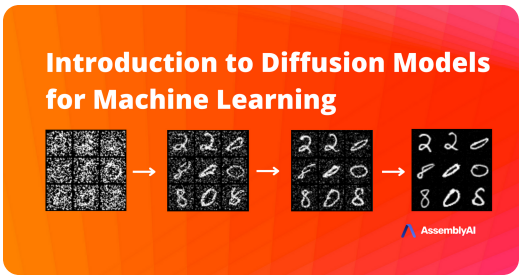
DEEP LEARNING

Introduction to Diffusion Models for Machine Learning



The meteoric rise of Diffusion Models is one of the biggest developments in Machine Learning in the past several years. Learn everything you need to know about Diffusion Models in this easy-to-follow guide.

Ryan O'Connor
Developer Educator at AssemblyAI

May 12, 2022

Diffusion Models are generative models which have been gaining significant popularity in the past several years, and for good reason. A handful of seminal ers released in the 2020s *alone* have

on image synthesis. Most recently, practitioners will have seen Diffusion Models used in DALL-E 2, OpenAI's image generation model released last month.

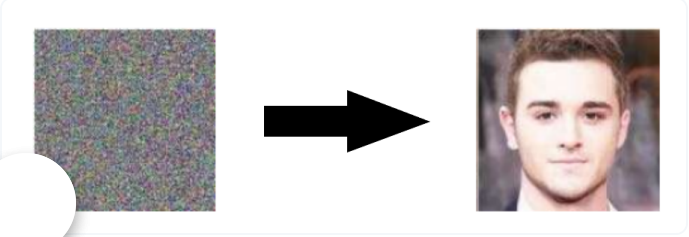
Various images generated by DALL-E 2 (source).

Given the recent wave of success by Diffusion Models, many Machine Learning practitioners are surely interested in their inner workings. In this article, we will examine the **theoretical foundations for Diffusion Models**, and demonstrate how to generate

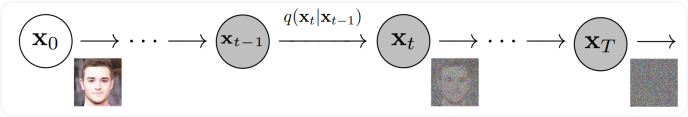
intuitive explanation of Diffusion Models, feel free to check out our article on how physics advanced Generative AI. Let's dive in!

Diffusion Models - Introduction

Diffusion Models are **generative** models, meaning that they are used to generate data similar to the data on which they are trained. Fundamentally, Diffusion Models work by **destroying training data** through the successive addition of Gaussian noise, and then **learning to recover** the data by *reversing* this noising process. After training, we can use the Diffusion Model to generate data by simply **passing randomly sampled noise through the learned denoising process**.



More specifically, a Diffusion Model is a latent variable model which maps to the latent space using a fixed Markov chain. This chain gradually adds noise to the data in order to obtain the approximate posterior $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$, where $\mathbf{x}_1, \ldots, \mathbf{x}_T$ are the latent variables with the same dimensionality as $\mathbf{x}_0$. In the figure below, we see such a Markov chain manifested for image data.



(Modified from source)

Ultimately, the image is asymptotically transformed to pure Gaussian noise. The **goal** of training a diffusion model is to learn the **reverse** process - i.e. training $p\theta(x_{t-1}|x_t)$. By traversing backwards along this chain, we can generate new

.

(Modified from source)

Benefits of Diffusion Models

As mentioned above, research into Diffusion Models has exploded in recent years. Inspired by non-equilibrium thermodynamics[1], Diffusion Models currently produce **State-of-the-Art image quality**, examples of which can be seen below:

(adapted from source)

other benefits, including **not requiring adversarial training**. The difficulties of adversarial training are well-documented; and, in cases where non-adversarial alternatives exist with comparable performance and training efficiency, it is usually best to utilize them. On the topic of training efficiency, Diffusion Models also have the added benefits of **scalability and parallelizability.**

While Diffusion Models almost seem to be producing results out of thin air, there are a lot of careful and interesting mathematical choices and details that provide the foundation for these results, and best practices are still evolving in the literature. Let's take a look at the mathematical theory underpinning Diffusion Models in more detail now.

