Recreating Denoising Diffusion Probabilistic Models: A Scaled-Down Approach

Kevin Jones
Purdue University
jone2342@purdue.edu

Abstract

My project recreated the denoising diffusion probabilistic models (DDPM) as described by Ho et al. I focused on a smaller scale implementation suitable for limited computational resources. I gained insight into the underlying principles of diffusion models and evaluated their performance on the CIFAR-10 dataset. Additionally, comparisons across different numbers of time steps and the effect of mixed precision training were studied.

1. Introduction

Diffusion models are the heart of image generation in the modern day. Initially introduced by Ho *et al*. [5] in 2020, the models work by gradually denoising a random image to generate a new, original image. My focus was on developing a smaller-scale version that can be trained using an NVIDIA 4070 GPU. This involved setting up the model architecture, training it the CIFAR-10 dataset, and evaluating its performance in terms of image quality metrics. The implementation was done in pytorch. In order to gain a complex understanding of diffusion models, I trained models from scratch and varied several features. I trained models with different numbers of time steps to understand the effect of the range of time steps through training and image generation. Additionally, in order to train these large models with limited computing resources, I studied the effect of mixed-precision training on the performance of the diffusion model.

2. Related Works

Diffusion probabilistic models have gained attention for their ability to generate high-quality images. The foundational work by Ho et al. [5] demonstrated their potential by achieving state-of-the-art results on various benchmarks. Other notable works include Song et al.'s exploration of score-based generative models [2] and Dhariwal et al.'s improvements on diffusion models through architectural enhancements [1]. Additionally, latent diffusion models

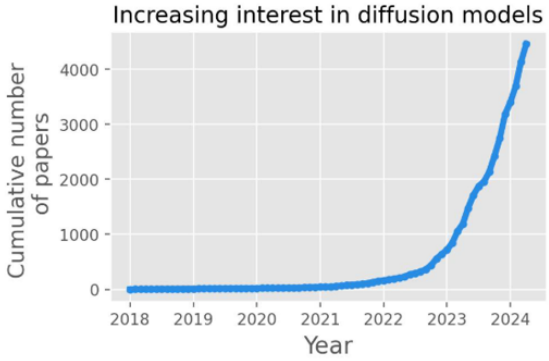


Figure 1. Growth of Diffusion Model Research

[3] and continuous time diffusion models [4] have further expanded the capabilities of diffusion-based approaches. Since the initial papers introducing diffusion models and score based models were published, the interest in the field has increased exponentially because of the promise of high quality results as seen in Fig. 1. Recently, researchers are attempting to apply diffusion models to video generation as demonstrated by Bar-Tal *et al*. [6]. Since diffusion models have been discovered, they have been applied to a plethora of use-cases and continue to be a very active, relevant area of research.

3. Approach

In order to develop a denoising diffusion probabilistic model, two key components are required. Diffusion models operate by iteratively adding and removing noise from images to learn a generative process. During training, a noise scheduler defines a forward process, gradually corrupting the original data over a series of time steps.

3.1. Diffusion Component

The input $\mathbf{x}_0$ is transformed into a sequence of increasingly noisy versions $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_T$ where $\mathbf{x}_T$ is pure Gaus-

sian noise and T is the number of time steps used. This process is modeled by a fixed schedule β_t where $t \in \{1, 2, \cdots, T\}$. Each β_t represents the variance of noise added at each step. The forward process can be expressed as

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{1}$$

The above defines a transition from a previous time step to the next time step. However, during training a time step is uniformly sampled and the appropriate amount of noise is added for the given time step. The cumulative transition from the original input $\mathbf{x}_0$ to $\mathbf{x}_t$ is given by

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{2}$$

where $\bar{\alpha}_t$ is defined as

$$\bar{\alpha}_t = \prod_{i=1}^{t}(1-\beta_i) \tag{3}$$

In the definitions above, β_t represents an arbitrary noise scheduler. In my implementation, I did not have enough time to experiment with multiple different noise schedulers. I used a linear noise scheduler where β_t is defined as:

$$\beta(t) = \beta_{small} + \frac{t}{T}(\beta_{large} - \beta_{small}) \tag{4}$$

Above, β_{small} is the starting value of the noise, or the minimum variance. In my implementation I used $\beta_{small} = 1e^{-4}$. β_{large} is the ending value of the noise, or the maximum variance. In my implementation I used $\beta_{large} = 0.02$. T is the total number of time steps and t is the current time step. The linear schedule interpolates between β_{small} and β_{large} ensuring that the noise level increases gradually from a small value to a larger value as t progresses. The resulting noise schedule plays a crucial role in determining the quality and stability of the training process. By setting $\beta_{small} = 1e^{-4}$ and $\beta_{large} = 0.02$, the schedule starts with a very small noise variance and increases linearly to a higher value. This ensures that the initial steps introduce minimal corruption and the later steps approach Gaussian noise. The generative process seeks to reverse this noise addition. During the reverse process, the algorithm begins by sampling a random starting image $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$. The image is then iteratively updated from t to $t-1$. At each time step the new image is derived with respect to the following equation

