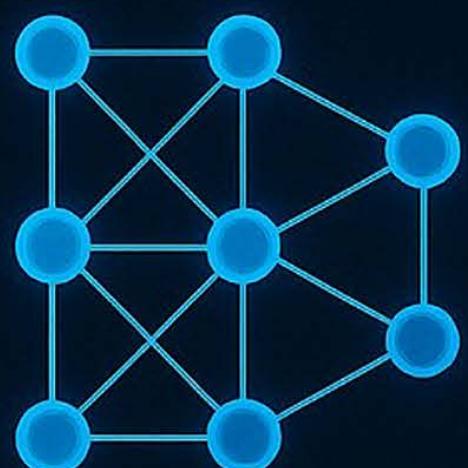


GENERATIVE FOUNDATION MODELS

A Comprehensive Beginner's Handbook



JUNBO JACOB LIAN
NORTHWESTERN UNIVERSITY

Generative Foundation Models

A Comprehensive Beginner's Handbook

Junbo Jacob Lian

Northwestern University

Jacoblian@u.northwestern.edu

Abstract

This handbook provides a comprehensive, hands-on exploration of modern generative foundation models. We cover the theoretical foundations and practical implementations of key architectures including the Transformer and Vision Transformer for sequence and image modeling, the Mamba model featuring linear-time sequence modeling through selective state space mechanisms, the U-Net convolutional network for image synthesis, Denoising Diffusion Probabilistic Models for generative modeling with forward and reverse stochastic processes, the Diffusion Transformer (DiT) which integrates transformer architectures into diffusion models, the Retentive Network (RetNet) that provides efficient attention-free sequence modeling, the Latent Diffusion Model (LDM) enabling high-resolution image generation via autoencoding, and text-to-3D generation techniques such as DreamFusion and Magic3D. Each section begins with an intuitive overview and then delves into rigorous mathematical formulations. We include PyTorch-style pseudo-code listings to illustrate core algorithms and architectures. Consistent notation is maintained throughout (vectors in bold, scalars in italics, distributions in sans-serif). Finally, each section concludes with reading questions to encourage deeper reflection on the presented material.

Table of Contents

Contents

1	Introduction	1
2	Transformers for Sequence Modeling	2
3	Vision Transformer (ViT) for Image Representation	5
4	Mamba: Linear-Time State Space Model for AIGC	7
5	U-Net Architecture for Image Generation	11
6	Denoising Diffusion Probabilistic Models (DDPM)	13
7	Diffusion Transformer (DiT)	16
8	Retentive Networks for Sequence Modeling	19
9	Latent Diffusion Models (LDM)	21
10	Text-to-3D Generation: DreamFusion and Magic3D	23
11	Conclusions	26

1 Introduction

Generative foundation models are a class of machine learning models that can learn to generate complex data (such as text, images, or 3D content) that closely mimics real-world data distributions. These models are called *foundation models* because they serve as a base for a wide range of downstream tasks, often after fine-tuning. They are trained on large-scale datasets and capture high-level abstractions, enabling them to synthesize new content. In recent years, such models have grown dramatically in scale and capability, exemplified by large language models for text and state-of-the-art image and 3D generation models.

In this handbook, we focus on several influential architectures and methods in the domain of generative models:

- **Transformers and Vision Transformers (ViT)** for sequence and image generation. Transformers [1] introduced the self-attention mechanism that has become a backbone of modern generative models, enabling models to capture long-range dependencies in data [2].
- **Mamba**, a linear-time state space model designed for efficient sequence modeling in generative tasks [3].
- **U-Net Convolutional Networks**, originally developed for image segmentation [11], now widely used as the architecture for generative models in diffusion-based image synthesis.
- **Denoising Diffusion Models (DDPM)** [12], a class of generative models that progressively destroy and then recover data, yielding high-quality images through an iterative denoising process.
- **Diffusion Transformers (DiT)** [13], an architecture that replaces the U-Net with a transformer backbone for diffusion models, marrying the strengths of attention with diffusion.
- **Retentive Networks (RetNet)**, which provide efficient attention-free sequence modeling via exponential decay-based retention [14].
- **Latent Diffusion Models (LDM)** [15], which perform diffusion in a learned latent space (through a Variational Autoencoder) to enable high-resolution generation efficiently, as exemplified by the Stable Diffusion model.
- **Text-to-3D Generation** techniques (e.g., DreamFusion [16] and Magic3D [17]) which leverage 2D diffusion priors to generate 3D content from text prompts via optimization and score distillation.

Each section of this handbook is structured to provide both a broad conceptual understanding and a detailed theoretical exposition. We maintain consistent notation: vectors (e.g., data samples, latent representations) are denoted in bold (such as \mathbf{x}), scalars in italic (x), and probability distributions or special functions in sans-serif font (e.g., N for a Normal distribution). Equations are kept to a necessary complexity and explained in words. Code examples in Python-like pseudocode are provided to illustrate how these models can be implemented in practice.

Through this handbook, a reader should gain insight into the mechanisms that enable generative models to create new content and develop an intuition for how these components (attention layers, convolutional networks, diffusion processes, etc.) work in concert. We also pose reading questions at the end of each section to facilitate reflection and deeper engagement with the material.

Notation and Preliminaries

Before diving into the core content, we summarize key notational conventions and preliminary concepts:

- \mathbb{R}^n denotes the space of n -dimensional real-valued vectors. We write $\mathbf{x} \in \mathbb{R}^n$ for a column vector and \mathbf{X} for a matrix.
- For stochastic processes, $p(\cdot)$ or $p_\theta(\cdot)$ denotes a probability distribution (with θ indicating dependence on model parameters). We often use $N(\mu, \sigma^2)$ for a Gaussian distribution with mean μ and variance σ^2 .
- The expected value of a random variable X is denoted $\mathbb{E}[X]$.
- When dealing with time-step indexed sequences (as in diffusion models), we use x_0 to denote the original data sample and x_t for its state at step t of a process.
- The symbol \mathbf{I} denotes the identity matrix. Notation like $\|\mathbf{x}\|^2$ represents the Euclidean norm squared of vector \mathbf{x} .

We assume familiarity with basic concepts of linear algebra (vectors, matrices), calculus, probability (random variables, distributions), and machine learning (neural networks, optimization). Wherever necessary, we briefly review advanced concepts within the respective sections.

2 Transformers for Sequence Modeling

Overview: The Transformer [1] is a sequence-to-sequence model architecture that has revolutionized natural language processing and beyond. Unlike recurrent networks, Transformers process sequences with a self-attention mechanism that allows each position in the sequence to attend to every other position. This enables capturing long-range dependencies without sequential gating or recurrence. Key innovations of the Transformer include *multi-head self-attention*, *positional encoding*, *residual connections* with layer normalization, and an architecture amenable to massive parallelization and scaling.

At a high level, a Transformer model consists of an encoder (and optionally a decoder for tasks like translation, though for many generative tasks a decoder alone as in GPT models is used). Each layer of the Transformer encoder contains a multi-head self-attention sublayer and a position-wise feedforward sublayer, with residual connections around each. The decoder layers similarly have self-attention, encoder-decoder attention, and feedforward sublayers. In generative settings, such as language modeling, the Transformer is often used in a decoder-only configuration with causal self-attention (each position can only attend to earlier positions, to prevent information leak from future tokens). In image generation, Vision Transformers (ViT) apply the Transformer architecture to image patches rather than word tokens.

Multi-Head Attention: At the core of the Transformer is the scaled dot-product attention mechanism. Given a set of queries Q , keys K , and values V (usually these are linear projections of the input sequence or previous layer outputs), the attention mechanism computes weighted sums of the values where the weights come from query-key similarity:

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

where d_k is the dimensionality of the key (and query) vectors, used to scale the dot products. The softmax operation ensures the weights are positive and sum to 1 across the sequence length of K . In practice, we compute multiple attention “heads” in parallel, each with its own learned projection matrices for Q, K, V . If we have h heads, and denote the i -th head’s projections as W_i^Q, W_i^K, W_i^V , then the multi-head attention output is:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O,$$

with $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$. The matrix W^O is a learned projection that maps the concatenated heads back to the model’s embedding dimension.

One advantage of multiple heads is that each head can focus on different patterns or positions in the input. For example, in language, one head might attend to syntactic dependencies while another attends to long-range semantic connections. Multi-head attention allows the model to jointly attend to information from different representation subspaces.

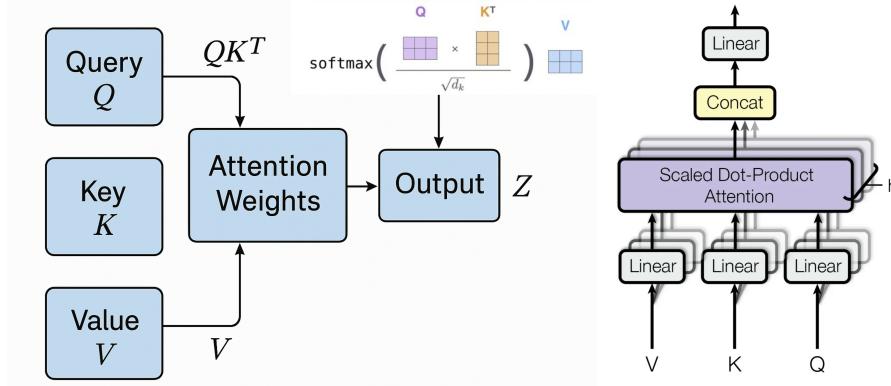


Figure 1: Illustration of scaled dot-product attention. The query Q and keys K produce an attention weight matrix via QK^T (scaled by $1/\sqrt{d_k}$, not shown in diagram). After softmax normalization, these weights are applied to the values V to produce output Z . In a multi-head setup, this process is replicated h times with independent learned projections.

Positional Encoding: Because a Transformer has no built-in notion of sequence order (unlike an RNN that processes tokens sequentially), we must inject position information. In the original Transformer, this is done via fixed sinusoidal positional encodings added to the input embeddings. For a token position pos in the sequence and embedding index i (starting from 0), the encoding is defined as:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right),$$

where d_{model} is the model’s embedding dimension. These sinusoids of varying frequencies allow the model to determine the relative positions of tokens. Alternatively, many modern implementations use learned positional embeddings (trainable vectors) of the same dimension.