Diffusion Models - A Deep Dive

diffusion process), in which a datum (generally an image) is progressively noised, and a **reverse process** (or **reverse diffusion process**), in which noise is transformed back into a sample from the target distribution.

The sampling chain transitions in the forward process can be set to conditional Gaussians when the noise level is sufficiently low. Combining this fact with the Markov assumption leads to a simple parameterization of the forward process:

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

Mathematical Note

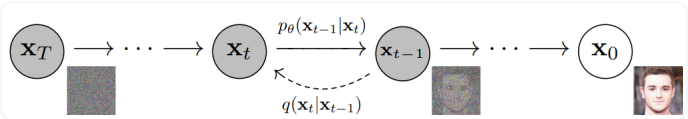
Where $\beta_1, \ldots, \beta_T$ is a variance schedule (either learned or fixed) which, if well-behaved, **ensures that x_T is nearly an isotropic Gaussian for sufficiently large**

Given the Markov assumption, the joint distribution of the latent variables is the product of the Gaussian conditional chain transitions (modified from source).

As mentioned previously, the "magic" of diffusion models comes in the **reverse process**. During training, the model learns to reverse this diffusion process in order to generate new data. Starting with the pure Gaussian noise $p(\mathbf{x}_T) := \mathcal{N}(\mathbf{x}_T, \mathbf{0}, \mathbf{I})$, the model learns the joint distribution $p\theta(\mathbf{x}_{0:T})$ as

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) := p(\mathbf{x}_T) \prod_{t=1}^{T} \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

where the time-dependent parameters of the Gaussian transitions are learned. Note in particular that the Markov formulation asserts that a given reverse diffusion transition distribution depends only on the previous timestep (or following timestep, depending on how look at it):



(Modified from source)

Want to learn how to build a Diffusion Model in PyTorch?

Check out our MinImagen project, where we go through building a minimal implementation of the text-to-image model Imagen!

Check it out

Training

A Diffusion Model is trained by **finding the reverse Markov transitions that maximize the likelihood of the training data**. In practice, training equivalently consists of minimizing the variational

$$\mathbb{E}[-\log p_\theta(\mathbf{x}_0)] \leq \mathbb{E}_q \left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] =: L_{vlb}$$

Notation Detail

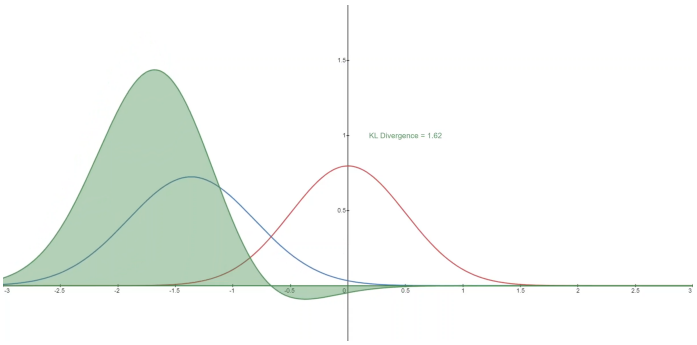
We seek to rewrite the L_{vlb} in terms of **Kullback-Leibler (KL) Divergences**. The KL Divergence is an asymmetric statistical distance measure of how much one probability distribution *P* differs from a reference distribution *Q*. We are interested in formulating L_{vlb} in terms of KL divergences because the transition distributions in our Markov chain are Gaussians, and **the KL divergence between Gaussians has a closed form**.

What is the KL Divergence?

The mathematical form of the KL divergence for continuous distributions is

The double bars indicate that the function is *not* symmetric with respect to its arguments.

Below you can see the KL divergence of a varying distribution P (blue) from a reference distribution Q (red). The green curve indicates the function within the integral in the definition for the KL divergence above, and the total area under the curve represents the value of the KL divergence of P from Q at any given moment, a value which is also displayed numerically.