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\beta_t}\cdot z \tag{5}$$

Above, $\frac{1}{\sqrt{\alpha_t}}$ serves as the pre-scaling factor. The role of this factor is to adjust the sample to remove the influence of noise added. At a given time step, α_t is defined as

$$\alpha_t = 1 - \beta_t \tag{6}$$

and β_t is given by Eq. (4). $\mathbf{x}_t$ represents the noisy image at the current time step t. The $\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}$ term is called the epsilon scale factor. The role of the epsilon scale factor is to scale the predicted noise to ensure that the correction is proportional to the intended noise removal. The term $\epsilon_\theta(\mathbf{x}_t, t)$ represents the predicted noise from the model parameterized by θ. The input to the model is the image at the current time step $\mathbf{x}_t$ and the time step itself t. This predicted noise is attained from a convolutional neural network which will be described in more detail later. Given just the parameters above, the reconstruction process is entirely deterministic. Given a random noisy input, the input is iteratively reconstructed according to the process above (ignoring the last term) and results in a final image. If this process were to be repeated many times with the same noisy input, the exact same picture would be obtained. In order to reintroduce some noise and randomness, the final term is added. This noise and randomness is desired as it allows the model to have some flexibility when generating new images. The term $\sqrt{\beta_t}\cdot z$ is the component that reintroduces randomness. Here again, β_t is obtained through Eq. (4) and z is defined as

$$z \sim \mathcal{N}(0, \mathbf{I}) \tag{7}$$

This term introduces controlled noise based on a predetermined variance schedule to ensure smooth transitions between states. Scaling by a factor of $\sqrt{\beta_t}$ ensures that the noise level matches the variance of the reversed process at each time step. Another way to view diffusion processes is as a Markov Chain.

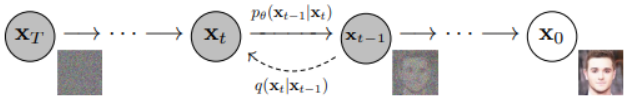


Figure 3. Image Generation as a Markov Chain

Fig. 3 demonstrates this Markovian process. Adding noise to the image is represented through a distribution q and described completely in Eq. (1). When removing noise from the image as done when sampling, the process is represented through a distribution p. The distribution p is defined as

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)) \tag{8}$$

Here μ_θ and Σ_θ are neural network parameterized functions trained to predict the mean and variance of the distribution used to generate an image. In my implementation, and considering the process described above, the mean is described as

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) \tag{9}$$

This provides the per pixel mean distribution for the next image $\mathbf{x}_{t-1}$ given the current image $\mathbf{x}_t$. The variance is im-

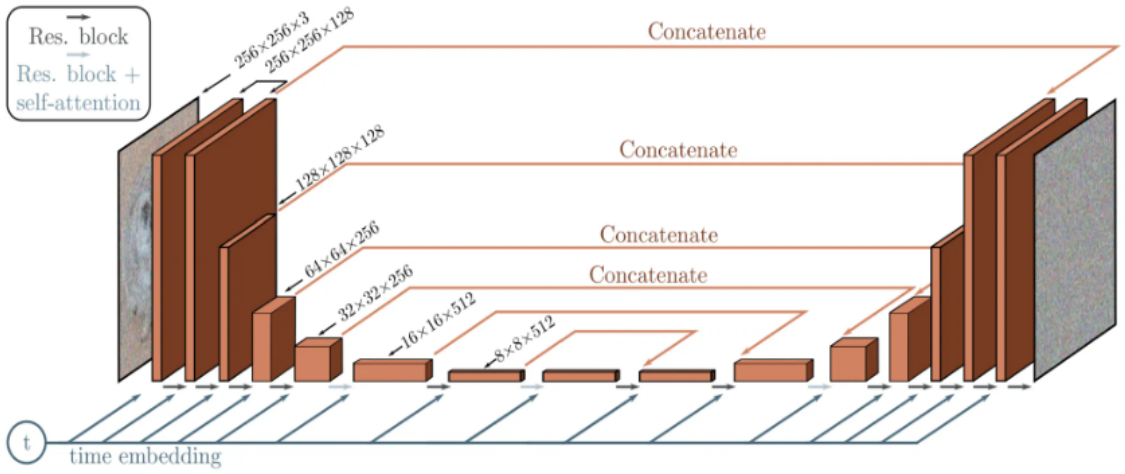


Figure 2. U-Net Architecture

plemented as

$$\Sigma_\theta(\mathbf{x}_t, t) = \beta_t \tag{10}$$

which scales the Gaussian noise z. Because z is distributed normally as defined in Eq. (7), scaling this noise provides a variance to the distribution p with respect to β_t. A more complete visual look at the forward and reverse diffusion process is depicted below.

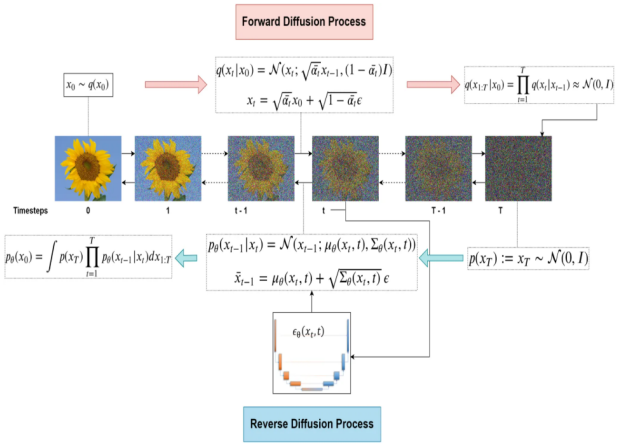


Figure 4. Forward and Reverse Diffusion Process

Fig. 4 demonstrates the cycle of adding noise to an image to destroy it and then reversing the process to remove the noise.

