Diffusion Models — DDPMs, DDIMs, and Classifier Free Guidance

betterprogramming.pub/diffusion-models-ddpms-ddims-and-classifier-free-guidance-e07b297b2869

Gabriel Mongaras March 13, 2023

Learn about the evolution of diffusion models from DDPMs to Classifier Free guidance

Photo by on

The big models in the news are text-to-image (TTI) models like DALL-E and text-generation models like GPT-3. Image generation models started with GANs, but recently diffusion models have started showing amazing results over GANs and are now used in every TTI model you hear about, like Stable Diffusion. In this article, I want to talk about where diffusion models started and some improvements that led them to where they are today.

To go along with this article, I coded everything I will talk about in this repo if you are interested.

A Little History

Interestingly, diffusion models have been around for a while. The earliest paper I could find referencing such a model is from 2015, which can be found here. The paper showed promising results, but since GANs were starting to get big at the time, I don't think people were looking at other generative models.

Recently, GANs have been reaching their limit. As good as they are, they are very unstable and tend to run into mode collapse, where they only generate a small part of the true data distribution. StyleGAN 3 is very impressive, but I feel it marks the ending of how far GANs can go. StyleGAN 3 didn't try to make the GAN part of StyleGAN better. Rather, it improved the generator model to move it to a continuous space.

I don't think GANs will be able to move forward much more, which is why diffusion is the next generation of generative models instead of improving GANs.

Diffusion models didn't start to become noticed until the 2020s when the DDPM (Denoising Diffusion Probabilistic Models) paper was released. It showed that diffusion models can achieve very good performance in image generation. It wasn't until the paper Diffusion Models Beat GANs on Image Synthesis which showed that diffusion models can do better than GANs with class coverage, image quality, and stability. Today, all the cool generative models like DALL-E and Stable Diffusion use diffusion models.

I want to talk about the more classic approaches to diffusion models and how they started emerging as the best generative models, which you can see today.

DDPMs (Denoising Diffusion Probabilistic Models)

As much as I would like to go into the math of DDPMs, my brain is too smooth. Fortunately, others go into great detail about them. AI Summer has a really good article on the math here which I highly suggest looking at. The Understanding Diffusion Models: A Unified Perspective takes a deeper look and derives all the formulas. I'll detail the main idea about DDPMs at a higher level here.

DDPMs are where the current diffusion model began. In this paper, the authors propose using a Markov Chain model, progressively adding noise to an image.

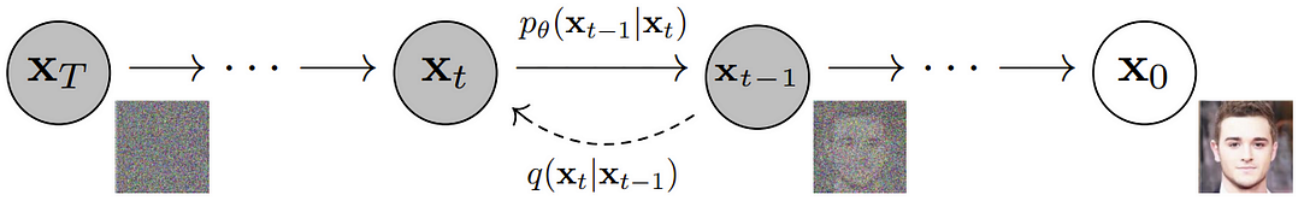


Figure 2: The directed graphical model considered in this work.

Diffusion Generation

A function, `q(x_t | x_t-1)`, is used to add noise to an image one step at a time. At each step, more noise is added to the image until the image is essentially pure Gaussian noise at time `T`.

The goal is to teach a model to reverse the process so that we can generate images given the noise from a Gaussian Distribution. This way, we can generate images from the noise, just like GANs.

The forward process

Going from time `t=0` to `time t=T` by progressively adding more noise to the input image is called the “forward process” (even though it’s going backward in the image). The function `q` defines the forward process and has a closed-form solution that allows us to directly model the forward process given x_0 (The image, `x`, at diffusion timestep `0`, the original image). The function is defined below:

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{4}$$

Forward process function

The function `q` uses a Normal (Gaussian) Distribution to model the noising process. There is a problem with this approach, though. The distribution must be sampled `t` times to get an image at time `t` from time `t=0`. We could store all images for all values of `t` in memory or load them from disk as needed, but normal values of `T` are greater than or equal to 1,000, so we would have to store 1,000 variations of each image to train the model, which is not desired.

To solve these issues, the authors model the forward process as follows:

$$q(x_t|x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, \sqrt{1-\bar{\alpha}_t}I) = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$$
$$\epsilon \sim \mathcal{N}(0,1)$$

Modeling the forward process using the reparameterization trick

This method uses the reparameterization trick, which allows us to model the distribution, but in a way where we can skip directly from timestep `0` to `t` according to α_t`_bar`. In a way, the formula above is weighing x_0 (the original image) and epsilon (sampled noise from a Normal Distribution) according to α_t`_bar` (the noise scheduler).

α_t`_bar` is calculated based on a noise scheduler. The lower this value is, the more noise is added. The authors define α_t as `1-`β_t and α_t`_bar` as a cumulative product of alpha values from time `0` to time `t`.

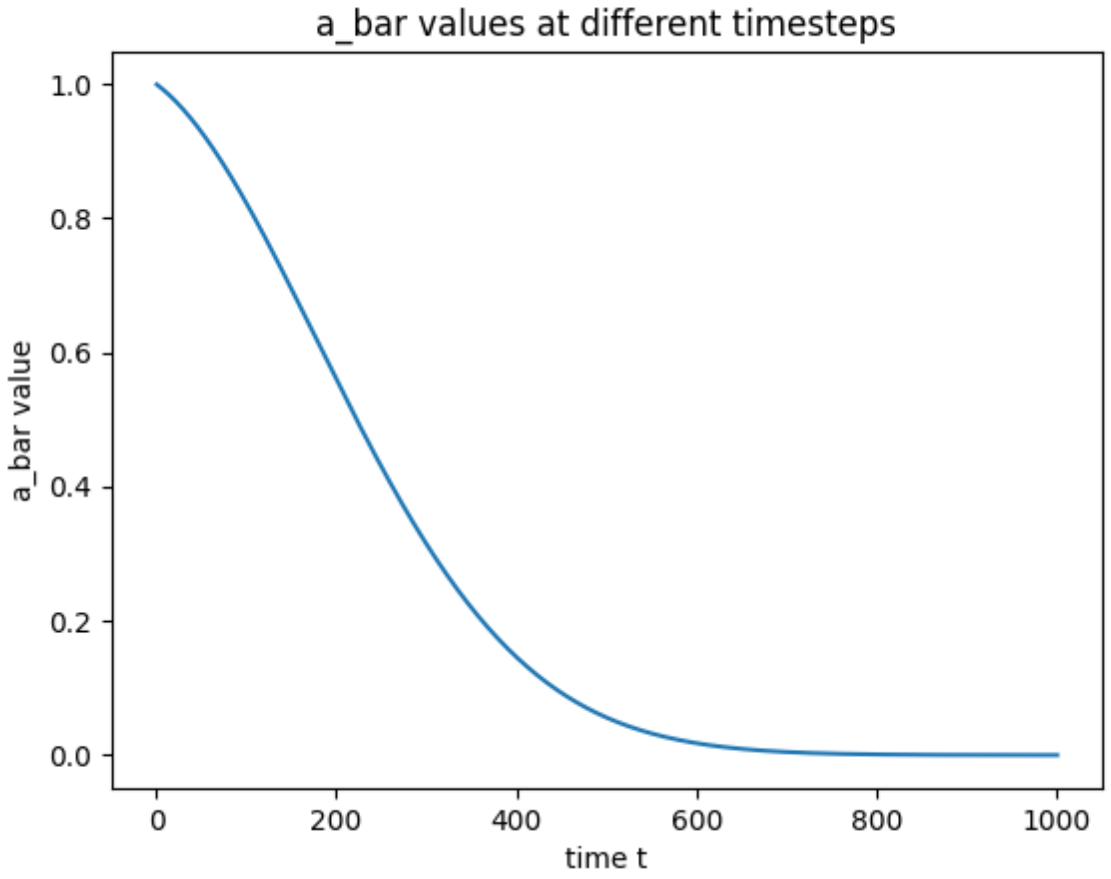
β_t is our noise scheduler. The authors of the DDPM paper use a linear scheduler between values of `10^-4` and `0.02`. At time `t=0`, the value of β_t will be `10^-4`. At time `T`, β_t will be `0.02`. These value kind of act like percentages for the amount of noise added at time `t` relative to time `t-1`.

$$\alpha_t = 1 - \beta_t$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} a_s$$

Normal and cumulative noise schedulers

Note that the amount of noise added at time `t` is not just a rate between `10^-4` and `0.02`, rather we are using α_t`_bar`. α_t is large at small values of `t` and small at large values of `t`. Additionally, α_t`_bar` is a product of all α_t values from `0` to `t`. So the noise added at time `t` is the product of all α_t values, meaning the amount of noise at each timestep increases exponentially, and the percent of the original image decreases exponentially. Below is a curve showing the values of α_t`_bar` at all timesteps from `t=0` to `t=T=1000`.