Casting L_{vlb} in Terms of KL Divergences

As mentioned previously, it is possible[1] to rewrite L_{vlb} almost completely in

$$L_{vlb} = L_0 + L_1 + ... + L_{T-1} + L_T$$

where

$$L_0 = -\log p_\theta(x_0|x_1)$$
$$L_{t-1} = D_{KL}(q(x_{t-1}|x_t, x_0) \,||\, p_\theta(x_{t-1}|x_t))$$
$$L_T = D_{KL}(q(x_T|x_0) \,||\, p(x_T))$$

Derivation Details

Conditioning the forward process posterior on x_0 in L_{t-1} results in a tractable form that leads to **all KL divergences being comparisons between Gaussians**. This means that the divergences can be exactly calculated with closed-form expressions rather than with Monte Carlo estimates[3].

Model Choices

With the mathematical foundation for our objective function established, we now need to make several choices regarding

the only choice required is defining the variance schedule, the values of which are generally increasing during the forward process.

For the reverse process, we much choose the Gaussian distribution parameterization / model architecture(s). Note the **high degree of flexibility** that Diffusion Models afford - the *only* requirement on our architecture is that its input and output have the same dimensionality.

We will explore the details of these choices in more detail below.

Forward Process and L_T

As noted above, regarding the forward process, we must define the variance schedule. In particular, we set them to be **time-dependent constants**, ignoring the fact that they can be learned. For example[3], a linear schedule from

perhaps a geometric series.

Regardless of the particular values chosen, the fact that the variance schedule is fixed results in L_T becoming a constant with respect to our set of learnable parameters, allowing us to ignore it as far as training is concerned.

$$L_T = D_{KL}(q(x_T|x_0) \| p(x_T))$$

Reverse Process and $L_{1:T-1}$

Now we discuss the choices required in defining the reverse process. Recall from above we defined the reverse Markov transitions as a Gaussian:

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

We must now define the functional forms of $\boldsymbol{\mu}_\theta$ or $\boldsymbol{\Sigma}_\theta$. While there are more

we simply set

$$\Sigma_\theta(x_t, t) = \sigma_t^2 \mathbb{I}$$
$$\sigma_t^2 = \beta_t$$

That is, we assume that the multivariate Gaussian is a product of independent gaussians with identical variance, a variance value which can change with time. We **set these variances to be equivalent to our forward process variance schedule**.

Given this new formulation of Σ_θ, we have

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \coloneqq \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t) \coloneqq \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \sigma_t^2 \mathbf{I})$$

which allows us to transform

$$L_{t-1} = D_{KL}(q(x_{t-1}|x_t, x_0) \,||\, p_\theta(x_{t-1}|x_t))$$

to

where the first term in the difference is a linear combination of x_t and x_0 that depends on the variance schedule β_t.

The exact form of this function is not relevant for our purposes, but it can be found in [3].

The significance of the above proportion is that **the most straightforward parameterization of μ_θ simply predicts the diffusion posterior mean**. Importantly, the authors of [3] actually found that training μ_θ to predict the *noise* component at any given timestep yields better results. In particular, let

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

where

$$:= 1 - \beta_t \quad \text{and} \quad \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$$

loss function, which the authors of [3] found to lead to more stable training and better results:

$$L_{\text{simple}}(\theta) := \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2\right]$$