3.2. Neural Network Component

The process described above explains how the diffusion process destroys and reconstructs an image. To generate new images, the reverse diffusion is implemented. The key step in the reverse diffusion is obtaining $\epsilon_\theta(\mathbf{x}_t, t)$ which is the predicted noise given the input image and the time step. In order to predict the noise, a large convolutional neural network is used. The type of convolutional neural network used is called a U-Net (Fig. 2). It gets its name because of its U-shape. In Fig. 2 the input image has size 256×256. However, in the Cifar-10 dataset, the images are 32×32. The architecture of the U-Net used is:

- DoubleConv(3, 64)
- Down(64, 128)
- Down(128, 256)
- SelfAttention(256, 8)
- Down(256, 256)
- SelfAttention(256, 4)
- Up(512, 128)
- SelfAttention(128, 8)
- Up(256, 64)
- Up(128, 64)
- OutConv(64, 3)

The DoubleConv block consists of two convolutions each followed by a GroupNorm. The GroupNorm simply normalizes the channels. In between the two convolutions is a GELU activation. The first convolution is a convolution from in_channels to in_channels, so no channel scaling occurs. The second convolution is a convolution from in_channels to the specified out_channels. This is used for scaling the number of channels up or down. The Down block decreases the size of the image while increasing the number of channels. These blocks are used for the downward part of the U-shape. Each Down layer begins with a max pooling layer consisting of a kernel of size 2 and a stride equal to the kernel size. This will reduce the height and width of the image by a factor of 2. After this, two DoubleConv blocks are used, the first with out_channels equal to in_channels. And the second with out_channels being twice the number of in_channels. Because each DoubleConv block has two convolutions, the Down block will have four total convolutions, with only the last one increasing the number of channels. The Up block increases the size of the image while decreasing the number of channels. These blocks are used for the upward part of the U-shape. The first part of the Up block is a bilinear upsampling of the image. This increases the height and width of the image by a factor of 2. After this, two DoubleConv blocks are used again the first with in_channels equal to out_channels, and the second with in_channels being four times the number of out_channels. Again, of the four convolutions, only the last decreases the number of channels. The reason that in_channels is four times the size of out_channels is because of the skip connections. This will become more clear once the path through the neural network is fully explained. There is also an OutConv block that simply applies one convolution to transform the image back to three channels for R, G, and B. The final block is the SelfAttention block. This block employs a single SelfAttention block that is comprised of a MultiHeadedAttention block with 4 heads, a linear layer after the attention block, and two residual connections from the input to the attention, and from the attention to the linear layer. This block helps the neural network understand and attend to surrounding pixels. One additional component of the neural network is the time encoding. Sometimes this is done in a learned fashion through an embedding layer. In this implementation it is implemented deterministically. The time encoding is defined as

$$t_{\text{enc}} = \begin{cases} \sin\left(t \cdot 10000^{\frac{c}{C}}\right) & \text{if } c \text{ is even,} \\ \cos\left(t \cdot 10000^{\frac{c}{C}}\right) & \text{if } c \text{ is odd} \end{cases} \tag{11}$$

where t is the time step, c is the current channel, and C is the total number of channels. Even-indexed channels use sin while odd-indexed channels use cos. There is also the frequency scaling factor which ensures that higher channels correspond to higher frequencies. The core idea of the sinusoidal time encoding is to represent the position time steps using a combination of periodic functions at different frequencies. Lower frequencies encode broad, coarse-grained time information such as general trends over time. Higher frequencies capture fine-grained time details and small-scale changes. This multi-frequency representation ensures that the encoding can effectively represent both local and global dependencies across time steps. A forward pass looks as follows: the input is a $32 \times 32 \times 3$ image with the three channels corresponding to R, G, B and a time step t. The image gets passed through a DoubleConv block which increases the number of channels from 3 to 64. The result is stored in x_1 and is used later for a skip connection. Its shape is $32 \times 32 \times 64$. The resulting x_1 is then passed through the first Down block where dimensionality is decreased by a factor of 2 and the number of channels increases to 128. The resulting shape is $16 \times 16 \times 128$. The time encoding for t is calculated and the resulting shape for the time encoding is also $16 \times 16 \times 128$ where each 16×16 spatial region has the same value for each pixel. The values only differ across channels. This time encoding is added element-wise to the output from the down block and stored in x_2 to be used for a skip connection later. This result goes through a second Down block where its new shape becomes $8 \times 8 \times 256$. The time embedding is calculated again and added to the result of the second Down block. This result then goes through the first SelfAttention block and is stored as x_3. From here, a final Down block is applied yielding a shape of $4 \times 4 \times 256$. The time encoding is calculated and added, and the result passes through the second SelfAttention block and is stored as x_4. At this point the forward pass progress back up the "U" shape. The first part of the Up block uses bilinear interpolation to upsample the input. So x_4 becomes a $8 \times 8 \times 256$ tensor. The first Up block is defined with 512 in_channels and 128 out_channels. The 512 in_channels come from the concatenation of x_4 and x_3 along the channel dimension. Both x_4 and x_3 have size $8 \times 8 \times 256$ and when concatenated together have size $8 \times 8 \times 512$. This implements the first residual connection. This concatenated tensor then goes through the rest of the Up block and becomes an $8 \times 8 \times 128$ tensor. The time encoding is calculated and added element wise to this tensor. The result passes through the final SelfAttention head. Afterwards, it enters the next Up block becoming a $16 \times 16 \times 128$ tensor. This gets concatenated with x_2 which is of the same size. The resulting concatenated shape is $16 \times 16 \times 256$ and this passes through the rest of the Up block with 256 in_channels and 64 out_channels becoming a $16 \times 16 \times 64$ tensor. The time encoding is added to this result. The result then passes through the last Up block becoming a $32 \times 32 \times 64$. This is concatenated with x_1 and continues through the final convolutions with 128 in_channels and 64 out_channels. The time embedding is

