Generative AI Notes

Ana Maria Sousa

Hugging Face

Content

1. Generative Models

1.1. Introduction

Generative models focus on understanding and learning the underlying distribution of the data. These models aim to generate new samples that are similar to the original data by modeling the joint probability distribution P(X,Y).

Machine learning models can generally be separated into two large families, generative models and discriminative models. Discriminative models are the most well-know and just concentrate on learning the boundaries that separate different classes within the data.

In summary:

- **Generative Models:** Aim to model the actual data distribution and can generate new data points. Learn P(X,Y) the joint probability distribution of inputs and outputs.
  Example: Variational Autoencoders (VAEs), Generative Adversarial Networks (GANs).

- **Discriminative Models:** Aim to find decision boundaries to classify data into different categories. Learn P(Y|X), the conditional probability of outputs given the inputs.
  Example: Convolutional Neural Networks (CNNs), Support Vector Machines (SVMs).

Generative Models are used for tasks such as image generation, inpainting, style transfer, and data augmentation. They can produce realistic images from learned distributions. Here we will consider some tasks as:

- *noise to image (DCGAN);*
- *text to image (diffusion models);*
- *image to image (StyleGAN, cycleGAN, diffusion models)*

1.2. Metrics

Evaluating generative models can be challenging because there often isn't a clear "ground truth" for the generated images, making it difficult to measure image quality.

1.2.1.1. Fréchet Inception Distance (FID)

FID is a crucial metric in assessing the quality of images generated by models, especially in the realm of GANs.

Unlike simpler metrics, FID offers a more robust evaluation by capturing high-level features and then comparing the distributions of features extracted from both real and generated images.

Lower FID scores signify a closer resemblance between generated and real images, indicating **higher quality**. Introduced as an improvement over the Inception Score, FID is known for its resistance to noise and artifacts, making it a reliable measure of image fidelity.

1.2.1.2. The Inception Score (IS)

IS another a metric used to evaluate the quality and diversity of images generated by generative models. It assesses the likelihood and distinctiveness of generated images based on their class predictions.

The score is calculated by first using a pre-trained classifier network (typically Inception-v3) to classify generated images and then computing the entropy of the class distribution and the marginal entropy of the generated images.

A higher Inception Score generally indicates **better quality and diversity** in the generated images, although it has limitations, such as being sensitive to mode collapse in GANs.

1.2.1.3. PSNR (peak signal-to-noise ratio)

PSNR measures the quality of images by comparing them to a reference image, typically by calculating the mean squared error.

It serves as a benchmark for assessing image quality by quantifying the level of noise present in an image compared to a reference image. It operates on the principle of mean squared error, providing a numerical value that indicates the fidelity of the image reconstruction process.

PSNR values typically range between 25 and 34, with results above 34 indicating exceptionally high-quality images where noise levels are minimal, thus making it a widely-used metric in various image processing applications.

1.2.1.4. SIM (Structural Similarity Index)

SSIM assesses the similarity between two images by considering their luminance, contrast, and structure components.

Ranging between 0 and 1, SSIM scores closer to 1 suggest a high degree of similarity between images, considering both pixel-wise and structural differences. This metric is particularly valuable in scenarios where precise visual fidelity is essential, such as medical imaging or video compression.

1.2.1.5. CLIP Score

Offers a novel approach to evaluating text-to-image models, focusing on the alignment between generated images and textual prompts.

By leveraging the CLIP model to compute cosine similarity, this metric assesses the model's ability to translate textual descriptions into visually coherent representations.

With scores ranging from 0 to 100, higher CLIP Scores indicate a stronger correspondence between textual input and generated images, facilitating the assessment and refinement of text-to-image generation models in natural language processing tasks.

2. Autoencoder

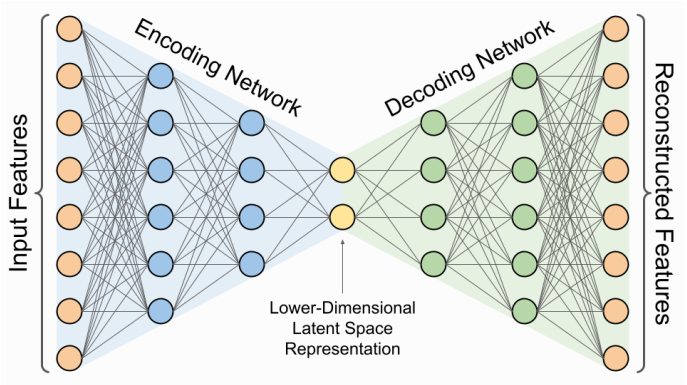
Autoencoders, a class of neural networks primarily utilized for **unsupervised learning** and **dimensionality reduction,** operate on the principle of encoding input data into a lower-dimensional representation and subsequently decoding it back to its original form while minimizing **reconstruction error.**

The architecture of an autoencoder comprises two main components:

- The encoder: The encoder transforms input data into a compressed or latent representation through one or more layers of neurons, progressively reducing dimensions.

- The decoder. The decoder takes the compressed representation from the encoder and endeavors to reconstruct the initial input data. It comprises layers arranged in the reverse order, gradually increasing dimensions to recreate the original data.

This process enables autoencoders to learn meaningful representations of input data and find efficient encodings for various tasks, including data compression, denoising, and anomaly detection.

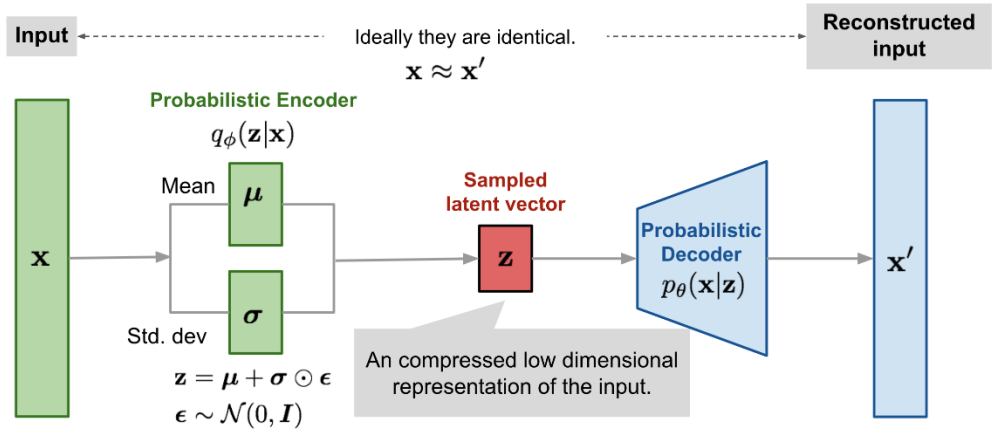


2.1. Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) represent a significant advancement over traditional autoencoders by introducing a probabilistic framework to the encoding and decoding process.

Unlike deterministic autoencoders, which produce a single fixed vector representation of input data, VAEs model the latent space as a probability distribution. This probabilistic nature enables VAEs to capture the uncertainty inherent in the data, allowing for a more nuanced representation of input encodings. In VAEs, the decoder samples from this probability distribution rather than relying on a deterministic latent vector.

The latent space in VAEs serves as a structured and continuous representation of the input data. This latent space facilitates smooth interpolation between different data points, enabling seamless transitions and ensuring that similar points in the latent space yield similar generations. This continuous nature of the latent space makes VAEs particularly suitable for tasks such as data generation and interpolation.



Variational Autoencoders (VAEs) represent this feature as a probabilistic distribution, allowing for variability in generated images by sampling from this distribution.

- Probabilistic Modeling

In VAEs, the latent space is modeled as a probability distribution, typically a multivariate Gaussian. This distribution is parameterized by mean and standard deviation vectors, produced by the encoder. The learned representation z is then sampled from this distribution and fed into the decoder.

- Loss Function

The VAE loss function combines reconstruction loss and KL divergence. The reconstruction loss measures how well the model reconstructs the input, while the KL divergence ensures the learned distribution resembles a chosen prior (usually Gaussian).

Together, these components encourage learning a latent representation that captures both data distribution and the specified prior.

- Encouraging Meaningful Latent Representations

Incorporating the KL divergence into the loss function encourages the VAE to learn a structured latent space where similar data points are closer.

This balance between latent loss and reconstruction loss is crucial: a smaller latent loss can lead to generated images that closely resemble the training set but may lack quality, while a smaller reconstruction loss results in well-reconstructed images during training but may hinder the generation of novel images.

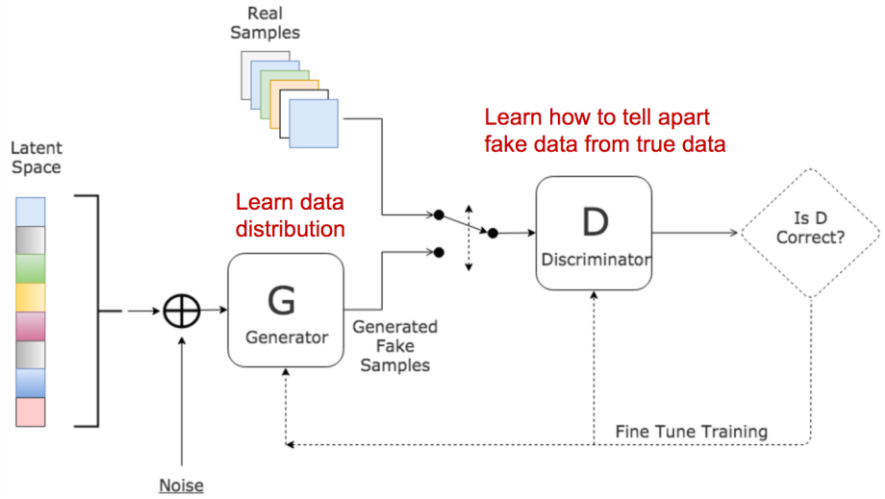
Achieving a balance between these two aspects is essential for optimal image reconstruction and generation.

3. Generative Adversarial Networks

Generative Adversarial Networks (GANs) were introduced by Goodfellow et al. in 2014 as a novel approach for generating new data samples by learning the statistical distribution of existing data through an adversarial process.

GANs consist of two sub-models trained simultaneously: **the Generator and the Discriminator**.

- The Generator learns to produce artificial samples from random vectors sampled from a defined latent space distribution.
- The Discriminator acts as a binary classifier, distinguishing between real and generated samples, and outputs 0 for artificial and 1 for real samples.



3.1.1.1. Training Process

GANs are trained through an adversarial game where the Generator tries to create realistic samples to fool the Discriminator, while the Discriminator aims to accurately distinguish between real and fake samples.

The models are improved iteratively until they reach a **Nash equilibrium**, where the Discriminator can no longer reliably tell the difference between real and generated samples, outputting equal probabilities for both. At this point, the GANs are considered to have converged, and the Generator can produce high-quality artificial samples.

$$\min_G \max_D f(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

However, this balance is difficult to achieve. The learning process consists of training two models simultaneously, most of the times they are affected by some instability, because each time the weights are updated, the improvements of one model appear to the detriment of the other. Formally, G and D play a minmax game where the evaluation function is defined by equation 1. When the training process is unstable, the model can end up in model collapse or other failure modes, producing undesired results.

3.1.1.2. GANs VS VAEs

Generative Adversarial Networks (GANs) excel in generating high-quality images with sharp details and realistic textures, making them ideal for tasks such as image generation.

However, their training process can be challenging and prone to instability, requiring careful tuning and management to achieve desirable results. Despite these challenges, GANs are well-suited for tasks like super-resolution and image-to-image translation, where they can produce visually stunning outputs.

On the other hand, **Variational Autoencoders (VAEs)** offer a more stable training process and are easier to train compared to GANs.

However, they may produce images with lower resolution and less detail, often appearing blurry or with unrealistic features.

VAEs are commonly used for tasks such as **image denoising and anomaly detection**, where their ability to learn meaningful latent representations proves beneficial.

Here's a table summarizing the key differences:

Feature	GANs	VAEs
Image Quality	Higher	Lower
Ease of Training	More difficult	Easier
Stability	Less Stable	More Stable
Applications	Image Generation, Super-resolution, image-to-image translation	Image Denoising, Anamoly Detection, Signal Analysis

While VAEs may not match the image quality of GANs, they still find utility in various applications that prioritize interpretability and stability over visual fidelity. Ultimately, the choice between GANs and VAEs depends on the specific requirements of the task at hand, weighing factors such as image quality, stability, and the nature of the data being generated or manipulated.

3.2. StyleGAN Variants

3.2.1.1. What is missing in Vanilla GAN?

Generative Adversarial Networks (GANs) generate realistic images **but lack control over specific feature**s in the generated images. In a standard/**Vanilla GAN** setup, the Generator creates images from **random noise**, while the Discriminator distinguishes between real and generated images. Despite training, traditional GANs don't allow for direct manipulation of image features, as these features are entangled.

StyleGAN addresses this limitation by **modifying the Generator** architecture while keeping the **Discriminator unchanged.** This new architecture **allows for explicit control over image features**. For example, with StyleGAN, you can generate images of a female wearing glasses by controlling high-level attributes (like gender and accessories) and low-level details (like skin texture and hair placement).

3.2.1.2. StyleGAN 1 components & benefits

StyleGAN improves upon traditional GANs by providing users with the ability to manipulate specific image characteristics, making it useful for tasks like privacy preservation and image editing.

The easiest way for GAN to generate **high-resolution images** is to remember images from the training dataset and while generating new images it can add random noise to an existing image. In

reality, StyleGAN doesn't do that rather it learn features regarding "human face" and generates a new image of the *"human face"* that doesn't exist in reality.

Diagram (a) shows ProgressiveGAN, a Vanilla GAN variant that progressively generates higher resolution images for more realistic results. The generator in ProgressiveGAN starts with low resolution (e.g., 4x4) and gradually increases it (e.g., 8x8).