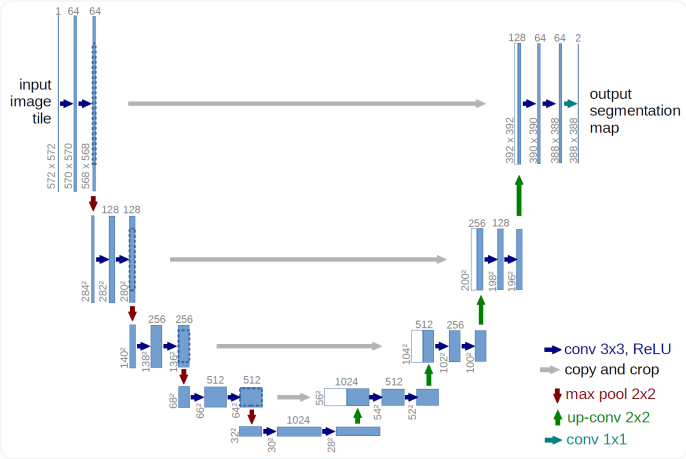
The authors of [3] also note connections of this formulation of Diffusion Models to score-matching generative models based on Langevin dynamics. Indeed, it appears that Diffusion Models and Score-Based models may be two sides of the same coin, akin to the independent and concurrent development of wave-based quantum mechanics and matrix-based quantum mechanics revealing two equivalent formulations of the same phenomena[2].

Network Architecture

While our simplified loss function seeks to train a model ϵ_θ, we have still not yet defined the architecture of this model. [illegible] that the *only* requirement for the

Table of contents

Given this restriction, it is perhaps unsurprising that image Diffusion Models are commonly implemented with U-Net-like architectures.



Architecture of U-Net (source)

Reverse Process Decoder and L_0

The path along the reverse process consists of many transformations under continuous conditional Gaussian distributions. At the end of the reverse process, recall that we are trying to produce an **image**, which is composed of integer pixel values. Therefore, we must ose a way to obtain **discrete (log)**

The way that this is done is by setting the last transition in the reverse diffusion chain to an **independent discrete decoder**. To determine the likelihood of a given image x_0 given x_1, we first impose independence between the data dimensions:

$$p_\theta(x_0|x_1) = \prod_{i=1}^{D} p_\theta(x_0^i|x_1^i)$$

where D is the dimensionality of the data and the superscript *i* indicates the extraction of one coordinate. The goal now is to determine how likely each integer value is for a given pixel *given* the distribution across possible values for the corresponding pixel in the slightly noised image at time $t = 1$:

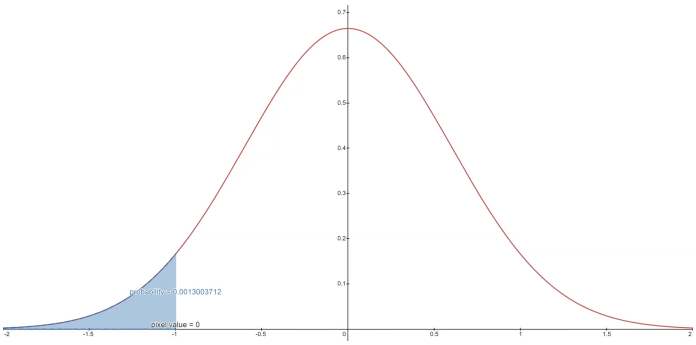
$$\mathcal{N}(x; \mu_\theta^i(x_1, 1), \sigma_1^2)$$

derived from the below multivariate Gaussian whose diagonal covariance matrix allows us to split the distribution into a product of univariate Gaussians, one for each dimension of the data:

$$\mathcal{N}(x; \mu_\theta(x_1, 1), \sigma_1^2 \mathbb{I}) = \prod_{i=1}^{D} \mathcal{N}(x; \mu_\theta^i(x_1, 1), \sigma_1^2)$$

We assume that the images consist of integers in $0, 1, \ldots, 255$ (as standard RGB images do) which have been scaled linearly to $[-1, 1]$. We then break down the real line into small "buckets", where, for a given scaled pixel value *x*, the bucket for that range is $[x - 1/255, x + 1/255]$. The probability of a pixel value *x,* given the univariate Gaussian distribution of the corresponding pixel in x_1, is the **area under that univariate Gaussian distribution within the bucket centered at *x***.

a mean-0 Gaussian which, in this context, corresponds to a distribution with an average pixel value of $255/2$ (half brightness). The red curve represents the distribution of a specific pixel in the *t=1* image, and the areas give the probability of the corresponding pixel value in the *t=0* image.



Technical Note

Given a *t=0* pixel value for each pixel, the value of $p\theta(x_0|x_1)$ is simply their product.

