



An In-Depth Guide to Denoising Diffusion Probabilistic Models – From Theory to Implementation

 Vaibhav Singh
MARCH 6, 2023 — [LEAVE A COMMENT](#)

AI Art Generation Deep Learning Diffusion Models Generative Models

Diffusion probabilistic models are an exciting new area of research showing great promise in image generation. In retrospect, diffusion-based generative models were first introduced in 2015 and popularized in 2020 when Ho et al. published the paper “Denoising Diffusion Probabilistic Models” (DDPMs). DDPMs are responsible for making diffusion models practical. In this article, we will highlight the key concepts and techniques behind DDPMs and train DDPMs from scratch on a “flowers” dataset for unconditional image generation.

Unconditional Image Generation

In DDPMs, the authors changed the formulation and model training procedures which helped to improve and achieve “*image fidelity*” rivaling GANs and established the validity of these new generative algorithms.

The best approach to completely understanding “*Denoising Diffusion Probabilistic Models*” is by going over both theory (+ some math) and the underlying code. With that in mind, let’s explore the learning path where:

- We’ll first explain what generative models are and why they are needed.
- We’ll discuss, from a theoretical standpoint, the approach used in diffusion-based generative models
- We’ll explore all the math necessary to understand denoising diffusion probabilistic models.
- Finally, we’ll discuss the training and inference used in DDPMs for image generation and code it from scratch in PyTorch.

Table of Contents

1. [The Need For Generative Models](#)
2. [What Are Diffusion Probabilistic Models?](#)
 - 2.1. [Forward Diffusion Process](#)
 - 2.2. [Reverse Diffusion Process](#)

3. [Itsy-Bitsy Mathematical Details Behind Denoising Diffusion Probabilistic Models](#)
 - 3.1. [Mathematical Details Of The Forward Diffusion Process](#)
 - 3.2. [Mathematical Details Of The Reverse Diffusion Process](#)
 - 3.3. [Training Objective & Loss Function Used In Denoising Diffusion Probabilistic Models](#)
4. [Writing DDPMs From Scratch In PyTorch](#)
5. [Creating PyTorch Dataset Class Object](#)
6. [Creating PyTorch Dataloader Class Object](#)
7. [Visualizing Dataset](#)
8. [Model Architecture Used In DDPMs](#)
9. [Diffusion Class](#)
10. [Python Code For Forward Diffusion Process](#)
11. [Training & Sampling Algorithms Used In Denoising Diffusion Probabilistic Models](#)
12. [Training DDPMs From Scratch](#)
13. [Generating images using DDPMs](#)
14. [Summary](#)

The Need For Generative Models

The job of image-based [generative models](#) is to generate new images that are similar, in other words, “representative” of our original set of images.

We need to create and train generative models because the set of all possible images that can be represented by, say, just (256x256x3) images is enormous. An image must have the right pixel value combinations to represent something meaningful (something we can understand).



An RGB image of a Sunflower

For example, for the above image to represent a “Sunflower”, the pixels in the image need to be in the right configuration

(they need to have the right values). And the space where such images exist is just a fraction of the entire set of images that can be represented by a (256x256x3) image space.

Now, if we knew how to get/sample a point from this subspace, we wouldn't need to build “*generative models*.” However, at this point in time, we don't. 🤔🤔

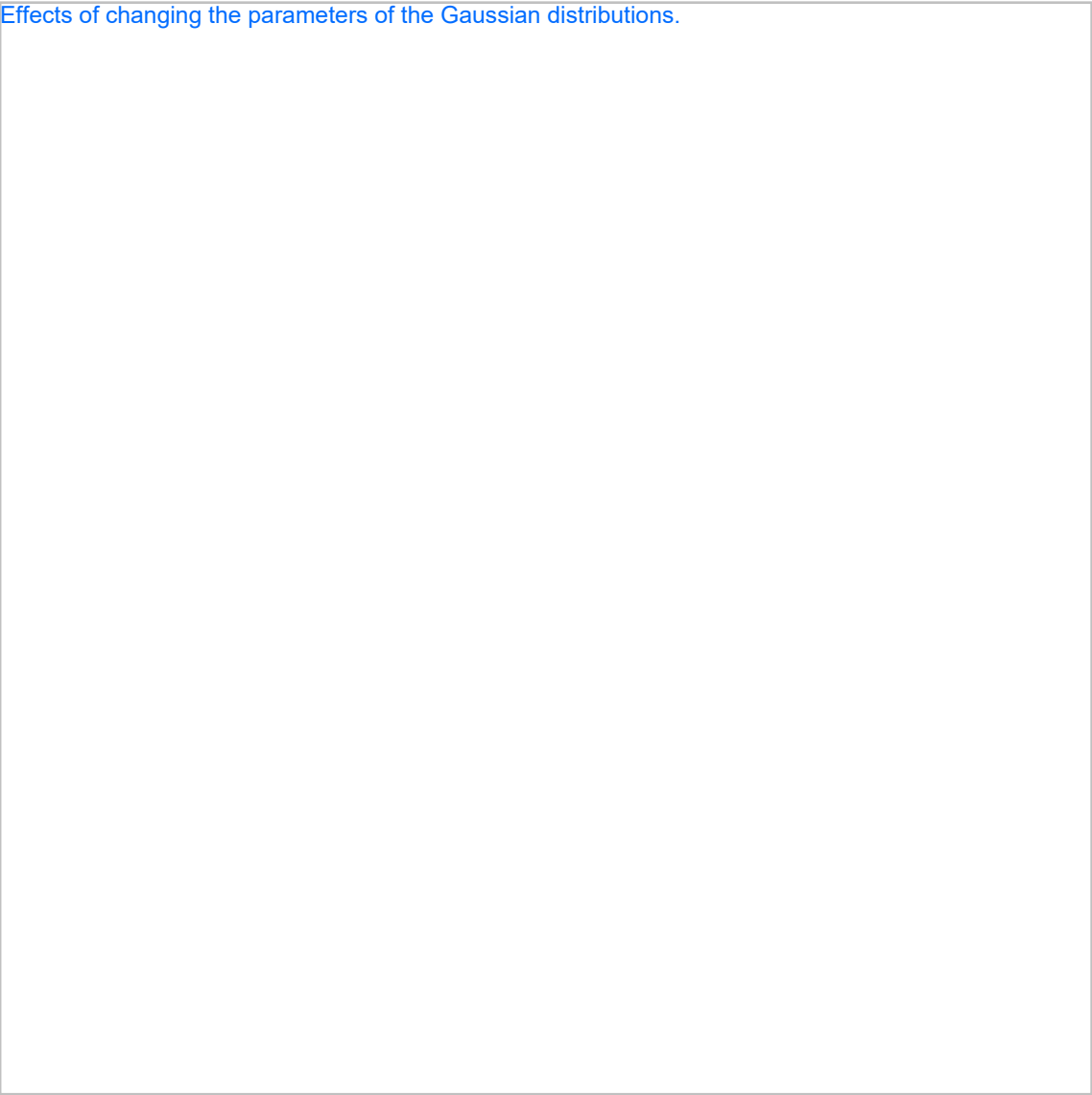
The probability distribution function or, more precisely, probability density function (PDF) that captures/models this (data) subspace remains unknown and most likely too complex to make sense.

This is why we need ‘Generative models — *To figure out the underlying likelihood function our data satisfies.*

PS: A PDF is a “probability function” representing the density (likelihood) of a continuous random variable – which, in this case, means a function representing the likelihood of an image lying between a specific range of values defined by the function's parameters.

PPS: Every PDF has a set of parameters that determine the shape and probabilities of the distribution. The shape of the distribution changes as the parameter values change. For example, in the case of a normal distribution, we have mean μ (mu) and variance σ^2 (sigma) that control the distribution's center point and spread.

Effects of changing the parameters of the Gaussian distributions.



Effect of parameters of the Gaussian Distribution

Source: <https://magic-with-latents.github.io/latent/posts/ddpms/part2/>

What Are Diffusion Probabilistic Models?

In our previous post, “[Introduction to Diffusion Models for Image Generation](#)”, we didn’t discuss the math behind these models. We provided only a conceptual overview of how diffusion models work and focused on different well-known models and their applications. In this article, we’ll be focusing heavily on the first part.

In this section, we’ll explain diffusion-based generative models from a logical and theoretical perspective. Next, we’ll review all the math required to understand and implement *Denoising Diffusion Probabilistic Models* from scratch.

Diffusion models are a class of generative models inspired by an idea in Non-Equilibrium Statistical Physics, which states:

“We can gradually convert one distribution into another using a Markov chain”

– Deep Unsupervised Learning using Nonequilibrium Thermodynamics, 2015

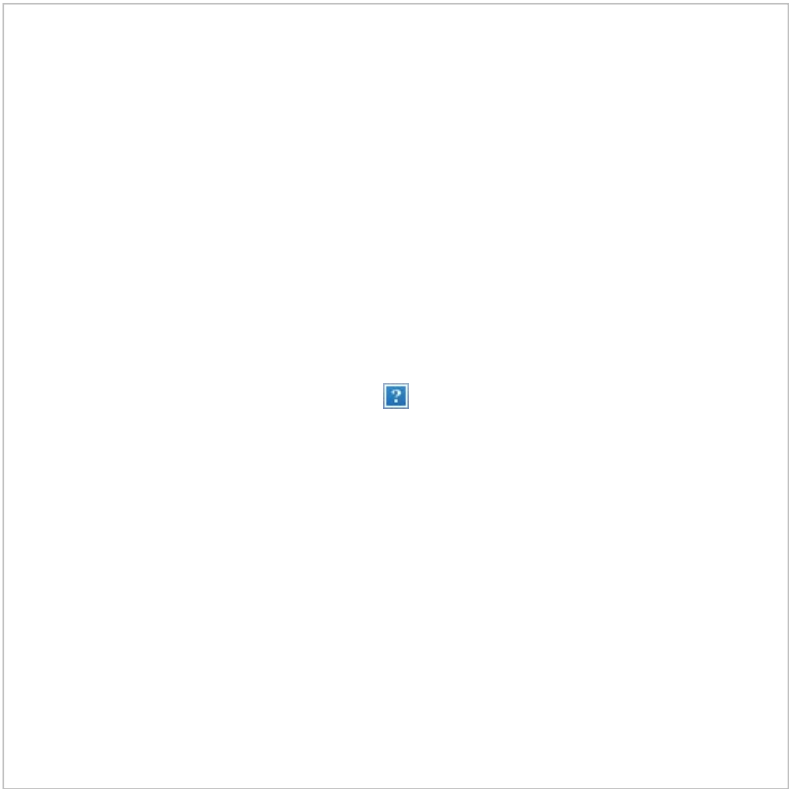
Diffusion generative models are composed of two opposite processes i.e., *Forward & Reverse Diffusion Process*.

Forward Diffusion Process:

“It’s easy to destroy but hard to create”

– Pearl S. Buck

- 1. In the “Forward Diffusion” process, we slowly and iteratively add noise to (corrupt) the images in our training set such that they “move out or move away” from their existing subspace.
- 2. What we are doing here is converting the unknown and complex distribution that our training set belongs to into one that is easy for us to sample a (data) point from and understand.
- 3. At the end of the forward process, the [images become entirely unrecognizable](#). The complex data distribution is wholly transformed into a (chosen) simple distribution. Each image gets mapped to a space outside the data subspace.