Diagram (b) represents the StyleGAN architecture, which introduces several key components:

- **Mapping Network:** Transforms the input noise vector into an intermediate latent space.
- **AdaIN (Adaptive Instance Normalization):** Normalizes features to control style and content separately.
- **Noise Vector Concatenation**: Adds randomness to fine details, enhancing image diversity.

These modifications allow **StyleGAN** to generate images with controllable features, addressing the limitations of traditional GANs.

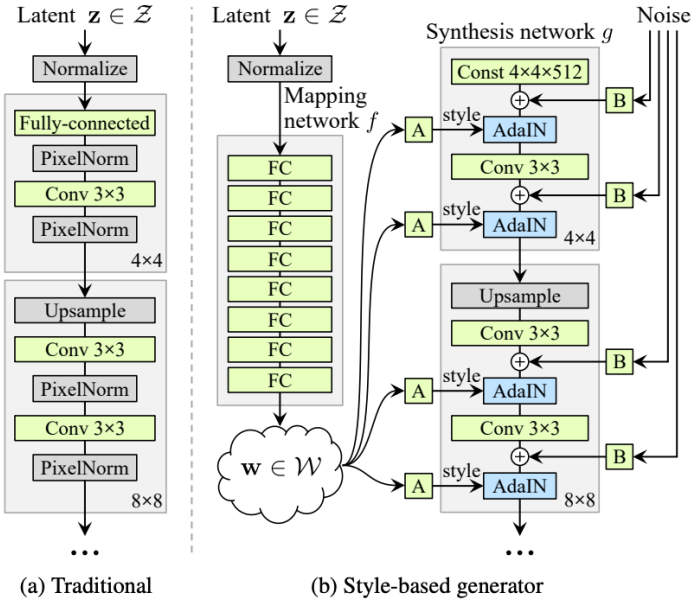


Figure 1. While a traditional generator [30] feeds the latent code though the input layer only, we first map the input to an intermediate latent space $\mathcal{W}$, which then controls the generator through adaptive instance normalization (AdaIN) at each convolution layer. Gaussian noise is added after each convolution, before evaluating the nonlinearity. Here "A" stands for a learned affine transform, and "B" applies learned per-channel scaling factors to the noise input. The mapping network f consists of 8 layers and the synthesis network g consists of 18 layers — two for each resolution ($4^2 - 1024^2$). The output of the last layer is converted to RGB using a separate 1×1 convolution, similar to Karras et al. [30]. Our generator has a total of 26.2M trainable parameters, compared to 23.1M in the traditional generator.

Mapping Network

In StyleGAN, instead of directly feeding the latent code (or noise vector) z to the Generator as in traditional GANs, it is first mapped to an intermediate latent space ww using a series of 8 Multi-Layer Perceptron (MLP) layers.

This process has two main **advantages**:

1. ***Disentanglement of Feature Space:*** Mapping z to w helps disentangle the feature space. In a disentangled space, changing a single feature value in the latent code should affect only one specific aspect of the generated image. For example, in a 512-dimensional latent code, if you alter the 4th value and this value corresponds to the **'smile'** feature, only the smile should change in the generated image.
2. ***Passing Latent Code to Each Layer:***
   i. Instead of providing the latent code w only to the initial layer of the Generator, StyleGAN passes w to every layer of the Synthesis Network (the Generator).
   ii. This approach allows different layers to control different aspects of the image. Lower layers, which handle lower resolutions, control high-level features like pose, general hairstyle, face shape, and eyeglasses.
   iii. Higher layers, which handle higher resolutions, control finer details like small-scale facial features, hairstyle intricacies, and whether the eyes are open or closed.

This enhances the ability to manipulate and generate images with specific desired characteristics.

Adaptive instance normalisation (AdaIN)

AdaIN (Adaptive Instance Normalization) in StyleGAN allows dynamic adjustment of normalization parameters (mean and standard deviation) based on style information derived from the latent code w.

Instead of directly using w , it is transformed into a 'style' representation y and applied to different blocks of the synthesis network. This enables the generator to modulate its behavior and apply different styles or characteristics to various parts of the generated image, enhancing control and flexibility in the image generation process.

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}$$

Concatenation of Noise vector

In traditional GANs, the generator must independently learn to create fine, stochastic features like hair positions and skin pores. This pixel-level randomness, which should vary across images, is difficult for the generator to achieve without explicit structure, often resulting in less diversity.

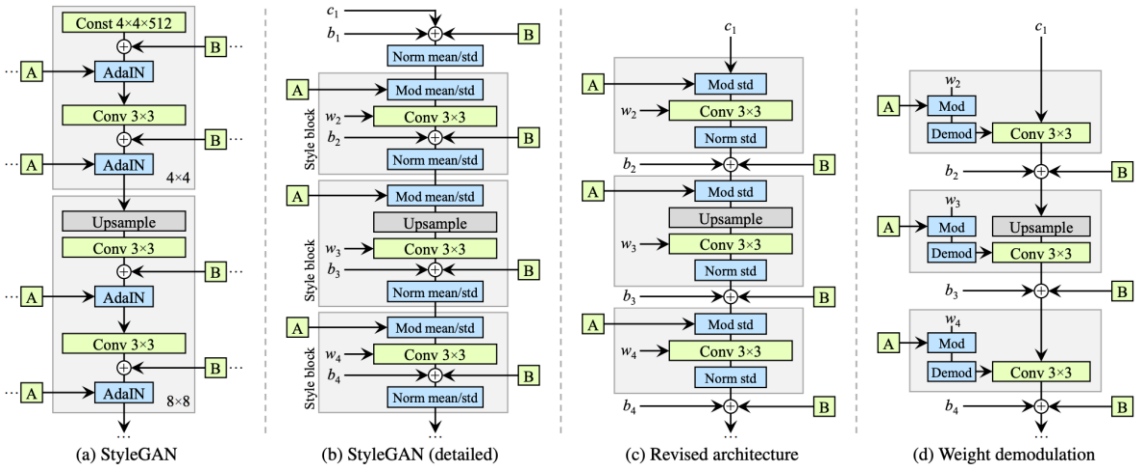
StyleGAN addresses this by **adding a noise map to the feature map** in each block of the synthesis network (generator). This noise map helps each layer introduce diverse stochastic details without relying solely on the generator's inherent capabilities. This approach effectively produces more varied and realistic fine details in the generated images.

(a) Generated image (b) Stochastic variation (c) Standard deviation

Figure 4. Examples of stochastic variation. (a) Two generated images. (b) Zoom-in with different realizations of input noise. While the overall appearance is almost identical, individual hairs are placed very differently. (c) Standard deviation of each pixel over 100 different realizations, highlighting which parts of the images are affected by the noise. The main areas are the hair, silhouettes, and parts of background, but there is also interesting stochastic variation in the eye reflections. Global aspects such as identity and pose are unaffected by stochastic variation.

StyleGAN is known for generating high-quality images, but it has some issues addressed in StyleGAN2.

3.2.1.3. StyleGAN 2 & StyleGAN 1



Blob-like Artifacts:

- Problem: In StyleGAN, there are common blob-like artifacts in the images, which are believed to come from the normalization process.

- Solution: StyleGAN2 introduces a new architecture that eliminates these blob-like artifacts by modifying how normalization is handled.

Location Preference Artifact:

- Problem: The original **StyleGAN** architecture, which uses progressive growing, tends to show a strong location preference. When changing the latent code w to adjust features like pose, the images often look unrealistic despite their high quality.

- Solution: StyleGAN2 avoids progressive growing and instead uses a skip generator and a residual discriminator. This approach helps produce more realistic images without the location preference issue.

Figure 6. Progressive growing leads to "phase" artifacts. In this example the teeth do not follow the pose but stay aligned to the camera, as indicated by the blue line.

These improvements in StyleGAN2 address key issues from the original StyleGAN, resulting in better image quality and more realistic outputs.

3.2.1.4. StyleGAN 2 & StyleGAN 1

The authors of StyleGAN2 discovered that the synthesis network's **reliance on absolute pixel** coordinates caused an aliasing effect. This issue means that when the latent code w is interpolated, textures in the generated images appear fixed in place, while only high-level attributes like face pose or expression change. This results in an artificial look in animations.

StyleGAN3 addresses this problem by **redesigning the architecture to remove this unhealthy dependence** on absolute pixel coordinates. The improvements in StyleGAN3 can be seen in animations where the textures move naturally along with the high-level attributes, resulting in more realistic and coherent changes.

3.2.1.5. StyleGAN kind use cases

StyleGAN's ability to produce **photorealistic images** has enabled various applications, including image editing, privacy preservation, and creative exploration.

In **image editing**, it excels in tasks like image inpainting and style transfer. For privacy-preserving applications, StyleGAN generates synthetic data to replace sensitive information and anonymizes images by altering identifiable features.

In creative fields, it generates diverse fashion designs and creates realistic virtual environments for gaming and education, exemplified by tools like **Stylenerf**, which offers style-based, 3D-aware high-resolution image synthesis.

3.3. CycleGAN Variants

CycleGAN, introduced by Zhu et al. in 2017, is a framework for image-to-image translation tasks that **doesn't** require paired examples. Traditional methods rely on large datasets of paired images, which can be difficult or impossible to obtain.

CycleGAN addresses this by enabling image translation **using unpaired datasets**, making it a significant advancement in computer vision and machine learning.

3.3.1. What Is Unpaired Image-to-Image Translation?

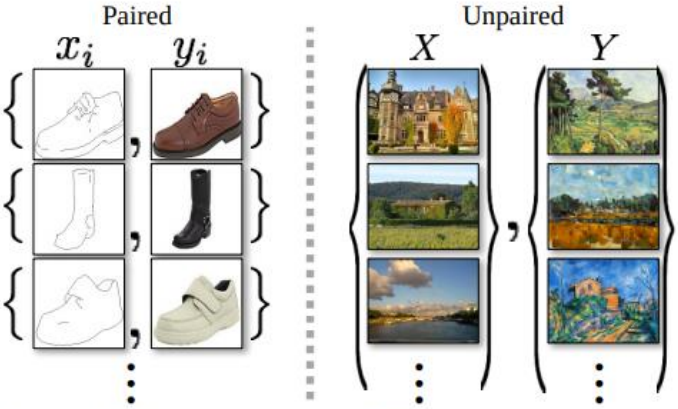


Figure 2: *Paired* training data (left) consists of training examples $\{x_i, y_i\}_{i=1}^N$, where the correspondence between x_i and y_i exists [22]. We instead consider *unpaired* training data (right), consisting of a source set $\{x_i\}_{i=1}^N$ $(x_i \in X)$ and a target set $\{y_j\}_{j=1}^M$ $(y_j \in Y)$, with no information provided as to which x_i matches which y_j.

In **unpaired image-to-image** translation, **datasets lack direct**, one-to-one image pairs. Instead, you have two distinct sets of images representing different styles or domains, such as realistic photographs and artworks, or different seasons like winter and summer.

The goal is for the model to learn the stylistic elements of each set and transform images from one domain to the other. This process works both ways, allowing for mutual transformation between the domains. This method is especially useful when exact image pairs are unavailable or difficult to obtain.

3.3.2. CycleGan components

CycleGAN employs two GANs (Generative Adversarial Networks), one for translating from the first set to the second (e.g., zebra to horse) and another for the reverse process (horse to zebra). This dual structure ensures realism (via the adversarial process) and content preservation (via cycle consistency).

3.3.2.1. Dual GAN Structure

CycleGAN uses two GANs to handle image translation between two domains, such as converting zebra images to horse images and vice versa.

- **PatchGAN Discriminators:** These discriminators evaluate patches of an image instead of the whole image, ensuring detailed and localized realism.
- **Generator Architecture:** The generators are based on U-Net and DCGAN architectures, involving encoding (downsampling), decoding (upsampling), and convolutional layers. They include residual connections to support deeper transformations and maintain identity functions.

3.3.2.2. Cycle Consistency Loss

Cycle consistency loss ensures that if you transform an image from one style to another and back again, it should closely resemble the original. This involves two stages:

1. **First Stage:** Transform the original image (e.g., sad face) to the target style (e.g., hugging face).
2. **Second Stage:** Transform the target style image back to the original style.

The loss is minimized by reducing the pixel difference between the original and the twice-transformed image. This loss is combined with the adversarial loss to form a comprehensive loss function for optimizing both generators.

3.3.2.3. Least-Square Loss

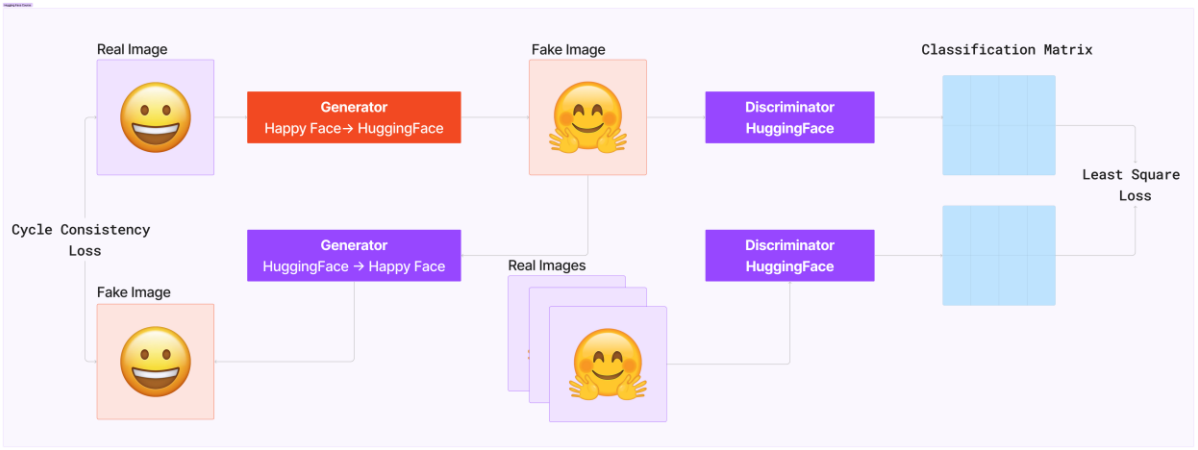
Least-square loss minimizes the squared differences between predicted values and actual labels (real or fake). For the **discriminator**, it reduces the squared distance between its predictions and **actual labels.** For the generator, it **aims to make its fake outputs appear real**.

