TU WIEN Informatics

Text in Bewegung umwandeln: Grundlagen von Diffusionstransformatoren für Text-zu-Video Generierung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Carmen Halbeisen
Matrikelnummer 52014160

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr.techn. Dipl.-Ing. Clemens Heitzinger

Wien, 22. Jänner 2025

Carmen Halbeisen	Clemens Heitzinger

TU WIEN Informatics

Transforming Text into Motion: Fundamentals of Diffusion Transformers for Text-to-Video Generation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Carmen Halbeisen
Registration Number 52014160

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr.techn. Dipl.-Ing. Clemens Heitzinger

Vienna, January 22, 2025

Carmen Halbeisen	Clemens Heitzinger

Erklärung zur Verfassung der Arbeit

Carmen Halbeisen

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 22. Jänner 2025

Carmen Halbeisen

Kurzfassung

Die Videogenerierung hat kürzlich bemerkenswerte Fortschritte erzielt und ermöglicht nun die Erstellung von Videos, die zunehmend qualitativ hochwertiger und realistischer sind. Moderne Ansätze greifen auf die Erfolge von Transformermodellen aus der Sprachverarbeitung zurück und ersetzen traditionelle U-NETs durch Diffusionsmodelle mit einer Vision-Transformer Architektur. Die Implementierungsdetails führender Modelle wie Sora sind nicht öffentlich zugänglich, jedoch basieren andere Modelle wie Latte, GenTron und SnapVideo auf den Konzepten von Sora, zu denen detailliertere Informationen zur Implementierung verfügbar sind. Diese Modelle basieren ebenfalls auf Diffusionstransformatoren und verwenden unterschiedliche Methoden, um die zeitliche Dimension zu modellieren, ohne die Präzision der einzelnen Frames zu beeinträchtigen.

In dieser Arbeit werden die architektonischen Ansätze von Latte, GenTron und SnapVideo untersucht, insbesondere ihre Strategien zur Erfassung räumlicher und zeitlicher Aspekte sowie zur Integration von Textanweisungen. Ausgangspunkt ist ein Diffusionstransformator für Bildgenerierung, der auf Videogenerierung erweitert wird. Darüber hinaus werden die Modelle Latte und SnapVideo so angepasst, dass sie statt kategorischer Eingaben nun auch Textanweisungen verarbeiten können. Die Bildgenerierungsergebnisse werden mit Metriken wie FID, CLIPSIM, SSIM, PSNR und LPIPS bewertet, während die Qualität der Videogenerierung sowohl mit FVD als auch durch die Analyse der einzelnen Frames mit den Bildmetriken beurteilt wird.

Die Modelle weisen klare Unterschiede bezüglich Qualität der generierten Videos auf. GenTron bearbeitet die räumliche und zeitliche Dimension separat innerhalb eines einzigen Transformer-Blocks. Dies führt zu präzisen Einzelbildern, jedoch treten gelegentlich inkohärente Bewegungen zwischen den Frames auf. Latte erreicht eine bessere zeitliche Kohärenz, indem es zwei separate Transformer-Blöcke nutzt – einen für die Analyse der Einzelbilder und einen für die Bewegungen im Video. Dies geht jedoch zulasten der Bildqualität. SnapVideo hingegen verarbeitet beide Dimensionen gleichzeitig, was häufig zu statischen und unscharfen Videos führt. Bei der Textintegration erzielt die Einbindung von Text während der Verarbeitung einzelner Frames die besten Ergebnisse, während die Integration auf der zeitlichen Ebene die Videoqualität negativ beeinflusst.
Die Ergebnisse verdeutlichen, wie stark die unterschiedlichen Ansätze zur Integration räumlicher, zeitlicher und textueller Informationen die Videoqualität beeinflussen und wie wichtig es ist, diesen Aspekten besondere Beachtung zu schenken.

Abstract

Video generation is a fast-evolving research field, producing increasingly impressive and lifelike videos. New advances draw inspiration from the success of Transformer models in Natural language processing, by utilizing Diffusion models with a Vision Transformer backbone, replacing the traditional U-NET. The architectural details of leading video generation models like Sora are not publicly disclosed; however, models such as Latte, GenTron, and SnapVideo are built upon the concepts of Sora, for which more detailed architectural information is available.

This thesis explores the architectural details of Latte, Gentron, and SnapVideo, examining how the models operate on both spatial and temporal dimensions while integrating textual guidance during the generation process. Beginning with a basic implementation of a Diffusion Transformer for image generation, the code is extended to video generation, following the implementation details of Latte, GenTron, and SnapVideo from their respective papers and, when available, their official code. Additionally, the Latte and SnapVideo models, originally conditioned on class label inputs, are adapted to video-to-text generation, testing various methods of integrating the textual prompt into the video generation pipeline. The image generation results are evaluated using the image-based metrics FID, CLIPSIM, SSIM, PSNR, and LPIPS, while the video generation samples are assessed using FVD and frame-by-frame comparisons based on the image generation metrics.

The different approaches to handling spatial and temporal dimensions across the three examined architectures resulted in significant differences in the quality of the generated videos. GenTron employs a Transformer block that separates the multi-head attention into spatial and temporal components, which leads to high spatial accuracy but shows occasional incoherent movement between frames. Latte utilizes two distinct Transformer blocks – one for spatial attention and the other for temporal attention, which improves the temporal coherence but results in lower frame quality. SnapVideo uses joint spatiotemporal attention, producing the least favorable results by generating static videos with blurry images. Regarding text guidance, the best results were achieved when the text was integrated during spatial attention. Conversely, adding the text at the temporal dimension decreased the overall quality of the generated video.
The results highlight the importance of carefully considering how spatial, temporal, and textual information are integrated to produce high-quality video generation.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Aim of this Work 2

2 Background **5**
2.1 Diffusion Models 5
2.2 Architecture 21
2.3 Evaluation Metrics 31

3 Methodology **35**
3.1 Image Generation 35
3.2 Video Generation 39
3.3 Text-to-Video Generation 49

4 Results **59**
4.1 Image Generation (DiT) 59
4.2 Video Generation 61

5 Discussion **67**
5.1 From Image to Video Generation 67
5.2 Video Generation – Spatial and Temporal Attention 68
5.3 From Class to Text Guidance 70
5.4 Evaluation Metrics 71
5.5 Limitations 71

6 Conclusion **73**

7 Appendix **75**
7.1 GenTron Pseudocode 75

7.2 Latte v1 Pseudocode 77
7.3 Latte v2 Pseudocode 79
7.4 Latte v3 Pseudocode 81
7.5 SnapVideo v1 Pseudocode 83
7.6 SnapVideo v2 Pseudocode 85

Overview of Generative AI Tools Used **87**

List of Figures **89**

List of Tables **91**

List of Algorithms **93**

Bibliography **95**

CHAPTER 1

Introduction

Image generation is a research field that is rapidly changing and has demonstrated continous improvements regarding image quality. Models like Imagen from Google [SCS+22], DALLE-2 from OpenAI [RDN+22], DiT from Meta [PX23] are based on the same underlying concept of diffusion models and produce photorealistic images with wide diversity.
Diffusion models are based on the principle of transforming a complex image distribution into a simple, well-known distribution like Gaussian. A denoising network is afterward trained to reverse this transformation; in other words, it learns to reconstruct the complex image distribution from the simplified one. Given an initial image sample $\mathbf{x}_0$ from the dataset, the diffusion process gradually corrupts the sample, until it eventually resembles pure noise. The denoising network tries to predict the noise contained in the corrupted image, learning to reconstruct the initial clear input sample $\mathbf{x}_0$. Over time, the model can generate new images from a given noisy sample.
The diffusion process only defines the mathematical background for noise addition and removal. The denoising network, on the other hand, is the key element for generating new images. As a result, a well-designed architecture for the denoising network is crucial for the model's performance. Current works primarily use a Transformer or U-Net-based model for training. Figure 1.1 visualizes the intuition behind the diffusion process.

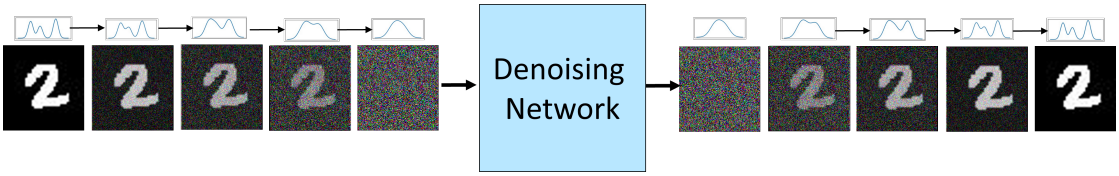


Figure 1.1: Intuition behind diffusion models

Building on the success of diffusion models in image generation, recent research has started to explore their potential for generating videos. Under the assumption that a

video is a sequence of – mostly independent – images, the pre-trained image generations were adapted for video predictions by inserting temporal layers. While these video generation models demonstrate promising results, the movements between frames often lack coherence. This led to extensive research to improve the denoising network backbone, with different approaches to model the spatial and temporal dynamics of the video.
Models like Sora [BPH+24] have set new standards and are able to generate high-quality videos that accurately represent a given text prompt. Drawing on the success of natural language processing (NLP) models that employ transformer architectures, newly released video generation models are based on the Vision Transformer. Vision Transformers are an adaptation of the original transformer model for image and video processing. These models can capture global dependencies within videos, making them ideal for video generation where an object's movement in a frame depends on previous frames. However, transformer-based models come with significant computational and storage costs. Processing long videos in a single batch exceeds the capacity of current hardware. To address this, different architectural modifications have been proposed to minimize redundant information while keeping important details.
GenTron [CXR+23] and Latte [MWJ+24a] suggest focusing separately on the spatial and temporal dimensions of a given video by first analyzing each frame individually, then following specific pixels and how they change over time. SnapVideo [MSS+24], on the other hand, attempts to simultaneously model both space and time by dividing a video into groups and processing each group independently. However, information reduction can be a difficult task, as some information is insignificant and can be ignored, while other details are necessary for the model to understand the input data. This makes it even more important to understand the architectural details that contribute to the success of leading video Diffusion Transformer models.
Yet, most of the prominent video generation models remain closed-source. While projects like Latte [MWJ+24a] present themselves as an "open-source Sora," the official code only includes class-based video generation.

1.1 Aim of this Work

This work aims to bridge the gap between understanding the theory behind the image and video generation models based on Diffusion Transformers and their implementation. By implementing three different architectures – GenTron [CXR+23], Latte [MWJ+24a], and SnapVideo [MSS+24] – this work tries to provide a deeper understanding of video generation models. Furthermore, different spatial and temporal modeling will be explored, as well as their effect on the quality of the predicted videos. The code is based on the official implementation of Denoising Diffusion Probabilistic Models (DDPM) [Nic21], the official implementation of the Diffusion Transformer (DiT) network [PX22] for class-to-image generation, as well as the official implementation of the Latte model [MWJ+24b] for class-conditional video generation. The Diffusion Transformer network implementation serves as a foundation, learning more about preprocessing techniques and architectural details required for generating high-quality images. The class-to-image generation model

will subsequently be adapted for video generation, following the architecture of GenTron, Latte, and SnapVideo. Additionally, different approaches will be tested to incorporate text guidance in the Vision Transformer backbone. The implementation of DiT will be trained on a class-to-image generation model, while the different video generation models will be trained on a text-to-video dataset. Using standard image and video generation metrics, the different models will be evaluated, where the evaluation results will be compared with the actual perceptual sampling quality.

CHAPTER 2

Background

2.1 Diffusion Models

Diffusion models are based on a field of physics that examines systems that are not in thermodynamic equilibrium. An illustrative example can be ink spreading in water: In the beginning, the ink is concentrated in one spot and the rest of the water remains clear. The system is in an imbalanced state, but eventually, by the laws of physics, the drop will diffuse into the water until it reaches an equilibrium. Trying to reverse this process, meaning bringing the ink back to a single drop, is not possible. However, this is the core challenge that diffusion models try to address: taking corrupted data and reconstructing its original form.

Diffusion models take a given image sample and systematically corrupt it with Gaussian noise. Over time, more noise is introduced to the data sample until it resembles random noise – analogous to the ink completely diffused in water. During this process, the complex distribution of the input image is converted into a simpler, more manageable one. A denoising network is now trained to reverse this process, by predicting the noise contained in the corrupted input image. Starting from the noisy input data, the denoising network gradually removes the estimated noise until the original data is reconstructed.

The advantage of diffusion models over other generative models, such as GANs (Generative Adversarial Networks) or VAE (Variational Autoencoders), is that the prediction process happens step-by-step. Instead of trying to model the entire complex data distribution instantly, diffusion models break it down into smaller, more manageable steps. During each step, the diffusion model tries to estimate small perturbations in the given noisy input data, until it has reconstructed the original image [SDWMG15].

2.1.1 The Generic Pipeline

The diffusion model defines a Markov chain of diffusion steps. The forward process describes the concept of adding noise to the input data, while the reverse process defines the procedure of gradually removing noise to bring the data back to its original distribution. The forward process is applied to the input data during training and sampling. However, it is important to note that the forward process does not include any trainable parameters.
A generative model subsequently has to predict the noise added to the data. More specifically, the model has to estimate the Gaussian mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\Sigma}$ of the noise added to the data. The reverse process involves removing the predicted noise, as a chain of reverse transitions [SDWMG15][CKS23].
After training the model, new data can be generated during the sampling procedure.
The diffusion model can be described in two different ways, using the discrete and continuous formulations. The key difference between these formulations is the way the diffusion timestep t is defined [Zha23]. The diffusion timestep defines the current stage of the forward and reverse diffusion. During the forward process, the time step gradually increases to its final value T. Conversely, the reverse process starts at the final time step T and decreases back to zero. In the discrete formulation, time steps are restricted to integer values, ranging from zero to a specified final time step T [HJA20]. Contrarily, the continuous formulations use continuous time steps within the interval $[0, 1]$, theoretically allowing an infinite number of time steps [SSDK+21]. Table 2.1 shows an overview of the notations used in the next chapters with their corresponding description.

2.1.2 The Nature of Noise: Gaussian Distribution

The diffusion process is based on probability distributions, with the Gaussian distribution playing a central role. The Gaussian distribution is defined by its mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\Sigma}$. The mean defines the center of the noise and the variance indicates the dispersion of the noise. The forward and reverse diffusion processes are defined utilizing a conditional distribution. Given two variables $\mathbf{x}$ and $\mathbf{y}$, the conditional distribution $p(\mathbf{x}|\mathbf{y})$ describes the probability that variable $\mathbf{x}$ takes a value if the value of $\mathbf{y}$ is known.
During the forward diffusion, the corruption of a given sample $\mathbf{x}_t$ at diffusion time step t is dependent on the state of the sample at the previous state denoted as $\mathbf{x}_{t-1}$, starting with the clear sample $\mathbf{x}_0$ at time step $t = 0$. This process can be described by the conditional probability

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t). \tag{2.1}$$

The notation in (2.1) describes the Gaussion probability density function $\mathcal{N}$ parameterized by $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ evaluated at point $\mathbf{x}_t$.
A common choice for the conditional probability distribution is a Gaussian distribution with a diagonal covariance structure, i.e.,

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}_t, \sigma_t^2\mathbf{I}), \tag{2.2}$$

Notation	Terminology
$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Variable $\mathbf{x}$ follows the Gaussian distribution with mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\Sigma}$
$\mathbf{x}_0$	Initial Training sample
t	Time step
T	Total number of time steps
$\mathbf{x}_t$	Noisy sample at time step t
$\boldsymbol{\epsilon}_t$	Noise at time step t
$q(\mathbf{x}_0)$	Distribution of original training sample at time step 0
$p(\mathbf{x}_t)$	Distribution of noisy sample at time step t
$q(\mathbf{x}_T)$	Distribution of noisy sample at final time step T
$q(\mathbf{x}_t \vert \mathbf{x}_{t-1})$	Forward transition
$q(\mathbf{x}_{t-1} \vert \mathbf{x}_t)$	Reverse transition
$\mathbf{I}$	Identity matrix
$\mathcal{N}(0, \mathbf{I})$	Gaussian distribution
$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	$\mathbf{x}$ is sampled from Gaussian distribution with mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\Sigma}$
θ	The parameters of the denoising network

Table 2.1: Overview Notations and Terminologies used to describe the diffusion model

where σ_t is the standard deviation.
To generate a noisier version $\mathbf{x}_1$ from a given clear sample $\mathbf{x}_0$, sampling from the distribution $\mathbf{x}_t \sim p(\mathbf{x}_t|\mathbf{x}_{t-1})$ is required. However, random sampling is not differentiable. Therefore, a re-parametrization trick is introduced, to express the random variable $\mathbf{x}_t$ as a deterministic variable. By sampling an independent random variable $\boldsymbol{\epsilon}_t$ from the Gaussian distribution $\boldsymbol{\epsilon}_t \sim \mathcal{N}(0, \mathbf{I})$, the sampling can be defined as

$$\mathbf{x}_t = \boldsymbol{\mu}_t + \sigma_t \boldsymbol{\epsilon}_t, \tag{2.3}$$

given the mean $\boldsymbol{\mu}_t$ and the standard deviation σ_t of the distribution. This re-parametrization trick is applied during both the forward and reverse diffusion steps.

2.1.3 Two Formulations of Diffusion Models

Discrete Formulation: Denoising Diffusion Probabilistic Models (DDPMs)

The DDPM is a popular discrete formulation of diffusion models due to its straightforward implementation and high quality in generated samples.

Forward Process During the forward process, DDPMs corrupt the training sample according to the Markovian process: given the distribution of the training dataset $q(\mathbf{x})$, calculate $q(\mathbf{x}_t)$ from known $q(\mathbf{x}_{t-1})$.
The Markov chain is a model, where the current state $\mathbf{x}_t$ at time step t is only influenced

by the previous one $\mathbf{x}_{t-1}$. Starting with the original (non-noisy) sample $\mathbf{x}_0$ at time step $t = 0$, the subsequent data point $\mathbf{x}_1$ can be computed by adding a small amount of Gaussian noise. This process follows the transition kernel which is defined as

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}), \tag{2.4}$$

where $\mathbf{x}_t$ is sampled from the Gaussian distribution of mean $\sqrt{1-\beta_t}\mathbf{x}_{t-1}$ and variance $\beta_t\mathbf{I}$. The variance schedule β_t is a hyperparameter and serves as a scaling factor for mean and variance, defining the rate of noise added at each step in the Markov chain.
The mean of the distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ is a scaled version of at previous state $\mathbf{x}_{t-1}$, shifting the sample by the factor $\sqrt{1-\beta_t}$. The variance introduces noise around the shifted version of $\mathbf{x}_{t-1}$, by adding independent noise to each element – in the context of image generation, to each pixel. A larger value for β_t at time step t results in a greater amount of noise being added.
Through multiple time steps, a sequence of noise samples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$ are produced, where the initial data sample $\mathbf{x}_0$ gradually loses its distinguishable features over time. After T steps, the original data $\mathbf{x}_0$ converges to pure Gaussian noise. The entire process, from the initial time step $t = 0$ to the final time step $t = T$, can be represented by the Markov chain

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}), \tag{2.5}$$

where $q(\mathbf{x}_{1:T}|x_0)$ states that q is repeatably applied from time step 1 to T, given the initial data point $\mathbf{x}_0$.
To obtain $\mathbf{x}_t$, one has to sample from the Gaussian distribution described in (2.4), using the re-parametrization trick in (2.3). Sampling $\mathbf{x}_t$ from the distribution can be expressed as

$$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\boldsymbol{\epsilon}, \tag{2.6}$$

with $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$. (2.6) describes a step-by-step process where the noise is gradually added to the initial sample $\mathbf{x}_0$ over time. However, this approach can be inefficient, particularly when the final time step T is large. To address this, the noise schedule β_t is reformulated, enabling the process to be expressed in terms of the initial time step $t = 0$. This allows for direct sampling of $\mathbf{x}_t$ at any arbitrary time step t from the original sample $\mathbf{x}_0$ [HJA20], i.e.,

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\,\mathbf{I}\right), \tag{2.7}$$

with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$ and $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$. Using the re-parametrization trick again described in (2.3), the noise sample $\mathbf{x}_t$ can be directly computed from initial sample $\mathbf{x}_0$ using

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}. \tag{2.8}$$

Following the (2.8), the noisy sample x_t can be derived by adding random noise $\boldsymbol{\epsilon}$ sampled from the Gaussian distribution to the initial sample $\mathbf{x}_0$. The amount of noise added is scaled by $\sqrt{1-\bar{\alpha}_t}$ and the extent of the original sample still present is scaled by the factor $\sqrt{\bar{\alpha}_t}$.

Reverse Process The reverse process involves denoising a corrupted sample $\mathbf{x}_t$, using the model's predictions to identify and subtract the noise at each step.
During training, the model has learned to reverse of the forward process $q(\mathbf{x}_t|\mathbf{x}_{t-1})$, by estimating $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$, also called posterior distribution. The approximated posterior distribution is denoted here as $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ with θ being the parameters of the denoising network.
More specifically, the denoising model learns to predict the mean $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t)$ and variance $\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)$ for each reverse diffusion step, i.e.,

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)). \tag{2.9}$$

The distribution $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is modeled as Gaussian for simple optimization, where the variance $\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)$ is isotropic and diagonal, i.e.,

$$\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t) = \sigma_t^2 \mathbf{I}. \tag{2.10}$$

Similarly to the forward process, the reverse process can be written as a Markov chain, i.e.,

$$p_\theta(\mathbf{x}_{T:0}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t). \tag{2.11}$$

As shown in (2.11), the reverse diffusion process begins at time step T, which corresponds to the final step of the forward diffusion, where the original data sample $\mathbf{x}_0$ has been transformed into pure noise, i.e.,

$$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I}), \tag{2.12}$$

with

$$p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; 0, \mathbf{I}). \tag{2.13}$$

Loss Function The marginal likelihood $p_\theta(\mathbf{x}_0)$ encapsulates how well the entire generative process, from forward to reverse diffusion, models the training data. It describes the probability of observing $\mathbf{x}_0$, a sample from the dataset, given the parameters θ. By considering all possible pathways the model can follow to derive from pure noise $\mathbf{x}_T$ the reconstructed data sample $\mathbf{x}_0$, the marginal likelihood can be computed as

$$p_\theta(\mathbf{x}_0) = \int p_\theta(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T}. \tag{2.14}$$

During training, the denoising model attempts to learn the distribution of the training dataset, optimizing its parameters θ.
A natural choice for the loss function would be minimizing the negative log-likelihood, which is defined as

$$-\log p_\theta(\mathbf{x}_0), \tag{2.15}$$

which would allow the model to indirectly learn how to reverse the forward diffusion process. As stated in (2.14), computing $p_\theta(\mathbf{x}_0)$ would involve integrating over all possible trajectories to receive the reconstructed data sample $\mathbf{x}_0$ from a noisy sample $\mathbf{x}_T$, which is intractable [SL23]. Instead, the evidence lower bound (ELBO) is employed, to derive a computable formula. The idea behind ELBO is to create a lower limit on how well a model can explain a set of observed data points. This limit helps evaluate the model's performance and is given by

$$\log p(\mathbf{x}) = \log \int p\left(\mathbf{x}_{0:T}\right) d\mathbf{x}_{1:T} \tag{2.16}$$

$$= \log \int \frac{p\left(\mathbf{x}_{0:T}\right) q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)} d\mathbf{x}_{1:T} \tag{2.17}$$

$$= \log \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\frac{p\left(\mathbf{x}_{0:T}\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\right] \tag{2.18}$$

$$\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p\left(\mathbf{x}_{0:T}\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\right] \tag{2.19}$$

where in the last step Jensen's inequality is applied. After some simplification, the loss function can be rewritten as

$$\begin{aligned} \mathbb{E}\left[-\log p_\theta\left(\mathbf{x}_0\right)\right] \leq \mathbb{E}_q[& \underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{x}_T \mid \mathbf{x}_0\right) \| \, p\left(\mathbf{x}_T\right)\right)}_{L_T} \\ & + \sum_{t>1} \underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) \| \, p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)\right)}_{L_{t-1}} \\ & \underbrace{-\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}_{L_0}]. \end{aligned} \tag{2.20}$$

The loss terms directly compare the outcomes of the forward diffusion steps of q and the reverse steps predicted by p_θ, utilizing the Kullback-Leibler (KL) divergence (except for L_0) [HJA20]. The Kullback-Leibler divergence $D_{\mathrm{KL}}(q||p)$ is defined as

$$D_{\mathrm{KL}}(q||p) = \int \log \frac{q(\mathbf{x})}{p(\mathbf{x})} d\mathbf{x} \tag{2.21}$$

and measures how much a probability distribution p differs from the true probability distribution q. The loss terms of (2.20) represent the following:

- L_T: measures KL divergence between $q(\mathbf{x}_T|\mathbf{x}_0)$ and $p(\mathbf{x}_T)$. It reflects how close $\mathbf{x}_T$ is to the Gaussian noise. This component of the loss is removed in the official paper [HJA20], since L_T has no trainable parameters.
- L_0: reflects how closely the model can reconstruct the original input data. In the original paper, this term is learned by a separate decoder [HJA20].

- L_{t-1}: measures the accuracy of the estimated denoised sample by calculating the difference between the desired denoising steps $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ and the estimated ones $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$. This represents the core objective of the denoising model.

Considering the KL divergence between two distributions p and q, where the variances are diagonal and equal, i.e., $\mathbf{\Sigma}_p = \mathbf{\Sigma}_q = \lambda\mathbf{I}$, the KL divergence simplifies to

$$D_{\mathrm{KL}}(q||p) = \frac{1}{2}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)^\intercal \mathbf{\Sigma}_p^{-1}\left(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q\right). \tag{2.22}$$

This allows to transform the loss term L_{t-1} of (2.20) to

$$L_{t-1} = \mathbb{E}_{\mathbf{x}_0,\epsilon}\left[\frac{1}{2\left\|\mathbf{\Sigma}_\theta\left(\mathbf{x}_t, t\right)\right\|_2^2}\left\|\tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t, \mathbf{x}_0\right) - \boldsymbol{\mu}_\theta\left(\mathbf{x}_t, t\right)\right\|^2\right], \tag{2.23}$$

where $\tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t, \mathbf{x}_0\right)$ is the mean of the conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ and $\boldsymbol{\mu}_\theta\left(\mathbf{x}_t, t\right)$ is the predicted mean of reverse diffusion step defined in (2.9).
The conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ can be described as a reverse diffusion step towards $\mathbf{x}_{t-1}$, if $\mathbf{x}_0$ is known as a reference. The reverse conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is intractable, however, conditioning the distribution on $\mathbf{x}_0$, i.e. $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$, makes it computable. It is defined by

$$q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t, \mathbf{x}_0\right), {\tilde{\sigma}_t}^2\mathbf{I}) \tag{2.24}$$

with

$$\tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t, \mathbf{x}_0\right) = \frac{\sqrt{\alpha_t}\left(\mathbf{1} - \bar{\alpha}_{\mathrm{t-1}}\right)}{1 - \bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1 - \bar{\alpha}_t}\mathbf{x}_0 \tag{2.25}$$

and

$$\tilde{\sigma}_t = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}\beta_t. \tag{2.26}$$

Replacing $\mathbf{x}_0$ with $\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}_t)$ from (2.8), the mean $\tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t\right)$ now only depends on $\mathbf{x}_t$ with

$$\tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t\right) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}}\boldsymbol{\epsilon}\right). \tag{2.27}$$

If the denoising network $\boldsymbol{\epsilon}_\theta$ is now trained to approximate $\boldsymbol{\epsilon}$ and consequently the mean $\tilde{\boldsymbol{\mu}}_t$, it can be rewritten as

$$\tilde{\boldsymbol{\mu}}_\theta\left(\mathbf{x}_t, t\right) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t, t\right)\right). \tag{2.28}$$

Inserting (2.28) and (2.27) to the loss term L_{t-1} defined in (2.23) and after further simplifications, the loss function can be re-parametrized to the estimated noise $\boldsymbol{\epsilon}_\theta$, which is also the final representation of the loss function, i.e.,

$$L_t := \mathbb{E}_{x_0,\boldsymbol{\epsilon}}\left[\frac{\beta_t^2}{2\|\boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t)\|_2^2\alpha_t\left(1-\bar{\alpha}_t\right)}\left\|\boldsymbol{\epsilon}-\boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t,t\right)\right\|^2\right], \tag{2.29}$$

with $\mathbf{x}_t$ being the corrupted sample at timestep t according to the forward diffusion process

$$\boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t,t\right) := \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \tag{2.30}$$

and $\boldsymbol{\epsilon}$ representing the current noise present in sample $\mathbf{x}_t$. The actual noise is compared to the predicted noise of the denoising model, described as $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)$.
In the original paper [HJA20], it was suggested to ignore the weighting term and therefore fix the variance to a constant $\boldsymbol{\Sigma}_\theta = (\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t)\mathbf{I}$. This simplifies to a mean squared error between the noise added during the forward process and the predicted noise from the model, i.e,

$$L := \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\|\boldsymbol{\epsilon}-\boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t,t\right)\|^2, \tag{2.31}$$

with

$$\boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t,t\right) := \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}. \tag{2.32}$$

Subsequent empirical studies revealed that predicting the variance enables sampling with fewer steps [ND21]. The loss function is updated enabling the model to predict the variance, defined as

$$L_{var} = L + \lambda\sum_{t=0}^{T} L_t, \tag{2.33}$$

where L is defined in (2.31) and L_t is defined in (2.29) with a regularization hyperparameter λ [DN21]. The variance $\boldsymbol{\Sigma}_\theta$ is not predicted directly from the model, instead, the model predicts a vector $\mathbf{v}$ that contributes to the calculation of the variance, described as

$$\boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t) = \exp\left(\mathbf{v}\log\beta_t + (\mathbf{1}-\mathbf{v})\log\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t\right), \tag{2.34}$$

where exp is the exponential operator.

Training The generative model is trained on noisy input samples, predicting the expected noise $\boldsymbol{\epsilon}_\theta$ contained in the corrupted samples, denoted as $\boldsymbol{\epsilon}$. During each training step, the model takes a clear input sample $\mathbf{x}_0$ from the training dataset and introduces noise to the input according to the forward process

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon} \tag{2.35}$$

with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$ and $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$. The current time step t defines the amount of noise returned by the noise scheduler β_t, where a larger t indicates more noise and smaller t represents less noise. During training, the current time step is sampled randomly from a range of $[0, T]$ with time step T as the final time step.
If the model predicts only the noise and the variance is fixed to $\boldsymbol{\Sigma}_\theta = \left(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t\right)\mathbf{I}$, the model computes the mean squared error between the actual noise $\boldsymbol{\epsilon}$ present in the corrupted input sample and the predicted noise $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ from the model. The training process is described in Algorithm 2.1. The noise scheduler β is predefined to one of

Algorithm 2.1: DDPM Training

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ ▷ Sample x_0 from training dataset
3: $t \sim \text{Uniform}(\{1, \dots, T\})$ ▷ Sample random timestep t
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ ▷ Draw new noise vector $\boldsymbol{\epsilon}$
5: $x_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ Corrupt data with Gaussian noise
6: $\boldsymbol{\epsilon}_\theta = \text{DDPM\_model}(\mathbf{x}_t, t)$ ▷ Predict noise ϵ_θ
7: Take gradient descent step on $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right\|^2$
8: **until** converged

the various options outlined in Section 2.1.4, where the schedule at a given time step t is defined as β_t. The values of the noise scheduler for each possible time step t are calculated before training.

Sampling (Inference) Once the model has been trained on the parameters θ to approximate the true reverse process $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ as closely as possible with the estimate $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$, it can be used to generate new images. The posterior distribution described by the model, $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is given by

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)). \tag{2.36}$$

The original paper [HJA20] chose to fix the variance to an untrained time-dependent constant that is

$$\boldsymbol{\Sigma}_\theta = \sigma_t^2 \mathbf{I}, \tag{2.37}$$

where

$$\sigma_t = \sqrt{\frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}\beta_t}. \tag{2.38}$$

Therefore the approximated posterior distribution can be rewritten as

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}\Big(x_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \sigma_t^2 \mathbf{I}\Big), \tag{2.39}$$

with

$$\boldsymbol{\mu}_\theta\left(\mathbf{x}_t, t\right) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t, t\right)\right). \tag{2.40}$$

Using the re-parametrization trick in (2.3), it is possible to directly sample from the posterior distribution estimate $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$, i.e.,

$$\mathbf{x}_{t-1} = \boldsymbol{\mu}_\theta(\mathbf{x}_t, t) + \sigma_t \boldsymbol{\epsilon}, \tag{2.41}$$

so that

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon}. \tag{2.42}$$

The sampling starts with $x_T \sim \mathcal{N}(0, \mathbf{I})$. The generative model iteratively removes the noise contained in the corrupted sample to retrieve the reconstructed data x_0. During sampling, the model estimates the entire noise present in the sample x_t to retrieve x_0. The predicted noise $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ is afterward removed from the noisy sample $\mathbf{x}_t$. But instead of removing all the noise at once, the model eliminates only a fraction of the predicted noise from the noisy sample. As the time step t approaches 0, progressively larger portions of the predicted noise are eliminated, with the entire predicted noise being removed at the final time step 0.
The sampling process can be described according to Algorithm 2.2.

Algorithm 2.2: DDPM Sampling

Input: noise scheduler β with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, $\sigma_t = \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t}$
Output: reconstructed sample x_0

$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = T, \ldots, 1$ do **do**
  if $t > 1$ **then**
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ ▷ Draw new noise vector z
  else
    $\mathbf{z} = 0$ ▷ Don't add noise at the last step
  end if
  $\boldsymbol{\epsilon}_\theta$ = DDPM_model$(\mathbf{x}_t, t)$ ▷ Predict noise $\boldsymbol{\epsilon}_\theta$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta\right) + \sigma_t \mathbf{z}$ ▷ Remove predicted noise$\boldsymbol{\epsilon}_\theta$
end for
return x_0

Continuous Formulation: Score SDE Formulation

Forward Process To understand the continuous formulation of the forward process, one has to first consider the sampling from the transition kernel from the DDPM forward

diffusion process, which – before using re-parametrization – is described as

$$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\mathcal{N}(0, \mathbf{I}). \tag{2.43}$$

The noise scheduler β can be interpreted using step size Δt, and therefore the sampling process is rephrased as

$$\mathbf{x}_t = \sqrt{1-\beta(t)\Delta t}\mathbf{x}_{t-1} + \sqrt{\beta(t)\Delta t}\mathcal{N}(0, \mathbf{I}), \tag{2.44}$$

where $\beta_t = \beta(t)\Delta t$ and Δt is the step size, with $\beta(t)$ as a function that adjusts the step size at time t.
If the step size is negligibly small, the first term of the sampling process in (2.44) can be rewritten using a Taylor expansion, i.e.,

$$\mathbf{x}_t \approx \mathbf{x}_{t-1} - \frac{\beta(t)\Delta t}{2}\mathbf{x}_{t-1} + \sqrt{\beta(t)\Delta t}\mathcal{N}(0, \mathbf{I}). \tag{2.45}$$

The sampling process formulated in (2.45) can be interpreted as an iterative update that corresponds to a certain discretization of a stochastic differential equation (SDE), describing diffusion in the infinitesimal limit, i.e.,

$$d\mathbf{x}_t = -\frac{1}{2}\beta(t)\mathbf{x}_t dt + \sqrt{\beta(t)}dw_t, \tag{2.46}$$

which can be more generally described as

$$d\mathbf{x}_t = f(\mathbf{x}, t)dt + g(t)dw, \tag{2.47}$$

where dw_t is the standard Wiener process based on Brownian motion, $f(\mathbf{x}, t)$ is a scalar function formulating the drift coefficient, determining the rate at which the process $d\mathbf{x}$ evolves on average, and $g(\mathbf{x}, t)$ is called the diffusion coefficient, defining how much noise is added over time.
By setting $f(\mathbf{x}, t) = -\frac{1}{2}\beta(t)\mathbf{x}_t$ and $g(t) = \sqrt{\beta(t)}$, the DDPM framework is described in continuous time, namely Variation Preserving SDE (VP SDE). As a result, DDPM is a special case of diffusion model with SDE, where (2.47) describes the general case. A Variation Explosion (VE) SDE has also been proposed; however, it will not be discussed in this context.
Considering the additive terms of the update $d\mathbf{x}_t$ on $\mathbf{x}_t$ over time individually, it becomes apparent that the first term describes a negative update direction towards state $\mathbf{x}_t$, pulling its contribution towards zero. At the same time, the second term $\sqrt{\beta(t)}dw_t$ gradually adds noise over time. Intuitively, by repeating this update process multiple times, $\mathbf{x}_t$ ultimately becomes pure noise, which is the intended outcome of the forward diffusion process.

Reverse Process Given a forward stochastic differential equation, one can describe the reverse of that SDE. [And82] The reverse generative diffusion SDE can be described as

$$d\mathbf{x} = \Big(f(\mathbf{x}, t) - g(t)^2\nabla_x \log p(\mathbf{x})\Big)\, dt + g(t)d\bar{w}. \tag{2.48}$$

In the context of a Variation Preserving SDE, this is defined as

$$d\mathbf{x}_t = \overbrace{\left(-\frac{1}{2}\beta(t)\mathbf{x}_t - \beta(t)\underbrace{\nabla_{\mathbf{x}_t}\log q_t(\mathbf{x}_t)}_{\text{“Score Function”}}\right)}^{\text{drift term}} dt + \overbrace{\sqrt{\beta(t)}d\bar{w}_t}^{\text{diffusion term}}, \tag{2.49}$$

where $d\bar{w}_t$ is a standard Wiener process with time flowing backward from T to 0. The score function $\nabla_{\mathbf{x}_t}\log q_t(\mathbf{x}_t)$ depends on the distribution of the training data $\mathbf{x}$ at time t. But the distribution of the training data $q(\mathbf{x})$ is not known. The idea is now to approximate the distribution with a denoising model p_θ, where θ are the network's parameters.

Loss Function The denoising model is trained to fit the score function $\nabla_{\mathbf{x}_t}\log q_t(\mathbf{x}_t)$, which can be described as the minimization problem

$$\min_{\boldsymbol{\theta}} \underbrace{\mathbb{E}_{t\sim\mathcal{U}(0,T)}}_{\substack{\text{diffusion}\\ \text{time } t}} \underbrace{\mathbb{E}_{\mathbf{x}_t\sim q_t(\mathbf{x}_t)}}_{\substack{\text{diffused}\\ \text{data } x_t}} \| \underbrace{\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)}_{\substack{\text{denoising}\\ \text{network}}} - \underbrace{\nabla_{\mathbf{x}_t}\log q_t(\mathbf{x}_t)}_{\substack{\text{score of}\\ \text{diffused data}}} \|_2^2. \tag{2.50}$$

The idea is to draw a diffusion time t from the uniform distribution in the range $[0, T]$, then sample diffused data at time step t and give this as input to the model $\mathbf{s}_{\boldsymbol{\theta}}$ with the time t. The model is trained to predict the gradient of the logarithm of the distribution $q_t(\mathbf{x}_t)$ of the data point $\mathbf{x}$ at time step t. The loss can be calculated utilizing a simple L^2 term between predicted score function $\mathbf{s}_{\boldsymbol{\theta}}$ and the log of the probability density function $\nabla_{\mathbf{x}_t}\log q_t(\mathbf{x}_t)$.
The problem of the above (2.50), as stated already, is that it is not tractable because there is no analytical expression of the distribution $q(\mathbf{x})$ of the dataset. If it were possible, there would be no need to train a model to approximate it.
Instead, denoising score matching is considered to train the model and describe the loss function. It utilizes the conditional density $q_t(\mathbf{x}_t|\mathbf{x}_0)$ given a particular data point $\mathbf{x}_0$ from the dataset instead of the full diffuse density $q_t(\mathbf{x}_t)$. The advantage of the conditional distribution is that $q_t(\mathbf{x}_t|\mathbf{x}_0)$ is tractable. The modified loss function can now be written as

$$\min_{\boldsymbol{\theta}} \underbrace{\mathbb{E}_{t\sim\mathcal{U}(0,T)}}_{\substack{\text{diffusion}\\ \text{time } t}} \underbrace{\mathbb{E}_{\mathbf{x}_0\sim q_0(\mathbf{x}_0)}}_{\substack{\text{data}\\ \text{sample } x_0}} \underbrace{\mathbb{E}_{\mathbf{x}_t\sim q_t(\mathbf{x}_t)}}_{\substack{\text{diffused data}\\ \text{sample } x_t}} \| \underbrace{\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)}_{\substack{\text{denoising}\\ \text{network}}} - \underbrace{\nabla_{\mathbf{x}_t}\log q_t(\mathbf{x}_t|\mathbf{x}_0)}_{\substack{\text{score of diffused}\\ \text{data sample}}} \|_2^2. \tag{2.51}$$

Now the network can be trained to approximate the score of a diffused individual data point $\mathbf{x}_0$.
Given a diffusion time t sampled from the uniform distribution of range $[0, T]$, the model

draws a particular sample $\mathbf{x}_0$ from the dataset and diffuses the data point $\mathbf{x}_0$ towards the corrupted variant $\mathbf{x}_t$ at time step t. The network takes the noised $\mathbf{x}_t$ as input, training the score of this one particular diffused data sample $\mathbf{x}_0$.
It turns out that by employing the loss function of (2.51), the model still learns to approximate the dataset distribution. This is intuitive because the model tries to predict the reconstructed data $\mathbf{x}_0$ from a given noisy sample $\mathbf{x}_t$. The noisy data could represent many possible values of the original data $\mathbf{x}_0$. The model essentially averages over the possible values, learning over time to approximate the score of the full diffused data distribution.
Using re-parametrization sampling with $\mathbf{x}_t = \gamma_t \mathbf{x}_0 + \sigma_t \boldsymbol{\epsilon}$, $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$, $\gamma_t = e^{-\frac{1}{2}\int_0^t \beta(s)ds}$ and $\sigma^2 = 1 - e^{-\frac{1}{2}\int_0^t \beta(s)ds}$ the score function simplifies to

$$\nabla_{\mathbf{x}_t} \log q_t\left(\mathbf{x}_t|\mathbf{x}_0\right) = -\frac{\boldsymbol{\epsilon}}{\sigma_t}. \tag{2.52}$$

Here, $\boldsymbol{\epsilon}$ represents the noise introduced during re-parametrized sampling and σ_t is the standard deviation at time step T. By parametrizing the denoising network with

$$\mathbf{s}_{\boldsymbol{\theta}} = -\frac{\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, T)}{\sigma_t}, \tag{2.53}$$

the loss function can be rewritten as

$$\min_\theta \mathbb{E}_{t\sim\mathcal{U}(0,T)} \mathbb{E}_{\mathbf{x}_0\sim\varphi_0(\mathbf{x}_0)} \mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(0,\mathbf{I})} \frac{1}{\sigma_t^2} \left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\left(\mathbf{x}_t, t\right)\right\|_2^2. \tag{2.54}$$

If the parametrization of the denoising network is chosen according to (2.53), the model is trained to predict the noise values $\boldsymbol{\epsilon}$ that were used to perturb the input data $\mathbf{x}_0$, equivalent to the training objective of the DDPM.
The advantage of the continuous structure of the Score SDE is that it also allows for conditional generation, introducing conditional information such as class labels or text prompts through the diffusion process, which is described in Section 2.1.4.

Sampling (Inference) Due to the decoupling from training, there exist multiple inference methods. Since the implementation will focus on DDPM models, this section will not go into detail about the different sampling methods. The following are three popular sampling methods, which try to numerically solve the reverse-time SDE:

- higher-order adaptive step size SDE solver,
- Euler-Maruyama,
- ancestral sampling.

2.1.4 Noise Schedule

The noise schedule β_t defines the amount of noise added to the input during the forward diffusion process and removed during reverse computation at time step t. It is a critical component of diffusion models, as too rapid noise addition and removal can result in premature information loss, preventing the model from ever converging, while an excessively slow schedule can lead to unnecessarily long computation times. The following paragraphs describe the linear schedule, cosine schedule, sigmoid schedule, exponential schedule, and guidance schedule.

Linear Schedule The Linear Schedule adds or removes noise at a constant rate, linearly increasing during the forward process and linearly decreasing during the reverse process. It can be described with

$$\beta_t = \beta_{\text{end}} + (\beta_{\text{start}} - \beta_{\text{end}}) \left(\frac{t}{T} \right), \tag{2.55}$$

where β_{start} is the initial noise level, β_{end} is the final noise level, t is the current time steps, T is the total numbers of time steps.

Cosine Schedule The cosine schedule ensures a smoother transition between noise levels. During the forward process, it introduces smaller noise at the beginning and increases the noise more rapidly toward the end. It is defined by

$$\beta_t = \beta_{\text{end}} + 0.5(\beta_{\text{start}} - \beta_{\text{end}}) \left(1 + \cos\left(\pi \frac{t}{T} \right) \right), \tag{2.56}$$

where β_t β_{start} is the initial noise level, β_{end} is the final noise level, t is the current time steps, T is the total numbers of time steps.

Sigmoid Schedule The sigmoid schedule introduces noise more gradually at the beginning and end of the forward diffusion process while taking larger steps in the middle. This characteristic follows the idea of preserving important features in the initial and later stages of diffusion modeling. It is described with

$$\beta_t = \beta_{\text{end}} + (\beta_{\text{start}} - \beta_{\text{end}}) \left(1 + e^{-k\left(\frac{t}{T} - \frac{1}{2}\right)} \right), \tag{2.57}$$

where β_t β_{start} is the initial noise level, β_{end} is the final noise level, t is the current time steps, T is the total numbers of time steps and k is a parameter.

Exponential Schedule The exponential schedule adds more noise at the beginning and gradually reduces the noise level in later stages. This facilitates more gradual refinement during the later stages while enabling faster extraction of fine details early on. It is defined by

$$\beta_t = \beta_{\text{start}} \left(\frac{\beta_{\text{end}}}{\beta_{\text{start}}} \right)^{\frac{t}{T}}, \tag{2.58}$$

where β_{start} is the initial noise level, β_{end} is the final noise level, t is the current time steps, T is the total numbers of time steps.

Guidance Mechanisms

Guidance mechanisms have been introduced to influence the output of generation models on conditions like class labels or text prompts.

Classifier Guidance Classifier guidance employs an additional classifier with trainable weights to control the generation process of a diffusion model. The conditioning input for this method consists of class labels. This entails that the influence of the classifier on the diffusion process is restricted to the categories the classifier has been trained on. To understand how the classifier is integrated into the diffusion model, one has to revisit the reverse process of the continuous formulation. It is formulated as a stochastic differential equation, i.e.,

$$d\mathbf{x} = \Big(f(\mathbf{x}, t) - g(t)^2 \nabla_x \log p_t(\mathbf{x})\Big)\, d\bar{t} + g(t) d\bar{w}, \tag{2.59}$$

where the model is trained to estimate $\nabla_x \log p(\mathbf{x})$, more specifically, it approximates $\nabla_x \log q(\mathbf{x}_t|\mathbf{x}_0)$ as described in (2.51).
To guide the diffusion model towards a given class label c, the model needs to predict the conditional score function $\nabla_x \log p(\mathbf{x}_t|c)$ instead. By applying Bayes' rule, the conditional score function can be decomposed into

$$p(\mathbf{x}_t|c) = \frac{p(c|\mathbf{x}_t)p(\mathbf{x}_t)}{p(c)}, \tag{2.60}$$

which can be simplified to

$$\log p(\mathbf{x}_t|c) = \log p(c|\mathbf{x}_t) + \log p(\mathbf{x}_t) - \log p(c), \tag{2.61}$$

$$\nabla_{x_t} \log p(\mathbf{x}_t|c) = \nabla_{x_t} \log p(c|\mathbf{x}_t) + \nabla_{x_t} \log p(\mathbf{x}_t), \tag{2.62}$$

with $\nabla_{\mathbf{x}_t} \log(c) = 0$ due to $\log(c) = \text{const}$.

By training a separate classifier to model $p_\phi(c|\mathbf{x}_t, t)$, which takes the corrupted data $\mathbf{x}_t$ and time step t as inputs and predicts the class label c, the conditional score function is described by

$$\nabla_{x_t} \log p_\theta(\mathbf{x}_t|c) = w \nabla_{x_t} \log p_\phi(c|\mathbf{x}_t) + \nabla_{\mathbf{x}_t} \log p_\theta(\mathbf{x}_t). \tag{2.63}$$

The primary drawback of this approach is that additional training of a classifier is needed. Moreover, the categories on which the diffusion model's generation can be conditioned are limited by the dataset used to train the classifier.

Classifier-Free Guidance Classifier-free guidance does not need an additional classifier as the name suggests but is based on the classifier guidance method. This method is employed in most of the latest generative diffusion models that utilize text as input because it allows for training without restrictions on the number of categories. Instead of

training a separate classifier, the model is trained to approximate both $p(\mathbf{x}_t|c)$ and $p(\mathbf{x}_t)$. In other words, the diffusion model is trained to represent both a conditional and an unconditional model. During training, the conditional input c is randomly removed and replaced with a special value representing an empty token. To express the conditioning term as a function of the conditional and unconditional score functions, the Bayes rule can be applied according to

$$p(\mathbf{x}_t|c) = \frac{p(\mathbf{x}_t|c)p(c)}{p(\mathbf{x}_t)}, \tag{2.64}$$

which simplifies to

$$\log p(c|\mathbf{x}_t) = \log p(\mathbf{x}_t|c) + \log p(c) - \log(\mathbf{x}_t), \tag{2.65}$$

and

$$\nabla_x \log p(c|\mathbf{x}_t) = \nabla_x \log p(\mathbf{x}_t|c) - \nabla_x \log p(\mathbf{x}_t). \tag{2.66}$$

With that, the conditional score function from the classifier guidance can be adapted according to

$$\nabla_x \log p_\theta(\mathbf{x}_t|c) = \nabla_x \log p(\mathbf{x}_t) + \gamma(\nabla_x \log p(\mathbf{x}_t|c) - \nabla_x \log p(\mathbf{x}_t)). \tag{2.67}$$

Training DDPM with Classifier-Free Guidance To allow classifier-free guidance, the original training of the DDPM can be rewritten as Algorithm 2.3.