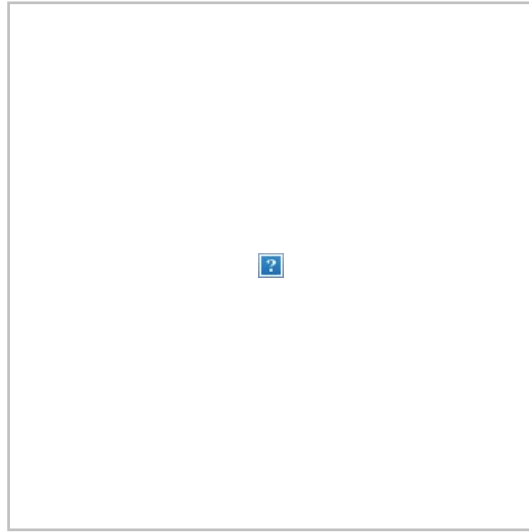


Source: <https://ayandas.me/blog-tut/2021/12/04/diffusion-prob-models.html>

Reverse Diffusion Process:

By decomposing the image formation process into a sequential application of denoising autoencoders, diffusion models (DMs) achieve state-of-the-art synthesis results on image data and beyond.

Stable Diffusion, 2022



A high-level conceptual overview of the entire image space.

1. In the “Reverse Diffusion process,” the idea is to reverse the forward diffusion process.
2. We slowly and iteratively try to reverse the corruption performed on images in the forward process.
3. The reverse process starts where the forward process ends.
4. The benefit of starting from a simple space is that we know how to get/sample a point from this simple distribution (think of it as any point outside the data subspace).
5. And our goal here is to figure out how to return to the data subspace.
6. However, the problem is that we can take infinite paths starting from a point in this “simple” space, but only a fraction of them will take us to the “data” subspace.
7. In diffusion probabilistic models, this is done by referring to the small iterative steps taken during the forward diffusion process.
8. The PDF that satisfies the corrupted images in the forward process differs slightly at each step.
9. Hence, in the reverse process, we use a deep-learning model at each step to predict the PDF parameters of the forward process.
10. And once we train the model, we can start from any point in the simple space and use the model to iteratively take steps to lead us back to the data subspace.
11. In reverse diffusion, we iteratively perform the “*denoising*” in small steps, starting from a noisy image.
12. This approach for training and generating new samples is much more stable than GANs and better than previous approaches like variational autoencoders (VAE) and normalizing flows.

A gif illustrating the inference stage of diffusion probabilistic models.



Since their introduction in 2020, DDPMs has been the foundation for cutting-edge image generation systems, including DALL-E 2, Imagen, Stable Diffusion, and Midjourney.

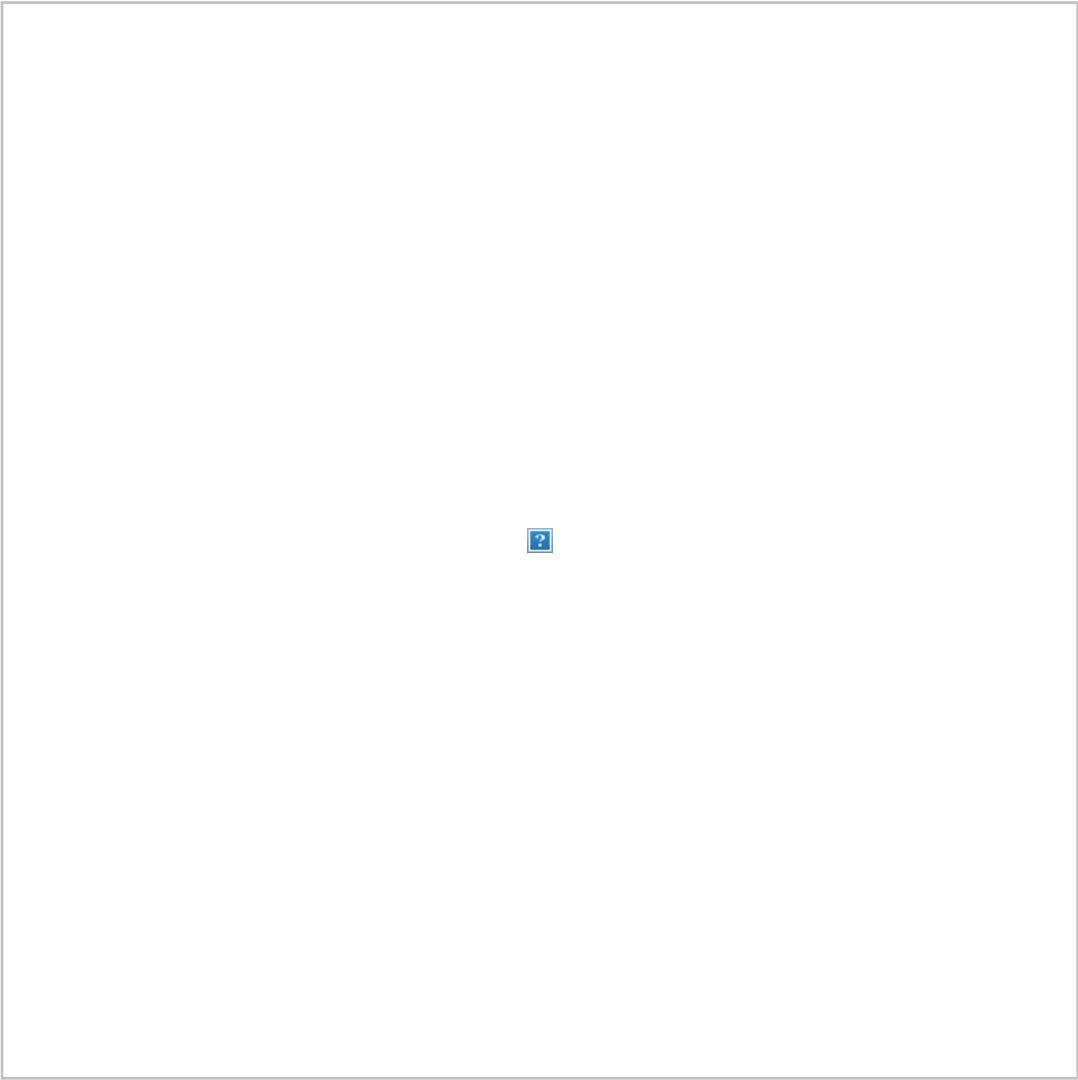
With the huge number of AI art generation tools today, it is difficult to find the right one for a particular use case. In our recent article, we explored all the different [AI art generation tools](#) so that you can make an informed choice to generate the best art.

Itsy-Bitsy Mathematical Details Behind Denoising Diffusion Probabilistic Models

As the motive behind this post is “*creating and training Denoising Diffusion Probabilistic models from scratch*,” we may have to introduce not all but some of the mathematical magic behind them.

In this section, we’ll cover all the required math while making sure it’s also easy to follow.

Let's begin...



There are two terms mentioned on the arrows:



- 1. This term is also known as the *forward diffusion kernel (FDK)*.
- 2. It defines the PDF of an image at timestep t in the forward diffusion process x_t given image x_{t-1} .
- 3. It denotes the “transition function” applied at each step in the *forward diffusion process*.

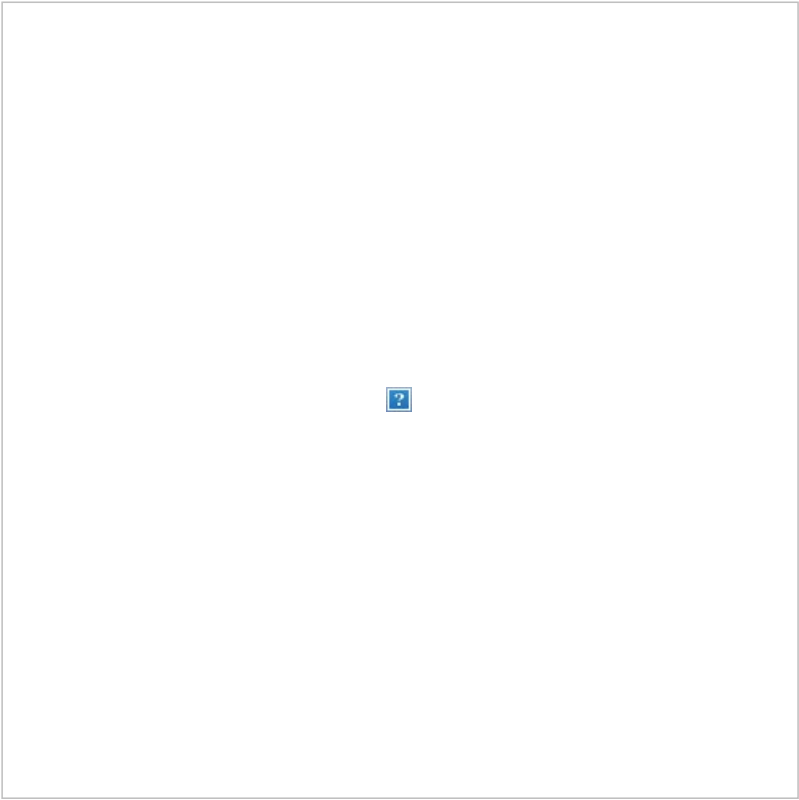


- 1. Similar to the forward process, it is known as the *reverse diffusion kernel (RDK)*.
- 2. It stands for the PDF of x_{t-1} given x_t as parameterized by θ_{θ} . The θ_{θ} means that the parameters of the distribution of the reverse process are learned using a neural network.

3. It's the “transition function” applied at each step in the *reverse diffusion process*.

Mathematical Details Of The Forward Diffusion Process

The distribution q in the forward diffusion process is defined as *Markov Chain* given by:



1. We begin by taking an image from our dataset: x_0 . Mathematically it's stated as sampling a data point from the original (but unknown) data distribution: $x_0 \sim q(x_0)$.
2. The PDF of the forward process is the product of individual distribution starting from timestep $1 \rightarrow T$.
3. The forward diffusion process is fixed and known.
4. All the intermediate noisy images starting from timestep 1 to T are also called “latents.” The dimension of the latents is the same as the original image.
5. The PDF used to define the FDK is a “Normal/Gaussian distribution” (eqn. 2).
6. At each timestep t , the parameters that define the distribution of image x_t are set as:



- Mean:
- Covariance:



7. The term β (*beta*) is known as the “diffusion rate” and is precalculated using a “*variance scheduler*”. The term I is an identity matrix. Therefore, the distribution at each time step is called *Isotropic Gaussian*.
8. The original image is corrupted at each time step by adding a small amount of gaussian noise (ϵ). The amount of noise added is regulated by the scheduler.

9. By choosing sufficiently large timesteps and defining a well-behaved schedule of β_t the repeated application of FDK gradually converts the data distribution to be nearly an isotropic gaussian distribution.

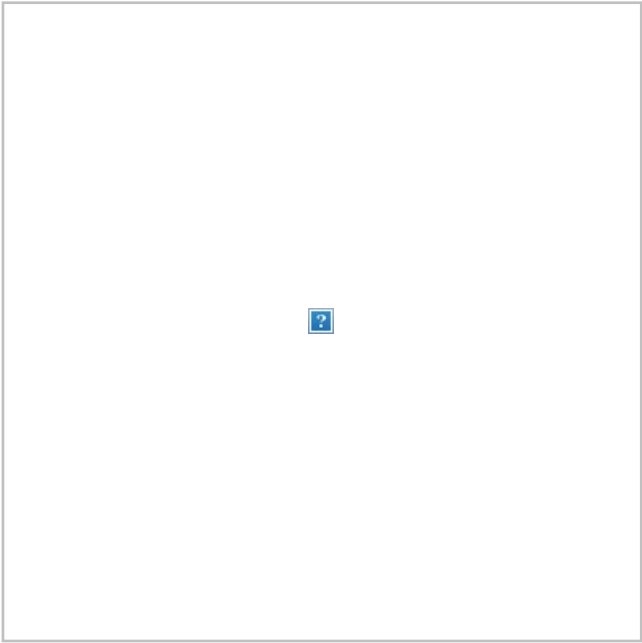
A modified image of diffusion process illustration focusing on forward diffusion process.