This approach addresses issues like vanishing gradients and mode collapse seen with binary cross-entropy loss.

3.3.2.4. Identity Loss

Identity loss helps **preserve colors and finer details in generated images**. If an image from the target domain (e.g., a horse image) is input into the generator meant for transforming into the same domain (e.g., zebra-to-horse), the output should ideally be unchanged. The loss is calculated as the pixel distance between the input and the output image. This loss is added alongside adversarial and cycle consistency losses and is adjusted using a weighting factor (lambda term).

Summary: CycleGAN uses two GANs with PatchGAN discriminators and U-Net/DCGAN-based generators for image translation between unpaired domains. It employs **cycle consistency loss** to ensure realistic and consistent transformations**, least-square loss** for stability and accuracy, and optional **identity loss** to preserve image details. These combined techniques enable effective and realistic image-to-image translation without the need for paired datasets.



This figure shows the combined GAN architecture functionality for both GANs. These GANs are linked by **cycle consistency**, forming a cycle. For real images, the classification matrix contains ones. For fake images, it contains zeros. In summary, **CycleGAN** intricately combines two GANs with various loss functions, including **adversarial, cycle consistency, and optional identity loss**, to effectively transfer styles between two domains while preserving the essential characteristics of the input images.

Additional refs:

- *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*
- *FID*
- *SSIM*
- *Improved Techniques for Training GANs*
- *CLIPScore: A Reference-free Evaluation Metric for Image Captioning*
- *The Role of ImageNet Classes in Fréchet Inception Distance*
- *Lilian Weng's Awesome Blog on Autoencoders*
- *Generative models under a microscope: Comparing VAEs, GANs, and Flow-Based Models*
- *Autoencoders, Variational Autoencoders (VAE) and β-VAE*
- *Lilian Weng's Awesome Blog on GANs*
- *GAN — What is Generative Adversarial Networks*
- *What are the fundamental differences between VAE and GAN for image generation?*
- *Issues with GAN and VAE models*
- *VAE Vs. GAN For Image Generation*
- *StyleGAN3*
- *PatchGAN*
- *DCGAN*
- *Imagen*

4. Diffusion Models

Diffusion models are emerging in computer vision known for their ability to generate high-quality images.

These generative models work in two key stages: forward diffusion stage and the reverse diffusion stage.

- In the forward stage, the model gradually **adds noise** to the input data, effectively distorting it over time.
- In the reverse stage, the model learns to reverse this process, step by step, removing the noise to **reconstruct the original data.**

This iterative process of adding and then removing noise enables the generation of realistic and detailed images. These generative models have excelled in the field of generative modeling, especially with models like Imagen and **Latent Diffusion Models (LDMs).**

In the **forward Stage, Gaussian noise is added gradually** to the training images, eventually **turning** them **into completely noise, unrecognizable images.** Through this process, the model learns to remove the noise step-by-step, hence it is capable of turning any Gaussian noisy image into a new diverse image **(can also be conditioned based on text prompts).**

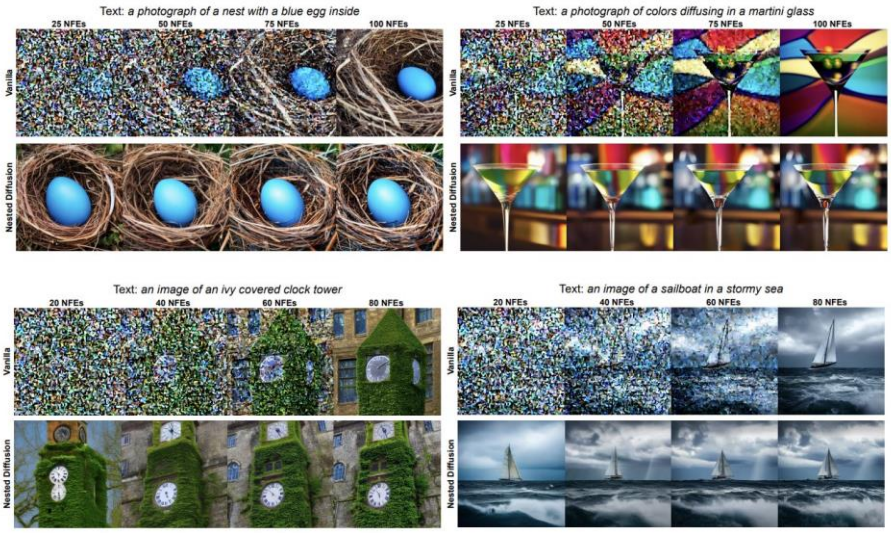


Figure 1. Results of intermediate predictions of Stable Diffusion from a reverse diffusion process of 100 steps (top) and 80 steps (bottom).

4.1.1. Variants of Diffusion models

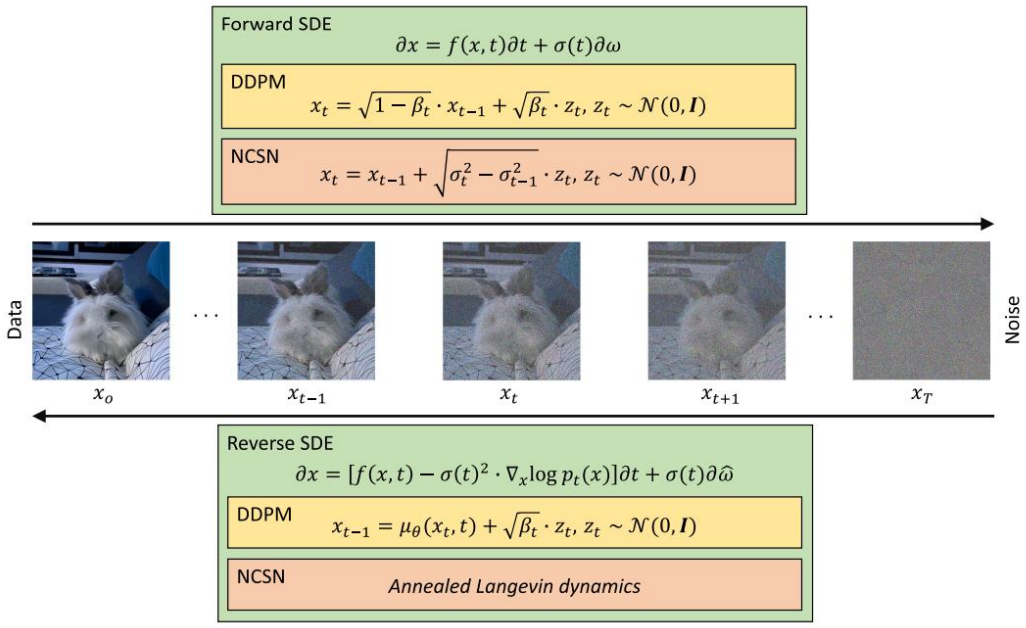
There are some major Diffusion models frameworks:

- *Denoising diffusion probabilistic models (DDPMs):* DDPMs are a foundational framework in diffusion models. DDPMs can be thought of as a specific type of generative model that uses latent variables to estimate the probability distribution of data, similar to how Variational Autoencoders (VAEs) work. In this analogy, **the two stages** of the diffusion process in DDPMs correspond to the encoding and decoding processes in VAEs.
  This reverse process is **probabilistic**, meaning the model predicts the probability distribution of the denoised image at each step, rather than a single deterministic output.

- Noise conditioned score networks (NCSNs): NCSNs are another type of diffusion model that leverages **score matching**, which involves **estimating the gradient (or score)** of the data distribution. It estimates the score function (defined as the gradient of the log density) of the perturbed data distribution at different noise levels.
  Instead of adding noise progressively like DDPMs, NCSNs work by **training a neural network to predict the score of noisy images**—essentially, how to move the image towards higher likelihood under the data distribution. The model then samples new images by following

these score estimates, which guide the noise removal process. This approach is more focused on learning the structure of the data directly from noisy samples.

- Stochastic differential equations (SDEs): It represents an alternative way to model diffusion. Modeling diffusion via forward and reverse SDEs leads to efficient generation strategies as well as strong theoretical results. This can be viewed as a **generalization over DDPMs and NCSNs.**
  They describe the process of adding and removing noise through differential equations that govern the random perturbations in the data. In the context of diffusion models, SDEs are used to model the evolution of data as it transitions from the original image to a completely noisy state and then back to a clean image. This continuous perspective allows for more flexible and theoretically grounded methods for sampling and denoising.



4.2. Diffusion models Vs GANs

Before the recent rise of diffusion models, GANs were widely regarded as the leading generative models in terms of the quality of the images they produced. However, **GANs** are **challenging to train** due to their **adversarial setup**, where a generator and discriminator compete against each other. This often leads to a problem known as **mode collapse.**

For instance, in the case of a model which task is to generate images of cats and dogs, which represent two different modes. If mode collapse occurs, the generator might only produce realistic images of either cats or dogs, but not both. This happens when the **discriminator,** which helps guide the generator, **gets stuck in a local minimum**, **consistently misclassifying one type of image** (either cat or dog) as fake.

In contrast, **diffusion models** offer a **more stable training** process and generate a **greater variety of images** because they are **based on likelihood, rather than an adversarial objective**.

One disadvantage is that **diffusion models** tend to be **computationally intensive** and require longer inference times compared to GANs due to the reverse step-by-step reverse process.

The **main drawback of diffusion models** is their **slow inference time**, requiring multiple steps to generate a single image. **Latent Consistency Models (LCMs)** have been proposed to speed up this process, allowing faster inference on pre-trained Latent Diffusion Models like **Stable Diffusion.** Besides that, **DDIM sampling** was proposed.

However, despite progress, GANs still outperform diffusion models in speed. Additionally, diffusion models using CLIP embeddings for text-to-image generation often struggle to render readable text, as CLIP embeddings lack detailed information about spelling, leading to issues with text accuracy in generated images.

4.2.1. Diffusion models' use cases.

Diffusion models have a wide range of applications, from **generating images based on text prompts (text-to-image)** to **enhancing image resolution, inpainting** damaged areas, and **editing images** while preserving their visual identity.

They are also used for tasks like **image-to-image** translation, such as changing backgrounds or attributes of a scene.

Beyond image manipulation, the **learned latent representations** from diffusion models can be leveraged for tasks like **image segmentation, classification, and anomaly detection.**

4.3. Implementation with Diffusers 🤗

Nowadays, many tools and libraries have made it easier to deploy and access high-performance models/approaches.
One popular library/API used is 🤗 Diffusers is divided into three main components:

1. **Pipelines**: high-level classes designed to rapidly generate samples from popular trained diffusion models in a user-friendly fashion.
2. **Models**: popular architectures for training new diffusion models, *e.g.* UNet.
3. **Schedulers**: various techniques for generating images from noise during *inference* as well as to generate noisy images for *training*.

Pre-define Pipelines are great for end-user, better they don't allow to control/adapt must for specific use cases. To create our own pipeline and train diffusion models the following steps must be implemented:

Step 1. *Load in some images from the training data (load dataset and data augmentation)*

This step includes the loading of the dataset, as well as compute data augmentation techniques. In this way, it is possible to guarantee the **standardization/formation of the data**, and improve the performance of the model, making it **robust** to perform adjustments/**variations in data**.

```python
import torchvision
from datasets import load_dataset
from torchvision import transforms

dataset = load_dataset("huggan/smithsonian_butterflies_subset", split="train")

# Or load images from a local folder
# dataset = load_dataset("imagefolder", data_dir="path/to/folder")

# We'll train on 32-pixel square images, but you can try larger sizes too
image_size = 32
# You can lower your batch size if you're running out of GPU memory
batch_size = 64

# Define data augmentations
preprocess = transforms.Compose(
    [
        transforms.Resize((image_size, image_size)),  # Resize
        transforms.RandomHorizontalFlip(),  # Randomly flip (data augmentation)
        transforms.ToTensor(),  # Convert to tensor (0, 1)
        transforms.Normalize([0.5], [0.5]),  # Map to (-1, 1)
    ]
)


def transform(examples):
    images = [preprocess(image.convert("RGB")) for image in examples["image"]]
    return {"images": images}


dataset.set_transform(transform)

# Create a dataloader from the dataset to serve up the transformed images in batches
train_dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Step 2. *Add noise, in different amounts (Define Scheduler)*

The noise schedule determines how much noise is added at different timesteps. In the following code we create a scheduler using the default settings for 'DDPM' training and sampling (based on the paper "Denoising Diffusion Probabilistic Models"):