Residual Connections and Layer Normalization: Each sublayer (attention or feed-forward) in the Transformer is wrapped with a residual connection [19] and followed by layer normalization. If we denote the sublayer function as $\text{sublayer}(x)$ (which could be multi-head attention or a feed-forward network), and the input to the sublayer as x , then the output of the Transformer sublayer block is:

$$\text{LayerNorm}(x + \text{sublayer}(x)),$$

meaning the sublayer’s output is added to the original input x (this is the skip connection or residual add), and then a layer normalization is applied to stabilize training and reduce covariate shift. These residual connections are critical for training deep architectures by mitigating vanishing gradient issues and enabling information to flow unchanged if needed.

Causal Masking (for autoregressive generative modeling): In tasks like language generation or any scenario where the model should not peek at future information (to genuinely generate rather than copy), a causal mask is applied in self-attention. The mask is a matrix (of shape sequence length \times sequence length) that marks forbidden connections. For a target position i and a source position j , the mask entry M_{ij} is 0 if $j > i$ (meaning token j comes after i and should not be attended to) and 1 otherwise. When computing attention, we add a large negative value (e.g., -10^9) to the attention logits QK^T wherever $M_{ij} = 0$ before the softmax. This effectively zeroes out attention from future tokens. Causal masking ensures the Transformer can be used for step-by-step generation (one token at a time).

Scaling Laws: Large Transformers have demonstrated that increasing model size (number of parameters), dataset size, and compute leads to predictable improvements in performance [18]. Empirical *scaling laws* show that for language models, the loss scales approximately as a power-law with respect to model parameters and data, until bottlenecked by one of those factors. This means that, in principle, making Transformers bigger and training on more data tends to yield better generative models, which has driven the trend of building ever larger models (e.g., GPT-3, with 175 billion parameters). However, such scaling must be balanced with efficient training and inference techniques.

Transformer Applications in Generative Modeling: While originally developed for tasks like translation, Transformers are now the de facto choice for generative text modeling (GPT series, etc.). They have also been applied in other modalities:

- In vision, the Vision Transformer (ViT) [2] treats an image as a sequence of patches (e.g., 16x16 pixel patches) and processes it similarly to words, enabling image classification and also generative image modeling when combined with appropriate training objectives.
- In audio, Transformers can generate coherent long-range audio by treating spectrograms or waveforms as sequences.
- For multimodal tasks, Transformers allow easy integration of different sources (e.g., text and image) via cross-attention mechanisms, which we will discuss later in the context of diffusion models and text-to-image generation.

The Transformer provides a foundation we will build upon in later sections, particularly when discussing how cross-attention is used to condition image generation on text, and how Transformer-based architectures can replace convolution-based networks in diffusion models.

```

1 import torch
2 import torch.nn.functional as F
3
4 def scaled_dot_product_attention(Q, K, V, mask=None):
5     d_k = Q.shape[-1] # dimension of keys and queries
6     # Compute raw attention scores via dot-product and scale them
7     scores = torch.matmul(Q, K.transpose(-2, -1)) / (d_k ** 0.5)
8     # Apply mask (if provided) to enforce causal structure
9     if mask is not None:
10         scores = scores.masked_fill(mask == 0, float('-inf'))
11     # Normalize scores to probabilities
12     attention_weights = F.softmax(scores, dim=-1)

```

```

13     # Generate output by weighted combination of values
14     return torch.matmul(attention_weights, V)
15
16 # Example usage with batch_size=B, sequence_length=T, embedding_dim=D
17 B, T, D = 2, 10, 64
18 Q = K = V = torch.randn(B, T, D)
19 mask = torch.tril(torch.ones(T, T)) # causal mask ensuring autoregressive property
20 output = scaled_dot_product_attention(Q, K, V, mask)

```

Listing 1: PyTorch-style pseudocode implementation of scaled dot-product attention for a single attention head. This code demonstrates computing raw attention scores from queries (Q), keys (K), and values (V), applying an optional causal mask to prevent attention to future tokens, scaling by the square root of the key dimension for numerical stability, normalizing the scores into probabilities via a softmax, and finally generating weighted sum outputs.

Reading Questions:

1. Why does multi-head attention use an $\frac{1}{\sqrt{d_k}}$ scaling factor for the dot-product scores, and what could happen if we omitted it for large d_k ?
2. How do residual connections and layer normalization contribute to the stable training of Transformers, especially as models become deeper and larger?
3. In what ways might learned positional embeddings be advantageous over fixed sinusoidal embeddings, and vice versa?
4. Consider the transformer scaling laws. What are the practical limitations of indefinitely scaling up model size and data, and how might future research address these challenges?

3 Vision Transformer (ViT) for Image Representation

Overview: The Vision Transformer (ViT) [2] extends the Transformer architecture to image data by splitting an image into a sequence of patches. Each patch (for example, a 16×16 pixel block) is flattened and linearly projected to serve as an input token. By treating patches similarly to words in a sentence, ViT enables the application of Transformers (which were developed for sequential data) to images. This approach has proven that purely attention-based architectures can match or exceed convolutional neural networks (CNNs) on image classification tasks given sufficient data and compute. As a foundation model, ViT can also be adapted for generative tasks, such as image generation or image completion, by training it with appropriate objectives (e.g., as part of a VQ-VAE or autoregressive image transformer).

Patch Embedding and Tokens: Consider an image $x \in \mathbb{R}^{H \times W \times C}$ (height H , width W , C color channels, typically $C = 3$ for RGB). We choose a patch size p (e.g., 16). The image is divided into $\frac{H}{p} \times \frac{W}{p}$ patches, each of shape $p \times p \times C$. We flatten each patch into a vector of length p^2C , then multiply by a learned weight matrix to project it into the Transformer embedding dimension d . This yields a sequence of N tokens where $N = \frac{H}{p} \times \frac{W}{p}$ (for example, a 256×256 image with $p = 16$ yields $N = 256$ tokens). In addition to these, a special *class token* (for classification tasks) or a *global token* can be prepended to the sequence to aggregate global information.

Formally, let $x_{(i,j)}$ be the patch at grid position (i, j) of the image (where $i = 1 \dots H/p$, $j = 1 \dots W/p$). We define the token embedding:

$$z_{ij} = W_p \cdot \text{flatten}(x_{(i,j)}) + b_p ,$$

where W_p is the patch projection matrix and b_p is a bias. The collection of all z_{ij} (and possibly an extra class token z_{cls}) forms the input sequence to a standard Transformer encoder. Positional encodings (learned or sinusoidal) are added to these patch embeddings to retain spatial information.

ViT Architecture: The architecture inside the ViT is almost identical to the original Transformer encoder: a series of Transformer blocks (multi-head self-attention + MLP with GeLU non-linearity, each with residual connections and layer norm). One minor difference is that ViT often uses *layer normalization before* the attention and MLP sublayers (a Pre-LN Transformer), which has been found to improve training at scale, as opposed to the original Post-LN in Vaswani et al. The output corresponding to the class token can be used for classification. For generative modeling, one could train a ViT to output image patch probabilities or use a decoder structure to sequentially generate patches.

Image Generation with Transformers: While CNNs and diffusion models currently dominate image generation, there has been work on using transformer architectures to generate images. For example, an autoregressive image transformer can generate an image pixel by pixel (or patch by patch) by treating the image as a long sequence. ViT can serve as the backbone for such models, albeit full autoregressive generation over pixels is extremely slow for high resolutions. Hybrid approaches use ViT for global structure and another mechanism for local detail, or utilize ViT within a diffusion model (as we will see with Diffusion Transformers).

However, Vision Transformers appear in generative modeling in another way: as part of *latent diffusion*, where an image is first encoded into a lower-dimensional latent (often using a CNN-based encoder), and then a ViT-like model is used to model the distribution of these latents.

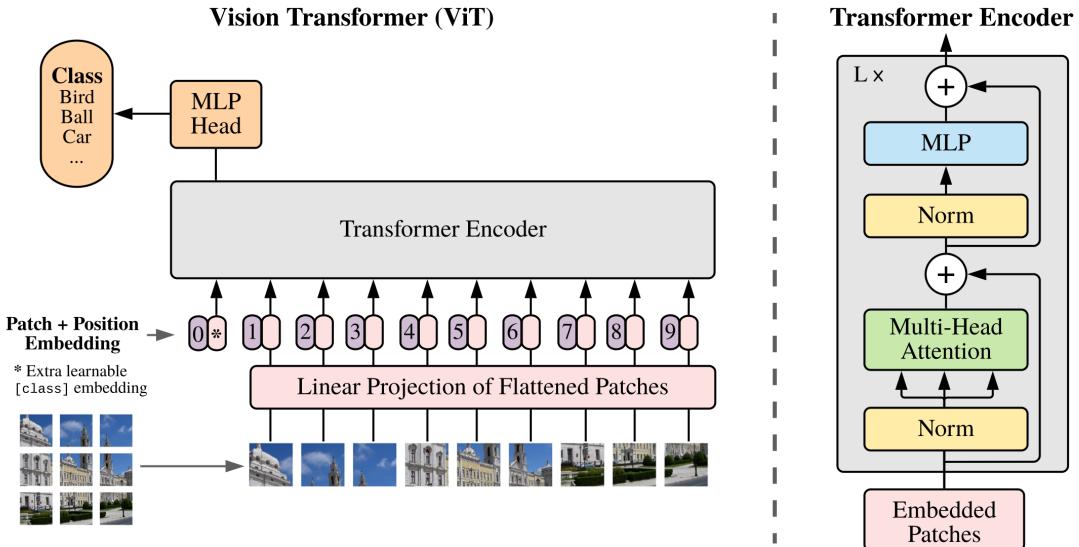


Figure 2: Vision Transformer patch embedding. The image is divided into patches (e.g., 3×3 pixels each), flattened, and linearly projected to form input tokens for a Transformer. A special class token (not shown in placeholder) can be added for classification outputs. The Transformer learns global image relationships through self-attention over these patch tokens.

Scaling and Data Requirements: ViTs have been found to require large datasets or pre-training to outperform CNNs, as they lack certain inductive biases (like translation equivariance) that CNNs have. However, when sufficient data is available (e.g., the JFT-300M dataset in the

original ViT paper), they excel and can learn those patterns from data. In the context of generative models, a ViT might be pre-trained on large image sets and then fine-tuned for generation, or trained end-to-end for generation if combined with techniques like masked image modeling (e.g., iGPT) or diffusion as mentioned.

Position Encoding in Images: The 2D nature of images means that 2D positional encodings could be used (two separate sine-cosine encodings for height and width combined, or learned 2D embeddings). ViT typically flattens 2D positions into one dimension and uses a 1D positional encoding since the patches are lined up in sequence (for example row by row through the image). Some variants use relative positional embeddings or more advanced schemes to better capture locality.