How do we get image x_t from x_{t-1} and how is noise added at each time step?

This can be easily understood by using the [reparameterization trick in variational autoencoders](#).

Referring to [the second equation](#), we can easily sample image x_t from a normal distribution as:



1. Here, the epsilon ϵ is the “noise” term that is randomly sampled from the standard gaussian distribution and is first scaled and then added (scaled) $x_{(t-1)}$.
2. In this way, starting from x_0 , the original image is iteratively corrupted from $t=1 \dots T$

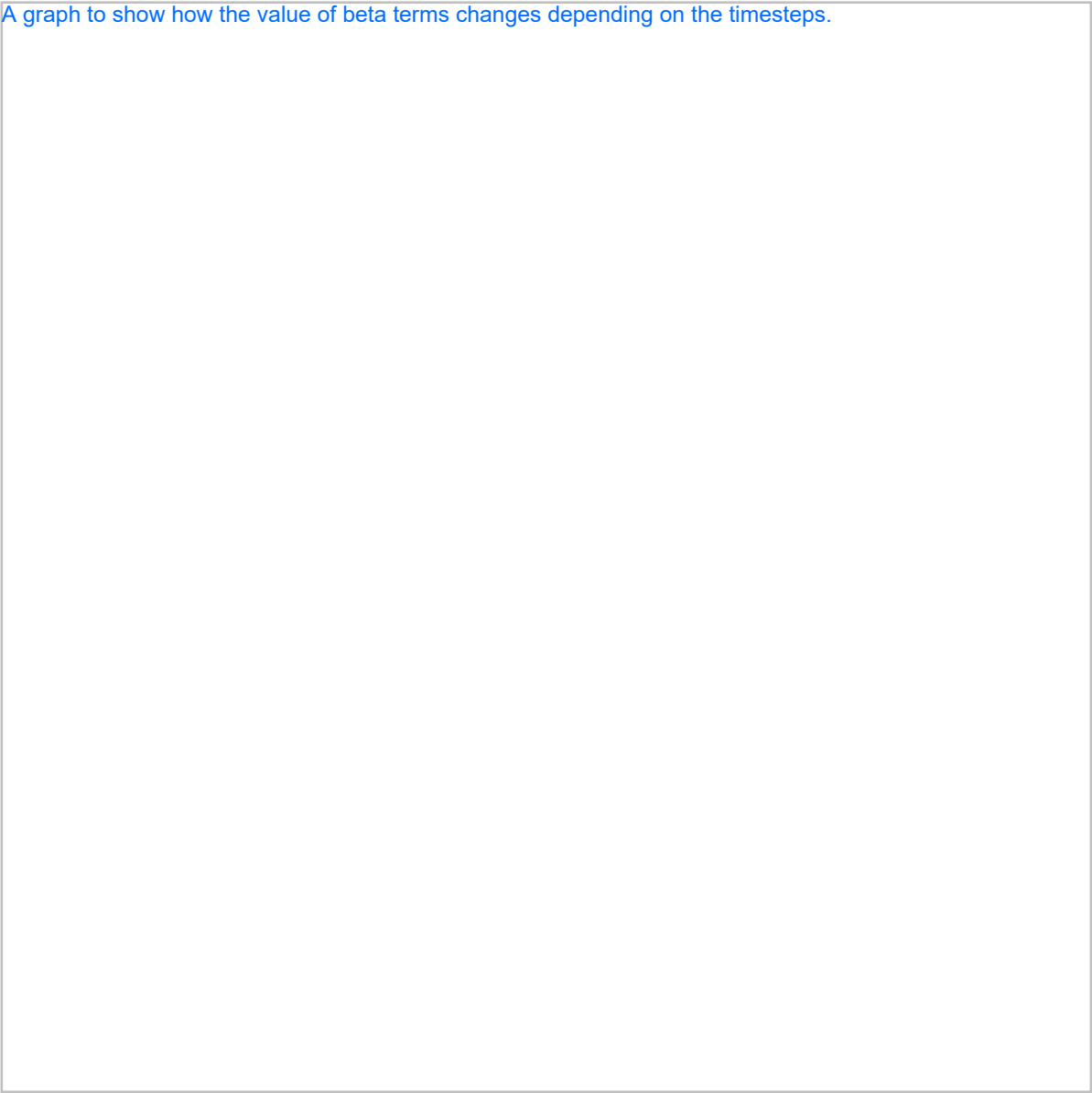
In practice, the authors of DDPMs use a “*linear variance scheduler*” and define β_t in the range $[0.0001, 0.02]$ and set the total timesteps $T = 1000$




“Diffusion models scale down the data with each forward process step (by a β_t factor) so that variance does not grow when adding noise.”

– Denoising Diffusion Probabilistic Models, 2020

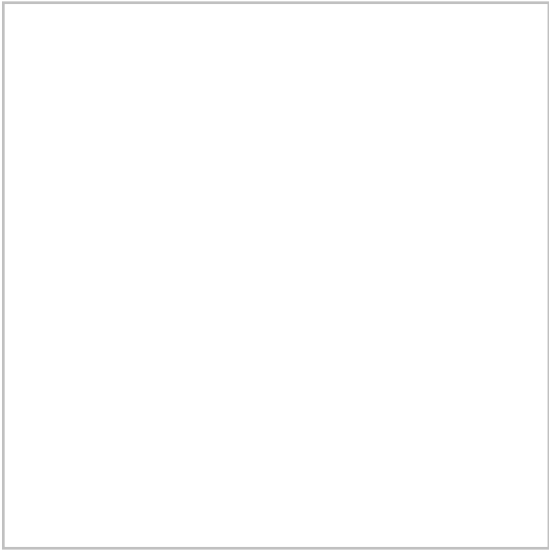
A graph to show how the value of beta terms changes depending on the timesteps.



Variance Scheduler vs timesteps

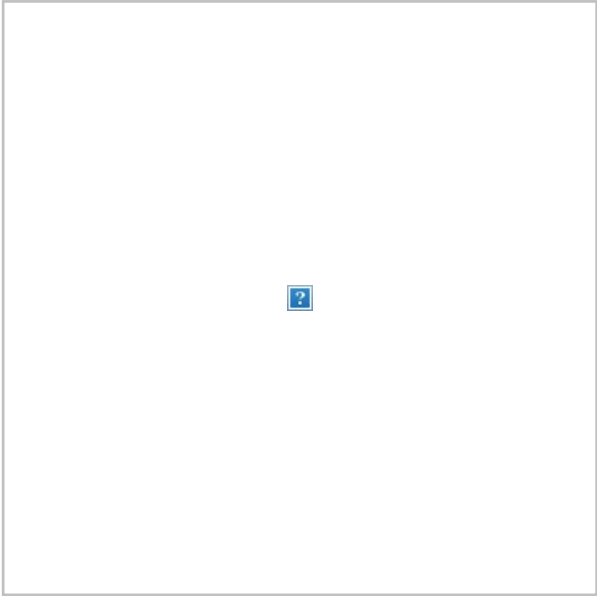
There’s a problem here, which results in an inefficient forward process .

Whenever we need a latent sample x at timestep t , we have to perform $t-1$ steps in the Markov chain.



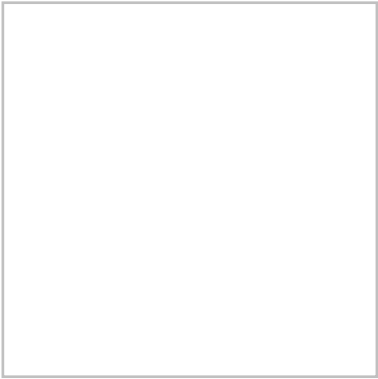
We have to follow through all $t-1$ intermediate states in Markov Chain to get x_t

To fix this, the authors of the DDPM reformulated the kernel to directly go from timestep 0 (i.e., from the original image) to timestep t in the process.



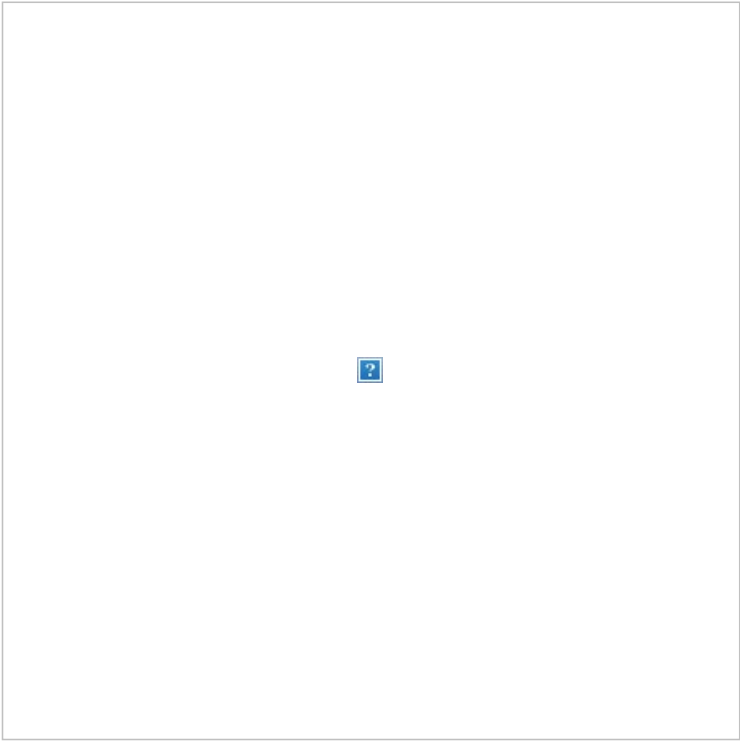
Skipping intermediate steps

To do so, two additional terms are defined:



where eqn. (5) is a cumulative product of β_t from 1 to t.

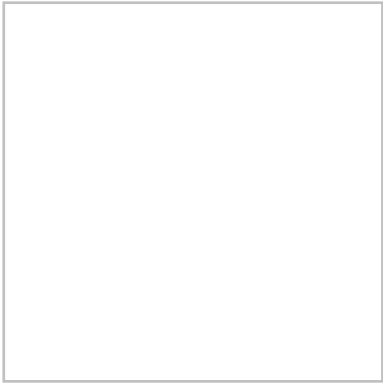
And then, by substituting \mathbf{x}_t 's with \mathbf{z}_t 's and using the [addition property](#) of Gaussian distribution. The forward diffusion process can be rewritten in terms of \mathbf{z}_t as:



Using the above formulation, we can sample at any arbitrary timestep t in the Markov chain.

That's all for the forward diffusion process.

Mathematical Details Of The Reverse Diffusion Process



*Czech Hiking Markers System.
Following the path to take in the
return journey.*

“In the reverse diffusion process, the task is to learn a finite-time (within T timesteps) reversal of the forward diffusion process.”

This basically means that we have to “undo” the forward process i.e., to remove the noise added in the forward process iteratively. It is done using a neural network model.

In the forward process, the transitions function q was defined using a Gaussian, so what function should be used for the reverse process p ? What should the neural network learn?