once again added and the resulting shape is $32 \times 32 \times 64$. Finally, this passes through one last simple convolution with 64 in_channels and 3 out_channels. This final result is the predicted noise added to the input image.

3.3. Training

Training the denoising diffusion probabilistic model involves learning to reverse a noisy corruption applied to the training images. The training objective is derived from variational inference where the goal is to maximize the likelihood of the data $p(\mathbf{x}_0)$. The Evidence Lower Bound (ELBO) provides a tractable surrogate for this objective. The ELBO is defined as

$$\text{ELBO} = \mathbb{E}_q \left[\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right] + \mathbb{E}_q \left[\sum_{t=2}^{T} \log p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) - \log q(\mathbf{x}_t|\mathbf{x}_{t-1})\right] \tag{12}$$

In the ELBO, q is the conditional distribution as defined in Eq. (1) which is the known forward process that adds noise. p_θ is the distribution parameterized by the neural network which predicts the noise given an image and a time step. The first term of the ELBO when maximized ensures that the final reconstruction is of high quality. The second term of the ELBO, the summation, ensures that the reconstruction error throughout the process is low because maximizing this term is equivalent to minimizing the KL divergence between the true distribution q and the learned distribution p_θ. This helps ensure that the reverse process is as close as possible to the true diffusion process, resulting in a better overall reconstruction throughout the entire process. The full ELBO can be simplified for the Gaussian diffusion process. This results in a practical objective that can reduce to minimizing the Mean Squared Error between the true noise ϵ and the predicted noise $\epsilon_\theta(\mathbf{x}_t, t)$ at each step:

$$L_t = \mathbb{E}_q \left[||\epsilon - \epsilon_\theta(\mathbf{x}_t, t)||^2\right] \tag{13}$$

In the implementation, during each epoch images are processed in batches. For each image in a batch, a time step is sampled as $t \sim \text{Uniform}(1, T)$. The noisy image is created using Eq. (2). The true noise added is stored as ϵ. The noisy image is fed through the U-Net and the predicted noise $\epsilon_\theta(\mathbf{x}_t, t)$ is returned. The MSE between ϵ and $\epsilon_\theta(\mathbf{x}_t, t)$ is calculated and used to back propagate through the model and update the parameter weights. This process is repeated for many epochs to train the model. Specifically for my implementation, I trained six models each for 500 epochs using a batch size of 256. Each epoch consisted of 50,000 training images from the CIFAR-10 dataset. Four of the models were trained with different time steps: 100, 500, 1000, 2000 while training at full precision. Two models were trained with 2000 time steps using mixed precision. For the training, a learning rate of $2e^{-4}$ was used throughout the first 400 epochs before being dropped to $2e^{-5}$ for the last 100 epochs. The optimizer used for training was ADAM. The size of the models stayed consistent throughout all models at 12,920,323 parameters. All models were trained on a NVIDIA-4070 GPU.

4. Results

The goal of training these different models was two-fold: firstly to observe the difference in image generation quality, training speed, and inference speed when using different time steps (100, 500, 1000, 2000), and secondly to observe the difference in image generation quality, training speed, and inference speed when using mixed precision training (float32, float16 with gradient scaler, bfloat16).

4.1. Time Step Study

The first four models involve the training and evaluation of models with different time steps.

4.1.1 100 Time Step Model

The first model trained was one with 100 time steps. The model took 6.78 hours to train for 500 epochs. The model converged fairly quickly, before all 500 epochs so only the first 100 are displayed in Fig. 5. The validation loss converges around 0.0978.

