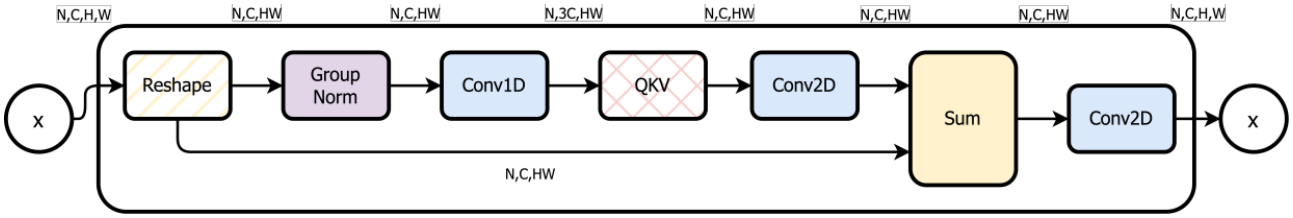


Fig. 3.2: AttnBlock

Additionally, the architecture utilises an EmbedBlock, shown in Figure 3.3, for the timestep input. This is responsible for converting the time input to the correct number of channels.

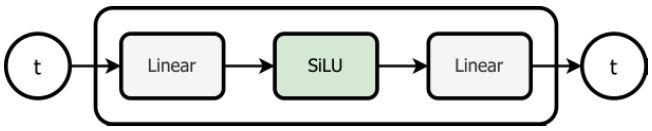


Fig. 3.3: EmbedBlock

Figure 3.4 provides a generalised view of the UNet architecture that can be specialised for different image resolutions. The ResAttn blocks refer to ResBlocks, optionally followed by AttnBlocks. The resolution levels vary based on the resolution of input images and the network configuration. The number of ResAttn blocks at each resolution varies between network configurations.

The architecture is divided into three key sections. The downsampling section reduces the image resolution. This induces an information bottleneck in the central section. After this, the upsampling section increases the resolution back to that of the input. The outputs from each ResAttn block, Downsample block and the initial Conv2D on the downsampling half of the architecture are forwarded to the ResAttn blocks in the upsampling section. These long distance (LD) connections are necessary for the upsampling section as information is lost in the downsampling section. Specifically, these LD inputs are summed with the input to the block

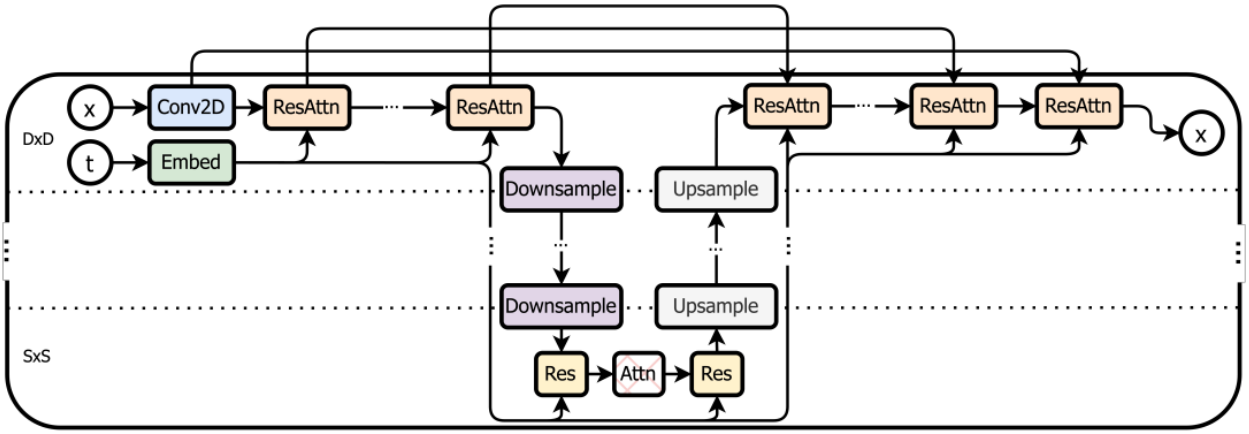


Fig. 3.4: General UNet Architecture

before performing the other operations. The upsampling section contains one more ResAttn block than in the downsampling section at the same resolution to incorporate LD connections from downsampling blocks in the downsampling section.

3.2.2 DDPM32 UNet Architecture

To demonstrate how the generalised architecture detailed in Section 3.2.1 can be applied, this section details the DDPM32 network. This model employs a network architecture with 4 resolution levels. Attention is employed at the 16x16 resolution level. The brackets in the ResBlocks indicate whether their skip connections utilise a convolution or the identity function.

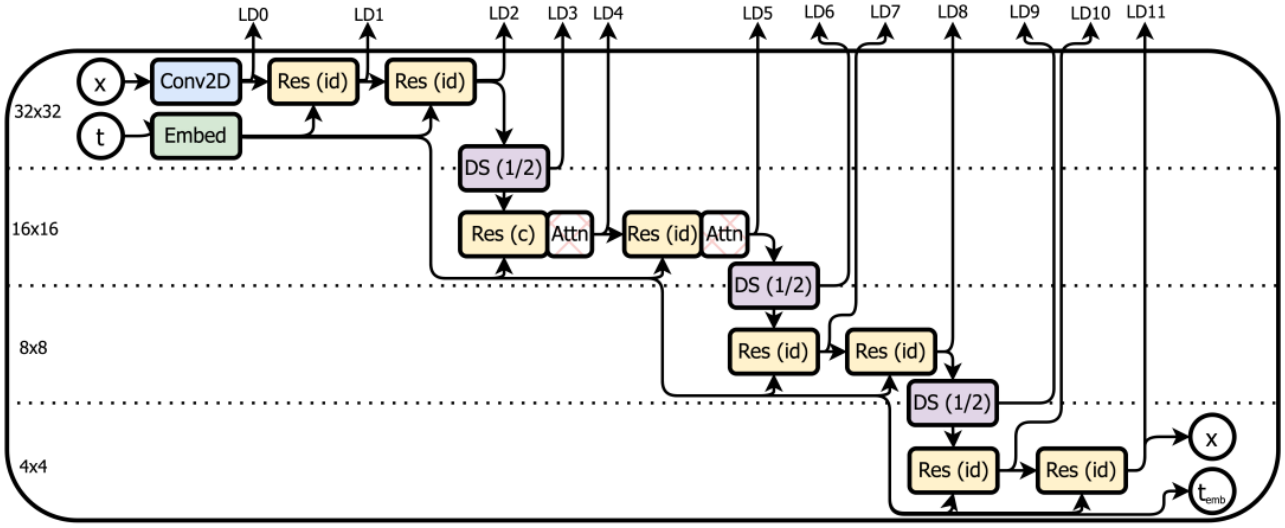


Fig. 3.5: DDPM32 UNet downsampling section

Figure 3.5 shows the full downsampling section. Each of the three downsampling blocks half the image resolution, transforming the images from 32x32 to 4x4. This section employs 2 ResAttn blocks at each resolution. The bottleneck, shown in Figure 3.6, passes the image through a ResBlock, an AttnBlock, and another ResBlock. The upsampling section is shown in Figure 3.7. Each of the ResAttn blocks takes a long distance input from the upsampling section. As the downsampling section employs two ResAttn blocks at each resolution level, the upsampling section utilises three.

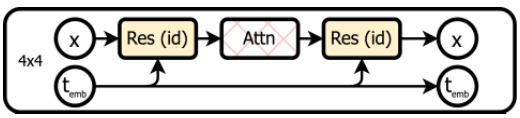


Fig. 3.6: DDPM32 UNet bottleneck

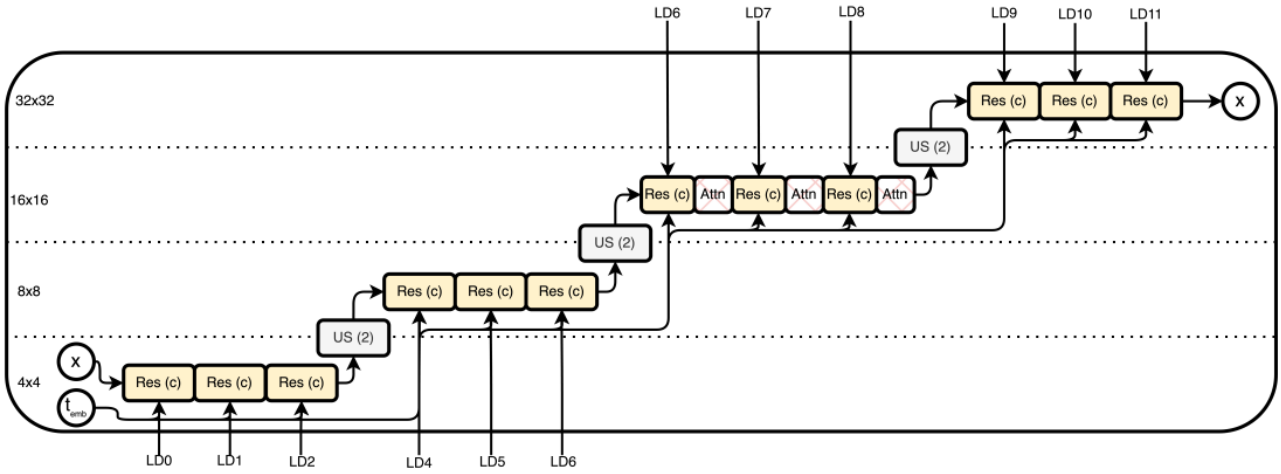


Fig. 3.7: DDPM32 UNet upsampling section

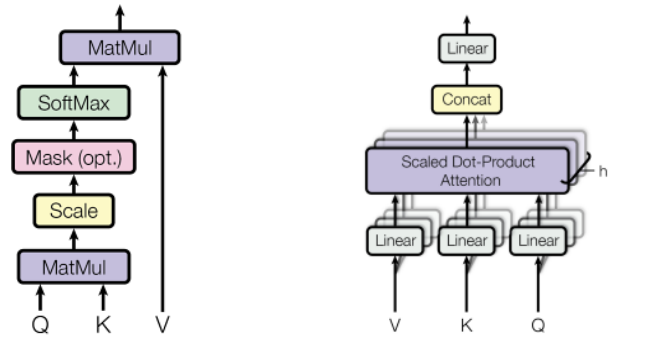
3.2.3 Modifications for Conditional Generation

The network architecture for conditional generation is largely unchanged. Theoretically, the model takes an additional input y as extra information to condition on. In practice, this is achieved by concatenating x and y together and passing this as the singular input to the model. x and y are both tensors with the shape $[N, C, H, W]$, where N is the batch size, C is the number of channels (3), and H and W are the height and width of the images respectively. The concatenation happens in the channel dimension so that the input to the model is now of the shape $[N, 2C, H, W]$. Accordingly, the key modification we make to the network for conditional generation is to modify the number of input channels. The output channels remains the same. This is conveniently managed by setting a condition in the config file (cf. Appendix B.1).

3.2.4 Improvements to the Network Architecture

I implemented various architecture modifications described in the literature to provide better performance or enable the later extensions.

Scaled Dot-Product (SDP) Attention can be replaced with Multi-Head Attention, which consists of several SDP Attention layers running in parallel, as shown in Figure 3.8. In practice, I implemented this by reshaping the input to the attention block according to the number of heads h, and then once again reshaping the output back to the original shape, as shown in Figure 3.9.



(a) SDP Attention (b) Multi-Head Attention

Fig. 3.8: Comparison of Attention Methods [23]

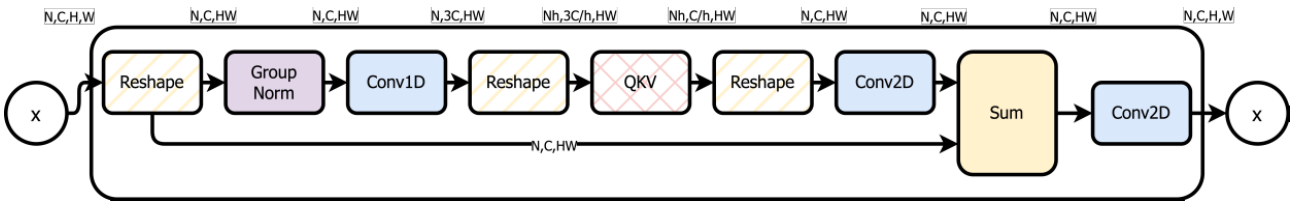


Fig. 3.9: Multi-Head Attention Block Implementation

Performance can often be improved by increasing the depth of the network, which can be achieved by increasing the number of ResAttn Blocks utilised at each resolution level. For example, some of the LVDDPMs utilise 3 or 4 ResAttn blocks per level, where some of the VPSDEs utilise 8. A deeper network decreases the speed of the network, which affects both training and sampling speeds.

Another modification is to the use of conditional time input, t. In the original architecture, we compute a conditioning vector c and inject it into hidden state h as GroupNorm$(h + c)$. In our modified architecture, we compute conditioning vectors w, which scales, and b, which shifts, and then inject them as GroupNorm$(h)(w+1)+b$. We call this modified implementation the SSResBlock (Figure 3.10) (as we use the embedding to (S)cale and (S)hift).

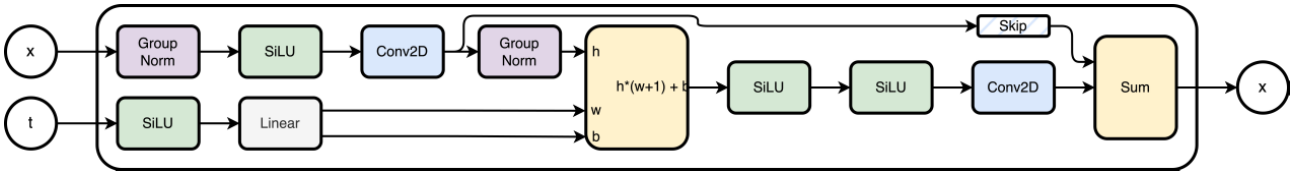


Fig. 3.10: Scale-Shift ResBlock (SSResBlock)

Performance can be improved by utilising convolutional resampling. Specifically, the Upsample and Downsample blocks increase and reduce (respectively) the resolution of the image by passing it through a Conv2D block, rather than utilising 2D average pooling.

3.2.5 Modifications for LVDDPMs

In order to learn reverse process variances, as discussed in Section 2.1.3.2, we utilise a vector v that needs to be output by the model. Specifically, v requires a component for each image dimension. We therefore modify the network architecture to output double the number of channels, so that that output now has $2C$ channels. This output is split into two, where the first half is considered the denoised image sample, and the second half is v.

3.2.6 Architecture Specifications

Table 3.1 describes the network architectures used by the models from this dissertation. c_i and c_o describe the number of input and output channels, respectively. **Resolution Levels** describes the resolution levels of the network architecture. **#RA** gives the number of ResAttnBlocks used at each resolution level. **Attn** describes the resolution levels at which attention is applied, and in case of multi-head attention, the number of heads (in brackets). **Conv** describes whether resampling uses convolutions rather than average pooling.