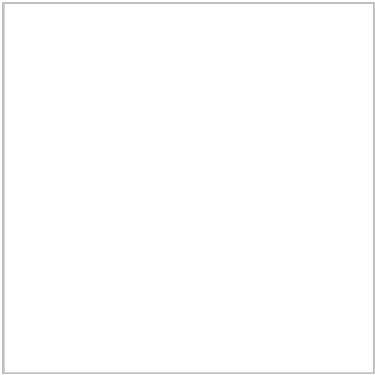
1. In 1949, W. Feller showed that, for gaussian (and binomial) distributions, the diffusion process’s reversal has the same functional form as the forward process.
2. This means that similar to the FDK, which is defined as a normal distribution, we can use the same functional form (a gaussian distribution) to define the reverse diffusion kernel.
3. The reverse process is also a Markov chain where a neural network predicts the parameters for the reverse diffusion kernel at each timestep.
4. During training, the learned estimates (of the parameters) should be close to the parameters of the FDK’s posterior at each timestep. We’ll talk more about [FDK’s posterior in the next section](#).
5. We want this because if we follow the forward trajectory in reverse, we may return to the original data distribution.
6. In doing so, we would also learn how to generate new samples that closely match the underlying data distribution, starting from a pure gaussian noise (we do not have access to the forward process during inference).

A modified illustration of diffusion process focusing on reverse diffusion process.

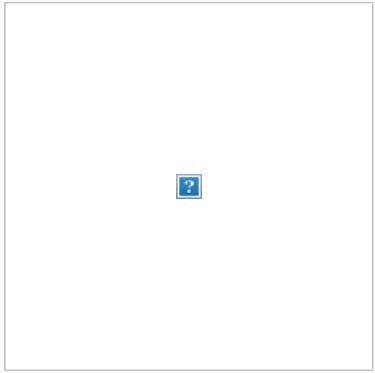


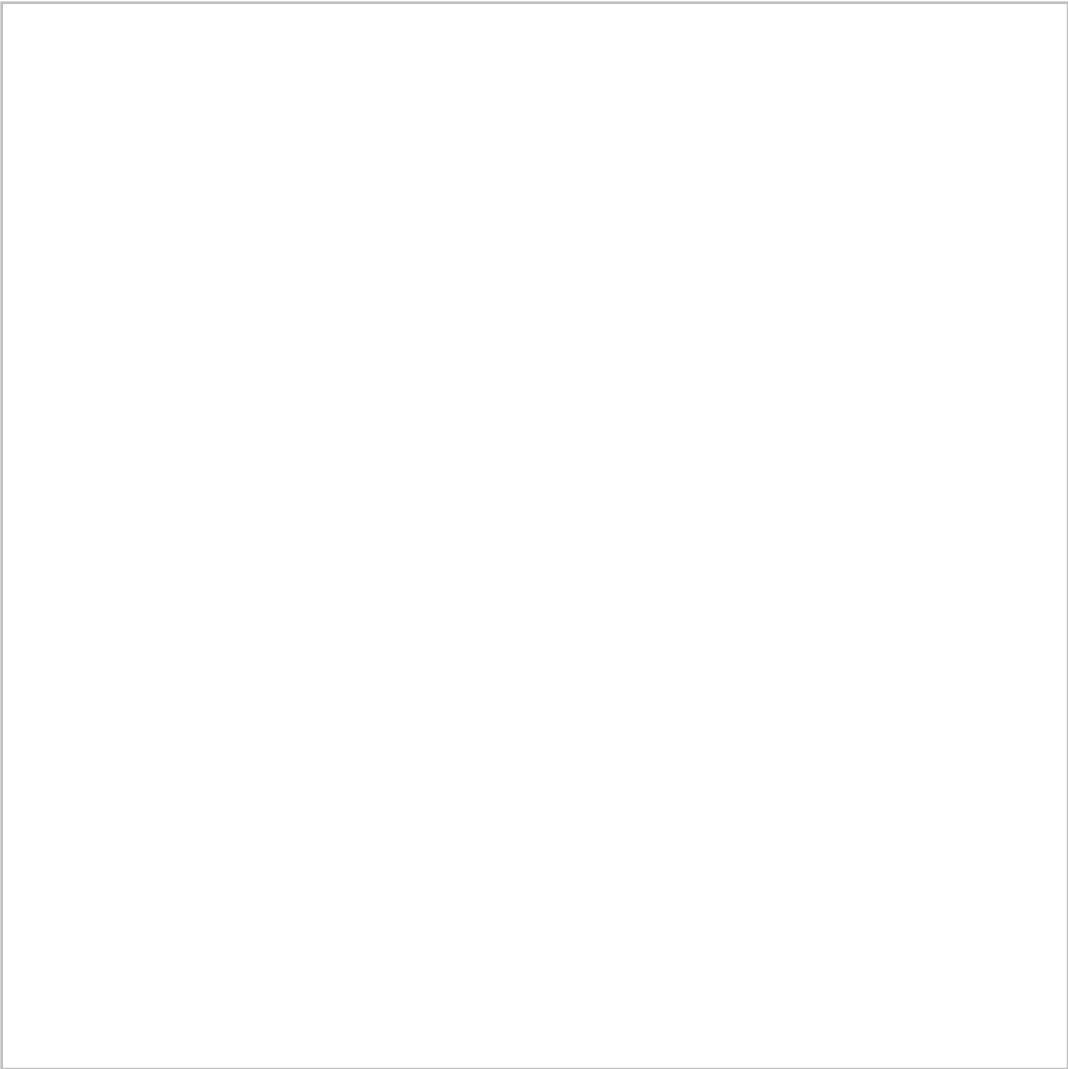
1. The Markov chain for the reverse diffusion starts from where the forward process ends, i.e., at timestep T , where the data distribution has been converted into (nearly an) isotropic gaussian distribution.

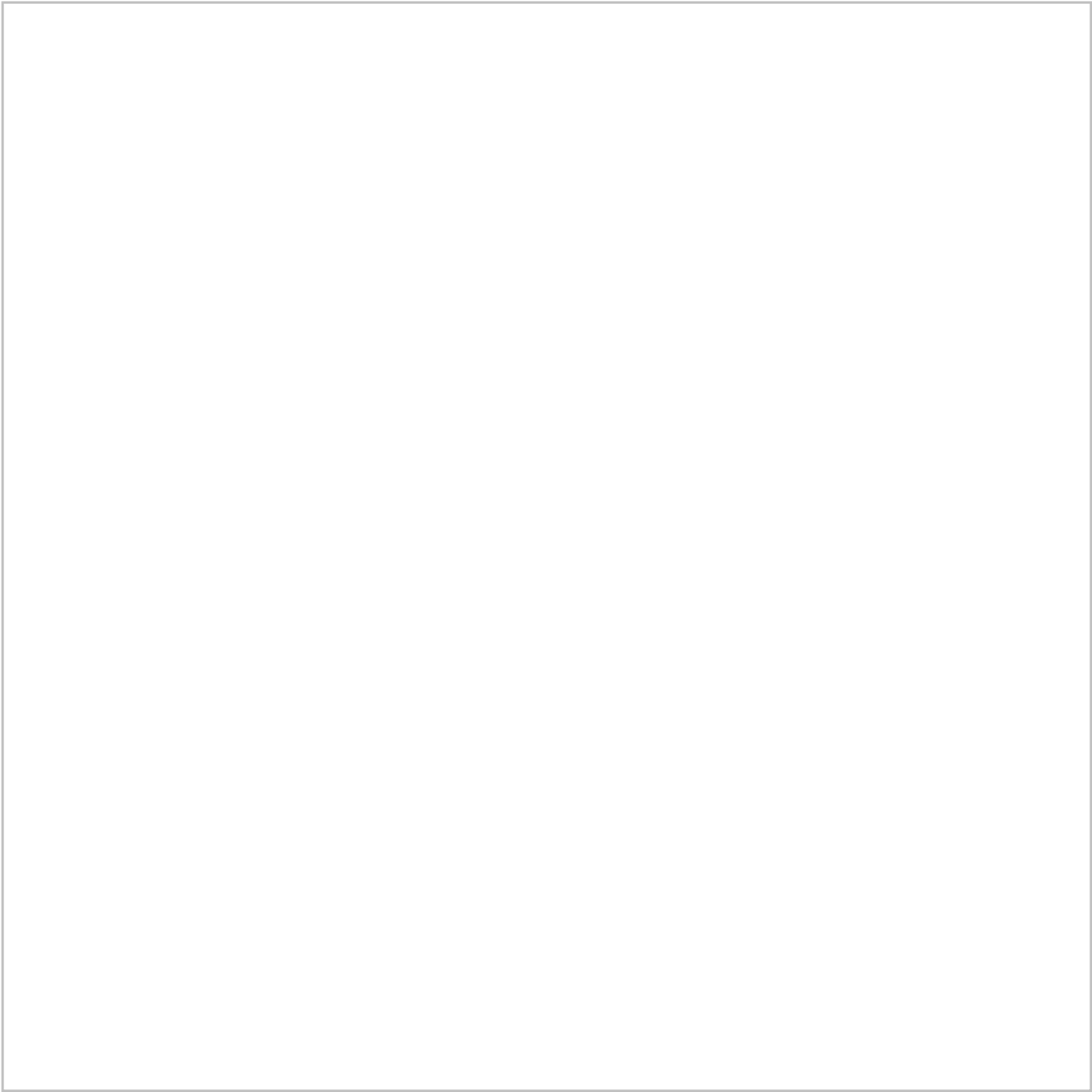




2. The PDF of the reverse diffusion process is an “integral” over all the possible pathways we can take to arrive at a data sample (in the same distribution as the original) starting from pure noise x_T .







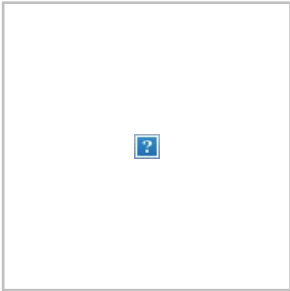
All equations related to the forward and reverse diffusion processes.

Training Objective & Loss Function Used In Denoising Diffusion Probabilistic Models

The training objective of diffusion-based generative models amounts to “*maximizing the log-likelihood of the sample generated (at the end of the reverse process) (x) belonging to the original data distribution.*”

We have defined the transition functions in diffusion models as “Gaussians”. To maximize the log-likelihood of a gaussian distribution, it is to try and find the parameters of the distribution (μ, σ^2) such that it maximizes the “*likelihood*” of the (generated) data belonging to the same data distribution as the original data.

To train our neural network, we define the loss function (L) as the objective function’s negative. So a high value for $p_{\mu, \sigma}(x_0)$, means low loss and vice versa.

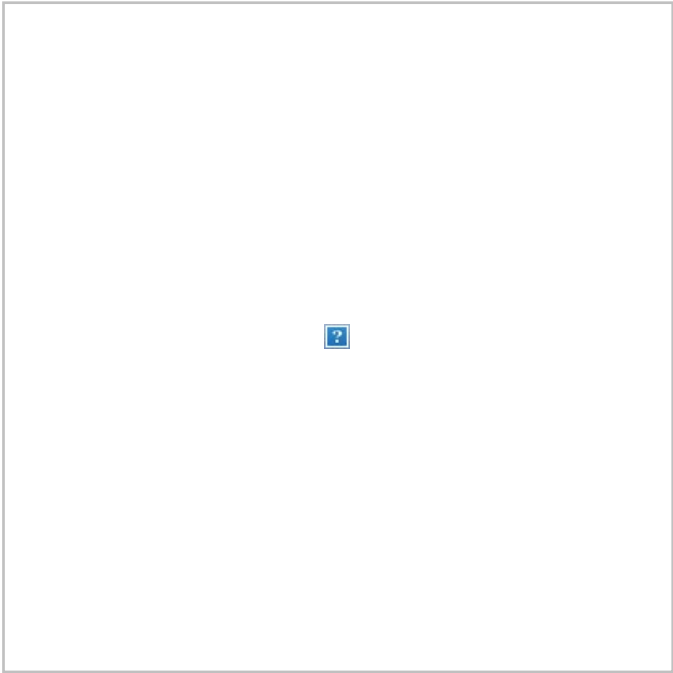


Turns out, this is intractable because we need to integrate over a very high dimensional (pixel) space for continuous values over T timesteps.

