UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

A STUDY ON DIFFUSION PROBABILISTIC MODELS FOR IMAGE GENERATION

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

In partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

ROMAN AHREM MUNOZ

Norman, Oklahoma

2023

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/340084

A STUDY ON DIFFUSION PROBABILISTIC MODELS
FOR IMAGE GENERATION

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Choon Yik Tang, Chair
Dr. Joseph P. Havlicek
Dr. Chao Lan

© Copyright by Roman Munoz 2023

All Rights Reserved.

Acknowledgments

I would like to express my sincere gratitude to Dr. Tang, my thesis advisor, for his valuable guidance, feedback, support, and encouragement throughout my research. His patience and expertise have been instrumental in helping me understand ideas and how to approach challenges and tasks. Dr. Tang has been one of my pillar instructors in my college career who taught me instrumental courses in circuits, digital signals and filtering, and linear system analysis.

I am also grateful to Dr. Havlicek who taught me fundamentals in digital image processing and digital signal processing, two courses I am very fond of and from whom I have learned valuable information. Additionally, I am thankful to Dr. Lan, who has taught me data structures and machine learning, two courses I am very passionate about.

I am incredibly grateful to my family for their support throughout my college career. They have always been there when I have needed them the most. I hope my accomplishments will inspire my younger sisters in their college careers and future endeavors. Finally, I would like to extend special thanks to Kaley Romero, who has undeniably supported all my decisions and goals.

Table of Contents

Chapter 1 Introduction.....**1**

1.1 Background 1

1.2 Thesis Motivation.....4

1.3 Thesis Contributions 6

1.4 Thesis Outline 7

Chapter 2 Theoretical Background.....**9**

2.1 Origins of Diffusion Probabilistic Models 9

2.2 Recent Improvements in Diffusion Probabilistic Models.....18

2.3 U-Net Architecture.....30

2.3.1 Image Convolution.....36

2.3.2 Max-Pooling.....38

2.3.3 Soft-Max Function.....41

2.3.4 Adams Optimizer.....42

Chapter 3 Application to Image Generation **45**

3.1 Logic and Structure of Diffusion Probabilistic Models 45

3.1.1 Dataset 47

3.1.2 Timesteps and Beta (β) Rate.....49

3.1.3 Number of Channels, Features, and Adams Optimizer. 50

3.1.4 Batch Size, Learning Rate, and Number of Epochs. 52

3.2 Pseudo-Code....54

Chapter 4 Hardware Specifications **57**

4.1 Central Processing Unit (CPU) 57

4.2 Graphics Processing Unit (GPU) 58

4.3 Random Access Memory 60

Chapter 5 Generating New Landscape Images....**61**

5.1 First Results on Landscape Dataset....61

5.2 Second Results on Landscape Images Dataset....68

5.3 Results on Landscape Images Dataset on Kaggle Kernel 73

Chapter 6 Generating Complex Images **78**

6.1 Results on City Images Dataset on Kaggle Kernel 78

6.2 Results on Car Images Dataset on Google Collab 83

6.3 Results on Human Faces Dataset on Kaggle Kernel....88

Chapter 7 Conclusion **93**

7.1 Thesis Contribution Analysis....93

7.2 Future Research Direction....95

Bibliography **96**

Appendix....**102**

List of Figures

Figure 2.1: U-Net architecture [41] 32

Figure 2.2: Image representation of size 6x6.....37

Figure 2.3: Small kernel representation of size 3x3 37

Figure 2.4: Max pooling applied to a squared matrix of size 4x4 40

Figure 3.1: Flow chart overview of logic design, functions, and variables in DPM 46

Figure 5.1: Epoch 10 from landscape images dataset on ASUS computer 62

Figure 5.2: Epoch 50 from landscape images dataset on ASUS computer 63

Figure 5.3: Epoch 100 from landscape images dataset on ASUS computer 64

Figure 5.4: Epoch 200 from landscape images dataset on ASUS computer 65

Figure 5.5: Epoch 500 from landscape images dataset on ASUS computer 66

Figure 5.6: Loss vs epoch from landscape images dataset on ASUS computer.....67

Figure 5.7: Epoch 10 from landscape images dataset on ASUS computer (128x128).....69

Figure 5.8: Epoch 100 from landscape images dataset on ASUS computer (128x128).....70

Figure 5.9: Epoch 300 from landscape images dataset on ASUS computer (128x128).....71

Figure 5.10: Loss vs epoch from landscape images dataset on ASUS computer (128x128).....72

Figure 5.11: Epoch 10 from landscape images dataset on Kaggle Kernel 74

Figure 5.12: Epoch 100 from landscape images dataset on Kaggle Kernel 75

Figure 5.13: Epoch 300 from landscape images dataset on Kaggle Kernel 76

Figure 5.14: Loss vs epoch from landscape images dataset on Kaggle Kernel 77

Figure 6.1: Epoch 10 from city-view dataset on Kaggle Kernel 79

Figure 6.2: Epoch 100 from city-view dataset on Kaggle Kernel 80

Figure 6.3: Epoch 250 from city-view dataset on Kaggle Kernel 81

Figure 6.4: Loss vs Epoch from city-view dataset on Kaggle Kernel 82

Figure 6.5: Epoch 10 from car images dataset on Google Collab 84

Figure 6.6: Epoch 100 from car images dataset on Google Collab 85

Figure 6.7: Epoch 200 from car images dataset on Google Collab 86

Figure 6.8: Loss vs epoch from car images dataset on Google Collab.....87

Figure 6.9: Epoch 10 from human faces dataset on Kaggle Kernel 89

Figure 6.10: Epoch 100 from human faces dataset on Kaggle Kernel 90

Figure 6.11: Epoch 250 from human faces dataset on Kaggle Kernel 91

Figure 6.12: Loss vs epoch from human faces dataset on Kaggle Kernel.....92

List of Tables

Table 4.1: CPU comparison among the three environments used 59

Table 4.2: GPU comparison among the three environments used 60

Abstract

Diffusion probabilistic models have emerged as powerful tools for image generation and synthesis tasks. This research delves into the intricate relationship between the hyperparameters of these models and the underlying hardware, aiming to provide insight into optimizing performance. The study specifically investigates the effects of key hyperparameters: the number of feature parameters, the number of timesteps, the image size, the number of images in the dataset, the learning rate, and the number of epoch iterations. Additionally, the influence of hardware, particularly GPU memory, on the overall performance is examined. Through a systematic experimentation framework, we consider different hyperparameters and hardware configurations to quantify their impact on model convergence, image generation quality, and computational efficiency. The research aims to identify optimal hyperparameter settings for diverse tasks while considering the constraints imposed by available hardware resources. Moreover, the study explores potential trade-offs and synergies between hyperparameter tunning and hardware specifications, shedding light on the interplay between algorithmic choices and computational capabilities. The findings from this research contribute to a nuanced understanding of how diffusion probabilistic models can be fine-tuned for image generation, considering the practical implications of hardware limitations. By bridging the gap between algorithmic design and hardware constraints, this work provides valuable guidance for practitioners seeking to leverage diffusion models effectively in real-world scenarios which require image generation.

Chapter 1 Introduction

This chapter begins with the background of Diffusion Probabilistic Models (DPM). The background relates to the evolution of artificial intelligence (AI) and the art expressions from humanity since DPM is a generative image AI model. It also highlights the thesis contributions made by this study and provides the overall thesis outline. Additionally, this chapter presents the motivation for such a study and applications of DPM beyond artistic expressions, including the of DPM in image segmentation and in medical fields such as neuroscience.

1.1 Background

The advancement and development of art has been part of the human experience since early civilizations. The evolution of artistic forms has been affected by external factors from each era, such as different materials used to portrait the reality of the time. During the era of paleolithic arts, carving and chipping rock techniques were used to create petroglyphs and then pictographs [11]. Centuries after, during the byzantine and renaissance movements, paintings were based on religious beliefs portrayed on churches such as mosaics and ceilings [3]. More recently, with the advancement of computers, artists have been able to develop electronic and virtual artistic designs that are no longer drawn on tangible surfaces, but rather in a digital environment consisting of pixels, which in essence are ones and zeros in a central processing unit (CPU) and a graphic processing unit (GPU) [18]. In recent years, this technology has evolved to a point where

human input could be reduced even further with the use of artificial intelligence (AI), where a model can learn to create such artistic digital images from a variety of datasets.

The introduction of AI began in the midway point of the 20th century with a paper in 1950 called *Computing Machinery and Intelligence* by Alan Turing, a young British polymath [49]. He argued that machines could use available information as well as reasoning to solve problems and make decisions, just as humans do. Five years later in 1955, the concept of AI was realized in a new program called *Logic Theorist* by Allen Newel, Cliff Shaw, and Hebert Simon. This program is considered the first of its kind related to AI and was demonstrated at the *Dartmouth Summer Research Project on Artificial Intelligence* (DSRPAI), which was hosted by John McCarthy and Marvin Minsky in 1956. While McCarthy hoped to bring an open-ended discussion on AI, the conference did not meet expectations [44]. This, however, did not stop the research community, and from 1957 to 1974, as computers became faster and more reliable in storage, machine learning algorithms became better and more efficient. One such algorithm was Newell and Simon's *General Problem Solve*r and Joseph Weizanbaum's *ELIZA* which showcased problem solving and the interpretation of spoken language, respectively [30]. With these advancements, AI researchers began to seek government agencies for funding such as the *Defense Advanced Research Projects Agency* (DARPA) [5]. However, computers at the time did not meet storage and computation requirements to continue with AI research, causing government funding to slow down. Despite all of this, many researchers kept working on AI projects such as John Hopfield and David Rumelhart's "deep learning" techniques that made computers learn from experience. Another breakthrough project was Edward Feigenbaum's *expert systems* that allowed non-experts in a particular field to get an expert answer. The Japanese government heavily funded *expert systems* and many other AI endeavors in their *Fifth*

Generation Computer Project [44]. However, the funding ceased, and AI struggled once more. Nevertheless, researchers kept developing new AI algorithms, such as for Aaron Cohen who created the first AI generative image algorithm known as AARON. By 1980, AARON was able to deal with internal aspects of human cognition, differentiate between figure and ground, inside and outside, as well as understand similarity, division, and repetition [12]. With AARON, Cohen created artistic works that were displayed in various galleries. Later in 1997, IBM's *Deep Blue* defeated chess world champion Gary Kasparov and a speech recognition program was introduced by Dragon Systems in Windows [23]. These new advancements were made possible thanks to Moore's Law which states that computer memory and speed doubles every year, and therefore the capabilities and needs of AI finally caught up and so did the generative art expression.

The use of deep learning and neural networks has enabled generative models to learn from larger datasets of different images, thus allowing AI to create intricate realistic artwork. To accomplish such tasks, different image generative models have been developed, including generative adversarial networks (GANS) that use a discriminator and a generator neural network [14], and variational autoencoders (VAE) that use an encoder and a decoder to generate images [20]. However, recently, there is a new generated model that has been popularized among AI enthusiasts and researchers called *diffusion probabilistic model*. This new model is inspired by the laws of thermodynamics. It is a latent variable model that maps the latent space using a Markov chain, upon which noise is gradually added to an image until the image is asymptotically converted to a pure Gaussian noise. Then, the diffusion model is trained to predict such a noise and reverse the process traversing backwards in the Markov chain until the noise is removed. Ultimately, the model creates a new image like that of the original image. Most recently,

diffusion models have been shown to outperform GANS and VAE models by producing high-resolution images with hyper-realistic features in a three-dimensional space in response to a user prompt conditioning a pre-trained model to generate new output images [43].

1.2 Thesis Motivation

Diffusion probabilistic models (DPM) have gained attention and popularity in recent years due to their ability to generate high-quality and diverse samples. The motivation behind these models stems from the limitations of traditional generative models and the desire to address challenges in capturing complex data distributions. Traditional generative models such as Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) sometimes struggle to model complex data distributions effectively. DPMs use a series of transformations to model the data distribution more flexibly, capturing intricate dependencies and patterns. By modeling data distribution through a series of steps, DPMs can handle high-dimensional data more effectively, capturing both global and local dependencies [43]. DPM encourages diversity in generated samples by iteratively transforming a simple distribution into the target distribution, allowing for a wide range of possible outcomes. For gradient-based training, some other generative models suffer from training instability or mode collapse, making it difficult to optimize effectively. In contrast, DPMs use a well-defined diffusion process with tractable likelihoods, allowing for stable and efficient gradient-based training [16]. Additionally, DPMs are widely used for image generation tasks. They can generate realistic and diverse images, making them suitable for applications in art, design, and content creation. They can also be used for data augmentation in

machine learning tasks. By generating diverse examples, they enhance the generalization ability of models trained on limited datasets.

The ability of DPMs to capture complex data distributions makes them useful for anomaly detection. They can identify deviations from normal patterns in various domains, such as medical imaging or cybersecurity. DPMs can also be extended to generate realistic and diverse video sequences. This has applications in entertainment, virtual reality, and simulation. DPMs can also find applications in solving inverse problems, where the goal is to recover the underlying structure or parameters given observed data. This is valuable in fields such as medical imaging and signal processing [43]. They can also be employed for style transfer in images, allowing for the transformation of visual style of an image while preserving its content. Nonetheless, the ability of diffusion models to generate realistic data while preserving privacy features makes them useful in privacy-preserving applications, such as generating synthetic data for research purposes without including sensitive information. Moreover, DPMs can be applied in drug discovery to generate molecular structures with desired properties. This has implications for designing new pharmaceutical compounds [31]. Thus, DPMs have a broad range of applications due to their ability to model complex data distributions, generate diverse samples, and provide stable training. Their versatility makes them suitable for various domains where generative modeling is essential for understanding, simulating, or creating data.

1.3 Thesis Contributions

This thesis is devoted to studying the impact of hyperparameters selection on the performance of a Diffusion Probabilistic Model (DPM). Under different environments with different hardware capabilities. The study is motivated by the fact that certain values of the hyperparameters require higher computational complexity, power consumption, and computer memory. The hyperparameters analyzed include: the number of timesteps, which refers to the number of step iterations the diffusion process goes to generate an output image; the number of features, which represents the dimensionality of the latent space or the hidden representation; the input dataset; the number of epoch iterations, which represents training iterations for the training dataset; as well as the beta rate (β). The performance of a DPM is determined by the loss between the predicted noise and the real noise added to the images, as well as the execution time and computational cost for each approach. Different choices of the hyperparameters were tested in diverse image datasets that range from datasets with low complexity such as landscape images, to datasets with higher complexity such as human faces and vehicles. The results of such images and their impact also were analyzed to understand the impact of tuning such hyperparameters. The thesis also provides observations on which hyperparameters have the highest influence when using DPMs to generate high-resolution images.

1.4 Thesis Outline

This thesis is organized into seven chapters. The first four chapters introduce the idea of Diffusion Probabilistic Models and their organization, as well as a full description of their structure, math and concepts. The last three chapters present the results and analysis from our experimentation with DPMs.

Chapter 2 describes the origins of DPMs and the use of Markov chains and neural networks to predict the noise added. It also summarizes the improvements made by later researchers on DPMs and describes the use of U-Net and its effectiveness in predicting the noise added to input images.

Chapter 3 presents our model development by outlining the flowchart and impact of various hyperparameters and by giving a detailed explanation of their influence on the model performance of the model. It also shows a comprehensive pseudo-code that implements our model.

Chapter 4 focuses on the hardware aspects used in our model and their importance for higher resolution results. It gives a broader view of the different environments used during our training, and how they compared to each other. It also highlights the importance of having a higher GPU capability to produce better performance.

Chapter 5 reports our first findings on generating landscape images from a given dataset. It shows the impact of tuning the number of feature parameters to produce higher quality images,

as well as the impact of increasing our batch size and image size for faster execution times while still having higher resolution images.

Chapter 6 reports our second findings on generating more complex images such as human faces and vehicle images. It builds on our findings from Chapter 5 to produce even higher quality images with easily interpretable results that are more concrete than our first findings.

Chapter 7 finally summarize our findings and observations from chapters 5 and 6 and describes how to obtain better resolution images without excessive energy consumption and without long execution times. Additionally, the chapter presents some future research directions.

Chapter 2 Theoretical Background

This chapter gives a high-level introduction of diffusion probabilistic models and concepts used to develop techniques for creating such models. The concepts introduce how the laws of thermodynamics were utilized in the math derivations and which neural network structure has better performance when training such models.

2.1 Origins of Diffusion Probabilistic Models

To understand diffusion models, it is important to investigate what does diffusion mean in terms of *non-equilibrium statistical physics*, which refers to the movement of particles or molecules from an area of high concentration to an area of low concentration, along the gradient of the concentration. Thus, the process used in diffusion probabilistic models is influenced by the random motion of particles derived from the laws of statistical mechanics and thermodynamics. The first researcher to introduce such concepts was Jascha Sohl-Dickstein et al. in a 2015 paper called “Deep Unsupervised Learning using Nonequilibrium Thermodynamics” [31]. In the paper, they outlined the essential idea of first systematically and slowly destroying a data distribution through an iterative forward diffusion process using a Markov chain and then learning a reverse diffusion process to reconstruct the data. They argued that one problem in machine learning models was the analytical and computational tractability of the learning, sampling, inference, and evaluation process when modeling complex datasets using probability distributions. They also argued that in a diffusion probabilistic model that uses a Markov chain, each step has an evaluable probability, and thus the full chain can by analytically evaluated [31]. This technique

involves estimating small perturbations to a diffusion process that are more tractable than trying to describe the full distribution with a single potential function.

In their paper, they label the data distribution $q(x^{(0)})$, which is then gradually converted to an analytically tractable distribution π (y) by applying the Markov diffusion kernel $T_\pi(y|y';\beta)$ for $\pi(y)$, where β is defined as the diffusion rate:

$$\pi(y) = \int dy'\ T_\pi(y|y';\beta)\ \pi(y') \tag{1}$$

$$q\ (x^{(t)}|x^{(t-1)}) = T_\pi\ (x^{(t)}\ |x^{(t-1)}\ ;\beta_t). \tag{2}$$

Therefore, the forward process starts at time 0 and takes T number of steps:

$$q\ (x^{(0\ldots T)}) = q(x^{(0)})\ \prod_{t=1}^{T} p(x^{(t-1)}\ |\ x^{(t)}). \tag{3}$$

From the previous equations, the authors were able to add the appropriate noise at each time step t going from 0 to T. This process is refer to as the forward diffusion process, which is the opposite of the reverse diffusion process whose goal is to denoise the image by going backward on the Markov diffusion kernel $T_\pi(y|y';\beta)$. Similarly, they argued that the generative distribution is trained to describe the trajectory in reverse [31], and that the reversal process has a similar functional form as the forward process:

$$p(x^{(T)}) = \pi(x^{(T)}) \tag{4}$$

$$p\ (x^{(0\ldots T)}) = p\ (x^{(T)})\ \prod_{t=1}^{T} p(x^{(t-1)}\ |\ x^{(t)}). \tag{5}$$

Note that the forward and reverse diffusion processes have similar forms. They are both continuous diffusions having a small step size β in the limit, and they are both Gaussian (binomial) distributions [13]. This is made possible given a small size step β_t for each time step.

In addition, as the trajectory described by the number of timesteps T is increased, that is, as T becomes larger, the diffusion rate β can be made significantly smaller, thus giving us Gaussian distributions in both directions.

In terms of the learning algorithm proposed by Sohl-Dickstein et al. they stated that the mean and covariance of the Gaussian diffusion kernel, or bit flip probability of such a kernel, needed to be learned [31]. To do so, the function $f_\mu(x^{(t)}, t)$ represents the mean of the reverse Markov diffusion kernel, and the function $f_\Sigma(x^{(t)}, t)$ represents the covariance of the same reverse function. Additionally, the function $f_b(x^{(t)}, t)$ represents the bit flip probability for the Gaussian distribution. Thus, the computational cost of running the algorithms presented in their paper is equal to the previous functions multiplied by the total number of time steps T. To represent such functions, they used a multi-layer perceptron, along with various regression and fitting function algorithms.

The probability assigned to the data by the generative model is given by:

$$P(x^{(0)}) = \int dx^{(1\ldots T)}\, p(x^{(0\ldots T)}). \tag{6}$$

However, they argued that this approach was intractable and therefore forced them to use the Jarzynski equality, which is an equation in statistical mechanics concerning free energy differences between two states and the irreversible work on the trajectories combining the same states. They evaluated the relative probability of the forward and reversed trajectories and then took the average over the forward trajectories in the following manner:

$$P(x^{(0)}) = \int dx^{(1\ldots T)}\, p(x^{(0\ldots T)})\, \frac{q\left(x^{(1\ldots T)} \middle| x^{(0)}\right)}{q\left(x^{(1\ldots T)} \middle| x^{(0)}\right)} \tag{3}$$

$$= \int dx^{(1\ldots T)}\, q(x^{(1\ldots T)} | x^{(0)})\, \frac{p\left(x^{(0\ldots T)}\right)}{q\left(x^{(1\ldots T)} \middle| x^{(0)}\right)} \tag{4}$$

$$= \int dx^{(1\ldots T)}\, q\,(\mathrm{x}^{(1\ldots T)} |\mathrm{x}^{(0)})\, p(\mathrm{x}^{(T)}) \prod_{t=1}^{T} \frac{p\left(x^{(t-1)}\middle|x^{(t)}\right)}{q\left(x^{(t)}\middle|x^{(t-1)}\right)} \tag{5}$$

According to them, this probability for the generative model can be evaluated by going forward in the trajectory and averaging over the samples represented by $q(\mathrm{x}^{(1\ldots T)}|\mathrm{x}^{0})$. To enable such an evaluation, β must be miniscule, and thus, both trajectories of the forward and reverse distributions can be evaluated similarly. This makes computation easier as it is only necessary to evaluate the integral with one sample from $q(\mathrm{x}^{(1\ldots T)}|\mathrm{x}^{(0)})$.

Then, they showed that the training of the model comes down to maximizing the log-likelihood [31]:

$$\mathrm{L} = \int dx^{(0)} q(\mathrm{x}^{(0)}) \log p(\mathrm{x}^{(0)})$$

$$= \int dx^{(0)} q(\mathrm{x}^{(0)}) \log \left[dx^{(1\ldots T)}\, q(\mathrm{x}^{(0\ldots T)}|\mathrm{x}^{0})\, p(\mathrm{x}^{(T)}) \prod_{t=1}^{T} \frac{p\left(x^{(t-1)}\middle|x^{(0)}\right)}{q\left(x^{(t)}\middle|x^{(t-1)}\right)}\right] \tag{10}$$

According to Jensen's inequality, which states that when one takes the expected value of a convex function g of a random variable X, it is always greater than or equal to the convex function of the expected value of that random variable, that is, $\mathrm{E}[g(\mathrm{X})] \geq g(\mathrm{E}[\mathrm{X}])$ [28]. Thus, the previous equation by Jensen's inequality has a lower bound as described by:

$$\mathrm{L} \geq \int dx^{(0\ldots T)}\, q(\mathrm{x}^{(0\ldots T)}) \log \left[\, p(\mathrm{x}^{(T)}) \prod_{t=1}^{T} \frac{p\left(x^{(t-1)}\middle|x^{(t)}\right)}{q\left(x^{(t)}\middle|x^{t-1}\right)}\right] \tag{11}$$

To analyze further, we need to understand the derivation of a conditional entropy bound. This analysis is reflected in the following equations where $H_q(\mathrm{X}^{(t-1)} \mid \mathrm{X}^{(t)})$ is the conditional entropy of one step in the reverse process:

$$H_q(\mathrm{X}^{(t-1)} \mid \mathrm{X}^{(t)}) = H_q(\mathrm{X}^{(t)} \mid \mathrm{X}^{(t-1)}) \tag{12}$$

$$H_q(X^{(t-1)} \mid X^{(t)}) + H_q(X^{(t)}) = H_q(X^{(t)} \mid X^{(t-1)}) + H_q(X^{(t-1)}) \quad (13)$$

$$H_q(X^{(t-1)} \mid X^{(t)}) = H_q(X^{(t)} \mid X^{(t-1)}) + H_q(X^{(t-1)}) - H_q(X^{(t)}) \quad (14)$$

Furthermore, when dealing with probability distributions, a distribution $\pi(y)$ has the maximum entropy among the distributions under consideration, that it is the distribution with the highest degree of randomness or uncertainty. This is true for a binomial distribution, which are discrete distributions described by parameters such as the success probability and the number of trials. Additionally, this is the case for Gaussian distributions with unit. That is, a Gaussian distribution with unit variance is the "widest" in terms of entropy compared to other Gaussian distributions [13]. Thus, to apply such concepts, our data must be scaled to have a unit norm and the data need not be whitened, that is need not be decorated. With all of these in mind, an upper bound on the entropy can be derived as follows:

$$H_q(X^{(t)}) \geq H_q(X^{(t-1)}) \quad (15)$$

$$H_q(X^{(t-1)}) - H_q(X^{(t)}) \leq 0 \quad (16)$$

$$H_q(X^{(t-1)} \mid X^{(t)}) \leq H_q(X^{(t)} \mid X^{(t-1)}) \quad (17)$$

Now, to determine a lower bound on the entropy difference, that is a limit on how much uncertainty or randomness difference in a system can decrease, we observe that as we progress through additional steps in a Markov chain, the information regarding the initial state does not change. This is the case since in Markov chains, the probability of transitioning to a future state does not depend on previous states but only the current state, and thus additional steps do not provide more information about the initial state. Furthermore, additional steps do not increase the conditional entropy of the initial state, that is the conditional entropy of the initial state remains the same or does not decrease as we move forward in the Markov chain. We can therefore write:

$$H_q(X^{(0)}|X^{(t)}) \geq H_q(X^{(0)}|\, X^{(t-1)}) \tag{18}$$

$$H_q(X^{(t-1)}) - H_q(X^{(t)}) \geq H_q(X^{(0)} \mid X^{(t-1)}) + H_q(X^{(t-1)}) - H_q(X^{(0)} \mid X^{(t)}) - H_q(X^{(t)}) \tag{19}$$

$$H_q(X^{(t-1)}) - H_q(X^{(t)}) \geq H_q(X^{(0)} \mid X^{(t-1)}) - H_q(X^{(0)}, X^{(t)}) \tag{20}$$

$$H_q(X^{(t-1)}) - H_q(X^{(t)}) \geq H_q(X^{(t-1)} \mid X^{(0)}) - H_q(X^{(t)} \mid X^{(0)}) \tag{21}$$

$$H_q(X^{(t-1)} \mid X^{(t)}) \geq H_q(X^{(t)} \mid X^{(t-1)}) - H_q(X^{(t-1)}, X^{(0)}) - H_q(X^{(t)}, X^{(0)}) \tag{22}$$

Now, by combining such expressions, we can represent the bounded conditional entropy for a single step on our trajectory as:

$$H_q(X^{(t-1)} \mid X^{(t-1)}) \geq H_q(X^{(t-1)} \mid X^{(t)}) \geq H_q(X^{(t)}, X^{(t-1)}) - H_q(X^{(t-1)}, X^{(0)}) - H_q(X^{(t)}, X^{(0)}) \tag{23}$$

Thus, based on these derivations, the upper and lower bounds on the conditional forward trajectory depend only on $q(x^{(1\ldots T)} \mid x^{(0)})$, which can be analytically computed.

Furthermore, going back to our initial lower bound on the log-likelihood in equation 11, we can analyze it with entropy where $L \geq K$, where:

$$K = \int dx^{(0\ldots T)}\, q(x^{(0\ldots T)}) \log \left[\, p(x^{(T)}) \prod_{t=1}^{T} \frac{p\left(x^{(t-1)} \middle| x^{(t)}\right)}{q\left(x^{(t)} \middle| x^{(t-1)}\right)}\right] \tag{24}$$

It follows that we can replace $p(x^{(T)})$ with entropy described earlier to obtain:

$$K = \int dx^{(0\ldots T)}\, q(x^{(0\ldots T)}) \sum_{t=1}^{T} \log \left[\frac{p\left(x^{(t-1)} \middle| x^{(t)}\right)}{q\left(x^{(t)} \middle| x^{(t-1)}\right)}\right] + \int dx^{(T)}\, q(x^{(T)}) \log p(x^{(T)}) \tag{25}$$

$$= \int dx^{(0\ldots T)}\, q(x^{(0\ldots T)}) \sum_{t=1}^{T} \log \left[\frac{p\left(x^{(t-1)} \middle| x^{(t)}\right)}{q\left(x^{(t)} \middle| x^{(t-1)}\right)}\right] + \int dx^{(T)}\, q(x^{(T)}) \log \pi(x^{(T)}) \tag{26}$$

Here, the cross entropy of the distribution $\pi(x^{(T)})$ is constant on the diffusion kernels, and since it is also the entropy of $p(x^{(T)})$, we can describe it as:

$$K = \sum_{t=1}^{T} \int dx^{(0\ldots T)} \; q(x^{(0\ldots T)}) \; \log\left[\frac{p\left(x^{(t-1)}\middle|x^{(t)}\right)}{q\left(x^{(t)}\middle|x^{(t-1)}\right)}\right] - H_p(x^{(T)}) \tag{27}$$

To get rid of edge effects, we can make the initial step of the forward diffusion process behave like the final step of the reverse process, that is, when t = 0, we write:

$$P(x^{(0)} \mid x^{(1)}) = q(x^{(1)} \mid x^{(0)}) \frac{\pi(x^{(0)})}{\pi(x^{(1)})} = T_\pi(x^{(0)} \mid x^{(1)}; \beta) \tag{28}$$

Thus, we can rewrite equation 27, where we omit the first time-step in the sum, as follows, where the sum starts at t = 2 instead of:

$$K = \sum_{t=2}^{T} \int dx^{(0\ldots T)} \; q(x^{(0\ldots T)}) \; \log\left[\frac{p\left(x^{(t-1)}\middle|x^{(t)}\right)}{q\left(x^{(t)}\middle|x^{(t-1)}\right)}\right] + \int dx^{(0)}\, dx^{(1)} q(x^{(0)}, x^{(1)})$$

$$\log\left[\frac{q\left(x^{(1)}\middle|x^{(0)}\right)\pi(x^{(1)})}{q\left(x^{(1)}\middle|x^{(0)}\right)\pi(x^{(1)})}\right] - H_p(x^{(T)}) \tag{29}$$

$$K = \sum_{t=2}^{T} \int dx^{(0\ldots T)} \; q(x^{(0\ldots T)}) \; \log\left[\frac{p\left(x^{(t-1)}\middle|x^{(t)}\right)}{q\left(x^{(t)}\middle|x^{(t-1)}\right)}\right] - H_p(x^{(T)}) \tag{30}$$

Next by using Bayes' rule, we can rearrange this in terms of the posterior, which would update the probability of an event or parameter based on new information since Bayes' rule relates the posterior probability, prior probability, likelihood, and evidence. This leads to:

$$K = \sum_{t=2}^{T} \int dx^{(0\ldots T)} \; q(x^{(0\ldots T)}) \; \log\left[\frac{p\left(x^{(t-1)}\middle|x^{(t)}\right)}{q\left(x^{(t)}\middle|x^{(t-1)}\right)} \frac{q(x^{(t-1)}\,|x^{(0)})}{q(x^{(t)}\,|x^{(0)})}\right] - H_p(x^{(T)}) \tag{31}$$

Observe that some of the terms are conditional entropies. Therefore, we can rewrite them as:

$$K = \sum_{t=2}^{T} \int dx^{(0\ldots T)} \; q(x^{(0\ldots T)}) \; \log\left[\frac{p\left(x^{(t-1)}|x^{(t)}\right)}{q\left(x^{(t)}|x^{(t-1)}\right)}\right] + \sum_{t=2}^{T}[H_q(X^{(t)} \mid X^{(0)}) + (X^{(t-1)} \mid X^{(0)})]$$

$$- H_p(x^{(T)}) \tag{32}$$

$$K = \sum_{t=2}^{T} \int dx^{(0\ldots T)} \; q(x^{(0\ldots T)}) \; \log\left[\frac{p\left(x^{(t-1)}|x^{(t)}\right)}{q\left(x^{(t)}|x^{(t-1)}\right)}\right] + H_q(X^{(T)} \mid X^{(0)}) + (X^{(1)} \mid X^{(0)})]$$

$$- H_p(x^{(T)}) \tag{33}$$

Finally, we can express our equation in terms of Kullback-Leiber (KL) divergence, also known as relative entropy, which is a measure for quantifying the difference between two probability distributions. KL divergence is needed to represent data from one distribution when it is assumed to be generated by another distribution [46]. This relative entropy is defined as $KL(P||Q) = \int P(X) \log[P(X)/Q(X)]\, dx$, where P(X) and Q(X) represent two probability distributions of an event or outcome X.

With the KL divergence, we can write:

$$K = -\sum_{t=2}^{T} \int dx^{(0)}\, dx^{(t)} \; q(x^{(0)}, x^{(t)}) \; D_{KL}(q(x^{(t-1)} \mid x^{(t)}, x^{(0)}) \,||\, p(x^{(t-1)} \mid x^{(t)})) + H_q(x^{(T)} \mid x^{(0)})$$

$$+ (x^{(1)} \mid x^{(0)})] \; - H_p(x^{(T)}) \tag{34}$$

Moreover, since both forward and reverse trajectories on the Markov chain are identical and since they both correspond to a quasi-static process, the inequality $L \geq K$ becomes an equality L=K, allowing us to carry out earlier analysis. As for training, the paper suggests that we need to find the reverse Markov chain transitions that maximize the lower bound on the log-likelihood which is presented on the equation below.

$$\underbrace{\arg\max}_{x \in S} K(X) \tag{35}$$

The task of training becomes understanding or predicting how the data is distributed and how to estimate the underlying probability distribution that generates such data. In their paper, they suggest using regression, which is a statistical technique used to model relationships among different variables. In this case, regression is used to model the function that determines the mean and covariance of a sequence of Gaussian distributions [13] based on some input features or variables. Additionally, they discussed another scenario involving Bernoulli trials, which can be used to estimate the probability of success or failure based on certain predictors. This is a common approach in statistical analysis and in machine learning contexts where understanding the underlying data distributions is important.

Another important contribution by this first publication describing DPM is the importance of setting a useful diffusion rate β_t, in a forward trajectory in the context of training a model. This choice plays a crucial role in determining how well the trained model will perform. They referred to AIS (Annealed Importance Sampling) where having the correct schedule of intermediate probability distributions can significantly enhance the accuracy of estimating the log partition function. This is a key concept in both statistical analysis and physics and is related to the understanding of complex systems. In the context of thermodynamics, the schedule used when transitioning between equilibrium distributions has an impact on how much free energy is dissipated or lost during the process. This is a critical consideration in understanding energy transformations and processes in physical systems. In the case of Gaussian distributions, we learn the forward diffusion schedule $\beta_{2...T}$ by doing a gradient ascent on L. In addition, the variance β_1 of the first step is set to a small constant to prevent overfitting, a phenomenon where a model is trained to fit the training data too closely, capturing noise or random fluctuations in the data rather than the underlying true patterns [29]. Overfitting could lead to poor generation,

where the model performs well on the training data but poorly on unseen or new data for that matter. Thus, schedule selection plays a crucial role in various fields and has a significant impact on model performance and physical processes. With all these algorithms and analysis, the authors introduced a novel solution on modeling probability distributions. They also tested it on different datasets such as CIFAR-10, which is a dataset created by the *Canadian Institute for Advanced Research* commonly used for machine learning algorithms, and the MNIST dataset, which is another dataset created by the *Modified National Institute of Standards and Technology* consisting of handwritten digits. Their results achieved a lower bound K on the log-likelihood of 5.4 ± 0.2 bits/pixel for the CIFAR-10 dataset and a log-likelihood of 317 ± 2.7 bits for the MINST dataset using their novel diffusion model [36].

The aforementioned work was a great contribution to the development of DPM since it got the machine learning and artificial intelligence community excited. The work also motivated other researchers to start to explore the core concepts of their algorithms that revolved around estimating the reversal of a Markov diffusion chain, which effectively maps data to a noise distribution [31]. As the number of steps increases in this diffusion process, the reversal distribution associated with each step becomes simpler and more straightforward to estimate.

2.2 Recent Improvements in Diffusion Probabilistic Models

In the following years, researchers started to look more into diffusion probabilistic model solutions and in 2020, Jonathan Ho et al. published a paper called *Denoising Diffusion Probabilistic Models* that showed substantial improvements to the initial publication and would become the stone pillar for the recent publications [27]. In this paper, they described progress in

DPM and showed that this type of model can generate high quality samples. They argued that the results obtained by DPM are better than those of other generative models such as Generative Adversarial Networks (GANS) or Variational Auto-Encoders (VAEs) [43]. The focal point of their study was in specific parametrization, which essentially means the way in which the model's internal parameters are defined. The authors showed that their chosen parametrization exhibits an equivalence with a technique called "denoising score matching" across various levels of noise during the training process. Denoising score matching is a statistical method used to estimate the probability distributions, and the equivalence suggests a deep connection between their parameterization and this technique [27]. Another important finding was the equivalence with "annealed Langevin dynamics" during the sampling phase. Langevin dynamics is a mathematical method for simulating the motion of particles in a physical system [35]. In their context, Langevin dynamics is a method used during the generation or sampling process of a diffusion model, and the authors found their parametrization to be connected to this method. Nevertheless, the authors also discovered that while their parameterization excelled in terms of sample quality, it did not fare as well when it comes to the log-likelihoods. They argued, however, that their models did have log-likelihoods which were better than the estimate produced by annealed importance sampling for energy-based models and score matching [27]. Additionally, they introduced the concept of "lossless codelengths" which implies that a significant portion of the models' capacity is used to describe subtle or imperceptible image detains, that is, the models may be encoding intricate information that is not readily noticeable. They presented a more refined analysis using the language of "lossy compression' which is a technique used to reduce the data size while preserving essential information. In this context, the authors applied a similar concept to their models and their data representations. They stated that

the sampling procedure of DPM is akin to “progressive decoding” which implies a step-by-step decoding process, that resembles autoregressive decoding [27], since these models generate data one element at a time, based on the previous element.

In their math derivations the authors used a similar approach to the previous model but with a distinct syntax that is the most used today. They described DPMs as latent variable models of the form $p_\Theta(X_0) := \int p_\Theta(X_{0:T})\, dx_{1:T}$, where $X_1, \ldots, X_T$ are latent variables with same dimensionality as the data $X_0 \sim q(X_0)$. In this representation the called the reverse process $p_\Theta(X_{0:T})$, and similar to the previous paper it is described by a Markov chain with learned Gaussian distribution transitions beginning at $p(X_T) = N(X_T; 0, I)$, where I represents the identity covariance matrix of same dimensionality as the variable element X, which in the case for diffusion models represents an image, and T is equal to the total number of time-steps in the diffusion process. With these notations we can write:

$$p_\Theta(X_{0:T}) := p(X_T) \prod_{t=1}^{T} p_\Theta(X_{t-1} \mid X_t) \tag{36}$$

$$p_\Theta(X_{t-1} \mid X_t) := N(X_{t-1}; \mu_\Theta(X_t, t), \Sigma_\Theta(X_t, t)) \tag{37}$$

What makes DPM different than most types of latent variable models are that the approximate posterior $q(X_{1:T} \mid X_0)$ is fixed to a Markov chain. These models are designed to learn and generate data by modeling a progression from an initial state X_0 to the final step X_T. This in turn characterizes what the authors called the forward or diffusion process within the model. In this case, the Markov chain is designed to gradually introduce Gaussian noise to the data as it progresses from X_0 to X_T. Another crucial aspect the author mentioned is that the noise added at each step in the Markov chain is governed by a variance schedule, denoted as $\beta_1, \beta_2 \ldots, \beta_T$. This schedule specifies how the variance of the Gaussian noise changes over time, affecting the level

of randomness introduced into the data as it evolves. Specifically, the forward process is giving by:

$$q(X_{1:T} \mid X_0) := \prod_{t=1}^{T} q\,(X_t \mid X_{t-1}) \tag{38}$$

$$q(X_t \mid X_{t-1}) := N(X_t; \sqrt{1-\beta_t}\,X_{t-1}, \beta_t, I) \tag{39}$$

And like the paper published in 2015, the authors optimized the usual variational bound on negative log-likelihood for the training of the DPM as follows:

$$E\,[-\log p_\Theta(X_0)] \leq E_q\,[-\log \frac{p\Theta\,(X_{0:T})}{q(X_{1:T}|X_0)}] = E_q\,[-\log(X_T) - \sum_{t\geq 1} log \frac{p\Theta(X_{t-1}|X_t)}{q(X_t|X_{t-1})}] =: L \tag{40}$$

One of the contributions from this paper is the reparameterization of the forward process variances denoted previously as β_t which can be held constant as hyperparameters [27]. This suggests that, instead of learning them, these variances can be predefined based on prior knowledge or as part of the model design. Additionally, the reverse process is partly significant because it ensures that both the forward and reverse processes share the same functional form, especially when β_t is small. This symmetry in functional form makes the modeling process simpler. The symmetry was also discussed in the previous paper by Jascha Sohl-Dickstein et al. However, in this new approach, Jonathan Ho et al. explained that a notable property of the forward process is the ability to sample any X_t at any arbitrary timestep (t) in a closed form [27]. This suggests that one can readily generate data at different points in the diffusion process, which can be advantageous for certain applications. They also introduced a remarkable notation that is referenced in many subsequent papers, normally $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, with this notation, the forward distribution can be rewritten as:

$$q\,((X_t \mid X_0) := N(X_t; \sqrt{\bar{\alpha}_t}\,X_0, (1-\bar{\alpha}_t), I) \tag{41}$$

As for training, they proposed a similar approach to the first paper which was to use KL divergence to directly compare $p_\Theta(X_{t-1} \mid X_t)$. Moreover, the forward process posteriors must meet the following criteria X_0 on:

$$q(Xt_{-1} \mid X_t, X_0) = N(X_{t-1}; \tilde{\mu}_t(X_t, X_0), \tilde{\beta}_t \mathbf{I}) \quad (42)$$

where the value of $\tilde{\mu}_t(X_t, X_0)$ is giving by:

$$\tilde{\mu}_t(X_t, X_0) := \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} X_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} X_t \text{ and } \tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t \quad (43)$$

With these criteria in mind, the authors of the paper used the KL divergences to measure differences between Gaussian probability distributions. They introduced the term "Rao-Blackwellized" to describe the method used for calculating the KL divergence, which is a statistical technique used to reduce the variance of estimators [17]. They used it in a way that minimizes estimation error and reduces variability, since the divergences are calculated using "closed form expressions." This means that there are explicit mathematical equations that can be used to compute these divergences in place of Monte Carlo estimates, which are popular methods for numerical estimation that can introduce randomness and variability which can be problematic when calculating divergences.

The authors also explained that efficient training is possible by optimizing random terms of *L* (see equation 40) with stochastic gradient descent [16]. Furthermore, additional improvement can be defined by rewriting *L* as:

$$E_q [\underbrace{D_{KL}(q(X_T|X_0)||p(X_T))}_{L_T} + \Sigma_{t>1} \underbrace{D_{KL}(q(X_{t-2}|X_t, X_0)||p_\Theta(X_{t-1}|X_1))}_{L_{t-1}} \underbrace{-logp_\Theta(X_0|X_1)}_{L_0}] \quad (44)$$

DPMs might seem like a constrained type of latent variable models. However, they offer significant degrees of freedom in their implementation. Degrees of freedom refer to the various choices and parameters that can be adjusted during model construction. To effectively utilize diffusion models, one needs to make crucial choices. Specifically, two key aspects need attention, namely, the variances β_t and the model architecture and Gaussian distribution parametrization. For the variances in the forward process, they dictate how the data evolve over time in the forward distribution. Thus, deciding on the values of β_t is critical in the model design. The reverse process, as well as the overall model architecture, needs to be carefully considered. This involves selecting how Gaussian distributions are parametrized and specifying the model structure. Additionally, the authors established a new and explicit connection between diffusion models and denoising score matching. This connection implies that insights from denoising score matching can be applied to enhanced diffusion models. The connection also results in a simplified, weighted variational bound objective for DPMs, making them more practical. The choices made in the paper results for the model design are guided by considerations of simplicity and empirical results. This suggests that design decisions were made to prioritize model simplicity while ensuring that the model performs well in practice.

An important aspect from Jonathan Ho et al. is that the learnability of the forward variances β_t is intentionally ignored [27]. Typically, in machine learning models, parameters are learned from the data during training. However, in this case, the authors decided to treat β_t as non-learnable. Thus, these variances were chosen to be fixed constants. This means that they have predefined values for $\beta_{t,}$ which do not change during training. An important implication of fixing β_t is that the approximate posterior distribution q has no learnable parameters, thus having static properties. Moreover, and L_T remains unchanged throughout training. As L_T is constant during

training, it can be effectively disregarded when analyzing the model's behavior. For the reverse process and $L_{1:T-1}$, the authors discussed their choices in the backward distribution $p_\Theta(X_{t-1} \mid X_t) := N(X_{t-1}; \mu_\Theta(X_t, t), \Sigma_\Theta(X_t, t))$ (see equation 37). They let the covariance matrix be $\Sigma_\Theta(X_t, t) = \sigma_t^2 \mathbf{I}$. Thus, $\Sigma_\Theta(X_t, t)$ becomes a covariance matrix with identical variances $\sigma_t^2 \mathbf{I}$ along its diagonal. These variances are untrained, meaning they are not learned from the data but treated as constants. According to them, in their experiments both $\sigma_t^2 = \beta_t$ and $\sigma_t^2 = \tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$ yielded similar results [27]. They also provided insights into the optimality of these choices under different conditions. For instance, they noted that the first choice $\sigma_t^2 = \beta_t$ is considered optimal when the initial state, X_0, followed a Gaussian distribution with mean 0 and a unit variance, $X_0 \sim N(0, \mathbf{I})$. In contrast, the second choice $\sigma_t^2 = \tilde{\beta}_t$ is deemed optimal when the initial state X_0 is deterministically set to a specific point. These choices represent two extreme scenarios that correspond to upper and lower bounds on the reverse process entropy.

The second choice they made in their approach was to represent the mean $\mu_\Theta = (X_t, t)$ by following a new reparameterization based on their analysis on L_t.

From their analysis and with forward diffusion process $p_\Theta(X_{t-1} \mid X_t) := N(X_{t-1}; \mu_\Theta(X_t, t), \sigma_t^2 \mathbf{I})$, which is a normal distribution with mean $\mu_\Theta(X_t, t)$ and covariance matrix by $\sigma_t^2 \mathbf{I}$, they rewrote L_{t-1} as:

$$L_{t-1} = E_q \left[\frac{1}{2\sigma_t^2} \| \tilde{\mu}_t(X_t, X_0) - \tilde{\mu}_\Theta(X_t, t)\|^2\right] + C \quad (45)$$

where C is a constant independent of the value of Θ. Therefore, to parameterize the mean μ_Θ, they introduce a model that predicts $\tilde{\mu}_t$, which is the forward process posterior mean.

Furthermore, the authors reparametrized equation 41 as $X_t(X_0, \varepsilon) = \sqrt{\bar{\alpha}_t} X_t + \sqrt{\overline{1-\alpha_t}}\varepsilon$ for

$\varepsilon \sim N(0, \mathbf{I})$, and then they apply the forward process posterior discussed in the previous formula in equation 41. This lead to:

$$L_{t-1} - C = E_{X0,\varepsilon}\left[\frac{1}{2\sigma_t^2}\| \tilde{\mu}_t(X_t, X_0), -\frac{1}{\sqrt{\overline{\alpha_t}}}(X_t(X_0, \varepsilon) - \sqrt{1-\alpha_t}\varepsilon) - \mu_\Theta(X_t(X_0, \varepsilon), t)\|^2\right] \quad (46)$$

$$L_{t-1} - C = E_{X0,\varepsilon}\left[\frac{1}{2\sigma_t^2}\| \frac{1}{\sqrt{\overline{\alpha_t}}}(X_t(X_0, \varepsilon) - \frac{\beta_t}{\sqrt{1-\alpha_t}}\varepsilon - \mu_\Theta(X_t(X_0, \varepsilon), t)\|^2\right] \quad (47)$$

With equation 47, the authors noted that the mean μ_Θ must predict $\frac{1}{\sqrt{\overline{\alpha_t}}}(X_t - \frac{\beta_t}{\sqrt{1-\alpha_t}})\varepsilon$ with a given X_t at any time *t.* The authors then made the following parametrization suggestion from the input X_t:

$$\mu_\Theta(X_t, t) = \tilde{\mu}_t(X_t, \frac{1}{\sqrt{\overline{\alpha_t}}}(X_t - \sqrt{1-\alpha_t}\varepsilon_\Theta(X_t))) = \frac{1}{\sqrt{\alpha_t}}(X_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha_t}}}\varepsilon_\Theta(X_t, t)) \quad (48)$$

In this equation, ε_Θ is used to predict the true noise ε from the noised image X_t.

To sample the denoised image from X_t, Jonathan et al. [27] came up with two novel algorithms that are widely cited in subsequent work. The first algorithm is used to train the noise predicted ε_Θ, while the second algorithm is used to sample and reduced the noise of X_t. The two algorithms are listed below for completeness:

First Algorithm: Training

- **Repeat**:
    - $x_0 \sim q(x_0)$
    - $t \sim$ Uniform ({1,...,T})
    - $\varepsilon \sim N(0, I)$
    - Take the gradient descent step on

- $\nabla_\theta \| \varepsilon - \varepsilon_\theta\left(\sqrt{\bar{\alpha}_t}x_0\right) + \sqrt{1-\bar{\alpha}_t}\varepsilon, t \|^2$

- **Until** converged.

Second Algorithm: Sampling

- $x_T \sim N(0, I)$
- **For** t = T,…,1 **do**
  - z ~ N (0, I) if t > 1, else z = 0
  - $x_{t-1} = \frac{1}{\alpha_t}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}}\right) \varepsilon_\theta\ (x_t, t) + \sigma_t z$
- **End** for loop and return x_0

Notice that in the second algorithm to sample $X_{t-1} \sim p_\Theta (X_{t-1} \mid X_t)$ is to compute $x_{t-1} = \frac{1}{\alpha_t}(x_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}})\ \varepsilon_\theta\ (x_t, t)$, where z ~ N(0,I). In other words, z represents a normal distribution with mean zero and unit variance. The authors also pointed out how the second algorithm resembles Langevin dynamics with ε_Θ representing the learned gradient of the data density [35]. The authors referred to them as stochastic gradient Langevin dynamics, which are a variant of Langevin dynamics that incorporate the principle of gradient descent (deterministic) and stochasticity (randomness) to optimize complex, high-dimensional objective functions [22]. Such dynamics are often used in Bayesian statistics and probabilistic modeling to draw samples from complex posterior distributions, including the second algorithm where they are used to sample the data distribution.

Equation 47 may be simplified further to produce:

$$L_{t-1} - C = E_{X0,\,\varepsilon}\left[\frac{\beta_t^2}{2\sigma_t^2\alpha_t\,(1-\bar{\alpha}_t)} \| \varepsilon - \varepsilon_\Theta(\sqrt{\bar{\alpha}_t}X_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon, t)\|^2\right] \quad (49)$$

From this analysis, the optimization process, akin to denoising score matching, is equivalent to using variational interference to fit the finite-time marginal of sampling chain resembling Langevin dynamics. Variational interference is a method used in probabilistic modeling to approximate complex probability distributions [40]. Here, it is used to approximate the marginal distributions of a Langevin-like process. For the training of the mean function approximator denoted as μ_Θ for the reverse process, it is used to predict either $\tilde{\mu}_t$ or ε, which is the true noise of the distribution. The choice between these two parameterizations is a design decision in the modeling process. In their experiments, the authors first began with predicting the initial state X_0, but this led to inferior sample quality. This indicates that different choices in the parameterization of μ_Θ can have varying effects on the performance of the diffusion model. However, the ε-prediction, which resembles Langevin dynamics, simplifies the diffusion model's variational bound concerning the prediction of the true noise ε. Nonetheless, this parameterization is just one of many possible choices for the distribution $p_\Theta(X_{t-1} \mid X_t)$.

The paper [27] assumes that the nature of the data is that image pixel values are integers within the range of {0,1,…,255}. These values are then linearly scaled to the range of [-1,1]. This scaling operation transforms the pixel values so that they are consistently represented within a standard interval. The reason behind the linear scaling is to ensure that the neural network reverse process, which is a component of the diffusion model, operates on inputs that have been consistently transformed to the range [-1,1]. This transformation is important for the model's operation to ensure that the data matches the standard normal prior distribution $p(X_T)$. The authors also described the desire to obtain discrete log-likelihoods, which measure the goodness of fit between the model prediction and the actual data in discrete terms (e.g., integers from 0 to 255). To obtain discrete log-likelihoods, the last term of the reverse process is modified. Instead

of a continuous decoder derived from a Gaussian distribution $N(X_0; \mu_\theta(X_1, 1), \sigma_1^2 I)$, the following Gaussian distribution is used as a source for generating discrete data:

$$p_\theta(X_0|X_1) = \prod_{i=1}^{D} \int_{\delta_-(x_0^i)}^{\delta_+(x_0^i)} N(x;\ \mu_0^i(x_1, 1), \sigma_1^2) dx \tag{50}$$

where:

$$\delta_+(x) = \begin{cases} \infty & \text{if } x = 1 \\ x + \frac{1}{5} & \text{if } x < 1 \end{cases} \tag{51}$$

$$\delta_-(x) = \begin{cases} -\infty & \text{if } x = -1 \\ x - \frac{1}{5} & \text{if } x > -1 \end{cases} \tag{52}$$

In equation 50, D represents the dimensionality of the data. In other words, it is the number of coordinates or features used to describe the data. The authors also mentioned the choice of using a particular type of decoder for a machine learning model. They argued that although a more powerful decoder such as conditional autoregressive model is an option, they decided to use an approach like discrete distributions used in VAE decoders and autoregressive models [34]. VAEs are a type of generative models used for learning representation of data, and autoregressive models are another type of models commonly used in sequence generation modeling [32]. One of the advantages of using this approach is that it ensures that the variational bound is "lossless codelength" of discrete data. In other words, it is a way to represent discrete data without introducing loss or errors in the representation. Additionally, unlike other methods, this approach does not require adding noise to the data or incorporating the Jacobian of the scaling operation into the log-likelihood. Noise addition is a common technique to improve the robustness of models, and Jacobian is often used in transformations involving derivatives. They mentioned at the end of their sampling algorithm that they were able to display the learned mean $\mu_\theta(X_1, 1)$,

without any noise present. Moreover, the variational bound is derived from equations 49 and 50. This variational bound is a technique in machine learning for approximating complex probability distributions. They mentioned that the variational bound is differentiable with respect to Θ, which is a crucial property for training models as it allows for gradient-based optimization methods. Training typically involves adjusting model parameters Θ to make the model perform better on the given task, in this case through the minimization of the loss function. They also stated that although the approach described so far is feasible for training, they found it beneficial to train on a slightly modified version of the variational bound [27]. This modified variant enhances the quality of sample and is simpler to implement:

$$L_{simple}(\Theta) := E_{t,X0,\varepsilon}[\,||\,\varepsilon - \varepsilon_{\Theta}(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon, t)||^2\,] \quad (53)$$

In this equation, the variable *t* is uniformly sampled from a range between 1 and T, where T is the total number of time steps used in the model. When *t* is equal to one, it corresponds to L_0. In this case, this approximates a discrete decoder definition from equation 50 using the Gaussian probability density function times the bin width, while ignoring σ_1^2 and edge effects. For *t* greater than one, the situation corresponds to a version of equation 49 that is unweighted, like the loss weighting function used by the NCSN denoising score matching model. However, L_t does not appear since the variances β_t for the forward process are fixed as discussed earlier. Thus, L_t is irrelevant. This simplified objective is noted to down-weight loss terms corresponding to small *t*. This down-weighting is beneficial because it allows the network to focus on more challenging denoising tasks at larger *t* values. In essence, it helps the network prioritize more complex denoising tasks over simpler ones.

Based on this approach, the authors were able to generate promising results. They presented several performance metrics including the Frechet- Inception Distance (FID) scores, and negative likelihoods (lossless codelengths) [27]. These metrics are commonly used in the evaluation of generative models. The metrics were evaluated on the CIFAR10 dataset, which is a well-known dataset in machine learning and computer vision. It contains images categorized into ten different classes. This is also the same dataset used in the previous publication. The authors also used a U-Net architecture for their model, which is a type of Convolutional Neural Network (CNN) widely used in DPM. They reported an FID score of 3.17. FID is a measure of the similarity between real and generated images, and the lower the FID score, the better the sample quality. This FID score of 3.17 was computed with respect to the training dataset. The FID score with respect to the test dataset was 5.24, which they argued is better than many of the FID scores reported in the literature at the time.

2.3 U-Net Architecture

To update the hyperparameters of a DPM, the U-Net architecture is of much help due to its properties and functions that sample images in an efficient way. This architecture was first introduced by Olaf Ronneberg, Phillip Fischer, and Thomas Box in 2015 [41]. The architecture was initially designed for use in biomedical image segmentation but has since found applications in various domains due to its efficacy and high flexibility. The architecture contains a network that can be divided into two parts: the contracting path (on the left side) and the expansive path (on the right side). This division takes a U-shape and is the reason architecture is referred to as U-Net.The contracting path follows a typical convolutional network structure. It involves repeated 3x3 convolutions without padding (unpadded convolutions), followed by a rectified linear unit (ReLU) activation function, and a 2x2 max-pooling operation with a stride of 2 for

down sampling [26]. Additionally, at each down sampling step, the number of feature channels is doubled. In contrast, the expansive path is responsible for up sampling and refining feature maps. This path includes an up-sampling operation, a 2x2 convolution ("up-convolution") to halve the number of feature channels, concatenation with a cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU activation function [26]. The cropping is due to the loss of border pixels during convolutions. The final layer employs a 1x1 convolution to map each 64-component feature vector to the desired number of classes. This final step is crucial for producing segmentation results. The network comprises a total of 23 layers, which illustrates the depth and complexity of the model as shown in Figure 2.1. The selection of the input tile size is important so that all 2x2 max-pooling operations are applied to a layer with an even x- and y-sizes. This ensures that the network produces segmentation maps that can be seamlessly tied, which is important for processing and analyzing larger images or databases, as is the case with DPM. In U-Net, the input images and their corresponding segmentation maps are used to train the neural network [41]. This is a common practice in supervised learning, where the network learns to produce segmentation maps that match the ground truth maps. The network is trained using stochastic gradient descent (SGD), and the implementation is done using Caffe, a deep learning framework [50]. This SGD is a widely used optimization algorithm for training neural networks. Due to the use of unpadded convolutions previously mentioned, the output image is smaller than the input by a constant border width. This can be a significant detail for maintaining consistency in segmentation results.

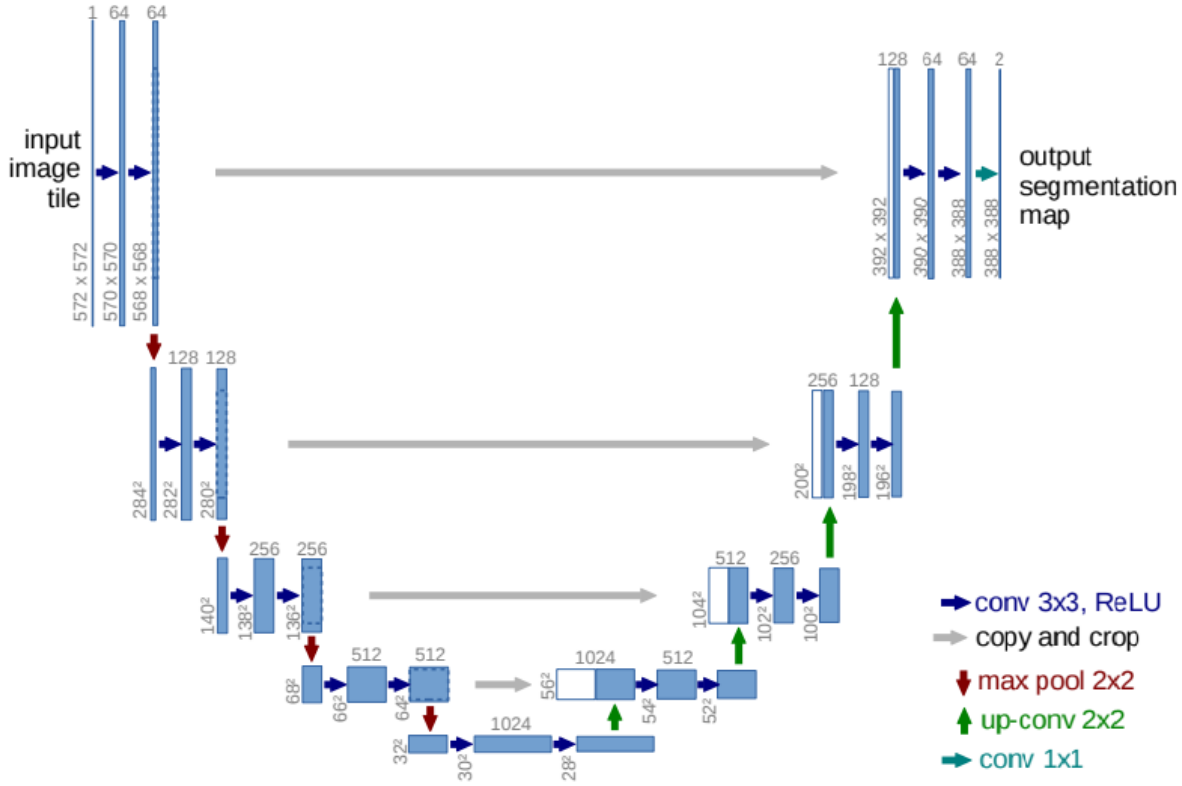


Figure 2.1: U-Net Architecture [41]

Each blue block in Figure 2.1 corresponds to a multi-channel feature map, and the number of such channels is indicated on top of each block. The x-y size is indicated in the lower left corner of each block. The white boxes are the copied feature maps from the left path.

To optimize GPU memory usage, it is favorable to have a large input tile over a large batch size according to the authors of U-Net. In this thesis, the batch size is reduced to a single image. Thus minimizing memory overhead. Using a single image per batch can be useful when working with large images or limited GPU memory. A high momentum value of 0.99 is used in the optimization process. Momentum in SGD helps accelerate training by allowing a larger number of previously seen training samples to determine the update in the current optimization step. The

energy function used for training is computed based on the final feature map. It involves a pixel-wise soft-max operation combined with the cross-entropy loss function. The soft-max operation is used to transform raw activations into class probabilities. In their paper entitled *U-net: Convolutional Networks for Biomedical Image Segmentation,* Ronneberger et al. [41] defined the soft-max function as:

$$p_{\mathrm{k}}(\mathrm{X}) = \frac{\mathrm{e}^{a_{k\prime}(\mathrm{X})}}{\sum_{k\prime}^{K} \mathrm{e}^{a_{k\prime}(\mathrm{X})}} \tag{54}$$

where $a_k(\mathrm{X})$ denotes the activation feature channel k at the pixel position x ∈ Ω, with $\Omega \subset \mathbb{Z}^2$. The integer K is the number of classes and p_k (X) is the approximated maximum-function, with high values corresponding to the class with the maximum activation and low values for all other classes. The cross-entropy loss is used to penalize the difference between the predicted class probabilities $p\ell_{(x)}$ (X) and the target value, which is typically 1 for the correct class and 0 for all other classes [41].

The cross entropy is giving by:

$$E = \sum_{\mathrm{X}\in\Omega} w(\mathrm{X}) \log\left(p\,\ell_{(x)}(\mathrm{X})\right) \tag{55}$$

where ℓ: $\Omega \rightarrow \{1,..,K\}$ is the true label of each pixel and w: $\Omega \rightarrow \mathbb{R}$ is a weight map that is introduced to give more priority to certain pixels during the training. For instance, the authors of U-Net precomputed the weight maps for each ground truth segmentation. These weight maps are used to adjust the importance of different pixels in the training data. The goal is to account for variations in pixel frequency within specific classes in the training dataset. One of the reasons for creating these weight maps is to address class imbalance in the dataset due to certain classes or categories having more or fewer pixels than others.

The weight maps aim to balance this effect, ensuring that the network learns from all classes effectively. They also encourage the network to learn the small separation borders between touching cells [41]. This is an important aspect of the task, as it helps the network distinguish between objects or regions that are in close proximity. The authors mentioned that the separation border is computed using morphological operations, which are a set of image processing techniques used for analyzing and processing shapes and structures in images. The weight map is given by:

$$w(x) = w_c(x) + w_0\, e^{\left(-\frac{d_1(x)+ d_2(x)^2}{2\sigma^2}\right)} \tag{56}$$

where $w_c(x) \rightarrow \mathbb{R}$ is the weight map to balance the class frequencies, $d_1 : \Omega \rightarrow \mathbb{R}$ represents the distance to the border of the nearest cell, and $d_2 : \Omega \rightarrow \mathbb{R}$ represents the distance to the border of the second nearest cell. Weight initialization is important in a deep neural network with many convolutional layers and complex network architectures such as the ones present in U-Net. This is because an inadequate initialization can lead to issues where certain parts of the network are overactivated, while others do not contribute effectively to the learning process. The authors remarked that an ideal weight utilization should be set such that each feature map in the network has a unit variance [41]. This ensures that all parts of the network can contribute meaningfully to the learning process. They also added that one way to achieve a desired weight utilization is by drawing the initial weights from a Gaussian distribution. The standard deviation, σ, of this Gaussian distribution should be $\sqrt{\frac{2}{N}}$, where N represents the number of incoming nodes of one neuron [41].

Another key factor for a successful U-Net is data augmentation in teaching a neural network the desired invariance and robustness properties. Data augmentation is a technique commonly used

to artificially increase the size of a training dataset by applying various transformations to the existing data. The need for data augmentation is particularly pronounced when there are only a few training samples available. In such cases, data augmentation is crucial for the network to learn effectively and generalize well to unseen data. The desired properties that the network needs to learn through data augmentation includes shift and rotation invariance, as well as robustness to deformations and variations in gray values. These properties are essential for processing microscopic images effectively. Additionally, random elastic deformations are especially important in training a segmentation network with very few annotated images. Elastic deformations involve introducing flexibility and distortion into the training samples, which is a key concept to improve the network's ability to handle variability in the data. To implement random elastic deformations, the authors generated smooth deformations using random displacement vectors on a coarse grid [41]. These deformations mimic the kind of variations that might occur in microscopic images. The displacement vectors are sampled from Gaussian distributions with a standard deviation of 10 pixels. This choice of standard deviation controls the degree of deformations introduced into the training samples. Moreover, pre-pixel displacements are computed using bicubic interpolation. Bicubic interpolation is a method for estimating pixel values during deformations, which helps create smooth and realistic distortions. Additionally, drop-out layers are used at the end of the contracting path. Drop-out is a technique that helps prevent overfitting by randomly deactivating a portion of neurons during training. In this context, it serves as another form of implicit data augmentation [41]. To dive deeper into the understanding of the U-Net architecture, we must understand how image convolution and max-pooling as well as the soft-max layer functionally work.

2.3.1 Image Convolution

Image convolution works similarly to a one-dimensional convolution, a fundamental concept in Digital Signal Processing (DSP). It suggests that images can be viewed as the sum of impulse, which are scaled and shifted delta functions [38]. Linear systems are characterized by their response to impulses called impulse responses. The output image from a linear system is a result of convolving the input image with the system's impulse response. Convolution is a mathematical operation that combines two functions to produce a third function, often used to transform one signal to another [4]. With image processing, the concept of a two-dimensional delta function is introduced. This function is depicted as an image with all zeros except for a single pixel at row 0 and column 0, which has a value of one. This representation is fundamental for understanding the convolution process in image processing. The row and column indices can have both positive and negative values, ensuring that the one (non-zero pixel) is at the center of a field of zeros. When this two-dimensional delta function is passed through a linear system, the single non-zero-point produces a two-dimensional pattern. The impulse response is the term used in image processing to describe this transformation, and it is often referred to as the point spread function (PSF) [4]. We can relate this concept to the human eye. The retina of the eye transforms an image represented as a pattern of light into an image presented as a pattern of nerve impulses. Moreover, assuming the system is linear, the processing within the retina can be understood by examining the image that appears at the optic nerve. The goal is to determine the PSF of the processing, which is a key concept in understanding how the human visual system works [45]. In a more objective sense, convolution is a straightforward mathematical operation. Convolution is a method for multiplying together two arrays of numbers. These arrays typically have different

sizes but share the same dimensionality [38]. The outcome of this multiplication is another array of numbers, and it maintains the same dimensionality. Convolution is applied in image processing, particularly in implementing operators that transform input pixel values into output pixel values. The transformed pixel values are usually obtained as simple linear combinations of specific input pixel values. In an image processing context, one of the input arrays is often a gray-level image. This image serves as the main data source for the convolution operation. The second input array, known as the "kernel" is usually much smaller in size compared to the main image, but it maintains a two-dimensional structure [1]. To illustrate such a scenario, consider an image depicted in Figure 2.2 and a kernel depicted in Figure 2.3

$I_{1,1}$	$I_{1,2}$	$I_{1,3}$	$I_{1,4}$	$I_{1,5}$	$I_{1,6}$
$I_{2,1}$	$I_{2,2}$	$I_{2,3}$	$I_{2,4}$	$I_{2,5}$	$I_{2,6}$
$I_{3,1}$	$I_{3,2}$	$I_{3,3}$	$I_{3,4}$	$I_{3,5}$	$I_{3,6}$
$I_{4,1}$	$I_{4,2}$	$I_{4,3}$	$I_{4,4}$	$I_{4,5}$	$I_{4,6}$
$I_{5,1}$	$I_{5,2}$	$I_{5,3}$	$I_{5,4}$	$I_{5,5}$	$I_{5,6}$
$I_{6,1}$	$I_{6,2}$	$I_{6,3}$	$I_{6,4}$	$I_{6,5}$	$I_{6,6}$

Figure 2.2: Image representation of size 6x6

$K_{1,1}$	$K_{1,2}$	$K_{1,3}$
$K_{2,1}$	$K_{2,2}$	$K_{2,3}$
$K_{3,1}$	$K_{3,2}$	$K_{3,3}$

Figure 2.3: Small kernel representation of size 3x3

The matrix in Figure 2.2 represents a small image of size *M* X *N*, which is 6x6 in this case, whereas the matrix on figure 2.3 represents a small kernel of size *m* x *n*, which is 3x3 in this case.

The method of convolution is performed by sliding the kernel, which is typically the smaller two-dimensional image, over the larger image. The process typically starts at the top left corner of the image and proceeds by moving the kernel through all positions where it can fit entirely within the boundaries of the image. Each position of the kernel corresponds to a single output pixel in the result image [1]. The value of this output is calculated by performing the following steps:

- For each cell in the kernel, multiply the kernel's value with the underlying image pixel value that corresponds to that cell.
- Then, add together all these multiplied values. Mathematically, the process is described by:

$$O\,(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I\,(i+k-1, j+l-1)\,K\,(k,l) \tag{57}$$

where $O(i, j)$ is a pixel obtained by convoluting the image and the kernel, M is the number of rows in the image, N is the number of columns, m is the number of rows in the kernel, and n is the number of columns. The output image O will have $M - m + 1$ rows and $N - n + 1$ columns.

2.3.2 Max-Pooling

Another important computation performed by U-Net is max-pooling. Max-pooling is a layer situated after the convolution layer in a CNN architecture such as U-Net. The primary purpose of the max pooling layer is to reduce the spatial dimensions of the feature maps produced by the

convolution layer. Despite the reduction in dimensions, the layer aims to retain the most important features from the feature map [10]. Thus, the max pooling layer takes as input the feature map generated by the preceding convolution layer. This layer applies the max pooling operation on the feature map to produce a reduced version of the feature map. This new feature map is then forwarded to subsequent layers within the convolutional neural network for further processing [10].

To perform this max-pooling operation, two parameters must be defined such as the window size and the stride. The window size is a fundamental parameter for the max pooling layer, which is essentially a two-dimensional array that traverses the feature map and performs the pooling operation. The window size must be defined when configuring the max pooling layer, and its specific value can vary depending on the implementation. A most commonly used window size is typically a 2x2 window size. The other fundamental parameter on a max pooling operation is the stride, which is the number of pixel “jumps” that a feature map makes per max-pooling. The stride value determines how many pixels the window moves after each operation. A large stride results in larger jumps, while a small stride leads to smaller jumps.

25	22	18	10
15	25	0	12
20	35	10	45
55	40	10	35

↓

25	18
55	45

Figure 2.4 - Max pooling applied to a squared matrix of size 4x4.

The top image in Figure 2.4 represents the feature map of an image of size 4x4, while the bottom image represents the resulting feature map of size 2x2 due to the max-pooling operation.

By decreasing the spatial dimensions of the feature map, the network requires fewer computations, resulting in a faster and more efficient performance which is vital when a large dataset and complex computations are part of the model, which is often the case in U-Net and DPMs. Processing speed is critical especially in a real-time application. Since max-pooling involves selecting the maximum value within a specified region of the feature map, the process ensures that the most salient or significant feature in that region is preserved and propagated forward through the network [15]. Another notable benefit of the max-pooling layer is its ability to reduce overfitting in a neural network model. Overfitting occurs when the network captures noise or irrelevant details in the data, making it less capable of generalizing to new unseen data [29]. The layer also helps alleviate overfitting by discarding noise and less informative elements within the input feature map, which is of vital usage in the case of DPM. This leads to a more robust and reliable model that can generalize better to a wider range of data.

2.3.3 Soft-Max Function

The soft-max function is a type of activation function used in neural computing [21]. This function is integral to the functionality of a neural network, especially in the U-Net architecture because it gives non-linearity to the neural network, without which the neural network would behave as a linear regression model. Moreover, the primary purpose of the soft-max function is to compute a probability distribution from a vector of real numbers. The function transforms real-numbered input values into a range of output values between 0 and 1. The soft-max function ensures that the sum of the computed probabilities equals one [2]. This is a fundamental characteristic of a probability distribution, where each value represents the likelihood of a specific outcome or class. The soft-max function is particularly useful for tasks where the goal is to determine the probability of various classes or outcomes. It is defined as:

$$Soft\text{-}max(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{59}$$

where z represents a vector of values derived from the neurons in the output layer of a neural network. These values, often referred to as logits, represent the network's raw or unprocessed predictions for each class or category [2]. The exponential function in equation 59 introduces non-linearity in the transformation of these values. It does so by exponentiating each value, which can magnify the differences between the logits. After the exponentiation, these values need to be normalized and converted into probabilities. This normalization is achieved by dividing each exponentiated value by the sum of all exponentiated values. This ensures that the resulting values fall within the range between 0 and 1 and add up to 1, adhering to the characteristics of a probability distribution. Additionally, the soft-max function allows for a straightforward comparison of class probabilities. It enables the model to select the class with the

highest probability as the predicted class. This function is commonly used in conjunction with the cross-entropy loss function [2]. This combination facilitates efficient training of neural networks, as the gradients are well-defined and suitable for backpropagation [48]. Moreover, the soft-max function is versatile and can handle a varying number of classes, making it suitable for a wide range of applications, including image classification, natural language processing, and image generation as the part of the U-Net architecture.

2.3.4 Adams Optimizer

Another fundamental component to obtain better training results in U-Net and in our model optimization is the Adams optimizer, which is short for Adaptive Moment Estimation. The Adams optimizer is a popular optimization algorithm used in training machine learning models, particularly in the context of deep learning [19]. It combines the benefits of two other popular optimization algorithms: the Momentum optimizer and the RMSprop optimizer. Adams is known for its efficiency and ability to adapt its learning rates during training, making it well-suited for a wide range of optimization tasks. It was first introduced by Diederik P. Kingma and Jimmy Ba in their 2014 paper entitled “Adam: A Method for Stochastic Optimization” [19]. The optimizer was designed to address some of the limitations of other optimization techniques such as the need for manual tuning of learning rates and the potential for slow convergence. Adams maintains two moving averages: the first moment (mean) and the second moment (uncentered variance) of gradients. The first is akin to the exponentially weighed average used in the Momentum optimizer that helps smooth the gradient updates [48]. The second is like the moving average of the squared gradients used in RMSprop that helps adapt the learning rate for each parameter. The algorithm for this optimizer is as follows:

- Define $\beta_1, \beta_2 \in [0, 1)$, which are the exponential decay rates and α, which is the step size.
- Define $f(\theta)$, which is the stochastic objective function, and θ_0, which is the initial parameter vector.
- Initialize m_0 and v_0, which are the first and second moment vectors respectively, and set the timestep variable t to 0.
- While θ_t has not converged do:
    - $g_t = \nabla_\theta f_t(\theta_{t-1})$, get the gradients with the stochastic objective at time t
    - $m_t = \beta_1 (m_{t-1}) + (1-\beta_1)(g_t)$, update the bias of the first moment estimate m
    - $v_t = \beta_2 (v_{t-1}) + (1-\beta_2)(g_t^2)$, update the bias of the second raw moment estimate v
    - $\hat{m}_t = \frac{m_t}{(1-ß_1^t)}$, compute the bias-corrected first moment estimate m
    - $\hat{v}_t = \frac{m_t}{(1-ß_2^t)}$, compute the second raw moment estimate v
    - $\theta_t = \theta_{t-1} - \alpha (\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon})$, update the parameters
- End while and return the parameters θ_t

The objective function is denoted as $f(\theta)$, which is a stochastic scalar function that depends on some parameters θ [22]. This function is noisy, meaning its value varies due possibly to sources of randomness, but is differentiable with respect to the parameters θ. The goal is to minimize the expected value of this function, denoted as $E[f(\theta)]$, with respect to the parameters θ. In other words, we want to find the optimal θ that minimizes the average value of $f(\theta)$ over all possible random variations. The stochasticity in this objective function can rise from various sources such as evaluating the function on random subsets (minibatches) of data points. It could also stem from inherent noise within the function itself [22]. The gradient of the objective function at a specific time step t is represented as g_t, which is the vector of the partial derivatives of f_t with

respect to θ. In simpler terms, it tells us the direction and magnitude of the steepest ascent of the function at that particular θ. The algorithm maintains two key moving averages: m_t (the first moment) and v_t (the second moment). These moving averages serve as estimates of the mean and uncentered variance of the gradient. The hyper-parameters β_1 and β_2 control the exponential decay rates of these moving averages, with values in the range [0, 1). A smaller value leads to a longer memory of past gradients. Initially, the moving averages m_t and v_t are initialized as vectors of zeros. This initialization leads to a bias in estimates, particularly early on when the decay rates β_1 and β_2 are small or close to 1. This bias means that the estimates of the first and second moments are skewed towards zero, which can affect the optimizer's performance. To counteract this initialization bias, the algorithm calculates bias-corrected estimates, denoted as m_t and v_t. These bias-corrected estimates provide more accurate and reliable approximations of the first and second moments of the gradient as the optimization process progresses.

Chapter 3 Application to Image Generation

This chapter describes the development of a test plan with which we used to create our model and perform testing on different datasets. It outlines the structure of a flow chart that was fundamental to understanding the primary goals of the model. Moreover, this chapter also provides the pseudo code used in the testing and sampling of our model and the reasoning behind the selection of different functions as well as description of the variables used.

3.1 Logic and Structure of Diffusion Probabilistic Models

The flow chart in Figure 3.1 presents an overview of the structure logic behind the DPM implemented for our experiments. It offers an overview of the interpretation of the functions and variables and how the whole structure interacts with each other. The main variables considered are the dataset, the number of time steps, the β rate, the number of channels, the optimizer used in the U-Net architecture, the batch size and learning rate, and the number of epochs which represent the number of iterations the model will run for. The flow chart also presents the two novel algorithms for DPM, namely, the training and sampling algorithms. These two algorithms are like the ones discussed in the publication by Jonathan Ho et al. reviewed in Chapter 2.2. After applying these two algorithms, the image is then converted from a tensor data type back to either a JPG or PNG format and saved to an output directory.

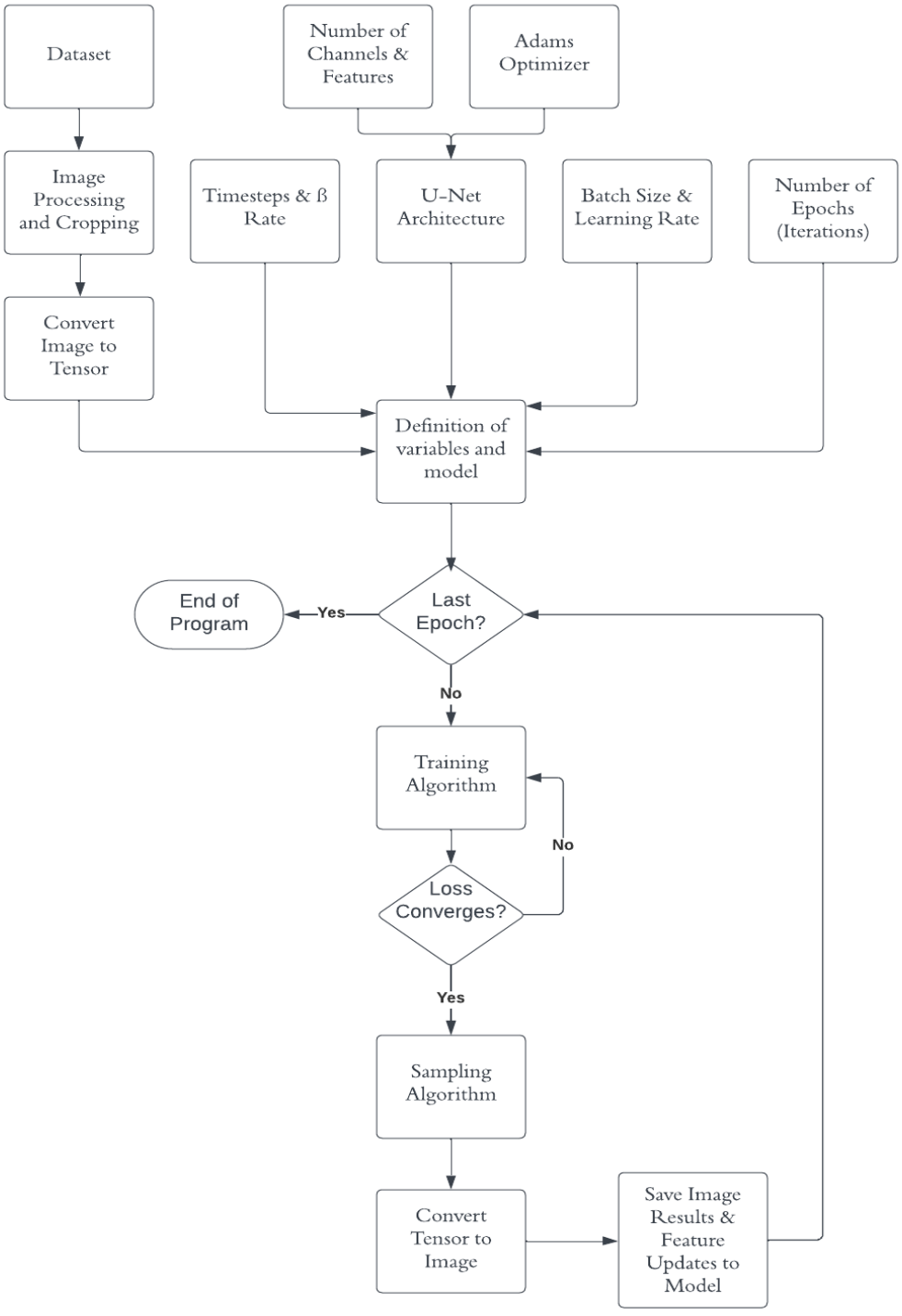


Figure 3.1: Flow chart overview of logic design, functions, and variables in DPM

3.1.1 Dataset

The dataset consists of the input images that will be processed in the model. These images, however, must first be cropped into a square and converted into a tensor. A tensor is a fundamental mathematical concept that has found widespread application in various fields, including mathematics, physics, computer science, and machine learning. It is a multidimensional array or data structure that can store and manipulate data of different types, such as scalars, vectors and matrices [37]. Scalars are rank-0 tensors, vectors are rank-1 tensors, and matrices are rank-2 tensors. Higher-rank tensors can have three or more dimensions. The rank of a tensor refers to the number of dimensions it has, and the shape of a tensor defines the size along each axis or dimension [37]. For a rank-2 tensor, its shape might be (m, n) to indicate m rows and n columns. Tensors can store data of various types, including integers, floating point numbers, and others, making them suitable for use with mathematical computation involving DPM, which requires elementwise operations and matrix multiplications. Thus, in machine learning and deep learning, tensors play a fundamental role. Neural networks, including convolutional and recurrent neural networks, process data in the form of tensors. In CNNs, input data is typically represented as a 3D or 4D tensor (height, width, color channels, batch size) or a higher dimension for more complex tasks. The ability of tensors to efficiently store data is critical in CNNs such as our U-Net architecture, as they are often required to work with large datasets and complex models. Tensors are also integral to convolution and pooling operations in CNNs, which have been discussed in Chapters 2.3.1 and 2.3.2.

Today, leading deep learning frameworks like TensorFlow and PyTorch are designed to work with tensors, making it easy to build, train, and deploy CNN models such as U-Net [47]. Tensors seamlessly integrate with these frameworks, allowing for efficient computation on graphic

processing units (GPUs) and tensor processing units (TPUs), which are specialized hardware accelerators developed by Google to speed up and enhance the performance of machine learning workloads. Additionally, tensors allow for the visualization and interpretation or learned features in CNNs. Intermediate sensor representation in hidden layers can provide insights into a neural network decision-making process. Furthermore, since they support the scaling of CNNs to larger and more complex datasets, tensors can address real-world applications with extensive data which is the case for our DPM.

Once tensors are processed in our DPM, they are converted back to an image format for visualization and for learning how to remove noise added during the forward process (see Chapter 2). The tensor that is converted back to an image format may be a feature map, which is an output of the U-Net layer. The shape of a tensor should match the dimensions of the original image, which in our case was cropped to squares of different sizes. To process a tensor, we need to reverse any scaling and normalization that was applied during the conversion from an image to a tensor. This step involves multiplying or dividing the values to match the original pixel range, which in our case is [0, 255] for 8-bit images. Additionally, since a tensor presents a color image, we need to ensure that the color channels are correctly ordered for RGB images and have the right shape [24]. After each tensor is processed, the resulting image will be saved in an output directory which includes all the updated feature maps and the trained model. This will enable the comparison of the results at different epochs later.

3.1.2 Timesteps and Beta (β) Rate

The next most important hyperparameters in our DPM are the timesteps and the β rate. Timesteps represent the number of discrete time steps needed to carry out the diffusion process discussed in Chapter 2. This hyper parameter defines the temporal dynamics of our DPM as well as the number of iterations through which stochastic noise is added to the tensor during the forward process. Similarly, this hyperparameter also determines the number of iterations through which a purely Gaussian noise is be removed from the tensor during the reverse process. The choice of the number of timesteps can significantly impact the quality of the generated samples. A higher number of timesteps may lead to smoother, more gradual transitions, while a lower number may introduce more fine-grained, rapid changes in the generated data. Selecting an appropriate number of timesteps is a trade-off between computational efficiency and the quality of generated samples. Specifically, larger timesteps typically require more computational resources. This hyperparameter can be tuned based on the specific task and the dataset considered, which in our case are image datasets.

The β rate controls the rate at which noise is added during the diffusion process. It determines how fast or slow the data evolve over time. The choice of the beta rate is critical for the quality of generated image samples. A higher beta rate leads to a faster diffusion, resulting in samples that are more like the initial data images. A lower beta rate makes the diffusion process more gradual and may lead to samples with more diversity which may not be ideal for our DPM. The beta rate often follows an annealing schedule, which means it decreases over time [25]. This allows the model to start with a high noise level and gradually reduce it, leading to samples that become more focused and less noisy as the diffusion process continues. However, in our case, the beta rate will start small and gradually increase in accordance with the timesteps. That way

Gaussian will be gradually added to the tensor data, allowing the model to learn on how to remove noise following the program logic. In some models, the beta rate is referred to as the "temperature" parameter. High temperature (high beta) encourages exploration and randomness, while low temperature (low beta) encourages exploitation and stability. In our model the former is sought. The choice of timesteps and beta rate collectively impacts the sampling strategy. Shorter timesteps and higher beta rates may result in more efficient sampling but may generate samples with less diversity. Meanwhile, longer timesteps and lower beta rates can produce more diverse samples but may require more computational resources. Because our model is primarily intended for image generation, selecting the right number of timesteps and beta rate is crucial for generating realistic and high-quality images. These parameters are vital hyperparameters that need to be tuned through experimentation and cross-validation to achieve the best results. Additionally, properly chosen timesteps and beta rates can contribute to training stability, whereas inadequate values may create convergence issues.

3.1.3 Number of Channels, Features, and Adams Optimizer

In a Diffusion Probabilistic Model (DPM), the number of channels and features plays a crucial role in shaping the model's capacity and its ability to capture complex patterns in the data. Channels and features also determine the expressive power of the model [33]. Each channel typically represents a distinct attribute in the data. In the case of images, channels can represent color information, texture, edges, or specific object parts. In our case, channels are used to encode the color information in an RGB image, so the number of channels is 3 [24].

In general, the number of channels and features should be adjusted to strike a balance between preserving essential information and reducing dimensionality. The complexity of the data at hand often influences what is the optimal number of channels and features. For example, natural language text and high-resolution images may require more features to accurately capture nuances in the data. The number of channels and features can also affect a model's generalization ability. Too few channels or features may result in underfitting, where the model cannot fully capture the complexity of the data. In contrast, an excessive number of channels and features may lead to overfitting, where the model learns noise in the data rather than meaningful patterns [29]. This can be counterproductive in our case since our input images contain a large amount of stochastic noise. Additionally, a higher number of channels and features generally increases the computational cost for training and inference. It may require more memory and computational resources. Therefore, there is often a trade-off between model capacity and computational efficiency.

Depending on the task, the number of channels and features can influence the transferability of the model. A model with a limited number of channels and features may be more transferable across different domains, while a highly specialized model may not generalize well. In some cases, adding redundant channels or features can improve the robustness of the model to noisy or incomplete data. This can help the model make better prediction when dealing with real-world data that is not perfectly clean or consistent. The interpretability of a model may also be influenced by the number of channels and features. A model with a smaller number of channels and features might provide more human-interpretable results, as it focuses on the most salient features of the data. Thus, the number of channels and features is often a hyperparameter that requires tuning during model development. Such tuning often involves experimentation and

validation on specific datasets to find the optimal configuration for a given task, which in our case is to produce high-quality output images that are readily interpretable by the human eye. This hyperparameter also influences the U-Net performance and thus may require further optimization.

To achieve the goal of optimizing this hyperparameter, the Adams optimizer is useful. As discussed in Chapter 2.3.4, the Adams optimizer allows one to monitor the training process and adjust hyperparameters if necessary to achieve the best results for a specific machine learning task [39]. In our case, the Adams optimizer is used to adjust the feature parameters in the U-Net architecture to minimize the loss, leading to higher-quality images [39].

3.1.4 Batch Size, Learning Rate, and Number of Epochs

In a diffusion probabilistic model, the batch size, learning rate, and number of epochs are crucial hyperparameters that can profoundly affect the training and performance of the model. To understand the impact of these hyperparameters, we first look at the role of batch size selection. The batch size is the number of data samples (samples of our dataset) used in each forward and backward pass during a single training iteration. The batch size affects the efficiency, regularization, and memory wage thus presenting some inevitable trade-offs. A smaller batch size allows for more frequent weight updates during training. This can speed up convergence as each update is based on less data. However, this can also increase computational cost. A larger batch size, on the other hand, can be computationally efficient but may converge more slowly. Additionally, a smaller batch size introduces noise in the weight updates, acting as a form of regularization. This can help the model generalize better by preventing it from overfitting the

training data [29]. In contrast, a larger batch size requires more memory for computation. Thus, smaller batch sizes are essential when working with limited GPU memory or large datasets. Large datasets are one of the biggest limitations in our experiments since our databases contain a vast collection of data images. Thus, the choice of batch size often leads to a trade-off between computation time and convergence speed. It is therefore crucial to experiment with different batch sizes to find the right balance for a specific task.

The learning rate is a hyperparameter that controls the step size in the optimization algorithm. It determines how much the model's parameters are updated after each batch of training data. This hyperparameter is important because it affects the convergence speed, the stability, and the ability of our DPM generate. An exceedingly small learning rate tends to yield slow convergence, while an exceedingly large learning rate can result in oscillations and even instability in a training and sampling process. An appropriate learning rate helps avoid overshooting the optimal solution and avoid getting stuck in a local minimum, which is a point in the parameter space in which the objective has a lower value than at all nearby points but not necessarily lower than at all other points in the entire parameter space. The learning rate can also impact the model's ability to generalize to unseen data. A well-tuned learning rate often leads to better generalization. The learning rate is therefore one of the most critical hyperparameters to tune during our model training. Techniques like learning rate schedules, adaptive learning rates, and learning rate warm-up can help fine-tune the learning rate during a training process.

Finally, an epoch is a complete pass through the entire training dataset during training. Upon completing an epoch, one could sample data to visualize the training progress and understand the impact different hyperparameters have at different epochs. The number of epochs determines how many times the model sees the entire dataset. An exceedingly small number of epochs may

result in underfitting, while an exceedingly large number of epochs may lead to overfitting. Finding the right balance is therefore essential, which is another key aspect of our experimentation process. When it comes to convergence, training often yields rapid improvement in performance which then stagnates. The number of epochs should be chosen to ensure the model converges to a good solution without overtraining, which can cause unnecessary power consumption. In practice, monitoring changes in performance validation and using techniques like early stopping (halting training when validation performance starts to degrade) help find the optimal number of epochs. Thus, an appropriate number of epochs can help optimize the trade-off between computational cost and model performance.

3.2 Pseudo-Code

The pseudo-code used during our experimentation was inspired by the first and second algorithms from the paper by Jonathan Ho et al. reviewed in Chapter 2. The pseudo-code also used Unet2DModel, which is the U-Net architecture first introduced by Ronneberger et al. mentioned in Chapter 2.3. Additionally, the pseudo-code utilized the Adams optimizer for optimizing our model and updated the features in each epoch. The hyperparameters were also varied throughout our experimentation to study their impact and the trade-offs involved. The pseudo-code used is listed below (see Appendix E for the actual code used during experimentation):

- Define the input data:
    - Define T as the number of timesteps, which is an input integer type, that is, $T \in \mathbb{Z}$

  - Define the beta (β) schedule as a 1D tensor, which in our case increases from 0.001 to 0.02 over several T timesteps, where each beta is a floating type. Thus, beta is a 1D tensor of size T: beta = $\underbrace{[0.001, 0.002, \dots, 0.2]}_{T-timesteps}$
  - Define the number of channels, which is 3 for our case since RGB images are considered.
  - Define the number of feature channels, which is the number of output channels for each U-Net block. This could range from 64 channels to 256 channels.
  - Define the batch size, which is the number of images processed per training iteration. The batch size is set to 16.
  - Define the number of epochs and learning rate, which in our case is 200 and 3×10^{-3}, respectively.
  - Define which dataset to use as X, the size of which would depend on the number of images in each dataset which is 1000 for most of the datasets used.

- Process input variables before beginning with the epoch iterations and training.
  - Create an alpha (α) schedule for easier computation where $\alpha = \beta - 1$, so that α is also a 1D tensor of size T.
  - Multiply all alphas together to produce $\bar{\alpha} = \prod_t^T \alpha_t$
  - Create a U-Net model with the number of channels and number of features, as well as the image resolution or image size, as input parameters.
  - Find an optimizer using the Adams algorithm for the selected the model parameters and learning rate. This optimizer will be used to perform gradient descent and backpropagation to update the feature parameters in our model.

- Begin the epoch iterations and apply both training and sampling algorithms for each iteration. Sampling can be done every certain number epoch iteration if desired.
- For each epoch do the following:
    - For each image in the dataset, I ∈ $\{1, \ldots, size\ of\ dataset\}$:
        - Define X_0^I to be an original image in the dataset, that is, an image without noise added.
        - Pick a random timestep t^I
        - Define an image X_t^I and add noise by $X_{t^I}^I = \sqrt{\bar{\alpha}_{t^I}} \cdot X_0^I + \sqrt{1 - \bar{\alpha}_{t^I}}$, where $\varepsilon^I \sim N(0, I)$. This expression is derived from equation 53 previously analyzed in Chapter 2.2.
        - Send the noised image X_t^I to the model to learn a predicted noise ε_θ^I
        - Define the loss from predicted noise ε_θ^I and the actual real noise ε^I, that is, $||\varepsilon^I - \varepsilon_\theta^I(\sqrt{\bar{\alpha}_t} \cdot X_0^I + \sqrt{1 - \bar{\alpha}_t}\varepsilon, t)||^2$.
        - Perform gradient descent and backpropagation using the optimizer and then update the features to reduce the loss.
    - After learning the predicted noise ε_θ^I and optimizing the parameters in the model, sample a new image $\tilde{X}_0$ from the noised image $\tilde{X}_T \sim N(0, I)$ by implementing the following algorithm:
        - For t = T, T-1, …, 1
            - $\tilde{X}_{t-1} = \tilde{X}_t - 1\ \frac{1-\alpha_t}{\sqrt{1-\alpha_t}} \cdot \varepsilon_\theta^I\ (\tilde{X}_t, t)$
        - Return $\tilde{X}_0$, which is the newly generated image.

Chapter 4 Hardware Specifications

This chapter describes the hardware used during the training and sampling of our DPM experimentation, as well as the importance and impact of each component for producing high quality results. The modification or upgrade of these hardware resources can potentially increase the performance of the model in both speed and memory management.

4.1 Central Processing Unit (CPU)

The CPU is the central processing unit in a computer that manages most of the general computing tasks [18]. In deep learning, which is what DPM uses, CPUs are involved in data processing, managing I/O operations, and orchestrating high-level model operations. While CPUs are not the primary workhorses for deep learning tasks, they still play a key role in the overall performance. Faster CPUs can help with data loading, preprocessing, and parallelization of certain tasks, leading to faster training pipelines. In our case for our experimentation, three different environments were adopted: a personal computer and two cloud-based Jupyter notebook environments called Google Collaboratory (Google Collab for short) and Kaggle. The reason for using three different environments was to optimize time and study the impact of different hardware designs on the results. Our first environment used an Intel® Core™ i5-10300H CPU @ 2.50GHz with 4 CPU cores, which is installed on a personal ASUS TUF Gaming Computer. For Google Collab, it used an Intel® Xeon® 2.30GHz with 2 CPU cores. Finally, Kaggle Kernel used an AMD EPYC 7B12 with 4 CPU cores.

Environment	Google Collab	Kaggle Kernel	ASUS Personal Computer
CPU	Intel Xeon R 2.30GHz	AMD EPUC 7B12 2.30GHz	Intel i5-10300H 2.50GHz
Clock	2400-5300MHz	2400-5300 MHz	2500-4500 MHz
L1 Cache	512 KB	512 KB	256 KB
L2 Cache	2 MB	2MB	2 MB
L3 Cache	16 MB	128 MB	8 MB
Core/Threads	8/16	24/48	4/8
TDP	45 Watt	155 Watt	45 Watt
Technology	14 nm	7 nm	14 nm

Table 4.1: CPU comparison among the three environments used.

4.2 Graphics Processing Unit (GPU)

GPUs are specialized hardware components designed for parallel computation [42]. They excel at performing matrix operations and are a crucial component for accelerating deep learning tasks, including training diffusion probabilistic models. The presence of a GPU can significantly speed up training times, as GPUs are capable of processing large batches of data and performing the numerous matrix multiplications required for a neural network training. More powerful GPUs with a higher number of CUDA cores and VRAM (video memory) can handle larger models and datasets. This is important with our DPM since we have to deal with datasets of more than 1000

input images. Having larger batch sizes during training has a positive impact in our time utilization and overall performance.

The three different environments we used not only have different CPU specifications, but they also have different GPUs that can significantly affect our results and the overall training performance. GPUs are therefore one of the most fundamental hardware components for DPM.

Environment	Google Collab	Google Collab (Free Version)	Kaggle Kernel	ASUS Personal Computer
GPU	NVIDIA GA100	NVIDIA T4 GPU	NVIDIA GP100	NVIDIA GeForce GTX 1650
Memory Size	40GB	16 GB	16GB	4GB
Memory Clock	1215 MHz	1590 MHz	715 MHz	1500 MHz
Memory Bus	5120 Bit	256 Bit	4096 Bit	128 Bit
Memory Bandwidth	1555 GB/s	320 GB/s	732.2 GB/s	128 GB/s
CUDA	16 MB	7.5	6.0	7.5
Process Size	7 nm	12 nm	16nm	12 nm
(TDP)	250W	70 Watt	250W	75 Watt

Table 4.2: GPU comparison among the three environments used.

4.3 Random Access Memory

The system memory, also known as Random Access Memory (RAM), is used to store data, model parameters, and intermediate results during training. More memory allows for larger models and datasets to be processed by GPUs. In deep learning, larger models and datasets can quickly deplete memory. Thus, it is important to make sure we have sufficient RAM in our environment during training. Insufficient RAM can lead to slower training times or even failure to load and process large models. Sufficient memory ensures smoother training and more efficient data handling. In our first environment where a personal ASUS TUF Gaming personal computer was used, our RAM was 8.0 GB. In our second implementation using Google Collab, our RAM was 40 GB. For our third environment using the Kaggle Kernel, our RAM was16 GB. As it turned out, Google Collab was able to handle our DPM implementation much faster than the other two approaches. However, the biggest drawback was there was a monetary cost for using Goggle Collab. The version of it was much slower and was comparable to the Kaggle Kernel. This difference is important when considering the cost and efficiency of our model.

Chapter 5 Generating New Landscape Images

5.1 First Results on Landscape Images Dataset

For all our results, we used the model implementation outlined in Chapter 3. We also used the first approach for our hardware, which was the ASUS Personal Gaming Computer. For our dataset, we used the first 2,000 images from Landscape Pictures in Kaggle, submitted by Arnaud Rougetet [7] (see Appendix A). We set the number epochs to 500, the number of timesteps to 1000 to match the values selected by Jonathan et al. Our learning rate was set at $3x10^{-4}$. We also set the learning and training batch size to 4. This was due to a limitation on the GPU memory as our model tried to allocate 4 GiB, only 1.72 GiB was reserved in total by PyTorch, and 1.66 GiB was already allocated with only 1.49 GiB remaining. Thus, there was a need to reduce the batch size for our model to run. We also set the number of features parameters to 64 in our U-Net architecture and the image crop size to 64x64.



Figure 5.1: Epoch 10 from landscape images dataset on ASUS computer

Figure 5.1 – 5.5 shows our first results on generating landscape images. Observe that after 10 epoch iterations, our model still produces images with noise, as seen in Figure 5.1. The loss for this iteration is found to be 0.56 which is quite high, meaning our model still needs more iterations to better predict the added Gaussian noise during the sampling process. We also see some color biases in the four-image output grid, where the image on the top left corner shows a purple color bias, while the other three output images showed a green-blue color bias.

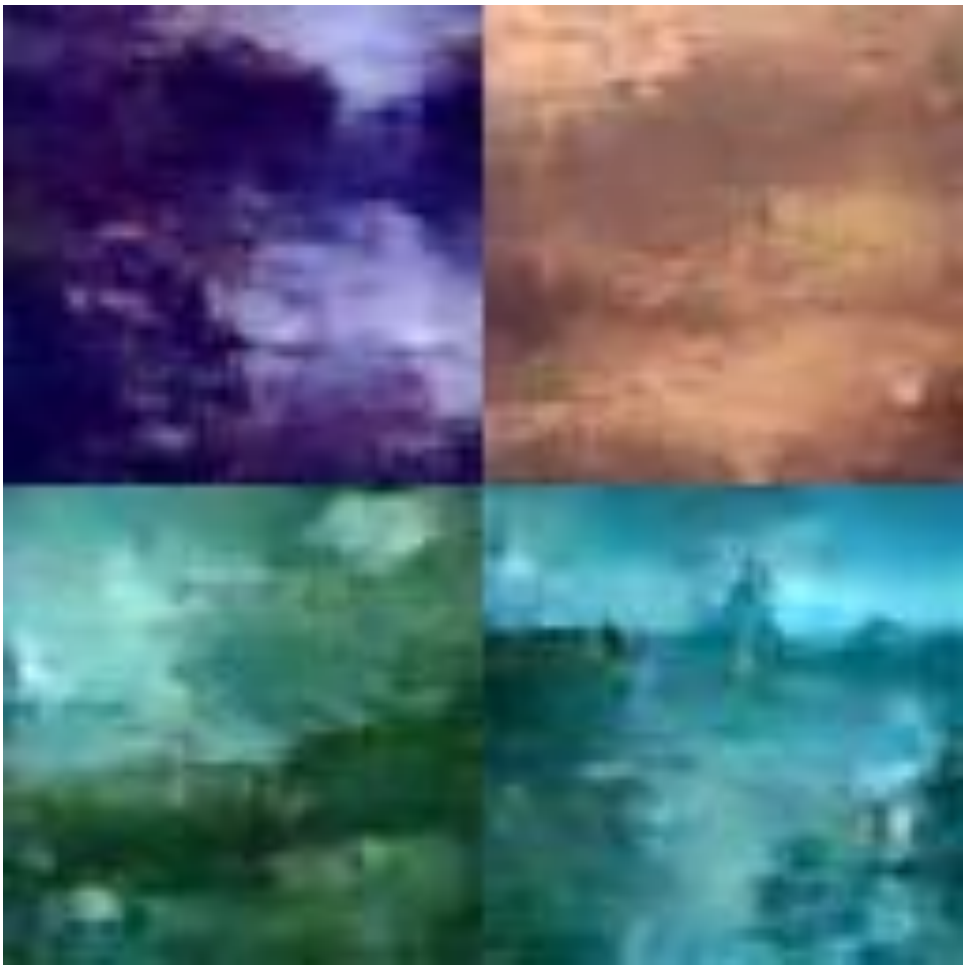


Figure 5.2: Epoch 50 from landscape images dataset on ASUS computer

Figure 5.2 shows the images after 50 epochs. Our model shows improvements in predicting noise, although the images are still abstract. The loss obtained during this iteration decreases to around 0.27, but the color biases persist in the four-image grid output. Indeed, the top left image still shows a purple color bias while the remaining three output images now have better diversity, especially the top right image with a brown color bias. Nonetheless, the bottom two images remained biased with green and blue.



Figure 5.3: Epoch 100 from landscape images dataset on ASUS computer

Figure 5.3 shows the images after 100 iterations. The DPM showed no significant improvement, and the loss remains stagnated at around 0.14. The color biases from our images remain consistent with our previous sampled result with the top right image output showing a red bias.



Figure 5.4: Epoch 200 from landscape images dataset on ASUS computer

Figure 5.4 shows the images after 200 epochs. Our model shows some improvements in having a better predicted noise that reduces the loss to around 0.08, which is significantly lower than our previous output images. The two upper output images remain color biased, with the top left image having purple bias, while the top right image having red bias. However, the bottom two images show only a slight monochrome biased by having a more distinct and less abstract result than the two top images.



Figure 5.5: Epoch 500 from landscape images dataset on ASUS computer

Figure 5.5 shows the images after 500 epochs, where there is no noticeable improvement compared to our precious sample from 200 epochs. The loss stagnated at around 0.07. There are almost no distinct differences compared between the two results with the two upper images displaying a monochrome biased towards purple and red, meanwhile the bottom two images look almost identical to the results after 200 epochs. The total run time from our DPM implementation was 7.25 hours, from which, more than half showed almost no improvement. Thus, our implementation resulted in an inefficient approach from cutting the number of epochs to 200 with negligible loss in image quality.

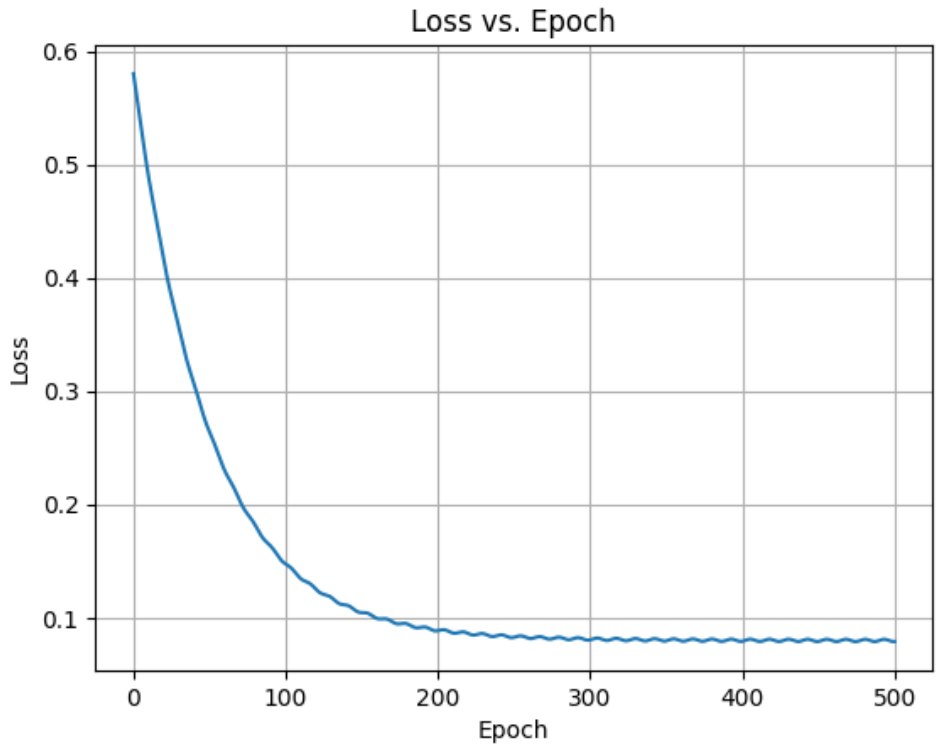


Figure 5.6: Loss vs epoch from landscape images dataset on ASUS computer

Observe from Figure 5.6 that the loss decreases as the number of epochs increases. In our analysis we see that the loss compared to the number of epoch iterations converges to around 0.6-0.8 after around 200 epochs. These observations suggest that major improvements in predicting noise in our U-Net architecture and DPM occur during the first 100 epoch iterations. Motivated by this information, our next study considers only images after 250 and 300 epoch iterations to reduce power consumption and the time needed.

5.2 Second Results on Landscape Images Dataset

Our second results are obtained under the following conditions. First, we reduced the number of epoch iterations as previously mentioned. Second, we increased the number of feature parameters in our U-Net architecture to 128 so that we could analyze the impact they have on power consumption and elapsed time in training and sampling of our DPM. All other hyperparameters remained constant in comparison to our previous approach such as the number of timesteps at 1000. Note that the image crop size must be increased to 128x128 to match our U-Net architecture. The learning rate also remained constant at around $1x10^{-4}$. The training and sampling batch size also remained constant at 4, the decision was made due to the ASUS personal computer approach with limited GPU memory at around 4GB. Thus, our second results focus on the impact of the feature hyperparameter in our U-Net model.



Figure 5.7: Epoch 10 from landscape images dataset on ASUS computer (128x128)

Figures 5.7-5.9 show our second results on generating landscape images. Observe that by increasing the number of timesteps, the model shows a greater improvement after 10 epoch iterations compared to our first results. The loss of 0.25 is also smaller. With such information in mind, we can conclude that the number of feature parameters in our model has a significant impact on the early loss, and thus on the prediction of noise learned in our model during the training process. The four-image grid output shows a very low color bias and many similar features among all four of them. However, the bottom right image shows a red bias unlike the other three. Another important observation is that the time needed increases from about 5.2 hours

to 6.1 hours, thus, by increasing the size of our images and the feature parameters, we also increased our power consumption and our time elapsed.



Figure 5.8: Epoch 100 from landscape imges dataset on ASUS computer (128x128)

After 90 more iterations, at 100 epochs, our model showed more improvements in noise prediction and reduction, with two images having less color biased, a bit less abstract representation. However, the upper left output image and the bottom right image in our output grid still showed significantly color biased, with blue and pink monochromatic biased respectively. This despite our loss reducing to 0.07, which is a bigger improvement when compared to our previous approach with the same number of iterations. Nonetheless, the time

elapsed also increased from the previous result from 1.25 hours to 2.5 hours for the same number of epochs produced, thus, it still showed some areas for improvement based on the color biased. Another important aspect from this new approach is the higher quality in the image this is mostly due to the increase in the image size from 64x64 to 128x128 resolution, which had a positive impact in our results.



Figure 5.9 - Epoch 300 from landscape images dataset on ASUS computer (128x128)

Figure 5.9 shows the images after 300 epoch iterations, observe that the four-image output grid does not show any noticeable improvement in color biased nor loss predicted and removed which stagnates at around 0.05, however, it does not show improvement in edge smoothing as the

quality of the images show a less sharp definition. Nonetheless, the time elapsed and power consumption was greater than our previous attempt when considering the same number of epoch iterations, making this new approach more expensive and without a significant improvement in color biased.

From our analysis on the graph 5.2 shown below, we can conclude that our loss converged faster than our previous attempt, thus supporting our decision to reduce the number of epochs from 500 to 300. Another important notice is that our loss convergence also got reduced from around 0.08 to a much lesser value at around 0.05. This is noticeable in our results when comparing the two last output image grids from both approaches. Nonetheless, we saw some room to improvement regarding the color biased in our outputs, while we saw some improvement in some of the images, we still did not saw some monochromatic biased in two of our four output images.

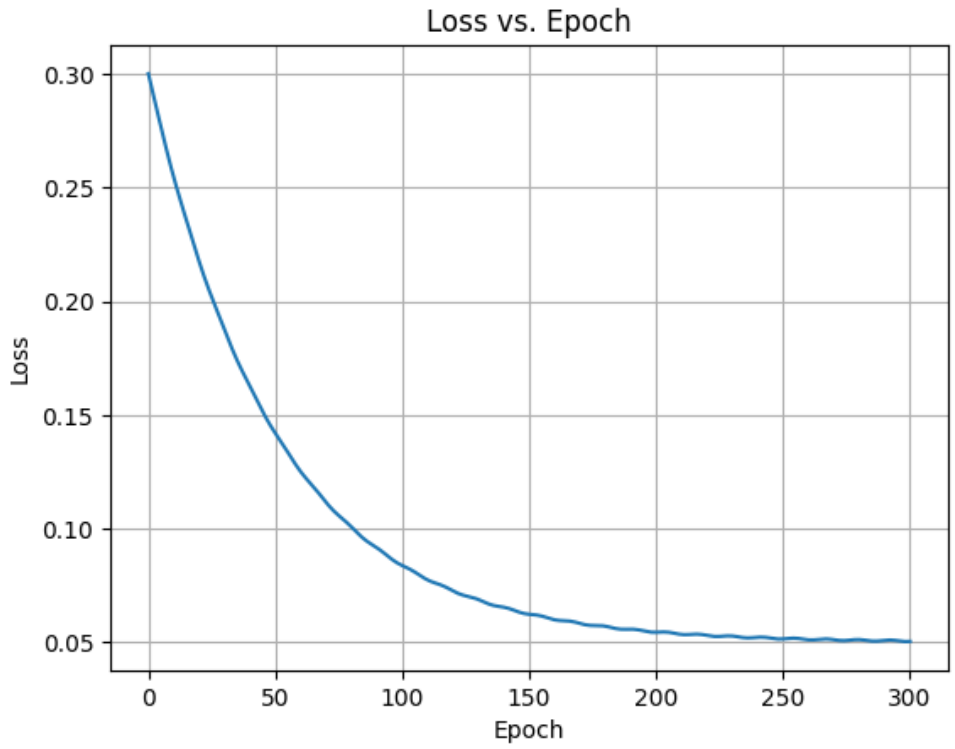


Figure 5.10: Loss vs epoch from landscape images dataset on ASUS computer (128x128)

5.3 Results on Landscape Dataset on Kaggle Kernel

For our third approach of results, we decided to increase the batch size to 16 in the learning and sampling process, this way we will have a larger output image grid of 16 pictures, thus increase our analysis from just having 4 sampled images per epoch iteration. We also increased the number of feature parameters in our U-Net architecture and DPM from 128 to 256. However, these changes impacted our GPU memory significantly, thus our ASUS TUF Gaming personal computer could not process such resolutions and batch sizes, which was trying to allocate 12.1 GiB from the limited 4 GiB in the NVIDIA GeForce GTX 1650. Thus, this new approach required a higher memory which the Kaggle Kernel provided with the use of the NVIDIA GP100, which could potentially allocate around 16 GB of GPU memory, more than the enough required for this new approach. However, to not exceed such memory limitations, we reduced the size images to 64, in the same way from our first two approaches. In essence, this new approach still focuses on the big impact from the feature parameters since they show a higher improvement when increased in our previous result outputs. The learning rate remain constant at $1x10^{-4}$ and the timesteps also remained unchanged at 1000 timesteps. Another important difference between the three previous approaches was the increase of the landscape dataset, from 2000 input images to 4000 input images, this was due to the way the Kaggle Kernel approach access the dataset. This dataset is easily accessible through the Kernel environment without the need to download it into the SSD memory, which was the case for the first approach using the ASUS computer. This dataset is thus stored in the Kernel environment, which is a more practical solution and thus saves local disk space. However, this approach is more susceptible to internet speed connectivity, which in our case follows a Wi-Fi 4(802.11n) protocol, and a network band

of 2.4 GHz. Nonetheless, this does not drastically affect our performance as it is analyzed in our results produced in each epoch iteration.



Figure 5.11: Epoch 10 from landscape images dataset on Kaggle Kernel

Figure 5.11 shows results from the first 10 epoch results using the Kaggle Kernel approach, we can appreciate a faster loss reduction which equates to around 0.09, which is the fastest when compared to the previous results produced. This is an indication of the high impact of the number of features in our U-Net architecture. The image results do not possess a substantial monochromatic bias in any of the output pictures, thus fixing one of our primary issues when training our DPM.



Figure 5.12: Epoch 100 from landscape images dataset on Kaggle Kernel

Figure 5.12 shows results after 90 more epoch iterations, the loss obtained from the noise learned in our DPM training process gets reduced significantly more to around 0.04. This is a major improvement from all previous 100 epoch iterations used in any previous approach. Thus, the increment of the feature parameters fed into the U-Net is vital to obtain desirable high-resolution output images. The 16-image output grid does not present any major monochromatic bias besides a high prevalence of blue, and darker tones present in almost all the output images. Nonetheless, the network has learned to predict the sky from ground and water from earth, thus showing a less abstract and more objective view of a landscape. From this result, the output data can also help

us analyze the data input images from our database by visualizing that almost all the pictures have a tendency of generating the sky, which suggests the dataset has many pictures with a sky above.

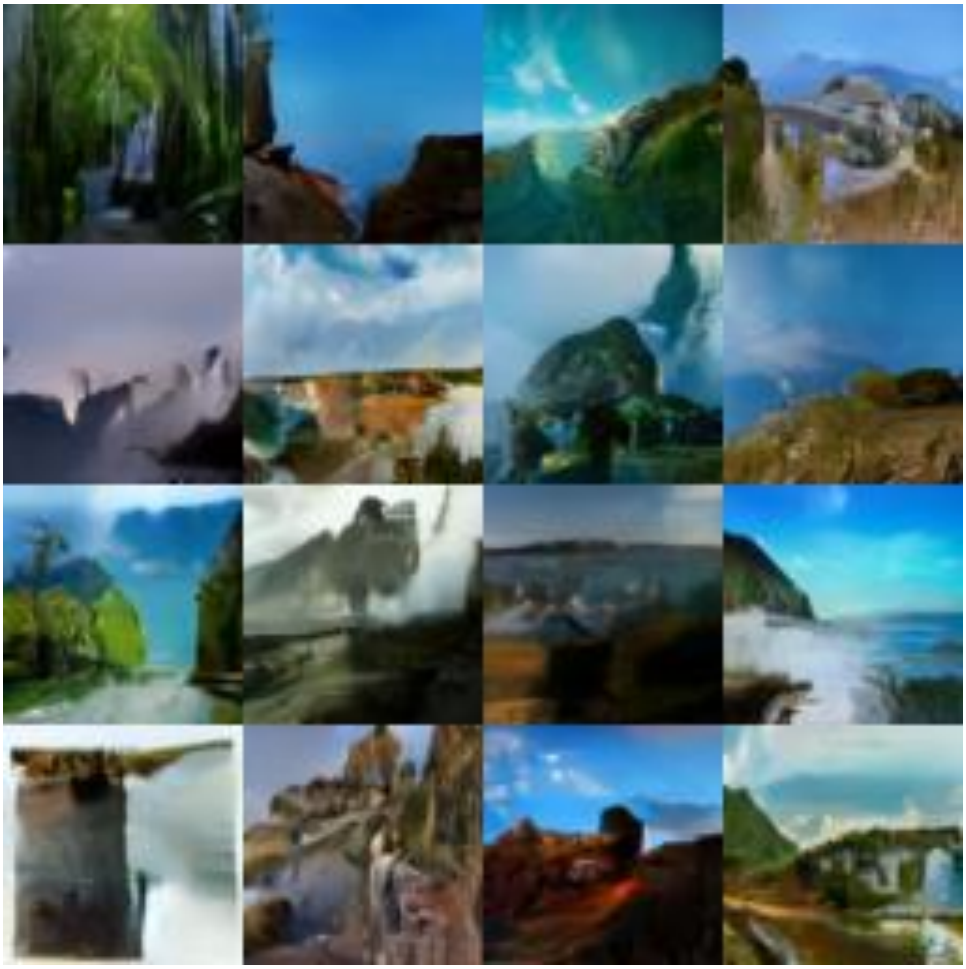


Figure 5.13: Epoch 300 from landscape images dataset on Kaggle Kernel

Figure 5.13 shows how once the program finished its execution at 300 epochs, the loss between the predicted noise and real noise in the sample images was reduced to the lowest value thus far at around 0.0097. With our analysis we observed the importance of selecting enough feature parameters, which the U-Net will learn to reduce such noise significantly in each epoch. The power consumption and time elapsed to run this model approach was also reduced to only 5.6

hours of runtime, this is due to the use of a bigger GPU memory of 16 GB provided by the Kaggle Kernel approach.

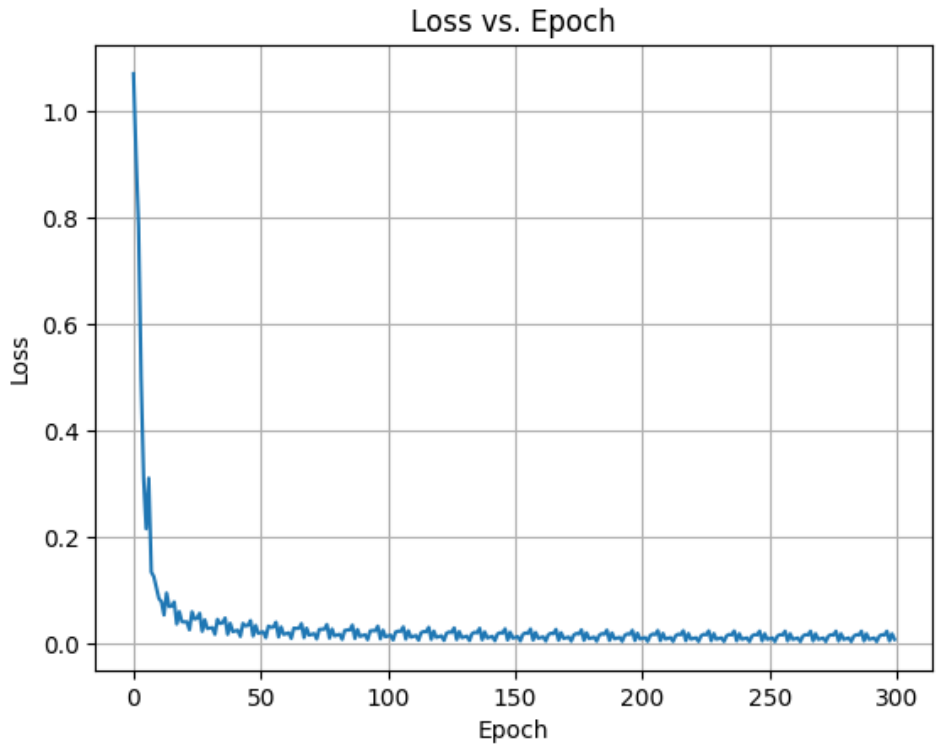


Figure 5.14: Loss vs epoch from landscape images dataset on Kaggle Kernel

In our data analysis from this new approach, we can observe that the loss converges faster to a value closer to zero at about 200 epochs, this means we may even be able to reduce our iterations even farther to save energy consumption and time while training and sampling our data. We also observe a few fluctuations of error of about 0.01 peak to peak, which means that some epoch iterations have less noise or greater noise independently of the number of epochs elapsed, in other words, our data does not gradually decrease after it has converted to a range of values with an interval of 0.01. Thus, encouraging the reduction of epoch iterations to improve our DPM even more.

Chapter 6 Generating Complex Images

In the next round of testing and evaluation on our DPM, we decided to run our model in more complex datasets. The previous datasets presented a large-scale of features that may be easily interpretable in a macro-analysis of the images. Nonetheless, we wanted to test the model in smaller-scale features to determine its performance in more complex images. For such a task, we selected a database which consisted of city images, vehicles and human faces. Adjacently, we ran the dataset on the Kaggle environment with human faces and the car dataset at the same time. By running two models simultaneously, we optimize our time for analysis and comparison in real time for both approaches.

6.1 Results on City Images Data Set on Kaggle Kernel

To reinforce our model performance results and assure our previous results are not extraneous sample images that may only produce this high-quality output images on a landscape dataset, we selected a new dataset from Kaggle of 749 city view images, from which 522 images were taken during the daytime, and another 227 images were taken during the nigh time [9] (see appendix B). The resolution of most images from this dataset is 1024x1024, but for simplicity and optimization of time, we cropped and reduced such images to a resolution of 128x128, this will save time and space when training and sampling our DPM. Similarly, to our previous approach, we used the Kaggle Kernel environment with a 16-batch size for training and sampling our

images, which was possible due to the high GPU memory of 16 GB provided by the NVIDIA GP100. We kept the feature parameters selection to 256 to match our previous attempt. We also reduced the number of epoch iterations to 250, this decision was made after observing the previous loss convergence in the landscape dataset. The only major difference between this new attempt was the slight change of the learning rate from $3x10^{-3}$ to $1x10^{-3}$. Many of the remaining hyperparameters remained constant such as the number of channels used in our U-Net, which was 3 due to our dataset having RGB colored input images. Since this new dataset is significantly smaller than the previous landscapes and the GPU memory being considerably bigger, we expected a faster completion time than any previous attempts.



Figure 6.1: Epoch 10 from city-view dataset on Kaggle Kernel

Figure 6.1 shows the first set of output results from the 16-image grid produced by our DPM after 10 epoch iterations. These results show some sort of abstract interpretations to what a city

may look. The prevalence of blue is observed in almost all output images, and there are also dark tonalities in some of the results as well. In concerns with the loss, this was substantially low from the beginning at around 0.092, which is very similar to the value we observed in the landscape dataset by using the same overall hyperparameters. This reinforces the impact that the feature parameters have in our DPM when learning the predicted noise which then gets subtracted from the noised input image. Additionally, the use of the high-performance NVIDIA GP100 provided by our Kernel environment farther improves our training and sampling time, since to sample this first 10 epoch output, our implementation took around 0.18 hours. One of our fastest sampling times for the first 10 epoch iterations thus far.



Figure 6.2: Epoch 100 from city-view dataset on Kaggle Kernel

Figure 6.2 shows results after 100 epoch iterations, the 16-image output presents less abstracts representations of a city view and more defined structures of urban appearance. Additionally, the

output images do not present a monochromatic bias towards a particular color unlike the first two approaches shown in the landscape dataset. We also observe the shift from the high influence of cool colors such as blue to warmer colors such as red and brown in most of the output images. Nonetheless, many output images do not show city-like features, such is the case of the bottom left output image in our grid, which still presents a more abstract representation of that of a city.

In terms of the predicted noise from our network, we observe a loss reduction to 0.03, which is more than half our first 10 epochs, and even smaller than that presented by the same number of epochs in the previous landscape dataset at around 0.04. Thus, the model still shows positive improvements regarding performance.



Figure 6.3: Epoch 250 from city-view dataset on Kaggle Kernel

Figure 6.3 shows the results after 250 epochs where the completion of the model execution is much more promising. The images present more city-like views of skylines, as well as samples of day and night views, which are much more easily interpretable than our previous epochs, thus reassuring our decision of increasing the feature parameters in our U-Net. The loss at the end of the execution was 0.0098, which was very similar to our previous result in the landscape dataset at 0.0097. Meaning, our implementation works well with other datasets with more complex structures.

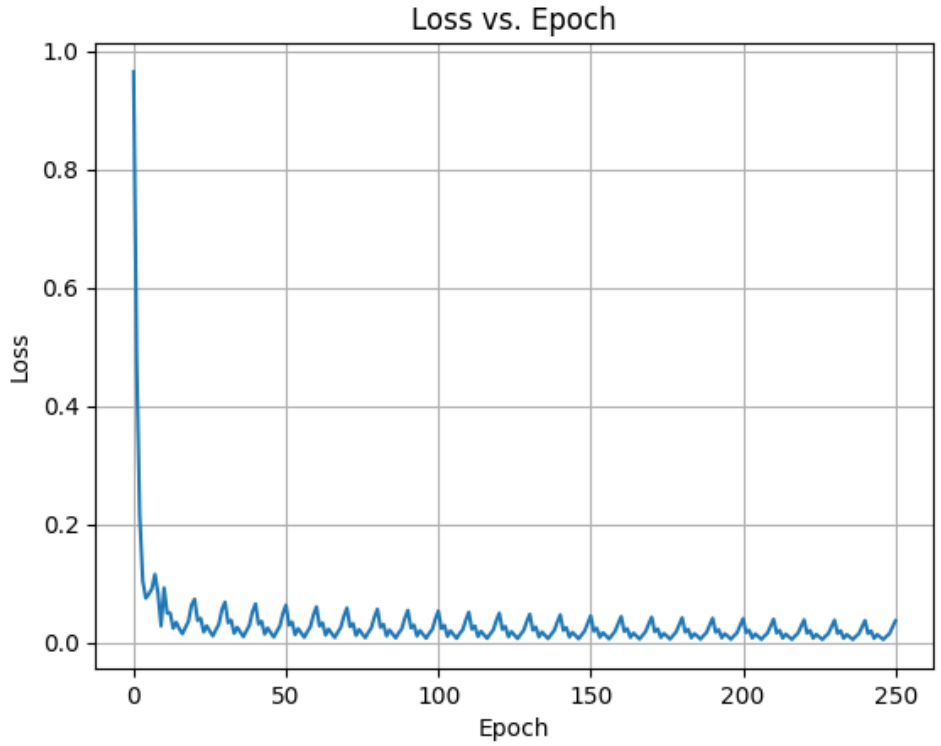


Figure 6.4: Loss vs epoch from city-view dataset on Kaggle Kernel

6.2 Results on Car Images Dataset on Google Collab

The dataset for the vehicle images was downloaded from the Kaggle dataset called *Multilabel car and color* dataset [8] (see appendix C). From which, we downloaded 2,000 images and store them in a Google Drive, which is cloud storage service provided by Google. The reason to store it in such drive was because in this new dataset we change our environment from the Kaggle Kernel to the Google Collaboratory, which is like our previous environment, but it utilizes a NVIDIA GA100 GPU, it also has 40GB of memory, which is substantially larger than any other GPUs we have used in our model thus far. For our first evaluation in the car dataset using the Google Collab environment, we kept most of the hyperparameters used in the city-view and landscape datasets, but we made a few adjustments such as increasing the image size to 128x128, and the feature hyperparameters to 512. Based on our previous results, this selection should yield a high-resolution image grid output. The timesteps for this dataset were also increased to 1500. The learning rate was kept at $1x10^{-3}$, and the batch size was also kept at 16, thus this should sufficient data samples to observe the impact of such hyperparameters in a more complex dataset than the two previous ones. Therefore, in this new dataset our model should learn more complex structures such as a tire, windshield, doors, and the overall aspect of what a vehicle may appear.



Figure 6.5: Epoch 10 from car images dataset on Goggle Collab

Figure 6.5 shows the first 10 epoch iterations, we can conclude that the model did not learn the basic structures of what a car constitutes. All images present color-biased, even though it is not as monochromatic as it appeared in the first approach on the landscape database, our model showed tendencies of prevailing colors in some of the output images such as the third upper picture from left to right, which showed high monochromatic bias towards green. The loss presented at 10 epoch iterations was relatively favorable at 0.024, which was even lower than the city-view dataset at the same epoch iterations. Our model also showed the way to learn the basic structure of a car, with some pictures showing what may look like a tire and the windows of a

vehicle. Additionally, the time elapsed for these first iterations in the training of our model was 0.13 hours, which is a desirable time of execution compared to the previous attempts thus far, especially considering the timesteps were increased by 1500.



Figure 6.6: Epoch 100 from car images dataset on Goggle Collab

Figure 6.6 shows the results after 100 epoch iterations showed surprising results, our model was able to predict the noise with much more accuracy with a loss of only 0.0094. We observe very complex structures of a car being displayed in our 16-image grid output such as wheels, windshields, doors and even well-defined backgrounds. This is more notorious on the two upper

left pictures, or the first two pictures of the first row, and the three most-left pictures, or the three first pictures in the first column. Nonetheless, some level of color biased is still observed.



Figure 6.7: Epoch 200 from car images dataset on Goggle Collab

Figure 6.7 shows the completion after 200 epochs. The model does not show any major improvement of loss, which stagnated at around 0.0087, thus inferring that we may be able to reduce the epoch iterations even further without affecting most of our output image data. The output grid resulted at the end of completion, showcase many similarities with the images produced at 100 epoch iterations, nonetheless, we can observe that some of the vehicles presented have changed orientation. This is the case for the red car located on the third output

picture of the first column, which originally was facing from the front at an angle, but on this new output grid, it faces on the side. This shows us how our model can predict not only colors, and complex structures but also angles as well.

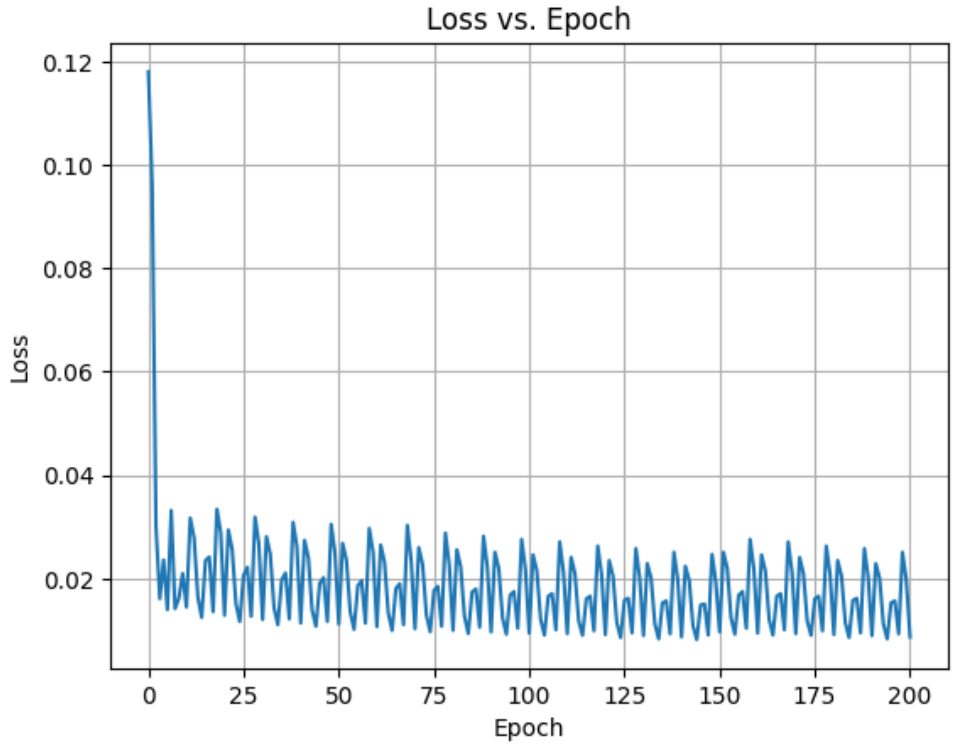


Figure 6.8: Loss vs epoch from car images dataset on Google Collab

From our analysis on the loss vs epochs, shown in Figure 6.1, we can observe how the initial loss was significantly reduced when compared to the previous graphs from all the other datasets, this concludes that adding more timesteps and increasing the number of feature parameters in our model highly-impact the predicted noise early on, thus eliminating the need for extra epoch iterations. Which in turn saves us power consumption and time. This is also backed by the fact that our model runtime was only 4.7 hours total.

6.3 Results on Human Faces Dataset on Kaggle Kernel

As previously mentioned in chapter 6.1, we ran both datasets for the car images and human faces simultaneously, this was done with the intention of optimize time and compare in real time both approaches. The dataset used for this approach was the *Human Faces* dataset from Kaggle [6] (see appendix D). The human dataset was run by using the Kaggle environment, which is the same approach used for the landscape and city-view. The biggest difference between both environments is that the Google Collab uses NVIDIA GA100 with 40 GB of memory, meanwhile the Kaggle environment uses NVIDIA GP100 with 16 GB of memory. Nonetheless, we restricted the training and sampling batch size to 16, thus the difference in GPU memory should not be of much concern. Like the previous approach with the car dataset, we kept most of the same hyperparameters such as setting the learning rate to $1x10^{-3}$, and the image resolution to 128x128, as well as the number of feature parameters to 512. However, the biggest difference was the number of timesteps which was kept at 1500, this to showcase if this hyperparameter will have a high impact on the results of the complexity or time of execution as compared to the car database. Additionally, the number of epoch iterations was reduced to 250, this was chosen based on the previous approaches, which showed that the loss began to converge and stagnated at around 150-180 epoch iterations. Thus, this decision reduced our time of execution without affecting our analysis on the performance of our DPM.



Figure 6.9: Epoch 10 from human faces dataset on Kaggle Kernel

Figure 6.9 shows the first 10 epochs. We can see the model is trying to learn the patterns from the face of a human which are more complex than all our previous attempts. The results also show monochromatic bias in every picture, which resembles the results from our very first attempts. The output grid also showcases many features from a human face such as eyes, mouth and nose in a very abstract representation. The loss from this first run of our model was about 0.0269 which is very similar to our approach with the car dataset, meaning we expect similar results after the program has finished the entire 250 iterations.



Figure 6.10: Epoch 100 from human faces dataset on Kaggle.

Figure 6.10 shows the results after running 100 epoch iterations show a significant improvement, like the previous results on the car dataset. The images showcase smoother edges and less abstract representation of human faces such a more apparent human eye, mouth or nose, which is the case for the upper right image on the corner of the 16-image output grid. The loss from this iteration was also significantly lower at around 0.0046. Moreover, this shows how our model can learn more abstract features from our input dataset images. Nonetheless, some images still showed signs of monochromatic bias and more abstract representations of a human face such as the two bottom right images, which are not as defined as the rest of them.

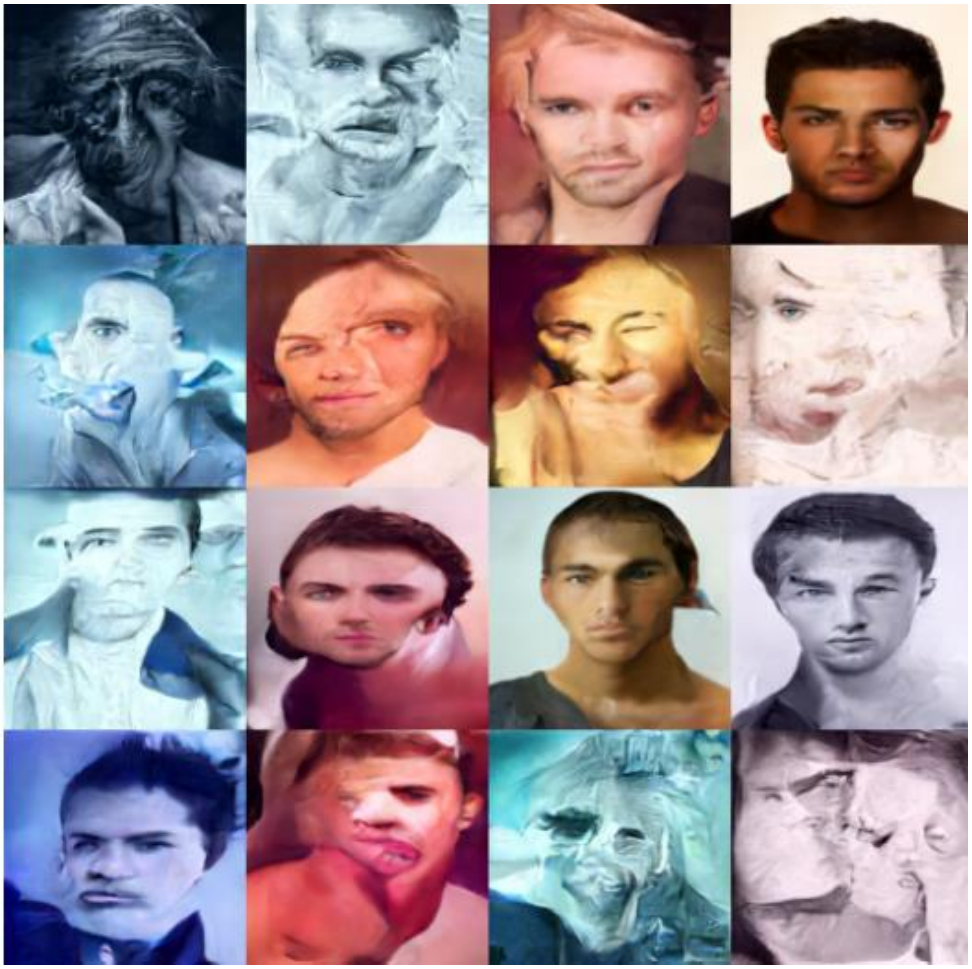


Figure 6.11: Epoch 250 from human faces dataset on Kaggle.

Figure 6.11 shows the results from our final output grid after running all the 250 iterations demonstrate promising improvements in our model. The best output image in our grid is in the right upper corner, thus demonstrating the potential for our model to learn and predict more accurately the noise. The other output images also show potential, with more defined eyes, mouth, noise and hair features that are less abstract. The overall loss at the end of execution oscillates at around 0.0038 Nonetheless, some output images still showcase monochromatic bias such is still the case for the two bottom right output pictures. Furthermore, the upper left corner image also presents more distortion and abstract features than the previous first 100 iterations.

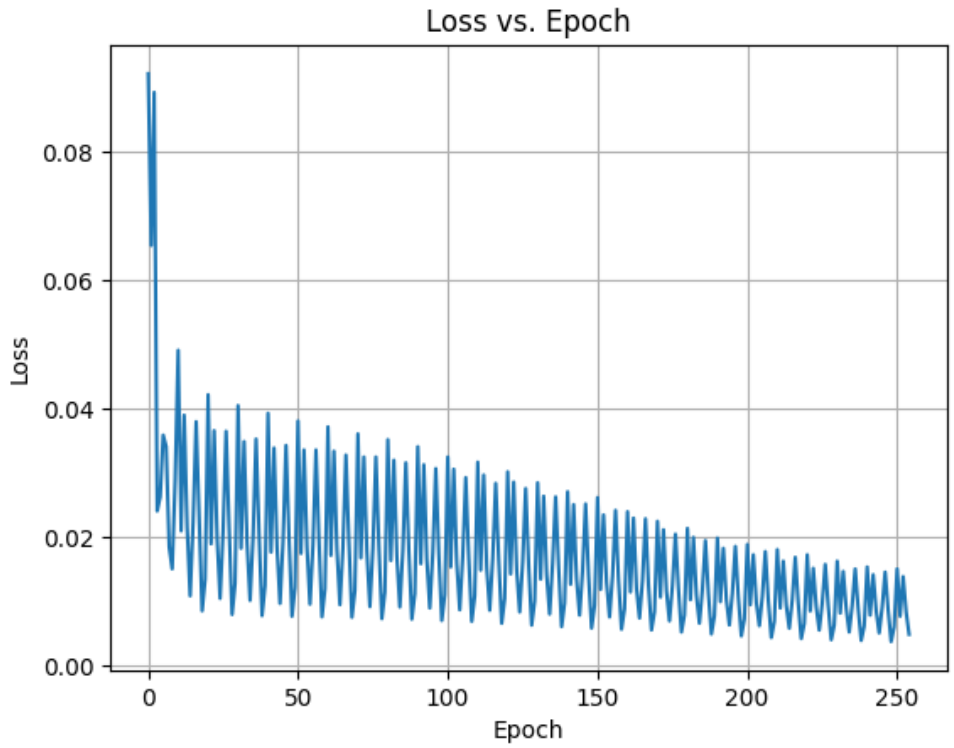


Figure 6.12: Loss vs epoch from human faces dataset on Kaggle Kernel

From our data analysis we can observe the reduction of the loss in our model, thus, we conclude the model predicts more accurately the noise added to our dataset input. We also observe more oscillation during the first epoch iterations and how it gets reduced as our number of iterations increases, providing more stable output results. Thus, this approach indicates the importance of having enough feature parameters in our network. Additionally, we were able to increase the resolution of the images by upgrading our GPU memory, which results in higher quality output pictures. Nonetheless, these new results have shown the improvement of our model on predicting more complex features, such as those in the human face with a high level of accuracy in some samplings.

Chapter 7 Conclusion

This thesis aimed at evaluating and testing the impact that different hyperparameters have on the performance of a Diffusion Probabilistic Model (DPM). These results were evaluated in different datasets, with low complexity such as the landscapes input dataset in Chapter 5, and other input datasets with higher complexity such as the city-view, the vehicle images, and human faces, shown in Chapter 6. By our analysis, we demonstrated the importance of feature engineering and feature selection capabilities, such as increasing the number of feature parameters in the U-Net to 512 to obtained better loss reduction results. Additionally, we explored the importance of adjusting the timesteps and resolution of the image and how they impact the execution time of our overall model. Thus, this thesis studied the best feature parameters for a basic unsupervised and unconditional DPM.

7.1 Thesis Contribution Analysis

In this study on DPM we observed that the most influential hyperparameters were the number of feature parameters used in our U-Net. The number of features represents the dimensionality of the latent space or the hidden representation. A higher feature dimension can capture more intricate patterns, but they also increase the model's complexity. Thus, this depends on the complexity of the dataset we are analyzing. A common practice then is to start with a moderate number such as 64 and experiment based on model performance. By beginning with a low value of feature parameters we can perform faster computations, and thus save our analysis time. If such dataset has a big complexity, it is recommended to increase the number of features to 512, but this may be a limitation in some instances where GPU memory is limited. In such a scenario,

it is recommended to decrease the number of features to 128. If computational demand is not strictly limited, then a value of 512 feature parameters yields the best results, this is based on our experiment analysis. In addition to the feature selection, another important factor for high-resolution images is the image size. In all our experiments we processed the images by reducing the resolution and cropping them to become square images and then we converted them from PNG or JPG format to tensors. We begin with a size of 64x64 and then we increased the resolution as needed. For instance, higher resolution images provide more detail but demand more computational power and memory. Lower resolutions are quicker to train, but for tasks requiring fine details, higher resolutions may be necessary. Thus, a size of 128x128 seemed the most appropriate that yielded the best results and with a reasonable execution time. A higher resolution image with better results may require even more computational memory and power, which in many cases may be limited.

For the timesteps, which refers to the number of steps or iterations the diffusion process goes through to generate an image, we conclude that more timesteps can capture complex tendencies in the data but may increase training time. Too few timesteps may result in incomplete image generation, but too many of them will consume more computational power. Based on our experiments, a range from 1000 to 1500 timesteps seems the most appropriate. The number of epochs, which is one complete pass through the entire training dataset was also crucial in our analysis and performance. Too few epochs may result in underfitting, while too any may lead to overfitting [29]. By early stopping and monitoring validation loss we determine that an appropriate number of epoch iterations for our model was 200-250, depending on the dataset. This was proven to be around the time the model began converging to a reasonable loss and producing higher quality image outputs.

7.2 Future Research Directions

This research has focused on the effects of the hyperparameter parameters selection and with the limited resources available. Thus, an even more ambitious approach with higher computational power may yield faster training times, and thus improving the performance of the model. Faster GPUs with higher memory that can train larger training and sampling data batches may alleviate our time of execution. Then we could make a trade-off analysis by comparing the execution time and the cost of using such expensive hardware components and recommend if such ambitious approaches are reasonable for the improvement achieve, otherwise, a less ambitious approach might be considered. Another important aspect of our approach was the selection of our dataset input images. Most of them while having high complexity such as the human face and the car datasets, they still showcase heterogeneity among the images. The use of a more diverse dataset may become too complex for our model to train and sample, thus opening the opportunity for further research using more classification techniques. Another area of research can be on the extension of DPM for the generation of animations or videos. Adjacently, developing attention mechanisms in DPM to focus on specific regions of the data can be of interest. Nonetheless, such experiments and analysis would require higher computational costs to perform, making such selections another case for the use of more ambitious approaches and better GPU memory.

Bibliography

[1] "Glossary - Convolution," *homepages.inf.ed.ac.uk*. https://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm#:~:text=Convolution%20is%20a%20simple%20mathematical

[2] "Softmax | What is Softmax Activation Function | Introduction to Softmax," *Analytics Vidhya*, Apr. 05, 2021. https://www.analyticsvidhya.com/blog/2021/04/introduction-to-softmax-for-neural-network/

[3] "Byzantium and Italian Renaissance Art," *www.torch.ox.ac.uk*. https://www.torch.ox.ac.uk/article/byzantium-and-italian-renaissance-art

[4] "Convolution," *www.dspguide.com*. https://www.dspguide.com/ch24/1.htm (accessed Oct. 27, 2023).

[5] "Defense Advanced Research Projects Agency (DARPA)." Accessed: Oct. 27, 2023. [Online]. Available: https://media.defense.gov/2022/Mar/10/2002953873/-1/-1/1/DARPA_SBIR_224_R2.PDF

[6] "Human Faces," www.kaggle.com. https://www.kaggle.com/datasets/ashwingupta3012/human-faces/ (accessed Oct. 12, 2023).

[7] "Landscape Pictures," www.kaggle.com. https://www.kaggle.com/datasets/arnaud58/landscape-pictures/ (accessed Sep. 09, 2023).

[8] "Multilabel car and color dataset," www.kaggle.com. https://www.kaggle.com/datasets/julichitai/multilabel-small-car-and-color-dataset/ (accessed Oct. 12, 2023).

[9] "Unpaired Day and Night cityview images," www.kaggle.com. https://www.kaggle.com/datasets/heonh0/daynight-cityview/ (accessed Oct. 05, 2023).

[10] "What is a max pooling layer in CNN?," *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/what-is-a-max-pooling-layer-in-cnn

[11] A. Buis, "Paleolithic Art," *pressbooks.bccampus.ca*, Available: https://pressbooks.bccampus.ca/cavestocathedrals/chapter/paleolithic/

[12] Garcia, "Harold Cohen and AARON—A 40-Year Collaboration," *Computer History Museum*, Aug. 23, 2016. https://computerhistory.org/blog/harold-cohen-and-aaron-a-40-year-collaboration/

[13] Ali Grami, "The Gaussian Distribution," in *Probability, Random Variables, Statistics, and Random Processes: Fundamentals & Applications* , Wiley, 2019, pp.201-238, doi: 10.1002/9781119300847.ch7.

[14] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In International Conference on Learning Representations, 2019.

[15] Bengio, Y. and Thibodeau-Laufer, E. Deep generative stochastic networks trainable by backprop. arXiv preprint arXiv:1306.1091, 2013.

[16] Meng, Y. Song, W. Li, and S. Ermon, "Estimating High Order Gradients of the Data Distribution by Denoising." Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2111.04726.pdf

[17] Robert and G. Roberts, "Rao-Blackwellization in the MCMC era." Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2101.01011.pdf

[18] Beren, "The History of the Modern CPU," *History-Computer*, Jan. 04, 2021. https://history-computer.com/the-history-of-the-modern-cpu/

[19] Kingma and J. Lei Ba, “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION,” Jan. 2017. Available: https://arxiv.org/pdf/1412.6980.pdf

[20] Diederik P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In Advances in Neural Information Processing Systems, pages 4743–4751, 2016

[21] Chigozie, W. Nwankpa, A. Ijomah, S. Gachagan, and Marshall, “Activation Functions: Comparison of Trends in Practice and Research for Deep Learning,” 2018. Available: https://arxiv.org/pdf/1811.03378.pdf

[22] Feller, W. On the theory of stochastic processes, with particular reference to applications. In Proceedings of the [First] Berkeley Symposium on Mathematical Statistics and Probability. The Regents of the University of California, 1949.

[23] Feng-Hsiung Hsu, "IBM's Deep Blue Chess grandmaster chips," in IEEE Micro, vol. 19, no. 2, pp. 70-81, March-April 1999, doi: 10.1109/40.755469.

[24] Flores-Vidal, P., Gómez, D., Castro, J. *et al.* New Aggregation Approaches with HSV to Color Edge Detection. *Int J Comput Intell Syst* **15**, 78 (2022). https://doi.org/10.1007/s44196-022-00137-x

[25] Grosse, R. B., Maddison, C. J., and Salakhutdinov, R. Annealing between distributions by averaging moments. In Advances in Neural Information Processing Systems, pp. 2769–2777, 2013.

[26] J. Dai *et al.*, “Demystify Transformers & Convolutions in Modern Image Deep Networks.” Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2211.05781.pdf

[27] J. Ho, A. Jain, and P. Abbeel, “Denoising Diffusion Probabilistic Models,” Dec. 2020. Available: https://arxiv.org/pdf/2006.11239.pdf

[28] J. Liu, “THE GRAND PICTURE BEHIND JENSEN’S INEQUALITY,” 2023. Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2211.08269.pdf

[29] J. Schmidt, “Testing for Overfitting,” 2023. Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2305.05792.pdf

[30] J. Weizenbaum, “ELIZA---a computer program for the study of natural language communication between man and machine,” *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, Jan. 1966, doi: https://doi.org/10.1145/365153.365168.

[31] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In International Conference on Machine Learning, pages 2256–2265, 2015:

[32] Kingma, D. P. and Welling, M. Auto-Encoding Variational Bayes. International Conference on Learning Representations, December 2013.

[33] Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. Computer Science Department University of Toronto Tech. Rep., 2009.

[34] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, “Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation.” Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/1802.02611v3.pdf.

[35] Langevin, P. Sur la theorie du mouvement brownie. CR Acad. Sci. Paris, 146(530-533), 1908

[36] LeCun, Y. and Cortes, C. The MNIST database of hand-written digits. 1998

[37] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning.” Available: https://arxiv.org/pdf/1605.08695.pdf

[38] M. Basavarajaiah, “6 basic things to know about Convolution,” *Medium*, Apr. 02, 2019. https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411

[39] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.” Available: https://arxiv.org/pdf/1706.08500.pdf

[40] M. Jankowiak and D. Phan, “Surrogate Likelihoods for Variational Annealed Importance Sampling.” Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2112.12194.pdf

[41] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” 2015. Available: https://arxiv.org/pdf/1505.04597.pdf

[42] Olena, “A Brief History of GPU,” *Medium*, Feb. 22, 2018. https://medium.com/altumea/a-brief-history-of-gpu-47d98d6a0f8a

[43] P. Dhariwal and A. Nichol, “Diffusion Models Beat GANs on Image Synthesis,” Jun. 2021. Available: https://arxiv.org/pdf/2105.05233.pdf

[44] R. Anyoha, “The History of Artificial Intelligence,” *Science in the News*, Aug. 28, 2017. https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/

[45] R. Holinirina *et al.*, “Calculating Point Spread Functions Calculating Point Spread Functions: Methods, Pitfalls and Solutions.” Accessed: Oct. 27, 2023. [Online]. Available: https://arxiv.org/pdf/2301.13515.pdf

[46] S. Ji, Z. Zhang, S. Ying, L. Wang, X. Zhao and Y. Gao, "Kullback–Leibler Divergence Metric Learning," in IEEE Transactions on Cybernetics, vol. 52, no. 4, pp. 2047-2058, April 2022, doi: 10.1109/TCYB.2020.3008248.

[47]

[48] S. Li *et al.*, “PyTorch Distributed: Experiences on Accelerating Data Parallel Training.” Accessed: Oct. 12, 2022. [Online]. Available: https://arxiv.org/pdf/2006.15704.pdf

[49] S. Ruder, “An overview of gradient descent optimization algorithms *,” Jun. 2017. Available: https://arxiv.org/pdf/1609.04747.pdf

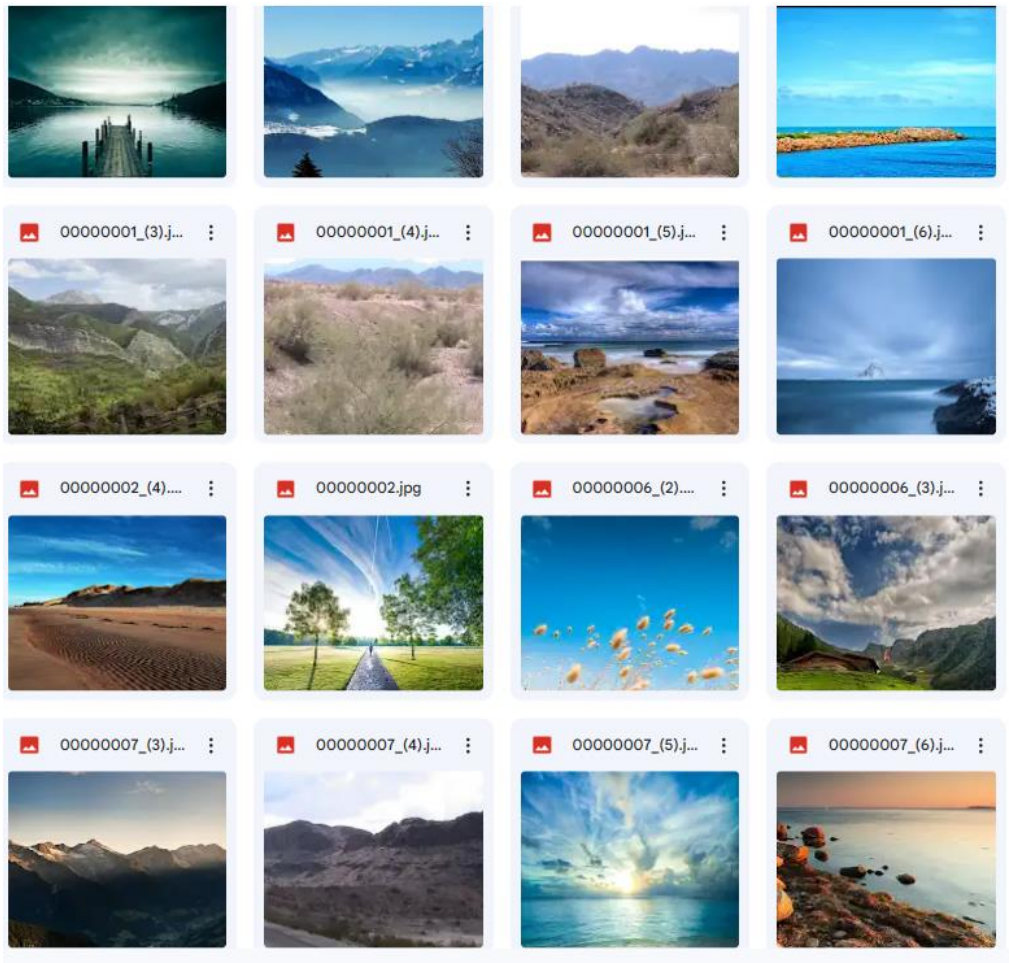
[50] Turing, “Computing Machinery and Intelligence,” *Mind*, vol. LIX, no. 236, pp. 433–460, 1950, doi: https://doi.org/10.1093/mind/lix.236.433.

[51] Y. Jia *et al.*, “Caffe: Convolutional Architecture for Fast Feature Embedding *.” Available: https://arxiv.org/pdf/1408.5093.pdf

Appendix

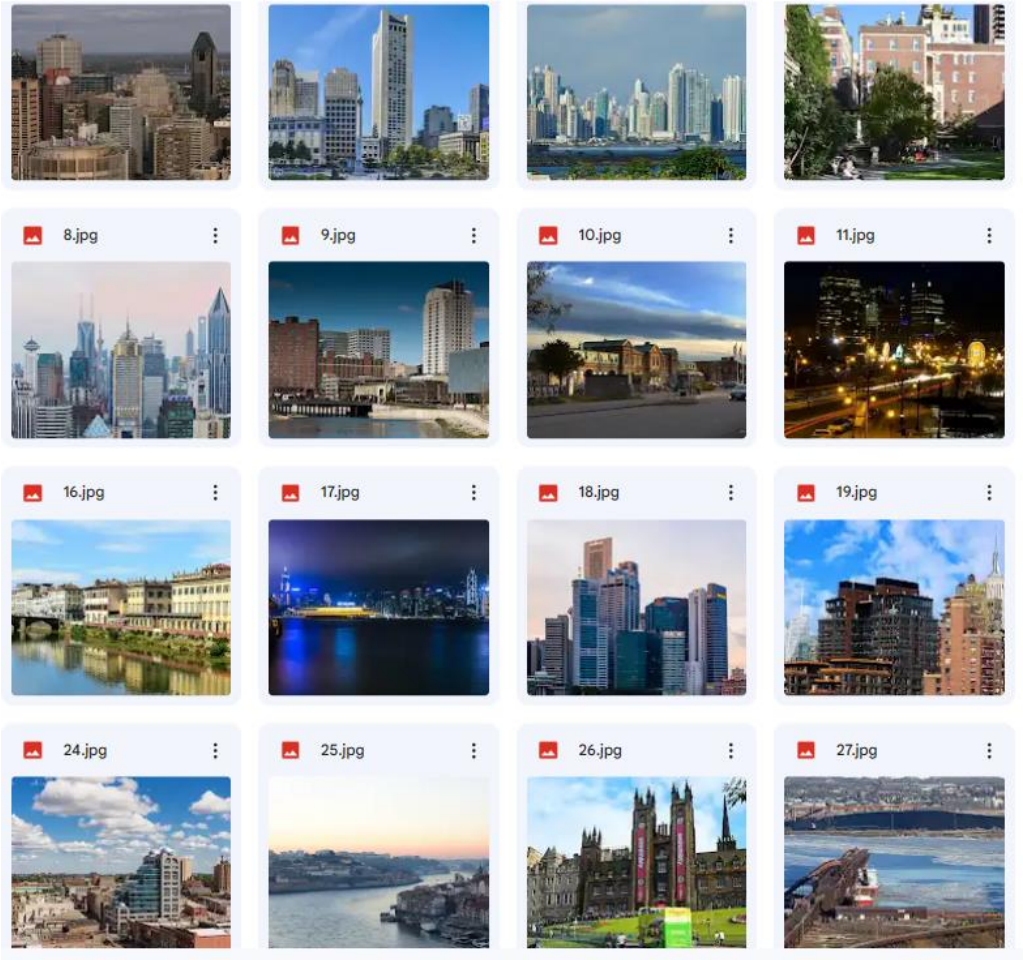
A. Original Landscape Dataset

Below are 16 sample images from the “Landscape Pictures” dataset downloaded from Kaggle [7]



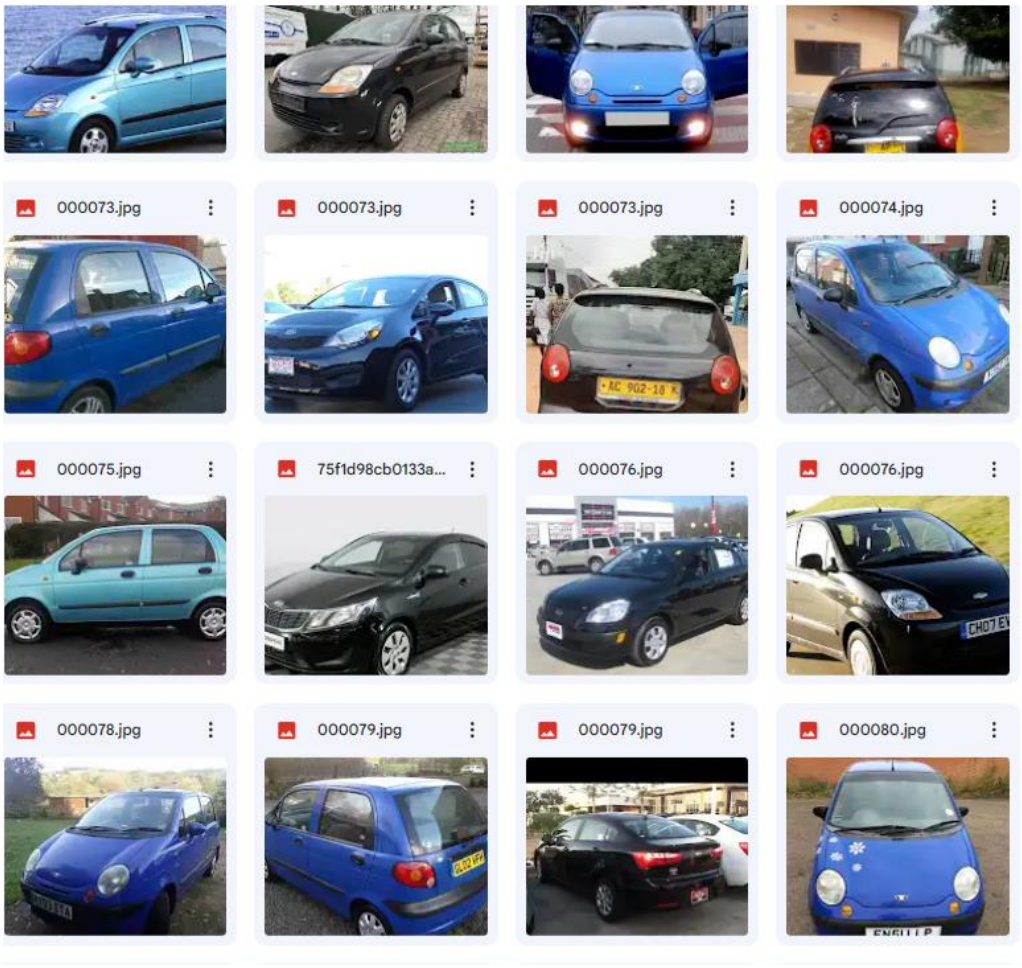
B. Original City-View Dataset

Below are 16 sample images from the “Unpaired day and night city-view images” dataset downloaded from Kaggle [9]



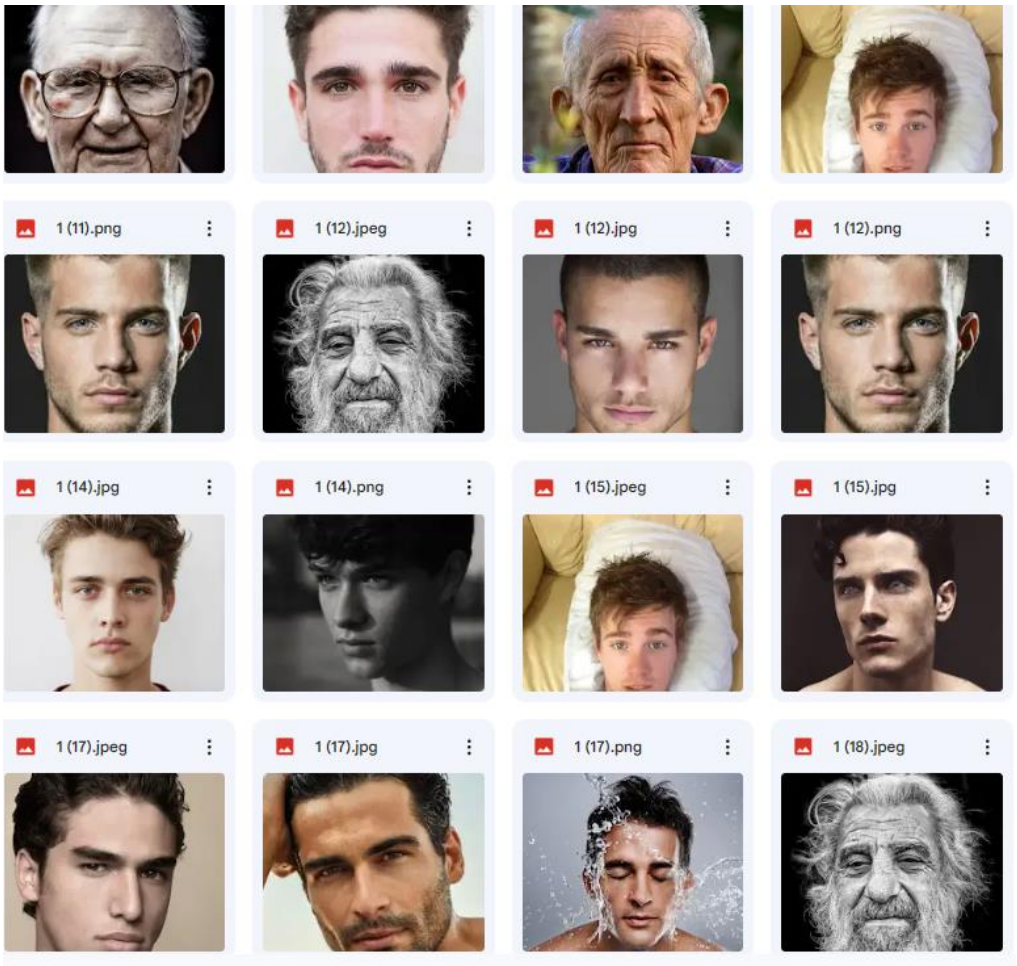
C. Original Car Dataset

Below are 16 sample images from the “Multilabel car and color” dataset downloaded from Kaggle [8]



D. Original Human Face Dataset

Below are 16 sample images from the Human Face Dataset downloaded from Kaggle [6]



E. Experimental Diffusion Probabilistic Model Code

Below is the experimental diffusion probabilistic model code used during training and sampling of our results shown in Chapters 5 and 6 by using the datasets shown in the previous appendices.

```
#The following Code is used to train and sample the Diffusion Probabilistic Model (DPM) results
!pip install diffusers[training]
from dataclasses import dataclass
import torch
from accelerate import notebook_launcher
from tqdm.auto import tqdm
from pathlib import Path
from datasets import load_dataset
import torch.nn.functional as F
from diffusers import DDPMPipeline
from diffusers.utils import make_image_grid
from diffusers.optimization import get_cosine_schedule_with_warmup
import math
import os

gradient_accumulation_steps = 1
warmup_steps = 500
precision = "fp16"
seed = 0

#The following function plots the images produced
def plot(images):
    plt.figure(figsize=(32, 32))
    plt.imshow(torch.cat([
```

```
        torch.cat([i for i in images.cpu()], dim=-1)
    ], dim=-2).permute(1, 2, 0).cpu())
    plt.show()

#The following function saves the images produced
def save(images, path):
    grid = torchvision.utils.make_grid(images)
    ndarr = grid.permute(1, 2, 0).to('cpu').numpy()
    im = Image.fromarray(ndarr)
    im.save(path)



def sample_data(batch_size,epoch, pipeline, seed):
    images = pipeline(
        batch_size=batch_size,
        generator=torch.manual_seed(seed),
    ).images
    grid = make_image_grid(images, rows=4, cols=4)
    output_path = os.path.join(config.output_dir, output)
    os.makedirs(test_dir, exist_ok=True)
    grid.save(f"{output_path}/{epoch:04d}.png")



#The following function reads the dataset data
def get_data(dataset_path, batch_size, image_size):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(80), # args.image_size + 1/4 *args.image_size
        torchvision.transforms.RandomResizedCrop(image_size, scale=(0.8, 1.0)),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

```
        ])
    dataset = torchvision.datasets.ImageFolder(dataset_path, transform=transforms)
    dataloader = DataLoader(dataset, batch_size = batch_size, shuffle=True)
    return dataloader

#Define Optimizer by using the Adams Algorithm
optimizer = torch.optim.AdamW(model.parameters(), lr=config.learning_rate)
scheduler = get_cosine_schedule_with_warmup(
    optimizer=optimizer,
    warmup_steps=warmup_steps,
    train_steps=(len(train_dataloader) * num_epochs),
)
#Define U-Net by using UNet2DModel from Diffusers
model = UNet2DModel(
    sample_size=image_size,
    in_channels=channels,
    out_channels=channels,
    layers_per_block=2,
    block_out_channels=(features/4, features/4, features/2, features/2, features, features),
    down_block_types=(
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",
        "DownBlock2D",
    ),
    up_block_types=(
        "UpBlock2D",
        "AttnUpBlock2D",
```

```
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
)

#Load the Data
data_load = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)

class DPM:
    def __init__(self, steps=1000, beta_0=1e-4, beta_T=0.02, size=256, device="cuda"):
        self._size = size
        self.device = device
        self.noise = noise
        self.beta_0 = beta_0
        self.beta_T = beta_T
        self.beta = self.noise_schedule().to(device)
        self.alpha = 1. - self.beta
        self.alpha_hat = torch.cumprod(self.alpha, dim=0)

    #Define Beta
    def noise_schedule(self):
        return torch.linspace(self.beta_0, self.beta_T, self.noise_steps)

    #Add Noise to Images
    def noise_images(self, x, t):
        temp = torch.sqrt(self.alpha_hat[t])[:, None, None, None]
        temp2 = torch.sqrt(1. - self.alpha_hat[t])[:, None, None, None]
        e = torch.rand_like(x)
```

```
        return temp * x + temp2 * e, e


    #Define Timesteps
    def timesteps(self, n):
        return torch.randint(low=1, high=self.noise, size=(n,))


    #Sample Images
    def sample(self, model, n):
        logging.info(f"Sampling {n} new images...")
        model.eval()
        with torch.no_grad():
            x = torch.randn(n, 3, self.size, self.size).to(self.device)
            for i in tqdm(reversed(range(1, self.noise)), position=0):
                t = (torch.ones(n) * i).long().to(self.device)
                predicted_noise = model(x, t)
                alpha = self.alpha[t][:, None, None, None]
                alpha_hat = self.alpha_hat[t][:, None, None, None]
                beta = self.beta[t][:, None, None, None]
                if i > 1:
                    noise = torch.randn_like(x)
                else:
                    noise = torch.zeros_like(x)
                x = 1 / torch.sqrt(alpha) * (x - ((1-alpha) / (torch.sqrt(1 - alpha_hat))) *
predicted_noise) + torch.sqrt(beta) * noise
        model.train()
        x = (x.clamp(-1, 1) + 1) / 2
        x = (x * 255).type(torch.uint8)
        return x
```

```
#The following function will train the DPM to predict the added noise
def train(run_name, epochs, batch_size, image_size, dataset_path, device, lr):
    save_image_epochs = 10
    save_model_epochs = 30
    setup_logging(run_name)
    device = device
    dataloader = get_data(dataset_path, batch_size, image_size)
    model = UNet().to(device)
    optimizer = optim.AdamW(model.parameters(), lr=lr)
    mse = nn.MSELoss()
    diffusion = Diffusion(img_size=image_size, device=device)
    logger = SummaryWriter(os.path.join("runs", run_name))
    l = len(dataloader)
    global_step = 0

    for epoch in range(epochs):
        logging.info(f"Starting epoch {epoch}:")
        pbar = tqdm(dataloader)
        for i, (images, _) in enumerate(pbar):
            images = images.to(device)
            t = diffusion.sample_timesteps(images.shape[0]).to(device)
            x_t, noise = diffusion.noise_images(images, t)


            progress_bar = tqdm(total=len(data_load), disable=not
accelerator.is_local_main_process)
            progress_bar.set_description(f"Epoch {epoch}")


            for step, batch in enumerate(data_load):
                clean_images = batch["images"]
                noise = torch.randn(clean_images.shape).to(clean_images.device)
```

```
            bs = clean_images.shape[0]

            timesteps = torch.randint(
                0, noise_scheduler.config.num_train_timesteps, (bs,), device=clean_images.device
            ).long()
            noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)
            noise_pred = model(noisy_images, timesteps, return_dict=False)[0]
            loss = F.mse_loss(noise_pred, noise)
            backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
            print(loss)
            progress_bar.update(1)
            logs = {"loss": loss.detach().item(), "lr": lr_scheduler.get_last_lr()[0], "step": global_step}
            progress_bar.set_postfix(**logs)

        if (epoch + 1) % config.save_image_epochs == 0 or epoch == epochs - 1:
            evaluate(config, epoch, pipeline)

        if (epoch + 1) % config.save_model_epochs == 0 or epoch == epochs - 1:
            pipeline.save_pretrained(config.output_dir)
            # sampled_images = diffusion.sample(x_t, model, n=images.shape[0])
        sampled_images = diffusion.sample(model, n=images.shape[0])
        save_images(sampled_images, os.path.join("results", run_name, f"{epoch}.jpg"))
        torch.save(model.state_dict(), os.path.join("models", run_name, f"ckpt.pt"))
def launch():
    image_size = input("Enter Image Size: ")# Recommended 128
    batch_size = input("Enter Batch Size: ")# Recommended 16
```

```
    timesteps = input("Enter Number of Timesteps: ")#Recommended 1500
    channels = input("Enter Number of Channels: ")#Recommended 3 for RGB Images
    epochs = input("Enter Number of Epoch Iterations: ")# Recommended 250
    learning_rate = input("Enter Learning Rate: ")# Recommended 1e-4
    features = input("Enter Number of Feature Parameters: ")#Recommended 512
    beta_0 = input("Enter Initial Beta: ")#Recommended 1e-4
    beta_T = input("Enter Final Beta: ")#Recommended 0.02
    dataset = input("Enter Dataset Path")#Path for Human Faces on Kaggle:
/kaggle/input/human-faces
    output = "Results"
    #train(run_name = "DDPM_Unconditional", epochs = 300, batch_size = 12, image_size =
64, dataset_path = "/notebooks/datasets", device = "cuda", lr = 3e-4)
    args = (model, image_size, )
    notebook_launcher(train, args, batch_size, timesteps, channels, epochs, learning_rate,
                      features, beta_0, beta_T, output, dataset num_processes=1)
if __name__ == "__main__":
    launch()
```