Noise level according to a_bar over time

To summarize the forward process, we can use the closed-form solution of the q function to add noise to an image from x_0 (the original image) to x_t (the image at diffusion step t) in a single operation.

The backward process

The backward process models the reverse of $q(x_t \mid x_{t-1})$ and is given by the function $p(x_{t-1} \mid x_t)$. Unfortunately, we cannot model this process directly as there are too many possibilities of image x_{t-1} when we want to get image x_t.

Neural networks to the rescue! Instead, we can estimate the reverse process using a neural network. So, the function becomes $p_\theta(x_t \mid x_{t-1}, t)$. The θ denotes the parameters of the neural network we are optimizing to estimate the function p.

Intuitively, since we use a normal distribution to model the forward process, we can also use a normal distribution to model the reverse process. So, we can have the model predict the mean and variance of a normal distribution where μ_θ is the predicted mean of the distribution and Σ_θ is the predicted variance or the distribution. Note that this normal distribution is predicted for all pixels; it's not one normal distribution for the entire image.

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

Reverse diffusion process

"We also see that learning reverse process variances (by incorporating a parameterized diagonal $\Sigma_\theta(x_t)$ into the variational bound) leads to unstable training and poorer sample quality compared to fixed variances." (4.2) The DDPM authors find that it's much easier to keep the variance, Σ_θ, constant (which we'll talk more about in the next section), and they set $\Sigma_\theta = \beta_t$ since β_t is the noise variance at timestep t.

Since we know the normal distribution that got us to step t using the function $q(x_t \mid x_{t-1})$, and we have a prediction for that distribution $p(x_{t-1} \mid x_t)$, we can use the KL divergence loss between the two distributions to optimize the model.

The authors note that since they keep the variance constant, they only have to predict the mean of the distribution. Better yet, we can just predict the noise, ε, that was sampled from the normal distribution and added to the image through the reparameterization trick. The authors found that predicting the noise was more stable. Since we just have to predict the noise added, we can use the MSE loss between the predicted noise and the actual noise added to the image.

$$\textbf{Model:} \quad \epsilon_\theta \left(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon \right)$$

$$\textbf{Loss:} \quad MSE \left[\epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon \right) \right]$$

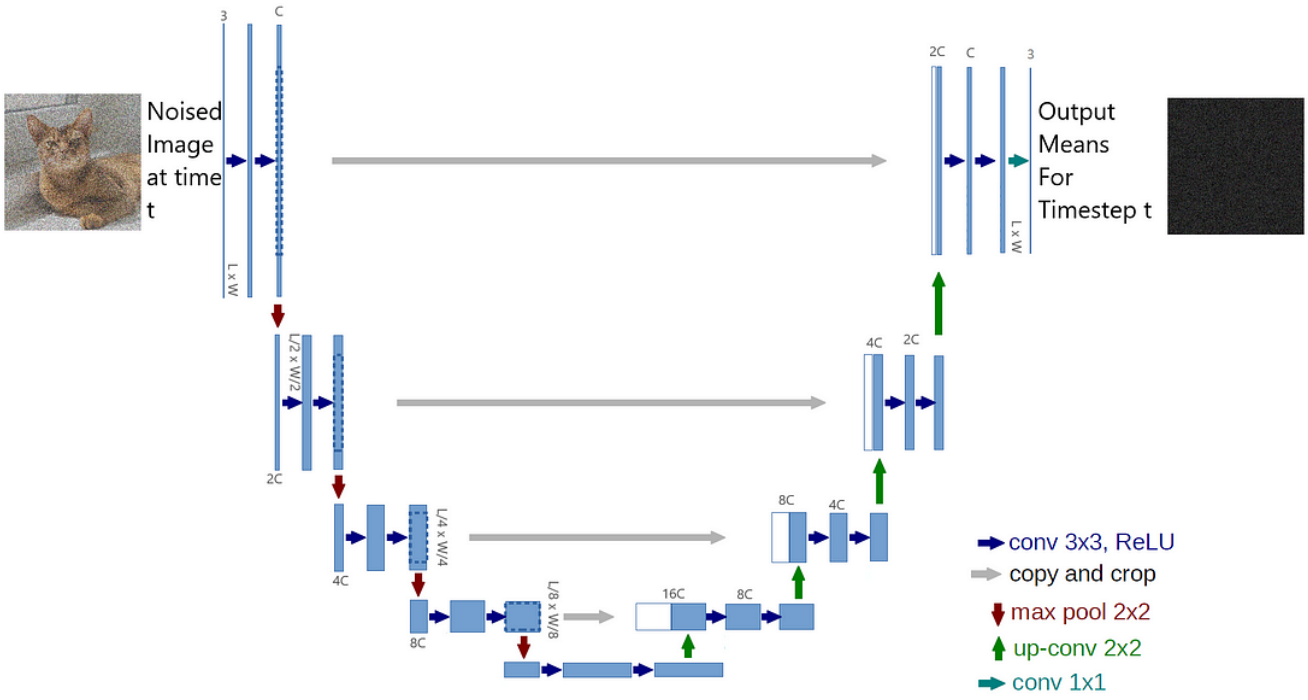
Model and MSE Loss

One may think it may be hard for a model to learn the noise since noise is random and a neural network is usually deterministic. But, if we give the model the noisy image at time `t` and the timestep `t`, then the model can find a way to extract the noise from the noisy image, which can be used to reverse the noising process.

Interestingly the authors note that “In particular, our diffusion process setup in Section 4 causes the simplified objective to down-weight loss terms corresponding to small `t`. These terms train the network to denoise data with very small amounts of noise, so it is beneficial to down-weight them so that the network can focus on more difficult denoising tasks at larger `t` terms.” (page 5, part 3.4)

So the authors construct the loss so that the model is more biased toward learning higher values of `t` which require it to denoise much more noise than lower values of `t`. The idea is that higher values of `t` construct high-level features of the object and lower levels of `t` construct more fine-grained features in the image. It’s more important to get the main shape of the object right than to make the object have some sort of texture.

The reverse process is typically modeled using a U-net, as shown below:

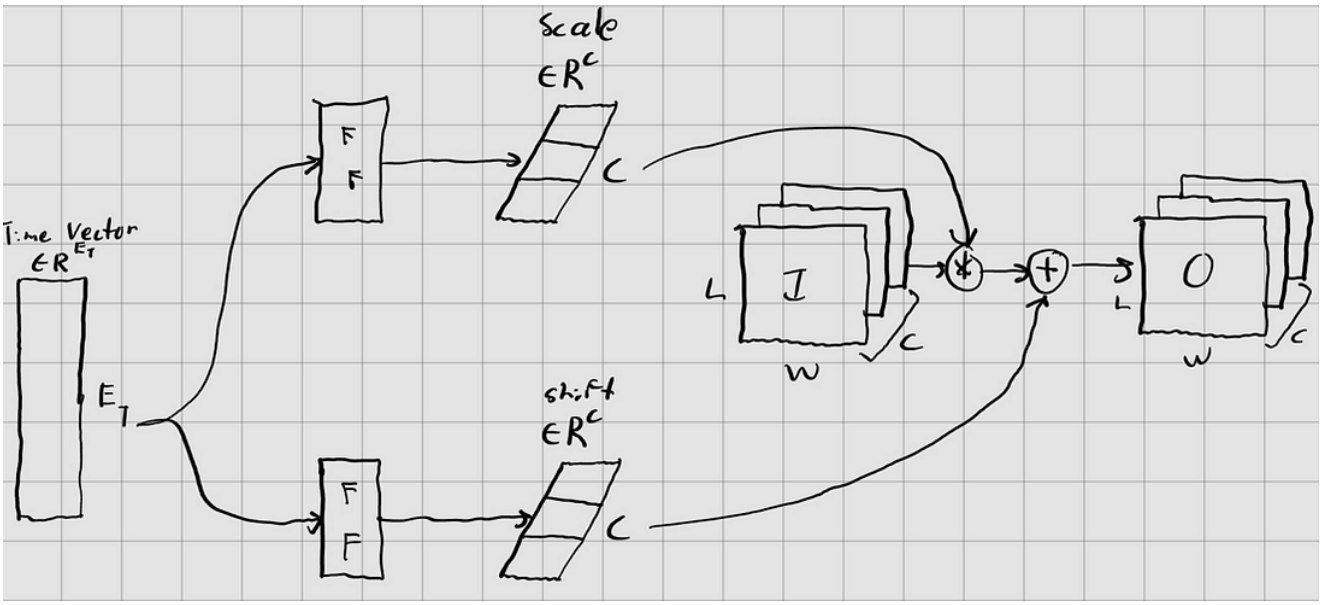


U-Net for extracting noise from a given image

The input is the image at time, `t`, and the output is the noise within that image. Additionally, at each layer in the network, we add time information to help the model know where it's at in the diffusion process. So the input is actually the input image at time, `t`, and the timestep itself, `t`.

$p(x_t\text{-}1 \mid x_t, t)$

To encode the timestep in a usable form, we can use the "Attention is All You Need" positional encodings. Instead of encoding the location in the sequence, we can treat the embeddings as timestep vectors where a vector represents a timestep.



Adding time information to the model

You can project the time vector to the number of channels and create two vectors, one to shift the intermediate image encodings and one to scale the intermediate image encodings.

The paper "Diffusion Models Beat GANs on Image Synthesis" proposes this method of adding time information and is called "Adaptive Group Normalization." Specifically, it adds the information after each `GroupNorm` layer:

$$\text{AdaGN}(h, y) = y_s \, \text{GroupNorm}(h) + y_b$$

AdaGN formulation

`y_s` is the scale vector, and `y_b` is the shift vector. In the paper, the authors make `y_b` class information instead of time information to give the model knowledge of what class we want it to generate. To create the class vector, `y_b`, one can one-hot encode the class and feed the one-hot vector through a feed-forward layer. The idea of class information addition will become very important later with classifier-free guidance.

The best part about this method of adding time information is the dependence only on the image channels and the independence on the spatial image size (L/W). Since the number of channels represents the number of image features, this value will always be static. The length and width, however, can be changed, and due to the nature of convolutions, the algorithm will still work. Adding time only on the channel dimension retains this feature. So, different-sized images can be generated instead of being restricted to a single-sized image.

Training loop

With the forward and backward processes defined, we can train the model and generate images by the following training/denoising loops:

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** $\mathbf{x}_0$

DDPM training loop

The left loop trains a model as follows:

1. Loop over epochs
2. Sample a batch of images from your dataset
3. Sample a value of uniformly for each image in the batch
4. Sample noise from a Gaussian Distribution with `0` mean and unit variance.
5. Each image is noised to that timestep `t`, and the model predicts the noise in that image.
6. Use MSE loss between the sampled noise and predicted noise for each image

Note that we don't have to model the entire diffusion process as a single process, but rather we can model each individual timestep individually. Doing this will speed up training and will likely lead to a more stable training setting. If we sample the value of `t` uniformly for each training image, the model should be able to learn how to model all values of t while learning how to model the real image distribution.

The right loop generates/samples images from noise as follows:

1. Sample noise from a Gaussian Distribution with `0` mean and unit variance. This represents our noisy image at time `T`.
2. Loop from time `t=T` to time `t=1`.
3. Sample new noise from a Gaussian Distribution, which will be used to move the image to the previous timestep, `t-1`.
4. Using our trained model, `ε_θ`, generate a prediction for the noise at the current timestep. Remove the noise and move the image to the previous timestep `t-1`.
5. Repeat from `2` until `t=1`.

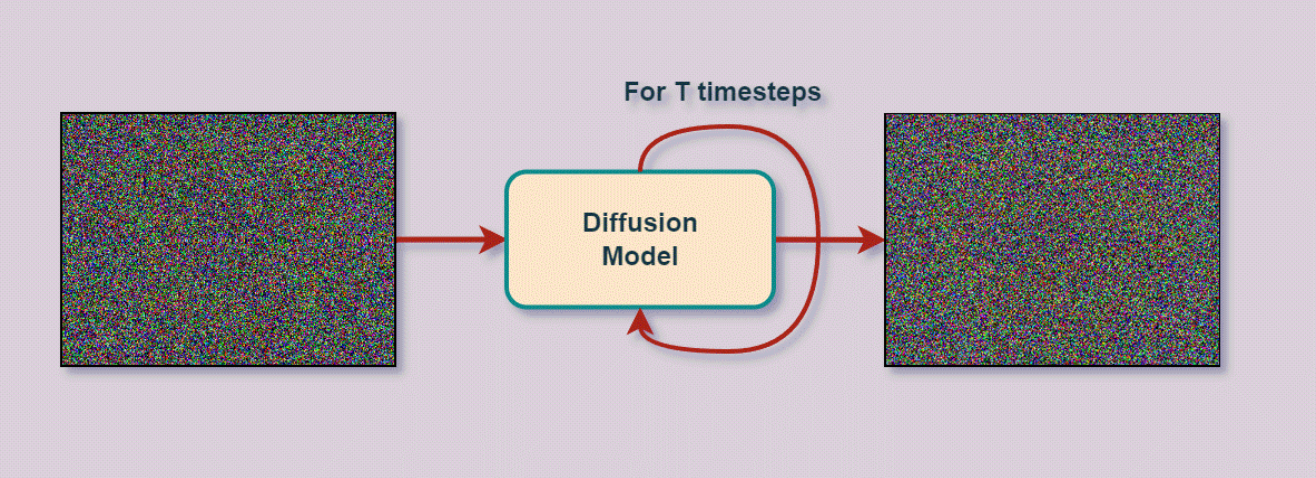
When all `T` iterations are done, a new image will be generated at timestep `0`.

Part 4 of algorithm two may be a little confusing, and if you want to learn more about how it's derived, the paper Understanding Diffusion Models: A Unified Perspective derives it at EQ 84.

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

Sampling Algorithm From Line 4

Intuitively, the predicted noise theoretically removes all noise from the image at timestep `t`, making the image x_0 when the noise is removed. This is what the first term does in the sampling algorithm. In reality, that was just a prediction, and since the image is all noise, the output won't resemble any sort of image. So, we must add more noise back to the image, but at timestep `t-1` and do this for all timesteps. The noise is re-added in term 2.



Improving DDPMs

The main issue DDPMs had is the log-likelihood score (meaning the model may be able to generate high-quality images but doesn't fit the dataset very well in terms of the distribution of the real image data), which the authors of the Improved DDPM paper wanted to solve. The improving DDPMs paper had a couple of methods to improve the log-likelihood:

1. Learn `Σ_θ(x_t)`, the variance of the predicted normal distribution instead of keeping it fixed at β_t.
2. Change the learning rate scheduler defined as a linear β_t interpolation between `10^-4` and `0.02` to a `cosine α_t_bar` interpolation.

The authors had a few other changes, but I just the two main improvements still used today are shown above. The authors also increase the number of timesteps to `T=4000` from `T=1000`. Increasing the number of steps from 1,000 to 4,000 may increase FID scores and log-likelihood scores a little, but waiting four times longer to generate an image gets really annoying.