```python
from diffusers import DDPMScheduler

noise_scheduler = DDPMScheduler(num_train_timesteps=1000)
```

According to this paper:

The DDPM paper describes a corruption process that adds a small amount of noise for every 'timestep'. Given x_{t-1} for some timestep, we can get the next (slightly more noisy) version x_t with:

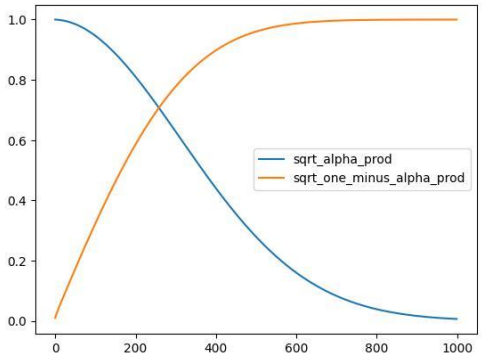
$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

That is, we take x_{t-1}, scale it by $\sqrt{1-\beta_t}$ and add noise scaled by β_t. This β is defined for every t according to some schedule, and determines how much noise is added per timestep. Now, we don't necessarily want to do this operation 500 times to get x_{500} so we have another formula to get x_t for any t given x_0:

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \text{ where } \bar{\alpha}_t = \prod_{i=1}^{T} \alpha_i \text{ and } \alpha_i = 1 - \beta_i$$

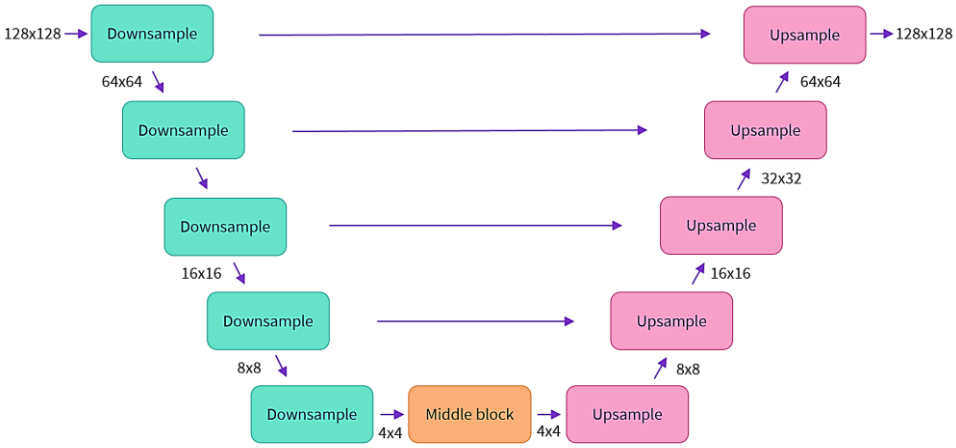
The maths notation always looks scary! Luckily the scheduler handles all that for us. We can plot $\sqrt{\bar{\alpha}_t}$ (labelled as `sqrt_alpha_prod`) and $\sqrt{(1-\bar{\alpha}_t)}$ (labelled as `sqrt_one_minus_alpha_prod`) to view how the input (x) and the noise are scaled and mixed across different timesteps:

By monitoring this process, we got the following graphs/plot below. In this the orange line corresponds to the 'noise' (which is almost zero at the beginning), and the blue one to the 'original image' (basically completely corrupted at the end of the noise process).



Step 3. *Feed the noisy versions of the inputs into the model. (Define Model)*

Most diffusion models use architectures that are some variants of a U-net. A key feature of this model is that it predicts images of the same size as the input.



Summary of U-net architecture:

- the model has the input image go through several **blocks of ResNet layers**, each of which halves the image size by 2.
- then through the same number of blocks that **upsample** it again.

- there are **skip connections linking** the features on the **downsample** path to the corresponding layers in the **upsample** path.

Using the Diffusers library this translates to the following:

```python
from diffusers import UNet2DModel

# Create a model
model = UNet2DModel(
    sample_size=image_size,  # the target image resolution
    in_channels=3,  # the number of input channels, 3 for RGB images
    out_channels=3,  # the number of output channels
    layers_per_block=2,  # how many ResNet layers to use per UNet block
    block_out_channels=(64, 128, 128, 256),  # More channels -> more parameters
    down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "DownBlock2D",
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "AttnDownBlock2D",
    ),
    up_block_types=(
        "AttnUpBlock2D",
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",
        "UpBlock2D",  # a regular ResNet upsampling block
    ),
)
model.to(device)
```

Note: When dealing with **higher-resolution inputs** you may want to use **more down and up-blocks** and keep the attention layers only at the lowest resolution (bottom) layers to reduce memory usage.

Step 4. *Evaluate how well the model does at denoising these inputs. (Training loop)*

Step 5. *Use this information to update the model weights, and repeat.*

The implementation of training Loop includes:

- Randomly select a set of timesteps.
- Add noise to the data based on the selected timesteps.
- Feed the noisy data through the model.
- Measure the difference between the model's predictions and the actual noise using mean squared error as the loss function.
- Update the model parameters via *loss.backward()* and *optimizer.step()*

```python
# Set the noise scheduler
noise_scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule="squaredcos_cap_v2")

# Training loop
optimizer = torch.optim.AdamW(model.parameters(), lr=4e-4)

losses = []

for epoch in range(30):
    for step, batch in enumerate(train_dataloader):
        clean_images = batch["images"].to(device)
        # Sample noise to add to the images
        noise = torch.randn(clean_images.shape).to(clean_images.device)
        bs = clean_images.shape[0]

        # Sample a random timestep for each image
        timesteps = torch.randint(0, noise_scheduler.num_train_timesteps, (bs,), device=clean_images.

        # Add noise to the clean images according to the noise magnitude at each timestep
        noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)

        # Get the model prediction
        noise_pred = model(noisy_images, timesteps, return_dict=False)[0]

        # Calculate the loss
        loss = F.mse_loss(noise_pred, noise)
        loss.backward(loss)
        losses.append(loss.item())

        # Update the model parameters with the optimizer
        optimizer.step()
        optimizer.zero_grad()

    if (epoch + 1) % 5 == 0:
        loss_last_epoch = sum(losses[-len(train_dataloader) :]) / len(train_dataloader)
        print(f"Epoch:{epoch+1}, loss: {loss_last_epoch}")
```

After training, how to get images with this model?

Step 6. *Generate Images*

```python
# Random starting point (8 random images):
sample = torch.randn(8, 3, 32, 32).to(device)

for i, t in enumerate(noise_scheduler.timesteps):

    # Get model pred
    with torch.no_grad():
        residual = model(sample, t).sample

    # Update sample with step
    sample = noise_scheduler.step(residual, t, sample).prev_sample

show_images(sample)
```

4.3.1. Implementation from Scratch VS DDPM

It is possible to implement the root diffusion model in pytorch, however this may have some additional challenges/differences. This section describes these differences:

- The **Diffusers UNet2DModel** is more advanced compared to the **basic UNet (in pytorch)**
- The **corruption process** is managed differently (usually linear and simpler in pytorch)
- The **training objective** focuses on **predicting the noise** rather than the denoised image.

```
noise = torch.randn_like(xb) # << NB: randn not rand
noisy_x = noise_scheduler.add_noise(x, noise, timesteps)
model_prediction = model(noisy_x, timesteps).sample
loss = mse_loss(model_prediction, noise) # noise as the target
```

  - Predicting the noise, rather than the denoised image, is more than just a mathematical convenience. It affects how the loss is weighted during training, particularly across different noise levels. By predicting noise, the model implicitly gives more weight to lower noise levels.
  - This can be adjusted by choosing more complex objectives, altering the noise schedule, or scaling the loss based on the noise level. Current research is exploring these different approaches to optimize model performance. For now, predicting noise is the preferred method, but alternative objectives may become more common as the field evolves.

- The **model is conditioned on noise** level through **timestep conditioning**, with **t** passed as an extra argument to the forward method.
  - The UNet2DModel uses both the input **x** and a **timestep**, which is converted into an embedding and integrated into the model.
  - This timestep conditioning provides the model with information about the noise level, helping it perform better. While not strictly necessary, including timestep conditioning is common in current implementations because it often improves performance.

- There are various **sampling strategies** available, which are expected to perform better than the basic approach.
  - **Instead of removing all the noise in one step**, we take multiple smaller steps, gradually reducing the noise based on the model's predictions. The process depends on the sampling method, which raises key design questions:

    i. How large should each step be? This relates to the noise schedule.
    ii. Should the update rely solely on the current model prediction, use higher-order gradients for accuracy, or incorporate past predictions? (DDPM or DDIM? ; higher-order methods and some discrete ODE solvers?; linear multi-step and ancestral samplers?)
    iii. Should additional noise be added for randomness, or should the process remain deterministic?

  These choices affect the quality and characteristics of the generated images.

4.4. Control over Diffusion Models

4.4.1. Need for Fine-tuning and Guidance

Although diffusion models and GANs can generate many unique images, they **can't always generate what you need exactly.** Furthermore, the training **diffusion models from scratch** can be **time-consuming!** Especially as we push to higher resolutions, the time and data required to train a model from scratch can **become impractical.**

Fortunately, there's a great solution: start with a pre-trained model! Instead of beginning with a model that knows nothing, you start with one that has already learned to perform a similar task, such as denoising images. This gives you a head start, as the model already has some **useful knowledge.**

Fine-tuning this pre-trained model usually **works best when** your new **data is somewhat similar** to the data the model was originally trained on. However, what's surprising is that fine-tuning **can still be effective** even if your **new data is quite different** from the original data. The knowledge it already has can still help, even when the task changes significantly.

Fine-tuning diffusion model

The implementation of fine-tuning a pre-trained model using the Diffusers library is quite similar to training a model from scratch, except that it starts with an existing model. Additionally, there's a small modification in the training loop: image_pipe.unet.parameters() is used to update the weights of the pre-trained model by setting the optimization target.

```
for epoch in range(num_epochs):
    for step, batch in tqdm(enumerate(train_dataloader), total=len(train_dataloader)):
        clean_images = batch["images"].to(device)
        # Sample noise to add to the images
        noise = torch.randn(clean_images.shape).to(clean_images.device)
        bs = clean_images.shape[0]

        # Sample a random timestep for each image
        timesteps = torch.randint(
            0,
            image_pipe.scheduler.num_train_timesteps,
            (bs,),
            device=clean_images.device,
        ).long()

        # Add noise to the clean images according to the noise magnitude at each timestep
        # (this is the forward diffusion process)
        noisy_images = image_pipe.scheduler.add_noise(clean_images, noise, timesteps)

        # Get the model prediction for the noise
        noise_pred = image_pipe.unet(noisy_images, timesteps, return_dict=False)[0]

        # Compare the prediction with the actual noise:
        loss = F.mse_loss(
            noise_pred, noise
        )  # NB - trying to predict noise (eps) not (noisy_ims-clean_ims) or just (clean_ims)