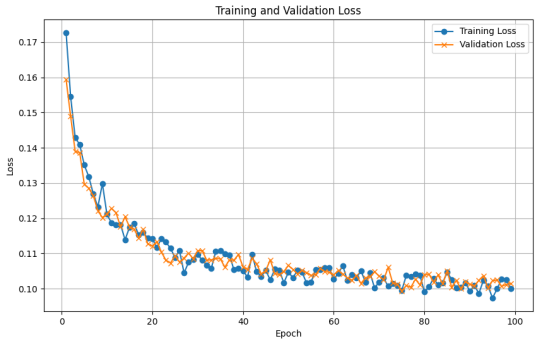


Figure 5. 100 Time Step Training and Validation Loss Curve

The training and validation loss essentially mirror each other throughout training suggesting that the model is not overfitting. In order to calculate the FID and Inception Score for the model, 5120 images were generated. For the FID score, 5120 images for the training set were used to compare the distribution of the real and fake images. The model took 217 seconds to sample 5120 images. As expected, this is significantly quicker than the other models. The resulting FID score was 72.307 indicating that the model is able to reasonably well mimic the images in the

training dataset but with a visible error. The resulting Inception Score was 4.9667 indicating that the images are of reasonable quality and diversity but there is still significant room for improvement. Fig. 6 displays a sample of 64 generated images.

Figure 6. 100 Time Step Generated Images

Viewing the generated images, they are able to attain a fair amount of structure. The backgrounds tend to be more monochromatic with something resembling an animal in the middle of the image. Interestingly, there do not appear to be any images generated resembling vehicles. These shortcomings can be attributed to the small number of time steps. The model does not have time to refine the images and add detail.

4.1.2 500 Time Step Model

The next model trained was one with 500 time steps. The model took 7.22 hours to train for 500 epochs. The model converged fairly quickly, before all 500 epochs so only the first 100 are displayed in Fig. 7. The validation loss converges around 0.0452. Again, the training and validation loss essentially mirror each other throughout training suggesting that the model is not overfitting. To calculate the FID and Inception Score for the model, 5120 images were generated. The model took 938 seconds to sample 5120 images. The sampling took approximately 5 times as long as sampling from the 100 time step model. Exactly, it took 4.32 times longer but this can be attributed to some other factors. This makes sense as the diffusion process has to go through 5 times as many steps to generate the image. The

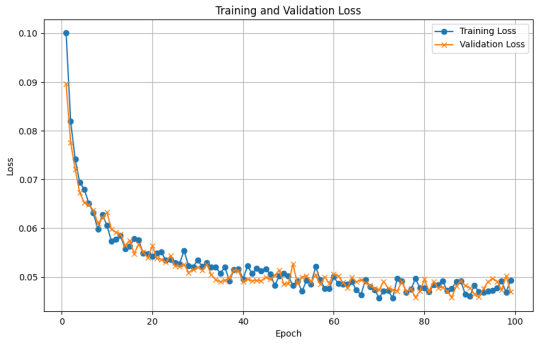


Figure 7. 500 Time Step Training and Validation Loss Curve

resulting FID score was 26.635 indicating that the model is able to produce images very similar to the training set. The resulting Inception Score was 7.7636, the highest of any of the models, indicating that the images are of high quality and diversity. Fig. 8 displays a sample of 64 generated images using 500 time steps.

Figure 8. 500 Time Step Generated Images

The resulting images are of impressively high quality and a significant improvement from the model using only 100 time steps. The images are a very diverse representation of the dataset with many vehicles and animals clearly distinguishable. Much of the visual error can be attributed to the low resolution of the images. The generated images are the same resolution as the images in the dataset, 32×32. This results in images that appear very pixelated. They are

not quite of the same quality as the training images but on the right track.

4.1.3 1000 Time Step Model

The third model trained was one with 1000 time steps. The model took 7.71 hours to train for 500 epochs. The model converged fairly quickly, before all 500 epochs so only the first 100 are displayed in Fig. 9. The validation loss converges around 0.0319.

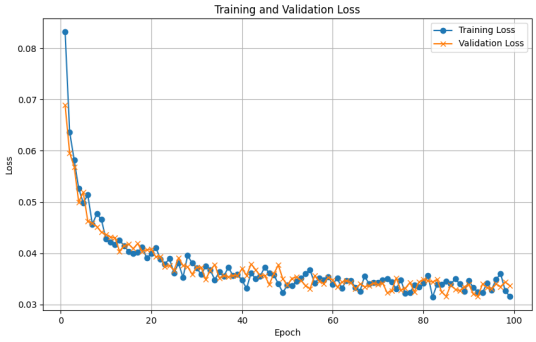


Figure 9. 1000 Time Step Training and Validation Loss Curve

Again, the training and validation loss essentially mirror each other throughout training suggesting that the model is not overfitting. To calculate the FID and Inception Score for the model, 5120 images were generated. The model took 1861 seconds to sample 5120 images. The sampling took almost exactly twice as long as sampling from the 500 time step model. This is expected as the diffusion process has to go through twice as many steps to generate the image. The resulting FID score was 25.517 indicating that the model is able to produce images very similar to the training set. The FID score was the lowest across all models. The resulting Inception Score was 7.5891 indicating that the images are of high quality and diversity. Fig. 10 displays a sample of 64 generated images using 1000 time steps. The resulting images are of impressively high quality. The images offer a very diverse representation of the dataset with many vehicles and animals clearly distinguishable. Much of the visual error can be again attributed to the low resolution of the images. There is not a large noticeable visual difference between the model with 1000 time steps and the one with 500 time steps, which is reflected in the similar FID score and Inception Score. When looking closely, it appears this model has slightly more detailed backgrounds in the generated images. The downside to this model is it took slightly longer to train and takes twice as long to generate an image.