Learning Σ_θ

One of the main improvements is the prediction of the variance, which the original DDPM paper decided not to do because “We also see that learning reverse process variances (by incorporating a parameterized diagonal $\Sigma_\theta(x_t)$ into the variational bound) leads to unstable training and poorer sample quality compared to fixed variances.” (page 6, DDPM)

The improved DDPM paper decides to learn the variances to help improve the model's log-likelihood. However, they run into an issue. They find that the instability of the variance predictions comes from the average size of the variances and find the variances are very small. Neural networks have issues predicting very small values as it may lead to vanishing gradients. So, they predict v to interpolate between the upper (β_t) and lower (β_t~) bounds in the log domain, which appears to yield stable predictions for the variances:

$$\Sigma_\theta(x_t, t) = \exp(v \log \beta_t + (1 - v) \log \tilde{\beta}_t) \qquad (15)$$

Variance parameterization

β_t is just the normal old variance value in the forward process, whereas β_t~ is a scaled form of β_t based on α_t_bar and α_t_bar_1 (a bar sub t-1).

The original DDPM paper states that “The first choice (β_t) is optimal for $x_0 \sim N(0,)$, and the second (β_t~) is optimal for x_0 deterministically set to one point.” (page 3, DDPM)

$$\tilde{\beta}_t := \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \beta_t$$

Upper bound variance formulation

Quick Side Note — Where Is This Derivation Coming From?

The quote above didn’t make much sense, so I tried to understand it from the paper preceding the DDPM paper. The original paper has the following derivation for the upper and lower bounds:

Upper Bound β_t Lower Bound $\tilde{\beta}_t$

$$H_q\left(\mathbf{X}^{(t)}|\mathbf{X}^{(t-1)}\right) \geq H_q\left(\mathbf{X}^{(t-1)}|\mathbf{X}^{(t)}\right) \geq H_q\left(\mathbf{X}^{(t)}|\mathbf{X}^{(t-1)}\right) + H_q\left(\mathbf{X}^{(t-1)}|\mathbf{X}^{(0)}\right) - H_q\left(\mathbf{X}^{(t)}|\mathbf{X}^{(0)}\right)$$

Upper and lower variance bounds

Note that this is in terms of the forward process, not the backward process. The H functions are just the Cross-Entropy function of the system.

$$H_q(X^{(t-1)}|X^{(t)}) = \text{System Entropy} = -\sum_i X_i^{(t-1)} log(X_i^{(t)})$$

Entropy of a system formulation

I am going to try to explain how I understand it.

When going from step `t-1` to step `t`, the amount of information is going to decrease, and the entropy is going to increase because the number of noise increases as per the definition of the diffusion process. Say we have a full black image for x_0 (represented by all `0`s in a tensor) and all Gaussian noise for x_t. Then when we go from step `t-1` to step `t`, the image at step `t` is essentially a Gaussian distribution but scaled since the image we are adding to it is all `0`s.

Since we are increasing how much the Gaussian distribution appears in the image from step `t-1` to step `t`, the Gaussian distribution becomes more abundant. The added Gaussian noise between steps has nothing to make the image at step `t` non-Gaussian because the original image is all zeros.

So, the difference between step `t-1` and step `t` is essentially a Gaussian, and since it's a Gaussian, you are adding noise between steps. You maximize the entropy between steps. This added noise creates the upper bound between the two distributions at `t-1` and `t` since Gaussian noise maximizes the difference between the distributions.

For any other image that's not all `0`s, the upper bound will be slightly smaller since pure Gaussian noise isn't added directly to the current timestep.

"A lower bound on the entropy difference can be established by observing that additional steps in a Markov chain do not increase the information available about the initial state in the chain, and thus do not decrease the conditional entropy of the initial state." (page 12 original paper)

The lower bound comes from a "corrected" version of the upper bound. Notice how the lower bound includes the upper bound and two extra terms:

1. The first term is the original upper bound, which is the difference between the distribution at `t-1` and the distribution at `t`.
2. The second term is the difference between x_0, the original distribution, and the previous distribution at `t-1`.
3. The third term is the difference between x_0, the original distribution, and the new distribution at `t`.

Adding 1. and 2. gives you the total difference between x_0 and x_{t-1} and the difference between x_{t-1} and x_t. Then we remove the actual difference between x_0 and x_t to give us the final result, the difference between x_{t-1} and x_t. The idea is that any "information" the diffusion process "adds" to the current distribution being generated could be added somewhere from time t to time $t-1$, which may alter the real KL divergence value according to the entire diffusion process. We want to remove this "information" since any image we generate has no more "information" than the original image at x_0. That's what the lower bound represents.

So, the upper bound is the immediate difference between the distribution x_{t-1} and the distribution x_t, while the lower bound is corrected so that "information" that could have come from noise isn't added to this difference.

The difference between distributions is a great way to model the variance because the variance at any step should model how much the distribution changes between timesteps. That's exactly what the upper and lower bounds model is.

These bounds are very useful to have the model estimate the variance at any timestep in the diffusion process. The improved DDPM paper notes that β_t and $\beta_t\sim$ represent two extremes on the variance. One when the original image, x_0, is pure Gaussian, and the other when the original image, x_0, is a single value. Any input image x_0 will fall either between a pure Gaussian or a single-valued image, making it intuitive to interpolate between these two extremes.

Optimizing the variance

Remember that the loss function is the MSE between the predicted and true noise in the image. While the noise can directly model the mean of the predicted distribution, there needs to be another way to model the variance of the output distribution. So, we have to change the loss to incorporate the variances. The loss is changed as follows.

$$L_{\text{hybrid}} = L_{\text{simple}} + \lambda L_{\text{vlb}} \qquad (16)$$

Combined loss for variance and mean

`L_simple` is the original objective, the MSE between the predicted and real noise sample.

Lambda is a weighting term that the authors set to `0.001`. This weighs the simple loss much higher than the VLB loss.

`L_vlb` is the variational lower bound objective, which the original DDPM paper formalizes as the KL divergence between the predicted Gaussian distribution and the actual Gaussian distribution of each pixel in the image at timestep *t*:

$$L_{t-1} := D_{KL}(q(x_{t-1}|x_t, x_0) \;||\; p_\theta(x_{t-1}|x_t)) \qquad (6)$$

KL divergence loss between predicted and actual distributions for each pixel

The KL divergence loss minimizes the difference between the two distributions. In our case, the KL divergence is between the predicted Gaussian distribution for each pixel in the image vs the actual Gaussian distribution for each pixel in the image. This loss is pretty intuitive since we want the loss to model how far the predicted pixel distributions are and real pixel distributions are at timestep `t`.

Our Gaussian distributions have the following formula:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

$$\sigma = \sqrt{\beta_t} \quad \text{or} \quad \sigma = \sqrt{\tilde{\beta}_t} \quad \text{or} \quad \sigma = \sqrt{\Sigma_\theta(x_t, t)}$$

$$\mu = \tilde{\mu}_t(x_t, x_0) \quad \text{or} \quad \mu = \mu_\theta(x_t, t)$$

Gaussian formulation DDPM context

For both the forward and backward process, the variance can be either the lower or upper bound variance, that is β_t or β_t~. But since we want to model the variance, the variance for the backward process becomes the parameterized variance `Σ_θ`. A known function parameterizes the mean for the forward process and the neural network for the backward process.

Since the Gaussian formula has both mean and variance, the KL Divergence loss optimizes both the mean and variance at the same time. The authors put a stop gradient for the mean statistic in this loss function since `L_simple` is already optimizing the mean. This way, the mean isn't being optimized by two loss functions representing the same function we want to minimize. So, this loss only optimizes the variance.

Model	ImageNet	CIFAR
Glow (Kingma & Dhariwal, 2018)	3.81	3.35
Flow++ (Ho et al., 2019)	3.69	3.08
PixelCNN (van den Oord et al., 2016c)	3.57	3.14
SPN (Menick & Kalchbrenner, 2018)	3.52	-
NVAE (Vahdat & Kautz, 2020)	-	2.91
Very Deep VAE (Child, 2020)	3.52	2.87
PixelSNAIL (Chen et al., 2018)	3.52	2.85
Image Transformer (Parmar et al., 2018)	3.48	2.90
Sparse Transformer (Child et al., 2019)	3.44	**2.80**
Routing Transformer (Roy et al., 2020)	**3.43**	-
DDPM (Ho et al., 2020)	3.77	3.70
DDPM (cont flow) (Song et al., 2020b)	-	2.99
Improved DDPM (ours)	**3.53**	**2.94**

Presented log-likelihood scores in the Improved DDPM Paper

The table above shows the presented log-likelihood scores (lower is better) for the original DDPM model, improved DDPM model, and others. The improved DDPM model does better than the original DDPM but still does not outperform SOTA (state-of-the-art) models at the time.