This process is succinctly encapsulated by the following equation:

where

$$\delta_{-}(x) = \begin{cases} -\infty & x = -1 \\ x - \frac{1}{255} & x > -1 \end{cases}$$

and

$$\delta_{+}(x) = \begin{cases} \infty & x = 1 \\ x + \frac{1}{255} & x < 1 \end{cases}$$

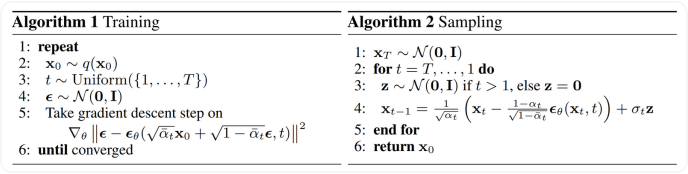
Given this equation for $p\theta(x_0|x_1)$, we can calculate the final term of L_{vlb} which is not formulated as a KL Divergence:

$$L_0 = -\log p_\theta(x_0|x_1)$$

Final Objective

As mentioned in the last section, the authors of [3] found that predicting the noise component of an image at a given timestep produced the best results. Ultimately, they use the following objective:

The training and sampling algorithms for our Diffusion Model therefore can be succinctly captured in the below figure:



Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

(source)

Diffusion Model Theory Summary

In this section we took a detailed dive into the theory of Diffusion Models. It can be easy to get caught up in mathematical details, so we note the most important points within this section below in order to keep ourselves oriented from a birds-eye perspective:

1. Our Diffusion Model is parameterized as a **Markov chain**, meaning that our latent variables $x1, \dots, xT$ depend only on the previous (or following) timestep.

the forward process requires a variance schedule, and the reverse process parameters are learned.

3. The diffusion process ensures that x_T is **asymptotically distributed as an isotropic Gaussian** for sufficiently large T.

4. In our case, the **variance schedule was fixed**, but it can be learned as well. For fixed schedules, following a geometric progression may afford better results than a linear progression. In either case, the variances are generally increasing with time in the series (i.e. $\beta_i < \beta_j$ for $i < j$).

5. Diffusion Models are **highly flexible** and allow for *any* architecture whose input and output dimensionality are the same to be used. Many implementations use **U-Net-like** architectures.

This is manifested as tuning the model parameters to **minimize the variational upper bound of the negative log likelihood of the data**.

7. Almost all terms in the objective function can be cast as **KL Divergences** as a result of our Markov assumption. These values **become tenable to calculate** given that we are using Gaussians, therefore omitting the need to perform Monte Carlo approximation.

8. Ultimately, using a **simplified training objective** to train a function which predicts the noise component of a given latent variable yields the best and most stable results.

9. A **discrete decoder** is used to obtain log likelihoods across pixel values as the last step in the reverse diffusion process.

With this high-level overview of Diffusion ...els in our minds, let's move on to see

Diffusion Models in PyTorch

While Diffusion Models have not yet been democratized to the same degree as other older architectures/approaches in Machine Learning, there are still implementations available for use. The easiest way to use a Diffusion Model in PyTorch is to use the `denoising-diffusion-pytorch` package, which implements an image diffusion model like the one discussed in this article. To install the package, simply type the following command in the terminal:

```
pip install denoising_diffusion_pytorch
```

Minimal Example

To train a model and generate images, we first import the necessary packages:

```
import torch
from denoising_diffusion_pytorch import Unet, GaussianDiffusion
```

specifies the number of feature maps before the first down-sampling, and the `dim_mults` parameter provides multiplicands for this value and successive down-samplings:

```
model = Unet(
    dim = 64,
    dim_mults = (1, 2, 4, 8)
)
```

Now that our network architecture is defined, we need to define the Diffusion Model itself. We pass in the U-Net model that we just defined along with several parameters - the size of images to generate, the number of timesteps in the diffusion process, and a choice between the L1 and L2 norms.