        # Store for later plotting
        losses.append(loss.item())

        # Update the model parameters with the optimizer based on this loss
        loss.backward(loss)

        # Gradient accumulation:
        if (step + 1) % grad_accumulation_steps == 0:
            optimizer.step()
            optimizer.zero_grad()
```

4.4.2. SOTA Techniques for Fine-tuning

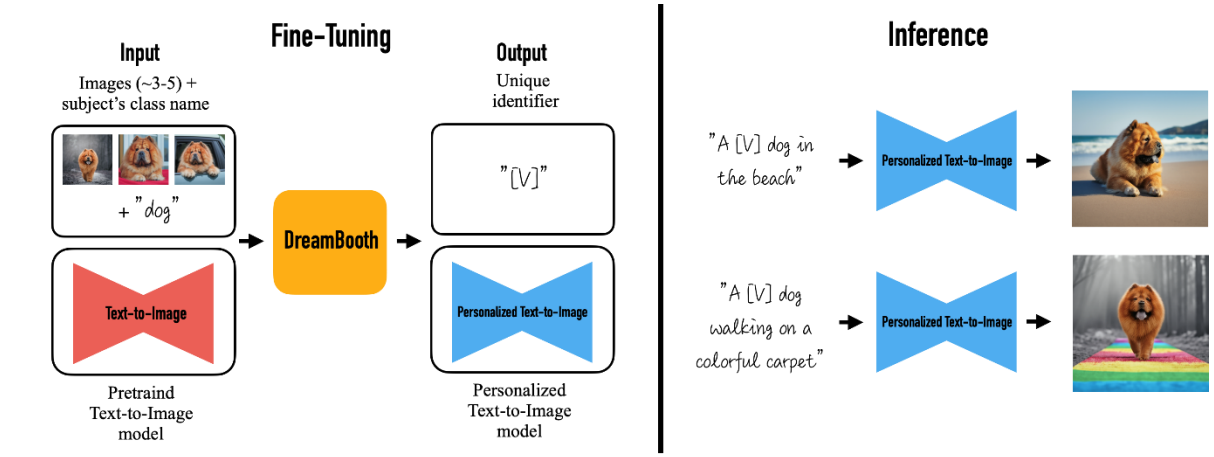
Although diffusion models and GANs can generate many unique images, they **can't always generate what you need exactly.** Some techniques can be used to personalize a model with just a few examples, avoiding the fine-tuning approach which usually requires a lot of data and computation.

A efficient method for customizing models is **Low Rank Adaptation (LoRA),** developed by Microsoft. LoRA improves the process of fine-tuning by focusing on updating only a small part of the model. It achieves this by breaking down the weight updates into two smaller, low-rank matrices while keeping the rest of the model fixed. This approach is more resource-efficient and faster, making it a popular choice for model adaptation.

Dreambooth is another technique for fine-tuning a text-to-image model to **'teach'** it a **new concept,** such as a specific object or style. This was developed by Google Research that allows for the customization of **stable-diffusion models** with just a few example images. Dreambooth fine-tunes a model by associating a unique keyword with a few images of a specific subject or style. This allows the model to generate new images of the subject in various poses and backgrounds.

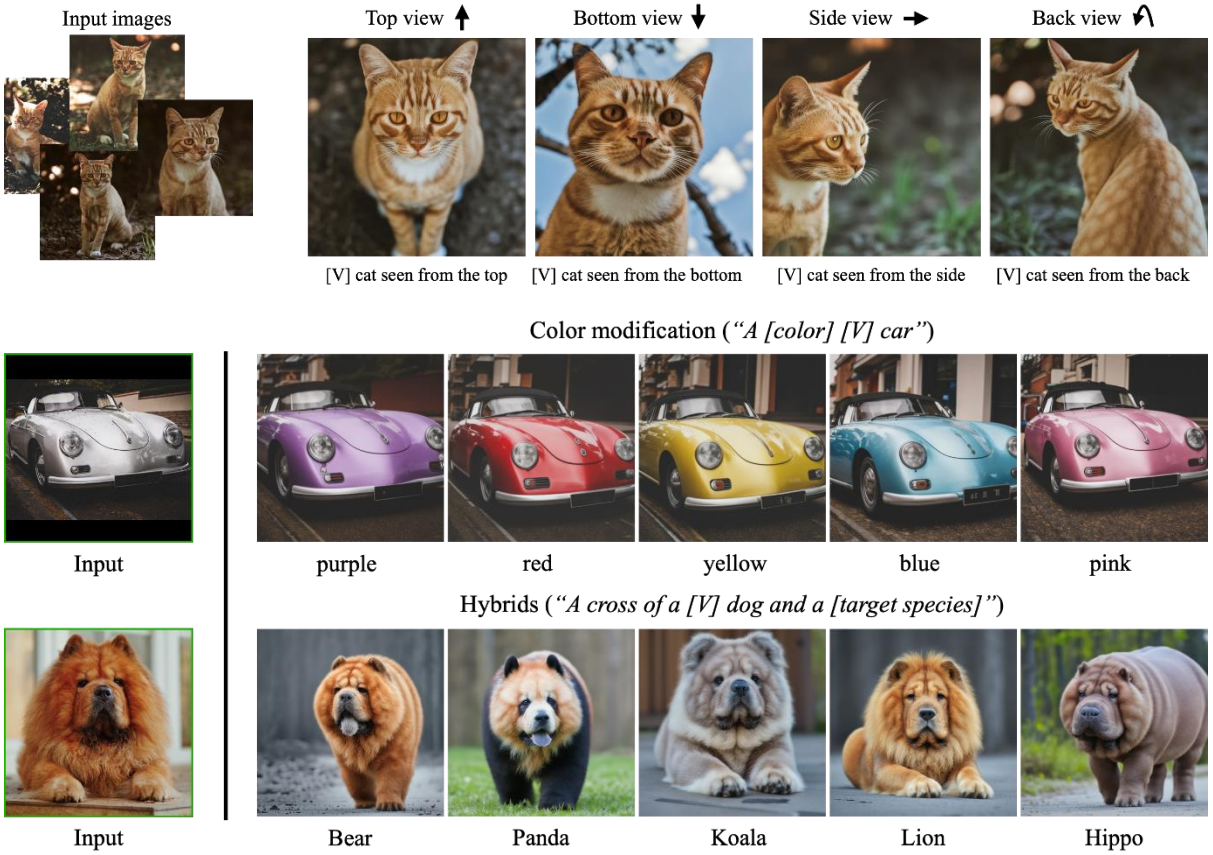
How does DreamBooth works?

1. **Collect Images**: Gather 10-20 images of the subject to customize by the model, for example "dog".
2. **Identifier**: Create a unique identifier, like "[V]" which will be used to represent the subject in text prompts during inference.
3. **Fine-Tune the Model:** Fine-tune the diffusion model by pairing each image with a text prompt that includes the unique identifier and the class name. For example, you might use the prompt "A photo of a [V] dog." This step teaches the model to associate the unique identifier with the specific subject.
4. (Optional) **Class-Specific Prior Preservation:** For the diversity and generalization within the subject's class (e.g., "dog"), you can apply a class-specific prior preservation loss. This step leverages the model's existing knowledge of the class to generate varied instances of the subject. It's particularly useful for human faces to ensure the model doesn't overfit and lose the ability to generate different faces within that class.



In addition to putting the subject in various locations of interess, DreamBooth can be used for **text-guided view synthesis**, allowing the subject to be viewed from different angles or perspectives.

DreamBooth can even **modify specific properties** of the subject, such as changing its color or blending characteristics from different animal species.



4.4.3. Guidance

Unconditional models on their own **don't provide much control** over what they create. **Guidance** is a method that helps **direct the model's predictions** at each step of the generation process. It **checks the model's output** against a guidance function and then tweaks the predictions so that the final image better matches what we're aiming for.

The great potential of the guidance function is that it can be **almost anything**, which makes this approach **very versatile!** For instance, you can control simple features like the color of the generated

image or take it further by using a more sophisticated pre-trained model like **CLIP**, which allows you to guide the generation based on a **text description/prompt.** This gives you much more control over what the final image looks like.

Implementing simple guidance

For cases where we aim to control simple characteristics of the generated images/data, this can be achieved by creating additional loss terms that account for those characteristics. For instance, in the case of color, a possible implementation would be:

```
def color_loss(images, target_color=(0.1, 0.9, 0.5)):
    """Given a target color (R, G, B) return a loss for how far away on average
    the images' pixels are from that color. Defaults to a light teal: (0.1, 0.9, 0.5)"""
    target = torch.tensor(target_color).to(images.device) * 2 - 1  # Map target color t
    target = target[None, :, None, None]  # Get shape right to work with the images (b,
    error = torch.abs(images - target).mean()  # Mean absolute difference between the i
    return error
```

Regarding the possible approaches for weights update, there are two options:

- set requires_grad on x **after** we get our noise prediction from the UNet, which is more memory efficient but gives a less accurate gradient.
- set requires_grad on x **first**, then feed it through the UNet and calculate the predicted x0. In this the outputs are arguably closer to the types of images the model was trained on.

CLIP Guidance

By guide the focus towards a specific color, we gain some level of control, but imagine if we could simply describe what we want in words instead.

Enter **CLIP,** a model developed **by OpenAI** that can evaluate the alignment between images and text descriptions. This capability is incredibly useful because it lets us measure how closely an image corresponds to a given prompt. Because this comparison process is differentiable, it can be used as a loss function to steer the training of our diffusion models!

*Here're the steps for **Clip**:*

Step 1. Generate a 512-dimensional CLIP embedding for the text prompt.

Step 2. **Create several versions** of the predicted denoised image (having multiple variations helps provide a clearer loss signal).

Step 3. For each version, compute its CLIP embedding and compare it to the text embedding of the prompt using a metric known as **'Great Circle Distance Squared.'**

Step 4. Compute the gradient of this loss with respect to the current noisy image and use this gradient to adjust the image before updating it with the scheduler.

4.4.4. SOTA Method for Guidance

Guided Diffusion via ControlNet

Diffusion models can be guided in various ways to produce specific outputs, such as using text **prompts, negative prompts, guidance scales, or inpainting.**

A notable method for enhancing guidance is **ControlNet**, introduced by Stanford University. ControlNet allows for more precise guidance by using an image that provides detailed information, like depth, pose, or edges, to steer the diffusion process. This helps achieve more consistent and controlled outputs from diffusion models, which often struggle with maintaining coherence.

ControlNet can be used in both text-to-image and image-to-image tasks. For example, if you input an image with edge information, ControlNet helps the model generate new images that follow the same shape as the input but with different colors. This makes the generated images more aligned with the provided guidance, ensuring the output closely matches the desired features.

4.4.5. Conditioning

Guidance is a useful technique to get more out of an unconditional diffusion model, but if we have extra information, like a class label or image caption, we can use this to create a conditional model. **A conditional model** allows us to control the output by feeding in specific data during inference. For example, **class-conditioned model** is trained to generate images based on **class labels.**

There are several ways to incorporate this conditioning information:

1. **Adding Channels:** If the conditioning info is similar in shape to the image (like a segmentation mask or depth map), it can be added as extra channels in the model's input.
2. **Embedding Projections:** create an embedding and then scale it to match the number of channels in the UNet's internal layers, adding it to the outputs. This method is used for timestep conditioning, where each Resnet block's output has a corresponding timestep embedding added. It's particularly useful when the conditioning information is a vector, like a CLIP image embedding.
3. **Cross-Attention Layers:** can be added to the model to focus on a sequence of conditioning data, like text. When using text as conditioning, it's converted into embeddings via a transformer model. These embeddings are then integrated into the UNet through cross-attention layers, influencing the image generation process. This approach is particularly effective for models like Stable Diffusion.

Implementation Class-Conditioned Diffusion Model

The implementation of Class-conditioned Diffusion model implies small changes in Unet and training, sampling techniques, such as:

```python
class ClassConditionedUnet(nn.Module):
    def __init__(self, num_classes=10, class_emb_size=4):
        super().__init__()

        # The embedding layer will map the class label to a vector of size class_emb_size
        self.class_emb = nn.Embedding(num_classes, class_emb_size)

        # Self.model is an unconditional UNet with extra input channels to accept the conditioning infor
        self.model = UNet2DModel(
            sample_size=28,  # the target image resolution
            in_channels=1 + class_emb_size,  # Additional input channels for class cond.
            out_channels=1,  # the number of output channels
            layers_per_block=2,  # how many ResNet layers to use per UNet block
            block_out_channels=(32, 64, 64),
            down_block_types=(
                "DownBlock2D",  # a regular ResNet downsampling block
                "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
                "AttnDownBlock2D",
            ),
            up_block_types=(
                "AttnUpBlock2D",
                "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
                "UpBlock2D",  # a regular ResNet upsampling block
            ),
        )

    # Our forward method now takes the class labels as an additional argument
    def forward(self, x, t, class_labels):
        # Shape of x:
        bs, ch, w, h = x.shape

        # class conditioning in right shape to add as additional input channels
        class_cond = self.class_emb(class_labels)  # Map to embedding dimension
        class_cond = class_cond.view(bs, class_cond.shape[1], 1, 1).expand(bs, class_cond.shape[1], w, h)
        # x is shape (bs, 1, 28, 28) and class_cond is now (bs, 4, 28, 28)

        # Net input is now x and class cond concatenated together along dimension 1
        net_input = torch.cat((x, class_cond), 1)  # (bs, 5, 28, 28)

        # Feed this to the UNet alongside the timestep and return the prediction
        return self.model(net_input, t).sample  # (bs, 1, 28, 28)
```

- Add additional input channels to Unet
- Map the class label via an embedding layer (class_emb_size)
- Concatenate this information as extra channels for the internal UNet (net_input)

While training the correct labels are added as a third argument (prediction = unet(x, t, y)). The same applies to prediction/test, we can sample some images feeding in different labels as our conditioning

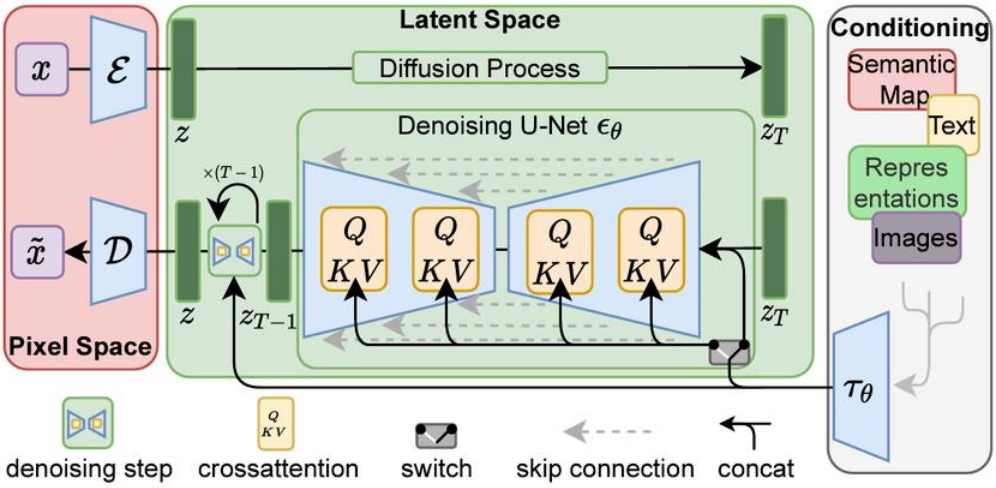
```python
# Our network
net = ClassConditionedUnet().to(device)

# Our loss function
loss_fn = nn.MSELoss()

# The optimizer
opt = torch.optim.Adam(net.parameters(), lr=1e-3)

# Keeping a record of the losses for later viewing
losses = []

# The training loop
for epoch in range(n_epochs):
    for x, y in tqdm(train_dataloader):

        # Get some data and prepare the corrupted version
        x = x.to(device) * 2 - 1  # Data on the GPU (mapped to (-1, 1))
        y = y.to(device)
        noise = torch.randn_like(x)
        timesteps = torch.randint(0, 999, (x.shape[0],)).long().to(device)
        noisy_x = noise_scheduler.add_noise(x, noise, timesteps)