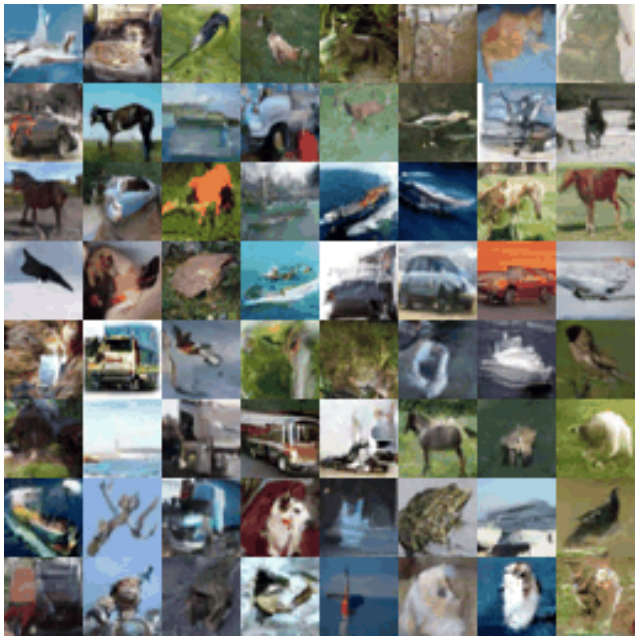


Figure 10. 1000 Time Step Generated Images

4.1.4 2000 Time Step Model

The final model trained for comparing time steps was one with 2000 time steps. The model took 8.51 hours to train for 500 epochs. The model converged fairly quickly, before all 500 epochs so only the first 200 are displayed in Fig. 11. The validation loss converges around 0.0217.

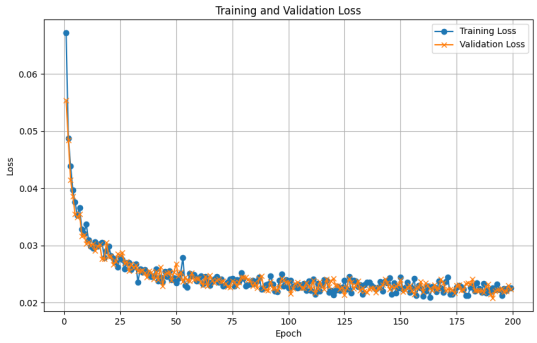


Figure 11. 2000 Time Step Training and Validation Loss Curve

Again, the training and validation loss essentially mirror each other throughout training suggesting that the model is not overfitting. To calculate the FID and Inception Score for the model, 5120 images were generated. The model took 3595 seconds to sample 5120 images. The sampling took almost exactly twice as long as sampling from the 1000 time step model. This is expected as the diffusion process has to go through twice as many steps to generate the image. The

resulting FID score was 27.174. The resulting Inception Score was 7.5891. While the FID and Inception Scores are comparable to the 1000 and 500 time step models; they offer no improvement despite the significantly larger amount of time steps. Fig. 12 displays a sample of 64 generated images using 2000 time steps.

Figure 12. 1000 Time Step Generated Images

The resulting images are once again of high quality, but do not seem to offer any visual advantage over the other two models. There is not a large noticeable visual difference between this model and the models with 1000 time steps and 500 time steps, which is reflected in the similar FID scores and Inception Scores. The downside to this model is it took slightly longer to train and takes significantly longer to generate images than the other models.

4.2. Mixed-Precision Training Study

The second topic in question throughout this project was the effect of mixed precision training on image quality. Mixed precision training and inference offers the benefit of less memory needed and quicker training and sampling. The question explored was if this comes at the cost of the performance of the model. I trained models using two different low-precision data types: float16 and bfloat16. The models were the same size as the models described above and all models were implemented with 2000 time steps. The control model is the 2000 time step model described above. This model was trained with full precision.

4.2.1 Float16 without Gradient Scaler

The first attempt at mixed precision training was using float16 instead of float32. When training this model an issue arose. After the loss initially went down, it began to increase again as seen in Fig. 13.

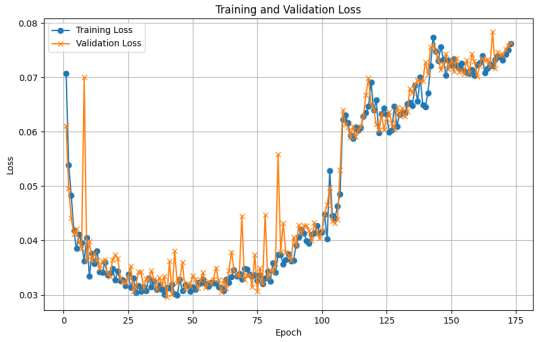


Figure 13. Float16 No Gradient Scaler Loss Curve

This behavior can be explained because of the use of Float16 as the datatype. The Float16 datatype can represent a smaller range of numbers than the Float32 datatype. This can lead to gradient under flow where small gradients cannot be represented and become zero, or gradient overflow where large gradients cannot be represented and become infinity. This causes gradient instability leading to inaccurate training. The way to fix this is to use a gradient scaler.

