M2 Applications of Machine Learning

CRSiD: tmb76

University of Cambridge

Contents

1 Training a Diffusion Model **3**
1.1 Denoising Diffusion Probabilistic Model (DDPM) 3
1.1.1 Diffusion Models 3
1.1.2 Denoising Diffusion Probabilistic Model (DDPM) 3
1.2 Training the Model on the MNIST Dataset 5
1.3 Running the Model for Different Hyperparameters 10

2 Custom Degradation Function **14**
2.1 A Row/Column Averaging degradation function 14
2.2 Training the modified model on the MNIST dataset 16
2.3 Unconditional Sampling 18

3 Appendix **20**
3.1 Other versions of the degradation function 20
3.2 REAMDE 22
3.2.1 M2 Coursework - Training Diffusion Models on the MNIST Dataset 22

Using a Diffusion Model on the MNIST Dataset

Chapter 1

Training a Diffusion Model

1.1 Denoising Diffusion Probabilistic Model (DDPM)

1.1.1 Diffusion Models

Diffusion models are a class of probabilistic latent variable models. They consist of an encoder and decoder. The encoder takes the input data and maps it to a latent space in a series of steps, resulting in a series of intermediate latent variables. The encoder is similar to variational autoencoders (VAEs) in that it maps the input data to a latent space. However, the particularity of the encoder here is that the mappings it will apply at each time step are predetermined by a schedule. The key part is that the decoder is trained to learn what is the reverse process of the encoder, predicting back through the series of latent mappings, therefore being able to reproduce the original sample [8, p.348].

1.1.2 Denoising Diffusion Probabilistic Model (DDPM)

In this report, the writing conventions of the Prince textbook will be followed [8]. The model used for this project is a DDPM. For this model, the encoder takes in some input data $\mathbf{x}$ and maps it to a latent space $\mathbf{z_T}$, of the same dimensionality as $\mathbf{x}$, in a series of steps: $\mathbf{z_0} \rightarrow \mathbf{z_1} \rightarrow \ldots \rightarrow \mathbf{z_T}$. This can be defined as a known Markov chain, which at each step adds Gaussian Noise following a Noise or Variance Schedule, $\beta_{1,\ldots,T}$. As it is a Markov Chain, and a type of variational autoencoder, the encoder can be described by an approximate probability distribution q such that [5]:

$$q(\mathbf{z}_{1:T}|\mathbf{x}) = \prod t = 1Tq(\mathbf{z}_t|\mathbf{z}_{1:t-1})) \tag{1.1}$$

Where the individual step is given by:

$$q(\mathbf{z}_t|\mathbf{z}_{t-1}) = \mathcal{N}(\mathbf{z}_t; \sqrt{1-\beta_t}\mathbf{z}_{t-1}, \beta_t\mathbf{I}) \tag{1.2}$$

In other words, β_t describes how much noise is going to be added to the input data at each step t. Prince's textbook also provides a closed form expression which shows this more clearly [8]:

$$\mathbf{z}_1 = \sqrt{1-\beta_1}\mathbf{x} + \sqrt{\beta_1}\epsilon_1 \tag{1.3}$$

And it can be shown that after t steps, this gives:

$$\mathbf{z}_t = \sqrt{\alpha_t}\mathbf{x} + \sqrt{1-\alpha_t}\epsilon \tag{1.4}$$

where $\alpha_t = \prod_{s=1}^{t} 1-\beta_s$ and $\epsilon \sim \mathcal{N}(0, \mathbf{I})$, is a sample from a standard normal distribution, and is the actual noise added. The decoder is trained to learn the reverse process of the encoder, or simply how to go from $\mathbf{z}_T$ to $\mathbf{z}_{T-1}$, continuing back through the latent variables to the input data $\mathbf{x}$. Coming back to the approximate probability distribution q, the decoder is trained to learn the reverse distributions $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$. Approximating them as normal distributions, they can be written:

$$Pr(\mathbf{z}_{t-1}|\mathbf{z}_t, \phi t) = \mathcal{N}_{z_{t-1}}(\mathbf{f}_t[\mathbf{z}_t, \phi_t], \sigma_t^2\mathbf{I}) \tag{1.5}$$

where $\mathbf{f}_t$ is a neural network that takes $\mathbf{z}_t$ as input and has parameters ϕ_t, which here is just the timestep t. The reason why the model predicts the mean of the normal distribution with the variance being fixed to $\sigma_t^2\mathbf{I}$ is discussed in greater detail in the Ho et al. (2020) paper [5]. The training algorithm can then be written as follows:

Training Algorithm for DDPM reverse process [5]

Input: Data $\mathbf{x}$
Output: $\mu_\mathbf{t} = \mathbf{f}_t[\mathbf{z}_t, \phi_t]$
repeat
  for $i \in \mathcal{B}$ **do** ▷ For each training example index in batch
    $t \sim \mathcal{U}(1, \ldots, T)$ ▷ Sample a random time step
    $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ▷ Sample noise
    $L = ||\epsilon - \epsilon_\phi(\sqrt{\alpha_t}\mathbf{x} + \sqrt{1-\alpha_t}\epsilon, \phi_t)||^2$ ▷ Compute individual noise
  end for ▷ Accumulate losses for batch and take gradient descent step
until convergence

One may notice that the algorithm is predicting the noise instead of the mean. This is a result of reparameterizing the network $\mathbf{f}_t[\mathbf{z}_t, \phi_t]$ [5], replacing it with $\hat{\epsilon} = \mathbf{g}_t[\mathbf{z}_t, \phi_t]$, such that [8, pp.361-362]:

$$\mathbf{f}_t[\mathbf{z}_t, \phi_t] = \frac{1}{\sqrt{1-\beta_1}}\mathbf{z}_t + \frac{\beta_t}{\sqrt{1-\alpha_t}\sqrt{1-\beta_t}}\mathbf{g}_t[\mathbf{z}_t, \phi_t] \tag{1.6}$$

1.2 Training the Model on the MNIST Dataset

Here, the model chosen to learn prediction of the noise is a Convolutional Neural Network (CNN). CNN's are often used in image data processing [8, p.161], partly since they provide a way to reduce the number of weights and biases, which becomes an issue quickly in images as they are high dimensional inputs. More importantly, image recognition or prediction requires more of a knowledge of what patterns define certain objects, and this whatever the position on the image. And this is something a fully connected neural network struggles with since it does not have any notion of spatial relationships between pixels, and would need to learn what a certain object looks in every rotation/position possible. This is key in the case of the MNIST dataset where only 9 object types are considered but they are found to be very varied, as they are handwritten.

The activation function used is GELU, which is a Gaussian Error Linear Unit, defined as:

$$\text{GELU}(x) = x\Phi(x) = x \cdot \frac{1}{2} + [1 + \text{erf}(\frac{x}{\sqrt{2}})] \tag{1.7}$$

where $\Phi(x) = P(X \leq x), X \sim \mathcal{N}(0, 1)$. It was introduced in 2016 by Hendrycks and Gimpel [4], and was found to provide better results in computer vision tasks among others(see Figure1.1).

The standard DDPM model was trained for 100 epochs on the MNIST dataset, with a batch size of 128. The model was trained using the Adam optimizer [6] with a learning rate of 2×10^{-4}, and the loss function used was the mean squared error [9].

First, the loss at each iteration was obtained and plotted in Figure 1.2. As can be seen the loss does decrease over time, fast at first then much slower which is expected as the CNN converges towards an MSE minimum.

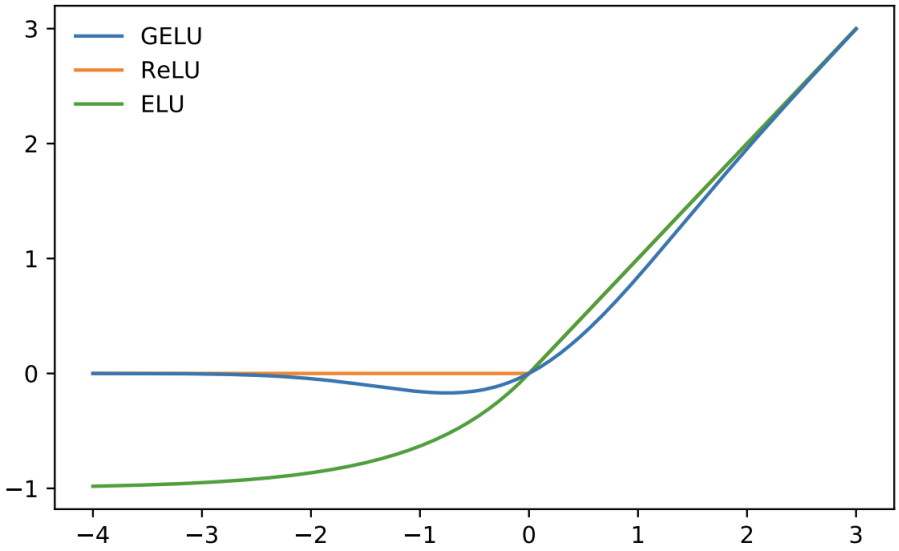

Figure 1.1: GELU activation function (blue) compared to ReLU (orange) and ELU (green)

One issue that arises in the context of the MNIST dataset is that the MSE is susceptible to be very low even though the samples generated are not good. This is because the MSE is purely looking at the problem quantitatively. To get a qualitative sense of the samples generated, 16 samples were generated for different epochs using the sampling algorithm described in the Prince textbook [8, p. 363], by giving the model a pure Gaussian noise input and letting it gradually denoise it, adding some small noise each iteration. The samples generated at epoch 1, 20, 40 and 60 are shown in Figure 1.3.

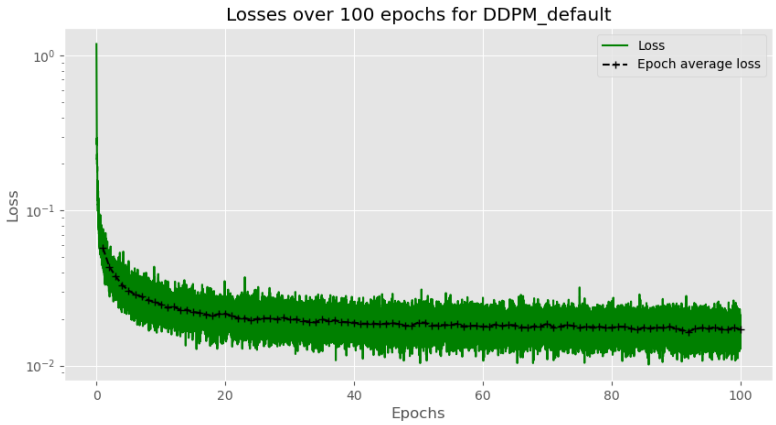

Figure 1.2: Loss as a function of epoch number (green), with the average loss for each epoch overlayed (dashed black)

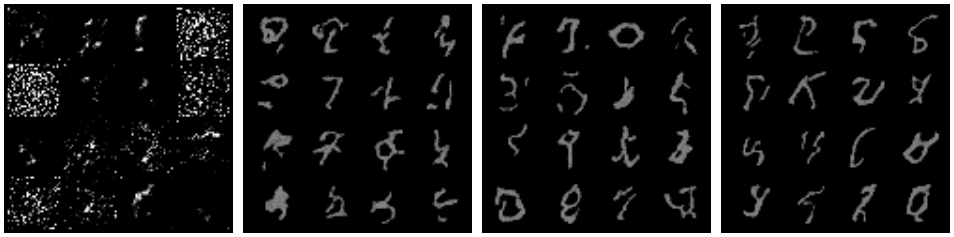

Figure 1.3: Samples generated by the DDPM model at epochs 1, 20, 40 and 60

As expected the samples generated at epoch 1 are very close to being just noise, though some samples show some patterns appearing. By epoch 20, the model is almost consistently generating symbols, with some ressembling numbers, like a 7 in cell (3,2). And it then takes a longer time to get to a point where the samples are consistently numbers. This comes back to the discussion above, since the symbols do result in a low MSE, and the gradient of the loss function is smaller, the model struggles to learn the last step of having the symbols be digits.

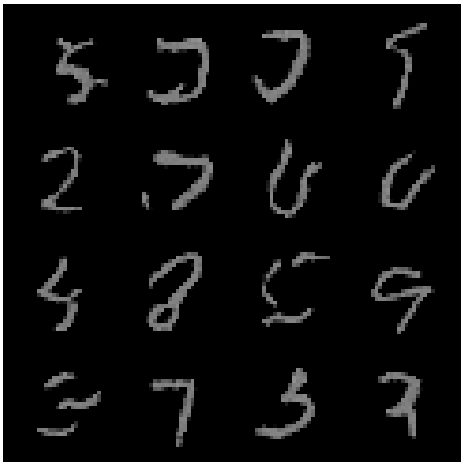

Figure 1.4: Samples generated by the DDPM model at epoch 90

At epoch 90, the symbols look more like digits, though there are still some meaningless symbols being generated (see Fig. 1.4). To quantitatively evaluate the quality of the samples, the Fréchet Inception Distance (FID) score was used. The FID score measures how similar two sets of images are by comparing the statistics of the computer vision feature representations of the images. The features are obtained using the inception v3 model, an image classification model [2]. The lower the score the better. The FID score was calculated for the samples generated at each epoch, and the results are shown in Figure 1.5. A more robust estimate is computed once all epochs are run, on a larger sample of generated images.

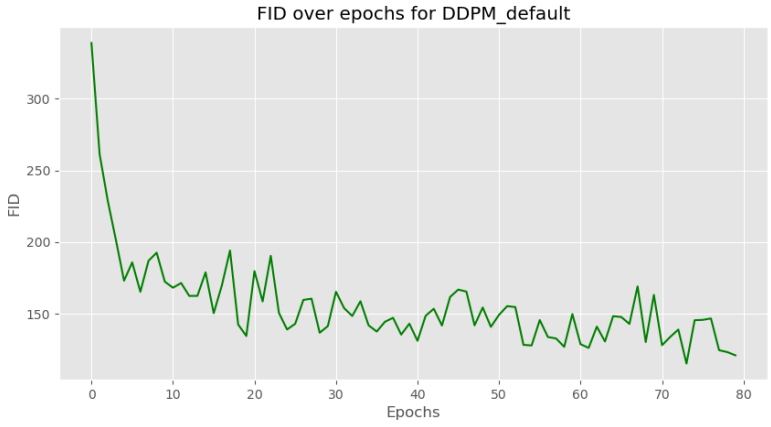

Figure 1.5: FID score at each epoch of the training process

Fig. 1.5 shows the FID score decreasing over time, which would match our expectations based on Figs 1.2, 1.3, and 1.4. The FID score at the final epoch was found to be 98.74.

Comparing to values obtained in the Ho et al. (2020) paper [5], or the Bansal et al. (2022) paper [1], it can be seen that the score obtained here indicates quite bad performance. Moreover, Fig 1.5 shows that a lot more epochs are needed to reach these values, if the FID continues decreasing. However, it is important to note that the model was trained for only 100 epochs, with quite a shallow CNN. Further, the FID here was used regardless of the digit, which may be different to in the papers mentionned. The rationale here was that with the FID score being based on the **distribution** of computer vision features [2], it should still be able to capture the quality of the samples generated when using it for all digits together. Finally, it is not a perfectly objective metric so it will, mainly be used for comparison of the different models trained in this report. In that aspect, it will be a more robust metric for analysis.

Additionally, the Inception Score (IS) was calculated for the samples generated at each epoch. The IS is a metric that measures the quality of generated images, by looking at the diversity and quality of the generated images [7]. More specifically, and in this context, it will measure the variety of images/digits generated, but also how much each image looks like a digit. The IS score has lowest value 1.0 and the larger it is, the better [3].

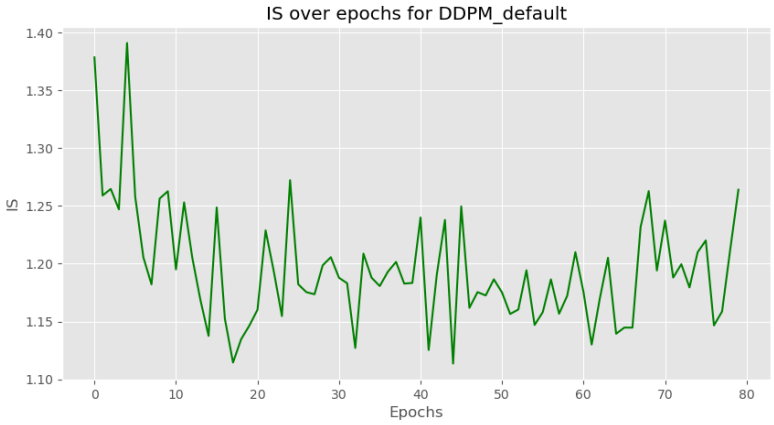

Figure 1.6: IS score at each epoch of the training process for the DDPM model

Figure 1.6 shows the IS score at each epoch of the training process. And the final epoch score was found to be: 1.38 ± 0.12. Here, the IS score decreases over the epochs, which is not what is expected. Once again, this result has to be considered with caution. For computing reasons, this was computed with a relatively small sample of generated images. The more important point is that this metric only compares the generated images to themselves, so it is only a measure of the diversity and quality of the generated images. For the first few epochs, it is possible the

generated images satisfied the IS score conditions listed above, and resulted in a larger score than at the end.

1.3 Running the Model for Different Hyperparameters

For the previous trained model, the set of hyperparameters the model came with was used (see Table 1.1).First, the noise schedule β's are set by a tuple, which sets the range of values that the noise can take. They are then defined as: $\beta_t = \frac{(\beta_2-\beta_1)\times t}{T+\beta_1}$, for $t = 0, \ldots, T$. Here, the tuple was set to (10^{-4}, 0.02). The number of timesteps is set to 1000, and the CNN was set to have 4 hidden layers with 16, 32, 32, and 16 hidden units respectively.

Hyperparameter	Value/Choice
β's	(10^{-4}, 0.02)
Number of timesteps	1000
Learning Rate	2×10^{-4}
Number of hidden layers & units	(16, 32, 32, 16)
Batch Size	128
Activation function	GELU

Table 1.1: Hyperparameters used for the training of the DDPM model

In this section, another set of hyperparameters is chosen and the DDPM model is trained again with these hyperparameters. The new hyperparameters are shown in Table 1.2.

Hyperparameter	Value/Choice
β's	(10^{-4}, 0.02)
Number of timesteps	1500
Learning Rate	4×10^{-4}
Number of hidden layers & units	(16, 32, 32, 16)
Batch Size	128
Activation function	GELU

Table 1.2: New hyperparameters used for the training of the DDPM model (`'testing2'`)

By making the number of timesteps larger, the model will have a "shallower" learning curve, as it will have more steps in which to learn to denoise an image. The learning rate is increased to 4×10^{-4}, making the jumps the gradient descent takes

in the Training Algorithm (Section 1.1.2) larger. This can help avoid falling into a local minimum of the loss function. This can result in the model jumping around the minimum instad of settling, but the idea here is to promote some variability in the model. Overall, these hyperparameters are chosen to see if the model can perform better with this set of hyperparameters, or if the answer is more to do with the model architecture (CNN size, ...).

Again, the model was trained for 100 epochs, and the loss at each iteration was obtained and plotted in Figure 1.7.

Figure 1.7: Loss as a function of epoch number over the training process (green), with the average loss overlayed (dashed black)

A much faster decrease of the loss can be seen here (Fig 1.7). Looking at the samples generated at epochs 1, 20, 40 and 60, shown in Figure 1.8, it is hard to tell if more coherent symbols are generated sooner than for the previous hyperparameter set. However, there seems to be more digit-looking symbols at epoch 40 (3,5,4,9, and a 7) and 60 (0,9's, 8, and a 2) than previously. At the latest epoch however, it may be a bad draw, but the samples are somewhat further from digits than at epoch 60 (Fig 1.9).

Figure 1.8: Samples generated by the DDPM model with `'testing2'` hyperparameters at epochs 1, 20, 40 and 60

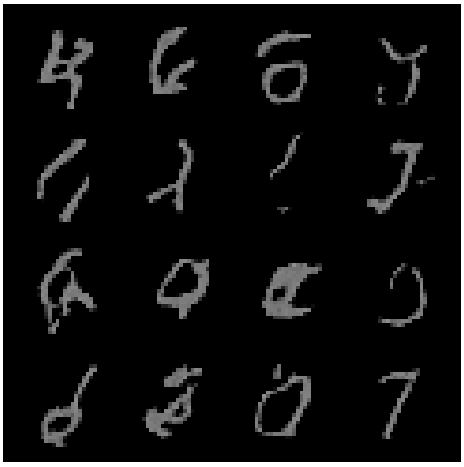

Figure 1.9: Samples generated by the DDPM model with `'testing2'` hyperparameters at epoch 80

The FID score at each epoch is shown in Figure 1.10. The FID score does go down to lower values than for the default hyperparameter set, with the final FID score being 95.35, slightly lower. The IS score at each epoch is shown in Figure 1.11, and the final IS score was found to be 1.5 ± 0.13. Again, we see an unexpected decrease for the first few epochs but the final score is higher than for the default hyperparameters, which does follow the trend of the observations made, which is that this model does perform better than the previous one.

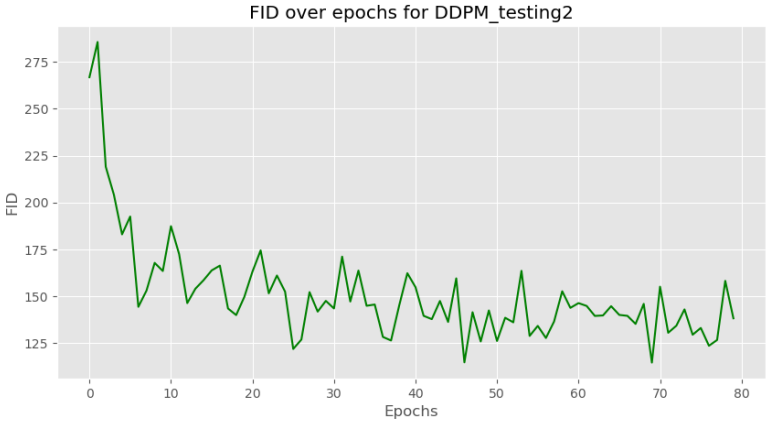

Figure 1.10: FID score at each epoch of the training process

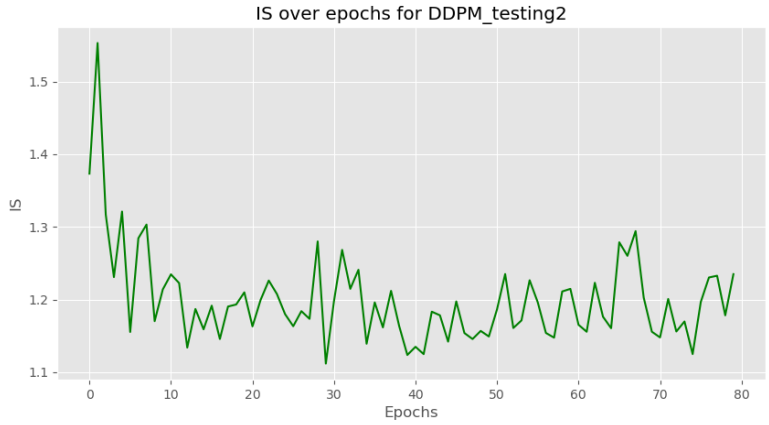

Figure 1.11: IS score at each epoch of the training process for the DDPM model

Chapter 2

Custom Degradation Function

In Bansal et al. (2022, [1]), a conceptual summary of degradation functions is given. Starting with image $\mathbf{x} \in \mathbb{R}$, the degradation of the image, or forward process of the encoder for the DDPM described in section 1.1.2, can be considered as follows: $\mathbf{x}_t = D(\mathbf{x}, t)$, where D is the degradation operator and t is the severity of the degradation. In other words, $D(\mathbf{x}, 0) = \mathbf{x}$. In Chapter 1, the D operator consisted of adding Gaussian noise with variance described by the Variance/Noise Schedule $\beta_{1,\dots,T}$. In this chapter, a custom degradation function, or operator D, will be described and used to train the model on the MNIST dataset. The important part of the degradation function is that an inverse process, R, is required to invert D and satisfies: $R(\mathbf{x}_t, t) \approx \mathbf{x}$, and $R(\mathbf{x}_t, 1) \approx \mathbf{x}_{t-1}$. As was discussed in Chapter 1, this is implemented through a neural network parameterized by ϕ_t and trained to minimize the loss $L = ||\mathbf{x} - R_\phi(D(\mathbf{x}, t), t)||$, taking an l_1 norm [1].

2.1 A Row/Column Averaging degradation function

Taking inspiration from the super-resolution degradation function described in Bansal et al. (2022) [1], a degradation function that averages the rows and columns of the image is proposed. An example of the degradation function for columns is described below:

Column Averaging Degradation Function in the forward process

Input: Data $\mathbf{x}$
Output: $\mathbf{x}_0 = \mathbf{f}_t[\mathbf{z}_t, t]$
repeat
for $i \in \mathcal{B}$ **do** ▷ For each training example index in batch
  $t \sim \mathcal{U}(1, \ldots, T)$ ▷ Sample a random time step
  for $j \in$ Column Schedule[1, . . . , t] : **do** ▷ Degradation of columns
    $\mathbf{z}_t[:, j] = \frac{1}{28} \sum_{k=1}^{28} \mathbf{x}[k, j]$
  end for
  $L = ||\mathbf{x} - \mathbf{f}_t[\mathbf{z}_t, t]||^2$ ▷ Predict the original image
end for ▷ Accumulate losses for batch and take gradient descent step
until convergence

In this forward process the column or row schedule is defined in 3 ways:
- Randomly: an order of rows covering all values from 1 to 28 is randomly chosen
- Outside-In: [1,28,2,27,3,26,...]
- Inside-Out: [14,15,13,16,12,17,...]

The degradation function then, given a time step t, will average the first t rows/-columns listed in the schedule. Here, only the random schedule will be discussed, the other 2 being there for future experimentation. The loss is calculated as the MSE between the original image and the image predicted by the model from that time step, and the model is trained to minimize this loss.

For the inverse process, algorithm 2 of the Bansal et al. (2022) paper is used, as it was found to give better performance for cold diffusion methods [1, p. 4]. The inverse process is described as follows:

Inverse Process for Column Averaging Degradation Function

Input: Degraded Data $\mathbf{z}_T$
Output: Prediction of original sample $\mathbf{x}$
for $s = T, T-1, \ldots, 1$ **do** ▷ For each time step in reverse order
  $\hat{\mathbf{x}} \leftarrow \mathbf{f}_T[z_T, T]$ ▷ Make a direct prediction
  $\mathbf{x}_{s-1} = \mathbf{x}_s - \mathbf{D}(\hat{\mathbf{x}}, s) + \mathbf{D}(\hat{\mathbf{x}}, s-1)$ ▷ Predict the previous time step's image
end for

Where $\mathbf{D}$ is the degradation function, so $\mathbf{D}(\hat{\mathbf{x}}, s)$ is the image $\hat{\mathbf{x}}$ degraded at time step s, with the first s rows/columns in the schedule order averaged. The degradation was originally made to average rows/columns by groups of 4, resulting in a degraded image with 7 rows/columns. However, these led to too much information lost and more importantly only 7 time steps over which the model could learn how to de-

average the rows/columns. Examples of the samples obtained with the groupings can be found in the Appendix. Both Fig 3.1 & 3.2 show that the model struggles to re-construct the images when only given 7 averaged rows/columns. One thing to point out is that the degraded samples values have been extended to the -0.5 to 0.5, accentuating the contrast essentially. This is so the degradation can be better visualised. Indeed, since the images are a majority of black pixels, averaging leads to low values, and the actual degraded images are quite dark. However, this should not be an issue for the CNN. Though a point could be made that a larger value range may help the CNN reconstruct the image more easily. The advantage that grouping offered was a lighter computational cost. Maybe with a deeper CNN, reconstruction could be succesful though that would just negate the latter point.

Thus, the model was trained with each row/column averaged on their own, which gives 28 time steps, and should allow the model to learn how to reconstruct the image better.

2.2 Training the modified model on the MNIST dataset

The non-grouped row averaging model is discussed here as it gave the best result (see Fig 3.3). The model was trained for 80 epochs, with the following hyperparameters:

Hyperparameter	Value/Choice
Row/Column Schedule	Random
Number of timesteps	28
Learning Rate	2×10^{-4}
Number of hidden layers & units	(16, 32, 32, 16)
Batch Size	128
Activation function	GELU

Table 2.1: Hyperparameters used for the training of the column averaging cold diffusion model

As before, samples are generated for epochs 1, 20, 40 and 60, and the results are shown in Figure 2.1. The first thing that can be noticed is that digits are being predicted as early as epoch 1. However, these are a majority of nines, showing some bias towards a certain type of digit. The predicitons can be seen to improve in quality over time, with different digits being predicted.

And for the last epoch, the samples generated are shown in Figure 2.2, and are compared to the original, degraded and directly predicted ones. As can be seen the model is able to re-construct the samples quite well. However, there is an issue which is that the model is quite "shy", with the digits being faint, though looking similar to

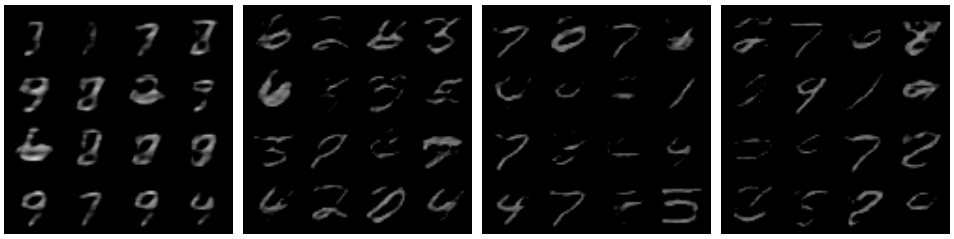

Figure 2.1: Samples generated by the row averaging degradation function, after 60 epochs, without grouping (Epoch 1: extreme left, Epoch 20: middle left, Epoch 40: middle right, Epoch 60: extreme right)

the original samples. Otherwise, these show agreement with the finding of Bansal et al. (2022), as using the gradual reconstruction algorithm (Algorithm 2) gives much better results than a direct prediction.

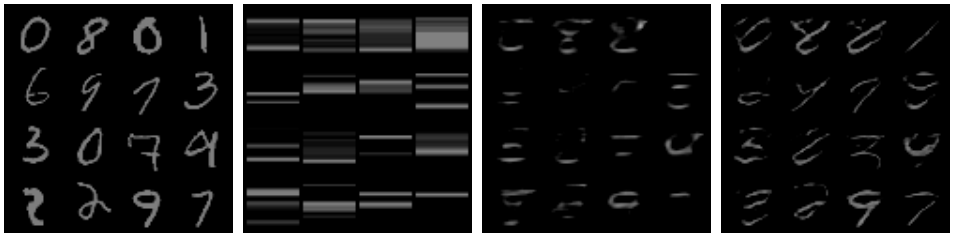

Figure 2.2: Samples generated by the row averaging degradation function, after 80 epochs, without grouping (Original: extreme left, Degraded: middle left, Direct prediction: middle right, Algorithmic generation: extreme right)

In terms of the metrics for this model. The loss at each iteration and average loss over epochs were obtained and are plotted together in Figure 2.3. The loss clearly falls quicker and lower than for the denoising diffusion model, and this agrees with the quality of the samples that were generated.

The FID score at each epoch is shown in Figure 2.4, and the final FID score was found to be 108.5. This is higher than expected, though it may be explained by the very faint predictions. The IS score at each epoch is shown in Figure 2.5, and the final IS score was found to be 1.9 ± 0.18, compared to 1.7 ± 0.24 for real images. And though this size difference is not as expected ($IS_{real} > IS_{generated}$), the IS score does indicate larger values than for the noise diffusion model, which can be considered a good sign.

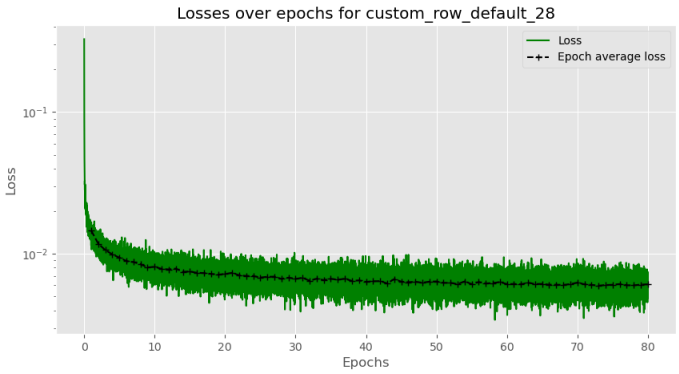

Figure 2.3: Loss at each iteration of the training process, and the average loss over each epoch

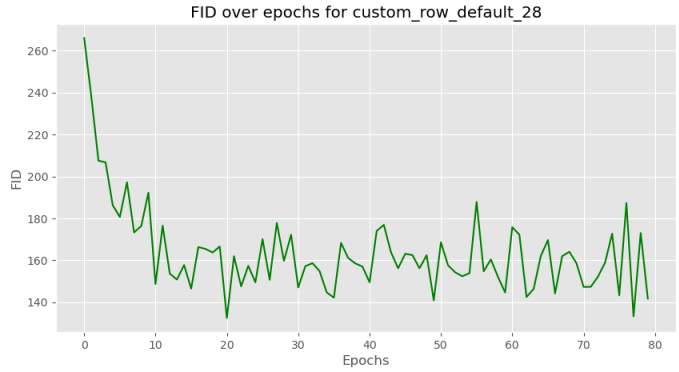

Figure 2.4: FID score at each epoch of the training process

2.3 Unconditional Sampling

An important point to make is the way the generated digits were sampled. Unlike for the DDPM, the degraded samples given to the decoder/reverse process were fully degraded MNIST sampled images. This gave the option to compare the generated samples to original ones, however, this means the sampling was conditional [1, Section 5]. In addition to being restricting, this is also less efficient than being able to generate a degraded sample from a distribution.

An attempt was made at unconditional sampling by simply creating a degraded image using different uniform distribution. Samples were generated with the fully trained row averaging model and the following results were obtained:

Fig 2.6 shows worse performance than for the conditional sampling, though it can be seen the degraded samples created are quite different to the ones obtained in the conditional sampling. Overall, this seems do-able though fine-tuning the distribution from which the row/column values are sampled is needed, adn should be the objective

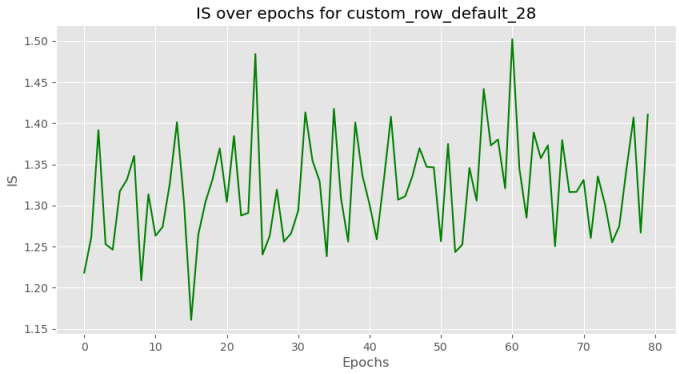

Figure 2.5: IS score at each epoch of the training process for the row averaging cold diffusion model

Figure 2.6: Samples generated by the row averaging degradation function, after 80 epochs, using degraded samples created from a distribution (Original: left, Degraded: middle, Algorithmic generation: right)

of any future work on this. Already, one such example of tuning was that they needed to be different for the row and column cases.

Chapter 3

Appendix

3.1 Other versions of the degradation function

Here is shown results obtained for the different versions of the degradation function.

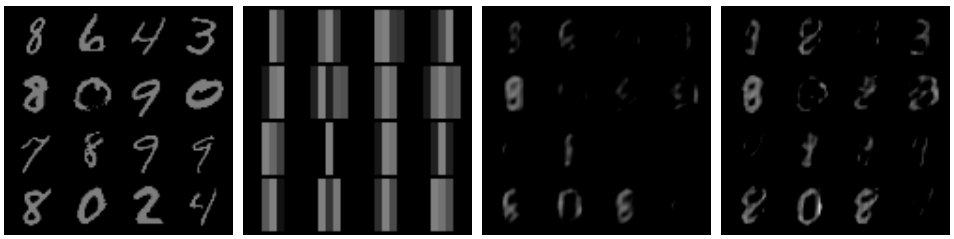

Figure 3.1: Samples generated by the column averaging degradation function, after 60 epochs, when grouping columns by 4 (Original: extreme left, Degraded: middle left, Direct prediction: middle right, Algorithmic generation: extreme right)

Metric	Final
Loss	0.013
FID	195.3
IS Generated	1.44

Table 3.1: Final loss, FID and IS for the column averaging degradation function with 7 columns, after 60 epochs

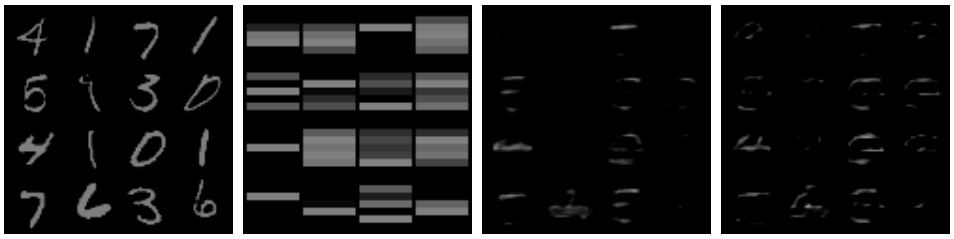

Figure 3.2: Samples generated by the row averaging degradation function, after 80 epochs, when grouping rows by 4,(Original: extreme left, Degraded: middle left, Direct prediction: middle right, Algorithmic generation: extreme right)

Metric	Final
Loss	0.0108
FID	189.84
IS Generated	1.97 ± 0.32
IS Real	1.79 ± 0.23

Table 3.2: Final loss, FID and IS for the row averaging degradation function with grouped rows, after 80 epochs

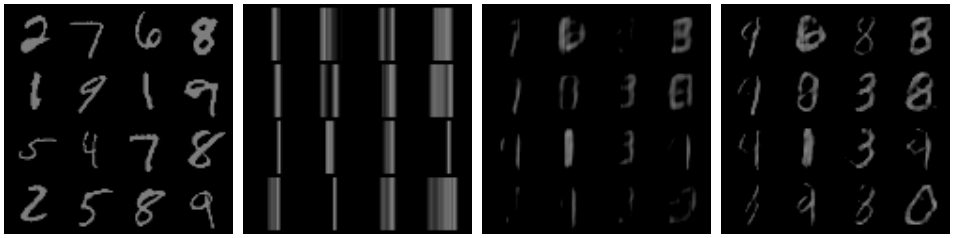

Figure 3.3: Samples generated by the column averaging degradation function, after 60 epochs, when not grouping columns,(Original: extreme left, Degraded: middle left, Direct prediction: middle right, Algorithmic generation: extreme right)

For ungrouped column averaging (Fig. 3.3), the generated samples are of decent quality but there is a bias towards certain digits, mainly those that have a more vertically uniform quality (1, 3, 8, 9, 0). This brings up the interesting point of there being more information lost when averaging columns rather than rows as numbers are generally taller than they are wide. In other words, when averaging rows, the model receives more rows that contain information about the digits than columns when averaging these. This is seen in the degraded samples image in Fig **??** and 3.3.

Metric	Final
Loss	0.0078
FID	115.9
IS Generated	1.31 ± 0.2
IS Real	1.26 ± 0.16

Table 3.3: Final loss, FID and IS for the row averaging degradation function with 28 rows, after 80 epochs

3.2 REAMDE

3.2.1 M2 Coursework - Training Diffusion Models on the MNIST Dataset

Description

This reporsitory contains the code and written report for the M2 Coursework. The aim was to first train a Denoising Diffusion Probababilistic Model (DDPM) for different hyperparameters sets and compare them. Second, a cold-diffusion model using a custom-made row or column averaging degradation function was developped and evaluate its performance compared to the DDPMs.

Contents

Inside this `tmb76/` directory, there are a few sub-directories one can explore. There's the code directory `src/`, which contains all the code used for the report. There is a `notebooks/` directory which contains notebook versions of the `src/` code. Including a `load_and_play.ipynb` notebook made to skip training the diffusion models. This project comes with pre-trained models weights and biases for all the models used in the project, for different epochs. These follow the following naming convention:

For the DDPM models:

```
ddpm_mnist_{epoch number}_{hyperparameter set used}.pth
```

Choices of epoch numbers are 0, 20, 40 , 60, and 80 (plus 100 for the default hyperparameter run). And the hyperparameter sets are either default, testing or testing2.

And for the custom degradation cold diffusion models:

```
custom_mnist_{epoch number}_{orientation}_{hyperparameter set used}.
    pth
```

where the epoch number choices are the same except for the grouped column averaging model (col_default_7) that only goes to 60 epochs.

In the `load_and_play.ipynb` notebook, one can initialise the different diffusion models, then load the chosen/corresponding model state files with wights and biases, and then use this model to generate samples and then calculate some image quality metrics.

An important note is that the code will give outputs in the command line but also store the plots in a `Plots/` directory which will be created as the first code file is run. So if there is no `Plots/` directory in the repository yet, running the solver once should lead to creating one. The same is true for the `contents/`, `contents_custom/`, and `contents_lap/` (for the `load_and_play.ipynb` notebook) directories in which generated and other samples are stored.

The last one is the `Report/` directory, which contains the LaTeX file for the report, as well as the pdf version of it, along with the references `.bib` file. More importantly, there are an `environment.yml`/`requirements.txt` and `Dockerfile` files, which one is advised to use.

How to run the code

If one needs to run the code on different hardwares/supercomputers or re-train the models for a desired set of hyperparameters or number of epochs, they can use the `Dockerfile` provided in this repository. For permissions reasons, the `Dockerfile` is not set up to pull the repository directly as it builds the image. Therefore, one must first download this repository to their local machine and then are free to build the Docker image from the `Dockerfile`.

To run the code on a Docker container, one first has to build the image and run the container. This can be done as follows:

```
$ docker build -t m2_coursework .
$ docker run --rm -ti m2_coursework
```

The `m2_coursework` is not a strict naming instruction, it can be set to any other name the user may prefer.

If there is a need to get the plots back on the local machine, the second line above can be ran without the `--rm` and also set the container name using `--name=container_name` (any valid name is fine). From there, run all the code as instructed below. Once all desired outputs and plots have been obtained. One can exit the container and then run:

```
$ docker cp docker cp container_name:/M2_Coursework/Plots ./Plots
```

The `Plots/` directory will get copied into the local folder the container was ran from. Similarly, for generated samples need to be copied locally, one can run:

```
$ docker cp docker cp container_name:/M2_Coursework/path ./path
```

where path can be: - `contents` - `contents_custom` - `notebooks/contents_lap`

With the build and run commands above, the Docker image will get built and the container ran, providing the user with a bash terminal-like interface where the code can be run. There are 3 files that can be run: `part_1.py`, `part_2.py`, and `training_grounds.py`. The first enables one to train the DDPM model for a chosen number of epochs and hyperparameters, save it, generate samples and compute some image quality metrics at different stages of the training.

```
$ python src/part_1.py {num_epochs} {hyper_parameters}
```

where num_epochs can be any desired integer (see note on time), and the hyperparameters must be a string in the following list: 'default', 'light', 'more_capacity', 'testing' and 'testing2'. The second file does the same for the custom degradation cold diffusion models and requires the additional argument of which axis the degradation function needs to average values:

```
$ python src/part_2.py {num_epochs} {hyper_parameters} {orientation}
```

where the hyperparameters now must be one of the following: 'default_7', 'default_28', and 'more_capacity'. `orientation` must be either 'row' or 'col'. Finally, the third file is there if one simply wants to train a model to use it elsewhere and not bother generating samples or evaluating the model with metrics. For this file, another argument is needed to specify if one wants to train the DDPM or custom degradation cold diffusion model:

```
$ python src/training_grounds.py {num_epochs} {hyper_parameters} {
    custom_deg} {orientation}
```

The hyperparameters are all of the above, depending on which type of model is run, with the exception of the 'testing' hyperparameter set for the DDPM model. The `custom_deg` argument is either 'True' or 'False' accordingly (Not a bool, but a string!).

Note on time: Running the `part_1.py` file was runnable locally overnight, with an average of 40 seconds per epoch. This is based on running all of these on a MacBook Air M2 (2022, Ventura 13.2.1), with 8 GB of Memory, so this may be slower on a container. Running the `part_2.py` most likely requires the use of a GPU or powerful remote machine. Training was done using Amazon Sagemaker's EC2 P3 instances (NVidia V100 Tensor Core GPUs), and took an approximate 40 minutes for the grouped averaging custom degradation, and up to 2 hours to train the non-grouped averaging degradation function.

Further development

If one wishes to further develop this code, such as adding more algorithms to try, when the image is built, git is installed and initialized and the pre-commit hooks are installed.

Use of Generative AI

GitHub Copilot's autocompletion feature was used when writing docstrings for the functions, though sometimes adding elements ourselves as the functions were modified after writing the docstrings, and for repetitive parts of the code. ChatGPT was also used to help in debugging the code, by providing the traceback as a prompt when an error was difficult to understand, asking to explain what the error refers to. One was to deal with an issue building the dockerfile failing to solve a process where conda was not found to be installed on the image when trying to build the environment using the environment.yml file. A solution proposed and adopted was to use pip install with a requirements.txt file instead. A few prompts were about relative imports in the notebooks once they were moved to the `notebook/` directory. Also encountered an error related to the absence of an operator in torchvision. A few suggestions were givent to troubleshoot and resolve the error, and one working solve was to update torch and torchvision to the latest versions. Asked how to troubleshoot a RuntimeError related to input and weights types mismatch in PyTorch, and was advised to check the types of input and weight tensors, ensuring they are on the same device or converting them to the same type if necessary. One final instance of using chatgpt was to ask how one could repeat the sample tensors so they would have 3 identical channels as that was required by the FID and IS metrics methods from torchmetrics, the `.repeat(1,3,1,1)` solution was implemented.

Bibliography

[1] Arpit Bansal, Eitan Borgnia, Hong-Min Chu, Jie S. Li, Hamid Kazemi, Furong Huang, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Cold diffusion: Inverting arbitrary image transforms without noise. *Journal of Computer Vision*, 2022.

[2] Jason Brownlee. How to implement the frechet inception distance (fid) from scratch. *Machine Learning Mastery*, 2022.

[3] Jason Brownlee. How to implement the inception score from scratch for evaluating generated images. *Machine Learning Mastery*, 2022.

[4] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

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

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2015.

[7] David Mack. A simple explanation of the inception score. *Medium*, 2022.

[8] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.

[9] Wikipedia. Mean squared error. *Wikipedia, The Free Encyclopedia*, 2022.