Instead, the authors take inspiration from VAEs and reformulate the training objective using a *variational lower bound (VLB)*, also known as “*Evidence lower bound*” (*ELBO*), which is this scary-looking equation $\diamond\diamond$:

[The ELBO loss term as defined in diffusion probabilistic models.](#)



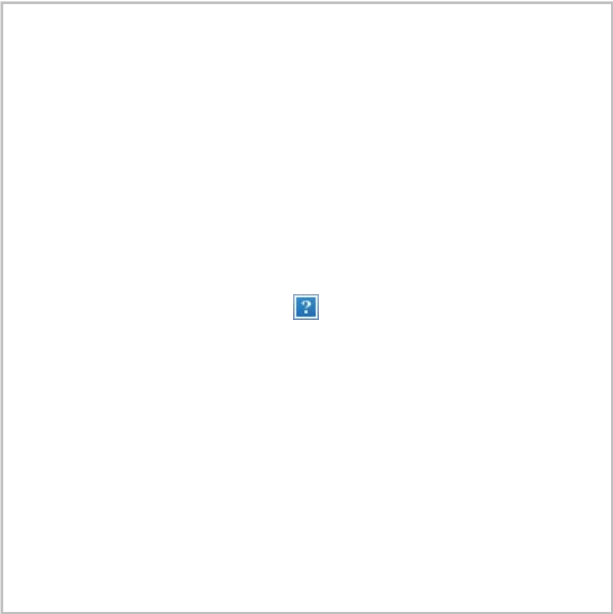


Prof. Andrew Ng to the rescue 🤖🤖🤖🤖

After some simplification, the DDPM authors arrive at this final L_{vlb} – Variational Lower Bound loss term:

The simplification of the ELBO loss as done in denoising diffusion probabilistic models (DDPMs).

We can break the above L_{vb} loss term into individual timestep as follows:

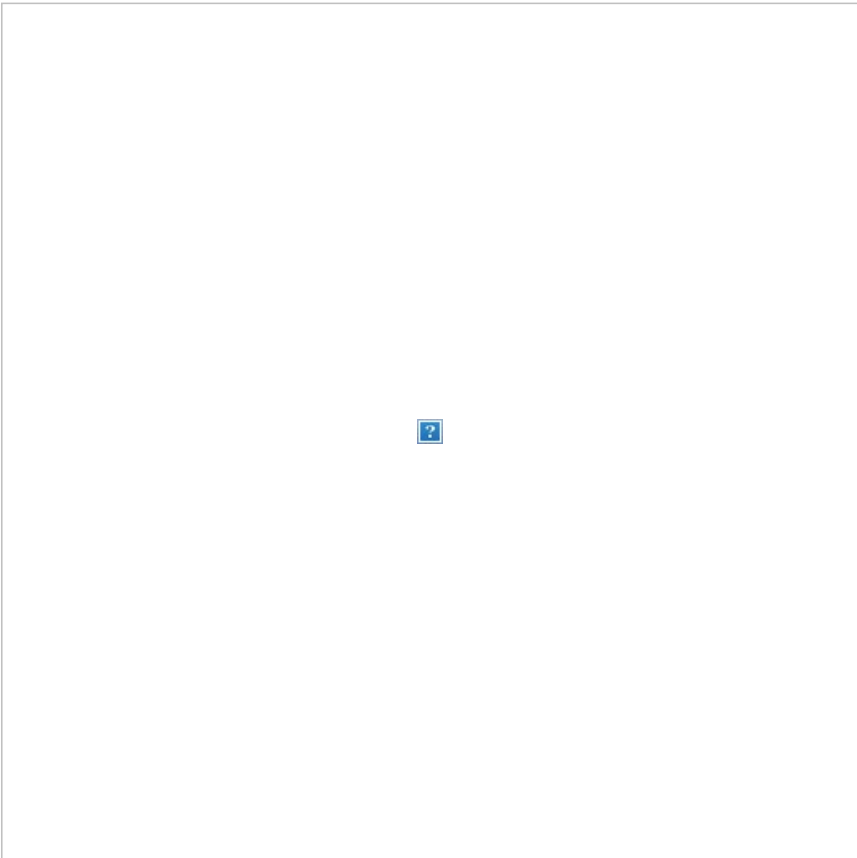


You may notice that this loss function is huge! But the authors of DDPM further simplify it by ignoring some of the terms in their simplified loss function.

The terms ignored are:

- 1. L_0 – The authors got better results without this.
- 2. L_T – This is the “*KL divergence*” between the distribution of the final latent in the forward process and the first latent in the reverse process. However, there are no neural network parameters involved here, so we can’t do anything about it except define a good variance scheduler and use large timesteps such that they both represent an Isotropic Gaussian distribution.

So L_{t-1} is the only loss term left which is a KL divergence between the “posterior” of the forward process (conditioned on x_t and the initial sample x_0), and the parameterized reverse diffusion process. Both terms are gaussian distributions as well.



The term $q(x_{t-1}|x_t, x_0)$ is referred to as “*forward process posterior distribution.*”

The job of our deep-learning model during training is to approximate/estimate the parameters of this (gaussian) posterior such that the KL divergence is as minimal as possible.

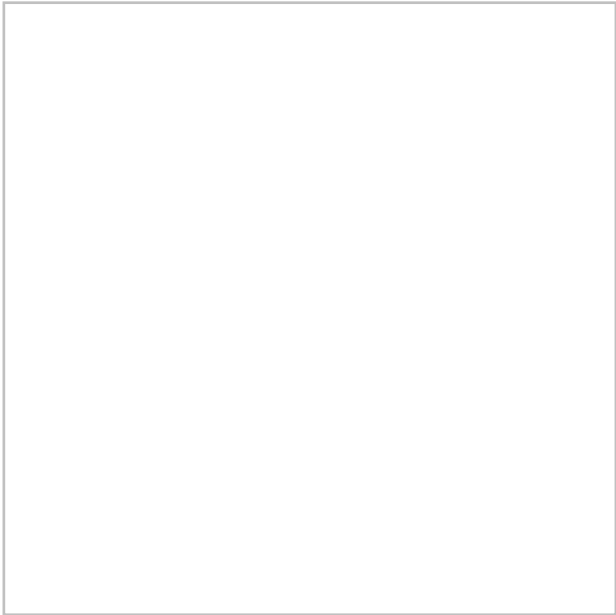


The parameters of the posterior distribution are as follows:

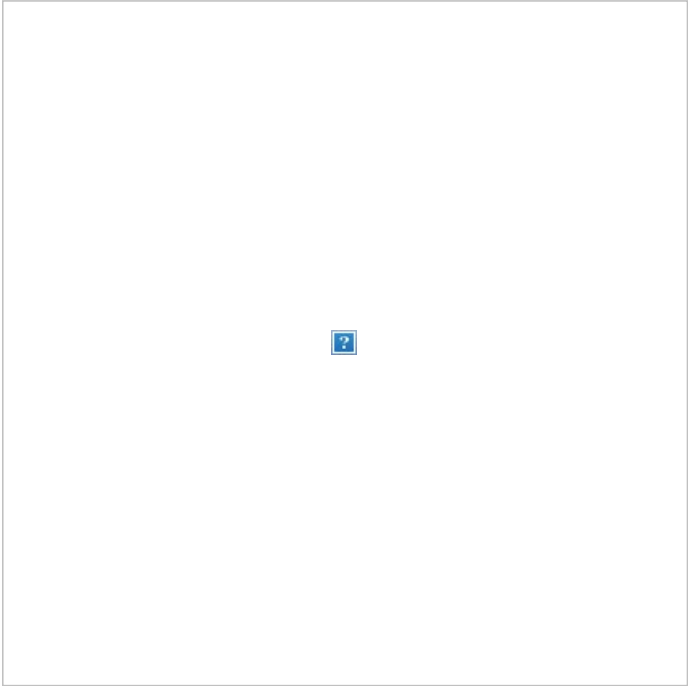
The formulation of the forward process posterior distribution.

To further simplify the task of the model, *the authors decided to fix the variance to a constant β_t* .

Now, the model only needs to learn to predict the above equation. And the reverse diffusion kernel gets modified to:



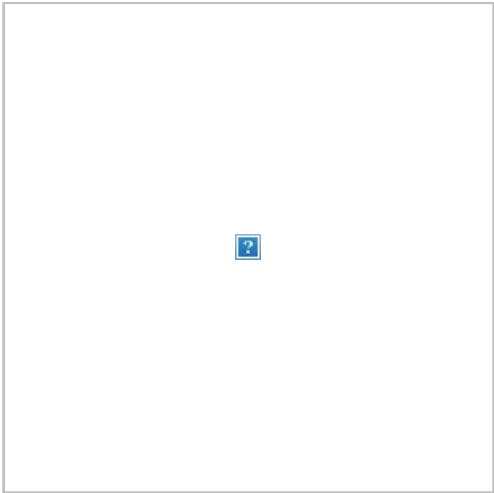
As we have kept the variance constant, minimizing KL divergence is as simple as minimizing the difference (or distance) between means (💎💎) of two gaussian distributions q and p ([for e.g. difference between the means of distributions in the left image](#)), which can be done as follows:



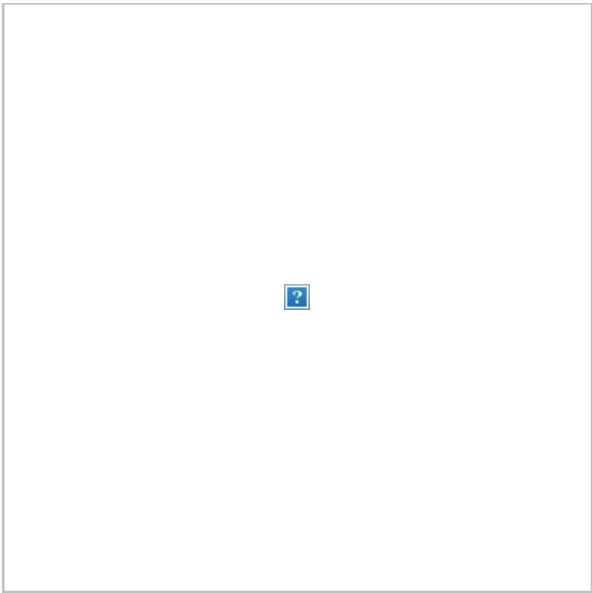
Now, there are three approaches we can take here:

- 1. Directly predict x_0 and find \square using it in the [posterior function](#).
- 2. Predict the entire \square term.
- 3. Predict the noise at each timestep. This is done by writing the x_0 in \square in terms of x_t using the reparameterization trick.

