Implementing a Denoising Diffusion Probabilistic Model

Clayton Allard

Abstract

The basis of this project is to implement a denoising diffusion probabilistic model (DDPM) closely following [HJA20]. I developed a DDPM from scratch using techniques from various papers, and trained it on the MNIST dataset. As a result, the model has 12,552,848 parameters and can generate samples of hand written digits. Due to time constraints, I could not implement advancements and instead opted to summarize methods from various papers that could improve the model performance. My contribution to the field is the simple implementation, and my summary of methods. Github repository can be accessed (here).

1 Introduction

Over the past decade, generative models have become very prevalent within the field of machine learning. In 2014, Ian Goodfellow revolutionized generative modeling with his paper on Generative Adverserial Networks (GAN) [GPAM$^+$14]. Since then, capabilities have come from only being able to create low-quality images, to creating high resolution images that are indistinguishable from reality. While GANs revolutionized the field, it is diffusion models that are at the forefront of modern cutting-edge models. In 2020, a research group from Berkeley published a paper about denoising diffusion probabilistic models (DDPM) [HJA20] which popularized the use of diffusion models. The field has further evolved to using conditional DDPMs [DN21], and stable diffusion [RBL$^+$22]. However, the focus of this paper will be on the plain DDPM since it is necessary to understand that before working with the more advanced models.

This report covers the underlying details of DDPM, and the model architecture used for the implementation on the MNIST dataset. I was unable to make further advancements due to time constraints, so this report will also contain a summary of methods from various academic papers.

2 Details

Diffusion models are a variant of a variational auto-encoder (VAE) that approximates a probability distribution of a dataset through an encoding and a decoding process. We encode an image by adding independent Gaussian noise to each pixel which is known as the *forward process*. Then we decode by sampling from a standard Gaussian distribution and remove noise until we are left with an image that looks like it belongs to the dataset. This is known as the *reverse process*. The premise of this approach is that we are creating a mapping from a high-dimensional manifold space that is impossibly hard to sample from (all possible images of hand written digits), to a simple space that is easy to sample from (standard Gaussian distribution). This way, we only need to sample from a standard Gaussian in order to generate a sample from the dataset. The amount of math involved to explain all the details of the DDPM would easily surpass the page limit, so I will only explain the essential details.

2.1 Forward Process

We start by taking an image x_0 and gradually adding Gaussian noise for each iteration $x_1, x_2, \ldots, x_T$. As t increases, each x_t gets noisier until we arrive at x_T which is approximately distributed as a standard Gaussian random variable.

We can define a Markov chain q that demonstrates this process

$$q(x_{1:T} \mid x_0) = \prod_{i=1}^{T} q(x_t \mid x_{t-1}), \tag{1}$$

$$\text{where } q(x_t \mid x_{t-1}) \sim \mathcal{N}(\sqrt{1-\beta_t}x_t, \beta_t \mathbf{I}) \tag{2}$$

From this definition, we can make a few notes.

$$x_0 = \text{the original image} \tag{3}$$

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \tag{4}$$

$$x_T \sim \mathcal{N}(0, \mathbf{I}) \quad \text{(approximately)} \tag{5}$$

The β_t in the expressions above is the *variance schedule*. We define $\beta_t \in (0, 1)$ for each $t \in \{1, 2, \ldots, T\}$. Each β_t can be learned, or be chosen manually. Although learning seems like the obvious way to go, it is far more difficult to train due to instability. Our primary reference [HJA20] chooses values for β_t using a *linear schedule* ranging from $\beta_1 = 0.0001$ to $\beta_T = 0.02$. The typical rule of thumb is that we want the β_t values to increase monotonically in t. It is also worth noting that (5) is only true depending on the choice of β_t. We need to choose each β_t so that we get the approximation in (5), and we still retain information from x_0. If β_t is too small, then this approximation doesn't hold. If β_t is too large, then we lose too much information about x_0. It is a delicate balance.

In order to determine an appropriate β_t schedule, we determine the distribution of x_t. We define $\bar{\alpha}_t = \prod_{i=1}^{t} 1 - \beta_i$ and we can derive the following from (4).

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \tag{6}$$

Since $\bar{\alpha}_t$ decreases monotonically in t, this approaches closer to a standard Gaussian. When choosing a β_t schedule, we want $\bar{\alpha}_T$ slightly greater than 0, and for $\bar{\alpha}_t$ to decrease gradually instead of rapidly.

While the linear schedule does an okay job, the problem with it is that it causes $\bar{\alpha}_t$ to decrease too rapidly which quickly destroys information. The *cosine schedule* introduced in [ND21] mitigates this issue by decreasing $\bar{\alpha}_t$ more gradually as seen in Figure 1. For cosine scheduling, we choose $\bar{\alpha}_t$ to derive β_t rather than the other way around with the linear schedule. Therefore, we can control the rate at which information is destroyed. This comes out to be

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \quad f(t) = \cos^2\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right) \tag{7}$$

where $s = 0.008$ is chosen as a constant for numerical stability. Then we can derive $\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}$.

