CS 682 Final Project Report - Optimizing Diffusion Models for MNIST

Alex Peterson
acpeterson@umass.edu

Efran Himel
ehimel@umass.edu

Anvita Patel
anvitapatel@umass.edu

Abstract

Image generation has become an increasingly popular machine learning task. There are various techniques and applications designed to generate images from scratch, one of which are diffusion models. These models are deep neural networks trained on large datasets which can hinder their deployment on smaller more specialized systems with resources constraints. We explore two diffusion techniques, NCSN and DDPM, to evaluate the quality of their image generation based off of the MNIST dataset and then apply three optimization techniques to evaluate the performance trade offs with model size and inference time. From our analysis, we have determined that optimization techniques, applied appropriately, do no significantly harm NCSN and DDPM model performance, and, in some instances, can improve image creation quality. This is especially true for DDPMs which can produce better results when passed through Depthwise Convolutional optimization compared to no optimization at all.

1. Introduction

In recent years, diffusion models have risen to prominence as an effective way to address image generation tasks compared to other techniques such as Generative Adversarial Networks (GANs). The success and notoriety of models such as Stable Diffusion and DALL-E have sparked public interest and lead to a growing need for image generation in a variety of use cases. Currently, most diffusion tasks rely on connecting to large models with access to large quantities of compute via the internet, but our team anticipates a need to deploy such models in a variety of environments, some of which will have hardware limitations. The goal of this project is to explore diffusion models and refine those models to gain experience in diffusion and model optimization.

1.1. Problem Statement

The main problem our team is attempting to tackle is:

- Can a trimmed down and more optimized diffusion model effectively operate without compromising too much performance in image generation?
- Can these optimized models exceed the performance of the baseline models?

2. Results Summary

From our preliminary results, we have determined that diffusion models can indeed capitalize on optimization techniques to reduce model size and maintain or exceed the original model's performance.

Specifically, we created two baseline diffusion models, NCSN and DDPM, and applied multiple optimization techniques. We found our baseline NCSN model improved image generation accuracy (FID Score) by about 18% and overall model size by 66% when Depthwise Convolution optimization was applied with group size of 2. Furthermore, our DDPM model saw a 13% increase in image generation accuracy and and a 26% decrease in model size when Depthwise Convolution optimization with group size of 2 and global L1 Unstructured pruning with a dropout percent of 10 was applied. We did not see significant changes training and inference times from our optimization techniques.

Overall we found our DDPM models to be better at accurately creating MNIST images compared to our NCSN model. Our optimizations provided noticable improvements to the DDPM model while reducing model size. We applied the best DDPM architecture on the CIFAR-10 data set to see how well it generalizes on a different image training set. The results can be viewed at the end in the conclusion section

2.1. Procedure Overview

We first produced two baseline diffusion models and recorded their metrics regarding model size, training time, and inferencing speed. The two baseline models were a NCSN model and a DDPM model, both of which were trained on the MNIST dataset. Then we created an FID module to produce FID scores for our models which serve as our image generation accuracy metric. We use this score to see how closely are models produce images similar to the original MSIST data set. This is done by making our models produce a random sample of outputs which the FID

algorithm then compares to a random sample of outputs from MNIST to determine the fake images similarity to the MNIST images. The other evaluation metrics we used, as mentioned previously, for all our models are model size, training time, and inferencing speed. After getting our baseline metrics, we applied three optimization techniques, quantization, pruning, and depthwise convolution, to the baselines to produce a variety of optimized diffusion models. Two of the optimizations required retraining the entire model, one technique did not require retraining and was applied after training. We recorded our evaluation metrics to the optimized models and compared it with the baseline to determine how the optimization techniques improved or harmed our baseline models. Lastly, we trained the best model architecture on the CIFAR-10 dataset to see how well that architecture generalizes image production on a more complicated diffusion task.

3. Technical Approach: Developing Baseline

We initially began our work by researching Denoising Diffusion Probabilistic Models (DDPM), Score-Based Generative Models (SGM), and Noise-Conditioned Score Networks (NCSN) individually. From our research we realized NCSNs are a subset of SGMs thus chose DDPM and NCSN as our two initial models. We chose NCSN because it seemed to be a refinement on SGM and we chose DDPM because it wasn't a subset of SGMs. We did notice from our research that it was claimed SGMs and DDPMs were very similar so we expected our results to be quite similar as well. To start out, we found various resources that provided good starting points for implementing NCSNs and DDPMs. We elaborate on our implementation and results experimenting with these models below.