4.2.2 Float16 with Gradient Scaler

A gradient scaler dynamically adjusts the scale of the gradients during training to avoid instability caused by very small or very large gradient values. Before gradients are computed via back propagation, the gradient scaler multiplies the loss by a scaling factor, S. Scaling the loss increases the magnitude of gradients during back propagation, reducing the risk of underflow. Gradients are computed on the scaled loss. After back propagation, the gradients are divided by the scaling factor S to return them to their original scale. If an overflow occurs, the scaling factor S is reduced. If training is stable, S is increased over time to optimize precision. Using the gradient scaler allows mixed-precision training but at an additional cost required to update the gradient scaler and scale the gradient. While there is an additional cost, using mixed-precision training with a gradient scaler is faster than the baseline model. The model took 7.35 hours to train for 500 epochs. The model converged fairly quickly, before all 500 epochs so only the first 200 are displayed in Fig. 14. The validation loss converges around 0.0224.

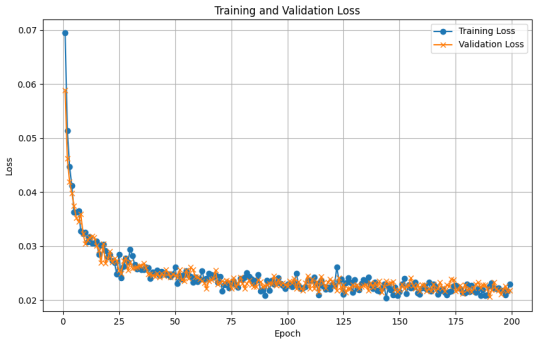


Figure 14. float16 Training and Validation Loss Curve

The training and validation loss essentially mirror each other throughout training suggesting that the model is not overfitting. To calculate the FID and Inception Score for the model, 5120 images were generated. The model took 3101 seconds to sample 5120 images. Sampling with mixed-precision offers a significant speed up compared to the baseline model. The resulting FID score was 29.379, slightly higher than the baseline model. The resulting Inception Score was 7.5263, also slightly higher than the baseline model. The FID Score and Inception Score suggest that the reduced precision does not come with the cost of reduced image quality. Fig. 15 displays a sample of 64 generated images using 2000 time steps and float16 precision.

Figure 15. float16 Generated Images

The resulting images are comparable to the baseline model's generated images as suggested by the similar FID and Inception Scores. This suggests that it is beneficial to train and sample with reduced precision as it does not appear to come at the cost of reduced quality.

4.2.3 bFloat16 without Gradient Scaler

Finally, I trained a model using the bfloat16 datatype. The bfloat16 datatype is capable of representing a larger range of numbers with lower precision than the float16 datatype due to its increased number of exponent bits and fewer mantissa bits. Fig. 16 depicts this difference.

Figure 16. bfloat16 vs float16 vs float32

Using bfloat16 instead of float16 has the advantage that it does not require a gradient scaler in this application because of the larger range of numbers it can represent. This comes at further decreased precision but provides faster training because the gradient scaler does not need to be updated. The model took 6.70 hours to train for 500 epochs, the fastest of all models. The model converged fairly quickly, before all 500 epochs so only the first 200 are displayed in Fig. 17. The validation loss converges around 0.0213, also the lowest of all models.

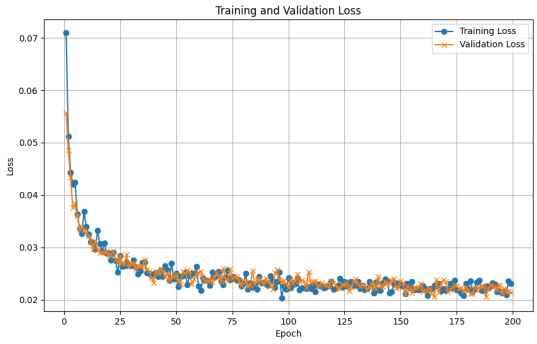


Figure 17. bfloat16 Training and Validation Loss Curve

The training and validation loss essentially mirror each other throughout training suggesting that the model is not overfitting. To calculate the FID and Inception Score for the

model, 5120 images were generated. The model took 3126 seconds to sample 5120 images. The resulting FID score was 25.889, slightly lower than the baseline model. The resulting Inception Score was 7.5745, slightly higher than the baseline model. The FID Score and Inception Score suggest that the reduced precision using bfloat16 does not come with the cost of reduced image quality. Fig. 18 displays a sample of 64 generated images using 2000 time steps and bfloat16 precision.

Figure 18. bfloat16 Generated Images

The resulting images are comparable to the baseline model's generated images and the float16 model's generated images as suggested by the similar FID and Inception Scores. This suggests that training with bfloat16 is beneficial as no gradient scaler is required providing an additional speed up with no reduction in quality.

5. Conclusion

This project offered an in-depth learning experience focused on the details of the implementation of diffusion models. Additionally, it provided insight into the effect of the number of time steps in a model and training and sampling with reduced precision.