2.2 Reverse Process

Now that we have $x_T \sim \mathcal{N}(0, \mathbf{I})$, we can start the reverse process to remove the noise and recover the original image. Our ultimate goal is to fit a model to approximate $p_\theta(x_0) = \int p_\theta(x_{0:T}) dx_{1:T}$. We define the following Markov chain.

$$p(x_T) \sim \mathcal{N}(0, \mathbf{I}) \tag{8}$$

$$p_\theta(x_t \mid x_{t-1}) \sim \mathcal{N}(\mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{9}$$

$$p_\theta(x_{0:T}) = p(x_T) \prod_{i=1}^{T} p_\theta(x_{t-1} \mid t) \quad \text{(Markov property)} \tag{10}$$

θ represents the model parameters and the expression (9) indicates that our model will try to predict the mean and variance. Just like with β_t for the forward process, the Σ_θ can be learned or chosen manually. In fact, [HJA20]

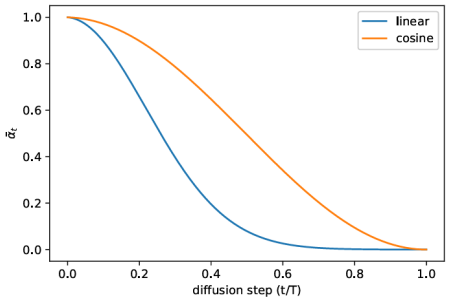

Figure 1: Taken directly from [ND21]. A comparison of $\bar{\alpha}_t$ in the diffusion process between the linear schedule and cosine schedule. The cosine schedule seems to be more gradual which makes training more efficient.

chooses $\Sigma_\theta(x_t, t) = \sigma_t^2 \mathbf{I}$ where $\sigma_t^2 = \beta_t$. It is worth noting that another common option is $\sigma_t^2 = \tilde{\beta}_t$ where $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}$. When we trace back the distribution of q, this is precisely the variance for $q(x_{t-1} \mid x_t, x_0)$. The paper provides more details about this derivation. However, the important distinction to make is that β_t is optimal for $x_0 \sim \mathcal{N}(0, \mathbf{I})$ and $\tilde{\beta}_t$ is optimal for x_0 deterministically set to one point. These are the two extreme choices that we'll discuss more about later.

2.3 Optimization

Our training objective is to find θ that maximizes $\mathbb{E}[\log p_\theta(x_0)]$. Since this computation is intractable, we use the *evidence lower bound* (ELBO) to find a lower bound which is tractable.

$$\mathbb{E}[\log p_\theta(x_0)] \geq \mathbb{E}_q \left[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)} \right] \tag{11}$$

It is standard to negate this to get an upper bound for minimizing the negative log-likelihood. We only neglect to negate it in (11) to demonstrate the ELBO. The following expressions is derived from the negative ELBO. After performing a long series of algebraic manipulation, we get the following equality.

$$\begin{aligned} L_{\text{vlb}} &= -\mathbb{E}_q \left[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)} \right] \\ &= \mathbb{E}_q \Big[\underbrace{\text{KL}(q(x_T \mid x_0) \| p(x_T))}_{L_T} + \sum_{t>1} \underbrace{\text{KL}(q(x_{t-1} \mid x_t, x_0) \| p_\theta(x_{t-1} \mid x_t))}_{L_{t-1}} \underbrace{- \log p_\theta(x_0 \mid x_1)}_{L_0} \Big] \end{aligned} \tag{12}$$

The full derivation is provided in appendix A of [HJA20]. Since L_T does not depend on the model, we can view this expression as a constant. For now we will only focus on L_{t-1}. Given that we've chosen $\Sigma_\theta(x_t, t) = \sigma_t^2 = \beta_t$, we minimize the following expression.

$$L_{t-1} = \mathbb{E}_q \left[\frac{1}{2\sigma_t^2} \| \tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t) \|^2 \right] \tag{13}$$

where $\tilde{\mu}(x_t, x_0)$ is the mean of $q(x_{t-1} \mid x_t, x_0)$. We can reparameterize $\mu_\theta(x_t, t)$ into $\epsilon_\theta(x_t, t)$ so that we predict the noise rather than predicting the mean.