In summary, Vision Transformers demonstrate that the Transformer architecture is flexible enough to handle non-text modalities, and they set the stage for purely transformer-based generative models in vision. Next, we will discuss U-Net, which provides an alternate (convolutional) approach that has been very successful, particularly as the backbone for diffusion models.

Reading Questions:

1. Why might a Vision Transformer require more data to achieve good performance compared to a convolutional neural network on the same task?
2. How does the patch size p affect the ViT model in terms of sequence length and computational cost? What are the trade-offs in choosing a smaller vs larger patch size?
3. Consider how you would adapt a ViT for image generation rather than classification. What additional components or training strategies might you need (e.g., an autoregressive decoder, a diffusion process, etc.)?
4. How could you incorporate locality or translational invariance into a ViT-based generative model to improve its ability to generate high-fidelity local details?

4 Mamba: Linear-Time State Space Model for AIGC

Overview: All big generative models or "foundation models" nowadays rely on the Transformer architecture and its self-attention mechanism [1]. However, attention has $O(T^2)$ cost with sequence length T , posing challenges for very long sequences in text, audio, etc. Many alternative architectures with sub-quadratic complexity have been studied, including linearized attention [5], convolutional networks with gating [6], recurrent models, and Structured State Space Models (SSMs) [8]. These approaches promise linear time complexity, but until recently they lagged behind Transformers on tasks like natural language processing. The *Mamba* model by Gu and Dao [3] addresses a key weakness of prior SSMs — their inability to perform content-based reasoning with discrete tokens — by introducing a *Selective State Space Model (SSSM)* mechanism. Mamba forgoes explicit attention entirely, yet achieves comparable or superior performance to Transformers in AI-Generated Content (AIGC) tasks such as language and audio generation.

Selective SSM Mechanism: In a standard SSM layer (e.g. S4 [8]), the hidden state update is linear and time-invariant: $h_t = A h_{t-1} + B x_t$, $y_t = C^\top h_t$, with A, B, C fixed matrices. Mamba generalizes this to a **Selective SSM** where A , B , and C vary at each time step as learned functions of the input. Formally, for each token x_t in the sequence, Mamba computes input-dependent parameters $A_t \in \mathbb{R}^{N \times N}$, $B_t \in \mathbb{R}^N$, and $C_t \in \mathbb{R}^N$ (with N the state dimension), and updates the state $h_t \in \mathbb{R}^N$ as:

$$h_t = A_t h_{t-1} + B_t x_t, \quad y_t = C_t^\top h_t, \quad (1)$$

where y_t is the output (e.g. logits or feature vector) at position t . By making the state dynamics (A_t, B_t, C_t) depend on x_t (through small feed-forward networks or linear projections), the model can *selectively* decide how much of the new information x_t to incorporate or how much of the past state to retain at each step (i.e. it can selectively propagate or forget information based on content) [3]. This effectively implements a content-sensitive gating mechanism akin to an RNN’s adaptive memory, addressing the “content-based reasoning” deficiency of earlier SSMs. For example, if x_t is deemed important, Mamba can choose A_t with smaller values (more decay of h_{t-1}) and a larger B_t influence, inserting the new content into the state; if x_t is unimportant, A_t may preserve most of h_{t-1} (like a forget gate ignoring irrelevant input). Unlike self-attention, which fully recombines token representations each step, the SSSM update keeps a running compressed representation (the hidden state) that evolves over time, making Mamba fundamentally a recurrent model (with hidden size N often termed the *state expansion* factor).

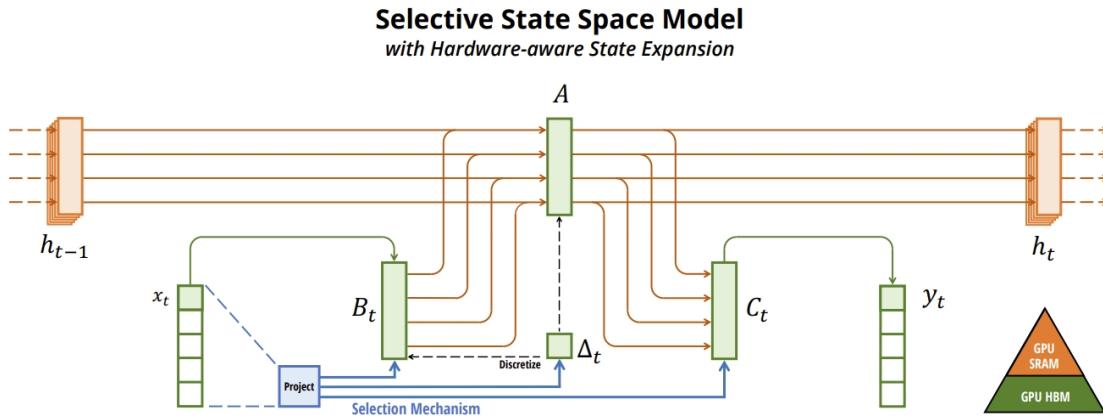


Figure 3: Selective State Space Model Architecture in Mamba. At each time step t , the model projects the input x_t to compute state update parameters (A_t, B_t, C_t , and an internal step size Δ_t). The previous state h_{t-1} is transformed by A_t (an $N \times N$ diagonal or structured matrix) and combined with the input influence $B_t x_t$ to yield the new state h_t . The output y_t is produced by projecting h_t via C_t . The *selection mechanism* (blue components) uses the content of x_t to determine how information is propagated or forgotten, allowing content-dependent sequence processing. Hardware-efficient design is employed (e.g. partitioning state memory across high-bandwidth on-chip SRAM vs. off-chip HBM) to achieve fast parallel computation.

Advantages for Generative Tasks: Mamba’s architecture provides several advantages in AIGC applications. First, it runs in **linear time** $O(T \cdot N)$ per layer (with T sequence length and N state size), eliminating the quadratic $O(T^2)$ bottleneck of attention for long contexts. This makes it viable for extremely long sequences (the authors demonstrate training on sequences up to $T = 10^6$ tokens) [3]. Memory usage also scales linearly, and generation can be done in a streaming fashion: at each step the model only needs to maintain the current hidden state h_t (and perhaps a small number of past states for numerical stability), rather than a full attention buffer of past keys/values. This means **causal generation** with Mamba is highly efficient – comparable to an RNN – requiring no growing memory cache as output length increases. In fact, Mamba achieved about $5\times$ higher decoding throughput than a Transformer of similar size in benchmarks [3]. Second, by virtue of the selective gating, Mamba can handle complex long-range dependencies and discrete reasoning tasks that static SSMs struggled with. It effectively bridges the gap between pure recurrence and

attention: although it does not explicitly compute pairwise token interactions, the input-dependent state update allows important information (like a key token or pattern) to influence the state and thus later outputs, while irrelevant information can be filtered out. Empirically, a 3 billion-parameter Mamba model outperforms a Transformer of the same size on language modeling and even matches the performance of a Transformer twice as large [3], demonstrating excellent modeling capacity for text generation. Similarly, Mamba models have shown state-of-the-art or competitive results on modalities like audio (long raw waveform modeling) and genomics [3], indicating its generality as a sequence backbone beyond NLP.

Comparison to Transformers: Compared to the Transformer, which relies on self-attention to mix information across positions, Mamba offers a different trade-off. Transformers provide direct *content-based interactions* between any pair of tokens via attention scores, which is powerful but expensive. Mamba restricts interactions to pass through the recurrent state, which acts as a compressed memory of the past. This makes Mamba *fully causal and order-dependent* (no exchange of information out-of-order), which naturally suits autoregressive generation. While a Transformer’s attention can flexibly attend to specific relevant tokens from the entire history at each step, Mamba must encode such relevance in its state representation. In practice, the selective SSM mechanism is rich enough to achieve similar outcomes: e.g. induction-style reasoning (finding a pattern in the context and recalling it later) can be handled by the state dynamics learning to preserve a key token and propagate it until needed. Another difference is in parallelization: Transformers can attend to all positions in parallel during training, whereas a naive recurrent model processes tokens sequentially. Mamba overcomes this with a *hardware-aware parallel scan algorithm* that partitions the sequence and updates states in parallel blocks, achieving high training throughput despite the recurrence [3]. Still, training a Mamba can be slightly less efficient on GPUs optimized for matrix multiplications (similar to the issues faced by recurrent networks before the Transformer era), since attention operations can be batched into large matrix multiplies, whereas Mamba’s stepwise computation is less amenable to off-the-shelf GPU routines. Ongoing research by the Mamba authors (e.g. the follow-up *State Space Dual* model [10]) aims to further close this gap by re-formulating the state updates as matrix operations. Once trained, however, Mamba is extremely efficient for inference and generation, especially for long sequences where it significantly outpaces attention-based models.

PyTorch-Style Implementation: Below is a simplified pseudo-code for a single Mamba layer in an autoregressive setting, illustrating the selective state update loop:

```

1 # Pseudocode for one Mamba layer (conceptual, not raw PyTorch)
2
3 def mamba_layer_forward(X, h0):
4     """
5         X : sequence of length T, each token embedding in R^{d_model}
6         h0 : initial hidden state in R^{N} (e.g., zeros)
7         W_A : Linear(d_model -> 1)      # decay gate projection
8         W_B : Linear(d_model -> N)      # additive input projection
9         W_C : Linear(d_model -> N)      # output projection
10    """
11    h = h0          # initial recurrent hidden state
12    Y = []          # list to store outputs at each step
13
14    for t in range(len(X)):           # iterate over the sequence positions
15        x_t = X[t]                  # current token embedding
16
17        # Compute content-dependent exponential decay term (0 < a_t < 1)
18        a_t = exp(-softplus(W_A(x_t)))
19

```

```

20      # Compute additive contribution to the hidden state
21      b_t = W_B(x_t)

22
23      # Update hidden state (selective state update)
24      h = a_t * h + b_t

25
26      # Compute output projection from current hidden state
27      c_t = W_C(x_t)
28      # produce scalar (or vector) output
29      y_t = dot(c_t, h)
30      Y.append(y_t)

31
32      # sequence of outputs and final hidden state
33      return Y, h

```

Listing 2: Detailed pseudocode for a single-head Mamba layer illustrating the selective state-space mechanism. Each input token embedding is projected into a decay term (a_t), an additive input contribution (b_t), and an output projection vector (c_t). At each sequence step, the hidden state (h) is updated via exponential decay and additive inputs, capturing long-range dependencies efficiently. Finally, outputs are generated as a projection of the updated hidden state.