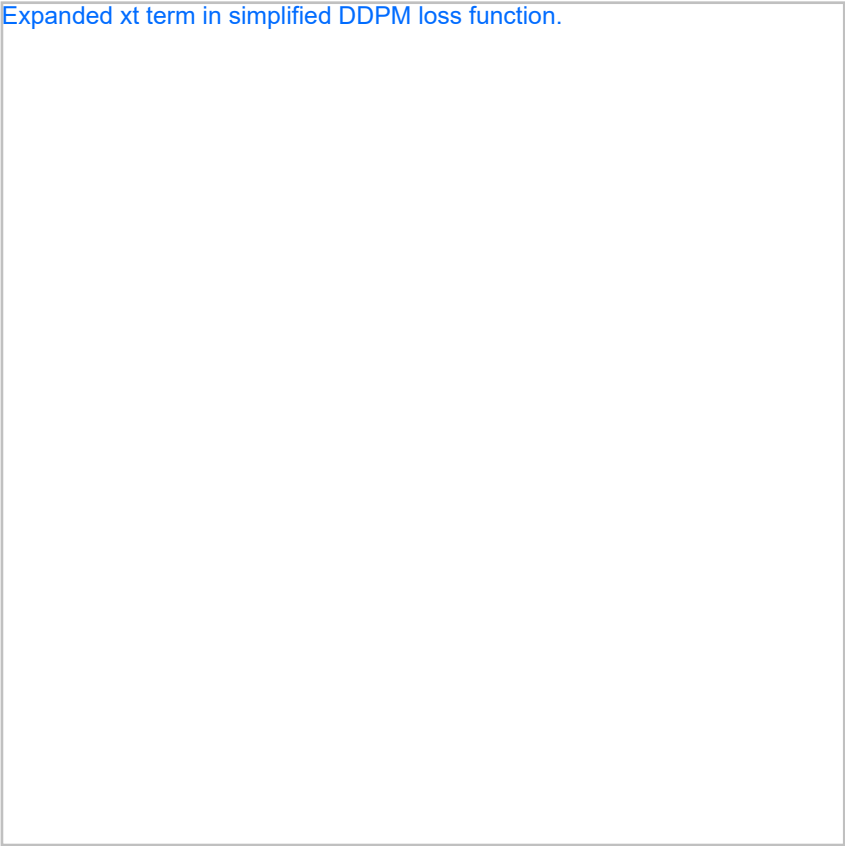
By using the third option, and after some simplification, \square can be expressed as:



Similarly, the formulation for $\epsilon_{\theta}(x_t, t)$ is set to:

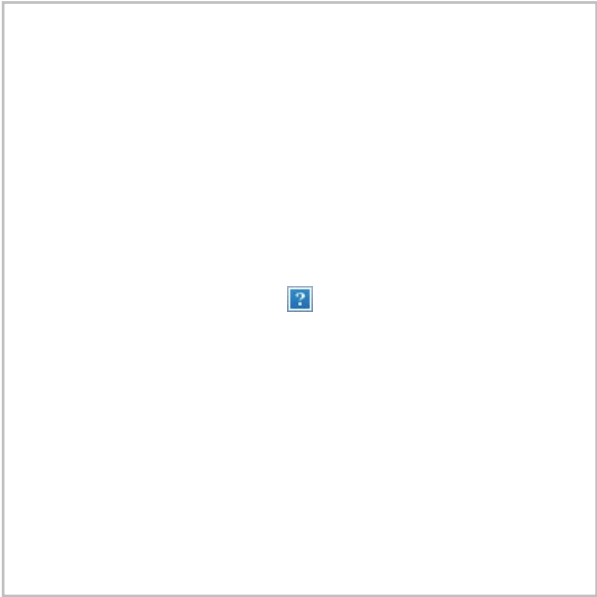


At training and inference time, we know the x_0 's, x_t 's, and x_1 . So our model only needs to predict the noise at each timestep. The simplified (after ignoring some weighting terms) loss function used in the *Denoising Diffusion Probabilistic Models* is as follows:



Comparing just the noise.

Which is basically:



This is the final loss function we use to train DDPMs, which is just a “Mean Squared Error” between the noise added in the forward process and the noise predicted by the model. This is the most impactful contribution of the paper Denoising Diffusion Probabilistic Models.

It’s awesome because, beginning from those scary-looking ELBO terms, we ended up with the simplest loss function in the entire machine learning domain.

Introduced in 2014 by [Ian Goodfellow](#), Generative Adversarial Networks (GANs) were the norm for generating image samples.

Many variations from the original GANs were created, such as:

1. [Conditional GAN \(cGAN\)](#): Controlling the class/category of the generated images.
2. [Deep Convolutional GAN \(DCGAN\)](#): architecture significantly improves the quality of GANs using convolutional layers.
3. [Image-to-Image translation with Pix2Pix](#): Converting images from one domain to another by learning a mapping between the input and output.

Writing DDPMs From Scratch In PyTorch

From this section, we'll code all the essential components required for training denoising diffusion probabilistic models from scratch in PyTorch. Instead of Colab, we used Kaggle kernels as it provides better GPUs than Colab free version and longer training times (which is crucial for diffusion models).

Note: code for regularly used helper functions is not added to the post.

💎💎 You can access the entire codebase for this and all our other posts by simply subscribing to the blog post, and we'll send you the link to download link.

Download Code To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

Download Code

First and foremost, we'll define configuration classes that will hold the hyperparameters for loading the dataset, creating log directories, and training the model.

```
1  from dataclasses import dataclass
2
3  @dataclass
4  class BaseConfig:
5      DEVICE = get_default_device()
6      DATASET = "Flowers" # "MNIST", "Cifar-10", "Flowers"
7
8      # For logging inference images and saving checkpoints.
9      root_log_dir = os.path.join("Logs_Checkpoints", "Inference")
10     root_checkpoint_dir = os.path.join("Logs_Checkpoints", "checkpoints")
11
12     # Current log and checkpoint directory.
13     log_dir = "version_0"
14     checkpoint_dir = "version_0"
15
16
17 @dataclass
18 class TrainingConfig:
```

```

19 Timesteps = 1000 # Define number of diffusion timesteps
20 IMG_SHAPE = (1, 32, 32) if BaseConfig.DATASET == "MNIST" else (3, 32, 32)
21 NUM_EPOCHS = 800
22 BATCH_SIZE = 32
23 LR = 2e-4
24 NUM_WORKERS = 2

```

Creating PyTorch Dataset Class Object

This article uses the “Flowers” dataset, which can be downloaded from Kaggle or quickly loaded in the Kaggle kernel environment. But as you may have noticed, in the `BaseConfig` class, we have also provided the option to load the MNIST, Cifar-10 and Cifar-100 datasets. You can choose whichever one you prefer.

The flowers dataset can be downloaded from over here: [Flowers Recognition | Kaggle](#)

When using Kaggle kernels, it’s as simple as just clicking on the “Add Data” component and selecting the dataset.

Here, we are creating two functions:

1. `get_dataset(...)`: Returns the dataset class object that will be passed to the Dataloader. Three preprocessing transforms, and one augmentation are applied to every image in the dataset.
 1. *Preprocessing*:
 1. Convert pixel values from the range $[0, 255] \rightarrow [0.0, 1.0]$
 2. Resize Images to shape (32×32)
 3. Change pixel values from the range $[0.0, 1.0] \rightarrow [-1.0, 1.0]$. This is done by the DDPM authors so that the input image roughly has the same range of values as a standard gaussian.
 2. *Augmentation*:
 1. A random horizontal flip, as used in the original implementation. In case you are using the MNIST dataset, be sure to comment out this line.
2. `inverse_transforms(...)`: This function is used for inverting the transforms applied during the loading step and reverting the image to the range $[0.0, 255.0]$.

```

1 import torchvision
2 import torchvision.transforms as TF
3 import torchvision.datasets as datasets
4 from torch.utils.data import Dataset, DataLoader
5
6
7 def get_dataset(dataset_name='MNIST'):
8     transforms = torchvision.transforms.Compose(
9         [
10             torchvision.transforms.ToTensor(),
11             torchvision.transforms.Resize((32, 32),
12                                           interpolation=torchvision.transforms.InterpolationMode.BILINEAR,
13                                           antialias=True),
14             torchvision.transforms.RandomHorizontalFlip(),
15             # torchvision.transforms.Normalize(MEAN, STD),
16             torchvision.transforms.Lambda(lambda t: (t * 2) - 1) # Scale between [-1, 1]
17         ]
18     )
19
20     if dataset_name.upper() == "MNIST":
21         dataset = datasets.MNIST(root="data", train=True, download=True, transform=transforms)
22     elif dataset_name == "Cifar-10":
23         dataset = datasets.CIFAR10(root="data", train=True, download=True, transform=transforms)
24     elif dataset_name == "Cifar-100":
25         dataset = datasets.CIFAR100(root="data", train=True, download=True, transform=transforms)
26     elif dataset_name == "Flowers":

```

```

27         dataset = datasets.ImageFolder(root="/kaggle/input/flowers-recognition/flowers",
28         transform=transforms)
29
30         return dataset
31
32     def inverse_transform(tensors):
33         """Convert tensors from [-1., 1.] to [0., 255.]"""
34         return ((tensors.clamp(-1, 1) + 1.0) / 2.0) * 255.0

```

Creating PyTorch Dataloader Class Object

Next, we define the `get_dataloader(...)` function that returns a `Dataloader` object for the chosen dataset.

```

1     def get_dataloader(dataset_name='MNIST',
2                         batch_size=32,
3                         pin_memory=False,
4                         shuffle=True,
5                         num_workers=0,
6                         device="cpu"
7                     ):
8         dataset = get_dataset(dataset_name=dataset_name)
9         dataloader = DataLoader(dataset, batch_size=batch_size,
10                                pin_memory=pin_memory,
11                                num_workers=num_workers,
12                                shuffle=shuffle
13                            )
14         # Used for moving batch of data to the user-specified machine: cpu or gpu
15         device_dataloader = DeviceDataLoader(dataloader, device)
16         return device_dataloader

```

Visualizing Dataset

First, we'll create the "dataloader" object by calling the `get_dataloader(...)` function.

```

1     loader = get_dataloader(
2         dataset_name=BaseConfig.DATASET,
3         batch_size=128,
4         device="cpu",
5     )

```

Then we can simply use `torchvision`'s `make_grid(...)` function to plot a grid of flower images.

```

1     from torchvision.utils import make_grid
2
3     plt.figure(figsize=(10, 4), facecolor='white')
4
5     for b_image, _ in loader:
6         b_image = inverse_transform(b_image)
7         grid_img = make_grid(b_image / 255.0, nrow=16, padding=True, pad_value=1)
8         plt.imshow(grid_img.permute(1, 2, 0))
9         plt.axis("off")
10        break

```

The flowers dataset used for training DDPMs from scratch.



Flowers Dataset

Model Architecture Used In DDPMs

In DDPMs, the authors use a UNet-shaped deep neural network which takes in as input:

- 1. The input image at any stage of the reverse process.
- 2. The timestep of the input image.

From the usual UNet architecture, the authors replaced the original double convolution at each level with “Residual blocks” used in ResNet models.