Algorithm 2.3: DDPM Training – Classifier-Free Guidance

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, probability of unconditional training p_{uncond}

1: **repeat**
2: $\mathbf{x}_0, c \sim q(\mathbf{x}_0, c)$ ▷ Sample x_0 and condition c from training dataset
3: $c \leftarrow \varnothing$ with probability p_{uncond} ▷ Randomly discard conditioning
4: $t \sim \text{Uniform}(\{1, \dots, T\})$
5: $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
6: $x_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ Corrupt data with Gaussian noise
7: $\boldsymbol{\epsilon}_\theta = \text{DDPM\_model}(\mathbf{x}_t, t, c)$
8: Take gradient descent step on $\nabla_\theta \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\|^2$
9: **until** converged

DDPM Sampling – Classifier-Free Guidance During Classifier-Free Guidance sampling, the predicted noise $\boldsymbol{\epsilon}_\theta(\mathbf{z}, t, c_{\text{cond}})$ based on conditional information c_{cond} and the noise predicted without additional guidance information $\boldsymbol{\epsilon}_\theta(\mathbf{z}, t, c_{\text{uncond}})$ are combined according to Algorithm 2.4.

Algorithm 2.4: Sampling – Classifier-Free Guidance

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, $\sigma_t = \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t}$, guidance strength w
Output: reconstructed sample x_0

$\mathbf{c}_{\text{cond}} \sim q(\mathbf{x}_0, c)$
$\mathbf{c}_{\text{uncond}} \leftarrow \varnothing$
$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = T, \ldots, 1$ do **do**
  if $t > 1$ **then**
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ ▷ Draw new noise vector z
  else
    $\mathbf{z} = 0$ ▷ Don't add noise at the last step
  end if
  $\boldsymbol{\epsilon}_{\theta_{\text{cond}}} = \text{DDPM\_model}(\mathbf{x}_t, t, c_{\text{cond}})$ ▷ Sample with condtion
  $\boldsymbol{\epsilon}_{\theta_{\text{uncond}}} = \text{DDPM\_model}(\mathbf{x}_t, t, c_{\text{uncond}})$ ▷ Sample without condtion
  $\boldsymbol{\epsilon}_t = (1 + w)\boldsymbol{\epsilon}_{\theta_{\text{cond}}} - w\boldsymbol{\epsilon}_{\theta_{\text{uncond}}}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right) + \sigma_t \mathbf{z}$
end for
return x_0

2.2 Architecture

2.2.1 U-Net

The U-Net [RFB15] is a convolutional neural network and was proposed by the paper Denoising Diffusion Probabilistic Models (DDPM) [HJA20] as the backbone architecture for diffusion models. Originally designed for image data, it still is a common network used for computer vision tasks because of its ability to extract local context and texture details.
The architecture is divided into an encoder and a decoder. The encoder's task is to compress the input image while the decoder reconstructs the image from the compressed information. Along the contracting path, each layer of the U-Net shrinks the input image by removing raw information. The decoder now faces the challenge of expanding the compressed data back to the same size as the original input image. By using skip connections between the contracting and expanding path, the decoder receives additional context from the encoder that should help during the decoding process.
Four attention blocks are used to incorporate conditional data like text information into the generation process. The architecture is adapted from the standard Transformer decoder architecture. Figure 2.1 visualizes the architectural details of the U-Net model.

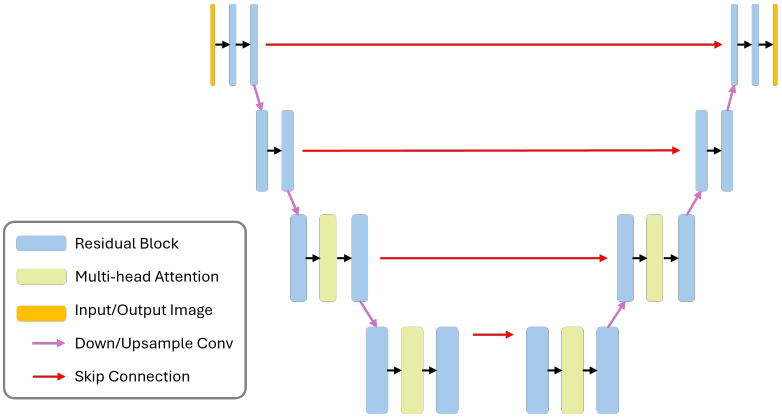


Figure 2.1: U-Net architecture

2.2.2 Vision Transformer

Revisiting the Transformer Network

The Transformer model [VSP+23] is a deep learning architecture originally designed for natural language processing (NLP) tasks. It is known for its ability to capture long-range dependencies, it has since been adapted for vision tasks as the Vision Transformer (ViT) [DBK+20]. To understand the principles underlying the Vision Transformer, the components of the original transformer are discussed in the following paragraphs. Figure 2.2 visualizes the architecture of the Transformer model. The architecture is built on an encoder-decoder structure, where each part includes multi-head attention mechanisms, feed-forward neural networks, and layer normalization.

Self-Attention Mechanism The attention operation is a method to analyze the relationship of tokens in a given input sequence and helps the transformer to focus on the relevant information. Given an input of shape

$$\mathbf{X} \in \mathbb{R}^{\text{batch} \times \text{tokens} \times d_{\text{model}}}, \tag{2.68}$$

with trainable weight matrices

$$\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{k}}}, \tag{2.69}$$

where

- d_{model} refers to the dimensionality of the embedding vector for each element in the input sequence,
- d_{k} represents the inner dimension specific to each self-attention layer,
- batch is the batch size,

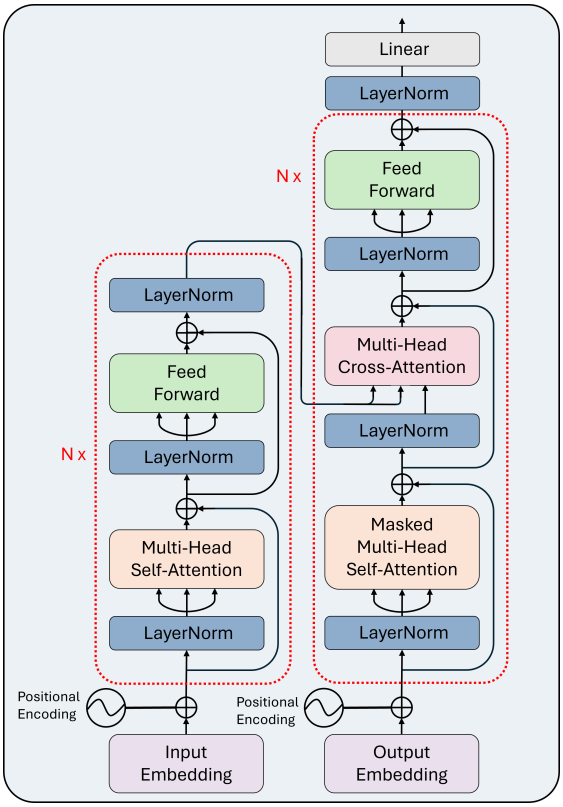


Figure 2.2: Transformer model

- tokens represents the number of elements in the sequence,

the self-attention mechanism is implemented utilizing the "scaled dot product attention" which is defined as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_\text{k}}}\right)\mathbf{V}. \tag{2.70}$$

The attention calculation is based on the matrix representation of queries $\mathbf{Q}$, keys $\mathbf{K}$, and value $\mathbf{V}$. These matrices are derived from the input $\mathbf{X}$, i.e.,

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \tag{2.71}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}_K, \tag{2.72}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}_V, \tag{2.73}$$

where

$$\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{\text{batch} \times \text{tokens} \times d_\text{k}}. \tag{2.74}$$

The weight matrices $\mathbf{W}_K$, $\mathbf{W}_V$, $\mathbf{W}_Q$ are learned during training of the neural network. To understand the attention calculation in detail, the following explains the method step-by-step:

1 Calculation of the attention score, i.e., $\mathbf{QK}^\intercal$: The dot product calculation between queries and keys is a crucial step in the attention calculation, as it represents how much attention each token should place on another token of the corresponding input sequence. A high value of the dot product signifies greater focus. The weight matrices $\mathbf{W}_K$ and $\mathbf{W}_Q$ learn this relationship during training.

2 Scaled attention scores, i.e., $\frac{\mathbf{QK}^\intercal}{\sqrt{d_\text{k}}}$: To ensure more stable gradients, the magnitude of the attention score is reduced by the factor $\frac{1}{\sqrt{d_\text{k}}}$, where d_k is the dimension of the key matrix $\mathbf{K}$.

3 Applying softmax, i.e., softmax $\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_\text{k}}}\right)$: After applying the softmax function to the scaled attention scores, the values range from 0 to 1. The probability values emphasize the tokens that should receive more attention in comparison to others.

4 Computing the context vector, i.e., softmax $\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_\text{k}}}\right)\mathbf{V}$: The softmax calculation in step three returns probabilities for each token in the corresponding input sequence. Those probabilities are combined with the input $\mathbf{X}$ scaled by the weight matrix $\mathbf{W}_V$, where only those tokens are kept that received a high attention score.

Figure 2.3 visualizes the process of self-attention, given input $\mathbf{X}$.

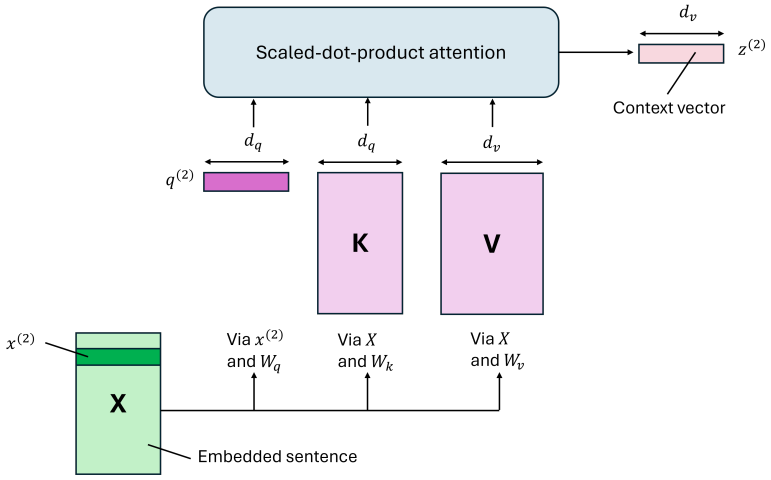


Figure 2.3: Self-attention mechanism

Cross Attention Mechanism The cross-attention computation is similar to the self-attention method. However, instead of learning the relationship between tokens within the same input sequence, cross-attention captures the relationship between two different inputs $\mathbf{X} \in \mathbb{R}^{\text{batch}\times\text{tokens}\times d_\text{model}}$ and $\mathbf{Y} \in \mathbb{R}^{\text{batch}\times\text{tokens}\times d_\text{model}}$. This can be simply

achieved by a key $\mathbf{K}$ and value $\mathbf{V}$ matrix representing the second input $\mathbf{Y}$, i.e.,

$$\mathbf{K} = \mathbf{Y}\mathbf{W}_K, \tag{2.75}$$

$$\mathbf{V} = \mathbf{Y}\mathbf{W}_V, \tag{2.76}$$

with matrix shapes according to 2.69. In the Transformer model, a cross-attention layer is used in the decoder block. The decoder block takes the output of the encoder block as input for the query matrix to be able to consider the output of the encoder when generating new sequences.

Multi-Head Attention The attention mechanism explained in Section 2.2.2 is computed multiple times during multi-head-attention. Each computation is called the attention head. By splitting the query matrix $\mathbf{Q}$, the key matrix $\mathbf{K}$, and the value matrix $\mathbf{V}$, each partition can afterward be independently passed through a separated head. After the attention calculation, the results of the heads are combined to produce the final attention score. Utilizing multiple heads allows the model to grasp multiple aspects of the input data, each head focusing on a different one. The multi-head attention calculation is visualized in Figure 2.4.

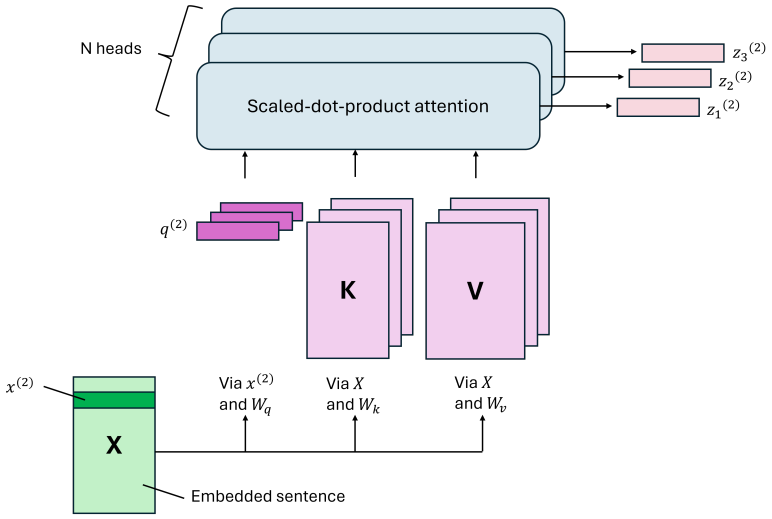


Figure 2.4: Multi-head attention

Feed-Forward network The feed-forward network is a sub-unit within the encoder and decoder of the Transformer model. It is placed after the self-attention calculation in the encoder block, as well as after the cross-attention layer of the decoder block. It introduces non-linearity to the model and can be interpreted as a refinement of the features extracted through attention calculation. During the forward pass of the input x through the feed-forward network, it undergoes the following steps:

1 Linear transformation: The input $\mathbf{X}$ is first passed through a linear module with learnable weight.

2 Activation function: The activation function introduces non-linearity, allowing the model to learn more complex and detailed patterns within the data.

3 Dropout: The dropout module prevents the model from over-fitting, by randomly ignoring some layer outputs during training.

4 Layer normalization: LayerNorm is applied at the final layer of the feedforward network to stabilize it by unifying the non-standard data into a specified format.

Residual Connection Between each multi-head attention and feedforward network layer a residual connection is added. Utilizing residual connections helps to combat the vanishing gradient problem, by connecting the input of each layer to its output as visualized in Figure 2.5.

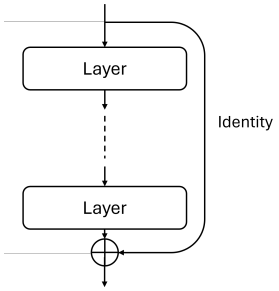


Figure 2.5: Residual connection concept

Layer Normalization Normalization is a preprocessing technique applied to the input data, transforming the features to a common scale. The original transformer architecture uses layer normalization [BKH16], which ensures that the intermediate layers have a consistent distribution. Each layer is handled independently, in other words, the mean and variance of the activations in each layer are computed separately. Given the input $\mathbf{X}$ according to 2.68, the layer normalization can be mathematically described as

$$\mathrm{LN}(\mathbf{X}) = \gamma \hat{\mathbf{x}}_i + \beta, \tag{2.77}$$

where $\mathbf{x}_i$ represent the elements in the last dimension of $\mathbf{X}$ and

$$\mu = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} \mathbf{x}_i, \tag{2.78}$$

$$\sigma^2 = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (\mathbf{x}_i - \mu)^2, \tag{2.79}$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{2.80}$$

where

- γ, β are the learnable parameters,
- ϵ is a small value for numerical stability.

During layer normalization, the activations of the previous layer of each example in a batch are normalized independently.

Positional Encoding Positional encoding gives the Transformer an understanding of each token's relative placement within the input sequence. The position of each word within a sentence defines the meaning of a statement. Rearranging the words can lead to a different interpretation of the message behind the sentence. Positional encoding incorporates the order of words, consequently allowing the Transformer to understand their context within the sentence.
Instead of integer positional encoding, the Transformer model employs sinusoidal positional encoding. The advantage of sinusoidal positional encoding is that the output of a sine and cosine function lies in $[-1, 1]$. When processing longer sequences, basic integer encoding would lead to too large values.

Algorithm 2.5: Positional Encoding

Procedure POSITIONAL ENCODING(x, L, d_{text})
for k=0 **to** L-1: **do**
  for i=0 **to** $\frac{d_{\text{model}}}{2}$ **do**
    $\mathbf{PE}_{(k,2i)} = \sin\left(\frac{k}{n^{(2i/d_{\text{model}})}}\right)$
    $\mathbf{PE}_{(k,2i+1)} = \cos\left(\frac{k}{n^{(2i/d_{\text{model}})}}\right)$
  end for
end for
End Procedure

With Algorithm 2.5, the positional encoding of a token in a sequence x of length L can be computed. The input sequence has the dimensionality d_{model}. For each position k in the sequence x, the positional vector is calculated, where i marks the current index in the positional vector. Essentially, for every two elements, set the even equal to $\mathbf{PE}(k, 2i)$ and the odd to $\mathbf{PE}(k, 2i + 1)$, until the encoding vector has the length of d_{model}.

Text Tokenizer In natural language processing, tokenization refers to the process of splitting the input text into smaller units, called tokens. These tokens can be individual characters or segments of words.

Text embedding Text embeddings transform text into numerical representations, enabling machine learning models to interpret and process the information effectively.
The embedder takes the segmented text from the tokenizer as input and returns a dense vector of real numbers. By placing words with a similar meaning close in the embedding

space, the machine learning model is able to understand the context between words. The numerical representation reduces the dimensionality of long sequences of text data and, therefore, achieves higher efficiency when processing large volumes of data.

CLIP embedding The CLIP embedding is used in many Computer vision tasks, as it effectively learns visual representations of natural language. The technique used in CLIP is called contrastive learning to calculate similarities between text and image. The idea behind contrastive learning is to train the model in such a way that the distance of two embeddings of the same class gets minimized while maximizing the distance between embeddings belonging to different classes. In CLIP, the two embeddings that are compared, are image embedding and text embedding. The idea is to find an optimal text description of a given image by increasing the similarity between a matching image and text embedding. The text encoder architecture employed in CLIP is a Transformer model, while the image encoder is a Vision Transformer (ViT). The advantage of using the CLIP text embedded in Vision Tasks, like text-to-video generation is that the embedded text lies in a similar embedding space as the corresponding image.

Vision Transformer for Image Data

The Vision Transformer is an adaptation of the Transformer for natural language processing. It has received significant recognition since its release. Most of the current models for image and video processing implement a refined version of the original Vision Transformer model. Studies have demonstrated that applying a Vision Transformer to tasks such as image segmentation produces promising results, where even U-Net-based architectures were outperformed [HXZ+24].
The Vision Transformer takes an image as input and divides it into square-shaped patches, which are later compressed into a lower-dimensional space. After arranging the patches next to each other, positional encoding is added. The sequence is afterward fed into the Vision Transformer. The architecture is analogous to the standard Transformer Encoder architecture, as can be seen in Figure 2.6.

Patch embedding An important component of the Vision Transformer is the patch embedding. It is used as a preprocessing step and divides the input image into patches. Usually, it is implemented using a convolution layer, which encodes each patch into vector space. The image patches are the equivalent of the word tokens consumed by the Transformer architecture in natural language processing. Figure 2.7 visualizes the concept. Given an input image of shape (h, w) and a patch embedding size of p as well as embedding dimension d, the image is encoded into a sequence of patches with the shape (s, d) where $s = \frac{hw}{p^2}$.

Adaptive Layer Normalization (adaLN) The adaptive layer normalization (adaLN) [GWY+22] can be described as

$$\text{adaLN}(x, c) = \gamma_c \cdot \text{LayerNorm}(x) + \beta_c. \tag{2.81}$$

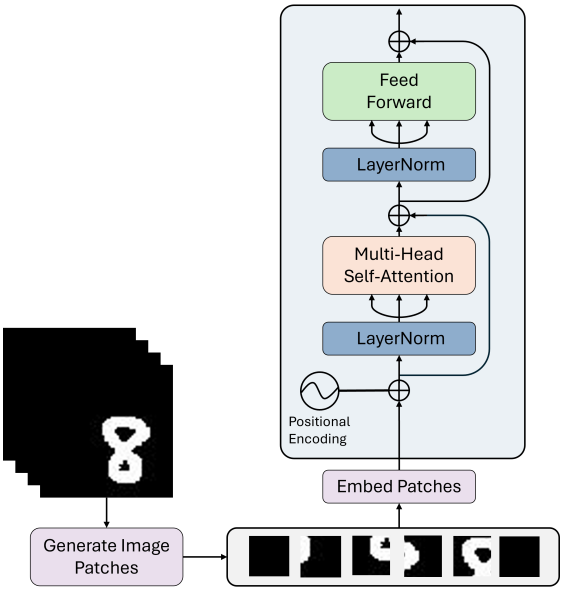


Figure 2.6: Vision Transformer architecture

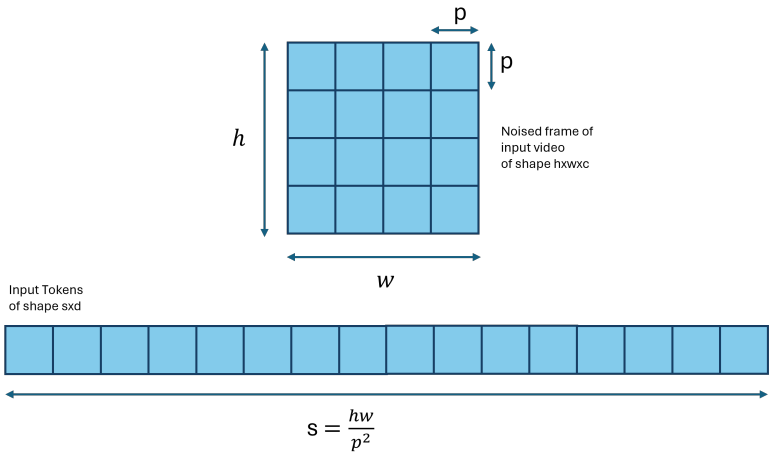


Figure 2.7: Patch embedding

It was designed to enable a Vision Transformer to learn across multiple domains. It involves adjusting the input x to align with a target task c during the LayerNorm calculation. The learnable parameters γ_c and β_c are derived through linear regression based on the condition c. By introducing γ_c and β_c, the conditioning embedding c is integrated into the feature channel. In text-to-video generation models, the input x represents the embedded video frames, while c corresponds to the embedded prompt combined with the diffusion time step embedding. Many recently released models use an adapted version of adaLN, known as scalable adaptive layer normalization (S-adaLN).

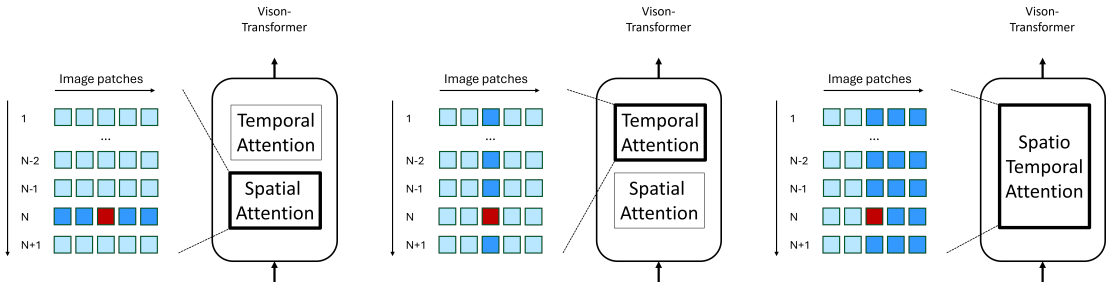
This is directly applied in the residual connection (RC) calculation, resulting in

$$\mathrm{RC}(x, c) = \alpha_c \cdot x + \mathrm{adaLN}(x, c). \tag{2.82}$$

Modeling of Space and Time Compared to image generation, video generation introduces additional complexity. It entails the challenge of modeling temporal consistency while handling additional computational overhead. To achieve temporal coherence, video generation models need to cover information across video frames. Three different attention methods will be examined in detail in this work. These include either separating the spatial and temporal attention into two distinct blocks or employing group-wise attention that allows spatial and temporal attention at the same time. The different methods can be described as:

- Spatial Attention: Each patch attends to patches within the same frame during spatial attention.
- Temporal Attention: During temporal attention, patches attend to patches from all frames within the video, but only patches in the same position are considered.
- Group-wise Spatio-Temporal Attention: In this approach, each patch attends only to a subset of patches within the same frame as well as to patches from preceding and subsequent frames within the same group.

Figure 2.8 visualizes the different space and time dimension handling of these methods:



(a) Spatial (b) Temporal (c) Spatio-Temporal

Figure 2.8: Spatial and temporal attention methods

In theory, it is also possible to implement Full Spatial-Temporal attention, where each patch attends to every other patch across all frames. While this approach ensures high spatial and temporal consistency, it is computationally expensive and cannot be used to generate long videos.

2.2.3 Latent Diffusion Models

Latent Diffusion Models have set new benchmarks in image and video generation by working in a lower spatial resolution rather than pixel space. Before feeding the input to the denoising model, the images are first encoded into latent space with the help of a Variational Autoencoder (VAE). As a result, the entire diffusion process is carried out in the latent space. Afterward, the predicted image is decoded into RGB. By operating in low dimensions, the computational costs are significantly smaller, which allows the generation of high-resolution images.

Variational Autoencoder

The Variational Autoencoder is based on the traditional autoencoder, where the input is passed through an encoder and decoder block. The encoder's role is to compress a large, complex dataset into the latent space. The decoder, on the other hand, is responsible for mapping the latent space back to the input space. In comparison to the traditional autoencoder, the Variational Autoencoder introduces regularization into latent space. By adding an extra loss term, the VAE ensures that the latent space follows a standard multivariate Gaussian distribution.
The loss function of an autoencoder is given by

$$L(x, x') = \|x - x'\|^2. \tag{2.83}$$

Whereas the loss function of a VAE can be mathematically described as

$$L(x, x', z) = \|x - x'\|^2 + D_{\mathrm{KL}}(q(z|x)\|p(z)). \tag{2.84}$$

Here, x is the input data and x' is the reconstructed input and z is the latent variable. The conditional likelihood distribution $q(z|x)$ is computed by the encoder, and $p(z)$ is the prior on the latent space – which is typically assumed to follow a Gaussian distribution. D_{KL} is the Kullback-Leibler divergence.
The input is translated into a latent distribution instead of a fixed tensor. This enables the Variational Autoencoder to generate new data that resembles the input data distribution.

2.3 Evaluation Metrics

The current evaluation metrics used for video quality testing can be categorized into image-level and video-level. The image-level metrics focus on the quality within each frame and the video-level try to evaluate both spatial and temporal aspects. The metrics SSIM and PSNR are content variant metrics and are employed to evaluate the quality of reconstructed video frames. Content variant metrics are applied when only one specific prediction for the images or video frames is accurate. For image and video generation, mostly content invariant metrics, like FID, LPIPS, and FVD, are out of interest. They measure consistency and overall similarity between generated videos and the ground truth instead of comparing the exact details.

2.3.1 Fréchet Video Distance (FVD)

Fréchet Video Distance is a video-level evaluation metric. The evaluation metric is based on the Fréchet Distance, also called 2-Wasserstein measuring how similar two distributions are. Given P_G as the distribution defined by the generative model and the corresponding mean μ_G and Σ_G, as well as the real-world distribution (in this case the dataset the generative model was trained on) P_R with μ_R and Σ_R, the Fréchet Distance is defined as

$$D^2_{\text{Fréchet}} = |\mu_R - \mu_G|^2 + \text{Tr}\left(\Sigma_R + \Sigma_G - 2\left(\Sigma_R\Sigma_G\right)^{\frac{1}{2}}\right). \tag{2.85}$$

The evaluation is based on the extracted features from pre-trained Inflated-3D Convnets (I3D) [CZ17].

Instead of directly comparing the distributions of the generative model and the dataset in raw image space, the FVD uses learned feature embeddings to calculate the distance. The feature embeddings are extracted from pre-trained Inflated-3D Convnets (I3D) [CZ17]. By feeding samples from the generative model and the dataset to the I3D model, their feature representations are recorded. The I3D model is trained on a video dataset for classification and considers temporal coherence, as well as the visual quality within frames. With the use of the I3D model, semantic information relevant to human perception can be better captured. For the calculation of the FVD, the activations of the last pooling layer – before the output classification of the videos – are summarized as multivariate Gaussian by calculating the mean and covariance. The similarities between the distribution of the generated videos and the actual videos represented in the dataset are computed using the extracted mean and covariance from the last pooling layer of the I3D model and calculating the distance according to Equation (2.85).
In summary, the FVD (Fréchet Video Distance) compares the activation distributions of the final deep layer of the I3D network. If these distributions are similar, it indicates that the underlying image distributions of the generated videos and the video dataset are alike. A low FVD value signifies a higher degree of similarity between the generated image and its corresponding image in the dataset.

2.3.2 Fréchet Inception Distance (FID)

Fréchet Video Distance (FVD) is based on the Fréchet inception distance (FID), which is an image generation metric. The main difference between the FVD and FID scores is the underlying model. FID uses the activation from the Inception V3 model pre-trained on an image classification dataset, to calculate the Fréchet Distance, described in Equation (2.85). A small FID value represents a high similarity between the generated image and the corresponding image from the dataset.

2.3.3 Structural Similarity Index (SSIM)

The structural Similarity Index is utilized to measure the similarity between frames. The SSIM index measure between a frame $\mathbf{Y} \in \mathbb{R}^{\text{height}\times\text{width}\times\text{channels}}$ from a generated video

and a frame $\mathbf{X} \in \mathbb{R}^{\text{height}\times\text{width}\times\text{channels}}$ from the dataset can be calculated as

$$\text{SSIM}(\mathbf{X}, \mathbf{Y}) = l(\mathbf{X}, \mathbf{Y})c(\mathbf{X}, \mathbf{Y})s(\mathbf{X}, \mathbf{Y}), \tag{2.86}$$

where the luminance l, the contrast c and the structures s are compared utilizing the mean μ_X, μ_Y and variance σ_X, σ_Y as well as covariance σ_{XY} of generated video frame and dataset video frame, i.e.,

$$l(\mathbf{X}, \mathbf{Y}) = \frac{2\mu_X\mu_Y + c_1}{\mu_X^2 + \mu_Y^2 + c_1}, \tag{2.87}$$

$$c(\mathbf{X}, \mathbf{Y}) = \frac{2\sigma_X\sigma_Y + c_2}{\sigma_X^2 + \sigma_Y^2 + c_2}, \tag{2.88}$$

$$s(\mathbf{X}, \mathbf{Y}) = \frac{\sigma_{XY} + c_3}{\sigma_X\sigma_Y + c_3}, \tag{2.89}$$

with

- μ_X the mean over the pixel values in $\mathbf{X}$,
- μ_Y the mean over the pixel values in $\mathbf{Y}$,
- σ_X^2 the variance of $\mathbf{X}$,
- σ_Y^2 the variance of $\mathbf{Y}$,
- σ_{XY} the covariance of $\mathbf{X}$ and $\mathbf{Y}$,
- $c_1 = (k_1 L)^2, c_2 = (k_2 L)^2$, two variables to stabilize the division,
- L the range of the pixel-values,
- $k_1 = 0.01$ and $k_2 = 0.03$ by default.

To conclude, SSIM assesses pixel intensities by comparing brightness, dynamic range, and spatial distribution. A higher SSIM value indicates greater similarity between the generated and original images.

2.3.4 Peak Signal-To-Noise (PSNR)

The peak signal-to-noise metric is an image-level metric and is based on the mean squared error between the original $\mathbf{X} \in \mathbb{R}^{\text{height}\times\text{width}\times\text{channels}}$ and generated video frame $\mathbf{Y} \in \mathbb{R}^{height\times width\times channels}$. It measures the absolute difference between the original and generated video, expressed in terms of a logarithmic decibel scale. The PSNR metric is defined as

$$\text{PSNR} = 20\log_{10}\left(\frac{\text{MAX}_f}{\sqrt{\text{MSE}}}\right), \tag{2.90}$$

with

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{X}_i - \mathbf{Y}_i\|^2 , \tag{2.91}$$

where

- $\mathbf{X}_i$ and $\mathbf{Y}_i$ represent the pixel value of image $\mathbf{X}$ and image $\mathbf{Y}$ at position i,
- N is the total number of pixels where $N = hwc$ with h being the height, w the width and c the number of channels of the images $\mathbf{X}$ and $\mathbf{Y}$,
- MAX_f is the maximum signal value of the original image $\mathbf{X}$.

The higher the PSNR, the better the predicted image $\mathbf{Y}$ resembles the original image $\mathbf{X}$.

2.3.5 LPIPS

LPIPS tries to simulate humans' ability to assess perceptual similarity by utilizing a model that is trained on a labeled dataset containing images judged on their similarity by human labeling. Akin to the FID and the FVD metric, the model compares the activations on the actual and predicted images. A low LPIPS value describes the high similarity between the generated image and the original image from the dataset.

2.3.6 CLIPSIM

CLIPSIM is a metric mostly used to evaluate the models' ability to generate images coherent to a given text prompt. It utilizes the CLIP model. CLIP is trained on image data with associated text descriptions, mapping them into the same embedding space. By encoding the given text prompt and the generated image with the CLIP model to embedding space, the cosine similarity can give an intuition of how well the generated image represents the text prompt.

To evaluate the results of image and video generation models, a common approach is to additionally average over the CLIPSIM results of the textual description and the actual visual representation from the dataset. Given a text description and the corresponding generated image, a high CLIPSIM value stands for an accurate visual representation of the given text prompt.

CHAPTER 3

Methodology

3.1 Image Generation

3.1.1 Class Label to Image Generation

The Diffusion Transformer (DiT) model [PX23] builds on the core principles of Denoising Diffusion Probabilistic Models (DDPMs) and employs a backbone architecture solely based on Transformers. The DiT model is trained on class-image pairs and integrates class information into the transformer backbone using adaLN.

Input Preprocessing

Image Preprocessing The input image of shape *(batch size, channels, height, width)* undergoes the following preprocessing steps.

- Patch embedding: First, the image is divided into smaller patches and projected linearly into an embedding space.
- Positional encoding: Sinusoidal position encoding is added to the flattened patches to provide spatial context for the model.

This results in an output with the shape *(batch size, sequence length, embed dim)*. The process is visualized in Figure 3.1.

Class Preprocessing Since the input classes are simple values ranging from 0 to 9, the class labels were mapped to the embedding space using a straightforward lookup table.

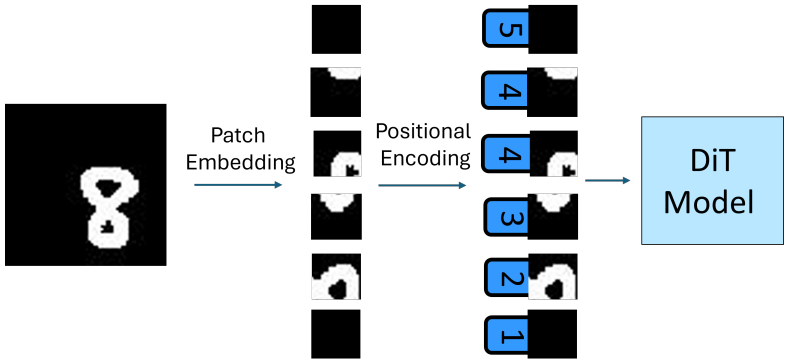


Figure 3.1: DiT preprocessing

Timestep Preprocessing The current time step in the diffusion process is introduced to the model using a time step embedding. It follows the same architectural design as the positional encoding.

DiT Block Design

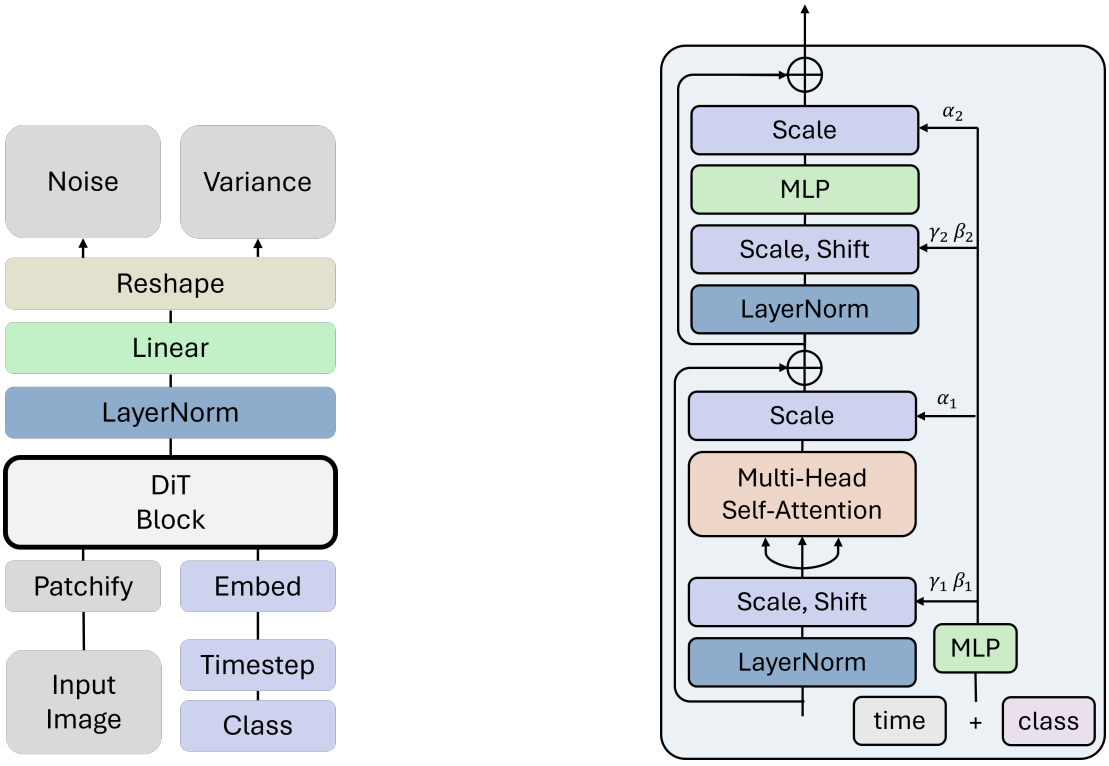
The DiT block takes the image patches, the class label, and the time step t as input. The time step t helps the model to be aware of the noise level that has been added to the input image. The architecture of the DiT block follows the ideas of the Transformer encoder from the ViT Transformer but adapts it accordingly to process conditional information such as the time step t and class labels c. To be able to generate images that correspond to the conditional input, the standard layer norm layers of the ViT Transformer are replaced with adaptive layer norm (adaLN). Instead of treating the weight and bias of the layer normalization as parameters, adaLN incorporates the concatenated input conditions c and t as normalization parameters for the image patches. Apart from this modification, the model follows the standard architecture of a Vision Transformer (ViT). The architectonic details of the Diffusion Transformer (DiT) are visualized in Figure 3.2.

3.1.2 Dataset

The Diffusion Transformer (DiT) was trained on the MNIST dataset. The MNIST dataset contains images of handwritten digits from zero to nine. Every image is labeled with its corresponding class. Some samples of the dataset are shown in Figure 3.3.

3.1.3 Training

The diffusion process was implemented from scratch using PyTorch, based on the official code of a Denoising Diffusion Probabilistic Model. The code for the architecture of the backbone model follows the architectural details outlined in the Diffusion Transformer paper [PX23].
For the image generation model, neither classifier-free guidance nor variance prediction



(a) DiT (b) DiT Block

Figure 3.2: DiT architectonic details

was used, following the basic implementation of DDPM models. The training is described in Algorithm 2.1.

3.1.4 Sampling

During sampling, an implementation detail known as thresholding was applied. In this process, the predicted sample $\mathbf{x}_0$ is clamped to a valid range, ensuring it falls within the fixed pixel range of $[0, 255]$. The predicted sample can be computed given the current sample $\mathbf{x}_t$ and the predicted noise $\boldsymbol{\epsilon}_t$, i.e.

$$\mathbf{x}_0 = \frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\theta}{\sqrt{\bar{\alpha}_t}}. \tag{3.1}$$

Once the computed sample $\mathbf{x}_0$ is obtained, its value range is clamped to $[0, 255]$. In the original DDPM sampling process, $\mathbf{x}_{t-1}$ is directly computed from the predicted noise $\boldsymbol{\epsilon}_\theta$. However, with thresholding, $\mathbf{x}_{t-1}$ is derived from the initial sample $\mathbf{x}_0$, i.e.

$$\mathbf{x}_{t-1} = \frac{\sqrt{\alpha_t}\,(1-\bar{\alpha}_{t-1})\,\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}\,(1-\alpha_t)\,\mathbf{x}_0}{1-\bar{\alpha}_t} + \sigma_t\boldsymbol{\epsilon}_t, \tag{3.2}$$

Figure 3.3: MNIST dataset

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(0, \mathbf{I})$.
To integrate thresholding, the sampling procedure can be rewritten as described in Algorithm 3.1.

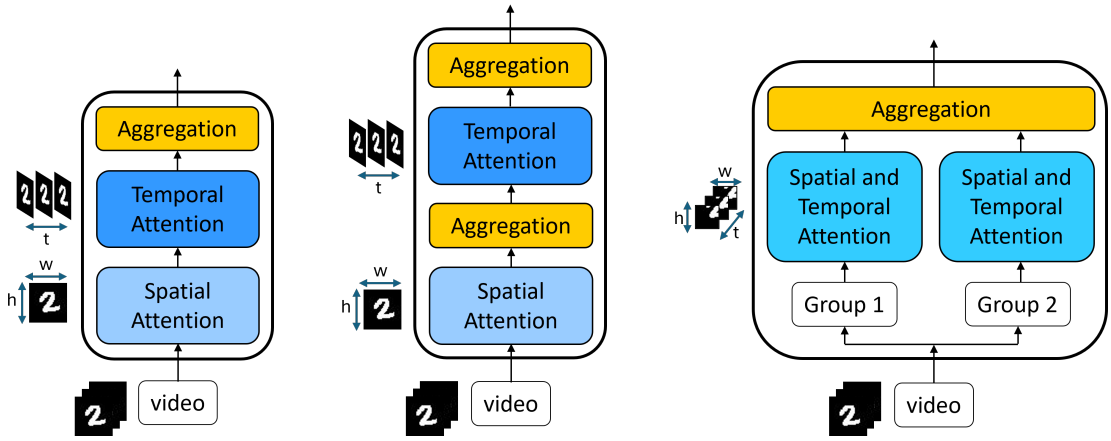
Algorithm 3.1: Sampling DDMP – Clamp x_0

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, $\sigma_t = \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}}\beta_t$
Output: reconstructed sample x_0

$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = T, \ldots, 1$ do **do**
  if $t > 1$ **then**
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ ▷ Draw new noise vector z
  else
    $\mathbf{z} = 0$ ▷ Don't add noise at the last step
  end if
  $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) = \text{DDPM\_model}(\mathbf{x}_t, t)$ ▷ Predict noise $\boldsymbol{\epsilon}_\theta$
  $x_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\theta}{\sqrt{\bar{\alpha}_t}}$ ▷ Reconstruct $\mathbf{x}_0$ from noise $\boldsymbol{\epsilon}_\theta$
  Clamp x_0 to range $[0, 255]$
  $\mathbf{x}_{t-1} = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t} + \sigma_t \cdot \mathbf{z}$ ▷ Remove noise using clamped $\mathbf{x}_0$
end for
return x_0

3.2 Video Generation

The three studied models GenTron [CXR+23], Latte [MWJ+24a] and SnapVideo [MSS+24] are based on the Diffusion Transformer architecture.
GenTron can be seen as an extension of DiT, which incorporates additional temporal attention into the Vision Transformer to enable video processing. Furthermore, GenTron adapts the DiT model from class to text conditioning.
Latte takes this a step further by using two separate Vision Transformer blocks – one for spatial attention and another for temporal attention. SnapVideo follows a different approach, using a FIT transformer [CL23] architecture as a backbone, allowing it to process temporal and spatial attention at the same time by dividing the input videos into groups.
The general concept behind the architectures of SnapVideo, GenTron, and Latte are shown in Figure 3.4.



(a) GenTron (b) Latte (c) SnapVideo

Figure 3.4: Spatial and temporal dimension modeling

Due to the high dimensionality of videos, various methods have been proposed to decompose the input and reduce the number of tokens processed in a single attention operation in video generation models. A survey on video transformers [SJE+23] provides an in-depth analysis of current video generation architectures. It highlights the importance of minimizing unimportant spatial information while ensuring that crucial motion features are not removed too early.
The three examined architectures, GenTron, Latte, and SnapVideo, address spatial and frame-level temporal information aggregation differently.
Latte implements two distinct types of Transformer blocks, one operating on the spatial dimension, and the other focusing on the temporal dimension. GenTron, on the other hand, captures both spatial and temporal information in one Transformer block. The Transformer block of GenTron's architecture computes self-attention on the spatial dimension, followed by temporal attention before aggregating the information in a feed-forward

network.
SnapVideo has a different approach. Unlike GenTron and Latte, where each attention layer of the Transformer only sees either spatial dimension or temporal dimension independently, SnapVideo's Transformer block operates on a 3D input capturing height, width, and time of the video input. To capture the spatiotemporal dimensions of the video input without risking information loss, SnapVideo divides the input video into groups along the spatial dimension, still covering the whole temporal dimension.
GenTron conditions the video generation on an input text prompt, while Latte and SnapVideo are designed for generating videos based on class labels. In the following sections, the original architecture of each model will be explained in more detail, while the next chapter focuses on different ways of adapting the Latte and SnapVideo models for text-to-video generation.

3.2.1 GenTron

Input Preprocessing

Video Preprocessing The input videos of shape *(batch size, number of frames, channels, height, width)* undergo several preprocessing steps.

- Autoencoder: The videos are initially encoded into a compressed latent representation using a pretrained autoencoder. This results in an output shape of *(batch size, number of frames, channels encoded, height encoded, width encoded)*. The autoencoder is trained independently on the MovingMNIST dataset.
- Patch embedding: Subsequently, each frame within a video is individually divided into patches resulting in a sequence of patches per frame, with an output shape of *(batch size, number of frames, sequence length, embedding dimension)*.
- Positional encoding: During the last preprocessing step, positional embedding is added – equivalent to the implementation of the image generation model.

Figure 3.5 visualizes the preprocessing process.

Text Preprocessing The lookup table used for label encoding in the image generation model was replaced with a CLIP text embedder. The preprocessing steps in GenTron entail processing the input text with the CLIP tokenizer and afterward mapping the token to embedding space using the CLIP text model. Additionally, classifier-free guidance was introduced, using conditioning dropout. This was achieved by randomly removing the conditional input and replacing it with an empty token. This is implemented before translating the input text to embedding space.

Timestep Preprocessing To introduce the current timestep t of the diffusion process to the model, the same timestep embedder employed for DiT was used.

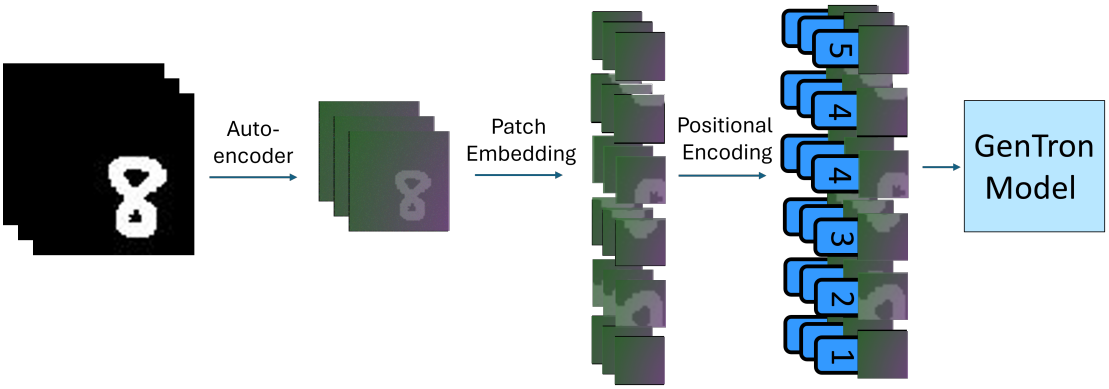


Figure 3.5: GenTron preprocessing

Architectonic Details

GenTron modifies the Transformer block for spatial-temporal modeling by splitting the multi-head attention into two halves – one for spatial modeling and the other for temporal modeling. While spatial and temporal attention are computed separately, both operations occur within the same Transformer.

Spatial Attention Similar to the Diffusion Transformer, the GenTron block first operates on the spatial dimension of the video. To be more precise, it aggregates spatially related information frame by frame using the spatial attention block. In the spatial attention block, the layer-normalization parameters are adjusted by the combination of timestep embedding and the aggregated text embedding, according to the approach of adaptive layer normalization (adaLN) [GWY+22]. The parameters γ_c and β_c for the adaLN are retrieved from a linear regression computed on the condition input $c = t + y_{\text{pool}}$, where t represents the timestep embedding and y_{pool} the pooled CLIP text embedding, resulting in the equation $\text{adaLN}(d, c) = \gamma_c \text{LayerNorm}(d) + \beta_c$, where d expresses the hidden embedding dimension within the Transformer. The timestep embedding lets the model be aware of the noise level that has been added during the forward step of the Denoising Diffusion Probabilistic Model [HJA20]. Moreover, the pooled text embeddings condition the feature channel on the input text prompt.
Given the video patches of shape *(batch size, number of frames, number of patches per frame, embedding dimension)*, spatial attention can be achieved by stacking batch dimension and frame count dimension together, so each frame is handled separately during the attention computing. The reshaping method to implement spatial attention can be described by

$$\mathbf{x} = \text{rearrange}(\mathbf{x}, \text{btnd} \to (\text{bt})\text{nd}), \tag{3.3}$$

$$\mathbf{x} = \mathbf{x} + \text{SpatialSelfAttention}(\text{LayerNorm}(\mathbf{x})). \tag{3.4}$$

where b, t, n, d represent the *batch size*, *number of frames*, *number of patches per frame*, and *embedding dimension*, and where rearrage is a notation from [Rog22].

Text Attention Along with text conditioning through adaLN, the GenTron incorporates an additional cross-attention layer to integrate the text embedding into the model. After the spatial attention block, the image features and textual embeddings interact directly through the attention mechanism.

Temporal Attention The text-cross-attention block is followed by a temporal self-attention block. To implement temporal attention on the given video patches of size *(batch size, number of frames, number of patches per frame, embedding dimension)*, the input dimension needs to be reshaped. By stacking the batch dimension and the patch per frame count together, the focus is shifted from spatial to temporal attention. As a result, each attention calculation concentrates exclusively on patches that have the same position across all video frames, enabling it to capture how the image features change over time within the input video. The reshaping mechanism can be described with

$$\mathbf{x} = \text{rearrange}(\mathbf{x}, \text{btnd} \rightarrow (\text{bn})\text{td}), \tag{3.5}$$

$$\mathbf{x} = \mathbf{x} + \text{TemporalSelfAttention}(\text{LayerNorm}(\mathbf{x})). \tag{3.6}$$

where b, t, n, d represent the embedding dimension *batch size*, *number of frames*, *number of patches per frame*, and *embedding dimension*, and where rearrange is a notation from [Rog22].

Aggregation Congruent with the ViT architecture [DBK+20], the last layer encloses a feed-forward network that allows the model to learn complex relationships between patches. To guide the final aggregation step with the conditional information of timestep and text information, the standard norm layers are again replaced with adaptive layer norm.

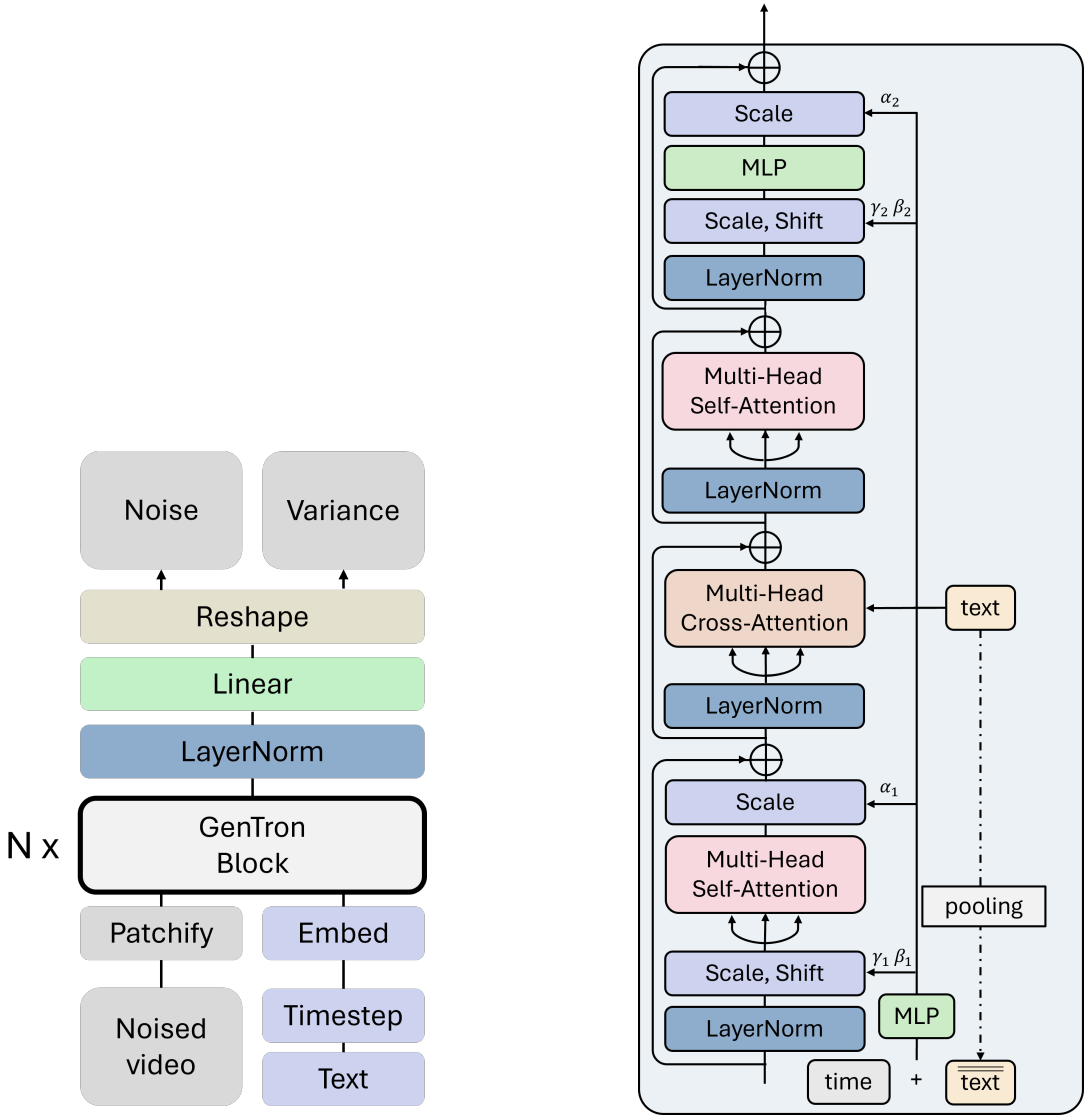
The Gentron block, entailing spatial, text, and temporal attention with a final feed-forward layer, is applied multiple times to the input video patch tokens, before predicting noise and variance. The architectural details are shown in Figure 3.6.

Training

During training, classifier-free guidance is used to condition the diffusion process on a given text prompt. Furthermore, following the original implementation of the GenTron model, the variance is learned during training. The model not only predicts the noise present in the corrupted data sample but also outputs a vector $\mathbf{v}$ of the same shape as the data sample, which is used to compute the variance. This process is summarized in Algorithm 3.2.

Sampling

The sampling process of the GenTron model, integrating the predicted variance, is summarized in Algorithm 3.3.



(a) GenTron (b) GenTron Block

Figure 3.6: GenTron architectonic details

3.2.2 Latte

Input Preprocessing

Video Preprocessing The preprocessing steps are the same as the ones implemented for GenTron, where the noisy input frames are first represented in latent space and afterward transformed into patch tokens with positional encoding.

Before the input videos of shape *(batch size, number of frames, channels, height, width)* are fed into the Latte model, the following preprocessing steps are applied:

Algorithm 3.2: DDPM Training – Classifier-Free Guidance, Variance learning

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\overline{\alpha}_t = \prod_{i=1}^{t} \alpha_i$ probability of unconditional training p_{uncond}

repeat
- $\mathbf{x}_0 \sim q(\mathbf{x}_0, c)$ ▷ Sample x_0 and condition c from training dataset
- $c \leftarrow \varnothing$ with probability p_{uncond} ▷ Randomly discard c to train unconditionally
- $t \sim \text{Uniform}(\{1, \ldots, T\})$
- $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
- $x_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ Corrupt data with Gaussian noise
- $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t), \mathbf{v} = \text{DDPM\_model}(\mathbf{x}_t, t, c)$
- $\Sigma_\theta = \exp\left(\mathbf{v} \log \beta_t + (\mathbf{1} - \mathbf{v}) \log \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \beta_t\right)$ ▷ Predict variance
- Take gradient descent step on $\nabla_\theta \left[\frac{\beta_t^2}{2\|\boldsymbol{\Sigma}_{\boldsymbol{\theta}}\|_2^2 \alpha_t (1-\bar{\alpha}_t)}\right] \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\|^2$

until converged

- Autoencoder: The videos are initially mapped to latent space, using the same pre-trained autoencoder implemented for GenTron.
- Patch embedding: Each frame is individually divided into patches, following the patch embedding architecture of GenTron. The output size after the patch embedding is *(batch size, number of frames, sequence length, embedding dimension)*.
- Positional encoding (spatial): Positional embedding is added to each token, again equivalent to the GenTron model.
- Positional encoding (temporal): Latte also incorporates temporal encoding to help the model identify the frame to which each video patch belongs.

The preprocessing steps are shown in Figure 3.7.

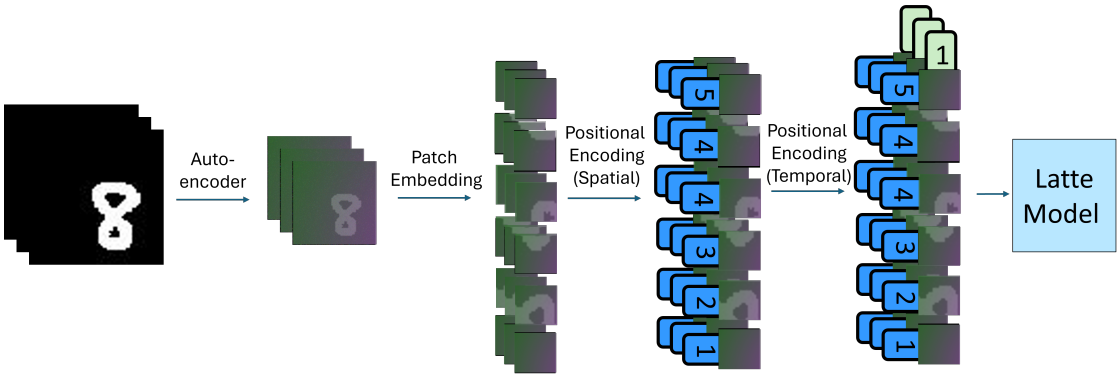


Figure 3.7: Latte preprocessing

Algorithm 3.3: DDPM Sampling – Classifier-Free Guidance, Variance learning

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\overline{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, guidance strength w
Output: reconstructed sample x_0

$\mathbf{c}_{\text{cond}} \sim q(\mathbf{x}_0, c)$
$\mathbf{c}_{\text{uncond}} \leftarrow \varnothing$
$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = T, \ldots, 1$ do **do**
  if $t > 1$ **then**
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$
  else
    $\mathbf{z} = 0$
  end if
  $\boldsymbol{\epsilon}_{\theta_{\text{cond}}}, \mathbf{v} = \text{DDPM\_model}(\mathbf{x}_t, t, c_{\text{cond}})$ ▷ Conditional sampling
  $\boldsymbol{\epsilon}_{\theta_{\text{uncond}}}, \_ = \text{DDPM\_model}(\mathbf{x}_t, t, c_{\text{uncond}})$ ▷ Unconditional sampling
  $\boldsymbol{\epsilon}_t = (1 + w)\boldsymbol{\epsilon}_{\theta_{\text{cond}}} - w\boldsymbol{\epsilon}_{\theta_{\text{uncond}}}$
  $\sigma_t = \sqrt{\exp\left(\mathbf{v} \log \beta_t + (\mathbf{1} - \mathbf{v}) \log \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \beta_t\right)}$ ▷ Remove predicted noise$\boldsymbol{\epsilon}_\theta$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_t\right) + \sigma_t \mathbf{z}$
end for
return x_0

Text Preprocessing Similar to GenTron, Latte uses the CLIP tokenizer and CLIP text model to map the input text to embedding space. By employing the same dropout architecture as GenTron, as described in Section 3.2.1, classifier-free guidance can be introduced during training.

Timestep Preprocessing The diffusion time step is preprocessed using the same timestep embedder implemented in DiT.

Architectonic details

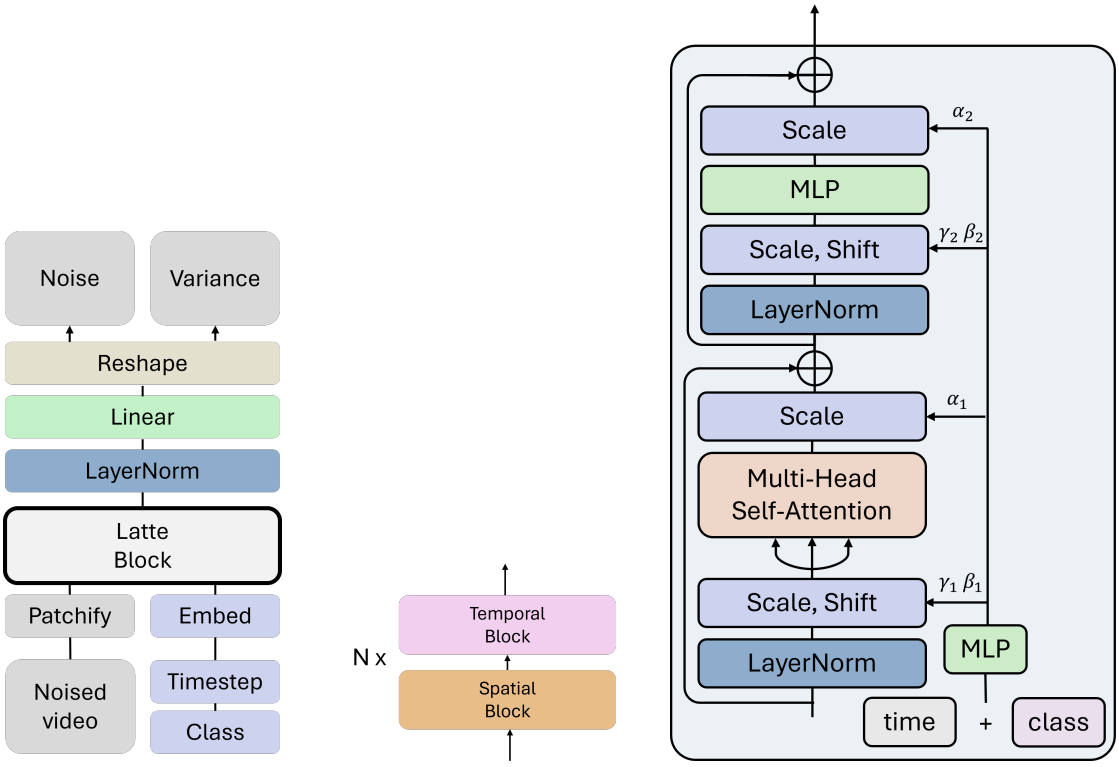
In contrast to GenTron, where spatial and temporal attention is performed within the same block, Latte uses separate Transformer blocks—one for spatial attention and another for temporal attention. Each Transformer block consists of multi-head attention, layer normalization, and a linear projection for handling spatial attention, similar to the Vision Transformer.

Spatial Attention The spatial attention Transformer captures information exclusively among patches within the same video frame. This is implemented using the same reshaping technique as in the GenTron block, which is described in Equation (3.4). The time embedding and class information are both injected in the adaptive layer normalization,

akin to GenTron's conditioning procedure.

Temporal Attention The temporal Attention block implements the same architecture as the spatial attention block described in Section 3.2.2. The difference lies in the shape of the input data. While the spatial attention layer operates on video frame patch tokens of shape $((b\ t)\ n\ d)$, the temporal attention layer operates on input data of shape $((b\ n)\ t\ d)$. In this context, b represents the batch size, t number of frames in the video, n the sequence length of the patch tokens, and d the hidden embedding dimension of the Transformer model. The reshaping mechanism is shown in Equation (3.6).

The Latte Block composes multiple spatial attention and temporal attention blocks, alternating between spatial and temporal attention computation. The exact details of the architecture are visualized in Figure 3.8.



(a) Latte (b) Latte Block (c) Spatial and temporal block

Figure 3.8: Latte architectonic details

Training

The training follows the implementation of GenTron, described in Section 3.2.1.

Sampling

The sampling process follows the implementation of GenTron, described in Section 3.2.1.

3.2.3 SnapVideo

SnapVideo employs a different approach compared to the GenTron and Latte architectures. It builds on the concept of FiT transformers [CL23] and extends it to the domain of video generation. The core idea of the FiT transformer is to work with latent tokens, which are designed to capture and represent the essential information from noisy input video frames in a compressed form. During training, these latent tokens become associated with specific regions of the input data, allowing them to learn which features to emphasize and which to ignore.

Video Preprocessing

Before the input videos of shape *(batch size, number of frames, channels, height, width)* are fed to the SnapVideo model, the following preprocessing steps are applied.

- Autoencoder: By mapping the videos into latent space, SnapVideo receives a compressed representation of the videos, making it easier to model the complex data. This step is implemented by using the same pre-trained autoencoder employed for GenTron.
- Patch embedding: Video frames are divided into patches using the same patch embedding method as in GenTron, with one key difference: After dividing the input into patches, the output is not flattened into a single sequence. Instead, the output retains the shape *(batch size, number of frames, sequence length height, sequence length width, embedding dimension)*.
- Positional encoding: Positional embedding is added to each token after patch embedding, following the implementation of GenTron
- Grouping: The video patches are grouped such that each group corresponds to a distinct region of the video's height and width while spanning the entire temporal dimension. This process results in an output with the shape *(batch size, number of groups, patches per group, embedding dimension)*.

The preprocessing steps are shown in Figure 3.9.

Text Preprocessing The text prompts are initially preprocessed using the CLIP tokenizer and CLIP model, following the text preprocessing steps outlined in GenTron and Latte. Additional learnable parameters are then appended to the text embeddings, inspired by the approach in [DOMB24], and subsequently processed through two self-attention layers.

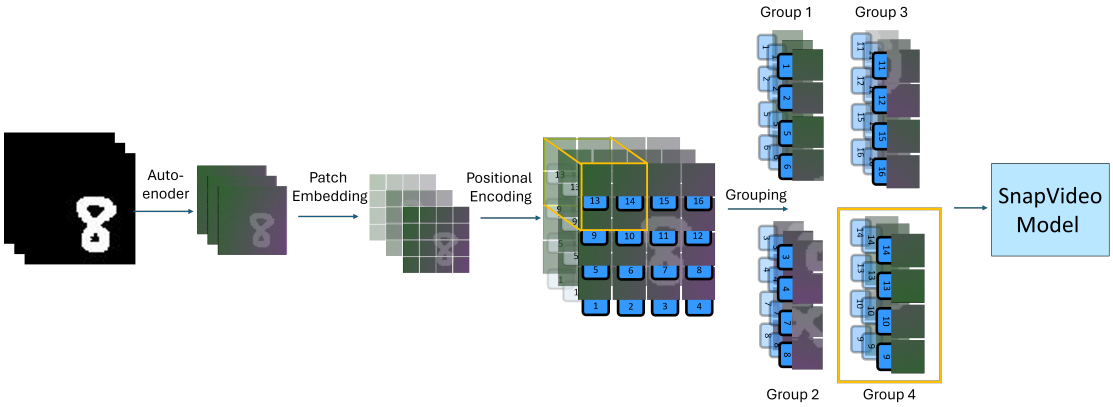


Figure 3.9: SnapVideo preprocessing

Timestep Preprocessing Using the same timestep embedder implemented in DiT, the current diffusion timestep t can be introduced to the model.

Architectonic Details

To manage the complexity of spatial and temporal information in video data, the model operates exclusively on learned latent tokens. These tokens aggregate information from the input video patch tokens, class labels, and timestep embeddings. During a cross-attention operation between the latent tokens and the combined class labels and timestep embeddings, the latent tokens extract conditional information. Next, the latent tokens gather information from each group of video patches through a "read" operation, implemented via cross-attention. The tokens then refine this aggregated information through a self-attention layer, allowing them to integrate intricate details from the input. Finally, the latent tokens "write" their updated insights back to the patch tokens, achieved through another cross-attention operation. The model outputs the predicted noise, the reverse process variance, and the latent tokens. These latent tokens are reused in the next iteration for self-conditioning, where the previous latent tokens directly influence the new latent states. Prior research has shown that this technique can significantly enhance sampling quality [CZH23].

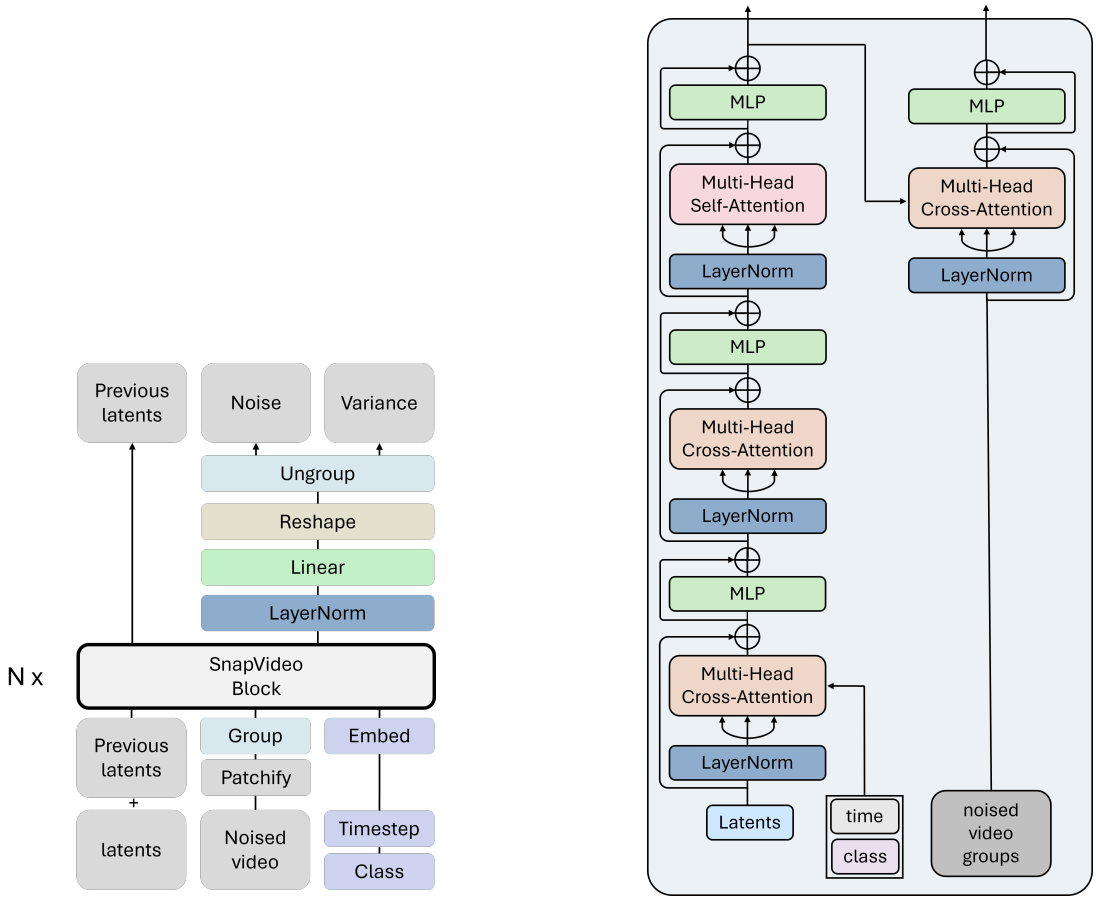
The architecture of the SnapVideo model is illustrated in Figure 3.10.

Training

The training of the SnapVideo model with latent self-conditioning is described in Algorithm 3.4.

Sampling

The model returns the predicted noise as well as the current latent tokens. The latent tokens from the previous iteration are used to condition the latent token of the current



(a) SnapVideo (b) SnapVideo Block

Figure 3.10: SnapVideo architectonic details

iteration, to maintain the compressed video representation learned in previous sampling steps, described in Algorithm 3.5.

3.3 Text-to-Video Generation

This chapter examines different ways of integrating text understanding during the video generation process. Both Latte and SnapVideo are designed for class-label conditioning. A text prompt corresponds to a textual description of the kind of video the model should generate, while class labels describe different categories in which the video data can be classified. The key difference between text prompts and class labels is their length, as text prompts can grow indefinitely and class labels have a fixed size.
GenTron focuses on two different approaches to embedding integration. Either using the adaptive layer norm to align the mean and variance of the video features with those of the text condition or employing cross-attention, where the model learns to combine the

Algorithm 3.4: DDPM Training – Classifier-Free Guidance, Variance learning, Latent Self-Conditioning

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\overline{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, probability of unconditional training p_{uncond}

latents $= \mathbf{0}$ ▷ Initialize latents
repeat
  $\mathbf{x}_0 \sim q(\mathbf{x}_0, c)$ ▷ Sample x_0 and condition c from training dataset
  $c \leftarrow \varnothing$ with probability p_{uncond} ▷ Randomly discard c to train unconditionally
  $t \sim \text{Uniform}(\{1, \dots, T\})$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
  $x_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ Corrupt data with Gaussian noise
  $\boldsymbol{\epsilon}_\theta, \mathbf{v}, \text{latents} = \text{DDPM\_model}(\mathbf{x}_t, t, c, \text{latents})$
  $\Sigma_\theta = \exp\left(\mathbf{v} \log \beta_t + (\mathbf{1} - \mathbf{v}) \log \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t\right)$ ▷ Predict variance
  Take gradient descent step on $\nabla_\theta \left[\frac{\beta_t^2}{2\|\boldsymbol{\Sigma}_{\boldsymbol{\theta}}\|_2^2 \alpha_t (1-\bar{\alpha}_t)}\right] \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\|^2$
until converged

textual information and embeds it into the feature space.

3.3.1 Latte Text Integration

The spatial and temporal transformers of the original Latte model remain unchanged. In the Latte model, adaLN is used to incorporate class information into the architecture. While the structure of adaLN is kept the same, the class input is replaced with the pooled text embedding, following a similar approach to GenTron. Additionally, a text attention block, identical to the one used in GenTron for text integration, is introduced. However, the placement of the text attention block within the architecture becomes a critical question. The first logical choice is to position the text attention block between the spatial transformer and the temporal transformer block. This concept is implemented in Model 1, illustrated in Figure 3.11. The algorithm behind this implementation is detailed in Appendix 7.2.

Model 2 utilizes the same text block as model 1, but this time the text block is integrated into the spatial attention block before the aggregation layer. Figure 3.12 visualizes the spatial transformer with text attention and the temporal transformer. The temporal transformer remains unchanged from the temporal transformer used in Model 1. The algorithm of Model 2 can be found in Appendix 7.3.

Model 3 integrates the text block in both spatial and temporal attention blocks, as shown in Figure 3.13. The algorithm can be found in Appendix 7.4.

Algorithm 3.5: DDPM Sampling – Classifier-Free Guidance, Variance learning, Latent Self-Conditioning

Input: noise scheduler β_t with $\alpha_t = 1 - \beta_t$, $\overline{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, guidance strength w
Output: reconstructed sample x_0

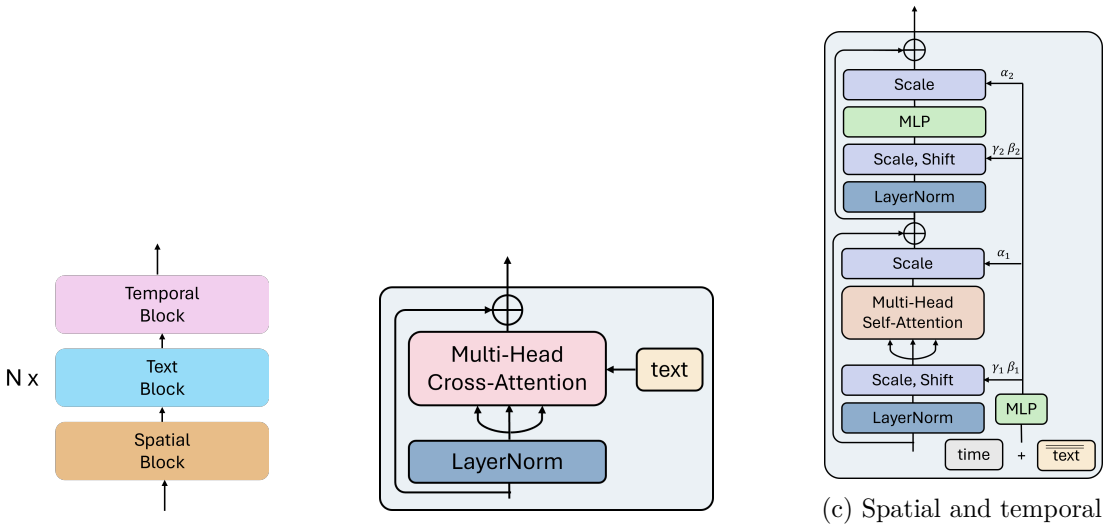
$\mathbf{c}_{\text{cond}} \sim q(\mathbf{x}_0, c)$
$\mathbf{c}_{\text{uncond}} \leftarrow \varnothing$
$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
$\text{latents}_{\text{cond}} = \mathbf{0}$
$\text{latents}_{\text{uncond}} = \mathbf{0}$
for $t = T, \ldots, 1$ do **do**
  if $t > 1$ **then**
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$
  else
    $\mathbf{z} = 0$
  end if
  $\boldsymbol{\epsilon}_{\theta_{\text{cond}}}, \mathbf{v}, \text{latents}_{\text{cond}} = \text{DDPM\_model}(\mathbf{x}_t, t, c_{\text{cond}}, \text{latents}_{\text{cond}})$
  $\boldsymbol{\epsilon}_{\theta_{\text{uncond}}}, \_, \text{latents}_{\text{uncond}} = \text{DDPM\_model}(\mathbf{x}_t, t, c_{\text{uncond}}, \text{latents}_{\text{uncond}})$
  $\boldsymbol{\epsilon}_t = (1 + w)\boldsymbol{\epsilon}_{\theta_{\text{cond}}} - w\boldsymbol{\epsilon}_{\theta_{\text{uncond}}}$ ▷ Classifier free guidance sampling
  $\sigma_t = \sqrt{\exp\left(\mathbf{v} \log \beta_t + (\mathbf{1} - \mathbf{v}) \log \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \beta_t\right)}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_t\right) + \sigma_t \mathbf{z}$
end for
return x_0

3.3.2 SnapVideo Text Integration

For the SnapVideo text integration, Model 1 is designed by following the text guidance of a text-to-video generation model, which uses a similar transformer architecture as SnapVideo. It is described in the paper "Don't Drop Your Samples! Coherence-aware Training Benefits Conditional Diffusion" [DBKP24]. This paper introduces a text-to-video transformer based on the RIN transformer architecture [JFC22], which implements the same fundamental idea as the FiT transformer [CL23] used in SnapVideo.

A key difference between the RIN transformer and the FiT transformer can be seen in the preprocessing step: the FiT transformer divides the input videos into groups, while the RIN transformer omits this step. Instead, the RIN transformer compresses the video dimension by applying a 3D patch embedding that spans height, width, and the number of frames. SnapVideo, on the other hand, uses a 2D patch embedding, which only covers the spatial dimensions.

Apart from preprocessing differences, the architectures of SnapVideo and the RIN text-to-video model are nearly identical. Therefore, a similar text conditioning method to that used in the RIN text-to-video model was implemented.



(a) Latte Block (b) Text Block (c) Spatial and temporal block

Figure 3.11: Latte Text integration model 1

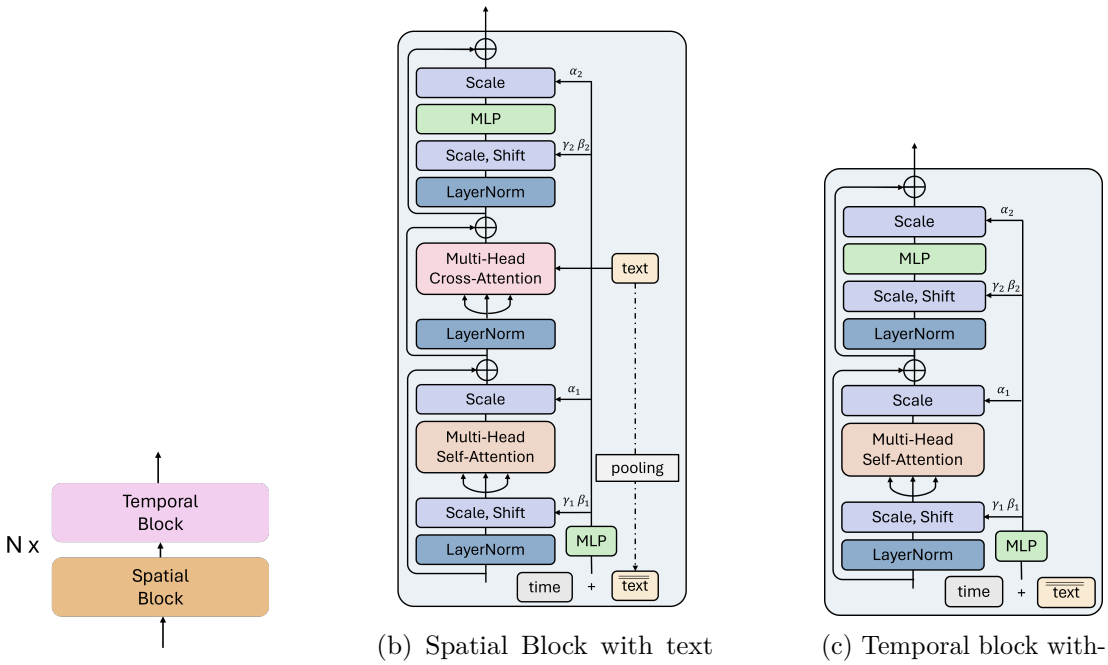
In the RIN model, the text is concatenated with 16 learnable parameters and passed through two self-attention layers. Adding learnable parameters to the text is a method described in the paper "Vision Transformers Need Registers" [DOMB24], which highlights the tendency of models to use local tokens for storing and processing global information. This can potentially lead to the loss of local patch information. The additional learnable tokens are used to counteract this effect by explicitly reserving space for storing additional information.

Figure 3.14 illustrates the implementation of text conditioning based on the RIN text-to-video model. The algorithm for this implementation is detailed in Appendix 7.5.

In Model 2, adaLN text conditioning, similar to the approach of the GenTron architecture, was introduced. The layer normalization of the cross-attention between the latent token and text projection, as well as the layer normalization of the self-attention, were replaced with adaLN. The architecture is illustrated in Figure 3.15, and the algorithm is provided in Appendix 7.6.

3.3.3 Dataset

This work prioritized selecting a dataset that was simple enough to allow training within a reasonable time frame and with limited resources while still being capable of demonstrating differences in spatial and temporal efficiency between models. The dataset selected for training the different Diffusion Transformer models is MovingMNIST. It is a simple dataset containing text video pairs, illustrating the movement of handwritten letters. In this work, the dataset is further reduced from the original dataset by only incorporating videos with the following characteristics.



(a) Latte Block

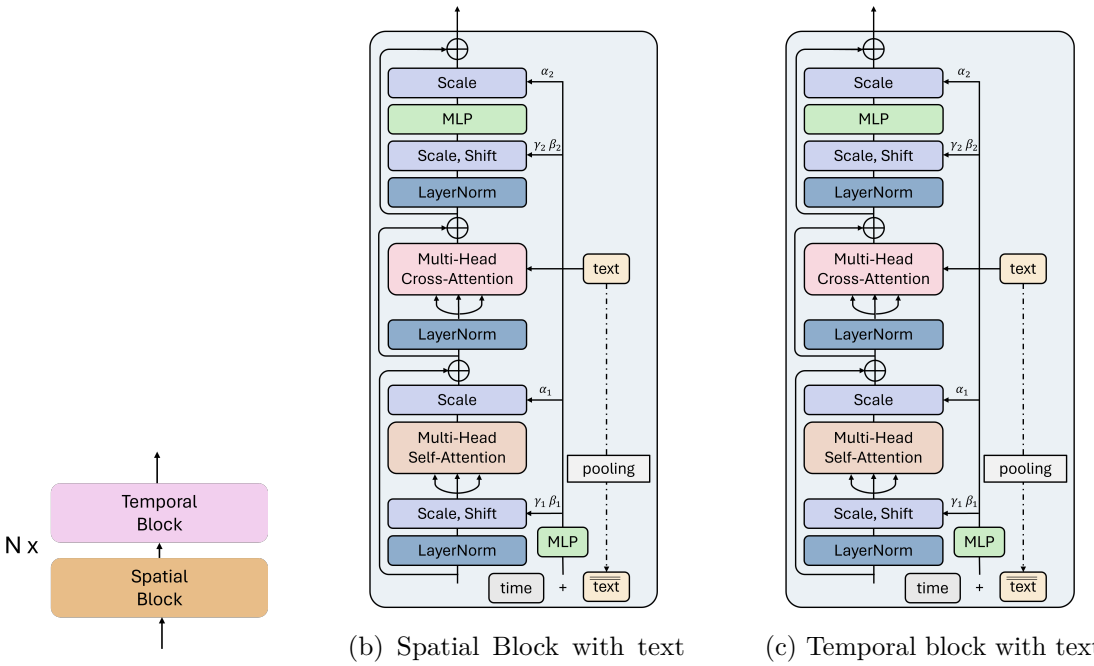
(b) Spatial Block with text integration

(c) Temporal block without text integration

Figure 3.12: Latte text integration model 2

- Each video shows only the movement of one digit. The digits used are 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Each video shows one of the three movements:
  - up and down,
  - left and right,
  - to and fro – standing for a constant movement backwards and forwards.
- Each video has a length of 10 frames.
- Each video is described with a corresponding text prompt stating the number contained in the video and the movement.

Figure 3.16 showcases three examples of the dataset, illustrating video frames (not all 10 video frames are shown in the image) with their corresponding text description.

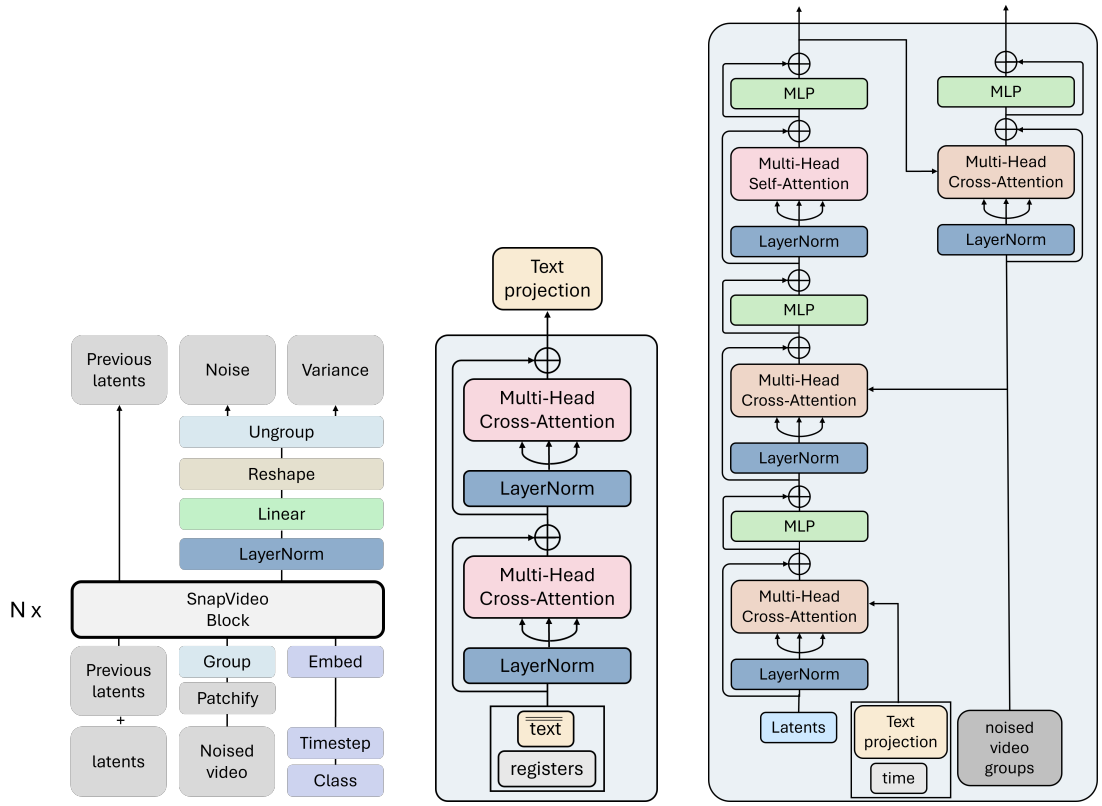


(a) Latte Block

(b) Spatial Block with text integration

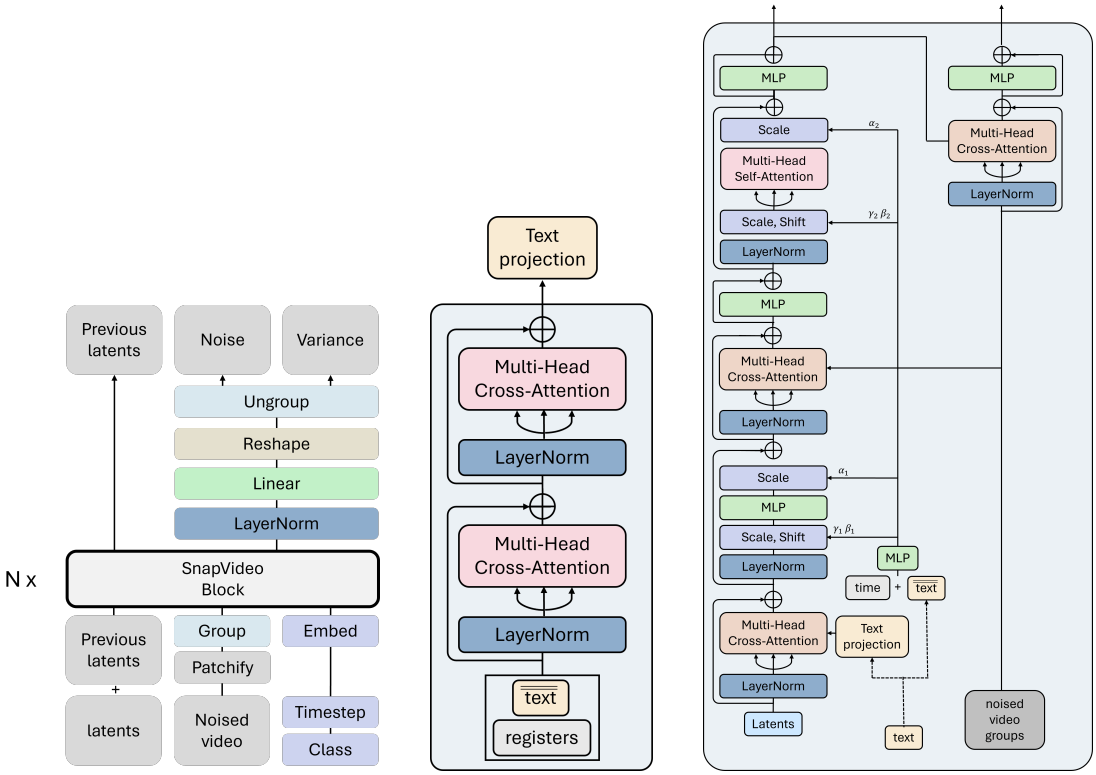
(c) Temporal block with text integration

Figure 3.13: Latte text integration model 2



(a) SnapVideo (b) Text Block (c) SnapVideo Block

Figure 3.14: SnapVideo text integration model 1



(a) SnapVideo (b) Text Block (c) SnapVideo Block

Figure 3.15: SnapVideo text integration model 2

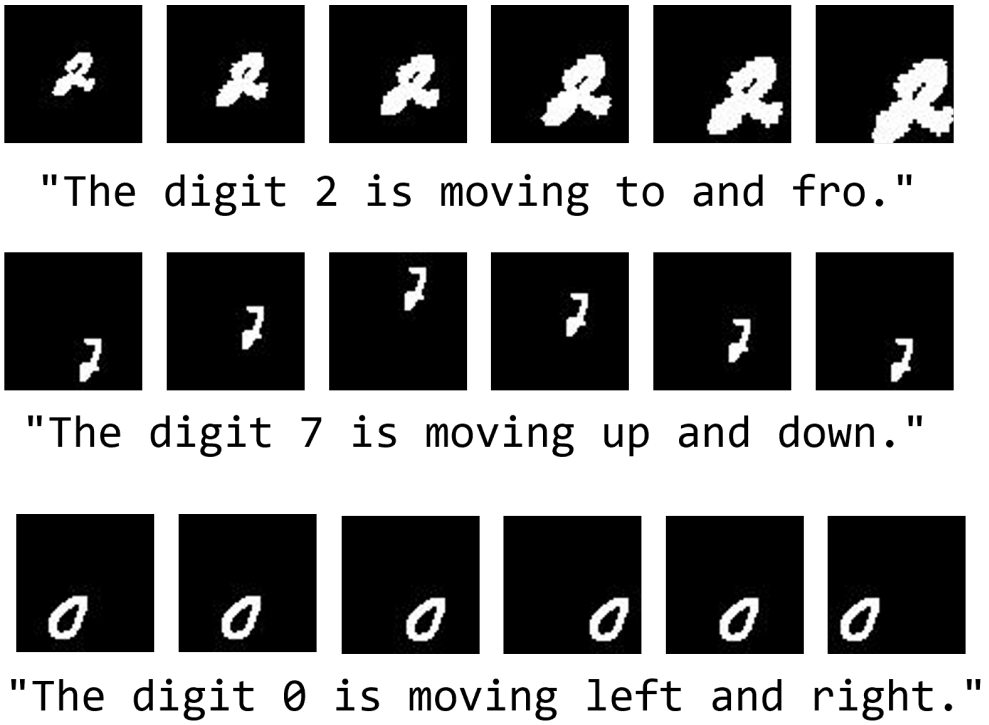


Figure 3.16: MovingMNIST dataset

CHAPTER 4

Results

4.1 Image Generation (DiT)

4.1.1 Inference Details

The final sampling step T was set to $T = 1000$.

4.1.2 Implementation Details

The training process was implemented from scratch following the implementation details of [Nic21]. The first version of the implementation followed the sampling process according to Algorithm 2.2. After 500 epochs, while the loss continued to decrease, the sampling results did not show any indication that the model was effectively learning to represent the training dataset. A detailed examination of the output revealed that the reverse process failed to generate samples within a valid pixel range. Specifically, the mean and variance did not converge toward the distribution of the training data, leading to an unstable output that significantly differed from the expected values.
Upon comparing my implementation with various diffusion model implementations, I discovered that many of them employ a technique called thresholding to counteract this effect. Although not mentioned in the original DDPM paper, this approach has been utilized in several subsequent works, such as IMAGEN [SCS+22]. During thresholding, the reverse process is redefined, retrieving the predicted sample $\mathbf{x}_0$ first before computing the reverse step towards $\mathbf{x}_{t-1}$. Afterward, the predicted sample x_0 is clamped to a valid pixel range of [0,255] and used to compute $\mathbf{x}_{t-1}$.
In other words, instead of retrieving $\mathbf{x}_{t-1}$ directly from the predicted noise $\epsilon_\theta(\mathbf{x}_t, t)$, the reverse process is reformulated to compute $\mathbf{x}_{t-1}$ from x_0. The sample $\mathbf{x}_0$ is obtained by subtracting the entire predicted noise ϵ_θ from the current corrupted sample $\mathbf{x}_t$, which is described in Algorithm 3.1.
After incorporating thresholding into the sampling process, the desired effect was achieved,

and the model began generating images that increasingly aligned with the training dataset. The training details of the DiT model are described in Table 4.2.

Training details		
Hardware	GPU	NVIDIA GeForce GTX 1050
	GPU memory	2GB
Diffusion	Train dataset size	60.000
	Image Size ($height \times width \times channels$)	$28 \times 28 \times 1$
	Batch Size	128
	Epochs	100
	Learning Rate	0,001
	Noise Scheduler	cosine noise schedule
DiT	Patch Size	2
	Embedding dimension	64

Table 4.1: Training details of the DiT model

4.1.3 Inference Results DiT

Figure 4.1 shows the sampling results of the DiT model in a collage view, where the images are generated from the given class labels sequence $[2, 3, 1, 8, 2, 1, 5, 5, 8, 0, 1, 2, 5, 8, 2, 1]$.

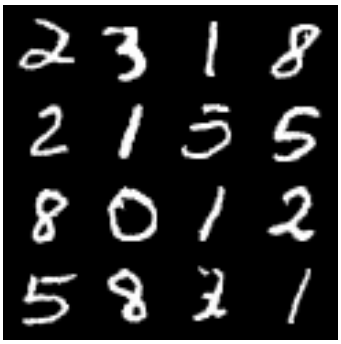


Figure 4.1: DiT generated images

4.1.4 Evaluation Results

The evaluation metrics were calculated over a dataset containing 1000 predicted images and 1000 images from the training dataset. The dataset for metric calculation consists of 100 images for each possible digit from zero to nine, sampled from the model. To compare the quality of the sampled images, 100 images for each digit were randomly chosen from the training dataset and added to the metric dataset. The evaluation results after training the model for 100 epochs are shown in Table 4.2.

FID ↓	CLIPSIM ↑	SSIM ↑	PSNR ↑	LPIPS ↓
12.790	0.9978	0.2304	10.046	0.1581

Table 4.2: Evaluation results of the DiT model trained on the MNIST dataset (100 epochs)

4.2 Video Generation

4.2.1 Implementation Details

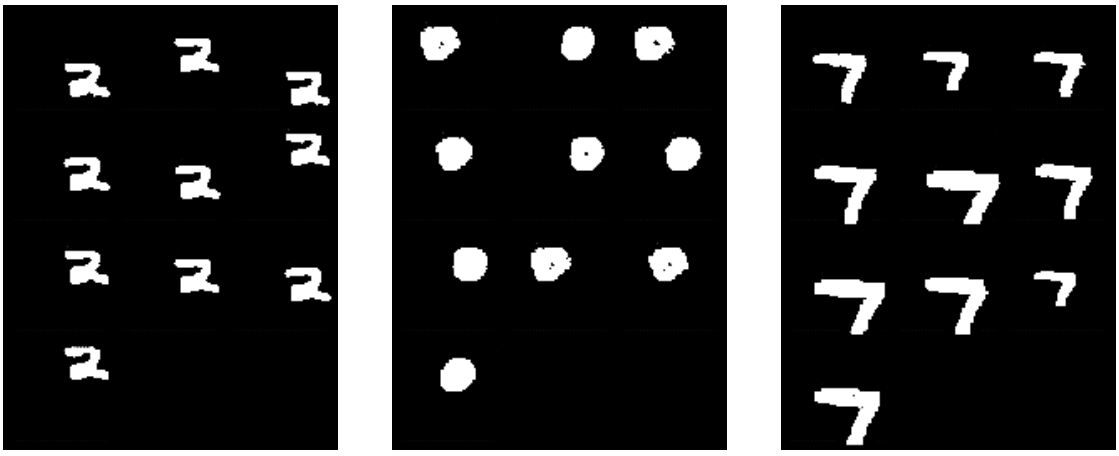
The diffusion code was not implemented from scratch but was instead adapted from the code provided by [SCS+22]. It is a widely used implementation adopted by many of the current Diffusion Transformer variants. The diffusion code provides the possibility to integrate variance prediction into the diffusion process. The training details of the video generation models are described in Table 4.3.

4.2.2 Inference Details

The final sampling step T was set to $T = 250$.

4.2.3 Inference Results GenTron

Figure 4.2 shows the frames of three sampled videos from GenTron after training it for 100 epochs on the MovingMNIST dataset. The images display video frames in a collage view, generated by the model in response to the text prompt provided in each caption.



(a) The digit 2 is moving up and down

(b) The digit 0 is moving left and right

(c) The digit 7 is moving to and fro

Figure 4.2: Sampling results of GenTron

Training details		
Hardware	GPU	NVIDIA A40
	GPU memory	48GB
Diffusion	Train dataset size	20.000
	Frame Size ($height \times width \times channels$)	$28 \times 28 \times 4$
	Batch Size	4
	Epochs	100
	Learning Rate	0,0001
	Noise Scheduler	cosine noise schedule
GenTron	Patch Size	2
	Dropout Probability	0.1
	CFG-Scale	7.5
	Embedding dimension	1152
	Number Transformer Blocks	5
Latte	Patch Size	2
	Dropout Probability	0.1
	CFG-Scale	7.5
	Embedding dimension	1152
	Number Transformer Blocks	5
SnapVideo	Patch Size	2
	Dropout Probability	0.1
	CFG-Scale	7.5
	Number Transformer Blocks	5
	Patch embedding dimension	768
	Latent embedding token	1024
	Number of Self-Attention Layers	4

Table 4.3: Training details of video generation models

4.2.4 Inference Results Latte v1

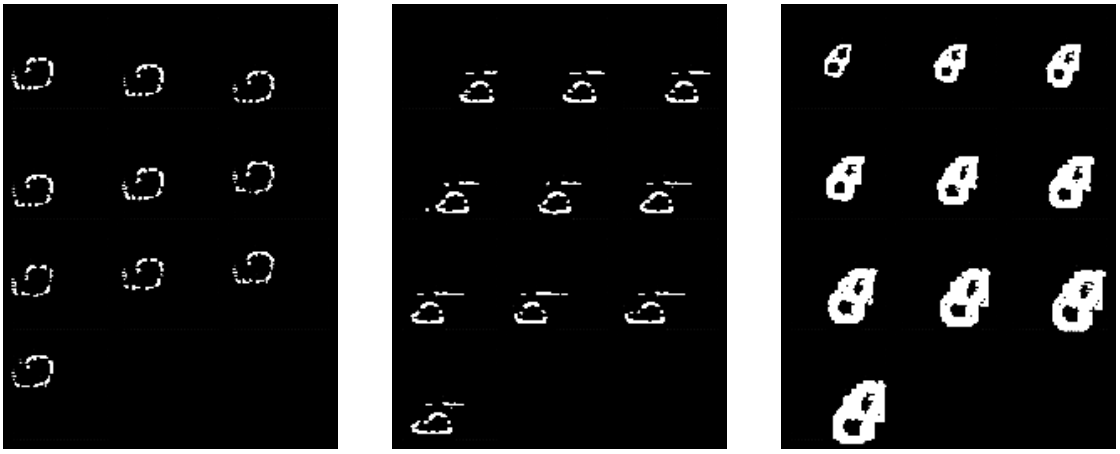
The sampling results of the first Latte model variant are shown in Figure 4.3, with the generated video frames according to the text prompts described in the captions.

4.2.5 Inference Results Latte v2

The sampling results of the second Latte model variant are shown in Figure 4.4, with the generated video frames and the corresponding text prompts.

4.2.6 Inference Results Latte v3

The generated video of the third Latte model are shown in Figure 4.5, where each image represents a video frame generated from the corresponding text description in the

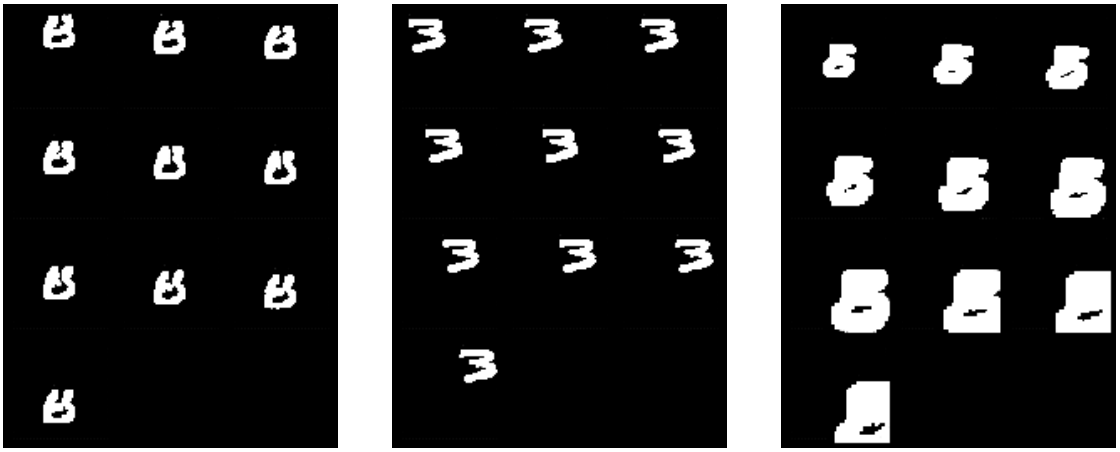


(a) The digit 0 is moving up and down

(b) The digit 5 is moving left and right

(c) The digit 8 is moving to and fro

Figure 4.3: Sampling results of Latte v1



(a) The digit 0 is moving up and down

(b) The digit 3 is moving left and right

(c) The digit 8 is moving to and fro

Figure 4.4: Sampling results of Latte v2

captions.

4.2.7 Inference Results SnapVideo v1

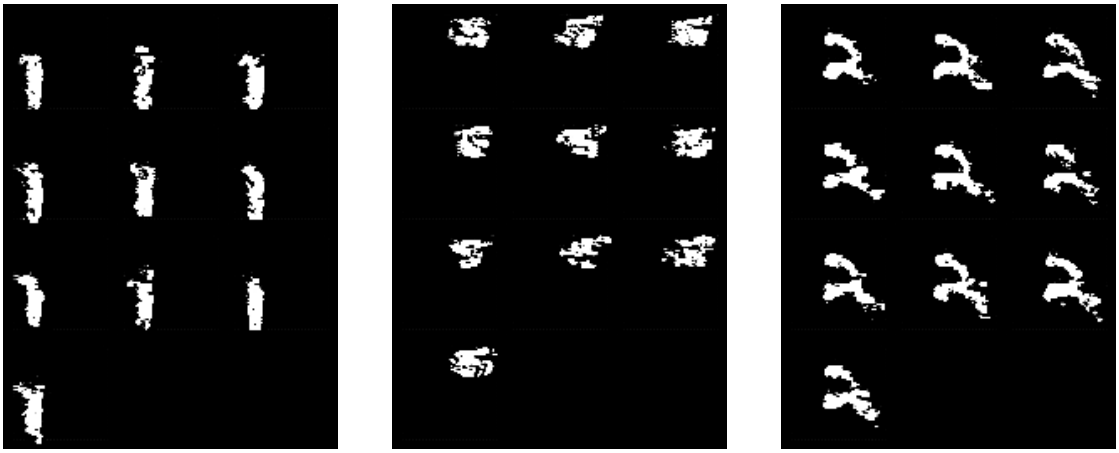
Figure 4.6 shows the inference results of the first SnapVideo variant, visualizing the video frames of the corresponding text captions.

(a) The digit 8 is moving up and down

(b) The digit 7 is moving left and right

(c) The digit 9 is moving to and fro

Figure 4.5: Sampling results of Latte v3



(a) The digit 8 is moving up and down

(b) The digit 5 is moving left and right

(c) The digit 2 is moving to and fro

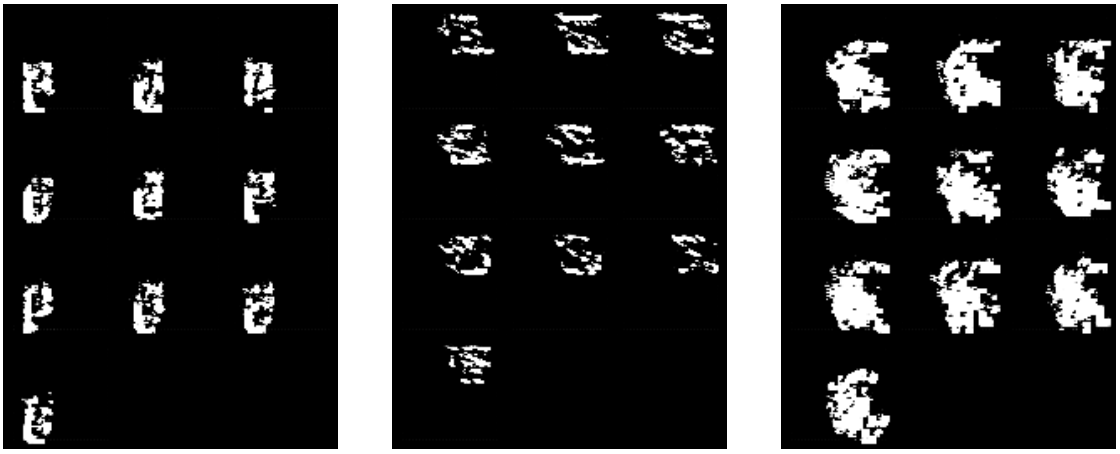
Figure 4.6: Sampling results of SnapVideo v1

4.2.8 Inference Results SnapVideo v2

Figure 4.7 shows the generated videos from the second SnapVideo model, with the sampled video frames given the corresponding text captions.

4.2.9 Evaluation Results

For the evaluation, a metric dataset was created, similar to the approach used for the Image generation model in Section 4.1.4. The metric dataset contains 40 videos for each possible digit (0–9) and movement (up and down, left and right, to and fro) combination,



(a) The digit 0 is moving up and down

(b) The digit 5 is moving left and right

(c) The digit 6 is moving to and fro

Figure 4.7: Sampling results of SnapVideo v2

sampled from the model (1200 videos) and additionally from the training dataset (1200 videos). The videos were sampled after training each of the models for 100 epochs. The evaluation results of the video generation models are shown in Table 4.4.

Method	**FVD** ↓	**CLIPSIM** ↑	**SSIM** ↑	**PSNR** ↑	**LPIPS** ↓
GenTron	111.0617	**0.7808**	0.6520	10.786	0.3045
Latte v1	**61.659**	0.7244	**0.6865**	11.302	**0.2871**
Latte v2	75.019	0.7040	0.6633	**11.517**	0.3376
Latte v3	11009.236	0.0611	0.0010	1.197	0.9542
SnapVideo v1, 4 Groups	302.663	0.7027	0.6033	10.1904	0.3629
SnapVideo v1, 49 Groups	540.2325	0.5240	0.5955	11.208	0.4104
SnapVideo v2, 4 Groups	401.691	0.6852	0.5452	9.610	0.4126

Table 4.4: Evaluation results of the video generation models trained on the Moving MNIST dataset (100 epochs)

CHAPTER 5

Discussion

5.1 From Image to Video Generation

The DiT class-to-image generation model was trained on a simple laptop GPU but still showed promising results after 170 hours of training with a batch size of 128 images. After adapting the class-to-image generation model according to the text-to-video generation model architecture of GenTron, training locally was cut short as even a batch size of 1 resulted in an out-of-memory error.
After gaining access to an external server, training on a higher-performance GPU was possible. However, the number of videos per batch was still limited due to memory constraints. It was remarkable to see the difference between image and video generation models in terms of memory usage. Due to the additional temporal dimension, in this case, ten frames per video, as well as the model complexity, successfully training a video generation model is only feasible on hardware with sufficient computational capacity, such as high-performance GPUs optimized for parallel processing and large-scale data throughput.
In the beginning, the preprocessing steps as well as the training and sampling algorithm were kept consistent with the implementation of the class-to-image model. However, after testing different learning rates, batch sizes, and patch sizes, the video generation model did not demonstrate any progress in learning. The first significant adaptation made was to encode the input videos into latent space utilizing a pretrained Autoencoder. This preprocessing step was crucial, marking the first time the training and evaluation loss decreased. Still, the sampled video frames showed blurry, indistinct shapes with scattered white pixels throughout the frames.
Further research showed that the implementation of current generative diffusion models is mostly based on the improved DDPM code from OpenAI [Nic21]. The key difference to the custom implementation used for the DiT model is the additional variance learning of the reverse diffusion process. Connecting the GenTron architecture with the improved

DDPM implementation resulted in higher sampling quality. The sampled videos displayed shapes concentrated in a single area, gradually beginning to resemble numbers. Further refinement was achieved by employing Classifier-Free Guidance, integrating the textual information not only into the architectural backbone but also throughout the diffusion process. Further training along with batch size and learning rate optimization, ultimately led to a prediction quality that matched the visual representation of the input text, resembling the training dataset samples. Nonetheless, the most significant step, with the greatest impact, was using an autoencoder to map the input video to a compressed representation. This demonstrates the importance of reducing information during preprocessing. Still, it is a delicate balancing act, as premature information loss can lead to poor sampling quality. On the other hand, insufficient information compression not only demands more memory but also hinders the models from finding patterns within the data. For example, using a larger patch size during preprocessing resulted in poorer outcomes, which is why a patch size of 2 was consistently maintained throughout the different implementations.

One challenge I frequently faced during implementation was selecting the correct shape at each step through the different layers of the architecture. Since videos, especially long ones, cannot be processed as a whole, they must be handled step by step. This requires different reshaping mechanisms to focus either on the temporal or spatial dimension of the video. The same applies to the textual input, which has to align with the shape of the video data to accurately integrate textual information with the corresponding video. The initial implementation of SnapVideo, for instance, failed to demonstrate any learning, producing random noise. After multiple code reviews, I identified the error in the the attention mechanism between input video and text, causing one video to receive information from all possible text prompts in the batch, rather than just the relevant one. As a result, the model could not understand the specific video it needed to generate. The need to manage various shapes throughout the process made implementing the video generation models more complicated than the implementation of the image generation model DiT, which can process image patches directly without reshaping.

5.2 Video Generation – Spatial and Temporal Attention

The key difference between GenTron, Latte, and SnapVideo is the spatial and temporal dimension handling of the input videos. GenTron employs one Transformer block for isolated spatial and temporal dimension processing. Latte, on the other hand, utilizes two distinct Transformer blocks, one focusing on spatial, the other focusing on temporal dimension. SnapVideo tries to process spatial and temporal dimensions concurrently. Interestingly enough, the different approaches are reflected in the sampling quality. GenTron shows the best results regarding frame-level image quality. The visualized numbers within frames demonstrate high precision and match the textual description. However, even though the movements between frames resemble the input prompt, sometimes the movement is discontinuous, which is especially visible when the digit is moving from the left to the right. The Transformer block of GenTron can be

split into two distinct focus areas, where the first part is responsible for learning the intricate details within each frame, while the second part learns how the frames change over time, aggregating the learned information in the end. However, as the Transformer block employs spatial attention immediately after the temporal attention, it might be the cause of losing important information during subsequent temporal modeling, leading to incoherent movement. Latte, on the other hand, dedicates two separate Transformer blocks for the temporal and spatial dimension handling – including multi-head attention, layer norm, and linear projection – instead of only employing multi-head attention during temporal modeling like GenTron. Latte's results demonstrate enhanced temporal coherence in the sampled videos compared to GenTron but fall short of accurately visualizing the numbers within the frames, which are often distorted and sometimes unrecognizable as actual numbers. Using a separate Transformer model has its trade-offs: while it can produce smoother videos over time compared to a single Transformer handling separate spatial and temporal attention, it also appears to require longer training durations.
SnapVideo adopts a significantly different approach in comparison to Latte and GenTron Rather than treating the spatial and temporal dimensions separately, it processes them concurrently by dividing the input video into groups along the spatial dimension. This allows the attention mechanisms to learn both frame-specific and movement-related information from different parts of the videos simultaneously. Moreover, GenTron and Latte perform information aggregation directly on the input video, meaning the newly learned information is integrated into the video as it passes through the model. SnapVideo, however, operates on a separate branch of learnable tokens, called latents. These latent tokens first extract key information from the input, refine it by removing irrelevant details and afterwards integrate the learned information into the input. By shifting the computation from the input to a more compact latent representation and dividing the input videos into groups, joint spatiotemporal attention is possible.

Prior research suggests that concurrent modeling of space and time can enhance temporal consistency [SJE+23]. However, the results did not demonstrate any improvements; on the contrary, the generated videos were static containing blurry figures, mostly failing to resemble recognizable numbers. The initial configuration split the input video into four spatial groups, leading to the assumption that the groups might be too large to capture the necessary spatial and temporal details. However, increasing the number of groups to 49 only worsened the outcome, producing random blotches across frames, and failing to learn the distribution of the input data. Since the original implementation does not mention any latent space mapping during the preprocessing stage, the model was also tested directly on pixel space, which resulted in black video frames. Further adjustments to hyperparameters failed to improve the results.

While increasing the training duration and further experiments regarding hyperparameter tuning may improve performance, the findings suggest that separate spatial and temporal modeling is more effective, particularly when working with simple datasets.

5.3 From Class to Text Guidance

The adaptation of Latte for text-to-video generation in this work was inspired by GenTron, while SnapVideo's adaptation was based on a similar model, CAD-RIN [DBKP24].
Adapting the model for text integration instead of class-based generation involves changes in both the architectural design and the preprocessing steps. To prepare the text for the denoising network, it was mapped to embedding space. Following GenTron's approach, this study uses the CLIP model to generate text embeddings.
SnapVideo also adds learnable tokens to its text embeddings, allowing the model to better store and process global text information. This technique was applied in the SnapVideo implementation after generating the text embeddings.
Once the preprocessing of the text is complete, the challenge is integrating text into the model. GenTron and CAD-RIN incorporate additional attention mechanisms for text integration. In comparison, class-based models like Latte do not integrate the conditional class label using additional attention mechanisms but instead modify the Vision Transformer backbone with adaptive layer norm layers (adaLN). GenTron's findings however show that only relying on adaLN layers for injecting class-based information is insufficient.
GenTron introduces text only in the spatial dimension, raising the question of whether alternative integration placements could improve results. This idea was examined in the Latte model by inserting the text at three separate points within the architectural framework. The evaluation revealed that embedding text in the spatial attention block before aggregation delivered the best results, accurately visualizing both the numbers and their described movements. Integrating text between spatial and temporal blocks produced videos that captured movements but failed to visualize the correct numbers according to the input text. Incorporating text into both spatial attention and temporal attention blocks caused the model to generate black video frames.

The initial SnapVideo model uses CAD-RIN-inspired text integration, while a second model extended this approach by adding adaLN to the Vision Transformer architecture based on GenTron's method of text integration. CAD-RIN-style method demonstrated basic textual understanding, producing blurry but identifiable numbers. Furthermore, the model seems to be able to distinguish between different types of movements described in the text. While the predicted videos were static, the placement of the figures shown in the videos as well as their shapes changed according to the movement described in the text. For instance, videos based on "left and right" displayed wide rectangular figures at the top of the frames, whereas those describing "up and down" showed narrow rectangular shapes lower-left corner. The "to and fro" movement caused the figures of the generated videos to appear more zoomed in. However, additional adaLN integration provided no notable improvements.

Overall, attention mechanisms play a critical role in successful text integration. When text is integrated during spatial attention, it provides a solid starting point for the model, helping it understand the context for generating new frames. In contrast, adding text at

later stages tends to worsen the outcomes.

5.4 Evaluation Metrics

Common metrics for image generation evaluation include FID, CLIPSIM, SSIM, PSNR, and LPIPS. Video generation, on the other hand, offers fewer established metrics, with FVD being the most commonly used. Moreover, there are no established metrics for video generation that evaluate whether the generated video aligns with a given text prompt. As a result, image-based metrics are often applied to video generation using frame-wise comparison. CLIPSIM can for example be extended as a video generation metric, by comparing the similarity of a given text with each frame individually and averaging the result over the CLIPSIM value of a corresponding video from the dataset. However, this method does not assess how well the motion described in the text is represented in the generated video, as it only considers each frame individually. Still, CLIPSIM provided the most accurate reflection of the perceived quality across the different models. FVD does not incorporate the input text to evaluate the models' sampling quality. The metric only compares how well the distribution of the generated videos matches the distribution of the training dataset. The FVD metric rated GenTron's generated video lower than Latte v1 and Latte v2, even though GenTron demonstrated better text alignment and more accurate visualization of the numbers in each frame. However, Latte v1 and Latte v2 generated videos with more coherent movement in comparison to GenTron. This suggests that FVD may be a better indicator of the movement quality within the frames, possibly prioritizing motion over the visual quality of individual frames.

The image generation metrics provided valuable insights into the performance of the image generation model DiT. However, regarding image generation, LPIPS and FID provide a more accurate evaluation metric, as they measure how well the generated videos represent the distribution of the training dataset. In contrast, SSIM and PSNR focus on pixel-by-pixel similarity between generated videos and the videos from the dataset. The results reveal that PSNR and SSIM values for the video generation models are typically higher than those of the image generation model DiT, despite DiT's ability to generate numbers with better visual quality and less disruption. Nevertheless, these image metrics can still be useful for detecting progress in model learning, as demonstrated by Latte v3, which produced mostly black video frames and received the lowest values, even when considering the image metrics SSIM and PSNR.

5.5 Limitations

Due to memory constraints, it was not possible to train the video generation models with larger batch sizes. The various implementations were trained with a batch size of 4, whereas GenTron managed to process 128 videos and SnapVideo processed 2048 videos in a single training step. A larger batch size allows the model to see more video samples

from the dataset before it takes a gradient step. This can have a big impact on the training results. Additionally, because of time constraints, only 100 epochs of training were possible. Further training might improve the current results.

For the diffusion process the cosine noise scheduler was consistently used, as it is a standard choice. Alternative noise schedulers were not tested. However, using sigmoid and exponential schedules could add more noise control and efficiency.

The dataset used in this work was a simple text-to-video dataset, containing black-and-white videos. Each video shows one of the 27 variations of numbers and movements. A more complex dataset might produce different results. Furthermore, the dataset contains text prompts which always follow the same pattern. Reordering the phrases did not have any impact on the results, however, completely rephrasing sentences often resulted in black or blurred images. This is probably because the video generation models had not been trained on such rephrased inputs and were unable to understand the unfamiliar text structures.

CHAPTER 6

Conclusion

Although implementing an image generation model is a difficult task, adapting the model for video generation introduces even more complexity. Even small adjustments, whether in the diffusion process or the selection of architectural components, can significantly impact the quality of the generated video. Effective preprocessing plays a key role in video generation. Extracting relevant information from the input video in the early stages makes it easier for the model to understand and process the content. Using a pre-trained autoencoder to compress the input video into a latent representation significantly improved the performance in video generation. The choice of architecture for video processing in Latte, SnapVideo, and GenTron also had a considerable effect on the results. Handling the temporal and spatial dimensions of the input video separately produced the best outcomes.
The evaluation metrics for image and video generation provided useful insights into the quality of the generated outputs. However, metrics like FID and LPIPS for image generation and FVD for video generation provide a more meaningful evaluation in comparison to SSIM and PSNR, as they measure how well the generated videos align with the training dataset rather than computing pixel-by-pixel similarity.

In conclusion, this work demonstrates that Diffusion Transformer models have the capacity to effectively comprehend and further produce complex visual data, including images and videos. It also shows that, even with limited hardware resources, training a video generation model with a Diffusion Transformer backbone on a simple dataset for just a few epochs can produce promising results.

CHAPTER 7

Appendix

7.1 GenTron Pseudocode

```
def spatialAttention_Block(x, scale_gamma, shift_beta, scale_alpha):
    # reshape for spatial attention
  x = rearrange(x, 'b t s e -> (b t) s e')
    # AdaLN text integration
  x_adaLN = x * (1+scale_gamma) + shift_beta
  x_attn = self_attn(x)
  x = x + scale_alpha * x_attn

  return x

def temporalAttention_Block(x):
    # reshape for temporal attention
  x = rearrange(x, '(b t) s e -> (b s) t e')
  x_attn = self_attn(x)
  x = x + x_attn
  x = rearrange(x, '(b s) t e -> (b t) s e')
  return x

def textCrossAttention_Block(x, y, mask):
  # text cross attention, mask out placeholder tokens
  x_attn = cross_attn(norm(x), y, mask)
  x = x + x_attn
  return x

def aggregationFF_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = norm(x) * (1+ scale_gamma) + shift_beta
  x_ff = feedforward(x)
  x = x + scale_alpha * x_ff
  return x_ff
```

```
class GenTron_Block():
  def forward(x, c, y, maks):
    # batch_size, number_of_frames, sequence_length, embed_dim
    b, t, s, e = x.shape

    # adaLN shift and scale parameters from conditional input c
    g1, b1, a1, g2, b2, a2 = feedforward(c).chunk(6)

    x = spatialAttention_Block(x, g1, b1, a1)

    # Cross attention between frames and text
    x = textCrossAttention_Block(x, y, mask)

    x = temporalAttention_Block(x)

    x = aggregationFF_Block(x, g2, b2, a2)

    return x

class GenTron():
  def forward(x, t, y):
    # (batch_size, channels, number_of_frames, height, width)
    b, c, t, h, w = x.shape
    x = rearrange(x, "b c t h w -> (b t) c h w")

        # patch embedding + positional Encoding
    # (batch_size, number_of_frames, sequence_length, embed_dim)
    x = x_embedder(x) + positional_encoding(x)

    # diffusion timestep
    t = t_embedder(x)

    # text embedder with random dropout for classifier-free guidance
    # mask is used to mask out placeholder tokens
    y, mask = y_embedder(y)

    y_pool = (y * mask).sum(dim=1) / mask.sum(dim=1)

    c = t + y_pool

    for GenTron_Block in GenTron_Block_list:
      x = GenTron_Block(x, c, y, mask)

    x = final_layer(x, c)

    x = unpatchify(x)
    x = rearrange(x, "(b t) c h w -> b c t h w")
    return x
```

7.2 Latte v1 Pseudocode

```
def attention_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = x * (1+scale_gamma) + shift_beta
  x_attn = self_attn(x)
  x = x + scale_alpha * x_attn
  return x

def aggregationFF_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = norm(x) * (1+ scale_gamma) + shift_beta
  x_ff = feedforward(x)
  x = x + scale_alpha * x_ff
  return x_ff

def textCrossAttention_Block(x, y, mask):
  # text cross attention, mask out placeholder tokens
  x_attn = cross_attn(norm(x), y, mask)
  x = x + x_attn
  return x

class Latte_Block():
  def forward(x, c, y, mask, is_first_block):
    # batch_size, number_of_frames, sequence_length, embed_dim
    b, t, s, e = x.shape

    if is_first_block:
            # positional Encoding (spatial)
      x = x + spatial_positional_encoding(x)

    x = rearrange(x, 'b t s e -> (b t) s e')

    # adaLN shift and scale parameters from conditional input c for spatial
                                    dimension
    g1, b1, a1, g2, b2, a2 = feedforward_spatial(c).chunk(6)
    x = attention_Block(x, g1, b1, a1)
    x = aggregationFF_Block(x, g2, b2, a2)


    # Text inseration version 1
    ###########################################################
    # Cross attention between frames and text
    x = textCrossAttention_Block(x, y, mask)
    ###########################################################


    x = rearrange(x, 'b t s e -> (b s) t e')

    if is_first_block:
            # positional Encoding (temporal)
      x = x + temporal_positional_encoding(x)
```

```
    # adaLN shift and scale parameters from conditional input c for temporal
                                        dimension
    g1, b1, a1, g2, b2, a2 = feedforward_temporal(c).chunk(6)
    x = attention_Block(x, g1, b1, a1)
    x = aggregationFF_Block(x, g2, b2, a2)

    return x

class Latte():
  def forward(x, t, y):
    # batch_size, channels, number_of_frames, height, width
    b, c, t, h, w = x.shape
    x = rearrange(x, "b c t h w -> (b t) c h w")

    # patch embedding
    x = x_embedder(x)

    # diffusion timestep
    t = t_embedder(x)

    # text embedder with random dropout for classifier-free guidance
    # mask is used to mask out placeholder tokens
    y, mask = y_embedder(y)

    y_pool = (y * mask).sum(dim=1) / mask.sum(dim=1)

    c = t + y_pool

    for i in range(len(GenTron_Block_list)):
      block = GenTron_Block_list[i]
      if i == 0:
        x = block(x, c, y, mask, True)
      x = block(x, c, y, mask, False)

    x = final_layer(x, c)

    x = unpatchify(x)
    x = rearrange(x, "(b t) c h w -> b c t h w")
    return x
```

7.3 Latte v2 Pseudocode

```
def attention_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = x * (1+scale_gamma) + shift_beta
  x_attn = self_attn(x)
  x = x + scale_alpha * x_attn
  return x

def aggregationFF_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = norm(x) * (1+ scale_gamma) + shift_beta
  x_ff = feedforward(x)
  x = x + scale_alpha * x_ff
  return x_ff

def textCrossAttention_Block(x, y, mask):
  # text cross attention, mask out placeholder tokens
  x_attn = cross_attn(norm(x), y, mask)
  x = x + x_attn
  return x

class Latte_Block():
  def forward(x, c, y, mask, is_first_block):
    # batch_size, number_of_frames, sequence_length, embed_dim
    b, t, s, e = x.shape

    if is_first_block:
            # positional Encoding (spatial)
      x = x + spatial_positional_encoding(x)

    x = rearrange(x, 'b t s e -> (b t) s e')

    # adaLN shift and scale parameters from conditional input c for spatial
                                    dimension
    g1, b1, a1, g2, b2, a2 = feedforward_spatial(c).chunk(6)
    x = attention_Block(x, g1, b1, a1)

    # Text inseration version 2
    ##########################################################
    # Cross attention between frames and text
    x = textCrossAttention_Block(x, y, mask)
    ##########################################################

    x = aggregationFF_Block(x, g2, b2, a2)


    x = rearrange(x, 'b t s e -> (b s) t e')

    if is_first_block:
            # positional Encoding (temporal)
      x = x + temporal_positional_encoding(x)
```

```
    # adaLN shift and scale parameters from conditional input c for temporal
                                        dimension
    g1, b1, a1, g2, b2, a2 = feedforward_temporal(c).chunk(6)
    x = attention_Block(x, g1, b1, a1)
    x = aggregationFF_Block(x, g2, b2, a2)

    return x

class Latte():
  def forward(x, t, y):
    # batch_size, channels, number_of_frames, height, width
    b, c, t, h, w = x.shape
    x = rearrange(x, "b c t h w -> (b t) c h w")

    # x.shape = (batch_size, number_of_frames, sequence_length, embed_dim)
    x = x_embedder(x)

    # diffusion timestep
    t = t_embedder(x)

    # text embedder with random dropout for classifier-free guidance
    # mask is used to mask out placeholder tokens
    y, mask = y_embedder(y)

    y_pool = (y * mask).sum(dim=1) / mask.sum(dim=1)

    c = t + y_pool

    for i in range(len(GenTron_Block_list)):
      block = GenTron_Block_list[i]
      if i == 0:
        x = block(x, c, y, mask, True)
      x = block(x, c, y, mask, False)

    x = final_layer(x, c)

    x = unpatchify(x)
    x = rearrange(x, "(b t) c h w -> b c t h w")
    return x
```

7.4 Latte v3 Pseudocode

```
def attention_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = x * (1+scale_gamma) + shift_beta
  x_attn = self_attn(x)
  x = x + scale_alpha * x_attn
  return x

def aggregationFF_Block(x, scale_gamma, shift_beta, scale_alpha):
    # AdaLN text integration
  x_adaLN = norm(x) * (1+ scale_gamma) + shift_beta
  x_ff = feedforward(x)
  x = x + scale_alpha * x_ff
  return x_ff

def textCrossAttention_Block(x, y, mask):
  # text cross attention, mask out placeholder tokens
  x_attn = cross_attn(norm(x), y, mask)
  x = x + x_attn
  return x

class Latte_Block():
  def forward(x, c, y, mask, is_first_block):
    # batch_size, number_of_frames, sequence_length, embed_dim
    b, t, s, e = x.shape

    if is_first_block:
            # positional Encoding (spatial)
      x = x + spatial_positional_encoding(x)

    x = rearrange(x, 'b t s e -> (b t) s e')

    # adaLN shift and scale parameters from conditional input c for spatial
                                    dimension
    g1, b1, a1, g2, b2, a2 = feedforward_spatial(c).chunk(6)
    x = attention_Block(x, g1, b1, a1)

    # Text inseration version 3
    ############################################################
    # Cross attention between frames and text
    x = textCrossAttention_Block(x, y, mask)
    ############################################################

    x = aggregationFF_Block(x, g2, b2, a2)

    x = rearrange(x, 'b t s e -> (b s) t e')

    if is_first_block:
            # positional Encoding (temporal)
      x = x + temporal_positional_encoding(x)
```

```
    # adaLN shift and scale parameters from conditional input c for temporal
                                        dimension
    g1, b1, a1, g2, b2, a2 = feedforward_temporal(c).chunk(6)
    x = attention_Block(x, g1, b1, a1)

    # Text inseration version 3
    ############################################################
    # Cross attention between frames and text
    x = textCrossAttention_Block(x, y, mask)
    ############################################################
    x = aggregationFF_Block(x, g2, b2, a2)

    return x


class Latte():
  def forward(x, t, y):
    # batch_size, channels, number_of_frames, height, width
    b, c, t, h, w = x.shape
    x = rearrange(x, "b c t h w -> (b t) c h w")

    # x.shape = (batch_size, number_of_frames, sequence_length, embed_dim)
    x = x_embedder(x)

    # diffusion timestep
    t = t_embedder(x)

    # text embedder with random dropout for classifier-free guidance
    # mask is used to mask out placeholder tokens
    y, mask = y_embedder(y)

    y_pool = (y * mask).sum(dim=1) / mask.sum(dim=1)

    c = t + y_pool

    for i in range(len(GenTron_Block_list)):
      block = GenTron_Block_list[i]
      if i == 0:
        x = block(x, c, y, mask, True)
      x = block(x, c, y, mask, False)

    x = final_layer(x, c)

    x = unpatchify(x)
    x = rearrange(x, "(b t) c h w -> b c t h w")
    return x
```

7.5 SnapVideo v1 Pseudocode

```
def crossAttention_Block(x, y):
  x_attn = self_attn(norm(x), y)
  x = x + x_attn
  return x

def selfAttention_Block(x, y):
  x_attn = self_attn(norm(x))
  x = x + x_attn
  return x

def feedForward_Block(x, y):
  x_ff = feedforward(norm(x))
  x = x + x_ff


class SnapVideo_Block():
  def forward(x, latents, c, c_pool, mask):
    # batch_size, number_of_groups, latents_in_group, latents_embed_dim
    b, n, l, e_l = latents.shape

    # batch_size, number_of_groups, patches_in_group, patches_embed_dim
    b, n, p, e_x = x.shape

    latents = rearrange(latents, '(b n) l e -> b (n l) e')
    latents = crossAttention_Block(latents, c, mask)
    latents = feedForward_Block(latents)

    latents = rearrange(latents, 'b (n l) e -> (b n) l e')
    latents = crossAttention_Block(latents, x)
    latents = feedForward_Block(latents)

    latents = einops.rearrange(latents, '(b n) l e -> b (n l) e')
    for self_attn_l, ff_l in global_attn:
      latents = selfAttention_Block(latents)
      latents = feedForward_Block(latents)

    latents = rearrange(latents, '(b n) l e -> b (n l) e')
    x = selfAttention_Block(x, latents)
    x = feedForward_Block(latents)

    return x, latents

class SnapVideo():
  def forward(x, t, y, latents_prev):
    # batch_size, channels, number_of_frames, height, width
    b, c, t, h, w = x.shape
    x = rearrange(x, "b c t h w -> (b t) c h w")

    # x.shape = (batch_size, number_of_frames, sequence_length, embed_dim)
    x = x_embedder(x) + positional_encoding(x)
    # x.shape = (batch_size, number_of_groups, patches_in_groups, embed_dim)
```

```
    x= group(x)

    # Self projection
    if latents_prev is not None:
      latents = latents + latent_prev_ln(self.latent_prev_ff(latents_prev))

    # diffusion timestep
    t = t_embedder(x)

    # text embedder with random dropout for classifier-free guidance
    # mask is used to mask out placeholder tokens
    y, mask = y_embedder(y)
    # add learnable parameters as text registers
    y = torch.cat[y, text_registers]
    # self attention on text
    for i in range(2):
      y = self_attn[i](y)

    c = torch.cat([t, y], dim=1)
    # only mask out empty tokens in text, don't mask timestep
    mask = torch.cat([torch.ones(batch_size, 1), mask])

    for SnapVideo_Block in SnapVideo_Block_list:
      x = SnapVideo_Block(x, latents, c, c_pool, mask)

    x = final_layer(x, c)

    x = ungroup(x)
    x = unpatchify(x)
    x = rearrange(x, "(b t) c h w -> b c t h w")
    return x, latents
```

7.6 SnapVideo v2 Pseudocode

```
def crossAttention_Block(x, y):
  x_attn = cross_attn(norm(x), y)
  x = x + x_attn
  return x

def selfAttention_Block(x, y, c=None):
  x_norm = norm(x)
  if c is not None:
    scale, shift, gate = feedforward(cond).chunk(3)
        # AdaLN text integration
    x_adaLN = (x_norm * (scale + 1)) + shift
    x = x + gate * self_attn(x_adaLN)
  else:
    x = x + self_attn(x_norm)
  return x

def feedForward_Block(x, y, c = None):
  x_norm = norm(x)
  if c is not None:
    scale, shift, gate = feedforward(cond).chunk(3)
        # AdaLN text integration
    x_adaLN = (x_norm * (scale + 1)) + shift
    x = x + gate * feedforward(x_adaLN)
  else:
    x = x + feedforward(x_norm)
  x = x


class SnapVideo_Block():
  def forward(x, latents, c, c_pool, mask):
    # batch_size, number_of_groups, latents_in_group, latents_embed_dim
    b, n, l, e_l = latents.shape

    # batch_size, number_of_groups, patches_in_group, patches_embed_dim
    b, n, p, e_x = x.shape

    latents = rearrange(latents, '(b n) l e -> b (n l) e')
    latents = crossAttention_Block(latents, c, mask)
        # AdaLN text ingegration
    latents = feedForward_Block(latents, c_pool)

    latents = rearrange(latents, 'b (n l) e -> (b n) l e')
    latents = crossAttention_Block(latents, x)
    latents = feedForward_Block(latents)

    latents = einops.rearrange(latents, '(b n) l e -> b (n l) e')
    for self_attn_l, ff_l in global_attn:
            # AdaLN text ingegration
      latents = selfAttention_Block(latents, c_pool)
      latents = feedForward_Block(latents)
```

```
    latents = rearrange(latents, '(b n) l e -> b (n l) e')
    x = selfAttention_Block(x, latents)
    x = feedForward_Block(latents)

    return x, latents

class SnapVideo():
  def forward(x, t, y, latents_prev):
    # batch_size, channels, number_of_frames, height, width
    b, c, t, h, w = x.shape
    x = rearrange(x, "b c t h w -> (b t) c h w")

    # x.shape = (batch_size, number_of_frames, sequence_length, embed_dim)
    x = x_embedder(x) + positional_encoding(x)
    # x.shape = (batch_size, number_of_groups, patches_in_groups, embed_dim)
    x= group(x)

    # Self projection
    if latents_prev is not None:
      latents = latents + latent_prev_ln(self.latent_prev_ff(latents_prev))

    # diffusion timestep
    t = t_embedder(x)

    # text embedder with random dropout for classifier-free guidance
    # mask is used to mask out placeholder tokens
    y, mask = y_embedder(y)
    y_pool = (y * mask).sum(dim=1) / mask.sum(dim=1)
    # add learnable parameters as text registers
    y = torch.cat[y, text_registers]
    # self attention on text
    for i in range(2):
      y = self_attn[i](y)

    c = torch.cat([t, y], dim=1)
    c_pool = y_pool + t

    # only mask out empty tokens in text, don't mask timestep
    mask = torch.cat([torch.ones(batch_size, 1), mask])

    for SnapVideo_Block in SnapVideo_Block_list:
      x = SnapVideo_Block(x, latents, c, c_pool, mask)

    x = final_layer(x, c)

    x = ungroup(x)
    x = unpatchify(x)
    x = rearrange(x, "(b t) c h w -> b c t h w")
    return x, latents
```

Overview of Generative AI Tools Used

ChatGPT (Version GPT-4, OpenAI, December 2024) was utilized to assist in finding synonyms.

List of Figures

1.1 Intuition behind diffusion models 1

2.1 U-Net architecture 22
2.2 Transformer model 23
2.3 Self-attention mechanism 24
2.4 Multi-head attention 25
2.5 Residual connection concept 26
2.6 Vision Transformer architecture 29
2.7 Patch embedding 29
2.8 Spatial and temporal attention methods 30

3.1 DiT preprocessing 36
3.2 DiT architectonic details 37
3.3 MNIST dataset 38
3.4 Spatial and temporal dimension modeling 39
3.5 GenTron preprocessing 41
3.6 GenTron architectonic details 43
3.7 Latte preprocessing 44
3.8 Latte architectonic details 46
3.9 SnapVideo preprocessing 48
3.10 SnapVideo architectonic details 49
3.11 Latte Text integration model 1 52
3.12 Latte text integration model 2 53
3.13 Latte text integration model 2 54
3.14 SnapVideo text integration model 1 55
3.15 SnapVideo text integration model 2 56
3.16 MovingMNIST dataset 57

4.1 DiT generated images 60
4.2 Sampling results of GenTron 61
4.3 Sampling results of Latte v1 63
4.4 Sampling results of Latte v2 63
4.5 Sampling results of Latte v3 64
4.6 Sampling results of SnapVideo v1 64

4.7 Sampling results of SnapVideo v2 65

List of Tables

2.1 Overview Notations and Terminologies used to describe the diffusion model 7

4.1 Training details of the DiT model 60
4.2 Evaluation results of the DiT model trained on the MNIST dataset (100 epochs) 61
4.3 Training details of video generation models 62
4.4 Evaluation results of the video generation models trained on the Moving MNIST dataset (100 epochs) 65

List of Algorithms

2.1 DDPM Training 13

2.2 DDPM Sampling 14

2.3 DDPM Training – Classifier-Free Guidance 20

2.4 Sampling – Classifier-Free Guidance 21

2.5 Positional Encoding 27

3.1 Sampling DDMP – Clamp x_0 38

3.2 DDPM Training – Classifier-Free Guidance, Variance learning 44

3.3 DDPM Sampling – Classifier-Free Guidance, Variance learning 45

3.4 DDPM Training – Classifier-Free Guidance, Variance learning, Latent Self-Conditioning 50

3.5 DDPM Sampling – Classifier-Free Guidance, Variance learning, Latent Self-Conditioning 51

Bibliography

[And82] Brian D.O. Anderson. Reverse-time Diffusion Equation Models. *Stochastic Processes and their Applications*, 12(3):313–326, May 1982.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization, July 2016. arXiv:1607.06450 [stat]. URL: http://arxiv.org/abs/1607.06450.

[BPH+24] Tim Brooks, Bill Peebles, Connor Holmes, Will DePue, Yufei Guo, Li Jing, David Schnurr, Joe Taylor, Troy Luhman, Eric Luhman, Clarence Ng, Ricky Wang, and Aditya Ramesh. Video Generation Models as World Simulators. 2024. URL: https://openai.com/research/video-generation-models-as-world-simulators.

[CKS23] Ziyi Chang, George Alex Koulieris, and Hubert P. H. Shum. On the Design Fundamentals of Diffusion Models: A Survey, October 2023. arXiv:2306.04542 [cs]. URL: http://arxiv.org/abs/2306.04542.

[CL23] Ting Chen and Lala Li. FIT: Far-reaching Interleaved Transformers, May 2023. arXiv:2305.12689 [cs]. URL: http://arxiv.org/abs/2305.12689.

[CXR+23] Shoufa Chen, Mengmeng Xu, Jiawei Ren, Yuren Cong, Sen He, Yanping Xie, Animesh Sinha, Ping Luo, Tao Xiang, and Juan-Manuel Perez-Rua. GenTron: Diffusion Transformers for Image and Video Generation, 2023. arXiv:2312.04557 [cs]. URL: https://arxiv.org/abs/2312.04557.

[CZ17] Joao Carreira and Andrew Zisserman. Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4724–4733, Honolulu, HI, July 2017. IEEE.

[CZH23] Ting Chen, Ruixiang Zhang, and Geoffrey Hinton. Analog Bits: Generating Discrete Data using Diffusion Models with Self-Conditioning, March 2023. arXiv:2208.04202 [cs]. URL: http://arxiv.org/abs/2208.04202.

[DBK+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale, 2020. arXiv:2010.11929 [cs]. URL: https://arxiv.org/abs/2010.11929.

[DBKP24] Nicolas Dufour, Victor Besnier, Vicky Kalogeiton, and David Picard. Don't Drop Your Samples! Coherence-aware Training Benefits Conditional Diffusion, May 2024. arXiv:2405.20324 [cs]. URL: http://arxiv.org/abs/2405.20324.

[DN21] Prafulla Dhariwal and Alex Nichol. Diffusion Models Beat GANs on Image Synthesis, June 2021. arXiv:2105.05233 [cs]. URL: http://arxiv.org/abs/2105.05233.

[DOMB24] Timothée Darcet, Maxime Oquab, Julien Mairal, and Piotr Bojanowski. Vision Transformers Need Registers, April 2024. arXiv:2309.16588 [cs]. URL: http://arxiv.org/abs/2309.16588.

[GWY+22] Yunhui Guo, Chaofeng Wang, Stella X. Yu, Frank McKenna, and Kincho H. Law. AdaLN: A Vision Transformer for Multidomain Learning and Predisaster Building Information Extraction from Images. *Journal of Computing in Civil Engineering*, 36(5):04022024, September 2022.

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851. Curran Associates, Inc., 2020.

[HXZ+24] Guantian Huang, Bixuan Xia, Haoming Zhuang, Bohan Yan, Cheng Wei, Shouliang Qi, Wei Qian, and Dianning He. A Comparative Analysis of U-Net and Vision Transformer Architectures in Semi-Supervised Prostate Zonal Segmentation. *Bioengineering*, 11(9):865, August 2024.

[JFC22] Allan Jabri, David Fleet, and Ting Chen. Scalable Adaptive Computation for Iterative Generation, 2022. arXiv:2212.11972 [cs]. URL: https://arxiv.org/abs/2212.11972.

[MSS+24] Willi Menapace, Aliaksandr Siarohin, Ivan Skorokhodov, Ekaterina Deyneka, Tsai-Shien Chen, Anil Kag, Yuwei Fang, Aleksei Stoliar, Elisa Ricci, Jian Ren, and Sergey Tulyakov. Snap Video: Scaled Spatiotemporal Transformers for Text-to-Video Synthesis, February 2024. arXiv:2402.14797 [cs]. URL: http://arxiv.org/abs/2402.14797.

[MWJ+24a] Xin Ma, Yaohui Wang, Gengyun Jia, Xinyuan Chen, Ziwei Liu, Yuan-Fang Li, Cunjian Chen, and Yu Qiao. Latte: Latent Diffusion Transformer

for Video Generation, January 2024. arXiv:2401.03048 [cs]. URL: http://arxiv.org/abs/2401.03048.

[MWJ+24b] Xin Ma, Yaohui Wang, Gengyun Jia, Xinyuan Chen, Ziwei Liu, Yuan-Fang Li, Cunjian Chen, and Yu Qiao. Latte: Latent Diffusion Transformer for Video Generation. *GitHub*, 2024. URL: https://github.com/Vchitect/Latte.

[ND21] Alex Nichol and Prafulla Dhariwal. Improved Denoising Diffusion Probabilistic Models, February 2021. arXiv:2102.09672 [cs]. URL: http://arxiv.org/abs/2102.09672.

[Nic21] Alex Nichol. Improved DDPM. *GitHub*, February 2021. URL: https://github.com/openai/improved-diffusion/.

[PX22] William Peebles and Saining Xie. Scalable Diffusion Models with Transformers. *GitHub*, 2022. URL: https://github.com/facebookresearch/DiT.

[PX23] William Peebles and Saining Xie. Scalable Diffusion Models with Transformers, March 2023. arXiv:2212.09748 [cs]. URL: http://arxiv.org/abs/2212.09748, doi:10.48550/arXiv.2212.09748.

[RDN+22] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical Text-Conditional Image Generation with CLIP Latents, April 2022. arXiv:2204.06125 [cs]. URL: http://arxiv.org/abs/2204.06125.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015. arXiv:1505.04597 [cs]. URL: http://arxiv.org/abs/1505.04597.

[Rog22] Alex Rogozhnikov. Einops: Clear and Reliable Tensor Manipulations with Einstein-like Notation. In *International Conference on Learning Representations*, 2022.

[SCS+22] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, Tim Salimans, Jonathan Ho, David J. Fleet, and Mohammad Norouzi. Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding, May 2022. arXiv:2205.11487 [cs]. URL: http://arxiv.org/abs/2205.11487.

[SDWMG15] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics, November 2015. arXiv:1503.03585 [cs]. URL: http://arxiv.org/abs/1503.03585.

[SJE+23] Javier Selva, Anders S. Johansen, Sergio Escalera, Kamal Nasrollahi, Thomas B. Moeslund, and Albert Clapés. Video Transformers: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(11):12922–12943, November 2023. arXiv:2201.05991 [cs]. URL: http://arxiv.org/abs/2201.05991.

[SL23] Inga Strümke and Helge Langseth. Lecture Notes in Probabilistic Diffusion Models, December 2023. arXiv:2312.10393 [cs]. URL: http://arxiv.org/abs/2312.10393.

[SSDK+21] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-Based Generative Modeling through Stochastic Differential Equations, February 2021. arXiv:2011.13456 [cs]. URL: http://arxiv.org/abs/2011.13456.

[VSP+23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762.

[Zha23] Zhe George Zhang. *Fundamentals of Stochastic Models*. CRC Press, Boca Raton, 1st edition, May 2023.