        # Get the model prediction
        pred = net(noisy_x, timesteps, y)  # Note that we pass in the labels y

        # Calculate the loss
        loss = loss_fn(pred, noise)  # How close is the output to the noise

        # Backprop and update the params:
        opt.zero_grad()
        loss.backward()
        opt.step()

        # Store the loss for later
        losses.append(loss.item())

    # Print out the average of the last 100 loss values to get an idea of progress:
    avg_loss = sum(losses[-100:]) / 100
    print(f"Finished epoch {epoch}. Average of the last 100 loss values: {avg_loss:05f}")
```

5. Stable Diffusion

Stable Diffusion is a type of Latent Diffusion Model (LDM) that gained significant attention for its ability to **generate high-quality images from text prompts**. Unlike traditional diffusion models, which operate directly on image pixels, **Stable Diffusion works in a lower-dimensional latent space**, making the process **more efficient** and allowing for faster image generation.

This approach involves **first mapping the image to a latent space,** applying the diffusion process within this space, **and then reconstructing the final image.**

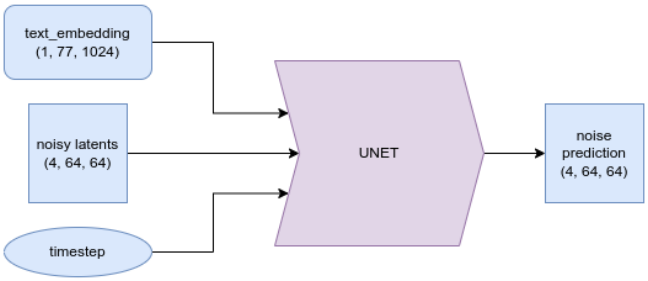
Stable Diffusion is particularly popular because it balances high image quality with relatively fast inference times compared to other diffusion models. However, like other diffusion models, it **still requires multiple steps to generate an image**, which can be slower than alternative methods like GANs.



Stable Diffusion is a powerful model for generating images from text prompts. Key components of the Stable Diffusion process include:

- **Image Compression:** To handle large images efficiently, Stable Diffusion **uses a Variational Auto-Encoder (VAE) to compress images** into a smaller, more manageable latent space. This compression is crucial because processing high-resolution images directly would require exponentially more computing power due to the quadratic increase in calculations with image size.

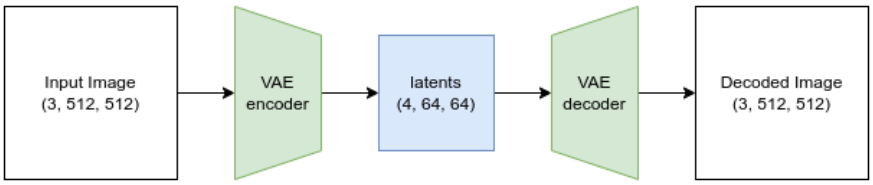
- **Text and Image Fusion:** For text-to-image generation, ***Stable Diffusion*** utilizes a **pre-trained transformer model known as CLIP**. CLIP encodes text prompts into high-dimensional vectors, which guide the diffusion process. During inference, a **text prompt is converted into a fixed-size vector representation,** which is **then used to condition** the model's denoising process.



- **Inductive Biases with U-Net and Cross-Attention:** To generate new and **diverse images,** Stable Diffusion employs a U-Net architecture with cross-attention mechanisms. The U-Net predicts the denoised image from noisy input, while cross-attention layers enable the model to incorporate relevant information from the text prompt at various spatial locations in the image.

5.1. Latent Diffusion

As image size grows, so does the **computational power required** to work with those images. This is especially pronounced in an operation called self-attention, where the number of operations grows quadratically with the number of inputs.



Latent diffusion helps to mitigate this issue by using a separate model called a Variational Auto-Encoder (VAE) to **compress** images to a smaller spatial dimension. By applying the diffusion process on these **latent representations** rather than on full-resolution images, we can get many of the benefits that would come from using smaller images (lower memory usage, fewer layers needed in the UNet, faster generation times…) and still decode the result back to a high-resolution image.

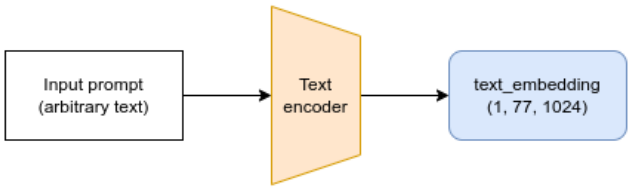
Although the decoding isn't perfect and can introduce minor quality tradeoffs, the efficiency gained by working with latents generally outweighs these small imperfections.

5.2. Text Conditioning

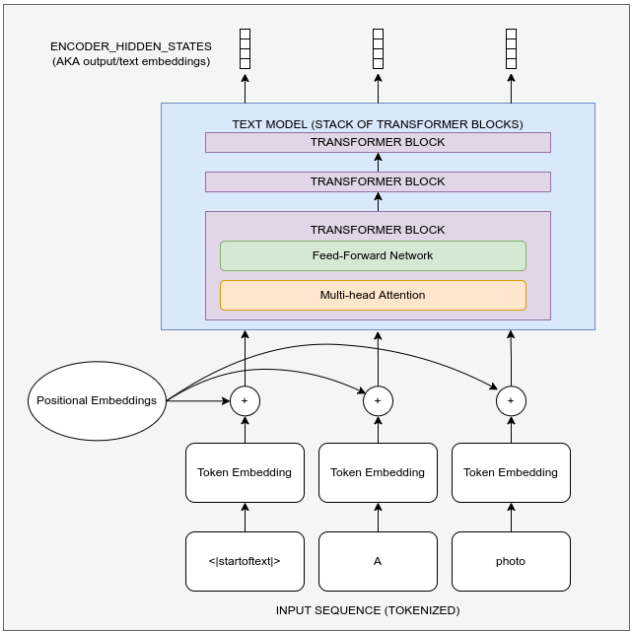
To have control over the images generated by a diffusion model like Stable Diffusion, **additional information** is incorporated into the **UNet architecture**—a process known as **conditioning**. This conditioning **allows** the model to **predict a less noisy version** of an image based on **extra cues**, such as a class label or, in the case of Stable Diffusion, a text description.

During inference, you start with random noise and a text prompt describing the image you want. The model then progressively refines this noise into an image that aligns with the given description.

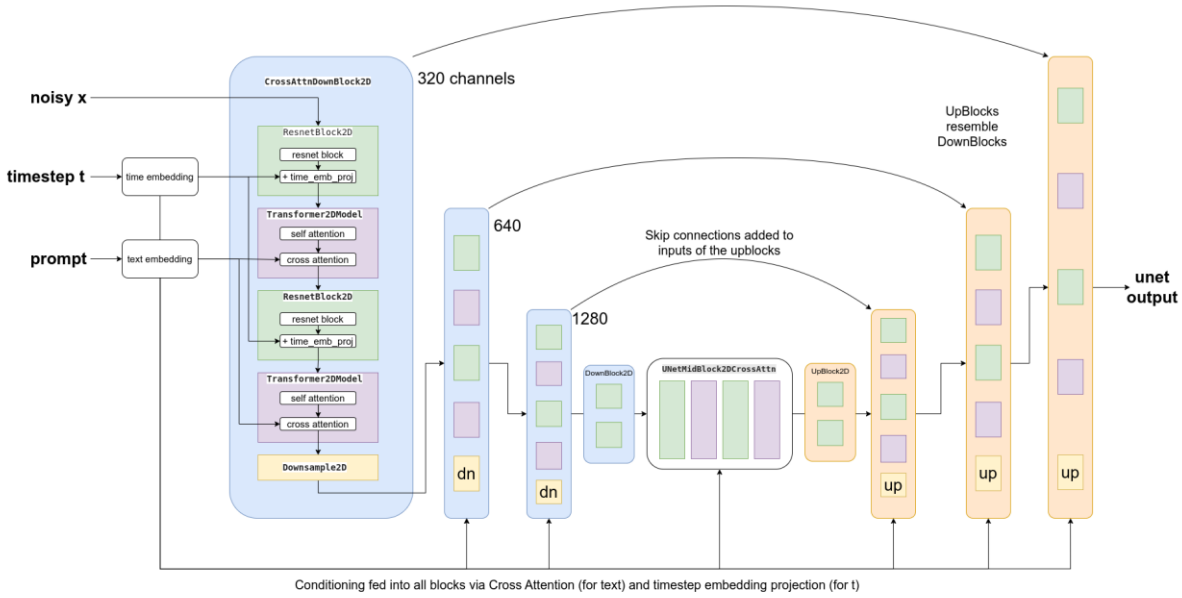
To encode the text prompt into a numerical form that the model can work with, Stable Diffusion uses a pre-trained transformer model called CLIP.



CLIP was originally designed to link images with their captions, making it ideal for converting text descriptions into useful numerical representations. When you input a prompt, it's first tokenized—each word or sub-word is converted into a token from a large vocabulary. This tokenized input is then processed by **CLIP's text encoder**, which outputs a vector representation for each token



The key to integrating this conditioning information into the UNet lies in a mechanism called cross-attention. **Cross-attention layers** are strategically placed throughout the UNet. These layers allow each **spatial location in the UNet to focus on different tokens** from the text conditioning, effectively pulling in relevant details from the prompt to guide the image generation process. This enables the model to align the generated image closely with the input text description.



NOTE: Despite all the effort put into making text conditioning effective, the model often still prioritizes the noisy input image over the text prompt when making predictions. *This happens because the model has learned that many captions are closely connected to their images, so it doesn't rely heavily on the descriptions*. This becomes a problem during image generation—if the model doesn't follow the prompt closely, the resulting images might not match the intended description at all. To fix this, a trick used is **Classifier-Free Guidance (CGF).**

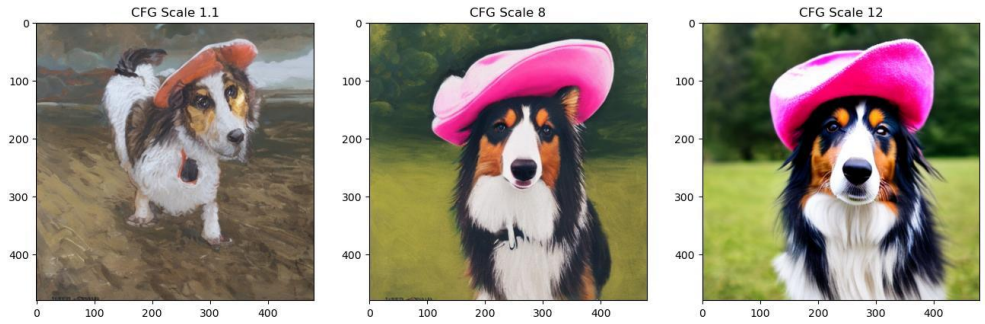
Classifier-free Guidance

In **CGF** training, the model sometimes **learns to denoise images without any text conditioning**, a process known as **unconditional generation**. This helps the model handle cases where no text information is available.

At **inference time**, however, we take advantage of both approaches by making two predictions:

- one with the text prompt as conditioning
- other without.

The key is to compare these two predictions. The **difference between them highlights** the influence of the text prompt. By amplifying this difference using a **scaling factor** called the **guidance scale (CFG scale)**, we can steer the final image more strongly in the direction suggested by the text.



This method increases the likelihood that the generated image closely matches the prompt. Higher guidance scales typically produce images that align better with the description.

5.3. Other Tasks & Types of Conditioning

Diffusion models are versatile and can be applied to a wide range of tasks beyond just generating images from text prompts. These tasks include image-to-image (**img2img**), **in-painting** (filling in missing parts/regions of an image), and **super-resolution** (increasing image resolution), etc.

In most cases, achieving these different tasks is simply a matter of applying the appropriate type of conditioning to the model.

5.3.1. Depth-2-Image

Img2Img is great, but sometimes we want to create a new image with the composition of the original but completely **different colors or textures**. It can be difficult to find an Img2Img strength that preserves what we'd like of the layout without also keeping the input colors. The depth information can be used conditioning the model to generate images to (hopefully) **preserve the depth** and structure of the initial image while **filling in completely new content**.

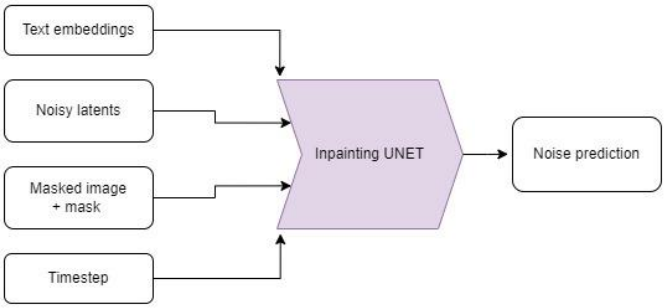
Depth-to-Image model has extra input channels that take in-depth information about the image being denoised, and at inference time we can feed in the depth map of a target image (estimated using a separate model) to hopefully generate an image with a similar overall structure.

5.3.2. Super-Resolution

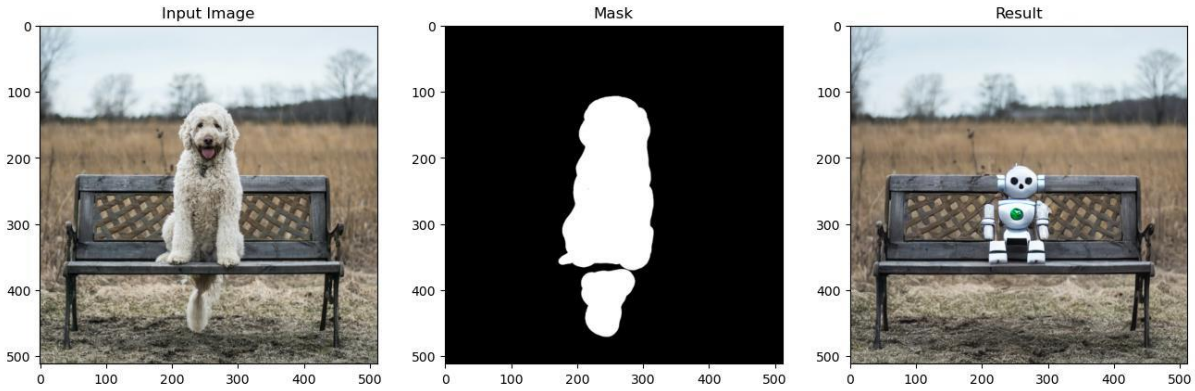
It is possible feed in a low-resolution image as the conditioning and have the model generate the high-resolution version (as used by the Stable Diffusion Upscaler).

5.3.3. Inpainting

What if we wanted to keep some of the input image unchanged but generate something new in other parts?



By feeding a mask showing a region of the image to be re-generated as part of the task. where the non-mask regions need to stay intact while new content is generated for the masked area.



Note: The mask image should be the same shape as the input image, with white in the areas to be replaces and black in the areas to be kept unchanged.

Additional refs:

- *Lilian Weng's Awesome Blog on GANs*
- *GAN — What is Generative Adversarial Networks*
- *Diffusion Models vs. GANs vs. VAEs: Comparison of Deep Generative Models*
- *U-Net*
- *Latent Diffusion Models*
- *Diffusers installation*
- *Unconditional image generation*
- *Diffusers*
- *OpenAI's GLIDE*
- *DDPM - Diffusion Models Beat GANs on Image Synthesis*
- *Denoising Diffusion Probabilistic Models*
- *Elucidating the Design Space of Diffusion-Based Generative Models*
- *Stable Diffusion Dreambooth Concepts Library*
- *High-Resolution Image Synthesis with Latent Diffusion Models*
- *CLIP*
- *GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models an early paper demonstrating text conditioning and CFG.*
- *Cold Diffusion: Inverting Arbitrary Image Transforms Without Noise*
- *Scalable Diffusion Models with Transformers (DiT)*
- *MaskGIT: Masked Generative Image Transformer*
- *Muse: Text-To-Image Generation via Masked Generative Transformers*
- *Fast Text-Conditional Discrete Denoising on Vector-Quantized Latent Spaces (Paella)*
- *Recurrent Interface Networks*

6. Key Innovations & Breakthroughs in Diffusion Models

Many improvements and extensions to diffusion models appearing in the latest research. This section summarizes/explores some of the main research topics.

6.1. Faster Sampling

One problem with the DDPM process is the speed of generating an image after training.

6.1.1. DDIM

The DDIM paper introduces a way to speed up image generation with little image quality tradeoff. It does so by redefining the diffusion process as a non-Markovian process.

Diffusion Models generate new samples by simulating a reversible Markov chain that gradually adds noise to input data and then learns a reverse process to transform the noise back into the original data type, enabling the generation of images, sound clips, and more.

Implicit Models represent relationships between inputs and outputs through a set of equations rather than a direct mathematical formula, offering flexibility in capturing complex relationships.

Deep Diffusion Implicit Models (DDIMs) combine the principles of diffusion models with the flexibility of implicit models, using diffusion processes for sample generation while leveraging implicit representations to manage complexity and enhance efficiency.