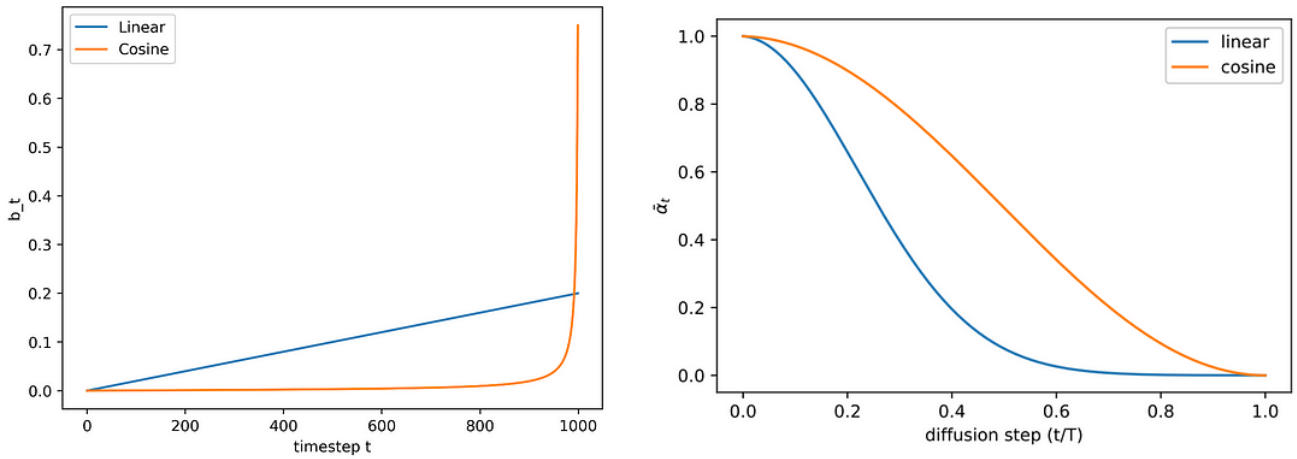
Interestingly, in my implementation, I found that the prediction of these variances produced values almost identical to the β_t value (or β_t~ value depending on how it was trained), but the model performance did look like it was doing better. I suspect this performance boost is because the model has a better understanding of the variance as it has to learn it. The variance is built into the model as opposed to it being passively added to the model's output.

New Learning Rate Scheduler

The second improvement the improved DDPM paper introduced is using a different learning rate scheduler. The authors note that "the end of the forward noising process is too noisy, and so doesn't contribute very much to sample quality." (Page 4, Improved DDPMs)

Linear vs cosine noise strategies

The main problem with the linear scheduler is for small images. The image is not far from pure Gaussian noise too early in the diffusing process, which may make it hard for the model to learn the reverse process. Essentially, noise is being added too fast. The cosine scheduler adds noise slower to retain image information for later timesteps.



Linear vs cosine cumulative and relative noise additions

As you can see, the linear scheduler on the right converges to an `α_t_bar` value of `0` early in the diffusing process, meaning the image will be nearly pure gaussian noise early in the diffusing process. The cosine scheduler, on the other hand, converges to `0` much later.

Additionally, the paper introduces a way to speed up the process, but the next section goes into a paper all about speeding up the diffusing process.

DDIMs (Denoising Diffusion Implicit Models)

One problem with the DDPM process is the speed of generating an image after training. Sure, we may be able to produce amazing-looking images, but it takes 1,000 model passes to generate a single image. Passing an image through the model 1,000 times may take a few seconds on a GPU but much longer on a CPU. We need a way to speed up the generation process.

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.

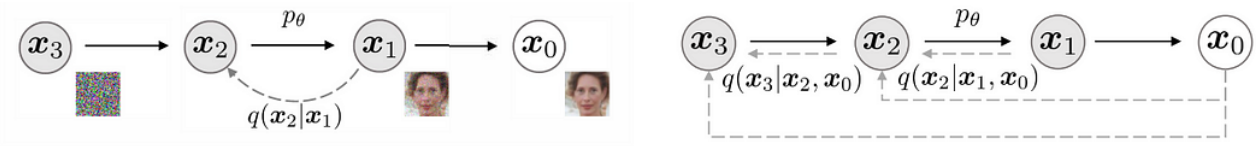


Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

DDPM vs DDIM models

The left figure is the original DDPM paper which requires all past denoising steps from time `T` to time `t-1` to obtain the next denoised image at time `t`. DDPMs are modeled as Markov Chains, meaning an image at time `t` cannot be generated until the entire chain before `t` has been generated.

The DDIM paper proposes a method to make the process non-markovian (in the right figure), allowing you to skip steps in the denoising process, not requiring all past states to be visited before the current state. The best part about DDIMs is they can be applied after training a model, so DDPM models can easily be converted into a DDIM without retraining a new model.

First, the reverse diffusion process for a single step is redefined:

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2} \cdot \epsilon_\theta^{(t)} + \sigma_t \epsilon_t$$

Redefinition of the reverse process

Note: The DDIM paper uses alphas without the bar, but the alpha values in the paper are alpha bar (cumulative alpha) values used in the DDPM paper. It's a little confusing, so I will replace their alphas with alpha bars to keep the notation consistent.

First, the reformalization is equivalent to the formalization in the DDPM paper, but only when the variance is equal to β_t~.

$$\sigma_t = \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}}\sqrt{1-\frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}} \quad \leftarrow \text{We know that: } \beta_t = 1-\alpha_t = 1-\frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}$$

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

$$\sigma_t = \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t} \quad \leftarrow \tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

$$\sigma_t = \sqrt{\tilde{\beta}_t}$$

Variance is just β_t~

The authors don't explicitly state that their formulation of sigma is just β_t~, but with a little algebra, you can find that's the case.

When `σ = 0`, we get a DDIM:

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1-\bar{\alpha}_{t-1}} \cdot \epsilon_\theta^{(t)}$$

DDIM Denoising Formula

Notice how there's no added noise to the data. This is the trick of the DDIM. When `σ = 0`, the denoising process becomes completely deterministic, and the only noise is the original noise at `x₀` because no new noise is added during the denoising process.

Since there is no noise in the reverse process, the process is deterministic, and we no longer have to use a Markov Chain since Markov Chains are used for probabilistic processes. We can use a Non-Markovian process, which allows us to skip steps.

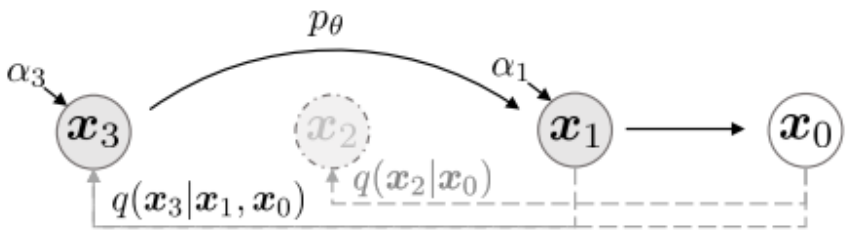


Figure 2: Graphical model for accelerated generation, where $\tau = [1, 3]$.

Non-Markovian reverse and forward process

In the diagram above, we skip from step `x_3` to `x_1`, skipping `x_2`. The authors model the new diffusion process as a subsequence, `τ`, which is a subset of the original diffusion sequence. For example, I could sample every other diffusion step in the diffusion process to get a subsequence of `τ = [0, 2, 4, …, T-2, T]`.

Finally, the authors decide the model the diffusion model variance as an interpolation between DDIMs and DDPMs using the following formula:

$$\sigma_t = \eta \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}} \sqrt{1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}} = \eta \sqrt{\tilde{\beta}_t}$$

DDIM variance

The diffusion model is a DDIM when `η=0` as there is no noise and an original DDPM when `η=1`. Any `η` between `0` and `1` is an interpolation between a DDIM and DDPM.

DDIMs perform much better than DDPMs when the number of steps taken is less than the original `T` steps. The chart below shows DDPM and DDIM FID scores (which score diversity and image quality) on `η` interpolations from 0 to 1 and on 10, 20, 50, 100, and 1000 generation steps. Note that the original model was trained on `T=1000` steps.