Model	c_i	c_o	Resolution Levels	#RA	Attn	Conv
DDPM32	3	3	32, 16, 8, 4	2	16	False
DDPM256	3	3	256, 256, 128, 64, 16, 4	2	16	False
DDPM32Colour	6	3	32, 16, 8, 4	2	16	False
DDPM256Inpaint	6	3	256, 256, 128, 64, 16, 4	2	16	False
LVDDPM32Colour	6	6	32, 16, 8, 4	3	16, 8 (x4)	True
DDPM256LVInpaint	6	6	256, 256, 128, 64, 16, 4	3	16	True
VPSDE32Colour	6	3	32, 16, 8, 4	8	16	True
VPSDE256Inpaint	6	3	256, 256, 128, 64, 32, 16, 8	2	16	True

Table 3.1: Model Architecture Specifications

3.2.7 Distributed Training

Diffusion models are intensive to train in terms of time and computational resources. It was therefore necessary to train models on CSD3 with multiple GPUs in parallel due to time constraints. All models were trained on an Ampere node with 4x NVIDIA A100 80GB GPUs.

PyTorch provides an API for distributed processing, which includes a CLI command called torchrun. This command is used as a wrapper for running Python files. It initialises the environment variables describing the multi-device environment and then runs the training script on each device (GPU). The model training program reads these environment variables to set up the network each device.

The training loop itself also requires a parallelised strategy. There are a few alternative approaches, I used PyTorch's Data Distributed Parallel (DDP). This module provides a decorator for a PyTorch network (i.e., the UNet), and then handles the synchronisation of network parameters (weights and biases). This effectively splits each batch between the GPUs, and then combining the gradients when performing back-propagation on the loss. Similarly, evaluation using TorchMetrics used the same setup to distribute the workload between GPUs. Validation samples that are generated during training made use of the multiple GPUs, but when I generated samples after training to measure sampling speed, I utilised a single GPU to minimise variability between runs.

3.2.8 Logging

Metric	Purpose
Loss	Training loss should reduce with epochs
MSE	All models use MSE as a component in their loss. This should reduce with epochs.
VLB	Models that learn reverse process variances incorporate VLB into their loss. This should reduce with epochs.
GradNorm	GradNorm should reduce with epochs, as each epoch performs finer tuning than the last.
FID	FID scores should increase with epochs.
Inception Score	Inception scores should increase with epochs, plateauing at the number of classes in the dataset.
Image Samples	Image samples generated by the models should visually improve in quality with epochs.

Table 3.2: Metrics logged at each training epoch

To ensure that model training was effective, I logged various metrics at each epoch, shown in 3.2. All logging was performed using TensorBoard, which provides a dashboard to visualise the logged values. It was also important to save checkpoints of the models throughout training. Training models often took longer than the maximum available time for a single job on CSD3 (36 hours). I therefore implemented a system that saves a model checkpoint every epoch, but only retains the last two (deleting any older checkpoints). The training program also automatically detects and restores checkpoints so that, on the completion of a CSD3 job, another could be started to resume training.

Retaining the last two checkpoints added a level of redundancy so that any issues with saving a particular checkpoint were not a problem. In case a job timed out while saving a checkpoint, I could delete the last (partial) checkpoint before resuming the process. I also retained checkpoints at larger intervals (every 250 epochs) so that any older checkpoints which achieved higher evaluation metrics could be recovered, i.e., in the case of overfitting.

3.3 Web-Application

3.3.1 Architecture

Figure 3.11 details the architecture of the web-app. It consists of a React-based front-end that runs on an npm httpserver. The user interacts with the system entirely through this front-end. The image is selected from the file-system and passed to the npm httpserver. The application then makes a fetch request to the server through the REST API endpoint exposed by the PHP server. The server calls the inpainting model, and returns its output in the response to the fetch call made by the front-end. The front-end displays the response to the fetch call to the user.

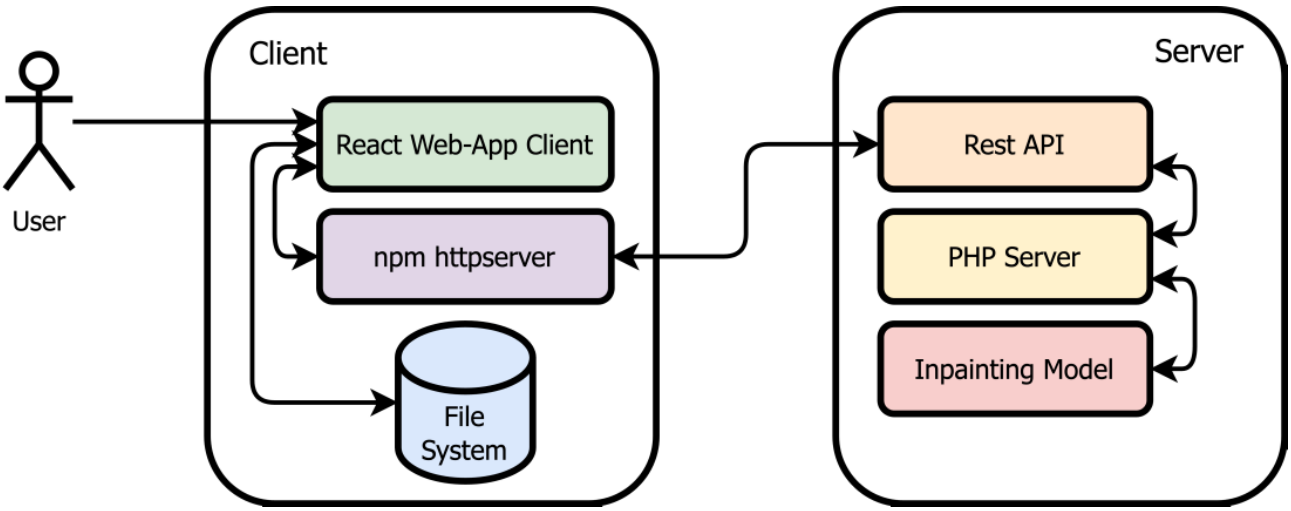


Fig. 3.11: Web-Application Architecture

3.3.2 Data Flow

Figure 3.12 shows the data flow through the web-app. The input image is taken from the client device file system and passed to the web-app. The web-app sends this image and the user-generated mask to the PHP server. The PHP server passes these two as inputs to the inpainting model, which generates an output. This output is passed back from the server to the client. The user is able to save this output to their file system.

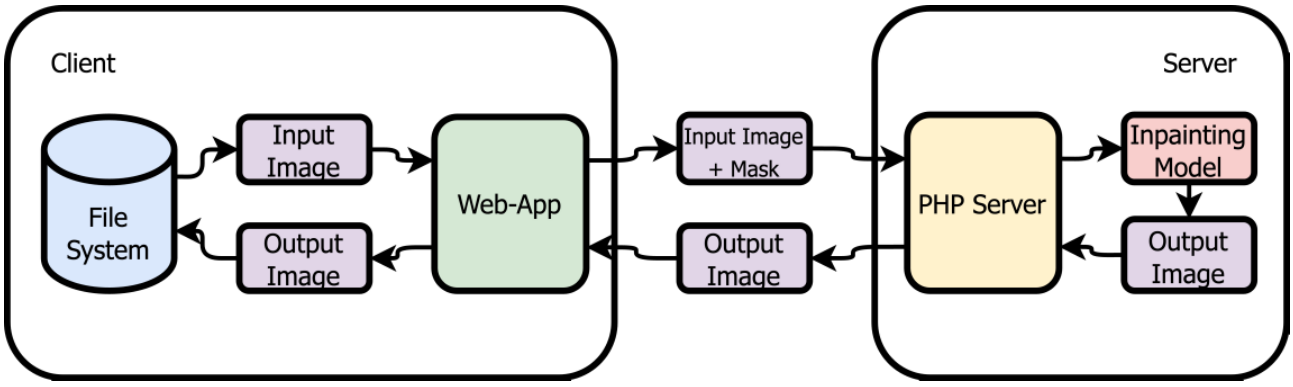
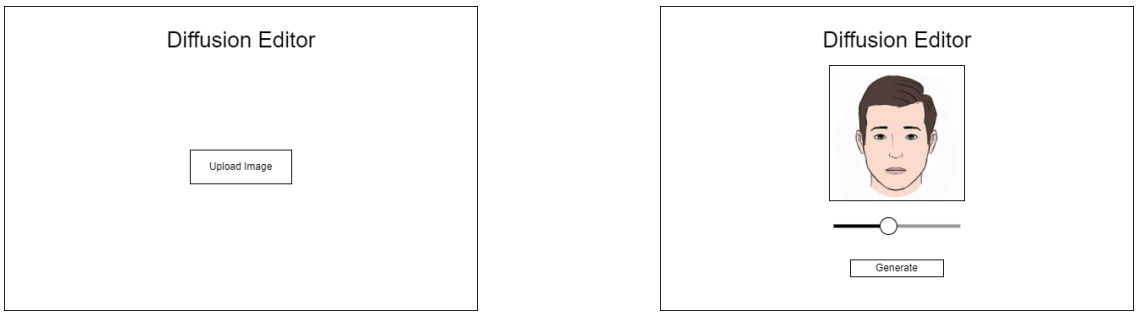


Fig. 3.12: Data Flow

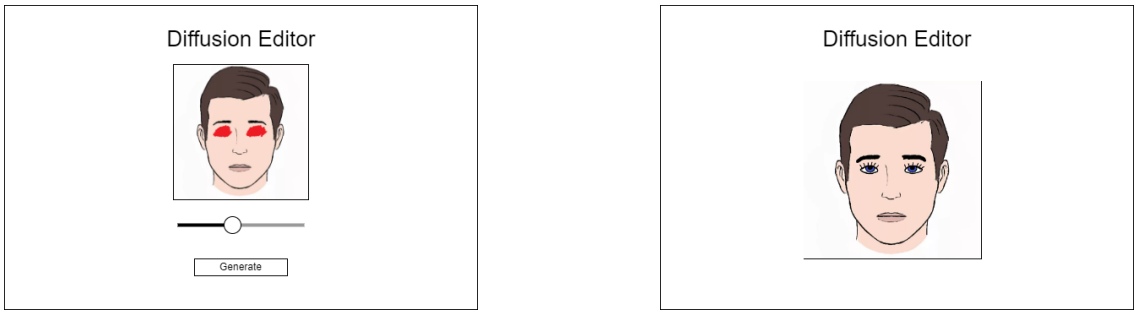
3.3.3 UI Design

Figures 3.13 and 3.14 show the plans for the UI flow. The user is presented with the option to upload an image, which triggers the system file picker. The image they choose is then displayed, along with an option to change the brush size. The user is able to draw a mask over the image. When the user clicks the generate button, the application generates an inpainted version, which is displayed to the user.



(a) Choose a file from the file-system (b) The brush size can be set

Fig. 3.13: Web App UI - Upload



(a) The user can draw a mask (b) The generated output is shown

Fig. 3.14: Web App UI - Generate

3.4 Codebase

Figure 3.15 shows the codebase listing for my project. The code is structured modularly, which made incorporating extensions in the project seamless. e.g., new models or networks can be added. Referring to the project requirements (Section 2.4.1), **T2** and **T3** both build on the algorithms from **T1**, as seen in Section 3.1. A modular codebase also enables a maximal amount of code reuse. The project involves evaluating many models, on multiple datasets, by submitting CSD3 jobs using SLURM scripts. Utilising a set of configuration files (stored in configs) which compile the modular codebase according to a set of parameters enables switching between DDPM models, network architectures, datasets, conditions and hyperparameters from a central location. An example config file is given in Appendix B.1. The SLURM scripts for different experiments can then simply utilise different configuration files. A key module is dist_util .py, which establishes the distributed environment for training, sampling and evaluation.

```
diffusion ..... model training, evaluation and sampling
├── core
│   ├── models
│   │   ├── sdes
│   │   │   ├── predictors
│   │   │   │   ├── predictor.py ..... parent class for predictor algorithms
│   │   │   │   ├── ancestral.py ..... implementation of ancestral sampling
│   │   │   │   ├── euler_maruyama.py ..... implementation of euler-maruyama
│   │   │   │   └── reverse_diffusion.py ..... implementation of reverse diffusion
│   │   │   ├── correctors
│   │   │   │   ├── corrector.py ..... parent class for corrector algorithms
│   │   │   │   └── langevin.py ..... implementation of langevin dynamics
│   │   │   ├── sde.py ..... parent class for sgmsde models
│   │   │   ├── vpsde.py ..... vpsde implementation
│   │   │   ├── dpm_solver.py ..... fast ode solver [53]
│   │   │   ├── RK45_colour.py ..... adaptation of RK45 for colourisation [37]
│   │   │   └── RK45_inpaint.py ..... adaptation of RK45 for inpainting [37]
│   │   ├── ddpm.py ..... ddpm/ddim/lvddpm implementation
│   │   └── model_util.py ..... utils for building models
│   ├── networks
│   │   ├── unet.py ..... unet implementation
│   │   └── blocks.py ..... blocks for building unet
│   ├── dataset.py ..... creates datasets and dataloaders
│   ├── dist_util.py ..... handles distributed processing
│   ├── eval_util.py ..... util for performing evaluation
│   ├── logger.py ..... logs to stdout and tensorboard
│   └── train_util.py ..... util for performing training
├── datasets
├── logs
├── chkpts
├── samples
├── scripts
│   ├── train.py ..... creates and trains models
│   ├── generate.py ..... generates image samples
│   └── evaluate.py ..... evaluates models
└── configs ..... example given in Appendix B.1
webApp
├── diffusionServer
│   ├── inputs ..... stores uploaded images and masks
│   ├── outputs ..... stores model outputs
│   ├── upload.php ..... web-server code
│   └── run_diffusion.py ..... runs diffusion model on inputs
└── ediffuser
    ├── src
    │   ├── index.css
    │   └── index.jsx ..... web-app code
    └── index.html
```

Fig. 3.15: Directory Listing

Chapter 4: Evaluation

This chapter first details the metrics used to evaluate the diffusion models explored in this dissertation, and then provides the results and samples for the various models. It explains the reasons for the results, and provides detailed comparisons between models. This is followed by an exploration into the application of inpainting for image editing, and the web-application I created - EDIFFUSER. Finally, there is a discussion on the ethical implications of diffusion models and generative modelling as a while.

4.1 Evaluation Metrics

4.1.1 Fréchet Inception Distance

Fréchet Inception Distance (FID - ↓) [16] is a measure of sample quality, and therefore acts as one of the most important metrics. It is the standard for papers in the generative modelling field, so vital for making comparisons with other works. It is given by $FID = |\mu - \mu_w| + tr(\Sigma + \Sigma_w - 2(\Sigma\Sigma_w)^{1/2})$. Here, μ and Σ are the mean and covariance of the multivariate normal distribution estimated from features detected by Inception v3 [16] applied to real images (i.e., from the data distribution). $\mathcal{N}(\mu_w, \Sigma_w)$ is the same but for images generated by the model. The 2048 feature layer is used, as it is the standard amongst the literature.

4.1.2 Inception Score

Inception Score (IS - ↑) is a measure of sample diversity, given by $IS = \exp(\mathbb{E}_x[D_{\mathrm{KL}}(p(y|x)||p(y))])$ [19]. This is informative given that a notable advantage of Diffusion Methods over GANs is their ability to generate more diverse samples. The conditional and marginal distributions are calculated from features extracted from the images, once again, using Inception v3.

4.1.3 Negative Log Likelihood

Negative Log Likelihood (NLL - ↓) measures the joint probability (likelihood) of a set of samples. For diffusion models, exact likelihood computation is intractable, so we provide the lower bound. A notable exception is when we utilise the probability flow ODE framing for SGMSDEs, where we are able to exactly compute the NLL. NLL is reported in bits/dimension.

4.1.4 Sampling Speed

Sampling speed (↓) is the mean sampling time required to generate 64 samples for CIFAR10 and 36 samples for CelebAHQ. Specifically, I sampled 10 batches from the models and recorded how long each batch took to generate. I took the mean and standard deviation. The standard deviation is reported when it exceeds 1s. Speed is reported in seconds.

4.1.5 Sampling Iterations / Number of Function Evaluations

Sample iterations (T - ↓), or the number of function evaluations (NFE) in the case of SGMSDEs, represents the number of sequential iterations performed by a model to generate a sample. This is an additional metric for sampling speed that is device agnostic. i.e., faster computers will be able to perform each iteration faster, but will still be limited by the sequential operations.

4.2 Results

4.2.1 Training Convergence

For all training, I recorded various metrics, as described in Section 3.2.8. This allowed me to ensure training was effective and over-fitting was not occurring, which is identifiable by increasing FID scores, despite reducing training loss. Shown in Table 4.1 are the plots for DDPM32, where both the training loss and FID score continually decrease, before converging.

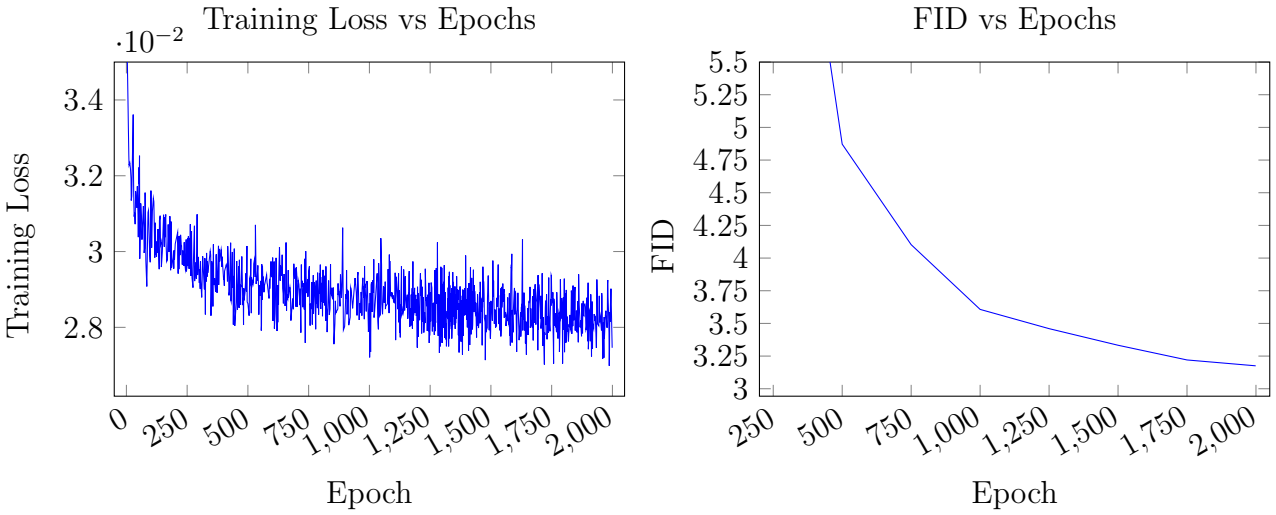


Fig. 4.1: Training plots for DDPM32.

4.2.2 CIFAR10 Unconditional Image Generation

This section provides the results for various model architectures and sampling techniques to perform unconditional CIFAR10 generation. These models act as the base for later experiments.

4.2.2.1 DDPM32

DDPM32 successfully matches the results for unconditional generation of images from CIFAR10 achieved in the state of the art from 2020 [27]. This was an important first step to build modifications and improvements upon. Table 4.1 shows how these results compare to others in the field. It shows that DDPMs are able to achieve excellent FID scores but suffer with log likelihood and have slow sampling speeds. See Figure 4.2 for image samples. The low FID score translates to high quality image samples. 95% Confidence intervals for Inception Scores are reported for my results. Also included are the intervals for results from other literature where available. NLL is not given for most models as it is not reported in the associated papers.

Model	**IS**	**FID**	**NLL**	**Speed**
Gated PixelCNN	4.60	65.93	3.03	
PixelIQN	5.29	49.46		
EBM	6.78	38.2		
NCSN	8.87±.12	25.32		
SNGAN	8.22	21.7		
SNGAN-DDLS	9.09	15.42		
NCSNv2	8.40±.07	10.87		
StyleGAN2 + ADA	9.74	3.26		
[27]	9.46±.11	3.17	3.75	22
DDPM32	9.47±.10	3.175	3.747	21

Table 4.1: DDPM32 compared to other methods

Fig. 4.2: DDPM32 Sample

4.2.2.2 DDIM32

Using the DDPM32 model, I performed DDIM sampling with a range of iteration counts to study the trade-off between sample quality and sample speed. The results are shown in Table 4.2. While the FID scores are worse with DDIM sampling, they are still acceptable. On the other hand the improvement in sampling speed is dramatic, making interactive applications possible.

T	IS	FID	NLL	Speed
1000	9.47	3.175	3.747	21
100	8.91	5.741	4.943	4
50	8.77	7.136	5.529	3
25	8.51	9.504	6.171	2

Table 4.2: DDIM32 metrics

Model	T	β Schedule	Loss
LVDDPM32LinSim	4000	Linear	$\mathcal{L}_{\text{simple}}$
LVDDPM32Sim	4000	Cosine	$\mathcal{L}_{\text{simple}}$
LVDDPM321k	1000	Cosine	$\mathcal{L}_{\text{hybrid}}$
LVDDPM32	4000	Cosine	$\mathcal{L}_{\text{hybrid}}$

Table 4.3: LVDDPM32 Configurations

4.2.2.3 LVDDPM32

I experimented with various configurations that utilised an improved network architecture (with Multi-Head Attention and Scale-Shift ResBlocks - see Section 3.2.4). These models selectively apply various improvements, highlighting the effects of each modification through the differences in performance between them. For each of the configurations, I also experiment with reducing the number of iterations used at inference time. The details of the ablations are shown in Table 4.3. Specifically, it shows the three key differences between these modified models and DDPM32: the number of training iterations, the β schedule, and the loss utilised.

T	IS	FID	NLL	Speed
4000	9.30	3.190	3.399	146
100	8.56	46.94	5.057	4
50	5.92	89.80	5.667	3
25	2.97	170.4	6.334	2

Table 4.4: LVDDPM32LinSim metrics

T	IS	FID	NLL	Speed
4000	9.37	3.064	3.256	143
100	9.65	25.46	4.622	4
50	8.20	52.61	5.159	2
25	5.01	108.3	5.797	2

Table 4.5: LVDDPM32Sim metrics

LVDDPM32LinSim (results in Table 4.4) is effectively the same as DDPM32, but with an improved architecture and more sampling steps. Despire the literature claiming that the improvements in log likelihood stem from utilising a new β schedule [33], this model still achieves better log likelihoods than DDPM32. This suggests that the improvements can at least partially be attributed to simply utilising a better model architecture and more training iterations. LVDDPM32Sim (results in Table 4.5) builds upon LVDDPM32LinSim, additionally incorporating the Cosine β schedule. As expected, this results in a further improvement in log likelihoods, as well as a significant improvement in FID scores. In fact, this model, even without utilising $\mathcal{L}_{\text{hybrid}}$, is able to achieve better NLL and FID scores than DDPM32.

For both LVDDPM32LinSim and LVDDPM32Sim, attempting to reduce the number of sampling iterations results in very poor performance. This suggests that learning the reverse process variances and utilising the new hybrid loss, $\mathcal{L}_{\text{hybrid}}$ is necessary for fast sampling with these models. As we are interested in utilising these models in interactive applications, we will not consider these two models for inpainting.

LVDDPM321k (results in Table 4.6) applies the Cosine β schedule and learns reverse process variances, but does not utilise additional training (or sampling) iterations as recommended

by literature, which results in faster training and sampling. As a result, while we can see improvements in log likelihoods due to the modified β schedule, they are not as significant. Additionally, the FID score suffers with the reduced sampling iterations. This is likely because additionally learning the reverse process variances requires more iterations. LVDDPM32 (results in 4.7), applies all the improvements and is therefore able to achieve the best FID score and log likelihoods so far. It has a slower sampling speed than DDPM32 due to the extra learned parameters and additional training/sampling iterations.

T	IS	FID	NLL	Speed
1000	9.36	3.544	3.316	36
100	9.38	3.800	4.078	4
50	9.23	4.832	5.199	2
25	8.91	7.621	8.658	2

Table 4.6: LVDDPM321k

T	IS	FID	NLL	Speed
4000	9.40	3.056	3.209	146
100	9.38	3.626	5.264	4
50	9.27	4.958	8.390	3
25	8.85	9.492	17.44	2

Table 4.7: LVDDPM32

Fig. 4.3: LVDDPM32 Sample using 25 steps

Both models, as a result of learning reverse process variances, are able to use far fewer iterations at sampling time and still achieve competitive metrics. Surprisingly, LVDDPM321k achieves better log likelihoods than LVDDPM32 when using reduced sampling iterations. Overall, LVDDPM32 achieves the best NLL and FID scores at 100 sampling iterations so far.

4.2.2.4 VPSDE32

Table 4.8 shows the results for VPSDE32, when sampling with Euler-Maruyama as the predictor and the identity function as the corrector. We perform one correction step at the end of the sampling process.

NFE	IS	FID	NLL	Speed
1000	9.637	2.446	2.991	67

Table 4.8: VPSDE32

VPSDE32 achieves the best NLL and FID scores so far, improving over both DDPM32 and LVDDPM32, but is far slower than the corresponding DDPM models. We therefore explore various faster sampling methods that utilise the corresponding ODE. Adding the corrector step greatly improves sample quality, and is therefore applied to all sampling methods. The various ODE solvers utilised are controlled by error tolerance values, rather than some fixed NFE. Tolerance values (tol) were chosen to give comparable sampling iteration counts.

Tol	NFE	IS	FID	NLL	Speed
1e-3	80	9.444	2.827	3.146	17
1e-2	62	9.686	2.886	3.831	14
2e-2	50	10.09	14.56	4.727	13

Table 4.9: VPSDE32 - ODE (RK45)

Tol	NFE	IS	FID	NLL	Speed
1e-3	62	9.547	2.897	3.443	15
1e-2	41	9.438	4.481	3.715	13
2e-2	35	9.484	4.899	3.977	10

Table 4.10: VPSDE32 - ODE (RK23)

RK45 (results shown in Table 4.9) is a black-box ODE solver that utilises the Runge-Kutta method with order 4(5). This is the black-box solver tested and recommended by the literature

[32]. Although 50 function evaluations does not appear to be sufficient for the sampling to successfully converge, once we exceed 62 iterations, we achieve excellent FID scores. At 80 function evaluations, we achieve the best combination of iterations, NLL and FID score so far.

RK23 (results shown in Table 4.10) is similar to RK45 but utilises the Runge-Kutta method with order 2(3). Surprisingly, the solver was able to converge using far fewer function evaluations. RK23 is able to deliver lower FID scores when using fewer function evaluations, but it is not able to deliver as competitive log likelihoods. Comparing the solvers at the same number of function evaluations (62), RK45 is still able to achieve both better NLL and FID scores.

Tol	NFE	IS	FID	NLL	Speed
1e-3	120	9.526	2.798	3.262	20
1e-2	62	9.453	3.660	3.740	15
2e-2	50	9.639	20.85	4.960	14

Table 4.11: VPSDE32 - ODE (DOP853)

Tol	NFE	IS	FID	NLL	Speed
1e-2	48	9.496	2.818	3.360	14
5e-2	36	9.517	2.945	3.385	12
1e-1	27	9.579	3.053	3.505	10

Table 4.12: VPSDE32 - ODE (DPMSolver)

DOP853 (results shown in Table 4.11) is the 8th order Runge-Kutta method. As a higher order method, this solver requires more function evaluations, failing to meaningfully converge at 50, and even 62 iterations. At 120 function evaluations, however, DOP853 delivers the best FID score of all the fast evaluation methods.

Finally, DPMSolver (results shown in Table 4.12) is a custom ODE solver developed specifically for VPSDEs [37]. It delivers the best balance of NFE and FID/NLL of all methods considered (cf. Figure 4.4). It also requires far fewer function evaluations to converge, for example, achieving better scores than DDPM32 in all evaluation metrics with just 27 function evaluations.

4.2.2.5 Fast Sampling Comparison

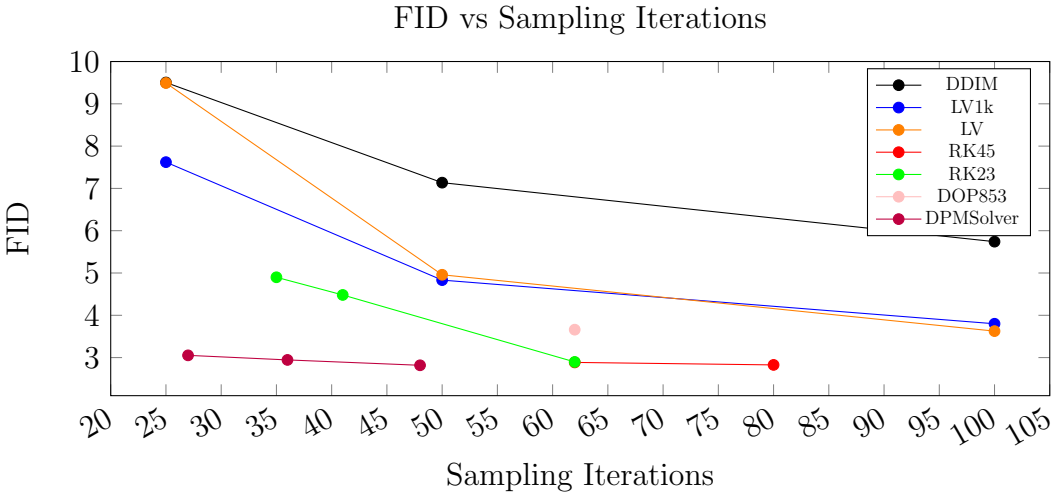


Fig. 4.4: Comparison of FID scores for various fast sampling techniques

Figure 4.4 provides a comparison of the various fast sampling techniques considered for unconditionally generating images from CIFAR10. It highlights the superior performance of VPSDEs, and specifically DPMSolver, compared to traditional DDPM32 and LVDDPM32. When considering the black-box ODE solvers, the higher order solvers typically require more function evaluations to achieve competitive metrics, but typically achieve better metrics. However, if we refer to the time to generate samples, these models are far slowr than LVDDPMs and DDIM sampling. The LVDDPMs achieve better metrics than DDIM across the range of sampling iterations considered.

4.2.3 CIFAR10 Colourisation

I performed experiments with colourisation on CIFAR-10 to provide insights into the performance of the various models for conditional tasks, before experimenting with CelebAHQ. This was helpful as CIFAR10 models are able to train much faster than CelebAHQ models. I did not explore inpainting on CIFAR10 as the images are too small (32x32).

4.2.3.1 DDPM32Colour and DDIM32Colour

Shown in Table 4.13 are the results for the base colourisation model for images from CIFAR10. It achieves excellent FID, NLL and IS scores, showing that the extra information provided by conditioning on a greyscale image makes realistic image generation a far easier task, which is a good indication for the performance to be expected for inpainting. Additionally, performing DDIM sampling allows us to generate samples faster, with a similar speed increase to that of unconditional CIFAR10 generation. However, a key difference is that the absolute FID, IS and NLL scores are much lower across the board, which corresponds to very high quality samples with just 25 sampling iterations.

T	IS	FID	NLL	Speed
1000	10.94	0.618	2.589	23
100	10.70	1.461	4.290	3
50	10.55	1.785	5.013	2
25	10.56	2.080	5.768	1

Table 4.13: DDIM32Colour

Model	T	β	Loss
LVLinSimColour	4000	Linear	$\mathcal{L}_{\text{simple}}$
LV1kColour	1000	Cosine	$\mathcal{L}_{\text{hybrid}}$
LVSimColour	4000	Cosine	$\mathcal{L}_{\text{simple}}$
LVColour	4000	Cosine	$\mathcal{L}_{\text{hybrid}}$

Table 4.14: LVDDPM32Colour

4.2.3.2 LVDDPM32Colour

I experimented with the same 4 advanced model configurations to test whether the improvements found with unconditional generation translate to conditional tasks like colourisation. The first two models (Appendix C.1), do not use the hybrid learning objective, and therefore do not learn the reverse process variances,. They deliver better log likelihood scores than DDPM32Colour, but worse FID scores and neither of the models is able to effectively perform faster sampling.

T	IS	FID	NLL	Speed
1000	10.77	0.764	1.760	36
100	10.66	1.154	2.551	4
50	10.68	1.458	3.285	2
25	10.58	1.848	4.954	2

Table 4.15: LVDDPM321kColour

T	IS	FID	NLL	Speed
4000	10.84	0.602	1.669	145
100	10.75	1.148	3.893	4
50	10.60	1.461	7.038	2
25	10.60	1.804	13.31	2

Table 4.16: LVDDPM32Colour

LVDDPM321kColour (4.15) delivers a further improvement in NLL, and achieves it much quicker as a result of only utilising 1000 sampling steps. However, it achieves the highest FID score of the four models. LVDDPM32Colour (4.16), is able to achieve the best FID scores and log likelihoods of all the models so far. Both of these models are able to generate samples using fewer iterations. Once again LVDDPM321kColour delivers better log likelihoods than LVDDPM32Colour in such cases.

(a) Input (b) Output (25 steps) (c) Output (50 steps)

Fig. 4.5: LVDDPMColour

4.2.3.3 VPSDE32Colour

Method	NFE	IS	FID	NLL	Speed
Euler-Maruyama	1000	10.26	6.308	3.916	70
Probability Flow	1000	9.48	7.427	2.133	77
DPMSolver	26	8.92	9.832	2.453	12

Table 4.17: VPSDE32Colour

(a) SDE (b) ODE (c) DPMSolver

Fig. 4.6: VPSDE32 Samples (using input from Figure 4.5a)

The results for the Euler-Maruyama PC sampling, Probability Flow ODE and DPMSolver sampling with VPSDE32Colour are shown in Table 4.17. These results are significantly worse than the previous models, despite the impressive results previously achieved by the VPSDE32 model for unconditional generation. This is likely because we are repurposing the unconditional model for conditional generation using a heuristic. Given this fact, the results are still impressive. Additionally, despite the NFE values matching the number of sampling iterations utilised in the LVDDPM32Colour model, the actual sampling times are much slower, which makes real-time/interactive usage a challenge. This suggests that the performance of the unconditional VPSDE256 model for inpainting will similarly be poorer than the performance of the corresponding LVDDPM, which is specifically trained for conditional generation.

4.2.4 CelebAHQ Unconditional Image Generation

Performing unconditional generation experiments with CelebAHQ enabled me to test whether insights on performance are able to transfer to larger datasets, before training inpainting models.

4.2.4.1 DDPM256

The results for DDPM256 are shown in the first row of Table 4.18. The model is able to achieve results approaching the state of the art from 2021 (cf. Table 4.20). In general, the absolute FID scores are higher for CelebAHQ than for CIFAR10 as we are dealing with larger images. Qualitatively, these scores correspond to very high quality samples, as seen in Figure 4.7.

4.2.4.2 DDIM256

The results achieved when utilising DDIM sampling are shown in Table 4.18. The sampling speeds are higher than for CIFAR10, but still a vast reduction over the standard DDPM sampling algorithm.

Fig. 4.7: DDPM256 Sample

T	IS	FID	NLL	Speed
1000	2.89	6.713	1.97	463
100	3.15	13.05	3.18	48
50	3.17	13.67	4.88	24
25	3.10	17.21	9.06	13

Table 4.18: DDIM256 Results

T	IS	FID	NLL	Speed
1000	2.91	6.409	1.571	483
100	2.86	7.205	2.781	49
75	2.80	8.385	3.184	37
50	2.81	9.174	3.872	25
25	2.78	13.92	6.943	13

Table 4.19: DDPM256LV

4.2.4.3 LVDDPM256

As the LVDDPM configuration that utilises a Cosine β schedule and the $\mathcal{L}_{\text{hybrid}}$ was found to be the most performant in Sections 4.2.2.1 and 4.2.3, this is the model configuration we proceed with for CelebAHQ. The results for LVDDPM256 are shown in Table 4.19. This model improves upon DDPM256 in IS, NLL and FID scores, but is slower to sample from. Based on previous results, this suggests that a VPSDE model would be far too slow. Notably, however, the fast sampling methods on LVDDPM256 have the same sampling speeds as the corresponding sampling iterations with DDIM256, while achieving better evaluation metrics. The sample shown in Figure 4.8b is produced using just 50 sampling iterations, taking 20x less time than the sample in Figure 4.7.

Model	FID
StyleSwin (2022)	3.25
WaveDiff (2022)	5.94
UNCSN++ (2021)	7.16
LSGM (2021)	7.22
DDGAN (2021)	7.64
VQGAN (2020)	10.2
DDPM256	6.71
LVDDPM256	6.41

Table 4.20: LVDDPM256 vs Recent Literature

(a) LVDDPM256 Sample (25 steps)

(b) LVDDPM256 Sample (50 steps)

4.2.5 CelebAHQ Inpainting

This section provides the results for inpainting, as well as some sampling images. It first explores the results of the DDPM256Inpaint model and DDIM256Inpaint model, which achieve excellent results and high quality samples. Following this is the results for LVDDPM256Inpaint model, which achieves the state of the art results for inpainting on CelebAHQ. Results for VPSDE256Inpaint are not included as the unconditional model, VPSDE256, is not able to achieve a similar level of performance for conditional tasks, as explained in Section 4.2.3.3. Additionally, despite sampling iterations being on par with LVDDPM256Inpaint, VPSDE256Inpaint takes much longer to sample from.

4.2.5.1 DDPM256Inpaint and DDIM256Inpaint

The results for DDPM256Inpaint and DDIM256Inpaint are shown in Table 4.21. The FID score achieved for image inpainting on CelebAHQ is the state of the art. The DDIM sampling results are equally impressive, achieving excellent FID scores with just 25 samples and in just 12s. A sample is shown in Figure 4.9. However, utilising less sampling iterations leads to significantly worse log likelihoods.

(a) Input

(b) Output

Fig. 4.9: DDIM256Inpaint using 25 Steps

T	IS	FID	NLL	Speed
1000	3.59	0.248	5.102	464
100	3.57	0.301	11.52	47
50	3.56	0.342	24.09	24
25	2.95	0.508	62.12	12

Table 4.21: DDIM256Inpaint

T	IS	FID	NLL	Speed
1000	3.551	0.236	4.772	469
100	3.498	0.284	5.868	49
75	3.435	0.305	6.211	37
50	3.561	0.322	6.797	25
25	3.412	0.476	8.522	13

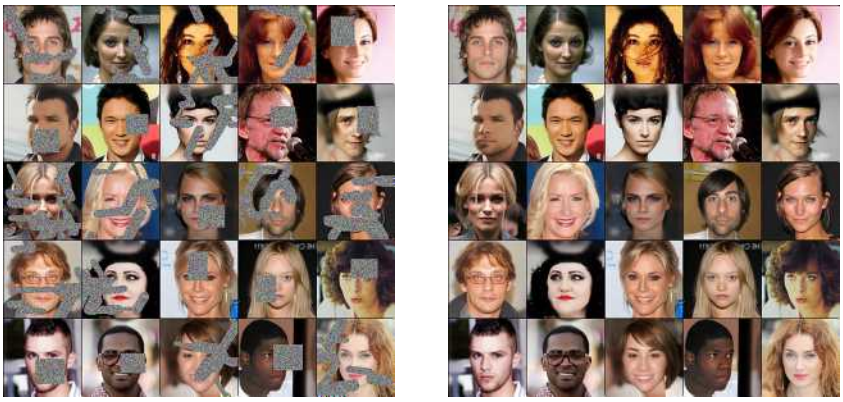
Table 4.22: LVDDPM256Inpaint

4.2.5.2 LVDDPM256Inpaint

The results for the LVDDPM256Inpaint are shown in Table 4.22. This model further improves upon the previous results, **achieving the state of the art FID scores for inpainting on CelebAHQ**. Furthermore, the fast sampling results using the LVDDPM have far more competitive log likelihoods than with DDIM sampling. Figure 4.10 shows a sample generated using 25 iterations in 13s.

Method	FID
MADF (2021)	10.43
AOT GAN (2021)	9.64
DeepFill v2 (2019)	5.69
EdgeConnect (2019)	5.24
LaMa (2021)	3.98
MAT (2022)	2.94
LVDDPM	0.236
LVDDPM (25)	0.476

Table 4.23: Inpainting Comparison



(a) Input (b) Output

Fig. 4.10: LVDDPM256Inpaint using 25 Sampling Iterations

4.2.6 Inpainting for Image Editing

This section explores how suitable inpainting is for performing editing images. The training procedure utilised for inpainting makes the models compatible with freeform and regular masks. The freeform masks are particularly applicable to image editing, which often requires very specific portions of the image to be replaced. Images are successfully generated, even when large portions of the image are masked as the training procedure allows masks to cover 15-50% of images.

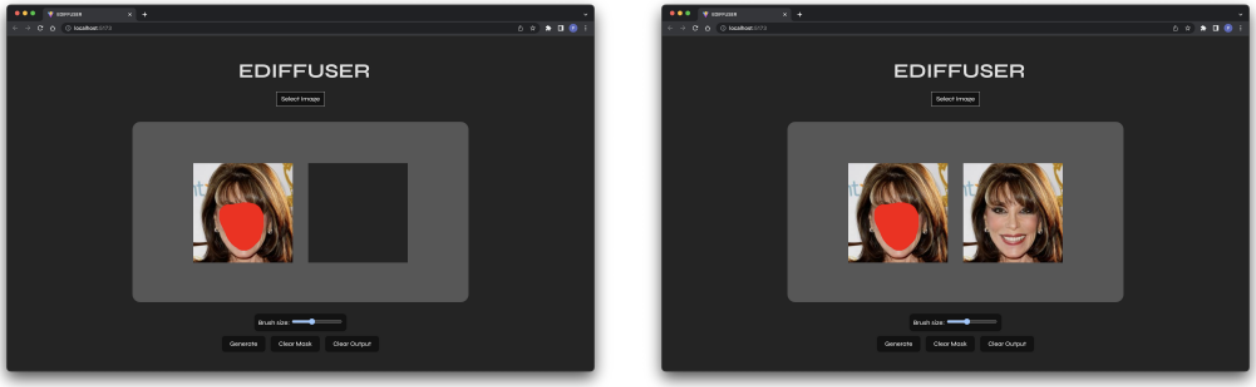
Figure 4.11 shows an example of how inpainting can be used for image editing. The input image contains a face with glasses, and we want to remove the glasses. Masking the region with the glasses and passing the image as an input to an inpainting model results in an output without glasses as desired. The output image is still realistic and the generated regions are consistent with the rest of the image, to the extent that the image is difficult to distinguish from a real picture.

(a) Input (b) Mask glasses region (c) Output

Fig. 4.11: Editing an image

4.2.7 Web-Application

Shown in Figures 4.12 shows screenshots from the web-application. See Appendix C.2 or the video provided with the codebase for a full UI walkthrough. The app utilises a more modern UI design than planned in Section 3.3, with stylised fonts, buttons and other UI elements. It also incorporates additional functionality. For instance, the input and output images are shown side by side as opposed to on separate pages, allowing the user to compare them. The user is able to redraw the mask and regenerate outputs as desired, using the buttons at the bottom. This functionality of the application is possible because DDPMs can generate multiple diverse outputs for a single input (cf. Figure 1.1), as the user can utilise the same mask and select generate again. The web-app also incorporates a loading indicator, which is useful to show the user that processing is occurring, while the diffusion model generates an output.



(a) User can set brush size and draw a mask

(b) Output is shown

Fig. 4.12: The Web-Application

4.3 Ethical Implications

4.3.1 Explainability

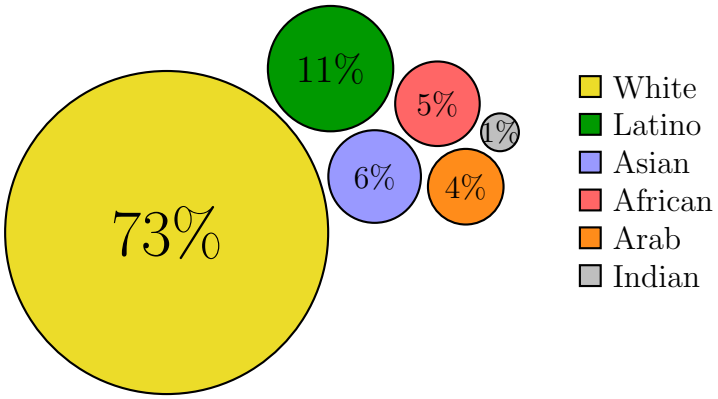
DDPMs, and the various modifications explored in this dissertation, are examples of black box models. i.e., while we are able to control the data, algorithms and hyperparameters used, we are unable to understand how the model combines the input information to make decisions. With rapid improvements in the abilities of such deep learning models to generate realistic samples, it is vital to consider the implications of the lack of explainability.

The combination of DDPM's stochastic and black-box natures means that we are not able to justify the generation of specific samples. Furthermore, deep generative models learn a representation of a target data distribution, but we are only able to understand the learnt representation through sampling. This means we cannot definitively know what images generated by the models may contain, and whether they contain sensitive content.

4.3.2 Learnt Biases

One of the aspects that we can control, the training data, has a massive impact on the model output. Deep generative models learn the biases present in the training data, so it is vital that we carefully select datasets to minimise harmful biases. As such generative models gain ubiquity, even seemingly insignificant biases can be amplified, as the decisions and outputs of models are compounded.

As a case study, we can specifically consider our results on CelebAHQ. There are some immediate concerns that arise when we deal with a dataset of faces. We should consider why specific faces are being generated, and what facial features are prevalent amongst the generated images. For example, it can be harmful if the model disproportionately produces faces of people of specific races, or, when producing faces of people from specific races, it disproportionately utilises specific facial features. When utilising DDPMs for inpainting, there are further considerations, such as which features are used to replace masked regions of faces.



Shown in Figure 4.13 is a breakdown of the demographics of CelebAHQ [47]. Notably, these demographic statistics are generated using other ML models [34], which is a prime example of the aforementioned compounding effect of biases with black-box models.

Fig. 4.13: Demographics of CelebAHQ

It is evident that the vast majority of the people in the dataset are White. This heavy imbalance appears to be amplified in the outputs of the various models. i.e., the models generate an even more extreme distribution of races. We can refer to a sample from DDPM256 (Figure 4.14a), where approximately 96% of the generated peoples' faces are White.

Shown in Figure 4.14b are inpainting samples generated by a (partially trained) model when provided an Black person's face as an input. These samples highlight the fact that, due to the dataset bias, diffusion models tend to inpaint the faces of less represented races in the dataset with features learnt from the majority class. Although this is addressed as models are trained for longer, the issue still exists, and disproportionately affects the minority classes in any case.

(a) Demographics of samples

(b) Bias in inpainting

Fig. 4.14: Diffusion Model Biases

4.3.3 Harmful Uses

Another consideration is the potential harmful uses of DDPMs, and deep generative modelling as a whole. In this dissertation, I explored conditional generation and techniques to improve sampling speed, which are vital for making DDPMs useful for practical purposes, but also enable malicious usage. At the turn of the century, with modes of communication rapidly evolving, researchers and analysts concluded that "political systems in most liberal democracies

[were] facing momentus changes ... that raise serious challenges" [4]. Since then, however, the prevalence of digital media and technology as a means for communication has only grown, and at an ever-increasing rate. The dissemination of information has become near instantaneous, regardless of its source or reliability.

Social media has a significant impact on public perceptions of individuals, organisations and ideas. For example, articles, headlines and images are often rapidly distributed, swaying opinions on celebrities [11]. Beyond this, information can have more serious and widespread impacts, such as the supposed COVID cures and vaccination myths that were spread during the pandemic [45]. One of the most commonly cited examples of the use of social media to manipulate the public is in political campaigns, where Instagram, Facebook and Twitter have become the new battle ground for opposing parties [29]. Misinformation has become a major factor, with third parties, and the candidates themselves, utilising the wildfire nature of social media to gain points in the polls [42].

The power of all the aforementioned uses and misuses of social media are amplified with the ability to rapidly generate "fake" images using text and image conditions. The efficacy of images vs text has been widely studied and stated, with images shown to make misinformation more believable and spreadable [26].

4.3.4 Mitigations

It is vital to consider approaches to tackle the aforementioned issues with diffusion models before making such tools widely available. The lack of explainability is somewhat inherent to deep generative models, though work has been done to address this, including attempting to disentangle latent representations within models [25]. However, specific studies into the explainability of diffusion models have not yet been conducted. Control over the output is best exercised by controlling the input. Filtering sensitive images out of the training dataset, especially the internet is used as a source, can help. Additionally, although this does not address the lack of explainability, the possibility of sensitive content being generated can be reduced by incorporating human and AI assisted output monitoring, as a layer between the model and the user.

The most obvious approach to avoiding learning dataset biases is to use a balanced dataset. This can be difficult in practice based on availability, and datasets popularised within the research the community. One tactic to address this is to filter datasets in order to achieve more balanced demographics. If the dataset is still somewhat biased, additional weight can be placed on learning samples from minority classes to promote a fairer distribution. Additionally, once the model has been trained, its bias can be measured and corrected for. When evaluating performance, it is important to consider whether models are able to deliver acceptable results across a diverse range of inputs. For a facial dataset like CelebAHQ, this could include testing that the model is able to produce acceptable samples of faces for all races.

Harmful uses can also be addressed by many of the aforementioned methods, especially dataset filtering, which can minimise potential harmful output. Additionally, restrictions on input text and image prompts can help to prevent harmful outputs. Restricting the volume of generation is also an effective method for making spreading misinformation harder, as misinformation tactics often involve flooding the internet with a consistent message [38]. Social media tools should incorporate flags for AI generated content, similar to generic misinformation flagging. This can help users incorporate a healthy scepticism as and when appropriate. AI-generated content detectors are already able make accurate classifications [54].

Chapter 5: Conclusion

This chapter reflects on the successes and shortcomings of this project, discussing the key takeaways and learnings. It also explores some potential future directions.

5.1 Achievements

Overall, this dissertation has been very successful. The most notable outcome was an advanced DDPM model that achieves **state of the art (SOTA) results for image inpainting**, and **an image editor web-app** that utilises it with fast sampling techniques. This was based on DDPMs implemented from scratch in a modular codebase designed for extension (the code is made publicly available here). Overall, I achieved all my project requirements, completing my base tasks and incorporating additional extensions based on the latest literature. Listed below are some of the key achievements of this project.

- Successfully implemented unconditional generation from scratch using DDPMs. Also implemented DDIM sampling, LVDDPMs and SGMSDEs.
- Provided a detailed comparison of various fast sampling techniques for unconditional image generation, including previously untested black-box ODE solvers (RK23, DOP853), which surpass the current standard (RK45) in some cases.
- Successfully implemented conditional generation for colourisation and inpainting from scratch using DDPMs. Also implemented conditional DDIM sampling and LVDDPMs, which have not previously been explored for conditional tasks. Achieved SOTA results for image inpainting using LVDDPMs.
- Combined fast sampling techniques and conditional generation to enable the use of diffusion models within interactive applications.
- Discovered and explained the shortcomings of SGMSDEs for conditional tasks, when unconditional models are repurposed for conditional tasks.
- Implemented a web-application that utilises the inpainting model created, and the fast sampling techniques explored in this project.

5.2 Challenges and Lessons Learnt

Diffusion models are complex, and utilising them required a solid understanding of their probabilistic fundamentals. This includes stochastic processes, differential equations and reversing Markov chains. I therefore now have a detailed understanding of a rapidly developing field that is becoming more and more prevalent with the current generative AI boom.

Much of the learning involved reading the latest literature being published by the research community, which allowed me to gain an appreciation for how researchers continually build on each others' ideas to deliver advancements in the field.

I dealt with more practical computing challenges. The computational requirements for diffusion models meant that I had to learn how distributed computing is applied to machine learning, which included multi-GPU training and inference. This also resulted in my first interactions with a job scheduler in CSD3.

While this project was primarily focused on researching deep learning techniques, I also incorporated a practical realisation of my work, in the form a web-app. This allowed me to learn how theoretical ideas are applied in practical, user-facing applications.

5.3 Future Directions

In terms of immediate next steps, it would be insightful to apply the models designed in the project to higher resolution datasets, with a wider diversity of images. Research has been conducted into how diffusion models scale, but there has not yet been a detailed study for conditional tasks. To perform diffusion on higher resolution datasets, we could utilise super resolution [49]. Alternatively, the task could be made more feasible by using a hybrid VAE+DDPM. In such an architecture, the latent space of the VAE can provide a good representation on which to perform diffusion [39]. Additionally, this dissertation discovered that DPMSolver, the latest ODE solver for SGMSDEs, is excellent for unconditional tasks, but not able to deliver sufficient performance for conditional tasks. This was likely because the model is not originally trained for conditional tasks. I therefore hypothesise that utilising the SGMSDE framework with task-specific training could solve these issues and deliver state-of-the-art performance for conditional tasks.

The potential for further developments is vast. Diffusion models are being applied to more and more generative tasks, following their success with images. For example, video [36] and 3D [50]. As such, many of the key learning from this dissertation could be applied in these other fields, such as my exploration into fast sampling techniques or conditional generation. Inpainting specifically has massive utility in a variety of media. For example, it could be used to reconstruct missing data in 3D scene representations or repair old, damaged video.

I am also particularly interested in exploring guidance in diffusion [35], as a logical next step for my application of inpainting in image editing. For instance, in Figure 4.11, where the person's glasses were edited away, we were constrained to the uncontrolled outputs of the model. Incorporating text-based guidance would allow us instruct the model to, for example, add sunglasses instead. Additionally, such guidance could incorporate the latest research on Large Language Models (LLMs), such as GPT4 [51], to allow natural language text prompts to be used to perform image edits. Some similar ideas are already being pursued by the research community [48]. I would also like to implement my original idea of utilising image segmentation to automate the creation of image masks for editing. i.e., allow the user to simply select an object to remove from an image, and automate the mask creation.

References

[1] William Feller. "On the Theory of Stochastic Processes, with Particular Reference to Applications". In: 1949.

[2] Brian D.O. Anderson. "Reverse-time diffusion equation models". In: *Stochastic Processes and their Applications* 12.3 (1982), pp. 313–326. URL: https://EconPapers.repec.org/RePEc:eee:spapps:v:12:y:1982:i:3:p:313-326.

[3] J. M. Deutch and I. Oppenheim. "The Lennard-Jones Lecture. The concept of Brownian motion in modern statistical mechanics". In: *Faraday Discuss. Chem. Soc.* 83 (0 1987), pp. 1–20. DOI: 10.1039/DC9878300001. URL: http://dx.doi.org/10.1039/DC9878300001.

[4] Gianpietro Mazzoleni and WINFRIED SCHULZ. ""Mediatization" of Politics: A Challenge for Democracy?" In: *Political Communication* 16.3 (1999), pp. 247–261. DOI: 10.1080/105846099198613.

[5] Marcelo Bertalmio et al. "Image Inpainting". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 417–424. ISBN: 1581132085. DOI: 10.1145/344779.344972. URL: https://doi.org/10.1145/344779.344972.

[6] B. Øksendal. *Stochastic Differential Equations: An Introduction with Applications*. Universitext (1979). Springer, 2003. ISBN: 9783540047582. URL: https://books.google.co.uk/books?id=VgQDWyihxKYC.

[7] James Hays and Alexei A Efros. "Scene Completion Using Millions of Photographs". In: *ACM Transactions on Graphics (SIGGRAPH 2007)* 26.3 (2007).

[8] Connelly Barnes et al. "PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing". In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28.3 (Aug. 2009).

[9] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[10] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: (2009), pp. 32–33. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[11] Hellmueller. *Media and celebrity: production and consumption of "well-knownness"*. 2010.

[12] Kaiming He and Jian Sun. "Statistics of Patch Offsets for Image Completion". In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 16–29.

[13] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[14] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[15] Jascha Sohl-Dickstein et al. *Deep Unsupervised Learning using Nonequilibrium Thermodynamics*. 2015. DOI: 10.48550/ARXIV.1503.03585. URL: https://arxiv.org/abs/1503.03585.

[16] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[17] Aäron van den Oord et al. “WaveNet: A Generative Model for Raw Audio”. In: *CoRR* abs/1609.03499 (2016). arXiv: 1609.03499. URL: http://arxiv.org/abs/1609.03499.

[18] Danilo Jimenez Rezende and Shakir Mohamed. *Variational Inference with Normalizing Flows*. 2016. arXiv: 1505.05770 [stat.ML].

[19] Tim Salimans et al. “Improved Techniques for Training GANs”. In: *CoRR* abs/1606.03498 (2016). arXiv: 1606.03498. URL: http://arxiv.org/abs/1606.03498.

[20] Bolei Zhou et al. *Places: An Image Database for Deep Scene Understanding*. 2016. arXiv: 1610.02055 [cs.CV].

[21] Tero Karras et al. “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. In: *CoRR* abs/1710.10196 (2017). arXiv: 1710.10196. URL: http://arxiv.org/abs/1710.10196.

[22] Tim Salimans et al. “PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications”. In: *CoRR* abs/1701.05517 (2017). arXiv: 1701.05517. URL: http://arxiv.org/abs/1701.05517.

[23] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[24] Diederik P. Kingma and Max Welling. “An Introduction to Variational Autoencoders”. In: *CoRR* abs/1906.02691 (2019). arXiv: 1906.02691. URL: http://arxiv.org/abs/1906.02691.

[25] Emile Mathieu et al. *Disentangling Disentanglement in Variational Autoencoders*. 2019. arXiv: 1812.02833 [stat.ML].

[26] Michael Hameleers et al. “A Picture Paints a Thousand Lies? The Effects and Mechanisms of Multimodal Disinformation and Rebuttals Disseminated via Social Media”. In: *Political Communication* 37.2 (2020), pp. 281–301.

[27] Jonathan Ho, Ajay Jain, and Pieter Abbeel. *Denoising Diffusion Probabilistic Models*. 2020. DOI: 10.48550/ARXIV.2006.11239. URL: https://arxiv.org/abs/2006.11239.

[28] Hongyu Liu et al. *Rethinking Image Inpainting via a Mutual Encoder-Decoder with Feature Equalizations*. 2020. arXiv: 2007.06929 [cs.CV].

[29] Nunziato. *Misinformation Mayhem: Social Media Platforms’ Efforts to Combat Medical and Political Misinformation*. 2020.

[30] Jiaming Song, Chenlin Meng, and Stefano Ermon. *Denoising Diffusion Implicit Models*. 2020. arXiv: 2010.02502. URL: https://arxiv.org/abs/2010.02502.

[31] Yang Song and Stefano Ermon. *Generative Modeling by Estimating Gradients of the Data Distribution*. 2020. arXiv: 1907.05600 [cs.LG].

[32] Yang Song et al. *Score-Based Generative Modeling through Stochastic Differential Equations*. 2020. arXiv: 2011.13456. URL: https://arxiv.org/abs/2011.13456.

[33] Alex Nichol and Prafulla Dhariwal. *Improved Denoising Diffusion Probabilistic Models*. 2021. DOI: 10.48550/ARXIV.2102.09672. URL: https://arxiv.org/abs/2102.09672.

[34] Sefik Ilkin Serengil and Alper Ozpinar. “HyperExtended LightFace: A Facial Attribute Analysis Framework”. In: *2021 International Conference on Engineering and Emerging Technologies (ICEET)*. 2021.

[35] Jonathan Ho and Tim Salimans. *Classifier-Free Diffusion Guidance*. 2022. arXiv: 2207.12598 [cs.LG].

[36] Jonathan Ho et al. *Video Diffusion Models*. 2022. arXiv: 2204.03458 [cs.CV].

[37] Cheng Lu et al. *DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps*. 2022. arXiv: 2206.00927 [cs.LG].

[38] F. Olan. *Fake news on Social Media: the Impact on Society*. 2022.

[39] Kushagra Pandey et al. *DiffuseVAE: Efficient, Controllable and High-Fidelity Generation from Low-Dimensional Latents*. 2022. arXiv: 2201.00308 [cs.LG].

[40] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. arXiv: 2204.06125 [cs.CV].

[41] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2022. arXiv: 2112.10752 [cs.CV].

[42] Muhammed T S. *The disaster of misinformation: a review of research in social media*. 2022.

[43] Chitwan Saharia et al. *Palette: Image-to-Image Diffusion Models*. 2022. DOI: 10.1145/3528233.3530757. URL: https://doi.org/10.1145/3528233.3530757.

[44] Chitwan Saharia et al. *Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding*. 2022. arXiv: 2205.11487 [cs.CV].

[45] Skafle. *Misinformation About COVID-19 Vaccines on Social Media: Rapid Review*. 2022.

[46] Zhisheng Xiao, Karsten Kreis, and Arash Vahdat. *Tackling the Generative Learning Trilemma with Denoising Diffusion GANs*. 2022. arXiv: 2112.07804 [cs.LG].

[47] Hao Zhu et al. *CelebV-HQ: A Large-Scale Video Facial Attributes Dataset*. 2022. arXiv: 2207.12393 [cs.CV].

[48] Tim Brooks, Aleksander Holynski, and Alexei A. Efros. *InstructPix2Pix: Learning to Follow Image Editing Instructions*. 2023. arXiv: 2211.09800 [cs.CV].

[49] Sicheng Gao et al. *Implicit Diffusion Models for Continuous Super-Resolution*. 2023. arXiv: 2303.16491 [cs.CV].

[50] Jiatao Gu et al. *Learning Controllable 3D Diffusion Models from Single-view Images*. 2023. arXiv: 2304.06700 [cs.CV].

[51] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[52] *PIL.Image - Pillow (PIL Fork) documentation*. 2023. URL: https://pillow.readthedocs.io/en/stable/_modules/PIL/Image.html#Image.convert.

[53] *scipy.integrate.RK45 - SciPy Manual*. 2023. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.RK45.html.

[54] Zeyang Sha et al. *DE-FAKE: Detection and Attribution of Fake Images Generated by Text-to-Image Generation Models*. 2023. arXiv: 2210.06998 [cs.CR].

Appendices

A Derivations

A.1 Training Objective, L, Derivation

Note that $\mathbb{E}_q$ denotes $\mathbb{E}_{q(x_{1:T}|x_0)}$.

$$
\begin{aligned}
L &:= -\mathbb{E}_q \left[\log p_\theta(x_0|x_{1:T})\right] + D_{\mathrm{KL}}(q(x_{1:T}|x_0)\|p_\theta(x_{1:T})) \\
&:= -\mathbb{E}_q \left[\log p_\theta(x_0|x_{1:T})\right] + \mathbb{E}_q \left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{1:T})}\right] \\
&:= \mathbb{E}_q \left[-\log p_\theta(x_0|x_{1:T}) + \log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{1:T})}\right] \\
&:= \mathbb{E}_q \left[-\log p_\theta(x_0|x_{1:T}) - \log \frac{p_\theta(x_{1:T})}{q(x_{1:T}|x_0)}\right] \\
&:= -\mathbb{E}_q \left[\log \frac{p_\theta(x_0|x_{1:T}) \times p_\theta(x_{1:T})}{q(x_{1:T}|x_0)}\right] \\
&:= -\mathbb{E}_q \left[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}\right] \\
&:= -\mathbb{E}_q \left[\log \frac{p_\theta(x_{0:T})}{\prod_{t=1}^{T} q(x_t|x_{t-1})}\right] \\
&:= -\mathbb{E}_q \left[\log \frac{p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)}{\prod_{t=1}^{T} q(x_t|x_{t-1})}\right] \\
&:= -\mathbb{E}_q \left[\log p(x_T) + \log \frac{\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)}{\prod_{t=1}^{T} q(x_t|x_{t-1})}\right] \\
&:= -\mathbb{E}_q \left[\log p(x_T) + \sum_{t\geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right]
\end{aligned}
$$

A.2 $\mathcal{L}_{\text{vlb}}$ Derivation

$$
\begin{aligned}
\mathcal{L}_{\text{vlb}} &:= -\mathbb{E}_q \left[\log p(x_T) + \sum_{t \geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})} \right] \\
&:= -\mathbb{E}_q \left[\log p(x_T) + \sum_{t > 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})} + \log \frac{p_\theta(x_0|x_1)}{q(x_1|x_0)} \right] \\
&:= -\mathbb{E}_q \left[\log p(x_T) + \sum_{t > 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} \cdot \frac{q(x_{t-1}|x_0)}{q(x_t|x_0)} + \log \frac{p_\theta(x_0|x_1)}{q(x_1|x_0)} \right] \\
&:= -\mathbb{E}_q \left[\log p(x_T) + \sum_{t > 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} + \log \frac{1}{q(x_T|x_0)} + \log \frac{q(x_1|x_0)}{1} + \log \frac{p_\theta(x_0|x_1)}{q(x_1|x_0)} \right] \\
&:= -\mathbb{E}_q \left[\log p(x_T) + \sum_{t > 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} + \log \frac{1}{q(x_T|x_0)} + \log p_\theta(x_0|x_1) \right] \\
&:= -\mathbb{E}_q \left[\frac{\log p(x_T)}{q(x_T|x_0)} + \sum_{t > 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} + \log p_\theta(x_0|x_1) \right] \\
&:= -\mathbb{E}_q \left[-D_{\text{KL}}(q(x_T|x_0) \| p_(x_T)) - \sum_{t > 1} D_{\text{KL}}(q(x_{t-1}|x_t, x_0) \| p_\theta(x_{t-1}|x_t)) + \log p_\theta(x_0|x_1) \right] \\
&:= \mathbb{E}_q \left[D_{\text{KL}}(q(x_T|x_0) \| p_(x_T)) + \sum_{t > 1} D_{\text{KL}}(q(x_{t-1}|x_t, x_0) \| p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1) \right] \\
&:= \mathbb{E}_q \left[\mathcal{L}_T + \sum_{t > 1} \mathcal{L}_{t-1} + \mathcal{L}_0 \right]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{L}_T &:= D_{\text{KL}}(q(x_T|x_0) \| p_\theta(x_T)) \\
\mathcal{L}_{t-1} &:= D_{\text{KL}}(q(x_{t-1}|x_t, x_0) \| p_\theta(x_{t-1}|x_t)) \\
\mathcal{L}_0 &:= -\log p_\theta(x_0|x_1)
\end{aligned}
$$

A.3 $\mathcal{L}_{t-1}$ Closed Form Derivation

$$
D_{\text{KL}}(A \| B) = \int_x [\text{pdf}_A(x)(\log \text{pdf}_A(x) - \log \text{pdf}_B(x))] \, dx
$$

In the case of Gaussians...

$$
\begin{aligned}
\text{pdf}_A(x) &= \mathcal{N}_{\mu_A, \sigma_A}(x) \\
\text{pdf}_B(x) &= \mathcal{N}_{\mu_B, \sigma_B}(x)
\end{aligned}
$$

$$
\begin{aligned}
D_{\mathrm{KL}}(A\|B) &= \int_x \mathcal{N}_{\mu_A,\sigma_A}(x)(\log \mathcal{N}_{\mu_A,\sigma_A}(x) - \log \mathcal{N}_{\mu_B,\sigma_B}(x))dx \\
&= \int_x \left(\frac{1}{\sqrt{2\pi}\sigma_A} \exp\left[-\frac{1}{2}\left(\frac{x-\mu_A}{\sigma_A} \right)^2 \right] \right) \\
&\qquad \left(-\frac{1}{2}\left(\frac{x-\mu_A}{\sigma_A} \right)^2 + \frac{1}{2}\left(\frac{x-\mu_B}{\sigma_B} \right)^2 + \ln \frac{\sigma_B}{\sigma_A} \right) dx \\
&= \int_x \left(\frac{1}{\sqrt{2\pi}\sigma_A} \exp\left[-\frac{1}{2}\left(\frac{x-\mu_A}{\sigma_A} \right)^2 \right] \right) \\
&\qquad \left(\frac{1}{2}\left[\left(\frac{x-\mu_B}{\sigma_B} \right)^2 - \left(\frac{x-\mu_A}{\sigma_A} \right)^2 \right] + \ln \frac{\sigma_B}{\sigma_A} \right) dx
\end{aligned}
$$

We consider the expectation with respect to distribution A.

$$
\begin{aligned}
D_{\mathrm{KL}}(A\|B) &= \mathbb{E}_A \left(\frac{1}{2}\left[\left(\frac{x-\mu_B}{\sigma_B} \right)^2 - \left(\frac{x-\mu_A}{\sigma_A} \right)^2 \right] + \ln \frac{\sigma_B}{\sigma_A} \right) \\
&= \mathbb{E}_A \left(\frac{1}{2}\left[\left(\frac{x-\mu_B}{\sigma_B} \right)^2 - \left(\frac{x-\mu_A}{\sigma_A} \right)^2 \right] \right) + C_1 \\
&= \mathbb{E}_A \left[\frac{1}{2\sigma_B^2}(x-\mu_B)^2 - \frac{1}{2\sigma_B^2}(x-\mu_A)^2 \right] + C_1 \\
&= \mathbb{E}_A \left[\frac{1}{2\sigma_B^2}(x-\mu_B)^2 \right] + C_2 \\
&= \mathbb{E}_A \left[\frac{1}{2\sigma_B^2}(x-\mu_A+\mu_A-\mu_B)^2 \right] + C_2 \\
&= \mathbb{E}_A \left[\frac{1}{2\sigma_B^2}\left((x-\mu_A)^2 + 2(x-\mu_A)(\mu_A-\mu_B) + (\mu_A-\mu_B)^2 \right) \right] + C_2
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{E}_A\left[(x-\mu_A)^2 \right] &= \sigma_A^2 \\
\mathbb{E}_A\left[(x-\mu_A) \right] &= 0
\end{aligned}
$$

$$
\begin{aligned}
D_{KL}(A\|B) &= \mathbb{E}_A \left[\frac{1}{2\sigma_B^2}\left(\sigma_A^2 + (\mu_A-\mu_B)^2 \right) \right] + C_2 \\
&= \mathbb{E}_A \left[\frac{1}{2\sigma_B^2}\|\mu_A-\mu_B\|^2 \right] + C_3
\end{aligned}
$$

$$
\begin{aligned}
p_\theta(x_{t-1}, x_t) &= \mathcal{N}(\mu_\theta(x_t,t), \sigma_t^2 \mathbf{I}) \\
q(x_{t-1}|x_t, x_0) &= \mathcal{N}(\tilde{\mu}_t(x_t,x_0), \tilde{\beta}_t \mathbf{I})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{L}_{t-1} &= D_{\mathrm{KL}}(q(x_{t-1}|x_t,x_0)\|p_\theta(x_{t-1}|x_t)) \\
&= \mathbb{E}_q \left[\frac{1}{2\sigma_t^2}\|\tilde{\mu}_t(x_t,x_0) - \mu_\theta(x_t,t)\|^2 \right] + C
\end{aligned}
$$

A.4 DDIM Details

We consider the family of distributions, $q_\sigma \in Q$.

$$q_\sigma(x_{1:T}|x_0) = q_\sigma(x_T|x_0)\prod_{t=2}^{T} q_\sigma(x_{t-1}|x_t, x_0)$$
$$q_\sigma(x_T|x_0) = \mathcal{N}(\sqrt{\alpha_t}x_0, (1-\alpha_T)\mathbf{I})$$
$$\forall t > 1.\ \ q_\sigma(x_{t-1}|x_t, x_0) = \mathcal{N}(\sqrt{\alpha_{t-1}}x_0 + \sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\frac{x_t - \sqrt{\alpha_t}x_0}{\sqrt{1-\alpha_t}}, \sigma_t^2\mathbf{I})$$

This family is chosen as its members define joint inference distributions with the desired marginals. i.e., $\forall t.\ \ q_\sigma(x_t|x_0) = \mathcal{N}(\sqrt{\alpha_t}x_0, (1-\alpha_t)\mathbf{I})$. We can then use Bayes' rule to compute the, now non-Markovian, forward process.

$$q_\sigma(x_t|x_{t-1}, x_0) = \frac{q_\sigma(x_{t-1}|x_t, x_0)q_\sigma(x_t|x_0)}{q_\sigma(x_{t-1}|x_0)}$$

Each x_t could depend on x_{t-1} and x_0, and the magnitude of σ controls how stochastic the forward process is. i.e., as $\sigma \to 0$, observing x_t and x_0 fixes x_{t-1} to a known value, making the process entirely deterministic.