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{1-\beta_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) \tag{14}$$

Even though it is mathematically the same using either parametrization, predicting ϵ leads to better sample quality. Another series of derivations gets us to minimizing the following objective function.

$$L_{t-1} = \mathbb{E}_{x_0,\epsilon}\left[\frac{\beta_t^2}{2\sigma_t^2(1-\beta_t)(1-\bar{\alpha}_t)}\|\epsilon - \epsilon_\theta(\underbrace{\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon}_{x_t}, t)\|^2\right] \tag{15}$$

From this point, it turns out that sample quality is better if we simplify this expression to remove the scaling, and let $t \in \{1, 2, \ldots, T\}$ be a uniform random variable.

$$L_{\text{simple}}(\theta) = \mathbb{E}_{t,x_0,\epsilon}\left[\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2\right] \tag{16}$$

Algorithm 1 Training

1: **repeat**
2: $x_0 \sim q(x_0)$
3: $t \sim \text{Uniform}(\{1, \ldots, T\})$
4: $\epsilon \sim \mathcal{N}(0, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2$
6: **until** converged

Algorithm 2 Sampling

1: $x_T \sim \mathcal{N}(0, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $z \sim \mathcal{N}(0, \mathbf{I})$ if $t > 1$ else $z = 0$
4: $x_{t-1} = \frac{1}{\sqrt{1-\beta_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sigma_t z$
5: **end for**
6: **return** x_0

Figure 2: Taken directly from [HJA20]. The training and sampling algorithms for the DDPM using $L_{\text{simple}}(\theta)$.

2.4 U-net

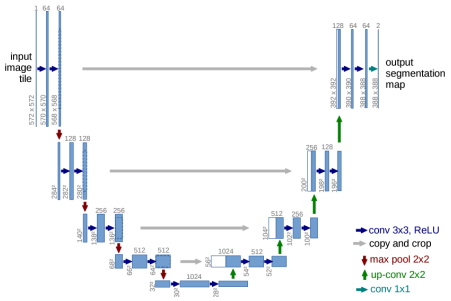

Figure 3: Taken directly from [RFB15]. This is the U-net architecture with the blocks of convolution layers between the down-sampling and up-sampling layers. Also demonstrates the skip connections between the respective layers of the up-sampling and down-sampling blocks.

The model that is typically used for p_θ is a U-net [RFB15]. A U-net is a fully convolutional neural network that learns by scaling down the image with max-pooling layers, and scaling it back up with transposed convolutional layers with multiple convolutional layers between each of these down-scaling/up-scaling layers. In the process of scaling up, we include skip connections from the corresponding layer of the scale down phase. This ensures that we can learn complex patterns in down-scaling without losing information.

3 Implementation

I implemented a DDPM from scratch using the algorithm above. I used the cosine schedule (7) with $T = 100$. As mentioned before, using the cosine schedule is more efficient than linear schedule as it allowed [ND21] to use $T = 25$ instead of [HJA20] using $T = 1000$. I trained for 2000 epochs with a batch size of 1000 which took about 8 hours on my laptop using CUDA acceleration on my RTX 4070 graphics card.

3.1 U-net architecture

My U-net architecture nearly matches Figure 3 with 4 layers of skip connections. Max-pooling for the down-sampling and transposed convolution for up-sampling. Between each down-sampling and up-sampling layer are two convolutional layers all with batch normalization and leaky ReLU. The first convolutional layer has 32 channels. Each down-sampling layer doubles the channel count until we reach the up-samping phase. Then the amount of channels is halved until we are back to 32 at the final up-sampling layer where we have one last convolutional block which gets it down to one channel.

To account for the time embedding, we have 32 time embedding dimensions where we use the sinosuidal position embedding as input to a fully connected neural network. Each convolutional block has its own trainable fully connected neural network where the output dimension is the number of channels for that layer. We then add the output of the network into each channel respectively. This is the only part of the implementation that I copied someone else's code for [Dom22].

In all, there are 12,552,848 parameters.

3.2 Results

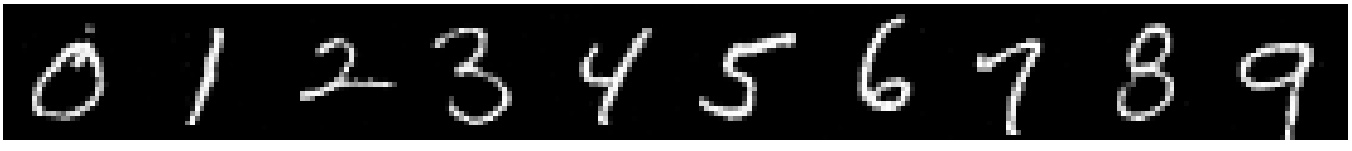

Figure 4: Samples from my DDPM model of each digit

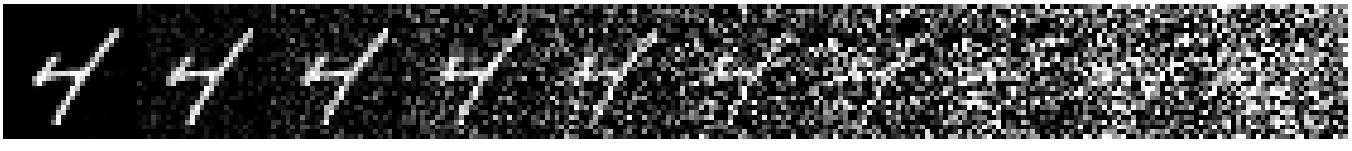

Figure 5: The reverse process for one sample. From left to right, $x_0, x_{10}, x_{20}, \ldots, x_{90}$.