Table 1: CIFAR10 and CelebA image generation measured in FID. $\eta = 1.0$ and $\hat{\sigma}$ are cases of DDPM (although Ho et al. (2020) only considered $T = 1000$ steps, and $S < T$ can be seen as simulating DDPMs trained with S steps), and $\eta = 0.0$ indicates DDIM.

		CIFAR10 (32×32)					CelebA (64×64)				
	S	10	20	50	100	1000	10	20	50	100	1000
η	0.0	**13.36**	**6.84**	**4.67**	**4.16**	4.04	**17.33**	**13.73**	**9.17**	**6.53**	3.51
	0.2	14.04	7.11	4.77	4.25	4.09	17.66	14.11	9.51	6.79	3.64
	0.5	16.66	8.35	5.25	4.46	4.29	19.86	16.06	11.01	8.09	4.28
	1.0	41.07	18.36	8.01	5.78	4.73	33.12	26.03	18.48	13.93	5.98
$\hat{\sigma}$		367.43	133.37	32.72	9.99	**3.17**	299.71	183.83	71.71	45.20	**3.26**

DDIM results with different Η values and different step sizes on different data sets.

The lower the FID score, the better. Although the DDPM performs the best at the original 1,000 steps, the DDIM closely follows when generating images with much fewer generation steps.

You essentially have a tradeoff between image quality and time to generate when using a DDIM, which the original DDPM did not offer. Now we can generate high-quality images with much fewer steps!

Classifier Guidance

Classifier guidance was introduced in the paper “Diffusion Models Beat GANs on Image Synthesis” and essentially uses a classifier to guide the diffusion model to generate images of a desired class.

Algorithm 1 Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale s.

Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma \nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
end for
return x_0

Algorithm 2 Classifier guided DDIM sampling, given a diffusion model $\epsilon_\theta(x_t)$, classifier $p_\phi(y|x_t)$, and gradient scale s.

Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\hat{\epsilon} \leftarrow \epsilon_\theta(x_t) - \sqrt{1-\bar{\alpha}_t}\, \nabla_{x_t} \log p_\phi(y|x_t)$
  $x_{t-1} \leftarrow \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1-\bar{\alpha}_{t-1}}\hat{\epsilon}$
end for
return x_0

Classifier guidance sampling for DDPMs and DDIMs

The authors defined different algorithms for both DDPMs and DDIMs. The main idea is to take a pretrained classifier on the data you train the diffusion model.

The above function plays a major role in classifier guidance. You essentially sample gradients from a classifier when classifying an image of a desired class and feed that gradient information into the diffusion model to lead it to generate the desired class.

$$\nabla_{x_t} \log p_\phi(y|x_t)$$

Gradient of the log of the classifier parameters

I won't go deeper into classifier guidance since classifier-free guidance is much more effective, easier/more efficient to train, and easier to set up. This article goes more into classifier guidance and guidance in general.

Classifier-Free Guidance

Classifier-Free Guidance improves classifier guidance by eliminating the classifier while still providing class guidance to the model.

Assuming we can add class information to our diffusion model (which we can do with AdaGN), we can configure the model to generate images with and without classes:

$$\text{With Class: } \epsilon_\theta(z_\lambda, c)$$

$$\text{Without Class: } \epsilon_\theta(z_\lambda, c = \varnothing) = \epsilon_\theta(z_\lambda)$$

Noise estimation model with and without class (null class)

- `z_λ` is the image interpolation at some timestep `t`.
- `ε_θ` is our model, which we train to predict noise within `z_λ`.
- `c` is the class, which can be represented as a one-hot vector.
- `∅` is the null class, which can be represented as a vector of all `0`s (any linear combination will lead to another `0` vector which essentially encodes no class information).

To add classifier-free guidance to our diffusion model, all we have to do is train the model to generate images with class information and without class information.

Algorithm 1 Joint training a diffusion model with classifier-free guidance

Require: p_{uncond}: probability of unconditional training

repeat
  $(\mathbf{x}, \mathbf{c}) \sim p(\mathbf{x}, \mathbf{c})$ ▷ Sample data with conditioning from the dataset
  $\mathbf{c} \leftarrow \varnothing$ with probability p_{uncond} ▷ Randomly discard conditioning to train unconditionally
  $\lambda \sim p(\lambda)$ ▷ Sample log SNR value
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  $\mathbf{z}_\lambda = \alpha_\lambda \mathbf{x} + \sigma_\lambda \boldsymbol{\epsilon}$ ▷ Corrupt data to the sampled log SNR value
  Take gradient step on $\nabla_\theta \left\| \boldsymbol{\epsilon}_\theta(\mathbf{z}_\lambda, \mathbf{c}) - \boldsymbol{\epsilon} \right\|^2$ ▷ Optimization of denoising model
until converged

Training a diffusion model for classifier-free guidance

The training loop is slightly changed so that we can effectively train the model to generate images with and without class information. The authors define `p_uncond` as the probability of replacing a class with a null class to force the model to learn how to generate images without class information.

1. Normal loop over epochs.
2. Sample images, `x`, and their respective classes, `c`.
3. With a probability `p_uncond`, we make some of the classes null classes.
4. Sample timestep, `λ`.
5. Sample noise, `ε`, from a normal distribution.
6. Create the noisy image `z_λ`.
7. Train the model `ε_θ` using normal MSE loss between the predicted noise and real noise.

Notice how the training loop is almost exactly the same as the DDPM training loop. We just have to incorporate class information (along with making class information null with a probability). A good value of `p_uncond` was found to be `0.1` or `0.2`, meaning 10% or 20% of the images will be modeled without classes during training.

After training, the generation/sampling loop is also slightly changed.

Algorithm 2 Conditional sampling with classifier-free guidance

Require: w: guidance strength
Require: $\mathbf{c}$: conditioning information for conditional sampling
Require: $\lambda_1, \ldots, \lambda_T$: increasing log SNR sequence with $\lambda_1 = \lambda_{\min}, \lambda_T = \lambda_{\max}$
1: $\mathbf{z}_1 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = 1, \ldots, T$ **do**
▷ Form the classifier-free guided score at log SNR λ_t
3: $\tilde{\boldsymbol{\epsilon}}_t = (1 + w)\boldsymbol{\epsilon}_\theta(\mathbf{z}_t, \mathbf{c}) - w\boldsymbol{\epsilon}_\theta(\mathbf{z}_t)$
▷ Sampling step (could be replaced by another sampler, e.g. DDIM)
4: $\tilde{\mathbf{x}}_t = (\mathbf{z}_t - \sigma_{\lambda_t}\tilde{\boldsymbol{\epsilon}}_t)/\alpha_{\lambda_t}$
5: $\mathbf{z}_{t+1} \sim \mathcal{N}(\tilde{\boldsymbol{\mu}}_{\lambda_{t+1}|\lambda_t}(\mathbf{z}_t, \tilde{\mathbf{x}}_t), (\tilde{\sigma}^2_{\lambda_{t+1}|\lambda_t})^{1-v}(\sigma^2_{\lambda_t|\lambda_{t+1}})^v)$ if $t < T$ else $\mathbf{z}_{t+1} = \tilde{\mathbf{x}}_t$
6: **end for**
7: **return** $\mathbf{z}_{T+1}$

Sampling with classifier-free guidance

1. Sample noise from a normal distribution.
2. Normal sampling loop. Loop from time `t=1` to time `t=T`.
3. Get the noise output from the model given the null class `ε_θ(ₜz)` and given the class `ε_θ(ₜz, c)` of the image you want to generate. Interpolate these noise predictions to get the new noise prediction `εₜ~`.
4. Calculate the next step like normal using the interpolated noise prediction rather than the usual noise prediction.

The sampling loop is almost exactly the same as the DDPM sampling loop, but with one replacement:

$$\tilde{\boldsymbol{\epsilon}}_\theta(\mathbf{z}_\lambda, \mathbf{c}) = (1 + w)\boldsymbol{\epsilon}_\theta(\mathbf{z}_\lambda, \mathbf{c}) - w\boldsymbol{\epsilon}_\theta(\mathbf{z}_\lambda)$$

Noise model parameterization for classifier-free guidance

The noise prediction requires two forward passes of the same image, z_t. One forward pass calculates the predicted noise not conditioned on a desired class, and the other calculates the predicted noise conditioned on the desired class information.