This code uses an exponential decay formulation for the state update: $a_t = \exp(-\text{softplus}(W_A(x_t)))$ ensures $0 < a_t < 1$. Thus $A_t \approx a_t I$ in this example, and the state update $h \leftarrow a_t h + B_t$ (with $B_t = W_B(x_t)$) corresponds to Equation (1). In practice, N (state size per head) might be 64 or 128, and multiple independent SSM heads are used (analogous to multi-head attention) to increase modeling capacity [3]. The projection C_t produces an output or can be further fed into downstream layers. Notably, all operations are simple element-wise multiplies and dot-products, which can be efficiently implemented on hardware. The main loop can be parallelized by unrolling blocks or using scan kernels to leverage GPU SIMD capabilities.

Efficiency and Scalability: Thanks to its linear complexity, Mamba scales gracefully to long inputs and high-throughput generation scenarios common in AIGC. It avoids the quadratic memory growth of Transformers, enabling, for instance, generation of extremely long texts or audio streams without the need to window or truncate contexts. The selective state update also means that if the input sequence has segments of low information, the model can essentially "skip through" by maintaining state, whereas attention would still compute numerous interactions for those positions. From a scalability perspective, the architecture has been demonstrated from small scales to billions of parameters [3, 9]. While the highest-performing LLMs today are still Transformer-based, Mamba provides an existence proof that alternatives can achieve comparable results with better efficiency. This has implications for deploying generative models on edge devices or in latency-critical applications where the overhead of attention is problematic. Moreover, the ideas from Mamba (content-driven recurrent dynamics) could inspire hybrid models that combine the best of both worlds — e.g., using attention for short-range local interactions and SSMs for long-range memory.

Reading Questions:

1. Why do fixed-parameter SSMs struggle with tasks like *selective copying* or *induction heads* in sequences? How does Mamba's input-dependent parameterization address this limitation?
2. Derive the time and space complexity of a Transformer layer versus a Mamba layer for a sequence of length T . At what sequence length might Mamba become significantly more efficient than a Transformer, and why?

3. Explain how Mamba generates sequences autoregressively. What are the memory requirements during generation compared to a Transformer, and what advantages does this confer for very long outputs?
4. Discuss potential ways to combine a selective SSM layer with an attention mechanism. What benefits might such a hybrid bring, and what challenges would need to be overcome?
5. Mamba has shown success in text and audio domains. How might the Mamba model be applied or adapted to image or multimodal generation tasks? Consider the differences in data structure (e.g. 2D spatial data) and whether a state-space approach is suitable.

5 U-Net Architecture for Image Generation

Overview: The U-Net is a convolutional neural network architecture originally introduced for biomedical image segmentation [11]. It has since become a cornerstone in generative modeling, especially in diffusion models and other image-to-image tasks. The characteristic *U* shape of the network comes from an encoder-decoder structure with skip connections: the input is progressively downsampled through convolutional layers (encoder), then upsampled back to the original resolution through corresponding deconvolutional or upsampling layers (decoder). Skip connections between corresponding layers in the encoder and decoder provide local high-resolution information to the decoder, aiding in preserving details. This architecture is well-suited for tasks where the output is an image (or image-like) of the same size as the input, such as image segmentation, image translation, and in our context, denoising images in diffusion models.

In generative diffusion models (like DDPM or Stable Diffusion), the U-Net typically serves as the *denoising model* $x_t \rightarrow x_{t-1}$, where it takes a noisy image at step t (often concatenated with additional inputs like the timestep t encoding, and possibly a text embedding for conditional generation) and predicts a less noisy image, or directly the noise component. The U-Net’s architecture with multi-resolution analysis and synthesis is adept at capturing both global structures (via the encoder’s deep, coarse feature maps) and fine details (via the high-resolution skip connections).

Architecture Details: A typical U-Net for image generation consists of:

- **An encoder (downsampling path):** Several stages of convolutional layers, each stage often consisting of two or more convolutions (with ReLU/SiLU activation and possibly group or batch normalization), followed by a downsampling operation (like a stride-2 convolution or pooling). As we go deeper, the number of feature channels often increases (e.g., doubling at each downsample) while the spatial resolution halves.
- **A bottleneck:** At the lowest resolution, some number of convolutions capture the most abstract representation (this is the bottom of the U).
- **A decoder (upsampling path):** For each encoder stage, there is a corresponding decoder stage. Upsampling can be done by transpose convolutions or interpolation followed by a convolution. The skip connection from the encoder provides the decoder with the feature map of the same spatial size, which is typically concatenated along the channel dimension. The decoder then processes this combined information (with convolutional layers) and continues to the next higher resolution.
- The final layer outputs an image (or image-like tensor) of the same size as the input. In diffusion models, the output often has the same number of channels as the image (e.g., 3 for RGB) representing either a denoised image prediction or the predicted noise.

One key aspect is that the U-Net is fully convolutional and thus agnostic to input image size (beyond memory constraints), making it flexible for various resolutions.

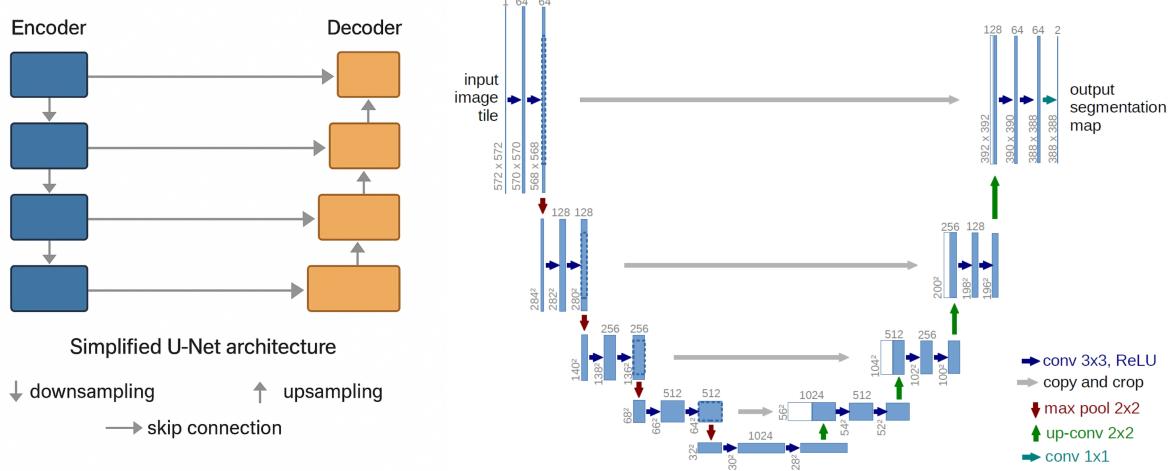


Figure 4: Simplified U-Net architecture. Each downward arrow represents downsampling (encoder), each upward arrow represents upsampling (decoder), and horizontal connections denote skip connections that pass feature maps from encoder to decoder. This design helps the network output retain fine details from the input.

Mathematical Formulation: While the U-Net is primarily an architecture and not a single equation, we can describe its effect as a function $f_\theta(x)$ where x is an input image (possibly with noise) and $f_\theta(x)$ is the output image. Internally, let E_i denote the operation of the encoder up to stage i , and D_i the operation of the decoder from stage i . The skip connection means that at decoder stage i , the input is $[D_{i+1}(z), e_i]$ where $e_i = E_i(x)$ is the feature map from the encoder and z is the code from the lower level (with D_n at the bottom being the identity on the bottleneck code). The concatenation $[., .]$ is along channels. So, schematically:

$$e_n = E_n(E_{n-1}(\dots E_1(x) \dots)), \quad (\text{bottom feature map})$$

$$y = D_1([E_1(x), D_2([E_2(x), \dots D_n(e_n) \dots])]),$$

where $y = f_\theta(x)$ is the output image. Each E_i and D_i may consist of multiple convolutional layers. The exact composition can vary (for example, some implementations include attention blocks at certain resolutions, like spatial self-attention when the feature map is not too large, as done in Stable Diffusion’s U-Net for 16x16 features).

Temporal Conditioning and Other Additions: In diffusion models, the U-Net often takes a timestep t as additional input (since the denoising function needs to behave differently for different noise levels). This is typically done by computing a embedding of t (e.g., sinusoidal position embedding or learned embedding) and injecting it into the network via adding to feature maps or through adaptive normalization layers. For example, one common method is to use the t embedding to produce a bias and scale for each feature map in normalization layers (this is similar to the concept of adaptive instance normalization or FiLM conditioning). If the diffusion model is conditional (e.g., text-to-image), the U-Net might also take a text embedding that influences its layers, either by concatenating across channels, or via cross-attention layers that attend to text tokens (as done in Stable Diffusion).

Example Usage in Diffusion: In a Denoising Diffusion Probabilistic Model (DDPM), at each training step we have a data example x_0 and a random timestep t . We create a noisy version $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$ (where $\epsilon \sim N(0, \mathbf{I})$ and $\bar{\alpha}_t$ is a known constant from the noise schedule). The U-Net model f_θ is tasked to predict ϵ (or a clean x_0) from x_t and t . The U-Net's output $\hat{\epsilon}_\theta = f_\theta(x_t, t)$ is then used to compute a loss $\|\hat{\epsilon}_\theta - \epsilon\|^2$. This setup leverages the U-Net's capacity to interpret the noisy image at varying levels of noise and gradually refine it.

```

1 # Pseudocode for U-Net forward (conceptual, not actual code)
2 def U_Net_forward(x_t, t, text_emb=None):
3     # Encode (downsample path)
4     feats = [] # to store features for skip connections
5     h = x_t
6     for down_block in encoder_blocks:
7         h = down_block(h, t, text_emb) # might include convs, normalization, etc.
8         feats.append(h)
9         h = downsample(h)
10    # bottleneck
11    h = bottleneck_block(h, t, text_emb)
12    # Decode (upsample path)
13    for up_block in decoder_blocks:
14        h = upsample(h)
15        skip_feat = feats.pop() # get corresponding encoder feature
16        h = torch.cat([h, skip_feat], dim=channel_dim)
17        h = up_block(h, t, text_emb)
18    # output layer
19    output = final_conv(h) # predict noise or image
20    return output

```

Listing 3: Illustrative pseudocode for a U-Net forward pass in a diffusion context. This assumes functions *encode*, *decode* for brevity, and that time and optional text conditioning are injected appropriately.

Reading Questions:

1. How do skip connections in the U-Net help in preserving fine details in the output, especially when the network is used to reconstruct an image from a noisy input?
2. In what ways might the U-Net architecture be modified or extended to further improve its performance in image synthesis? (Think about adding attention mechanisms, more levels, different normalization, etc.)
3. Why is the U-Net a good choice for the denoiser model in diffusion frameworks? What properties make it suitable for handling different noise levels and image structures?
4. Consider training a U-Net for a task like image colorization or super-resolution. How would you adapt the input/output and the loss function for those tasks compared to the diffusion denoising task?