Key features of DDIM:

I. Non-Markovian Diffusion Processes: DDIM uses a non-Markovian diffusion process, which is different from the traditional Markovian process used in diffusion models like Denoising Diffusion Probabilistic Models (DDPM). This change enables faster sampling by reducing the number of steps required to generate a sample.

II. Implicit Probabilistic Model: DDIM is an implicit probabilistic model, meaning that it does not directly model the joint distribution of the data but instead models the conditional distribution of the data given the noise. This approach allows for efficient inference and faster sampling.

III. Training Objective: DDIM uses the same training objective as DDPM, which is to minimize the difference between the generated data and the original data. This ensures that the generated samples are of high quality and similar to the original data.

IV. Sampling Process: The sampling process in DDIM involves sampling from the prior distribution and then iteratively sampling from the conditional distributions. This process is faster than traditional diffusion models because it does not require simulating the entire Markov chain.

DDIM Sampling

At a given time t, the noisy image x_t is some mixture of the original image (x_0) and some noise (ϵ). Here is the formula for x_t from the DDIM paper, which we'll be referring to in this section:

$$x_t = \sqrt{\alpha_t} x_0 + \sqrt{1-\alpha_t}\epsilon$$

ϵ is some gaussian noise with unit variance α_t ('alpha') is the value which is confusingly called $\bar{\alpha}$ ('alpha_bar') in the DDPM paper (!!) and defined the noise scheduler. In Diffusers, the alpha scheduler is calculated and the values are stored in the `scheduler.alphas_cumprod`. Initially (timestep 0, left side of the graph) we begin with a clean image and no noise. $\alpha_t = 1$. As we move to higher timesteps, we end up with almost all noise and α_t drops towards 0.

During sampling, we begin with pure noise at timestep 1000 and slowly move towards timestep 0. To calculate the next t in the sampling trajectory (x_{t-1} since we're moving from high t to low t) we predict the noise ($\epsilon_\theta(x_t)$, which is the output of our model) and use this to calculate the predicted denoised image x_0. Then we use this prediction to move a small distance in the 'direction pointing to x_t'. Finally, we can add some additional noise scaled by σ_t. Here's the relevant section from the paper showing this in action:

4.1 Denoising Diffusion Implicit Models

From $p_\theta(x_{1:T})$ in Eq. (10), one can generate a sample x_{t-1} from a sample x_t via:

$$x_{t-1} = \sqrt{\alpha_{t-1}} \underbrace{\left(\frac{x_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\alpha_t}} \right)}_{\text{“predicted } x_0\text{”}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2} \cdot \epsilon_\theta^{(t)}(x_t)}_{\text{“direction pointing to } x_t\text{”}} + \underbrace{\sigma_t \epsilon_t}_{\text{random noise}} \tag{12}$$

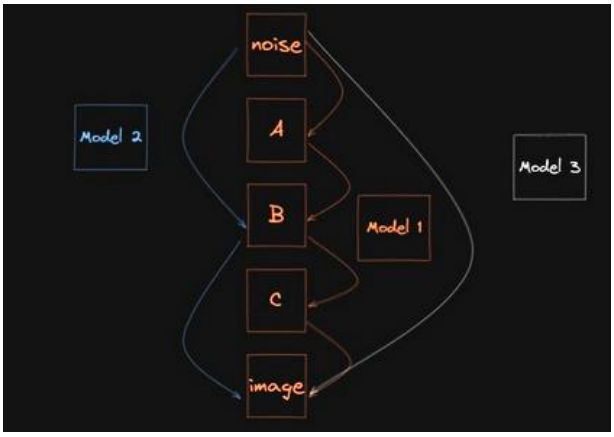
where $\epsilon_t \sim \mathcal{N}(0, I)$ is standard Gaussian noise independent of x_t, and we define $\alpha_0 := 1$. Different choices of σ values results in different generative processes, all while using the same model ϵ_θ, so re-training the model is unnecessary. When $\sigma_t = \sqrt{(1-\alpha_{t-1})/(1-\alpha_t)}\sqrt{1-\alpha_t/\alpha_{t-1}}$ for all t, the forward process becomes Markovian, and the generative process becomes a DDPM.

We note another special case when $\sigma_t = 0$ for all t[5]; the forward process becomes deterministic given x_{t-1} and x_0, except for $t = 1$; in the generative process, the coefficient before the random noise ϵ_t becomes zero. The resulting model becomes an implicit probabilistic model (Mohamed & Lakshminarayanan, 2016), where samples are generated from latent variables with a fixed procedure (from x_T to x_0). We name this the *denoising diffusion implicit model* (DDIM, pronounced /d:ɪm/), because it is an implicit probabilistic model trained with the DDPM objective (despite the forward process no longer being a diffusion).

6.1.2. Via Distillation

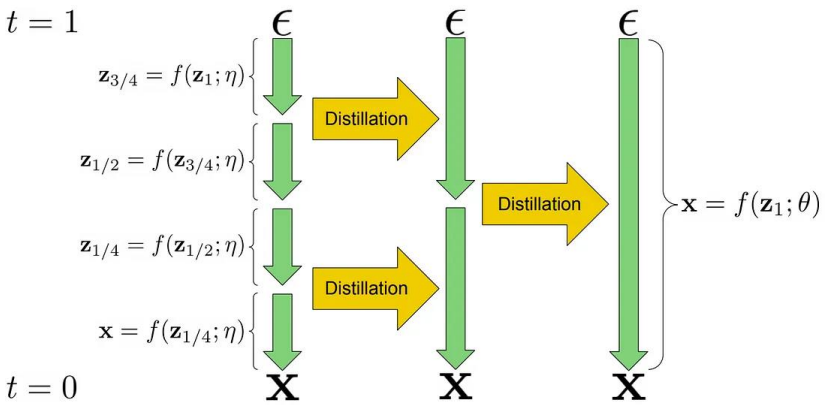
Another approach to tackle the sampling constraint is to apply distillation techniques to shorten the process.

Progressive distillation is a method designed to make diffusion models faster at generating images or data, without losing much quality. The main idea is to use a trained model (with good performance), that works as a teacher, to train a new model (student) that can maintain the quality of the generated data in less steps.



Here's how it works:

- **Teacher and Student Models**: Start with a diffusion model that's already been trained. This is called the "teacher" model. Then create a "student" model, which begins with the same weights (parameters) as the teacher model.
- **Training Process:** The goal is to train the student model to do in fewer steps what the teacher model does in more steps. Specifically, during training, the teacher model generates a sample by taking two steps. The student model tries to mimic that result, but it has to do it in just one step. This process forces the student model to learn how to produce good results more efficiently.
- **Iterative Refinement:** Once the student model has been trained to match the teacher model in fewer steps, this student model can then **become the new teacher**. The process is repeated—this time, the new student might be trained to produce similar results in even fewer steps. This iterative refinement can continue until the student model is able to generate samples in very few steps (like 4 or 8), while still maintaining quality similar to the original teacher model.
- **Result:** The final product of this process is a model that can generate high-quality results much faster than the original model, because it needs fewer steps to reach a good output. Progressive distillation teaches a model to **"shortcut"** the lengthy generation process of a diffusion model, gradually compressing it without sacrificing much accuracy.



Note: According to the paper, in this case/technique, we shouldn't predict the noise since it doesn't work that well. Instead, we should predict the image or image + noise.

Recent advancements have been made in accelerating the inference process of **classifier-free guided diffusion models**, which are known for their effectiveness in high-resolution image generation but are computationally expensive due to the need to evaluate **both a class-conditional and an unconditional model** multiple times during sampling.

A new approach has been proposed that distills these models into a single model, which combines the outputs of both the conditional and unconditional models. This combination is achieved by using a parameter, **w,** which represents the weight of the guides. The distilled model is then further refined to reduce the number of sampling steps required for high-quality image generation.

Additional refs:

- *Progressive Distillation For Fast Sampling Of Diffusion Models*
- *On Distillation Of Guided Diffusion Models*
- *Progressive Distillation illustrated*
- *Progressive Distillation for Fast Sampling of Diffusion Models (paper sumary)*
- *On Distillation of Guided Diffusion Models*
- *The Paradox of diffusion distillation*

6.2. Training Improvements

Efforts have been made to improve diffusion model training. Here's a summary of key improvements in diffusion model training, each briefly introduced to highlight its significance:

I. ***Tuning Noise Schedules, Loss Weighting, and Sampling Trajectories:***
Diffusion models rely on adding and removing noise to data in a structured manner. Optimizing the **noise schedule** (how noise is added), **loss weighting** (how errors are penalized during training), and **sampling trajectories** (the path the model follows to generate data) can significantly enhance training efficiency and model performance. "Elucidating the Design Space of Diffusion-Based Generative Models" explores choices and showcase their impact on training dynamics.

II. ***Training on Diverse Aspect Ratios:***
Diffusion models have been trained on images with fixed aspect ratios, limiting their flexibility in handling real-world data. **Training on diverse aspect ratios** allows models to **better generalize** and produce high-quality images across various dimensions and formats. Know more

III. ***Cascaded Diffusion Models:***
Train an initial model to generate **low-resolution images** and then employing one or more **super-resolution models to enhance the image quality.** This technique, used in SOTA systems like DALLE-2 and Imagen, enables the generation of high-resolution images while maintaining computational efficiency. It's a key strategy for producing detailed and realistic images in high-resolution formats.

IV. *Enhanced Conditioning with Rich Text Embeddings*

Enhanced conditioning uses advanced text embeddings, like those from large language models (e.g., T5 used in Imagen), to provide richer contextual information. This leads to more coherent and contextually appropriate image generation, as the model has a deeper understanding of the text prompts it is conditioned on. (eDiffi)

V. *Knowledge Enhancement*

Incorporating external knowledge into the training process can greatly improve the quality of generated images. Techniques such as **using pre-trained image captioning and object detection models** allow the diffusion model to generate more informative captions and understand the content of images better.

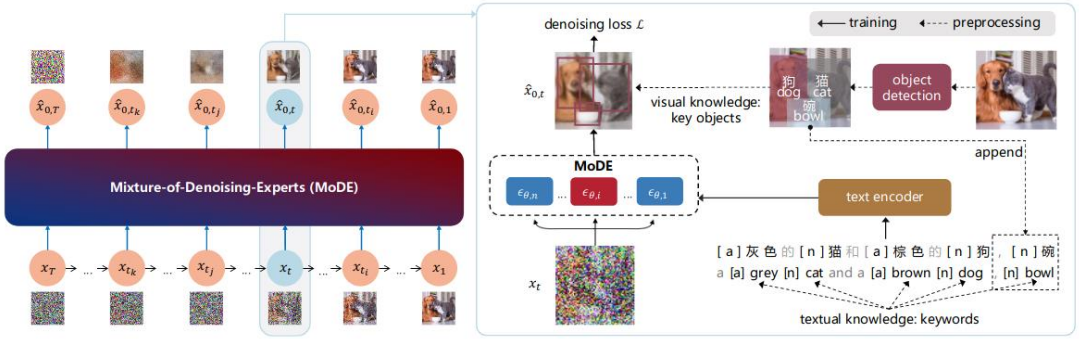


Figure 2: The illustration of ERNIE-ViLG 2.0 model architecture.

ERNIE-ViLG employs this 'knowledge enhancement' approach to improve its performance significantly, leveraging existing models to provide additional context during training.

VI. *Mixture of Denoising Experts (MoDE)*

MoDE involves **training multiple variants of a diffusion model**, each **specialized** in handling **different levels of noise** during the denoising process. This allows the model to better manage the noise at various stages of the generation process, leading to improved image quality and training efficiency. The ERNIE-ViLG 2.0 paper illustrates how this approach can be effectively implemented, showing enhanced results in image generation tasks.

Additional refs:

- *Elucidating the Design Space of Diffusion-Based Generative Models*
- *eDiffi: Text-to-Image Diffusion Models with an Ensemble of Expert Denoisers*
- *ERNIE-ViLG 2.0: Improving Text-to-Image Diffusion Model with Knowledge-Enhanced Mixture-of-Denoising-Experts*
- *Imagen - Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding (demo site)*
- ERNIE-ViLG 2.0 paper

6.3. More Control for Generation and Editing

The advancements in diffusion models also focus on enhancing sampling and inference phases leading to new capabilities in image generation and editing. These innovations can be categorized into four primary techniques.

The video **'Editing Images with Diffusion Models'** gives an overview of the different methods being used to edit existing images with diffusion models.

1) Noise Addition and Denoising with a New Prompt