When `w=0`, the model is a normal DDPM with class information.

When `w>0`, we utilize classifier-free guidance. The goal is to produce an image of class `c`. The idea is that the class-informed model will generate an output about the class we want to generate, but the class signal could be stronger.

To strengthen the signal from the class information, we can remove the signal from the model without class information (which should generate a random image). As `w` increases, we are removing more “null” images. Theoretically, the more information we remove with the null class, the more information we will have of the desired class.

This method works well up to a point. I’ve found a `w` value within the range [5, 20] works well but using a high `w` value removes too much signal from the image and essentially begins producing random noise since so much signal is removed.

Model	FID (↓)	IS (↑)
ADM (Dhariwal & Nichol, 2021)	2.07	-
CDM (Ho et al., 2021)	**1.48**	67.95
Ours	$p_{\text{uncond}} = 0.1/0.2/0.5$	
$w = 0.0$	1.8 / 1.8 / 2.21	53.71 / 52.9 / 47.61
$w = 0.1$	1.55 / 1.62 / 1.91	66.11 / 64.58 / 56.1
$w = 0.2$	2.04 / 2.1 / 2.08	78.91 / 76.99 / 65.6
$w = 0.3$	3.03 / 2.93 / 2.65	92.8 / 88.64 / 74.92
$w = 0.4$	4.3 / 4 / 3.44	106.2 / 101.11 / 84.27
$w = 0.5$	5.74 / 5.19 / 4.34	119.3 / 112.15 / 92.95
$w = 0.6$	7.19 / 6.48 / 5.27	131.1 / 122.13 / 102
$w = 0.7$	8.62 / 7.73 / 6.23	141.8 / 131.6 / 109.8
$w = 0.8$	10.08 / 8.9 / 7.25	151.6 / 140.82 / 116.9
$w = 0.9$	11.41 / 10.09 / 8.21	161 / 150.26 / 124.6
$w = 1.0$	12.6 / 11.21 / 9.13	170.1 / 158.29 / 131.1
$w = 2.0$	21.03 / 18.79 / 16.16	225.5 / 212.98 / 183
$w = 3.0$	24.83 / 22.36 / 19.75	250.4 / 237.65 / 208.9
$w = 4.0$	26.22 / 23.84 / 21.48	**260.2** / 248.97 / 225.1

Table 1: ImageNet 64x64 results ($w = 0.0$ refers to non-guided models).

Different FID and IS scores given different

The authors show one of the downsides to classifier guidance. Classifier guidance has a tradeoff between FID score and IS (inception score). FID measures quality and mode coverage, while IS measures the quality of images.

As you can see, as `w` increases (meaning more guidance), the FID score decreases, and the IS score increases. This means that as `w` increases, images have higher quality but have less variance.

My Results

I coded all the parts from scratch and combined them into one model. The code can be found here.

When creating the model and performing tests, I found the following parameters worked well, and I kept them constant:

1. Image Resolution: 64x64
2. Channel multiplier — 1
3. Number of U-net blocks — 3
4. Timesteps — 1000
5. VLB weighting Lambda — 0.001
6. Beta Scheduler — Cosine
7. Batch Size — 128 (across 8 GPUs, so 1024)
8. Gradient Accumulation Steps — 1
9. Number of steps (Note: This is not epochs, a step is a single gradient update to the model)— 600,000
10. Learning Rate — 3*10^-4 = 0.0003
11. Time embedding dimension size— 512
12. Class embedding dimension size — 512
13. Probability of null class for classifier-free guidance — 0.2
14. Attention resolution — 16

The only thing I really changed in my model is the number of embedding channels and the architecture of the u-net blocks. The u-net blocks can consist of the following:

- Resnet blocks (`res`) — A normal old residual connection convolution block with skip connections. The block is slightly edited to include optional time and class information.
- block (`conv`) — Normal ConvNext block with optional class and time information encoding.
- Normal attention (`atn`) — Following the , this block performs self-attention on the spatial dimension of the image embeddings.
- Class attention block (`clsAtn`) — Adds class information by constructing an attention matrix from class vectors that attends to the current embedding.
- Channel Attention (`chnAtn`) — An block to self-attend to the channels in the image embedding.

I created four models with 128 embedding channels (Notation is based on how embeddings are sequentially fed through the u-net block):

- `res` ➜ `conv` ➜ `clsAtn` ➜ `chnAtn` (Res-Conv)
- `res` ➜ `clsAtn` ➜ `chnAtn` (Res)
- `res` ➜ `res` ➜ `clsAtn` ➜ `chnAtn` (Res-Res)

- `res` ➜ `res` ➜ `clsAtn` ➜ `atn` ➜ `chnAtn` (Res-Res-Atn)

And one model with 192 embedding channels:

`res` ➜ `clsAtn` ➜ `chnAtn` (Res Large)

All u-net blocks have at least a single resnet layer, class attention layer, and efficient channel attention, which I found to be a good base to test on.

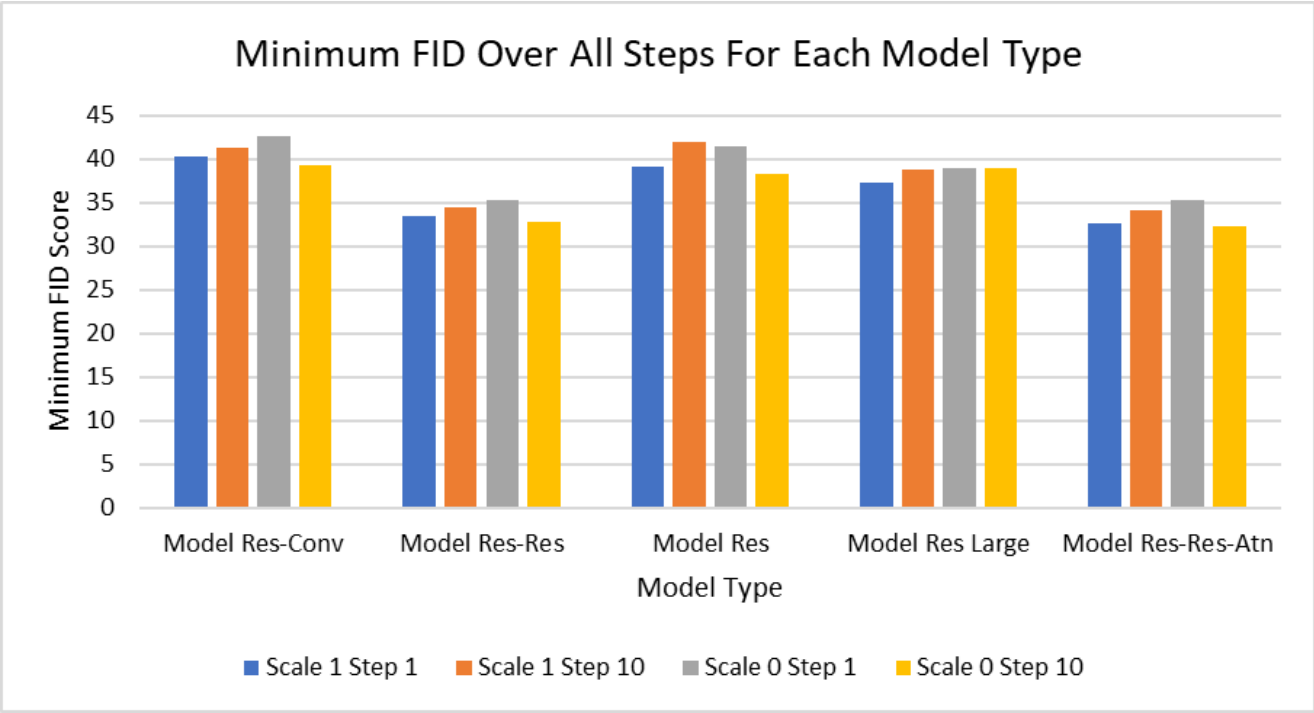
Although I trained with classifier-free guidance, I calculated FID scores without guidance as adding guidance requires me to test too many parameters. Additionally, I only collected 10,000 generated images to calculate my FID scores as that already took long enough to generate.

By the way, long FID generation times are one of the problems with diffusion, generation times take forever, and unlike GANs, you are not generating images during training. So, you can't continuously collect FID scores as the model is learning.