6 Denoising Diffusion Probabilistic Models (DDPM)

Overview: Denoising Diffusion Probabilistic Models (DDPMs) [12], often just called diffusion models, are generative models that learn to synthesize data by reversing a gradual noising process. They are inspired by non-equilibrium thermodynamics (diffusion processes) and score-based generative modeling [20]. The key idea is to start from a sample of real data (e.g., an image), gradually

add noise to it over T time steps until it becomes (nearly) pure noise, and then train a model to invert this process: starting from random noise at T and step-by-step removing noise to yield a coherent sample. The forward noising process is fixed (usually predefined Gaussian noise schedule), and the reverse denoising process is learned.

DDPMs have emerged as a powerful class of generative models, achieving state-of-the-art image generation quality on many benchmarks and rivalling GANs in fidelity while often surpassing them in coverage (mode coverage, diversity of samples). However, they are computationally expensive at generation time because they require many iterative steps.

Forward Process (Diffusion): We define a forward *Markov chain* of latent variables x_1, x_2, \dots, x_T starting from the original data x_0 . At each step t , a small amount of Gaussian noise is added:

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

where $\{\beta_t\}_{t=1}^T$ is a variance schedule (e.g., linearly increasing small numbers) controlling how much noise is added at each step. This means

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_{t-1},$$

with $\epsilon_{t-1} \sim \mathcal{N}(0, \mathbf{I})$. After T steps, x_T is nearly an isotropic Gaussian (if T is large enough and β_t not too small). An important property is that we can sample x_t at an arbitrary step t in closed form from x_0 :

$$q(x_t | x_0) = \mathcal{N}\left(x_t; \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t) \mathbf{I}\right),$$

where $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^t \alpha_s$. This comes from composing the linear transformations of each step. Practically, this means we can sample a noised version of x_0 at any timestep directly:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon,$$

with $\epsilon \sim \mathcal{N}(0, \mathbf{I})$.

Reverse Process (Denoising / Generation): The model we train (often a U-Net, as discussed) is used to approximate the reverse of the forward process. The reverse process is also assumed to be a Markov chain with Gaussian transitions:

$$p_\theta(x_{t-1} | x_t) = \mathcal{N}\left(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)\right),$$

where μ_θ and Σ_θ are outputs of a neural network (with parameters θ) taking the current noised sample x_t and an index t (the timestep, which can be provided as an encoding to the network). In many implementations, Σ_θ (the variance) is not predicted but fixed to a certain schedule or a function of β_t (for example, some use $\Sigma_\theta(x_t, t) = \tilde{\beta}_t \mathbf{I}$ for a fixed value $\tilde{\beta}_t$ derived from β_t).

If we have a good model for $p_\theta(x_{t-1}|x_t)$, we can generate a sample by: 1. Sampling $x_T \sim \mathcal{N}(0, \mathbf{I})$ (pure noise in the data space). 2. For $t = T, T-1, \dots, 1$: sample $x_{t-1} \sim p_\theta(x_{t-1}|x_t)$ using the model (with the given x_t). 3. Output x_0 which should be a data point similar to the training data distribution.

Training Objective (ELBO and Simplified Loss): Diffusion models are often trained by maximizing the evidence lower bound (ELBO) on the data likelihood. The ELBO can be derived by considering the combined forward (encoder) and reverse (decoder) processes as a VAE-like structure. The ELBO typically breaks down into a sum of terms L_t for each timestep (which are expected Kullback-Leibler divergences between the model's Gaussian $p_\theta(x_{t-1}|x_t)$ and the true posterior $q(x_{t-1}|x_t, x_0)$) plus a term L_0 for the reconstruction at $t = 0$. Ho et al. [12] showed that

a weighted objective, in particular one where each L_t is weighted by $\frac{1}{2}\bar{\alpha}_t\frac{\beta_t}{1-\bar{\alpha}_t}$, simplifies the loss and that a further simplification (setting all weights equal) still produces good results. This leads to the *simple loss* formulation:

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

where t is uniformly sampled from $\{1, \dots, T\}$, $\epsilon \sim \mathcal{N}(0, \mathbf{I})$, and $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ as in the forward process. Here $\epsilon_\theta(x_t, t)$ is the model's prediction for the noise added. In other words, the model is trying to predict the exact noise ϵ that was added to a clean sample to produce x_t . This simple loss is essentially a reweighted form of the ELBO (neglecting certain terms), and it has been empirically found to work well.

By learning to predict ϵ , one can recover a prediction of the denoised x_0 as:

$$\hat{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(x_t, t)).$$

This can be plugged into the reverse mean formula:

$$\mu_\theta(x_t, t) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\hat{x}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t,$$

which is a linear combination of x_t and the predicted \hat{x}_0 . (This formula comes from the analytical expression of $q(x_{t-1}|x_t, x_0)$ and substituting \hat{x}_0 for x_0 .)

Conditional Diffusion: The above formulation describes unconditional generation. To guide generation with conditioning information (like class labels or text), one can provide the model with additional inputs. Two popular methods are:

- *Conditional input*: e.g., concatenate a one-hot class embedding to x_t or into the network, or in the case of text, use cross-attention layers in the U-Net to attend to text embeddings (this is what GLIDE and Stable Diffusion do).
- *Classifier-free guidance* [21]: train the model sometimes with the condition and sometimes without (using a special null token), then at sampling time, interpolate between conditional and unconditional predictions to guide the sample. This effectively amplifies the influence of the condition.

Sampling Efficiency: A drawback of DDPM is needing T sequential steps. Various improvements allow using fewer steps:

- Use a smaller T with carefully designed noise schedules or by training the model with fewer steps from the start.
- Use higher-order differential equation solvers or annealed Langevin dynamics interpretations (score-based models) to jump larger steps (e.g., Denoising Diffusion Implicit Models, DDIM).
- Knowledge distillation or learning explicit samplers that achieve the same result in fewer steps.

However, these are beyond the scope of this handbook's theoretical focus; we acknowledge them as practical considerations.

```

1 # Pseudocode for diffusion model training (simplified)
2 for x0 in dataset:
3     t = random.randint(1, T)           # sample a timestep uniformly
4     epsilon = torch.randn_like(x0)    # sample random noise
5     alpha_bar_t = alpha_bar[t]        # precomputed product of (1-beta) up to t
6     # Diffuse the clean image to time t
7     x_t = (alpha_bar_t**0.5) * x0 + ((1 - alpha_bar_t)**0.5) * epsilon
8     # Model predicts the noise from x_t
9     epsilon_pred = model(x_t, t)      # e.g., U-Net with t embedding
10    # Compute loss (mean squared error between true and predicted noise)
11    loss = ((epsilon_pred - epsilon)**2).mean()
12    loss.backward()
13    optimizer.step()

```

Listing 4: Illustration of diffusion training loop in pseudocode. This trains the model to predict added noise ϵ .

Reading Questions:

1. Why is it beneficial to predict the noise ϵ added (as in L_{simple}) rather than directly predicting the denoised image x_0 in diffusion model training?
2. How does the choice of noise schedule $\{\beta_t\}$ affect the performance of a diffusion model? What are some desirable properties of a good noise schedule?
3. Compare the diffusion model training objective to a variational autoencoder (VAE). In what ways are they similar (think ELBO), and how do they differ in terms of what the model learns?
4. In conditional diffusion (e.g., text-to-image), why might one prefer the classifier-free guidance approach over training a separate classifier to guide the diffusion process?
5. What challenges arise when trying to generate samples with as few diffusion steps as possible, and what strategies have researchers developed to address this?

7 Diffusion Transformer (DiT)

Overview: Diffusion Transformer (DiT) [13] is a model that integrates transformers into the architecture of diffusion models. In standard diffusion models for images, the U-Net (a CNN) has been the dominant choice for the denoising network. DiT explores replacing this with a pure Transformer architecture (akin to a Vision Transformer) that operates either on image pixels or latent patches. This is appealing because Transformers offer great flexibility in modeling complex relationships and can benefit from the vast body of research on transformer scaling and optimization. The DiT model has achieved competitive results to CNN-based models, indicating that transformers can indeed serve as a backbone for diffusion.

Tokenizing the Image: A key step in using a Transformer for images (as with ViT) is to tokenize the image into a sequence. DiT typically operates on latent representations of images, such as the 2D feature map output by an encoder or the latent space of a VAE (like in LDM). For instance, if we have an image of 256×256 that is encoded by a VAE into a $32 \times 32 \times 4$ latent (as in Stable Diffusion), this 32×32 spatial latent can be thought of as 1024 tokens of dimension 4. However, in practice, to use a Transformer, one would project these 4-dimensional vectors to a

higher dimension d (the model dimension) via a linear layer (patch embedding). Alternatively, one could treat patches of the image or latent as tokens.

For example, DiT might use a patch size $p = 2$ on a $32 \times 32 \times 4$ latent, yielding $16 \times 16 = 256$ tokens, each token representing a 2×2 patch of the latent (with $2 * 2 * 4 = 16$ values, then projected to d). The smaller the patch, the longer the token sequence, which increases Transformer computation but retains more local detail per token. Each token is then supplemented with a positional embedding as in ViT (usually 2D-aware positional embeddings or flattened 1D positional embeddings).

Transformer Architecture in DiT: The Transformer used in DiT is broadly similar to a ViT encoder: multi-head self-attention and MLP blocks, with layer norm and residual connections. However, the diffusion model scenario introduces conditioning on the timestep (and optionally other conditioning like class labels or text). The conditioning can be incorporated in various ways:

- *In-context token*: Append extra tokens to the sequence that represent the timestep t and any other conditional information (like a class embedding). The model then treats these as part of the sequence. This is analogous to how BERT uses a classification token, or how some models include text tokens in a visual Transformer sequence for multimodal learning.
- *Cross-Attention*: Augment each Transformer block to include an attention layer where the queries come from the image tokens and the keys/values come from condition tokens (like text embeddings). This is how text conditioning is done in Stable Diffusion’s U-Net, and DiT can do similarly for any conditioning signal.
- *Adaptive Layer Norm (AdaLN)*: Modify the layer normalization in each block to be conditioned on the extra information. In AdaLN, the scale and bias of the layer norm (usually set to 1 and 0 in standard LN) are made into functions of the conditioning (like the timestep embedding). Essentially, for a given hidden state h in the transformer and conditioning c , the AdaLayerNorm would compute $\mu = \frac{1}{N} \sum_j h_j$ and $\sigma^2 = \frac{1}{N} \sum_j (h_j - \mu)^2$ as usual, but then output $\gamma(c) \frac{(h - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta(c)$, where $\gamma(c)$ and $\beta(c)$ are learned affine transforms of the condition c . If c includes the time encoding and other context, this allows the transformer to modulate its activations based on t or other conditions at every layer.
- *AdaLN-Zero*: A variant where the initial scale γ is set in such a way that the conditioning has a null effect at initialization (e.g., start γ as zero vectors, meaning the layer norm initially does nothing special for condition, which can stabilize training).

These mechanisms were compared in the DiT study. Cross-attention allows the model to incorporate information from a different modality or source, while AdaLN directly conditions the internal representation in a subtler way. In practice, DiT can use both: for example, class-conditioning might be done via token embedding, while text conditioning via cross-attention, and always condition on t via AdaLN in each block.

Model Sizes and Performance: DiT has been scaled to different sizes (analogous to ViT small, base, large). The design choices like patch size and depth affect the computational load. A smaller patch size yields more tokens and higher compute, but potentially better output fidelity. Researchers found that DiT models can reach generation quality on par with CNN-based diffusion models like guided diffusion, although transformers might require more parameters or compute to do so. The benefit is a unified architecture that can leverage advancements in transformer research.

Advantages and Challenges: Using a Transformer for diffusion opens up interesting possibilities: - Transformers have large receptive fields by default, so modeling global image coherence is

- They can easily integrate multimodal inputs (e.g., adding text tokens for text-to-image).
- They might be more amenable to future improvements like better memory utilization or sparse attention for higher resolutions.

However, they also pose challenges:

- Handling high-resolution images directly would mean extremely long token sequences if done naively (which is why operating in a latent space or with reasonably large patches is important).
- Transformers have quadratic complexity in sequence length, which can be a bottleneck for image data.
- The lack of built-in locality means the model might need more data or parameters to learn fine details (though combining with convolutional inductive bias or hybrid models is possible).

In summary, Diffusion Transformers represent a cross-over of vision transformer architectures with diffusion probabilistic modeling. They show that CNNs are not the only choice for image generation, and as transformers continue to evolve, they may become even more prevalent in generative modeling.

```

1 # Pseudocode outline for one DiT block (not actual code)
2 def DiT_block(x_tokens, cond_tokens=None, t_emb=None):
3     # x_tokens: (batch, N, d) input image tokens
4     # cond_tokens: (batch, M, d) optional conditioning tokens
5     # (e.g., text embeddings)
6     # t_emb: conditioning embedding for timestep, used in AdaLN
7
8     # Self-attention on image tokens (with AdaLN)
9     h = LayerNorm(x_tokens, weight=gamma(t_emb), bias=beta(t_emb))
10    h = MultiHeadSelfAttention(h) # standard MHSA
11    x_tokens = x_tokens + h # residual connection
12
13    # Cross-attention (if cond_tokens provided)
14    if cond_tokens is not None:
15        h = LayerNorm(x_tokens, weight=gamma_c(t_emb), bias=beta_c(t_emb))
16        # queries from h, keys/values from cond_tokens
17        h = MultiHeadCrossAttention(h, cond_tokens)
18        x_tokens = x_tokens + h # residual add
19
20    # Feed-forward network
21    h = LayerNorm(x_tokens, weight=gamma_ff(t_emb), bias=beta_ff(t_emb))
22    h = FeedForward(h) # typically a 2-layer MLP with GeLU
23    x_tokens = x_tokens + h # residual add
24
25    return x_tokens

```

Listing 5: Conceptual outline of a DiT forward pass for one transformer block with adaptive layer norm (AdaLN) and optional cross-attention for conditioning.

Reading Questions:

1. What are the potential benefits of using a Transformer (like DiT) as the denoising model in diffusion, compared to a CNN-based model like U-Net?
2. How does the choice of patch size (or tokenization strategy) in DiT influence the model's performance and computational requirements?
3. In conditioning a DiT on text (for text-to-image), contrast the approach of in-context token vs cross-attention. What are the pros and cons of each?
4. Consider memory and compute: Transformers scale quadratically with sequence length. What

strategies might be used to scale DiT to higher-resolution images or to save computation (hint: think of sparse attention, patch merging, or hierarchical processing)?

5. DiT and similar models borrow from ViT and diffusion. Can you think of other domains where such a replacement (transformer in place of a convolutional backbone) has been attempted and what the outcomes were?

8 Retentive Networks for Sequence Modeling

Overview: Retentive Networks (RetNet) [14] are an attention-free sequence modeling architecture developed as an efficient alternative to Transformers. While Transformers rely heavily on self-attention, leading to quadratic complexity in sequence length, RetNet introduces a linear-time *retention mechanism*. This mechanism maintains past context through exponential decay, effectively capturing long-range dependencies without computational overhead associated with self-attention. Key innovations include *multi-scale retention*, *exponential decay-based context aggregation*, *relative positional phase encoding*, and efficient autoregressive inference capabilities.

At a high level, RetNet layers consist of a retention sublayer and a feed-forward sublayer, similar in style to Transformers. However, unlike Transformers, RetNet employs a fixed exponential decay mask instead of attention weights computed via softmax normalization. This ensures that both training and generation have significantly reduced computational and memory requirements.

Retention Mechanism: The central operation in RetNet is the retention mechanism. Given input embeddings projected into queries (Q), keys (K), and values (V), RetNet computes outputs using a causal exponential decay mask:

$$\text{Retention}(Q, K, V) = (QK^\top \odot D)V,$$

where D is a causal mask defined as:

$$D_{nm} = \begin{cases} \gamma^{n-m}, & m \leq n \\ 0, & m > n \end{cases}$$

Here, $0 < \gamma < 1$ controls the exponential decay of past information. Higher γ values imply slower decay (better long-range retention), while lower values rapidly diminish past influence.

In practice, multi-scale retention employs multiple heads, each with distinct decay parameters (γ_i), capturing a diverse range of temporal dependencies—short-term as well as long-term—with a single model.

Relative Positional Phase Encoding: To inject positional information without absolute position embeddings, RetNet employs a rotary-style relative positional phase encoding. Queries and keys are multiplied element-wise by complex phases:

$$Q_n \leftarrow Q_n \odot e^{-in\theta}, \quad K_n \leftarrow K_n \odot e^{+in\theta},$$

resulting in positional information encoded as phase differences within the query-key dot products.

Gating and Feedforward Layers: RetNet layers integrate a gating mechanism and feed-forward sublayers similar to Transformers. After multi-head retention outputs $[O^{(1)}, \dots, O^{(h)}]$ are concatenated, group normalization and a gating mechanism with the Swish activation function are applied:

$$Y = \text{GroupNorm}_h([O^{(1)}; \dots; O^{(h)})], \quad G = \sigma(XW_G), \quad \text{Output} = (G \odot Y)W_O,$$

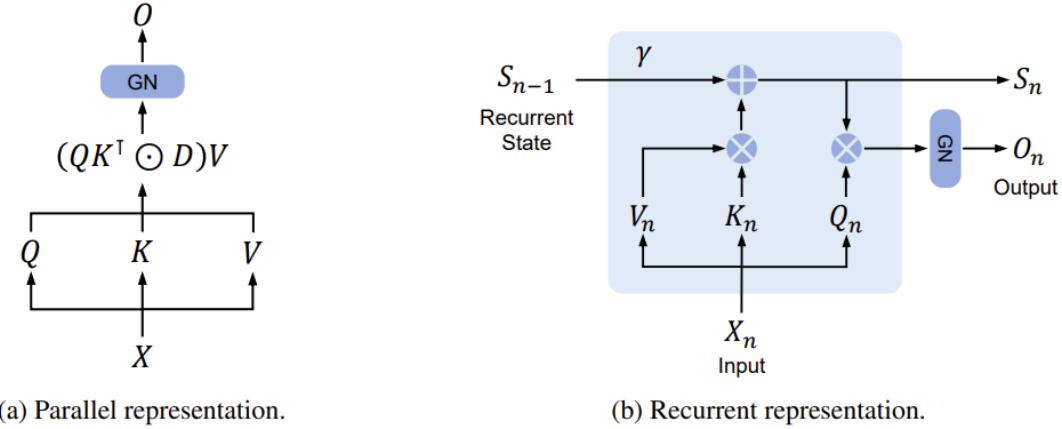


Figure 5: Illustration of the Retention mechanism. The left panel shows the parallel computation used during training, where each token position aggregates past tokens through the exponential decay mask. Darker shades represent stronger contributions (less decay). The right panel depicts the recurrent computation used at inference, showing the hidden state S_n being updated at each step by decaying previous state S_{n-1} and incorporating the current token’s information.

where σ denotes the Swish activation, and W_G, W_O are learned projections.

Efficient Autoregressive Inference: A key advantage of RetNet is constant-time autoregressive inference per token. Instead of maintaining a large attention cache, only a compact hidden state S_n is needed:

$$S_n = \gamma S_{n-1} + K_n^\top V_n, \quad O_n = Q_n S_n.$$

This yields memory usage and computational complexity independent of context length during generation.

```

1 # Parallel computation (training)
2 def retention_parallel(X, W_Q, W_K, W_V, gamma):
3     Q, K, V = X@W_Q, X@W_K, X@W_V
4     T = X.size(0)
5     mask = torch.tril(gamma ** (torch.arange(T).view(-1,1)-torch.arange(T).view(1,-1)))
6     scores = (Q @ K.transpose(-2,-1)) * mask
7     return scores @ V
8
9 # Recurrent computation (inference)
10 def retention_step(x_t, S_prev, W_Q, W_K, W_V, gamma):
11     q = x_t @ W_Q
12     k = x_t @ W_K
13     v = x_t @ W_V
14     S_t = gamma * S_prev + (k * v).sum()
15     o_t = (q * S_t).sum()
16     return o_t, S_t

```

Listing 6: PyTorch-style pseudocode for RetNet retention (parallel and recurrent forms).

Applications in Generative Modeling: RetNet achieves comparable or superior generative performance to Transformers while significantly reducing inference costs. It has demonstrated effectiveness in language modeling, long-context dialogue, and potentially multimodal scenarios due to its efficient handling of long sequences and stable training dynamics.

Reading Questions:

1. What role does the decay parameter γ play in balancing short-term versus long-term memory in RetNet?
2. Compare the computational complexity and memory usage of RetNet during inference to that of a standard Transformer. Why is RetNet more efficient?
3. Explain the benefits and potential drawbacks of using relative positional phase encoding in RetNet.
4. How could dynamically adjusting the decay parameter γ based on content improve or hinder RetNet's generative performance?

9 Latent Diffusion Models (LDM)

Overview: Latent Diffusion Models [15] are a class of diffusion-based generative models that perform the diffusion process not in the high-dimensional pixel space, but in a lower-dimensional latent space of an autoencoder. The primary motivation is efficiency: modeling directly in pixel space for high-resolution images (e.g., 512×512) can be extremely computationally intensive for diffusion models, as the network has to predict noise for millions of pixels at once. By first compressing images into a latent representation (through a learned encoder, typically a variational autoencoder (VAE) or similar), the diffusion model can operate on a much smaller representation. After diffusion sampling in the latent space, a decoder (the other half of the autoencoder) reconstructs the image.

Stable Diffusion is a prominent example of an LDM, where a 512×512 image is encoded to a 64×64 latent (with 4 channels), reducing the data size by a factor of $48 \times$. Diffusion on this latent yields tremendous speed-ups and memory savings, while the autoencoder ensures that the semantics of the image are largely preserved in the latent space.

Autoencoder (VAE) component: An LDM consists of:

- An encoder $E_\phi : \mathbb{R}^{H \times W \times 3} \rightarrow \mathbb{R}^{h \times w \times c}$ that compresses an image to a latent code $z = E_\phi(x)$. Often $h = H/d$, $w = W/d$ for some downsampling factor d (like 8 or 16), and c is the number of latent channels (e.g., 4 or 8). The encoder is usually trained as a VAE, meaning it actually outputs a distribution (mean and maybe variance) for z . But in practice, often a deterministic encoding (taking the mean) is used at sampling time.
- A decoder $D_\phi : \mathbb{R}^{h \times w \times c} \rightarrow \mathbb{R}^{H \times W \times 3}$ that reconstructs an image from the latent z . This is the generative part that upsamples and adds details. It also is part of the VAE training so that $D_\phi(E_\phi(x)) \approx x$ for training images.
- A diffusion model $p_\theta(z_{0:T})$ in the latent space that learns to generate latent codes. This is conceptually similar to the pixel diffusion model in DDPM, but now z_0 is a latent instead of an image. The diffusion process definitions remain the same, except they operate on z .

The VAE is usually trained first on images to learn a good compression. It may use a perceptual loss and a regular VAE KL loss to ensure the latent space is well-behaved. Once the VAE is trained, it is frozen, and the diffusion model is trained on the latent codes of images. Since the autoencoder has limited capacity, it cannot capture extremely fine pixel detail; thus the diffusion model in latent space focuses on getting the global structure and rough appearance right. The decoder then fills in some of the detail.

Benefits of Latent Diffusion: - *Efficiency:* If the image is downsampled by factor d in each dimension, the number of pixels goes down by d^2 . For $512 \rightarrow 64$ (factor 8 in each dimension), that's a factor 64 reduction in data. This means the U-Net or Transformer in the diffusion model deals with 1/64th the number of values. This results in much faster training and sampling, and ability to use larger model capacity for the same memory. - *Quality:* The VAE can be trained to focus on perceptually important features (often using a perceptual loss, e.g., LPIPS, plus adversarial or KL losses). This can yield latents that capture the essence of the image but not the pixel-level noise. Thus, diffusion in latent space is less burdened by modeling imperceptible pixel-level details and can converge faster. - *Flexibility:* One can plug in different decoders for different resolutions or domains once a latent model is trained. Also, the latent diffusion model can be conditioned on various inputs (text, etc.) similarly but with less compute.

However, one must accept some loss of fidelity due to compression. If the autoencoder reconstruction error is significant, the generated outputs might have artifacts (e.g., slightly blurry or less crisp details). In practice, high-quality VAEs with a slight adversarial training can produce very good reconstructions.

Mathematical Formulation: Training of an LDM is essentially training two parts: 1. The autoencoder: typically trained with a combination of a reconstruction loss and a regularization loss. For example, optimize E_ϕ, D_ϕ to minimize

$$\mathcal{L}_{VAE} = \mathbb{E}_{x \sim q_{\text{data}}} [\|D_\phi(E_\phi(x)) - x\|^2] + \beta D_{\text{KL}}(q_\phi(z|x) \| p(z)),$$

where $p(z)$ is a prior (usually standard normal) and $q_\phi(z|x)$ is the encoder's distribution (which for an autoencoder might be deterministic plus some Gaussian noise). β might be 1 for a standard VAE or adjusted. Some implementations forego an explicit KL term in favor of just an autoencoder with a narrower bottleneck and/or some regularization. 2. The diffusion: train $p_\theta(z_t|z_{t+1})$ as in standard diffusion, but using latents. The training objective is the same L_{simple} but on z . That is:

$$\mathbb{E}_{z_0 \sim E_\phi(x), \epsilon \sim \mathcal{N}(0, \mathbf{I}), t} [\|\epsilon - \epsilon_\theta(z_t, t, c)\|^2],$$

where c could be conditioning like text. Note $z_0 = E_\phi(x)$ for real image x , and $z_t = \sqrt{\bar{\alpha}_t} z_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$.

At sampling time, we generate a latent by starting from $z_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively denoising T steps to get $z_0 \sim p_\theta(z_0)$, then decode: $\hat{x} = D_\phi(z_0)$. Often, D_ϕ is deterministic (taking the mean of the VAE decode distribution).

Example: Stable Diffusion Implementation Details: Stable Diffusion uses a VAE that compresses 256 or 512-sized images to 64 (i.e., factor 4 or 8 per side). The diffusion model is a U-Net operating on this 64x64 latent, with text conditioning via cross-attention. After generating the 64x64 latent, the VAE decoder produces the final image. The autoencoder in Stable Diffusion also has an encoder that outputs not just mean but also uses a trick called "latent quantization" in some versions (like VQGAN style), but let's keep it simple.

Advantages Recap: LDMs like Stable Diffusion demonstrated that it is possible to generate 512x512 images (and larger) on consumer GPUs with limited memory, something that was previously infeasible with pixel-space diffusion without massive resources. The modular approach (VAE + diffusion) also allows reusing the VAE for other purposes and focusing attention on the generative model in a compressed domain.

Reading Questions:

1. What are the trade-offs between training a diffusion model in pixel space versus a latent space? Consider factors like image quality, diversity, training complexity, and artifacts.

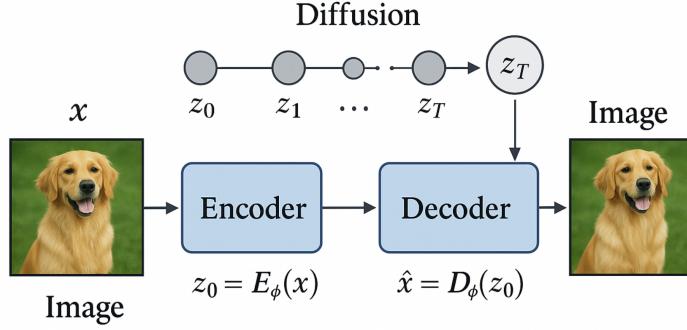


Figure 6: Latent Diffusion Model pipeline. During training, images x are encoded to latents $z_0 = E_\phi(x)$. Diffusion is applied in latent space ($z_0 \rightarrow \dots \rightarrow z_T$) and a model is trained to reverse it. For generation, we sample noise z_T in latent space, apply the learned reverse diffusion to get z_0 , then decode z_0 to an image $\hat{x} = D_\phi(z_0)$.

2. How does the choice of latent dimensionality (both spatial size and channel count) impact the performance of an LDM?
3. Could one train a latent diffusion model without a Variational Autoencoder, for example using a deterministic encoder or even an encoder learned jointly with diffusion? What might be the challenges in that approach?
4. Discuss how the autoencoder’s quality (reconstruction fidelity) affects the overall output of the latent diffusion model. If the autoencoder is imperfect, how does that manifest in the final images?
5. LDMs separate the concern of compression and generation. Can you think of analogous approaches in other domains (e.g., audio, 3D) where first compressing the data and then generative modeling in the compressed space would make sense?

10 Text-to-3D Generation: DreamFusion and Magic3D

Overview: Text-to-3D generation is an emerging field where the goal is to create a 3D object or scene given a text description (prompt). Unlike images, 3D assets (like meshes or NeRFs) can be rendered from any viewpoint. Recent breakthroughs like DreamFusion [16] and Magic3D [17] have shown that we can leverage the power of 2D diffusion models (trained on image-text data) to supervise the creation of 3D content, without needing any 3D training data. The core idea is to use a *score distillation* technique, where a pre-trained 2D diffusion model (e.g., Imagen or Stable Diffusion) acts as a critic that guides an optimization process to produce a 3D model whose renderings are consistent with the input text prompt.

The challenge is that generating 3D entails ensuring that all views of an object are consistent with a single underlying 3D shape and appearance. The diffusion model, however, only sees 2D images. So one must carefully craft a loss that, when optimized, yields a coherent 3D asset rather than a collection of unrelated 2D images per view.

Neural 3D Representations: DreamFusion uses a Neural Radiance Field (NeRF) as the 3D representation. NeRF represents a 3D scene implicitly via a neural network that, given a 3D coordinate and viewing direction, outputs a density and color. Rendering is done by volumetric

rendering (integrating along camera rays). Magic3D builds on this by also using a mesh representation in a second stage. There are other choices: one could use voxel grids, point clouds, etc. The requirement is that the representation is differentiable with respect to parameters so that we can optimize it via gradient descent.

Score Distillation Sampling (SDS): This is the key technique introduced in DreamFusion. The idea is as follows: - We have a diffusion model (like Imagen or Stable Diffusion) which defines a score (or noise prediction) for images given a text prompt. Specifically, given an image I and text y , the diffusion model can tell us how to adjust I (in the space of noisy images) to make it more aligned with y . This comes from the diffusion model's capability to predict noise or denoise images conditioned on text. - If we have a current 3D model (NeRF) with parameters Θ , we can render an image $I = \text{render}(\Theta, v)$ from a random camera viewpoint v . Now we want to update Θ so that I is more likely under the diffusion model for prompt y .

Mathematically, SDS can be described by taking the diffusion model's predicted noise $\epsilon_\phi(I_t, t, y)$ (where I_t is a noised version of the rendered image at noise level t) and comparing it to the actual noise added. We then compute gradients w.r.t. the image I and then by chain rule w.r.t. the 3D parameters Θ :

$$\nabla_\Theta \mathcal{L}_{SDS} = \mathbb{E}_{t, \epsilon, v} \left[w(t) \cdot (\epsilon_\phi(I_t, t, y) - \epsilon) \frac{\partial I_t}{\partial \Theta} \right],$$

where $I_t = \sqrt{\bar{\alpha}_t} I + \sqrt{1 - \bar{\alpha}_t} \epsilon$ as usual in diffusion (applied to the rendered image), and $w(t)$ is a weight (often $1/\sqrt{\bar{\alpha}_t}$ or similar). Intuitively, if the diffusion model thinks the rendered image I is not aligned with text y (so it predicts a certain noise to add/remove), this creates a gradient that changes Θ to make I closer to what the model expects for that prompt.

The actual implementation avoids explicitly constructing large Jacobians; it uses the technique of "score distillation" derived elegantly from the diffusion model's loss function. Practically, one can backpropagate through the diffusion model's prediction and through the NeRF's rendering.

DreamFusion (Single-stage, NeRF-only): DreamFusion starts with a randomly initialized NeRF (which usually means a multi-layer perceptron with random weights that implicitly define density and color). It then iteratively updates the NeRF using SDS. To avoid the trivial solution of the NeRF just increasing brightness or other degeneracies, they include some regularizers: - A distortion loss (from NeRF literature) to encourage a reasonable distribution of density (to avoid foggy or fragmented solutions). - An entropy loss on the rendered pixels or some sparsity to avoid the NeRF representing everything as equally likely. - Optionally, an "orientation loss" (some implementations mention normal smoothing, etc.). DreamFusion used the Imagen diffusion model at a low image resolution like 64×64 for the SDS guidance. Despite the low resolution, the NeRF can be rendered at higher resolution once trained because the NeRF itself is continuous; however, fine details might be lacking. DreamFusion outputs a NeRF which can then be converted to a mesh if needed, but the quality might not be super high-res.

Magic3D (Two-stage: NeRF + Mesh): Magic3D improves upon DreamFusion in several ways: - Speed: it uses a faster NeRF implementation (instant-npg style with a hash grid) for the coarse stage. - Quality: after obtaining a coarse NeRF at low resolution (64^2 renders), it converts that to a mesh (via marching cubes or similar) and then uses a second stage optimization. In the second stage, they use a differentiable renderer for the mesh and a higher resolution diffusion model (like Stable Diffusion at 256^2 or 512^2) to refine the texture and details on the mesh. The mesh is parameterized by vertex positions and a texture map (or vertex colors, etc.), which are updated. - This two-stage approach yields a final high-res textured mesh that can be directly used in graphics engines, and it's done faster than DreamFusion due to the efficient coarse stage and requiring fewer iterations in the fine stage.

One key challenge in both is ensuring multi-view consistency. If each view were optimized independently to match the prompt, the model could end up painting different details on different sides of the object (like contradictory appearances). The 3D representation ties them together, but there's still the risk that the diffusion model might encourage a change that improves one view while hurting another. In practice, random view sampling and the fact that the NeRF or mesh is a single object helps propagate consistent features, but some subtle inconsistencies can happen (like a logo or text appearing different on different sides).

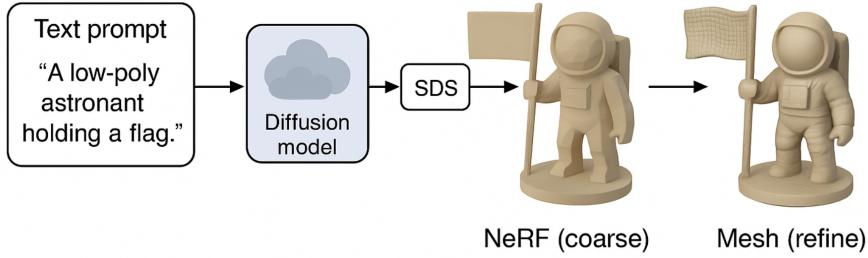


Figure 7: Coarse-to-fine text-to-3D generation pipeline. In Stage 1, a NeRF representation is optimized so that its renderings are aligned with the text prompt via a diffusion model’s guidance (SDS loss). In Stage 2, the coarse NeRF is converted to a mesh and further optimized using a higher resolution diffusion prior, yielding a detailed 3D mesh model.

Other Approaches and Considerations: While DreamFusion and Magic3D are pioneering methods, the field is quickly evolving. Some others have looked at improving geometry (e.g., enforcing symmetry or smoothness in shapes), using multiple prompts or views to compose scenes, or speeding up convergence (since these optimizations can take tens of minutes to hours). There are also text-to-3D approaches using retrieval or parametric models (like deforming a known template), but the diffusion-driven ones are the most general as they do not require a starting shape or 3D training data.

Reading Questions:

1. Explain in your own words what the Score Distillation Sampling (SDS) loss is doing. Why is a pre-trained diffusion model critical in this process?
2. What are the main differences between DreamFusion and Magic3D in terms of methodology and outcomes? Why does Magic3D achieve faster and higher-resolution results?
3. Multi-view consistency is a core challenge in text-to-3D. How does optimizing a single 3D representation (like a NeRF or mesh) inherently enforce some consistency, and what additional measures might be used to ensure the model doesn’t cheat by making different views inconsistent?
4. Consider limitations of the text-to-3D approach: what happens if the text prompt describes an object that is not symmetric or has a complex interior (like ”a hollow sphere with something inside”)? How might current methods handle or fail on such cases?
5. Looking forward, how could one integrate 3D diffusion models (if large 3D datasets were available) with 2D diffusion to improve text-to-3D? What advantages could a native 3D generative model bring?

11 Conclusions

Generative foundation models have made remarkable strides in their ability to create content across different modalities. From the sequence modeling power of Transformers enabling fluent language generation, to the pixel-level mastery of diffusion models creating photorealistic images, and now the extension of these techniques to efficient linear-time sequence models like Mamba and attention-free approaches like RetNet, the pace of advancement has been extraordinary. The architectures covered in this handbook—Transformers, Vision Transformers, Mamba, U-Nets, Diffusion Models, Diffusion Transformers, RetNet, Latent Diffusion, and text-to-3D pipelines—each contribute uniquely to the understanding and generation of high-dimensional complex data.

Several unifying themes emerge:

- *Scale*: Larger models and more data generally lead to more powerful generative models, albeit with diminishing returns and increased engineering complexity.
- *Modularity*: Combining components like autoencoders with diffusion models (in LDMs), or integrating 2D models with 3D representations (in DreamFusion) highlights the strength of decomposing complex generative tasks into manageable sub-tasks.
- *Attention and Retention mechanisms*: Attention blocks in Transformers and U-Nets, along with the exponential decay-based retention in RetNet, have demonstrated versatile capabilities in capturing dependencies within sequential, spatial, and cross-modal data.
- *Optimization and Objectives*: Many generative models rely on specialized loss functions and objectives (e.g., the diffusion ELBO or score distillation sampling) to indirectly achieve desired outcomes in the absence of explicit supervised signals.
- *Quality versus Efficiency*: The trade-off between high-fidelity generation and computational efficiency is continuously addressed through innovations like latent modeling, selective state-space mechanisms in Mamba, distillation techniques, and multi-stage generation frameworks.

As foundation models continue to evolve, one can envision even more sophisticated systems integrating multiple modalities within unified frameworks—systems capable of generating text, images, and 3D scenes by adaptively routing data through combinations of transformer backbones, diffusion modules, and efficient retention architectures. The theoretical foundations and architectures discussed herein will likely underpin these future developments.

We hope this handbook has not only clarified the mechanics of current generative models but also provided insight into thoughtful model design and application. The reading questions included throughout are intended to encourage further exploration and research. Numerous open challenges remain, such as reducing generation steps without quality loss, integrating external knowledge or constraints into generative frameworks, and ensuring ethical deployment of these powerful technologies. Understanding these foundational models represents an essential step toward addressing such future challenges and contributing to the ongoing advancement of generative AI.

If this handbook supports your research or dissertation,
please consider citing it!

References

- [1] Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). *Attention is All You Need*. NeurIPS.
- [2] Dosovitskiy, A., Beyer, L., Kolesnikov, A., et al. (2021). *An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale*. ICLR.
- [3] Gu, A., Dao, T. (2023). *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. arXiv:2312.00752.
- [4] Katharopoulos, A., Vyas, A., Pappas, N., Fleuret, F. (2020). *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*. ICML.
- [5] Katharopoulos, A., Vyas, A., Pappas, N., Fleuret, F. (2020). *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*. ICML.
- [6] Dauphin, Y.N., Fan, A., Auli, M., Grangier, D. (2017). *Language Modeling with Gated Convolutional Networks*. ICML.
- [7] Dao, T., Fu, D.Y., Ermon, S., Rudra, A., Ré, C. (2022). *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. NeurIPS.
- [8] Gu, A., Goel, K., Ré, C. (2022). *Efficiently Modeling Long Sequences with Structured State Spaces*. ICLR.
- [9] Peng, B., Biderman, S., Koón, J., et al. (2023). *RWKV: Reinventing RNNs for the Transformer Era*. arXiv:2305.13048.
- [10] Dao, T., Gu, A. (2024). *Transformers are SSMs: Generalized Models and Efficient Algorithms through Structured State Space Duality*. arXiv:2405.21060.
- [11] Ronneberger, O., Fischer, P., Brox, T. (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation*. MICCAI.
- [12] Ho, J., Jain, A., Abbeel, P. (2020). *Denoising Diffusion Probabilistic Models*. NeurIPS.
- [13] Peebles, W., Huang, P., Xie, S., et al. (2022). *Scalable Diffusion Models with Transformers*. arXiv:2212.09748 / ICCV 2023.
- [14] Sun, Z., Yu, B., Zhang, T., et al. (2023). *Retentive Network: A Successor to Transformer for Large Language Models*. arXiv:2307.08621.
- [15] Rombach, R., Blattmann, A., Lorenz, D., et al. (2022). *High-Resolution Image Synthesis with Latent Diffusion Models*. CVPR.
- [16] Poole, B., Jain, A., Barron, J. T., Mildenhall, B. (2022). *DreamFusion: Text-to-3D using 2D Diffusion*. arXiv:2209.14988.
- [17] Lin, C.-H., Gao, J., Tang, L., et al. (2023). *Magic3D: High-Resolution Text-to-3D Content Creation*. CVPR.
- [18] Kaplan, J., McCandlish, S., Henighan, T., et al. (2020). *Scaling Laws for Neural Language Models*. arXiv:2001.08361.
- [19] He, K., Zhang, X., Ren, S., Sun, J. (2016). *Deep Residual Learning for Image Recognition*. CVPR.
- [20] Song, Y., & Ermon, S. (2019). *Generative Modeling by Estimating Gradients of the Data Distribution*. NeurIPS.
- [21] Ho, J., & Salimans, T. (2022). *Classifier-Free Diffusion Guidance*. NeurIPS Workshop.