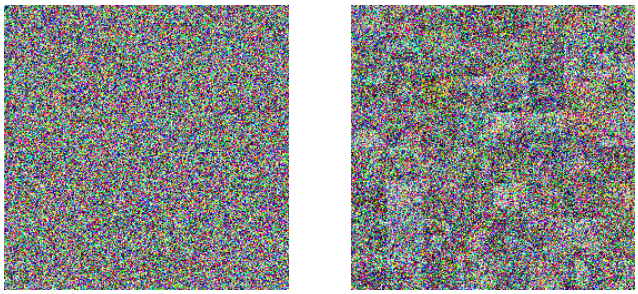
5.1. Time Step Analysis

Four models with different time steps were trained and compared. Table 1 summarizes the results from all four models.

Time Steps	FID Score	IS	Training Time (hr)	Sampling Time (s)	Val Loss
100	72.307	4.9667	**6.78**	**217**	0.0978
500	26.635	**7.7636**	7.22	938	0.0452
1000	**25.517**	7.5891	7.71	1861	0.0319
2000	27.174	7.4758	8.51	3595	**0.0217**

Table 1. Comparison of models with different numbers of time steps. Metrics include FID Score, Inception Score, Training Time, Generation Time, and Final Validation Loss.

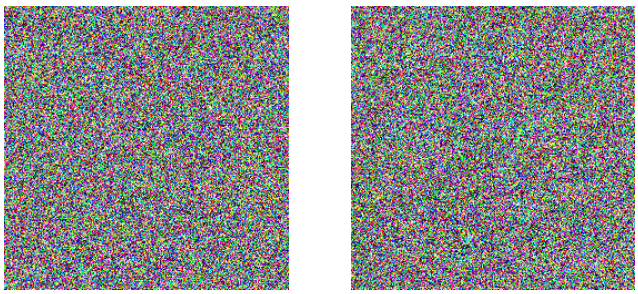
Though the 100 time step model offered quick training and inference, it resulted in poor image quality. The 500 time step model offered a balance between high quality images and lower training and generation time. The 1000 time step model offers perhaps very small image quality improvements from the 500 time step model but at increased training and sampling time.



(a) Time Step 1000 (b) Time Step 500

Figure 19. Comparison of Time Steps in 1000 Time Step Model

Fig. 19 shows the intermediate denoising steps of the 1000 time step model. It is clear that at time step 500, the model is beginning to develop some structure compared to time step 1000. This suggests that using 1000 time steps may be slightly beneficial to the quality of the image generation as it has longer to begin to develop structure.



(a) Time Step 2000 (b) Time Step 1000

Figure 20. Comparison of Time Steps in 2000 Time Step Model

The model with 2000 time steps while reaching a lower

validation loss, does not result in a better FID Score or Inception Score and does not provide visually better images. As seen in Fig. 20, even after the first 1000 denoising steps, the noise just moves around, no structure begins to form. This suggests that there is no benefit when increasing the number of time steps above 1000 for the CIFAR-10 dataset. Specifically for the CIFAR-10 dataset with 32×32 images, it appears that the optimal number of time steps is around 500. This provides relatively quick training and sampling with high quality results. For other more complex datasets, increasing the number of time steps is likely useful. In order to push the performance with this dataset, increasing the size of the model while keeping the number of time steps around 500 would be the best use of resources.

5.2. Mixed-Precision Training Analysis

Three different levels of precision were considered during the training: float32, float16, and bfloat16. The model trained with bfloat16 offered advantages over the other data types because of its ability to train quickly without a gradient scaler.

Prec Level	FID Score	IS	Train Time	Sample Time (s)	Val Loss
float32	27.174	7.475	8.51	3595	0.0217
float16	29.379	7.526	7.35	**3101**	0.0224
bfloat16	**25.889**	**7.574**	**6.70**	3126	**0.0213**

Table 2. Comparison at 2000 time steps using different precision methods. Metrics include FID Score, Inception Score, Training Time (hrs), Generation Time, and Final Validation Loss.

Given Table 2 the model trained using bfloat16 appears to outperform the other two models in nearly every respect. The lower validation loss, lower FID score, and higher Inception Score can likely be attributed to some randomness throughout the training. However, the table does confirm that the reduced precision does not reduce the quality of the generated images. Reducing precision does provide a significant training speed-up as well as a sampling speed-up. These results suggest that using mixed-precision training with bfloat16 can be very beneficial for training diffusion models and machine learning models in general.

References

[1] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis. *arXiv preprint arXiv:2105.05233*, 2021. 1

[2] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2021. 1

[3] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *arXiv preprint arXiv:2112.10752*, 2022. 1

[4] Giulio Franzese, Giulio Corallo, Simone Rossi, Markus Heinonen, Maurizio Filippone, and Pietro Michiardi. Continuous-time functional diffusion processes. *arXiv preprint arXiv:2303.00800*, 2023. 1

[5] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *arXiv preprint arXiv:2006.11239*, 2020. 1

[6] Omer Bar-Tal, Hila Chefer, Omer Tov,Charles Herrmann,Roni Paiss,Shiran Zada,Ariel Ephrat,Junhwa Hur,Guanghui Liu,Amit Raj,Yuanzhen Li,Michael Rubinstein,Tomer Michaeli,Oliver Wang,Deqing Sun,Tali Dekel,Inbar Mosseri. Lumiere: A space-time diffusion model for video generation. *arXiv preprint arXiv:2401.12945*, 2024. 1