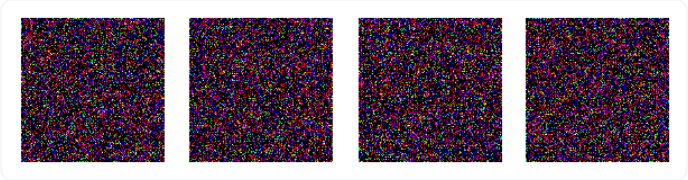
```
diffusion = GaussianDiffusion(
    model,
    image_size = 128,
    timesteps = 1000,   # number of s
teps
    loss_type = 'l1'    # L1 or L2
)
```

Now that the Diffusion Model is defined, ...me to train. We generate random

```
training_images = torch.randn(8, 3, 1
28, 128)
loss = diffusion(training_images)
loss.backward()
```

Once the model is trained, we can finally generate images by using the `sample()` method of the `diffusion` object. Here we generate 4 images, which are only noise given that our training data was random:

```
sampled_images = diffusion.sample(bai
ch_size = 4)
```



Training on Custom Data

The `denoising-diffusion-pytorch` package also allow you to train a diffusion model on a specific dataset. Simply replace the `'path/to/your/ima` string with the dataset directory

appropriate value. After that, simply run the code to train the model, and then sample as before. Note that PyTorch must be compiled with CUDA enabled in order to use the `Trainer` class:

```
from denoising_diffusion_pytorch impo
rt Unet, GaussianDiffusion, Trainer

model = Unet(
    dim = 64,
    dim_mults = (1, 2, 4, 8)
).cuda()

diffusion = GaussianDiffusion(
    model,
    image_size = 128,
    timesteps = 1000,   # number of s
teps
    loss_type = 'l1'    # L1 or L2
).cuda()

trainer = Trainer(
    diffusion,
    'path/to/your/images',
    train_batch_size = 32,
    train_lr = 2e-5,
    train_num_steps = 700000,
 # total training steps
    gradient_accumulate_every = 2,
 # gradient accumulation steps
    ema_decay = 0.995,
 # exponential moving average decay
    amp = True
 # turn on mixed precision
)

trainer.train()
```

MNIST digits akin to reverse diffusion:

Final Words

Diffusion Models are a conceptually simple and elegant approach to the problem of generating data. Their State-of-the-Art results combined with non-adversarial training has propelled them to great heights, and further improvements can be expected in the coming years given their nascent status. In particular, Diffusion Models have been found to be essential to the performance of cutting-edge models like DALL-E 2.

Did you enjoy this article?

you don't miss content like this in the future.

Follow

References

[1] Deep Unsupervised Learning using Nonequilibrium Thermodynamics

[2] Generative Modeling by Estimating Gradients of the Data Distribution

[3] Denoising Diffusion Probabilistic Models

[4] Improved Techniques for Training Score-Based Generative Models

[5] Improved Denoising Diffusion Probabilistic Models

[6] Diffusion Models Beat GANs on Image Synthesis

[7] GLIDE: Towards Photorealistic Image eration and Editing with Text-Guided

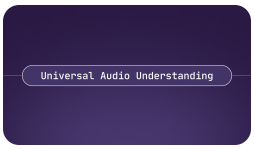
[8] Hierarchical Text-Conditional Image Generation with CLIP Latents

Popular posts

Feb 20, 2024

AI trends in 2024: Graph Neural Networks

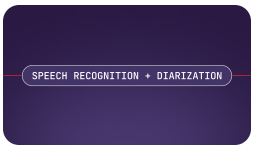
Marco Ramponi
Developer Educator at AssemblyAI



Dec 7, 2023

AI for Universal Audio Understan Qwen-Audio Explained

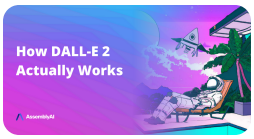
Marco Ramponi
Developer Educator at AssemblyAI



Oct 27, 2023

Combining Speech Recognitic and Diarization in one model

Marco Ramponi
Developer Educator at AssemblyAI



Sep 29, 2023

How DALL-E 2 Actually Works

Ryan O'Connor
Developer Educator at AssemblyAI