We now need a trainable generative process $p_\theta(x_{0:T})$, where each $p_\theta(x_{t-1}|x_t)$ leverages knowledge of $q_\sigma(x_{t-1}|x_t, x_0)$. Given a noisy observation x_t, we predict x_0 and then obtain a sample x_{t-1} using $q_\sigma(x_{t-1}|x_t, x_0)$. We predict x_0 using approximator, x_{0_θ}.

$$x_{0_\theta}(x_t, t) = \frac{x_t - \sqrt{(1-\alpha_t)}\cdot\epsilon_\theta(x_t, t)}{\sqrt{\alpha_t}}$$

We can then define the generative process using the fixed prior $p_\theta(x_T) = \mathcal{N}(0, \mathbf{I})$

$$p_\theta(x_{t-1}|x_t, t) = \begin{cases}\mathcal{N}(x_{0_\theta,t}(x_1), \sigma_t^2\mathbf{I}) & \text{if } t = 1\\ q_\sigma(x_{t-1}|x_t, x_{0_\theta}(x_t, t)) & \text{otherwise}\end{cases}$$

For fast sampling, the sequential forward process is defined over $x_{\tau_1:\tau_S}$ such that $q(x_{\tau_i}|x_0) = \mathcal{N}(\sqrt{\alpha}_{\tau_i}x_0, (1-\alpha_{\tau_i})\mathbf{I})$ matches the marginals.

$$q_{\sigma,\tau}(x_{1:T}|x_0) = q_{\sigma,\tau}(x_{\tau_S}|x_0)\prod_{i=1}^{S}(x_{\tau_{i-1}}|x_{\tau_i}, x_0)\prod_{t\in\bar{\tau}} q_{\sigma,\tau}(x_t|x_0)$$
$$\bar{\tau} = [1, ..., T]\backslash\tau$$

A.5 Fast Sampling with LVDDPMs

Considering a model we train with T steps and training noise schedule $\bar{\alpha}_t$, we may sample with an arbitrary sequence, S, of timestep values, t. We use the sampling noise schedule, $\bar{\alpha}_{S_t}$, to obtain sampling variances. $\Sigma_\theta(x_t, t)$ is re-scaled for shorter diffusion processes by fixing it to lie between the two extreme variances, as before.

$$\beta_{S_t} = \frac{1-\bar{\alpha}_{S_t}}{\bar{\alpha}_{S_{t-1}}} \qquad \tilde{\beta}_{S_t} = \frac{1-\bar{\alpha}_{S_{t-1}}}{1-\bar{\alpha}_{S_t}}\beta S_t \tag{5.1}$$

$$\Sigma_\theta(x_t, t) = \exp(v\log\beta_{S_t} + (1-v)\log\tilde{\beta}_{S_t}) \tag{5.2}$$

We can use this to compute $p(x_{S_t}|x_{S_t})$. Then, the sampling process is reduced from $T \approx 4000$ to $K \approx 50$ steps by setting S to K evenly spaced steps between 1 and T, with minimal impact on sample quality.

$$p(x_{S_t}|x_{S_t}) = \mathcal{N}(\mu_\theta(x_{S_t}, S_t), \Sigma_\theta(x_{S_t}, S_t)) \tag{5.3}$$

B Code

B.1 Example Config File

```
from scripts.train import main as trainScript
from core.dataset import CIFAR10
from core.utils import ConditionType, ScheduleSamplerType
from core.models.model_utils import VarType,
  MeanType, LossType, NoiseSchedule, NetworkType, ModelType

class dotdict(dict):
    __getattr__ = dict.get
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__


def main():
    run_name = "base_cifar"
    var_type = VarType.FIXED_LARGE
    batch_size = 128

    dataset = dotdict(dict(
        dataset=CIFAR10,
        data_dir="./datasets",
        pin_memory=False,
        drop_last=True,
    ))

    network = dotdict(dict(
        network_type=NetworkType.UNET,
        image_size=32,
        channel_mult=(1, 2, 2, 2),
        num_channels=128,
        num_res_blocks=2,
        attention_resolutions="16",
        var_type=var_type,
        num_heads=1,
        num_heads_upsample=-1,
        use_scale_shift_norm=False,
        dropout=0.1,
        conv_resample=False,
        condition=ConditionType.none,
    ))

    model = dotdict(dict(
        model_type=ModelType.DDPM,
        diffusion_steps=1000,
        noise_schedule=NoiseSchedule.LINEAR,
        loss_type=LossType.MSE,
```

```
        var_type=var_type,
        mean_type=MeanType.EPSILON,
        rescale_timesteps=False,
        timestep_respacing="",
    ))

    train = dotdict(dict(
        epochs=2000,
        batch_size=batch_size,
        lr=2e-4,
        weight_decay=0.0,
        ema_rate="0.9999",
        schedule_sampler=ScheduleSamplerType.UNIFORM,
    ))

    args = dotdict(dict(
        run_name=run_name,

        dataset=dataset,
        network=network,
        model=model,
        train=train,

        log_dir="/rds/user/------/hpc-work/backups/" +
          run_name + "/logs",

        save_interval=1,
        chkpt_dir="/rds/user/------/hpc-work/backups/" +
          run_name + "/chkpts",

        sample_dir="/rds/user/------/hpc-work/backups/" +
          run_name + "/samples",
        sample_intv=25,
        sample_size=32,

        eval_intv=250,

        backup_dir="/rds/user/------/hpc-work/backups/" +
          run_name + "/chkpts",

    ))

    trainScript(args)


if __name__ == "__main__":
    main()
```

C Additional Results

C.1 CIFAR10 Colourisation

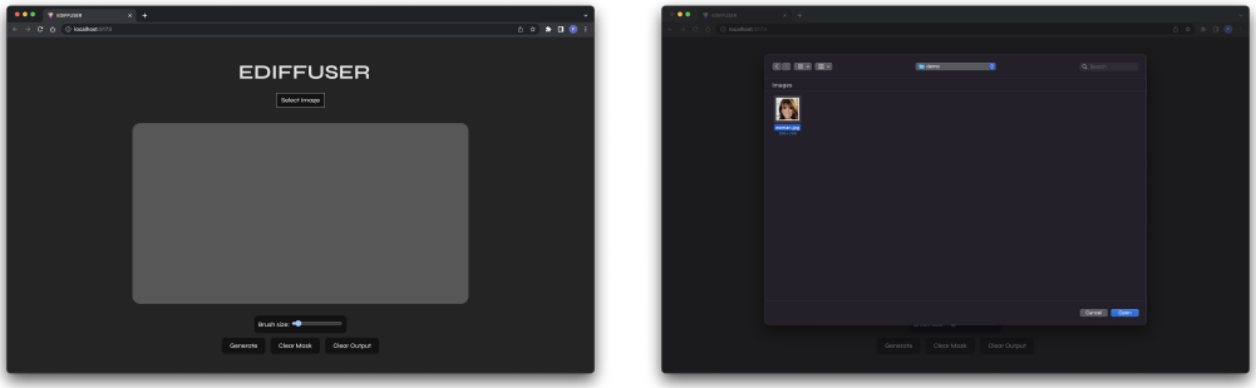
T	IS	FID	NLL	Speed
4000	10.82	0.746	2.078	159
25	6.438	96.16	5.990	2
50	9.794	56.25	5.208	2
100	11.46	30.35	4.458	4

Table 5.1: LVDDPM32LinSimColour

T	IS	FID	NLL	Speed
4000	10.86	0.656	1.888	144
25	9.928	55.94	5.386	2
50	11.53	29.84	4.600	2
100	11.89	12.68	3.894	4

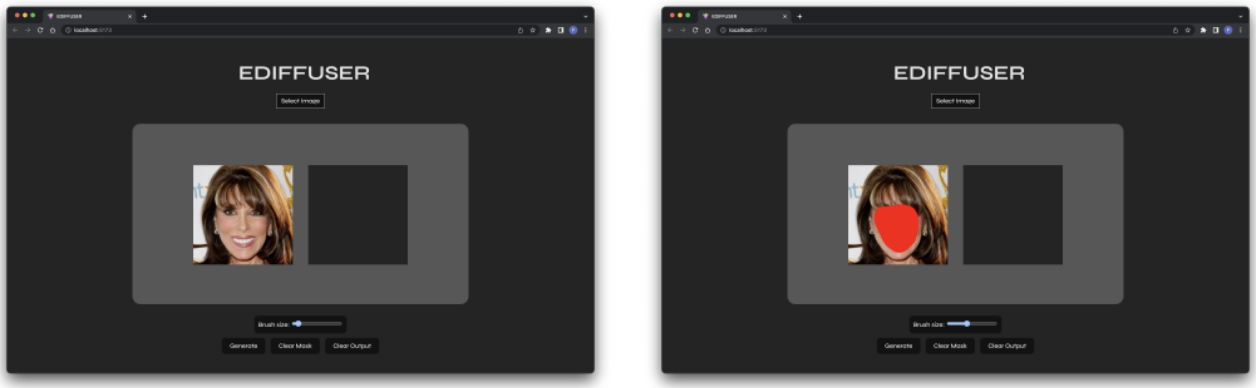
Table 5.2: LVDDPM32SimColour

C.2 Web-Application



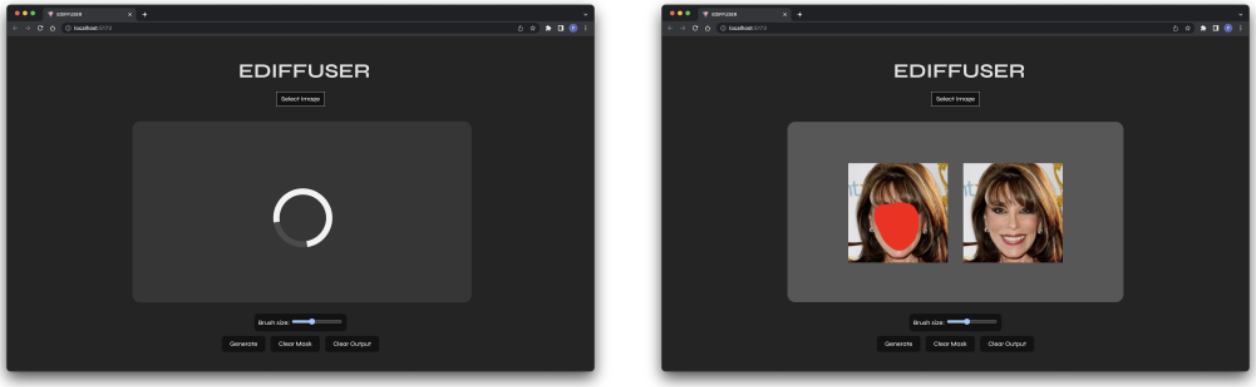
(a) Landing Page (b) Upload an Image

Fig. 5.1: The Web-App



(a) Uploaded image is shown (b) User can set brush size and draw a mask

Fig. 5.2: Drawing a mask



(a) Loading symbol is shown during processing (b) Output is shown

Fig. 5.3: Drawing a mask

D Image Samples

D.1 CIFAR10 Unconditional Generation

D.1.1 DDPM32

Fig. 5.4: DDPM32

D.1.2 DDIM32

(a) 25 steps (b) 50 steps (c) 100 steps

Fig. 5.5: DDIM32 Samples

D.1.3 LVDDPM32

Fig. 5.6: LVDDPM32LinSim Sample

(a) 25 steps

(b) 50 steps

(c) 100 steps

Fig. 5.7: LVDDPM32LinSim Samples using Fast Sampling

Fig. 5.8: LVDDPM321k Sample

(a) 25 steps (b) 50 steps (c) 100 steps

Fig. 5.9: LVDDPM321k Samples using Fast Sampling

Fig. 5.10: LVDDPM32Sim Sample

(a) 25 steps (b) 50 steps (c) 100 steps

Fig. 5.11: LVDDPM32Simp Samples using Fast Sampling

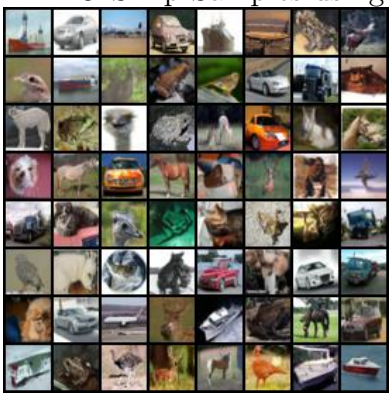
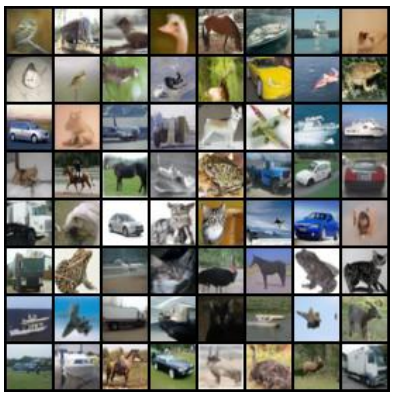


Fig. 5.12: LVDDPM32 Sample

(a) 25 steps



(b) 50 steps

(c) 100 steps

Fig. 5.13: LVDDPM32 Samples using Fast Sampling

D.1.4 VPSDE32

(a) VPSDE32 Sample

(b) VPSDE32 (DOP) Sample

(c) 100 steps

Fig. 5.14: VPSDE32 (DPMSolver) Sample

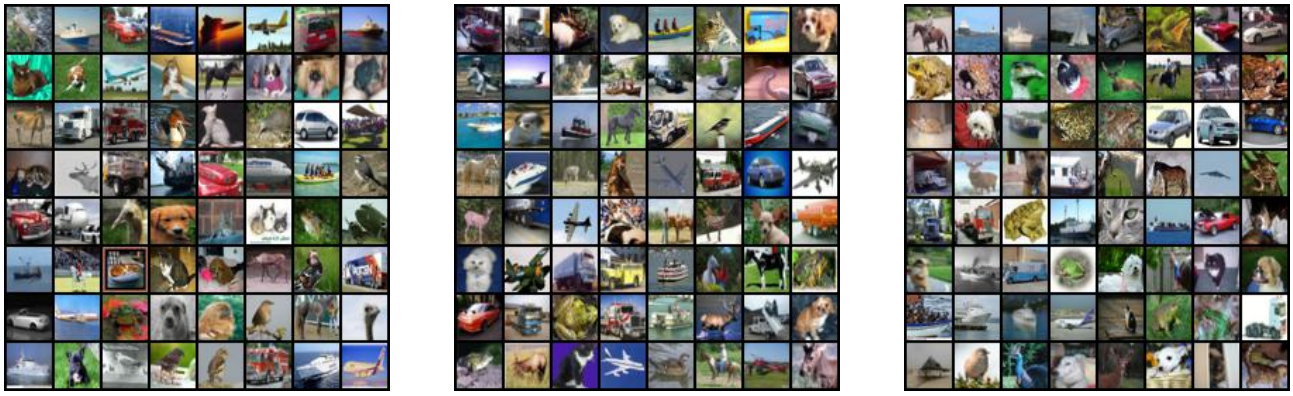
D.2 CIFAR10 Colourisation

D.2.1 DDPM32Colour

(a) Sample (b) BW (c) Ground Truth

Fig. 5.15: DDPM32Colour Samples

D.2.2 DDIM32Colour



(a) 25 steps (b) 50 steps (c) 100 steps

Fig. 5.16: DDIM32Colour Samples

D.2.3 LVDDPM32Colour

Fig. 5.17: LVDDPM32LinSimColour Sample

(a) 25 steps

(b) 50 steps

(c) 100 steps

Fig. 5.18: LVDDPM32LinSimColour Samples using Fast Sampling

Fig. 5.19: LVDDPM321k Sample

(a) 25 steps

(b) 50 steps

(c) 100 steps

Fig. 5.20: LVDDPM321kColour Samples using Fast Sampling

Fig. 5.21: LVDDPM32Sim Sample

(a) 25 steps

(b) 50 steps

(c) 100 steps

Fig. 5.22: LVDDPM32SimpColour Samples using Fast Sampling

Fig. 5.23: LVDDPM32 Sample

(a) 25 steps

(b) 50 steps

(c) 100 steps

Fig. 5.24: LVDDPM32Colour Samples using Fast Sampling

D.2.4 VPSDE32Colour

(a) Sample

(b) BW

(c) Ground Truth

Fig. 5.25: VPSDE32Colour Samples

Fig. 5.26: VPSDE32Colour (ODE) Sample

Fig. 5.27: VPSDE32Colour (DPM) Sample

D.3 CelebAHQ Unconditional Generation

D.3.1 DDPM256

Fig. 5.28: DDPM256 Sample

D.3.2 DDIM256

(a) 25 steps

(b) 50 steps

(c) 100 steps

Fig. 5.29: DDIM256 Samples

D.3.3 LVDDPM256

Fig. 5.30: LVDDPM256 Sample

(a) 25 steps (b) 50 steps (c) 100 steps

Fig. 5.31: LVDDPM256 Fast Samples

D.4 CelebAHQ Inpainting

D.4.1 DDPM256Inpaint

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.32: DDPM256Inpaint Samples

D.4.2 DDIM256Inpaint

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.33: DDIM256Inpaint (25) Samples

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.34: DDIM256Inpaint (50) Samples

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.35: DDIM256Inpaint (100) Samples

D.4.3 LVDDPM256Inpaint

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.36: LVDDPM256Inpaint Samples

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.37: LVDDPM256Inpaint (25) Samples

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.38: LVDDPM256Inpaint (50) Samples

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.39: LVDDPM256Inpaint (75) Samples

(a) Sample (b) Masked (c) Ground Truth

Fig. 5.40: LVDDPM256Inpaint (100) Samples

Part II Project Proposal - Denoising Diffusion Probabilistic Models for Image Inpainting

[redacted]

October 2022

Contents

1 Introduction

Denoising Diffusion Probabilistic Models (DDPMs) are a new class of latent variable models, introduced in Sohl-Dickstein et al. 2015 that are inspired by Non-equilibrium Statistical Physics. These models have been shown to deliver very promising results for certain generative applications compared to Generative Adversarial Networks (GANs), Variational Auto-Encoders (VAEs) and Normalising Flows. Subsequent literature improves on various aspects of the original models. A recent paper (Saharia et al. 2022) introduces a framework for conditional image synthesis using DDPMs and shows that they can produce state-of-the-art samples for certain applications. Image synthesis can be conditioned on various parameters. DALL-E and other similar image generators are conditioned on a text prompt. It is possible to condition on a class label to generate images, for example, that contain cars. In this project, I will consider image inpainting, in which a model realistically fills in user-selected masked regions of an image. For image inpainting, synthesis is conditioned on the unmasked region of the image.

2 Background

2.1 Image Synthesis using Deep Generative Models

Deep generative models automatically learn complex data distributions from training data. The models can then be used to generate new samples that could have plausibly been drawn from those distributions. These can be applied to synthesising images by learning from image training data. In recent years, such generation of images has hugely grown in popularity as the samples produced have become more realistic.

2.2 Diffusion Probabilistic Models

Probabilistic models will ideally be flexible enough to describe complex data but still tractable, (feasible to compute). However, these two objectives are usually conflicting. For example, a Gaussian distribution is tractable but not very flexible. On the other hand, we can define probabilistic models in terms of any non-negative function $\phi(x)$, yielding a distribution $p(x) = \frac{\phi(x)}{Z}$, where Z is a normalisation constant. However, computing the normalisation constant is generally intractable and the model typically requires an expensive Monte Carlo process to evaluate, train or draw samples from (Sohl-Dickstein et al. 2015).

DDPMs overcome this trade-off. They offer flexibility in modelling target distributions by using a Markov chain trained using variational inference to gradually convert a simple Gaussian into the target. They also offer exact sampling as, rather than using the Markov chain to approximate the target distribution, the endpoint of the chain is taken exactly as the target.

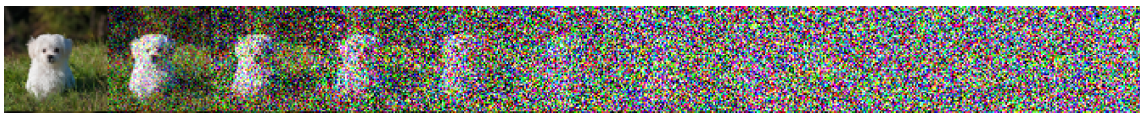


Figure 1: Illustration of a diffusion process (Sohl-Dickstein et al. 2015).

They are latent variable models with latents of the same dimensionality as the original data. DDPMs have a forward and reverse process. Figure 1 shows the forward process (left to right). Unlike other latent variable models, the forward process is fixed. It is a Markov chain that gradually adds Gaussian noise to the data according to a variance schedule. The forward process variances can be learnt by reparameterisation or held as constant hyper-parameters. The reverse process is defined as a Markov chain with learned Gaussian transitions that reverse the addition of noise. It is possible for the reverse process to be learnt as, when the variances are small, both the forward and reverse processes have the same functional form. Training is performed by optimising the variational bound on negative log likelihood. This has been shown to be equivalent to maximising the Evidence Lower Bound (ELBO).

Since the work of Sohl-Dickstein et al. 2015, various papers have improved on DDPMs. Ho et al. 2020 identifies how reparameterising can lead to an equivalence with denoising score matching with Langevin dynamics. Nichol et al. 2021 identifies modifications that can allow DDPMs to achieve competitive log-likelihoods and also shows that learning variances of the reverse diffusion process

allows sampling with an order of magnitude fewer forward passes, witg a negligible difference in sample quality.

2.3 Image Inpainting

Image inpainting is a form of conditional image generation where, given an image with a user-selected masked portion, the region is filled in realistically. This has real-world applications such as removing undesired objects from an image.

Inpainting has existed with varying quality for over a decade. Early methods worked well on textured regions but struggled with generating semantically consistent structure. A notable early consumer-facing example is Adobe's Content-Aware Fill, in which a masked region of an image is filled based approximate nearest-neighbour matches using the PatchMatch algorithm. Another more recent example is Google's Magic Eraser, where the user can draw a mask on an image on their Pixel phone. The computation is performed on-device and within seconds.

Figure 2: Left: Image with mask. Centre: Sample from Palette. Right: Original image. (Saharia et al. 2022)

One way of performing image inpainting is using deep generative models to learn the conditional distribution of output images (complete images) given the input (masked image). The models should be able to capture multi-modal distributions in the high-dimensional space of images. GANs are used widely, but often require auxiliary objectives (e.g. on structure) and struggle with sample diversity (Saharia et al. 2022). Instead, inpainting can be implemented using DDPMs conditioned on the unmasked regions of the image. Specifically, the conditional diffusion models have the form $p(y|x)$ where x is a masked image and y is filled in. An example is shown in Figure 2.

3 Work to be Undertaken

3.1 Base

3.1.1 Task 1

I will implement DDPMs for unconditional image generation according to Ho et al. 2020. This initial implementation will be on smaller images than the paper uses (likely 32×32). This acts as a base for conditional image generation.
Starting Point: I plan on making this implementation from scratch. The paper does provide an official code implementation which I may refer to for comparison of results.

3.1.2 Task 2

I will extend **Task 1** with conditional image generation to perform image inpainting according to Saharia et al. 2022. Here, I will increase the image resolution used to match the literature.
Starting Point: There is no official implementation of this work. The sample outputs and image masks from the paper are available for comparison.

3.2 Extensions

3.2.1 Task 3

I will incorporate the improvements detailed in various literature into **Task 2** (including Nichol et al. 2021) and compare my results to Saharia et al. 2022.
Starting Point: Saharia et al. 2022 releases an official code implementation, but not for inpainting. This will be my own implementation from scratch.

3.2.2 Task 4

I will create a web-application that uses semantic segmentation and the DDPM developed in **Task 3** to allow a user to upload an image and mask semantic regions so the DDPM can fill it in.
Starting Point: There are many existing implementations of semantic segmentation, so I will focus on incorporating this into the work flow and engineering a web app rather than re-implementing it.

3.3 Challenges and Risks

Challenge One of the main concerns is the amount of time taken to train the models.

Mitigation I will use smaller images than the 2020 paper for unconditional image generation. I will be using CSD3 to speed up training. If this is still too slow, I will bring forward improvements from the 2021 paper that enable faster training times to the start of the timeline. I could also use smaller images throughout the project.

Risk Due to the training time of the models, corruption of the model can be very detrimental to the project timeline.

Mitigation Constant backups of every model and records of the results associated with each model. I will use a version control system (Git).

Risk Failure to obtain access to CSD3.

Mitigation There are other computing clusters available, which I will try to use instead. I will use smaller images throughout rather than trying to match the resolutions used in the literature.

4 Success Criteria

4.1 Evaluation Methodology

In the past few years, there have been an plethora of methods used to evaluate the generation of image samples by neural networks. I will use methods according to the literature I am basing my implementations on so that I am able to compare results.

For unconditional sample generation I plan on using Fréchet Inception Distance (FID), Inception Scores (IS) and Precision and Recall (Measured using features detected by Inception-V3). I will use the CIFAR and MNIST datasets.

For image inpainting, I will use FID, IS, Perceptual Distance (PD) and Classification Accuracy (CA). To measure sample diversity for image inpainting, I will produce multiple samples and then measure similarity between them using SSIM and LPIPS scores. I will use subsets of the ImageNet and Places2 datasets as described in Saharia et al. 2022.

4.2 Base Success Criteria

4.2.1 Task 1

1. Successfully implement DDPMs for unconditional image generation
2. Measure performance of model using FID, IS, Precision and Recall

4.2.2 Task 2

1. Successfully implement DDPMs for image generation conditioned on a masked image for image inpainting
2. Measure performance of model using FID, IS, CA and PD

4.3 Extension Success Criteria

4.3.1 Task 3

1. Successfully incorporate changes from literature to improve performance of conditional DDPM for image inpainting
2. Successfully incorporate changes from literature to improve training time of DDPM
3. Measure performance of model using FID, IS, CA and PD

4.3.2 Task 4

1. Successfully implement a semantic segmentation network that works with the chosen image resolution
2. Successfully implement pipeline made of segmentation network and DDPM
3. Successfully implement a web-app allowing users to use the pipeline to remove subjects from images and realistically replace them

5 Work Packages and Milestones

The table below outlines my planned timetable for the project. **Tx** refers to **Task x** from Section **3**.

Stage	Weeks	Dates	Work to Complete
Package 1	1 and 2	17/10 – 30/10	– **T1** Make detailed notes on DDPMs – **T1** Experiment with the network architectures used in the literature – **Write-up** Complete Introduction
Package 2	3 and 4	31/10 – 13/11	– **T1** Implement DDPMs for unconditional image generation (at smaller resolutions) – **Write-up** Start **T1** Implementation
Package 3	5 and 6	14/11 – 27/11	– **T1** Fine-tune code and parameters – **T1** Compare results with Ho et al. 2020 – **Write-up** Complete **T1** Implementation
Milestone 1			– Completed **T1** – Completed Introduction for **Write-up** – Completed **T1** Implementation in **Write-up**
Package 4	7 and 8	28/11 – 11/12	– **T2** Make detailed notes on conditional DDPMs for image inpainting – **T2** Start implementing conditional DDPMs for inpainting – **Write-up** Start **T2** Implementation – **Write-up** Start Preparation
Break	9 and 10	12/12 – 25/12	– Holiday and catch-up
Package 5	11 and 12	26/12 – 8/1	– **T2** Finish implementing conditional DDPMs for inpainting – **T2** Start fine tuning code and parameters – **Write-up** Continue **T2** Implementation – **Write-up** Continue Preparation
Package 6	13 and 14	9/1 – 22/1	– **T2** Finish fine tuning code and parameters – **T2** Compare results with Saharia et al. 2022 – **Write-up** Complete **T2** Implementation – **Write-up** Continue Preparation
Milestone 2			– Completed **T2** – Completed **T2** Implementation in **Write-up**

Package 7	15 and 16	23/1 – 5/2	– **T3** Make notes on the improvements presented in various literature, including Nichol et al. 2021 – **T3** Start incorporating the improvements into the work from **T2** – **Write-up** Start **T3** Implementation – **Write-up** Continue Preparation – **Write-up** Progress Report for 3/2
Package 8	17 and 18	6/2 – 19/2	– **T3** Finish incorporating the improvements into **T2** – **Write-up** Continue **T3** Implementation – **Write-up** Complete Preparation
Package 9	19 and 20	20/2 – 5/3	– **T3** Fine tune code and parameters – **T3** Compare with results from Saharia et al. 2022 – **Write-up** Complete **T3** Implementation – **Write-up** Start Evaluation
Milestone 3			– Completed **T3** – Completed **T3** Implementation in **Write-up** – Completed Preparation
Package 10	21 and 22	6/3 – 19/3	– **T4** Research and implement segmentation network for image size used by DDPM – **T4** Connect segmentation network with DDPM – **Write-up** Start **T4** Implementation – **Write-up** Continue Evaluation
Package 11	23 and 24	20/3 – 2/4	– **T4** Build server that runs the models – **Write-up** Continue Evaluation – **Write-up** Continue **T4** Implementation
Package 12	25 and 26	3/4 – 16/4	– **T4** Build web-app that connects to the server – **Write-up** Complete **T4** Implementation – **Write-up** Complete Evaluation
Milestone 4			– Completed **T4** – Completed **T4** Implementation in **Write-up** – Completed Evaluation
Package 13	27, 28, 29	17/4 – 12/5	– **Write-up** Complete Conclusion
Milestone 5			– Submission

6 Resource Declaration

- **Personal Laptop**: I will use my personal device for research, writing implementations and writing the write-up. My computer is a Microsoft Surface Book 3 with 16GB RAM and a NVIDIA GeForce GTX 1650 Max-Q. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. I have a backup machine and will use a revision control system (Git) to which I will make regular backups.
- **CSD3**: In order to train my models, I will use the free tier of CSD3.
- **Google Colab**: I may also use Google Colab for training of models on smaller images.

References

Sohl-Dickstein, Jascha et al. (2015). *Deep Unsupervised Learning using Nonequilibrium Thermodynamics*. DOI: 10.48550/ARXIV.1503.03585. URL: https://arxiv.org/abs/1503.03585.

Ho, Jonathan et al. (2020). *Denoising Diffusion Probabilistic Models*. DOI: 10.48550/ARXIV.2006.11239. URL: https://arxiv.org/abs/2006.11239.

Nichol, Alex et al. (2021). *Improved Denoising Diffusion Probabilistic Models*. DOI: 10.48550/ARXIV.2102.09672. URL: https://arxiv.org/abs/2102.09672.

Saharia, Chitwan et al. (2022). *Palette: Image-to-Image Diffusion Models*. DOI: 10.1145/3528233.3530757. URL: https://doi.org/10.1145/3528233.3530757.