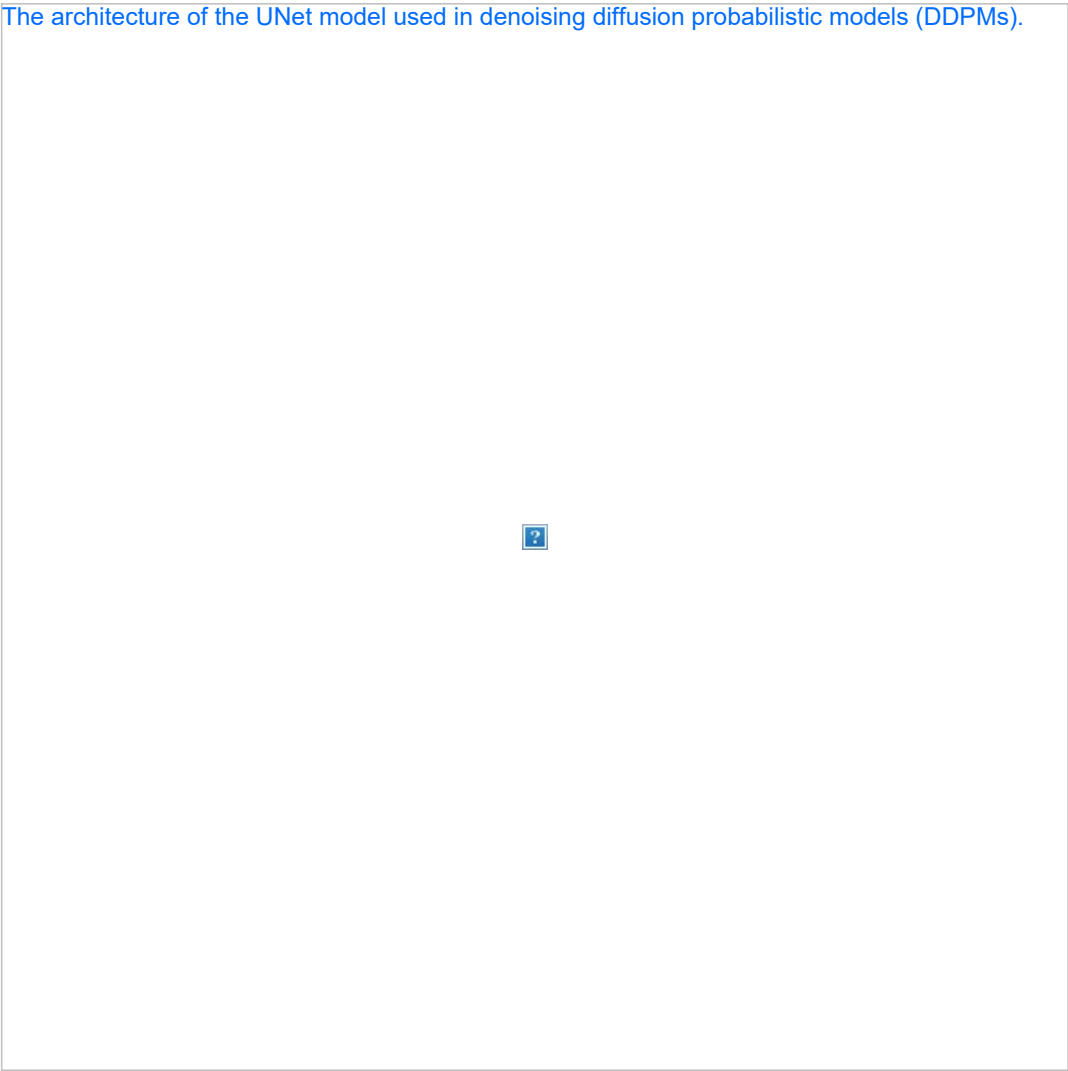
The architecture comprises 5 components:

- 1. Encoder blocks
- 2. Bottleneck blocks
- 3. Decoder blocks
- 4. [Self attention modules](#)
- 5. Sinusoidal time embeddings

Architectural Details:

1. There are four levels in the encoder and decoder path with bottleneck blocks between them.
2. Each encoder stage comprises two residual blocks with convolutional downsampling except the last level.
3. Each corresponding decoder stage comprises three residual blocks and uses 2x nearest neighbors with convolutions to upsample the input from the previous level.
4. Each stage in the encoder path is connected to the decoder path with the help of skip connections.
5. The model uses “Self-Attention” modules at a single feature map resolution.
6. Every residual block in the model gets the inputs from the previous layer (and others in the decoder path) and the embedding of the current timestep. The timestep embedding informs the model of the input’s current position in the Markov chain.

The architecture of the UNet model used in denoising diffusion probabilistic models (DDPMs).



The U-Net architecture used in DDPMs

In this article, we are working on an image size of (32×32). Only two minor changes exist between our model and the original model used in the paper.

1. We use 64 base channels instead of 128.
2. There are four levels in both encoder and decoder paths. The feature maps size at each level are kept as follows: 32 → 16 → 8 → 8. We are applying self-attention at feature map sizes of both (16×16) and (8×8) as opposed to the original, where they are applied just once at a feature map size of (16×16).

Please note that we are not adding the model code because the code for the UNet + these modifications is quite easy, but because of all the different components. it becomes just too big to be added to the post.

Diffusion Class

In this section, we are creating a class called SimpleDiffusion. This class contains:

1. Scheduler constants required for performing the forward and reverse diffusion process.
2. A method to define the linear variance scheduler used in DDPMs.
3. A method that performs a single step using the updated forward diffusion kernel.

```

1  class SimpleDiffusion:
2      def __init__(
3          self,
4          num_diffusion_timesteps=1000,
5          img_shape=(3, 64, 64),
6          device="cpu",
7      ):
8          self.num_diffusion_timesteps = num_diffusion_timesteps
9          self.img_shape = img_shape
10         self.device = device
11         self.initialize()
12
13     def initialize(self):
14         # BETAS & ALPHAS required at different places in the Algorithm.
15         self.beta = self.get_betas()
16         self.alpha = 1 - self.beta
17
18         self.sqrt_beta = torch.sqrt(self.beta)
19         self.alpha_cumulative = torch.cumprod(self.alpha, dim=0)
20         self.sqrt_alpha_cumulative = torch.sqrt(self.alpha_cumulative)
21         self.one_by_sqrt_alpha = 1. / torch.sqrt(self.alpha)
22         self.sqrt_one_minus_alpha_cumulative = torch.sqrt(1 - self.alpha_cumulative)
23
24     def get_betas(self):
25         """linear schedule, proposed in original ddpm paper"""
26         scale = 1000 / self.num_diffusion_timesteps
27         beta_start = scale * 1e-4
28         beta_end = scale * 0.02
29         return torch.linspace(
30             beta_start,
31             beta_end,
32             self.num_diffusion_timesteps,
33             dtype=torch.float32,
34             device=self.device,
35         )

```

Python Code For Forward Diffusion Process

In this section, we are writing the python code to perform the “forward diffusion process” in a single step as per the equation mentioned here.

The `forward_diffusion(...)` function takes in a batch of images and corresponding timesteps and adds noise/corrupts the input images using the [updated forward diffusion kernel](#) equation.

```

1  def forward_diffusion(sd: SimpleDiffusion, x0: torch.Tensor, timesteps: torch.Tensor):
2      eps = torch.randn_like(x0) # Noise
3      mean = get(sd.sqrt_alpha_cumulative, t=timesteps) * x0 # Image scaled
4      std_dev = get(sd.sqrt_one_minus_alpha_cumulative, t=timesteps) # Noise scaled
5      sample = mean + std_dev * eps # scaled inputs * scaled noise
6
7      return sample, eps # return ... , gt noise --> model predicts this

```

Visualizing Forward Diffusion Process On Sample Images

In this section, we'll visualize the forward diffusion process on some sample images to see how they get corrupted as they pass through the Markov chain for T timesteps.

```

1 sd = SimpleDiffusion(num_diffusion_timesteps=TrainingConfig.TIMESTEPS, device="cpu")
2
3 loader = iter( # converting dataloader into an iterator for now.
4     get_dataloader(
5         dataset_name=BaseConfig.DATASET,
6         batch_size=6,
7         device="cpu",
8     )
9 )

```

Performing the forward process for some specific timesteps and also storing the noisy versions of the original image.

```

1 x0s, _ = next(loader)
2
3 noisy_images = []
4 specific_timesteps = [0, 10, 50, 100, 150, 200, 250, 300, 400, 600, 800, 999]
5
6 for timestep in specific_timesteps:
7     timestep = torch.as_tensor(timestep, dtype=torch.long)
8
9     xts, _ = sd.forward_diffusion(x0s, timestep)
10    xts = inverse_transform(xts) / 255.0
11    xts = make_grid(xts, nrow=1, padding=1)
12
13    noisy_images.append(xts)

```

Plotting sample corruption at different timesteps.

```

1 _, ax = plt.subplots(1, len(noisy_images), figsize=(10, 5), facecolor='white')
2
3 for i, (timestep, noisy_sample) in enumerate(zip(specific_timesteps, noisy_images)):
4     ax[i].imshow(noisy_sample.squeeze(0).permute(1, 2, 0))
5     ax[i].set_title(f"t={timestep}", fontsize=8)
6     ax[i].axis("off")
7     ax[i].grid(False)
8
9 plt.suptitle("Forward Diffusion Process", y=0.9)
10 plt.axis("off")
11 plt.show()

```

Images are corrupted in the forward process of diffusion probabilistic models.



The original image gets increasingly corrupted as timesteps increase. At the end of the forward process, we are left with noise.

Training & Sampling Algorithms Used In Denoising Diffusion Probabilistic Models

The training and sampling algorithm as described in the DDPMs paper.

Training code based on Algorithm 1:

The first function defined here is `train_one_epoch(...)`. This function is used for performing “one epoch of training ” i.e., it trains the model by iterating once over the entire dataset and will be called in our final training loop.

We also use Mixed-Precision training to train the model faster and save GPU memory. The code is pretty straightforward and almost a one-to-one conversion from the algorithm.

```
1  # Algorithm 1: Training
2
3  def train_one_epoch(model, loader, sd, optimizer, scaler, loss_fn, epoch=800,
4                      base_config=BaseConfig(), training_config=TrainingConfig()):
5
6      loss_record = MeanMetric()
7      model.train()
8
9      with tqdm(total=len(loader), dynamic_ncols=True) as tq:
10         tq.set_description(f"Train :: Epoch: {epoch}/{training_config.NUM_EPOCHS}")
11
12         for x0s, _ in loader: # line 1, 2
13             tq.update(1)
14
15             ts = torch.randint(low=1, high=training_config.TIMESTEPS, size=(x0s.shape[0],),
16                               device=base_config.DEVICE) # line 3
17             xts, gt_noise = sd.forward_diffusion(x0s, ts) # line 4
18
19             with amp.autocast():
20                 pred_noise = model(xts, ts)
```

```

21         loss = loss_fn(gt_noise, pred_noise) # line 5
22
23         optimizer.zero_grad(set_to_none=True)
24         scaler.scale(loss).backward()
25
26         # scaler.unscale_(optimizer)
27         # torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
28
29         scaler.step(optimizer)
30         scaler.update()
31
32         loss_value = loss.detach().item()
33         loss_record.update(loss_value)
34
35         tq.set_postfix_str(s=f"Loss: {loss_value:.4f}")
36
37         mean_loss = loss_record.compute().item()
38
39         tq.set_postfix_str(s=f"Epoch Loss: {mean_loss:.4f}")
40
41     return mean_loss

```

Sampling or Inference code based on Algorithm 2:

The next function we define is `reverse_diffusion(...)` which is responsible for performing inference i.e., generating images using the reverse diffusion process. The function takes in a trained model and the diffusion class and can either generate a video showcasing the entire diffusion process or just the final generated image.

```

1  # Algorithm 2: Sampling
2
3  @torch.no_grad()
4  def reverse_diffusion(model, sd, timesteps=1000, img_shape=(3, 64, 64),
5                        num_images=5, nrow=8, device="cpu", **kwargs):
6
7      x = torch.randn((num_images, *img_shape), device=device)
8      model.eval()
9
10     if kwargs.get("generate_video", False):
11         outs = []
12
13     for time_step in tqdm(iterable=reversed(range(1, timesteps)),
14                          total=timesteps-1, dynamic_ncols=False,
15                          desc="Sampling :: ", position=0):
16
17         ts = torch.ones(num_images, dtype=torch.long, device=device) * time_step
18         z = torch.randn_like(x) if time_step > 1 else torch.zeros_like(x)
19
20         predicted_noise = model(x, ts)
21
22         beta_t = get(sd.beta, ts)
23         one_by_sqrt_alpha_t = get(sd.one_by_sqrt_alpha, ts)
24         sqrt_one_minus_alpha_cumulative_t = get(sd.sqrt_one_minus_alpha_cumulative, ts)
25
26         x = (
27             one_by_sqrt_alpha_t
28             * (x - (beta_t / sqrt_one_minus_alpha_cumulative_t) * predicted_noise)
29             + torch.sqrt(beta_t) * z
30         )
31
32     if kwargs.get("generate_video", False):
33         x_inv = inverse_transform(x).type(torch.uint8)
34         grid = make_grid(x_inv, nrow=nrow, pad_value=255.0).to("cpu")
35         ndarr = torch.permute(grid, (1, 2, 0)).numpy()[ :, :, ::-1]
36         outs.append(ndarr)
37
38     if kwargs.get("generate_video", False): # Generate and save video of the entire reverse
39 process.
40         frames2vid(outs, kwargs['save_path'])
41         display(Image.fromarray(outs[-1][ :, :, ::-1])) # Display the image at the final
42 timestep of the reverse process.
43         return None
44
45     else: # Display and save the image at the final timestep of the reverse process.
46         x = inverse_transform(x).type(torch.uint8)
47         grid = make_grid(x, nrow=nrow, pad_value=255.0).to("cpu")
48         pil_image = TF.functional.to_pil_image(grid)
49         pil_image.save(kwargs['save_path'], format=save_path[-3:].upper())

```

```
display(pil_image)
return None
```

Training DDPMs From Scratch

In the previous sections, we have already defined all the necessary classes and functions required for training. All we have to do now is assemble them and start the training process.

Before we begin training:

- We'll first define all the model-related hyperparameters.
- Then initialize the *UNet* model, *AdamW* optimizer, *MSE* loss function, and other necessary classes.

```
1  @dataclass
2  class ModelConfig:
3      BASE_CH = 64 # 64, 128, 256, 256
4      BASE_CH_MULT = (1, 2, 4, 4) # 32, 16, 8, 8
5      APPLY_ATTENTION = (False, True, True, False)
6      DROPOUT_RATE = 0.1
7      TIME_EMB_MULT = 4 # 128
8
9  model = UNet(
10     input_channels = TrainingConfig.IMG_SHAPE[0],
11     output_channels = TrainingConfig.IMG_SHAPE[0],
12     base_channels = ModelConfig.BASE_CH,
13     base_channels_multiples = ModelConfig.BASE_CH_MULT,
14     apply_attention = ModelConfig.APPLY_ATTENTION,
15     dropout_rate = ModelConfig.DROPOUT_RATE,
16     time_multiple = ModelConfig.TIME_EMB_MULT,
17 )
18 model.to(BaseConfig.DEVICE)
19
20 optimizer = torch.optim.AdamW(model.parameters(), lr=TrainingConfig.LR) # Original → Adam
21
22 dataloader = get_dataloader(
23     dataset_name = BaseConfig.DATASET,
24     batch_size = TrainingConfig.BATCH_SIZE,
25     device = BaseConfig.DEVICE,
26     pin_memory = True,
27     num_workers = TrainingConfig.NUM_WORKERS,
28 )
29
30 loss_fn = nn.MSELoss()
31
32 sd = SimpleDiffusion(
33     num_diffusion_timesteps = TrainingConfig.TIMESTEPS,
34     img_shape = TrainingConfig.IMG_SHAPE,
35     device = BaseConfig.DEVICE,
36 )
37
38 scaler = amp.GradScaler() # For mixed-precision training.
```

Then we'll initialize the logging and checkpoint directories to save intermediate sampling results and model parameters.

```
1  total_epochs = TrainingConfig.NUM_EPOCHS + 1
2  log_dir, checkpoint_dir = setup_log_directory(config=BaseConfig())
3  generate_video = False
4  ext = ".mp4" if generate_gif else ".png"
```

Finally, we can write our training loop. As we have divided all our code into simple, easy-to-debug functions and classes, all we have to do now is call them in the `epochs` training loop. Specifically, we need to call the “training” and “sampling” functions defined in the previous section in a loop.

```
1  for epoch in range(1, total_epochs):
2      torch.cuda.empty_cache()
3      gc.collect()
```

```
4
5 # Algorithm 1: Training
6 train_one_epoch(model, sd, dataloader, optimizer, scaler, loss_fn, epoch=epoch)
7
8 if epoch % 20 == 0:
9     save_path = os.path.join(log_dir, f"{epoch}{ext}")
10
11     # Algorithm 2: Sampling
12     reverse_diffusion(model, sd, timesteps=TrainingConfig.TIMESTEPS,
13                       num_images=32, generate_video=generate_video,
14     save_path=save_path,
15                       img_shape=TrainingConfig.IMG_SHAPE, device=BaseConfig.DEVICE,
16     nrow=4,
17     )
18
19     # clear_output()
20     checkpoint_dict = {
21         "opt": optimizer.state_dict(),
22         "scaler": scaler.state_dict(),
23         "model": model.state_dict()
24     }
25     torch.save(checkpoint_dict, os.path.join(checkpoint_dir, "ckpt.pt"))
26     del checkpoint_dict
```

If all goes well, the training procedure should start and print the training logs similar to:



Generating Images Using DDPMs

You can let the training complete for 800 epochs or interrupt in between if you are satisfied with the samples generated at every 20 epochs.

To perform the *inference*, we simply have to reload the saved model, and you can use the same or a different logging

directory to save the results. You can re-initialize the `SimpleDiffusion` class as well, but it's not necessary.

```

1  # Reloading model from saved checkpoint
2  model = UNet(
3      input_channels      = TrainingConfig.IMG_SHAPE[0],
4      output_channels     = TrainingConfig.IMG_SHAPE[0],
5      base_channels       = ModelConfig.BASE_CH,
6      base_channels_multiples = ModelConfig.BASE_CH_MULT,
7      apply_attention     = ModelConfig.APPLY_ATTENTION,
8      dropout_rate        = ModelConfig.DROPOUT_RATE,
9      time_multiple       = ModelConfig.TIME_EMB_MULT,
10 )
11 model.load_state_dict(torch.load(os.path.join(checkpoint_dir, "ckpt.tar"),
12 map_location='cpu')['model'])
13
14 model.to(BaseConfig.DEVICE)
15
16 sd = SimpleDiffusion(
17     num_diffusion_timesteps = TrainingConfig.TIMESTEPS,
18     img_shape               = TrainingConfig.IMG_SHAPE,
19     device                  = BaseConfig.DEVICE,
20 )
21
22 log_dir = "inference_results"

```

The inference code is simply a call to the `reverse_diffusion(...)` function using the trained model.

```

1  generate_video = False # Set it to True for generating video of the entire reverse
2  diffusion proces or False to for saving only the final generated image.
3
4  ext = ".mp4" if generate_video else ".png"
5  filename = f"{datetime.now().strftime('%Y%m%d-%H%M%S')}{ext}"
6
7  save_path = os.path.join(log_dir, filename)
8
9  reverse_diffusion(
10     model,
11     sd,
12     num_images=256,
13     generate_video=generate_video,
14     save_path=save_path,
15     timesteps=1000,
16     img_shape=TrainingConfig.IMG_SHAPE,
17     device=BaseConfig.DEVICE,
18     nrow=32,
19 )
20 print(save_path)

```

Some of the results we got:

Example 1 of unconditional flower image generation using the trained model.

Example 2 of unconditional flower image generation using the trained model.



In conclusion, diffusion models represent a rapidly growing field with a wealth of exciting possibilities for the future. As research in this area continues to evolve, we can expect even more advanced techniques and applications to emerge. I encourage readers to share their thoughts and questions about this topic and to engage in conversations about the future of diffusion models.

To summarise this article💡💡, we covered a comprehensive list of related topics.

1. We began by providing an intuitive answer to the fundamental question of why we need generative models.
2. Then we continued the discussion to explain diffusion-based generative models from a logical and theoretical perspective.
3. After building the theoretical base, we introduced all the necessary mathematical equations derived for DDPMs one by one while also maintaining the flow so that it's easy to grasp.
4. Finally, we concluded by explaining all the different pieces of code required for training DDPMs from scratch and performing inference. We also demonstrated the results we got from our experiments.

References

1. [What are Diffusion Models?](#)
2. [DDPMs from scratch](#)
3. [Diffusion Models | Paper Explanation | Math Explained](#)
4. Paper – [Deep Unsupervised Learning using Nonequilibrium Thermodynamics](#)
5. Paper – [Denoising Diffusion Probabilistic Models](#)
6. Paper – [Improved Denoising Diffusion Probabilistic Models](#)
7. Paper – [A Survey on Generative Diffusion Model](#)
8. [An introduction to Diffusion Probabilistic Models – Ayan Das](#)
9. [Denoising diffusion probabilistic models – Param Hanji](#)

We would love to hear from you. Please feel free to ask questions in the comment section; we are more than happy to converse with you.

💡💡Happy learning!

Subscribe & Download Code

If you liked this article and would like to download code (C++ and Python) and example images used in this post, please [click here](#). Alternately, sign up to receive a free [Computer Vision Resource](#) Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Download Example Code



- Tags:

ddpm explained

ddpm github

ddpm implementation

ddpm pytorch

ddpm sampling

ddpm tutorial

ddpms
- explained

denoising diffusion models

denoising diffusion probabilistic models

denoising diffusion probabilistic models explained
- denoising diffusion probabilistic models github

denoising diffusion probabilistic models pytorch

denoising diffusion probabilistic models
- review

denoising diffusion probabilistic models tutorial

OpenCV BootCamp

Learn Computer Vision and AI Using OpenCV

Join FREE OpenCV Course

Subscribe To My Newsletter

TensorFlow 2.0 Bootcamp for Beginners (TFBC)

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
import tensorflow as tf
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.models import Model
```

Join FREE TensorFlow Course

BIGVISION

Expert Consulting Services In AI, Computer Vision & Deep Learning

bigvision.ai

contact@bigvision.ai

Popular

Related

Recent

Top 5 AI papers of

September 2023
October 17, 2023



Advanced Driver Assistance Systems (ADAS): Empowering Drivers
October 10, 2023



Top 5 AI papers of August 2023
September 12, 2023

Subscribe Now



Disclaimer

All views expressed on this site are my own and do not represent the opinions of OpenCV.org or any entity whatsoever with which I have been, am now, or will be affiliated.



Getting Started

- Installation
- PyTorch
- Getting Started with OpenCV
- Keras & Tensorflow

Course

- Opencv Courses
- CV4Faces (Old)

Information

[Privacy Policy](#)
[Terms and Conditions](#)

About LearnOpenCV

In 2007, right after finishing my Ph.D., I co-founded TAAZ Inc. with my advisor Dr. David Kriegman and Kevin Barnes. The scalability, and robustness of our computer vision and machine learning algorithms have been put to rigorous test by more than 100M users who have tried our products.

[Read More](#)