3.1. Benchmarks Two Baseline Models

For each, we detail the network architecture and implementation, the training and inference time benchmarks, and the model size in terms of number of parameters and bytes on disk. For our baseline we trained DDPM and the NCSN model on the MNIST dataset.

3.1.1 Noise-Conditioned Score Network

Score-based generative models (SGM) aim to directly learn the gradient of the data distribution and they are trained to assign high scores to observed data and low scores to generated samples. NCSNs are a subset of SGMs that use noise conditioning to improve the modeling of complex data distributions. We evaluated an implementation of a NCSN from [10], with all results collected in a Google Colab environment with access to a single NVIDIA T4 GPU.

Architecture

1. Embed the input time steps using a Gaussian Fourier Projection
2. A set of four consecutive encoding layers for the input image consisting of Conv2D, add Linear projection of embedded time steps, Group Norm layers followed by a swish activation where we increase the channels and decrease the resolution
3. A set of three decoding layers consisting of a ConvTranspose2d, add Linear projection of embedded time steps, group norm layer and a swish activation to increase the resolution.

3.1.2 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models (DDPM) perform diffusion in two phases, a forward noise phase and a reverse diffusing phase. In the first phase, DDPM gradually applies gaussian noise to an image over multiple steps until the image is complete noise. The noising process is actually reversible, thus during the reverse diffusing phase, a neural network is trained on gradually denoising the noised image until a new clear image is produced. Classifier Free Guidance (CFG) is also implemented in this DDPM for slightly better image quality [5]. This is done by embedding data labels in the forward and backward phases. Essentially, in CFG, the model is allowed to generate any image (classifier free) unconditionally. During the start of training, the CFG allows 10 percent of training to be classifier free. In every iteration the CFG linearly scales until all models are generated conditionally on the classifier. This model was trained and tested in a Google Colab environment on a V100 GPU and initially was built by following a tutorial on github [4].

Architecture

1. First we apply noise to the images until it is complete noise. Normally this is done in multiple iterate steps but the iteration can be simplified to a single step.
2. Then we use a UNet which will downsample the image, pass it through a 2 layer convolutional neural network, and then upscale the results back to the original image size. A UNet is comprised of a downsampler, a bottleneck, and an upscaler.
   (a) For the first part of the unit, the downsampler, is comprised of a double convolutional layer followed by a down sampler and self attention layer which repeats 3 times.
   (b) Then the results go through the bottleneck which is 3 double convolutional layers.

(c) Lastly, the bottleneck results go through the upscaler which is the opposite of the downsampler. So 3 Up sample and self attention layers with a double convolutional net at the end. After, the result is projected back to the original inputs

3. For training, we iterate through each epoch. During each epoch we get a sample of our diffused noise image, apply the model (UNet) to predict and denoise the image, then calculate the loss to train our model and network.

3.2. Evaluation Metrics

As mentioned previously and shown in the metrics sections below, we keep track of the test time for our produced models as well as the size of the models in terms of bytes and number of parameters. We will keep these as a baseline and see if we can reduce the model size without compromising generation quality when during the optimization phase.

3.2.1 Metrics for NCSN Baseline

Training Benchmark
To train the model, we define a loss function. Train time observed for the MNIST data set with 50 epochs using a batch size of 32 was: 1300 seconds (21 minutes).

Inference Benchmark
To generate samples, we sample from the NCSN using one of three methods to solve for reverse-time-SDE: Euler-Maruyama, Ordinary Differential Equations Sampler or Predictor-Corrector Sampler. A summary of the inference times for each of the three sampling methods can be found below:

Sampler	Samples	Total Time	Avg Time
Euler Maruyama	128	4896ms	38ms
ODE Sampler	128	7381ms	58ms
Predictor-Corrector	128	10370ms	81ms

Model Size
The model we trained used 1,115,297 parameters and occupied 4,472,986 bytes on disk.

3.2.2 Metrics for DDPM Baseline

Training Benchmark
To train the model, we define a loss function and use MSE for scoring. Train time observed for the CIFAR-10 data set with 50 epochs using a batch size of 16 was approximately 8hrs.

Inference Benchmark
A summary of the inference time for DDPM using CFG is shown below:

Type	Samples	Total Time	Avg Time
CFG	130	812607ms	6250ms

Model Size
The model we trained used 23,335,299 parameters and occupied 91 mega bytes on disk.

4. Technical Approach: Optimizing Models

After creating the baseline models, we developed an FID score module that we would use to score and compare our diffusion results. Then we explored applying three optimization techniques to those models to tackle our problem statement. The three optimization techniques we applied to the baselines were quantization, pruning, and depthwise convolution.

Note: For the following results table

1. **Size**: Size of compressed model in bytes
2. **Training**: Model training time in ms per epoch
3. **Inference**: Model inference time in ms per image

4.1. Fréchet Inception Distance

We utilized the Fréchet Inception Distance (FID) [6] metric to assess the quality of the generated images from the models. An FID value of zero indicates a perfect match between the real and generated images. The comparison between the real dataset and the images generated by the baseline NCSN model resulted in a FID value of 182.1. The comparison between the real dataset and the images generated by the baseline DDPM model resulted in a FID value of 150.4.

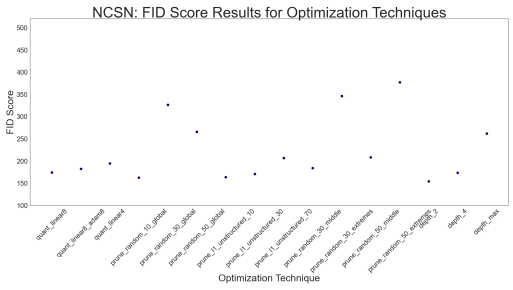

Figure 1. NCSN Optimization Techniques FID Scores

Although, DDPM model performed better with the optimization technique, there was more variability in the scores compared to the NCSN model.

4.2. Optimization Methods

4.2.1 Quantization

Quantization is an optimization technique that can take a big model and reduce it's size and speed up its inference time. Essentially it does this by converting the the size of weights in a network into smaller bytes, thus taking up less

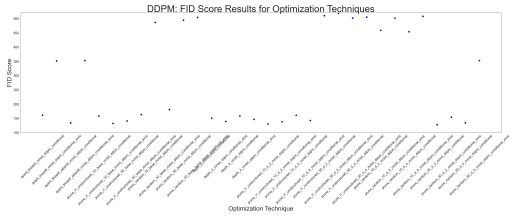

Figure 2. DDPM Optimization Techniques FID Scores

space for the neural network. Rounding out the weights may reduce model size and inference time but can potentially lead to performance loss, we will be observing if this loss is significant.[7].

We used the bitsandbytes library [2][3] to apply quantization in our models. From bitsandbytes we used 4bit and 8bit quantization linear layers and the AdamOptimizer and AdamWOptimizer modules. In our models we replaced the regular linear layers and optimizer in our model architecture with the ones mentioned from bitsandbytes then retrained on models to record the results.

4.2.2 Pruning

Pruning is the process of zeroing out weights in a network. We used several PyTorch pruning methods to zero out portions of the model's weights. In order to see the benefits of this procedure, we compressed the state dictionaries of the models to calculate are space savings size.

Inference time does not improve because the model cannot take advantage of the sparsity of the matrix. Some effort was spent attempting to incorporate libraries that enable sparse matrix calculations in PyTorch but several obstacles prevented their implementation. While a huggingface implementation of SparseLinearMatrix exists, there was no implementation for Conv2d or ConvTranspose2d layers.

We experimented with two different unstructured pruning methods: Random Unstructured and L1 Unstructured. Unstructured pruning refers to the process of replacing elements within parameter matrices with 0 value. This removes the influence of the element on the parameter output, but does not change the dimensions of the parameter matrix. Structured pruning reduces the dimensionality of the matrix itself, which we did not experiment with.

In random unstructured pruning, a fixed proportion of randomly selected elements are removed from the parameter matrix. In l1 unstructured pruning a fixed proportion of elements with the lowest l1 norm are removed from the parameter matrix.

In addition to global applications of the pruning, we also experiment with local applications. Using the insight that our model conforms to the U-Net architecture [8] that is widely used in diffusion settings, we apply pruning to the "inner" section of the U for our NCSN model.

The following list shows each module in U-Net in order. Modules in **bold** were designated "middle modules", while the rest were designated "end modules".

Since the channels grow in the middle of the unit, the activation volumes grow larger. This means that while the number of modules in each category are roughly the same, there are about 9x the number of parameters in the middle as compared to the ends. In total, there are 942,592 elements in the middle modules and 105,505 elements on the end.

1. module.conv1.weight: 288
2. module.dense1.dense.weight: 8192
3. module.dense1.dense.bias: 32
4. module.conv2.weight: 18432
5. module.dense2.dense.weight: 16384
6. module.dense2.dense.bias: 64
7. **module.conv3.weight: 73728**
8. **module.dense3.dense.weight: 32768**
9. **module.dense3.dense.bias: 128**
10. **module.conv4.weight: 294912**
11. **module.dense4.dense.weight: 65536**
12. **module.dense4.dense.bias: 256**
13. **module.tconv4.weight: 294912**
14. **module.dense5.dense.weight: 32768**
15. **module.dense5.dense.bias: 128**
16. **module.tconv3.weight: 147456**
17. module.dense6.dense.weight: 16384
18. module.dense6.dense.bias: 64
19. module.tconv2.weight: 36864
20. module.dense7.dense.weight: 8192
21. module.dense7.dense.bias: 32
22. module.tconv1.weight: 576
23. module.tconv1.bias: 1

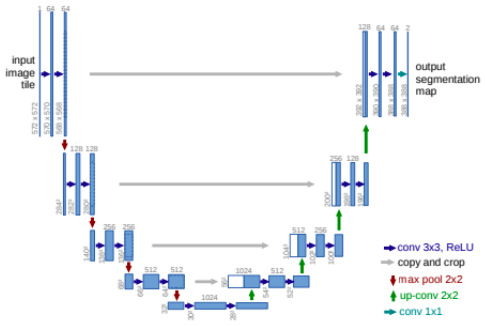

Figure 3. Example of U-Net Architecture [9]

4.2.3 Depthwise Convolution

Depthwise Convolution is a method for reducing the number of parameters in a convolution (or transpose convolution) layer. It achieves this by splitting the the filters in a convolutional layer. Typically, without depthwise convolution, 1 filter is applied to all channels in a convolutional layer. With Depthwise Convolution, the input is split up into n groups, and n filters are applied to a single group. This is no different than splitting the inputs into and filters to n groups, performing separate convolutions, and then concatenating the result. This results in fewer parameters across the convolutional layer thus reducing they layer's size.[1]

Figure 4. Depthwise Convolution

We applied depthwise convolution by using the built in support in PyTorch's Conv2d and TransposeConv2d classes. We modified our model architecture to take in a group variable during any Conv2d or TransposeConv2d step and retrained our entire model with group = 2 and group = 4.

We experimented with using Depthwise Convolutional layers globally across our score model. The group parameter has the restriction that it must divide the $out_channel$ argument evenly, so the smallest value that may be used is 2 and the largest is $min(in\_channel, out\_channel)$. We experiment with both these extremes.

4.3. NCSN Optimization Results

4.3.1 Quantization Results

In the table below, we summarize the results of quantization using 4bit Linear Layers and Adam optimizers. We replaced the Linear layers and Adam optimizer with the 4 and 8 bit versions and retrained our score model using the same procedure as our baseline. For each of the three settings, we report the serialized model size after compression, the total elapsed training time, inference time for a single generation and FID scores.

Technique	Size	Inference	FID
4BitLin	4146973	24.726	174.099
8BitLin	4147464	24.631	182.233
8BitAdam	1387.4	24.831	194.643

Table 1. Quantization Results

Training Time: 4BitLin 1364ms/epoch

Training Time: 8BitLin 1332ms/epoch
Training Time: 8BitLin + 8BitAdam 1387ms/epoch (50 epochs)

4.3.2 Quantization Observations

Surprisingly, the lowest bit quantization method yielded the lowest FID score. Lowest inference and training speeds were observed for the model that leveraged 8Bit Linear layers.

4.3.3 Pruning Results

The following table summarizes results for 10 pruning settings. We experiment with random and l1 global pruning at 10, 30, 50 and 70 percent. We experiment with random pruning in selected layers of the model at 30% and 50%. We report the compressed model size, the number of parameters and the FID score.

Technique	Size	Inference	FID
Rand 10 Global	4,021,665	24.85	162.357
Rand 30 Global	3,676,115	24.94	326.117
Rand 50 Global	2,596,545	24.96	265.807
L1 10 Global	4,002,980	25.06	163.601
L1 30 Global	3,639,353	25.03	170.784
L1 70 Global	2,781,940	25.11	206.551
Rand 30 Middle	3,775,050	25.35	183.64
Rand 30 Extremes	4,097,460	25.20	346.27
Rand 50 Middle	3,456,117	25.45	208.063
Rand 50 Extremes	4,054,127	25.49	377.001

Table 2. Pruning Results

4.3.4 Pruning Observations

The lowest FID score of 162.357 observed for the model produced by randomly pruning 10 percent of parameters across all modules. However we see the penalty for pruning more weights using the L1 method is much more permissive. Indeed, we are able to prune 70% of weights globally and observe a lower FID score than randomly pruning 50% of weights.

Our experiments in selectively pruning modules in the middle and extremes of the the U-Net architecture offer encouraging results. We see that pruning 30% and 50% of weights in the middle layers only produce better FID scores (183.64 and 208.063 respectively) than randomly pruning 30% and 50% globally (326.117 and 264.807 respectively). Interestingly, we see much higher FID scores when we prune only the extremes of the model. This suggests that the weights on the extremes *are more important to the quality of generations* than weights in the middle.

A future route to explore might be to increase the percentage of weights that we prune as we iterate across modules that are closer to the center of the architecture. Additionally, experimenting with L1 pruning using such a percentage schedule might yield even more promising results.

4.3.5 Depthwise Convolution Results

We experimented with using Depthwise Convolutional layers globally across our score model. The group parameter has the restriction that it must divide the $out_channel$ argument evenly, so the smallest value that may be used is 2 and the largest is $min(in\_channel, out\_channel)$. We experiment with both using 2, 4 and the maximum allowed value for each layer. We report the compressed model size in bytes, the inference time and the FID score.

Technique	Size	Inference	FID
2 Groups	2,537,197	33.35	154.340
4 Groups	1,731,476	17.38	173.295
Maximum Groups	965,473	25.60	261.493

Table 3. Quantization Results

Training Time: 1399ms/epoch (50 epochs)

4.3.6 Depthwise Convolution Observations

We observed, predictably, that the quality of generation degrades as we use more groups. However it is notable that we are able to achieve smaller models and better FID scores than pruning. We achieved an FID score of 154 for depthwise convolution with two groups which yielded a model with 682k parameters. In comparison, random 50% at extremes yields a model of similar number of parameters at 644k but produced an FID score of 208. This suggests that reducing the parameter counts using Depthwise Convolution is a more efficient approach to reducing total number of parameters then pruning a trained model.

4.4. DDPM Optimization

Before we began optimizing the DDPM model, we also implemented Exponential Moving Averages (EMA) into the model. EMA applies a moving average to the learning process, biasing new data over old data points when updating the gradient. We included this because during additional research we found it yielded positive results when used with CFG. DDPM now had two baseline models, CFG and CDF + EMA, which were being optimized.

Training Time: ∼510000ms/epoch (50 epochs)

Technique	Size	Inference	FID
CFG	∼66538096	∼5400	150.38
CFG + EMA	∼66538096	∼5400	142.88

Table 4. Baseline Model Results

4.5. DDPM Optimization Results

The following are the optimization results for the DDPM model with CFG and CFG + EMA. The quantized models were trained on a A100 GPU for 50 epochs at a batch size 40 while the other models were trained on a V100 GPU at batch size 16.

4.5.1 Quantization Results

Technique	Size	Inference	FID
8BitLin	∼66244000	∼5400	161.22
8BitLin + 8BitAdam	∼415000	∼5400	134.34

Table 5. Quantization Results: CFG

Training Time: ∼404000ms/epoch (50 epochs)

4.5.2 Quantization Observations

The table above shows that quantization can produce a slight decrease in model size without compromising on the FID score. Inference time is unchanged. The training time is lower but we believe this is due to using a larger batch size (40 per batch instead of 16) when training the model. We did this because bitsandbytes is restricted to certain GPUs, thus we had to use the A100 instead of the V100. The A100 has more VRAM, thus we could use bigger batches for training, this is why training time per epoch is lower.

Quantization with CFG + EMA was not recorded as model resulted in only black images being produced.

4.5.3 Pruning Results

Technique	Size	Inference	FID
Rand 10 Global	∼62981643	∼5300	486.27
Rand 30 Global	∼53349095	∼5200	494.35
L1 10 Global	∼62448687	∼5400	157.89
L1 30 Global	∼52029548	∼5300	141.13
Rand 10 Global + EMA	∼62982310	∼5200	181.53
Rand 30 Global + EMA	∼53349687	∼5200	504.45
L1 10 Global + EMA	∼62448687	∼5400	132.22
L1 30 Global + EMA	∼52030922	∼5200	163.77

Table 6. Pruning Results: CFG and CFG + EMA

4.5.4 Pruning Observations

Unlike the NCSN model, the DDPM models only had global pruning applied during optimization. The random middle, random extreme methods, 50 and 70 percent pruning were not used. Since pruning can be applied globally after a model has been trained, there is no training time overhead.

From the tables above, L1 Pruning maintains a relatively similar FID score to the baseline and quantized model while also being a smaller model size. Random pruning on the other hand greatly degrades the models FID scores, thus we don't believe its a good optimization technique for this architecture. Although L1's pruning seems promising, as mentioned previously, the space saving advantage is mostly theoretical since the weights still exist in the model. Inference time was slightly improved with all the models.

4.5.5 Depthwise Convolution Results

Technique	Size	Inference	FID
2 Groups	∼52203120	∼5300	150.90
4 Groups	∼45030932	∼5300	157.93
2 Groups + EMA	∼52203120	∼5500	139.29
4 Groups + EMA	∼45031920	∼5300	146.77

Table 7. Depthwise Convolution Results: CFG and CFG + EMA

Training Time: ∼510000ms/epoch (50 epochs)

4.5.6 Depthwise Convolution Observations

Depthwise Convolution with group size 2 and 4 both reduce the model by a good amount, about ∼30 percent and ∼40 percent respectively, while maintaining a relatively similar FID scores to the baseline DDPM model.

CFG + EMA paired with this optmization technique fairs better than just only a CFG model. Also, having 4 groups instead of 2 groups hurts the FID score a bit more which makes sense due to there being fewer weights created with 4 groups. Its results are similar to pruning with L1 Globaly and is slightly better than only quantization regarding good FID score and reduced model size.

4.5.7 Depthwise Convolution + Pruning

Because Depthwise Convolutions yielded a nice results, we lastly prune the depthwise optimized models and show the top best ones below.

Surprisingly we gain a lower FID score compared to the base while reducing the model size when combining Depthwise with random and l1 pruning.

Technique	Size	Inference	FID
2 Groups + Random 10	∼41978895	∼5500	127.74
2 Groups + L1 10	∼49014554	∼5500	130.44
4 Groups + Random 30	∼36269068	∼5400	134.35
2 Groups + L1 10 + EMA	∼49015306	∼5400	138.15

Table 8. Depthwise Convolution + Pruning Results

5. Conclusion

After analyzing our results, we have answered our initial problem statement and determined that you can optimize diffusion models without hurting performance. In fact, after our optimizations, we have shown that performance gains regarding FID score can be made. We proved this by producing multiple optimized models which have lower model sizes but greater FID scores compared to our original baseline models. We believe this is because some of the optimization models removed unnecessary noise from the models, thus making them more accurate and smaller.

Regarding our other two evaluation metrics, training time and inference time, we observe that our optimization techniques did not have significant impact in reducing these two metrics compared to the baseline. This indicates that if inference and training time are key metrics for a diffusion task, other optimization techniques may need to be considered over quantization, pruning, and depthwise convolution.

Of the three optimization techniques, depthwise yielded the best result for NCSN with a group size of 2. Despite this, increasing the group size significantly hurt the model. Meanwhile, quantization and pruning with L1 unstructured yielded more consistent results which were close to depthwise with group size 2. This implies for NCSN models, depthwise convolution may potentially offer greater performance benefits at the cost of model sensitivity while quantization yields and pruning yields safer more consistent results.

Of the three optimization techniques, depthwise convolution and L1 pruning yielded the best results for DDPM. The other optimization technique configurations had higher variance in improving/harming the DDPM model. Unlike NCSN, increasing the group size from 2 to 4 in depthwise convolution did not significantly harm the DDPM model, which may indicate our implementation for DDPM is more robust than NCSN. DDPM with CFG and CFG + EMA both yielded similar results regardless of optimization except for in quantization where EMA greatly harmed the model and resulted in blank or noisy images. Surprisingly, combining depthwise convolution with pruning yielded some of the best results.

From our observations, we have determined our DDPM model performs better than our NCSN model in most cases. This is surprising because in the beginning we assumed they

would yield similar results. This may indicate that during our initial implementation of our baselines, DDPM may have been implemented better than NCSN. Another reason we determine our DDPM models to be better is that our DDPM had CFG implemented. This allowed the model the option of generated random samples and generated images based on labels passed to the model. Unlike DDPM, our NCSN can only generate a random sample of images. In the future we may implement CFG into NCSN to see if it yields similar results to DDPM. The last reason our DDPM are better is because they are image size and channel agnostic while our NCSN model can only take in images with 1 color channel.

5.0.1 Best Model Results and Comparisons

Our best NCSN model was NCSN optmized with depthwise convolution at a group size of 2. Our best DDPM model was DDPM + CFG model with depthwise convolution with a group size of 2 and pruned through an L1 Unstructured pruning method with dropout chance of 10%. Below is a table and image comparison of best models with their baseline:

Technique	Size	Inference	FID
Base NCSN	∼4147037	∼25.60	182.1
Best NCSN	∼2537197	∼33.35	134.35
Base DDPM	∼17868145	∼5300	146.77
Best DDPM	∼12668288	∼5500	130.44

Table 9. Quantization Results

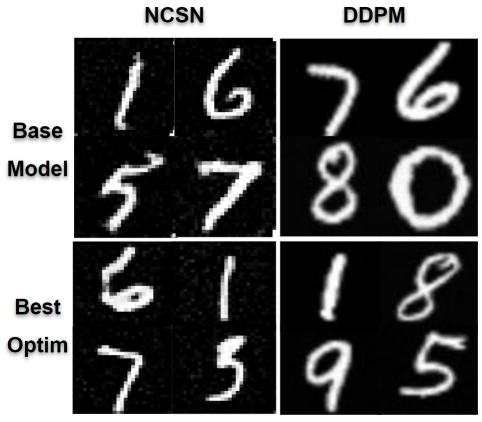

Figure 5. NCSN and DDPM Base and Best Models

5.0.2 Applying to CIFAR-10

After determining our best models, for NCSN and DDPM, we determined overall that the DDPM + CFG model with depthwise convolution with a group size of 2 and pruned through an L1 Unstructured pruning method with dropout chance of 10% was the best model. As a final step, we were curious to see how this model architecture generalizes to a more diverse and complicated image set, so we trained this exact model on the CIFAR-10 data set. Below are our results in the order from top left to bottom right: truck, horse, dog, plane, car, bird, boat, deer, boat, bird

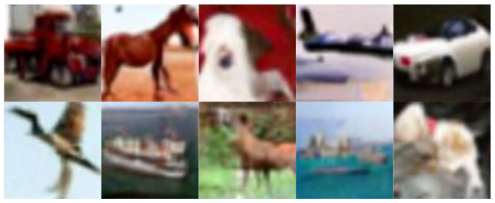

Figure 6. CIRFAR-10 Generated Images

Given that we simply plugged the architecture into a new data set, we are pleasantly surprised by the results.

References

[1] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2017. 5

[2] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022. 4

[3] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization, 2022. 4

[4] dome272. Diffusion-models-pytorch, 2022. 2

[5] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance, 2022. 2

[6] Håkon Hukkelås. Fréchet inception distance (fid) for pytorch. https://github.com/hukkelas/pytorch-frechet-inception-distance, 2023. 3

[7] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021. 4

[8] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. 4

[9] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. 4

[10] Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *CoRR*, abs/1907.05600, 2019. 2