Two of the most comment metrics for evaluating generative models is Inception score (IS) and Frechet Inception Distance (FID). Both of these metrics are algorithms made to give an objective score for the quality and diversity of the samples. Unfortunately I had a lot of difficulty trying to implement these for use. So I decided to scrap it and use the eyeball test. The images in Figure 4 look great. Out of the 100 samples I did, they were the best of each digit. About half the samples look indistinguishable from human written digits. Most of the ones that aren't are still recognizable, and only a few are unreadable. One thing I did notice though is that there is a lack of sample diversity as 40 out of 100 samples were '1'.

Number	0	1	2	3	4	5	6	7	8	9	?
Count	2	40	4	10	15	5	5	11	2	3	2

Figure 6: Out of 100 samples, this is the total number of times each number appeared. Numbers are counted based on my own judgement. '?' indicates that the image did not look like a digit.

4 Discussion

I had a very frustrating time getting my implementation to work. It took me about 2 weeks of debugging to finally get it to work. I wanted to be more ambitious with what I accomplished in this project, but it was getting too close to the deadline to try anything new. I tried training my model on CIFAR-10 with a similar architecture for the same number of epochs. Unfortunately I could not get any meaningful results. An obvious way to fix this is to make the model bigger and train for longer. The only thing I didn't try from [HJA20] was add self-attention layers.

There are numerous techniques I could use from [ND21]. We can learn Σ_θ instead of choosing values manually.

$$\Sigma_\theta(x_t, t) = \exp(v\beta_t + (1-v)\tilde{\beta}_t) \tag{17}$$

where $v \in \mathbb{R}$ is a tunable parameter. We attempt to learn an ideal balance between the two extremes we discussed earlier in Section 2.2. L_{simple} for training (16) does not depend on Σ_θ, so we define a hybrid method that does.

$$L_{\text{hybrid}} = L_{\text{simple}} + \lambda L_{\text{vlb}}, \quad \lambda = 0.001 \tag{18}$$

Using this hybrid gives us a better choice for Σ_θ which makes the first few steps of the diffusion process more efficient.

One issue that was addressed above is the lack of diversity. I may see 20 ones before I see one eight. One way to get more diversity is to use classifier guidance [DN21]. This is where we can condition on a label in the reverse process to have more control over the model's output.

$$p_{\theta,\phi}(x_t \mid x_{t+1}, y) \propto \underbrace{p_\theta(x_t \mid x_{t+1})}_{\text{denoiser}} \underbrace{p_\phi(y \mid x_t)}_{\text{classifier}} \tag{19}$$

One last consideration is implementing stable diffusion [RBL+22]. The idea with stable diffusion is to embed data into a latent space enabling us to generalize to higher resolution images while keeping the runtime cost manageable. The use of a latent space also allows us to use any form of data as input. The most popular use of stable diffusion is to embed words into a latent space to generate images through the diffusion process.

5 Conclusion

This is a very suitable project for this course because it combines many key concepts we've covered including Markov chains, fully convolutional networks, and variational auto-encoders. In this project, I created my own implementation of the denoising diffusion probabilistic model (DDPM) from scratch training on the MNIST dataset. After training on 12,552,848 parameters for 2000 epochs, the model was successful. Most of the generated samples look indishtinguishable from a human written digit. While I was unable to generalize this to a more complicated setting, I am proud of my contributions. I have built a Github repository for a simple implementation and have summarized many of the key concepts about DDPMs across various papers. Through working on this project, I feel well equipped to make bigger contributions in the future.

References

[DN21] Prafulla Dhariwal and Alex Nichol, *Diffusion models beat gans on image synthesis*, arXiv:2105.05233 (2021).

[Dom22] Dome272, *Diffusion-models-pytorch*, GitHub repository, 2022.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, *Generative adversarial nets*, Advances in Neural Information Processing Systems **27** (2014), 2672–2680.

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel, *Denoising diffusion probabilistic models*, arXiv:2006.11239 (2020).

[ND21] Alex Nichol and Prafulla Dhariwal, *Improved denoising diffusion probabilistic models*, arXiv:2102.09672 (2021).

[RBL+22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer, *High-resolution image synthesis with latent diffusion models*, arXiv:2112.10752 (2022).

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, *U-net: Convolutional networks for biomedical image segmentation*, Medical Image Computing and Computer-Assisted Intervention (MICCAI) **9351** (2015), 234–241.