This approach involves adding noise to an existing image and then denoising it using a different prompt, using img2img pipeline. By modifying the noise addition and denoising steps, models can achieve significant control over image editing:

- SDEdit and MagicMix are examples of methods that extend this basic idea.
- DDIM Inversion refines the process by using the model to reverse the sampling trajectory rather than adding random noise, providing more precise control over the output.
- Null-text Inversion further enhances the effectiveness of this method by optimizing the unconditional text embeddings used for classifier-free guidance at each denoising step, allowing for high-quality text-based image modifications.

2) Masked Editing with Noise and Denoising

This technique builds upon the first category by introducing a mask that controls where the noise addition and denoising effects are applied, allowing for localized image edits:

- Blended Diffusion introduces this concept, where a mask guides the diffusion process to edit specific image regions.
- CLIPSeg-based Demo utilizes an existing segmentation model to create masks based on textual descriptions, enabling more accurate targeting of edits.
- DiffEdit leverages the diffusion model itself to generate masks directly from text prompts, automating the process of identifying regions for modification.
- SmartBrush enhances mask-guided inpainting by fine-tuning the diffusion model, leading to more precise and contextually appropriate edits.

3) Cross-Attention Control

Cross-attention mechanisms in diffusion models allow for **fine-grained spatial control** during image editing. By manipulating the **attention layers**, these methods enable edits that are highly localized and textually guided:

- Prompt-to-Prompt Image Editing with Cross Attention Control is the foundational work that introduced this idea, demonstrating how attention layers can be adjusted to modify specific parts of an image in response to new prompts.
- Paint-with-Words (eDiffi) applies this technique to enable precise, word-level control over the generated image, allowing users to dictate edits in specific areas based on textual input.

4) Single Image Fine-tuning

This method involves fine-tuning a diffusion model on **a single image**, enabling the generation of new variations based on that image. This technique allows for detailed and consistent edits while preserving the original image's characteristics:

- Imagic : Text-Based Real Image Editing with Diffusion Models.
- UniTune : Text-Driven Image Editing by Fine Tuning an Image Generation Model on a Single Image.

A **notable innovation** that **combines several of the above techniques** is **InstructPix2Pix,** which uses a synthetic dataset of image pairs alongside natural language editing instructions (generated via GPT-3.5). This approach trains a model to perform image edits directly based on descriptive instructions, representing a significant step towards more intuitive and user-friendly image editing tools.

6.4. Challenging Data

The application of diffusion models to new types of data is raising. Video and audio generation and editing is an emerging area of research that builds on the foundational concepts used for images. However, video and audio have their own characteristics and therefore specific challenges.

6.4.1. Video

In video, which can be seen as a sequence of images, diffusion models are extended to handle the temporal dimension, requiring adaptations in both architecture and processing techniques.

Video diffusion models aim to **generate and manipulate video** content by applying the principles of diffusion models across the entire video sequence. The primary **challenge** lies in **handling the increased data complexity**, as **high-frame-rate videos** involve significantly more data than single images.

To address this, researchers have developed specialized architectures like **3D UNets**, which **process entire sequences** rather than individual frames. These models are designed to capture both **spatial and temporal dependencies** within the video, ensuring coherent and smooth transitions between frames during generation.

Since **high-frame-rate video** involves a lot more data than still images

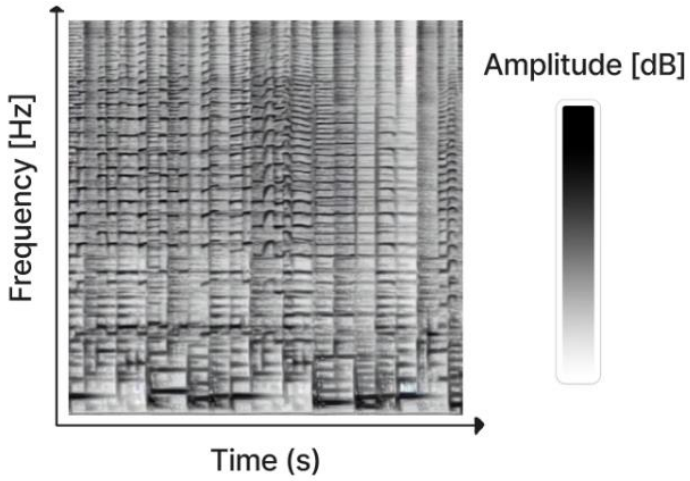
IMAGEN VIDEO: High-Definition Video Generation with Diffusion Models. The approach tackles the high data demands of video generation by employing a two-step process:

a) **Low-Resolution and Low-Frame-Rate Generation**: Initially, the model generates a low-resolution, low-frame-rate version of the video. This step simplifies the data processing and ensures the model can focus on capturing the essential structure and motion in the video sequence.
b) **Spatial and Temporal Super-Resolution**: Once the base video is generated, spatial and temporal super-resolution techniques are applied. Spatial super-resolution enhances the image quality of individual frames, while temporal super-resolution increases the frame rate, resulting in smooth, high-definition video output.

6.4.2. Audio

The audio generation using diffusion models is evolving, with significant progress being made in converting audio signals into a form that can be effectively processed by these models.

While there have been direct attempts to generate audio with diffusion models, such as **DiffWave**, the most successful approach so far has **involved converting audio into spectrograms**.



Spectrograms encode audio as 2D "images," enabling the use of well-established diffusion techniques originally developed for image generation. These spectrograms can then be converted back into audio using existing methods, allowing for high-quality audio synthesis.

Recent advances:

I. ***Direct Audio Generation: DiffWave***
By modeling the raw audio waveform, DiffWave **generates audio directly** without requiring intermediate representations like spectrograms.

II. ***Spectrogram-Based Audio Generation: Riffusion***
Riffusion leverages the idea of **converting audio into spectrograms**, treating them as 2D images that diffusion models can process. By fine-tuning the Stable Diffusion model to generate spectrograms conditioned on text, Riffusion can create diverse audio outputs based on user-provided prompts. The generated spectrograms are then converted back into audio, demonstrating the effectiveness of this approach for text-to-audio generation.

III. ***MusicLM***

Design by Google, MusicLM is a breakthrough in text-to-audio generation, capable of producing consistent and coherent **audio from textual descriptions**. This model can also be conditioned on hummed or whistled melodies, making it highly flexible for music generation.

IV. RAVE2 and AudioLDM

RAVE2 is a new version of a Variational Auto-Encoder that will be useful for latent diffusion on audio tasks. The upcoming AudioLDM, builds up on the previous and will employ latent diffusion models to enhance audio synthesis.

V. Noise2Music

Noise2Music is a diffusion model specifically trained to generate high-quality 30-second audio clips from text descriptions. By focusing on creating longer and more coherent audio sequences, Noise2Music addresses the challenges of producing extended audio content with diffusion models.

VI. Make-An-Audio

Make-An-Audio introduces a prompt-enhanced diffusion model for text-to-audio generation. This model is designed to generate a wide variety of sounds based on text prompts, making it useful for applications that require diverse audio outputs, such as sound effects and environmental audio synthesis.

Additional refs:

- *Video Diffusion Models*
- *IMAGEN VIDEO: HIGH DEFINITION VIDEO GENERATION WITH DIFFUSION MODELS*
- *DiffWave: A Versatile Diffusion Model for Audio Synthesis*
- *'Riffusion' (and code)*
- *MusicLM by Google generates consistent audio from text and can be conditioned with hummed or whistled melodies.*
- *RAVE2, AudioLDM model.*
- *Noise2Music*
- *Make-An-Audio: Text-To-Audio Generation with Prompt-Enhanced Diffusion Models -*
- *Moûsai: Text-to-Music Generation with Long-Context Latent Diffusion*

6.5. Towards 'Iterative Refinement'

Initially, diffusion models were defined by the process of gradually adding noise (often Gaussian) to data and then training a model to reverse this process. However, recent research is expanding this concept into a broader class of models that focus **on iterative refinement**—where **various forms of corruption**, not just Gaussian noise, are incrementally undone to produce images or other data types.

6.5.1. Cold Diffusion: Expanding the Concept of Corruption

The Cold Diffusion paper illustrated that the idea of iterative refinement can be applied to many **types of corruption** beyond Gaussian noise. It was showed that different corruption methods, such

as **blurring or down sampling,** can be iteratively reversed to generate high-quality images. This broadens the scope of what we consider a "diffusion model," suggesting that the key element is the process of **progressive refinement** rather than the specific type of noise added during the forward process.

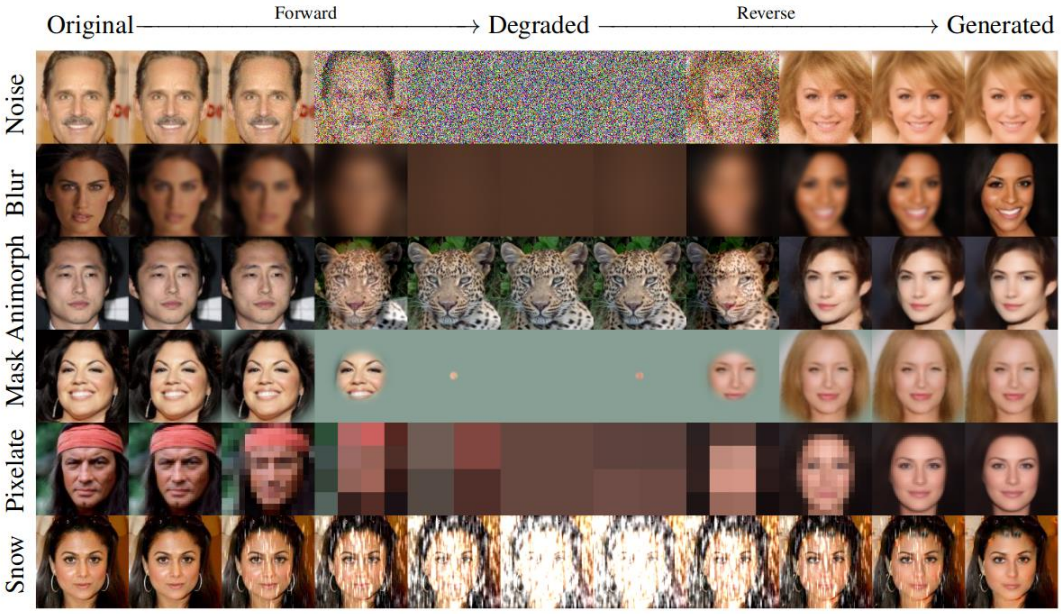
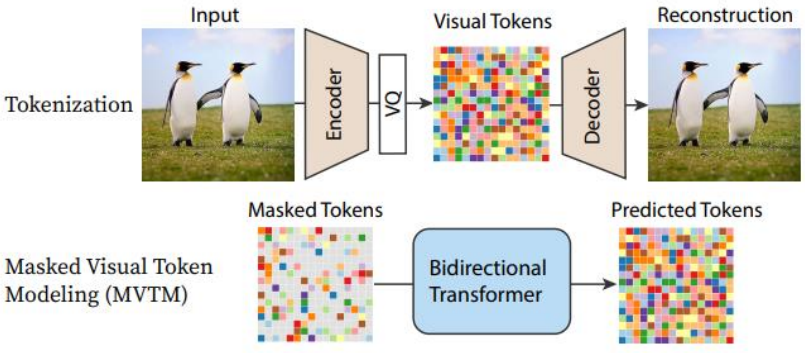


Figure 1: Demonstration of the forward and backward processes for both hot and cold diffusions. While standard diffusions are built on Gaussian noise (top row), we show that generative models can be built on arbitrary and even noiseless/cold image transforms, including the ImageNet-C *snowification* operator, and an *animorphosis* operator that adds a random animal image from AFHQ.

6.5.2. Transformer-Based Approaches: A Shift from UNet

The traditional UNet architecture, which has been central to diffusion models, is increasingly being replaced by **transformer-based architectures** that bring new capabilities and efficiencies. **DiT (Scalable Diffusion Models with Transformers)** replaces UNet with a transformer within a standard diffusion framework, demonstrating that transformers can effectively handle iterative refinement with impressive results.



Recurrent Interface Networks introduces a novel transformer-based architecture aimed at improving efficiency by refining data processing iteratively, thereby reducing computational costs while maintaining high output quality.

Token-Based Architectures, such as **MaskGIT** and **MUSE**, leverage transformers to work with **tokenized image representations**, treating images as sequences of tokens. This allows these models

to iteratively generate or refine specific image parts, like how language models predict word sequences.

NOTE: Despite the growing popularity of transformers, the Paella model illustrates that UNet architectures remain effective even within token-based regimes.

Additional refs:

- *Video Diffusion Models*
- *IMAGEN VIDEO: HIGH DEFINITION VIDEO GENERATION WITH DIFFUSION MODELS*
- *DiffWave: A Versatile Diffusion Model for Audio Synthesis*
- *'Riffusion' (and code)*
- *Cold Diffusion: Inverting Arbitrary Image Transforms Without Noise*
- *Scalable Diffusion Models with Transformers (DiT)*
- *MaskGIT: Masked Generative Image Transformer*
- *Muse: Text-To-Image Generation via Masked Generative Transformers*
- *Fast Text-Conditional Discrete Denoising on Vector-Quantized Latent Spaces (Paella)*
- *Recurrent Interface Networks*
- *simple diffusion: End-to-end diffusion for high-resolution images*