Although I keep the classifier guidance value constant, I wanted to test variations between DDIM and DDPM, so I looked at the step size and the DDIM scale. Note that a DDIM scale of 1 means DDPM, and a scale of 0 means DDIM. A step size of 1 means use all 1000 steps to generate images, and a step size of 10 means use 100 steps to generate images:

1. DDIM scale 1, step size 1
2. DDIM scale 1, step size 10
3. DDIM scale 0, step size 1
4. DDIM scale 0, step size 10

Below are the FID results for all u-net block variations on all four different DDIM/DDPM parameter changes:

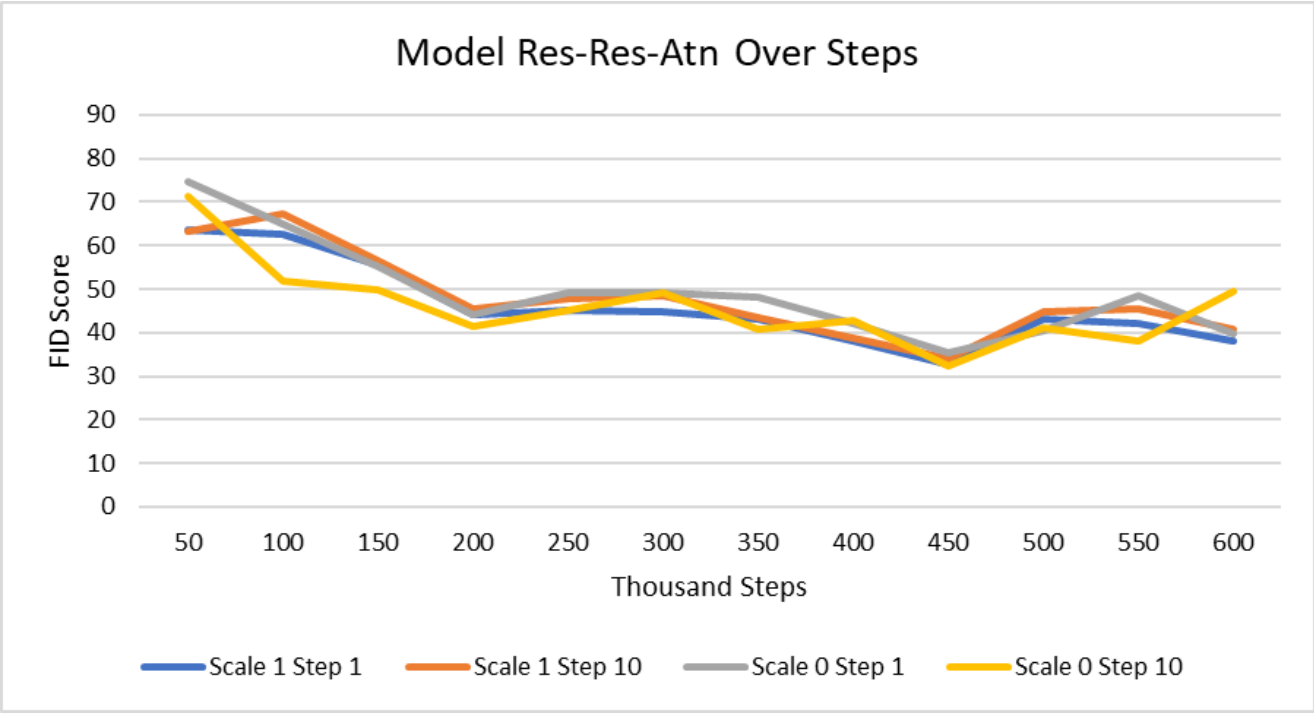


I calculate the FID score every 50,000 steps. I only show the minimum FID score over all 600,000 steps to reduce clutter.

Clearly, the models with two residual blocks performed the best. As for the attention addition, it doesn't look like it made much of a difference.

Also, using a DDIM (0 scale) with a step size of 10 outperformed all other DDPM/DDIM methods of generation. I find this fact interesting since the model was explicitly trained for DDPM (1 scale) generation on 1,000 steps but performs between with DDIM on 100 steps.

Just to get an idea of how the FID was moving around during training, here's the FID graph of the Res-Res-Atn model:



Model with a u-net blocks with a sequential res block -> res block -> cls attention block -> attention block -> channel attention block and its FID scores over steps

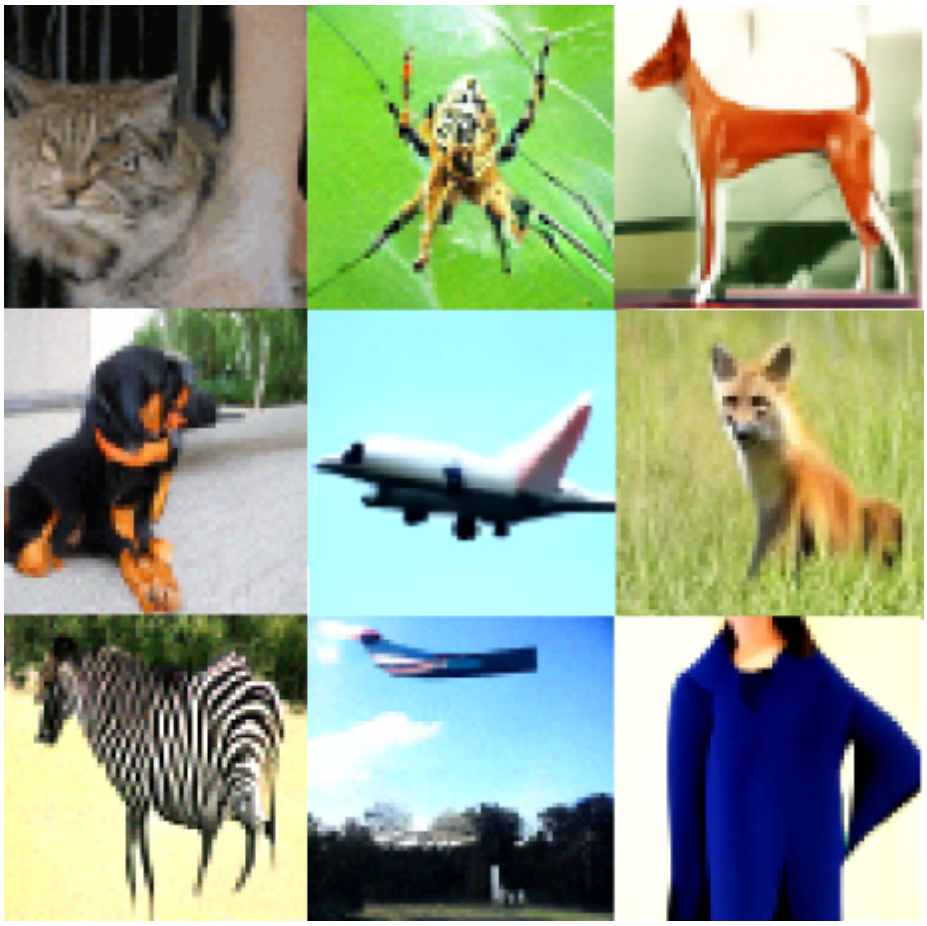
All models appeared to have this sudden increase in FID score around 450K steps which is very weird. Since FID doesn't measure overfitting, I cannot say it's overfitting at this point, and the loss curves show that the model is not taking a large step that destroys learning.

Finally, I am going to generate a batch of images with the following parameters:

- Classifier Guidance factor of 4
- DDIM (0 scale)
- 100 steps (10-step size)
- corrected = False
- Random class labels
- Using the Res-Res-Atn model at 450,000 steps since that had the lowest FID score.

Below are nine random images on random classes I pulled from the 1,000 classes the models are trained on.

Compressing a 65 MB gif to 22 MB doesn’t look very good. Oh well.



Resulting images

Overall, the results are OK. If I test different architectures a little more and fiddle with the hyperparameters, I may be able to improve the FID score some more, but I'm satisfied with what's produced for now.

If you are interested, the code is available in this repo with some pre-trained ImageNet models.

Sources

- Deep Unsupervised Learning using Nonequilibrium Thermodynamics:
- DDPM:
- Improved DDPM:
- DDIM:
- Diffusion Models Beat GANs on Image Synthesis:
- Classifier-Free Diffusion Guidance: