M2 Coursework

Cailley Factor

March 2024

Section 1

a) The denoising diffusion diffusion model (DDPM) consists of an encoder and a decoder [1]. The encoder transforms a data example x into intermediate variables $z_1, z_2, ...z_T$ by adding Gaussian noise [1]. The hyperparameters β_t of the model comprise the noise schedule and determine how quickly the noise is added to the data examples [1]. The 'ddpm_schedules' function creates a linear schedule of β_t over T steps evenly spaced between the hyperparameters β_1 and β_2. The encoder can operate in one step because the diffusion kernel can be written in closed form and enables the z_T to be calculated for a given value of x, based on $\alpha_t = \prod_{s=1}^{t} 1 - \beta_t$ [1]. The encoder is defined within the forward step of the 'DDPM' class in the calculation $z_t = \sqrt{\alpha_t} x_i + \sqrt{1 - \alpha_t}\epsilon$ [1]. The decoder, called g_t in the model, which is a convolutional neural network (CNN) is used to predict the added noise [1]. The decoder maps the latent variables z_t to z_{t-1} back to x, by approximating the reverse distributions as normal distributions [1].

For the CNN decoder model, the 'CNNBlock' class is defined which represents a building block for the CNN, comprising a convolutional layer, a layer normalisation, and an activation function, which is GELU by default. The 'CNN' class is defined which initialises a sequence of CNN block layers, specified with a kernel size of 7 and padding of 3. The output channels of each CNN block layer is specified by the 'n_hidden' input. A final convolution layer adjusts the output dimensions to match the input dimensions without applying an activation function at the end. A time embedding mechanism transforms a scalar time variable 't' into a high-dimensional representation via a sinusoidal transformation followed by several linear transformations. In the forward pass, the time encoding is added after the first convolution block is applied and embeds temporal information across the spatial dimensions of the feature maps. The spatial dimensions (height and width) of the input are preserved throughout the network.

To train the model, for each item in the batch, a random time-step is uniformly sampled and Gaussian noise is sampled. Using the diffusion kernel, the input data is transformed to the noised image. A re-parameterised loss function is used which calculates the mean squared error between the output of the CNN model predicting the noise added and the true noise. The losses are accumulated for the batch and the Adam optimizer is used to minimise the loss.

The sampling function within the 'DDPM' class creates a randomly generated image of Gaussian noise, z_t. For each time-step, the previous latent variable is predicted by subtracting off the noise ϵ predicted by the CNN weighted by the noise scheduler. Random normal noise is then added back to the previous latent variable. This is repeated for each time step to result in a sample without noise after T iterations.

b) The code from the notebook was converted into modularised python files and the model was initially trained on the provided dataset with the default hyperparameters including four hidden layers, with the 'n_hidden' hyperparameter set to (16, 32, 32, 16). The model was trained for 100 epochs and a batch size of 128 and a learning rate of 2×10^{-4}. The loss curve comprising both the training and testing losses were plotted, and a grid of samples was outputted after each epoch to monitor the improvements in the performance of the model. The loss curves plotted over each epoch are shown in figure 1.

The hyperparameter describing the number of hidden units in the middle was modified to add in an extra hidden layer, such that the 'n_hidden' hyperparameter equaled (16, 32, 64, 32, 16). Other modifications to the hyperparameters could have included modifying the noise schedule, however, this may result in more overall noise added to the latent variable z_T after the encoder process such that loss function values can not be used to compare the results between the default and modified hyperparameters. As 'n_hidden' was

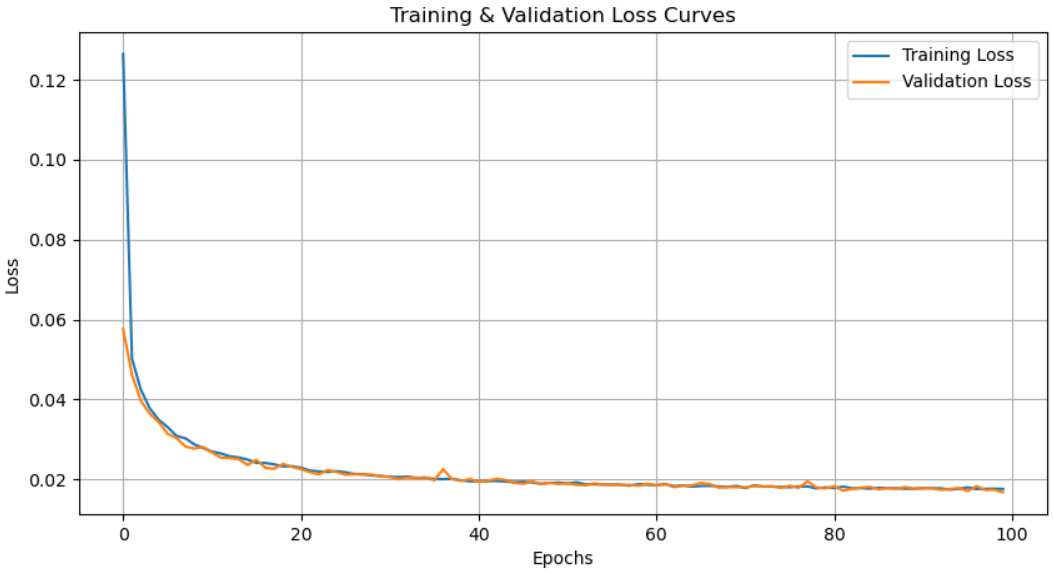


Figure 1: Plot of the training and validation losses for the model trained with the default hyperparameters

chosen as the hyperparameter to modify, the loss curves were used for quantitative comparison, as were the sampling results after various epochs. Adding more hidden layers to the CNN can increase the model's ability to learn more complex features, i.e., a more complex denoising function, at the expense of an increased computational cost and potential for overfitting. Following training with the changed hyperparameter, the loss curves plotting the training and testing losses were generated, as shown in figure 2.

To better analyse differences in the loss curves of the two models, the training and validation loss curves were compared between the models in figure 3. 'Model 2', the model with an additional hidden layer, has slightly lower training and validation losses than 'Model 1' with the default hyperparameters. With more epochs, the differences in the loss curves between the models converge. It would have been useful to use a greater number of hidden layers to better assess the impact of this change. However, given the length of time that the code took to run with five hidden layers, the run time would have increased too substantially to be trained locally if more were added. One thing of note about the loss curves is that the validation loss was lower than the training loss at some points along the loss curves. This is because the validation losses are calculated for each epoch after the model has been trained on all of the batches for that epoch, whereas the training losses are accumulated as the model trains on the batches of that epoch.

The quality of the samples generated by the models were sampled after each epoch. The samples generated after 1, 25, 50, 75, and 100 epochs for both the models were plotted in figures 4, 5, 6, 7, and 8, respectively. The model with the extra hidden layer appears to generate slightly higher quality samples earlier in training than the model with the default hyperparameters, as shown in figure 5, in line with the results from the loss curves comparison. Nonetheless, 16 is a small sample size and it is not possible to draw too much information from these qualitative differences. By the 100th epoch, an advantage of the model with the extra hidden layer could not be clearly distinguished with samples from both models generating multiple images that appear to be handwritten digits.

c) Individual samples after each epoch were generated from both models trained for 100 epochs. In figure 8a, in the third row and second and third columns, higher quality samples can be seen clearly resembling the numbers '8' and '9'. Lower quality samples include the third and fourth samples in the first row, which bear no resemblance to specific digits. For the model with an extra hidden layer, in figure 8b, higher quality samples can be seen in the first row and third and fourth columns, clearly resembling a '3' and '0', respectively. Lower quality samples can be seen in the second row and second column and second row and third column, bearing no clear resemblance to specific digits.

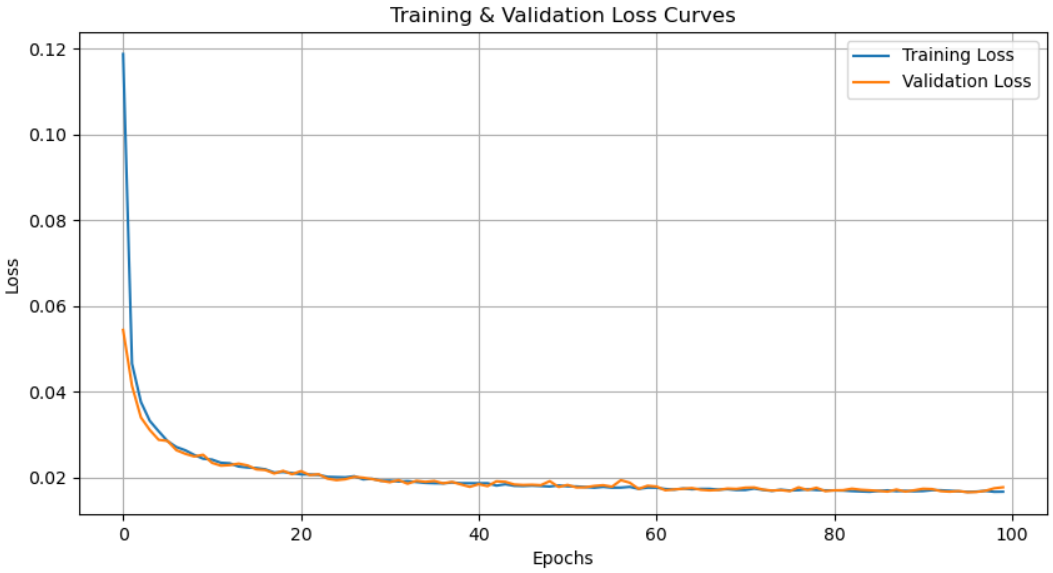


Figure 2: Plot of the training and validation losses for the model trained with an extra hidden layer compared to the default hyperparameters

Section 2

a) Bansal et al. describe how to implement cold diffusion, i.e., instead of utilising Gaussian noise, they describe how a diffusion model can be built around arbitrary image transformations, such as Gaussian blurring or the ImageNet-C *snowification* operator [2]. As a first strategy, Gaussian blurring was selected as the arbitrary image transformation based on the Bansal et al. paper and code from 'deblurring-diffusion-pytorch' from their repository was modified to apply iterative Gaussian blurring on the MNIST images and used to inform the development of the forward process [2, 3]. The 'get_conv' function was based on code in the Bansal et al. repository and initialises a Gaussian kernel from 'torchgeometry' with specific dimensions and standard deviation values [3]. The convolutional layer is configured to use this kernel for the blurring operation. The 'get_kernels' function was also informed by code in the Bansal et al. repository and initialises a list of convolutional layers, each configured with a Gaussian kernel [3]. The kernel standard deviation is scaled according to the corresponding time step, allowing for a gradual increase in the blurring intensity over this parameter and the kernels are stored for each time step. The 'blur' function involves utilizing the convolutional layers initialized in the previous step to apply Gaussian blurring to the input tensor based on the provided timestep parameter 't'. Each item in the batch is iterated over and the corresponding Gaussian kernel applied based on the input timestep parameter.

b) As in standard diffusion models, there is an image degradation operator to degrade the image and this is scheduled to vary based on a timestep parameter 't' to transform a data example x into intermediate variables $z_1, z_2, ...z_T$ [2]. This can be denoted as $z_T = D(x, t)$, where D represents the degradation operator [2]. As with standard diffusion, a restoration model, i.e., decoder, g_t, the CNN in this code, can be trained such that it inverts the degradation operator D [2]. The forward process was modified such that a tensor 't' randomly samples different timesteps with which to blur each item in the batch. The 'blur' function was called and applied to the input tensor based on the batch and corresponding timestep. The forward method then calculates the mean squared error between $g_t(D(x, t), t)$ and x, i.e., the degraded image restored via the CNN compared to the true image, as discussed in the Bansal et al. paper [2].

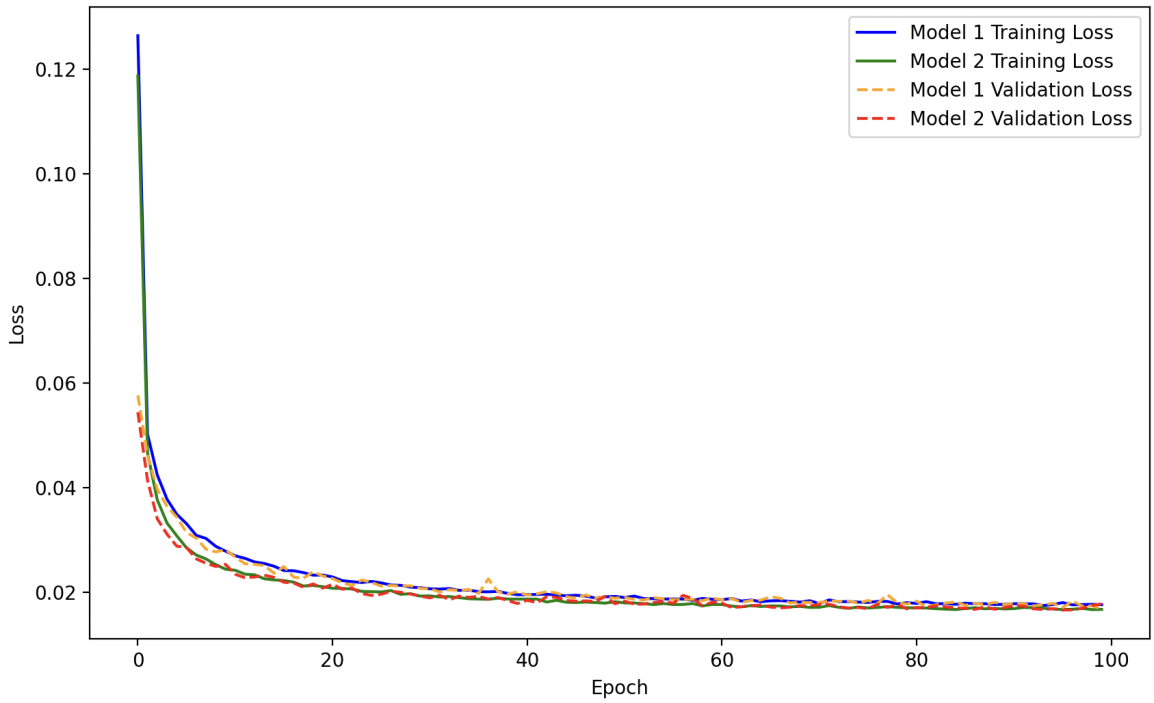


Figure 3: Plot of the training and validation losses for the model trained with two sets of hyperparameters with Model 1 being the model trained with the default hyperparameters, and Model 2 having an extra hidden layer.

The sampling method was altered in line with the discussion in the Bansal et al. paper about how to improve sampling for cold diffusion. In noise-based diffusion, Algorithm 1 in equation 1 is used to sample from a degraded sample [2]. In the case of the starter code, the degraded sample is a tensor of random numbers drawn from the standard normal distribution. However, the Bansal paper found Algorithm 1 (1) to yield poor results in the case of cold diffusion and Algorithm 2 in equation 2 was proposed for sampling from a cold diffusion model [2].

Algorithm 1

For a degraded sample z_t

for $s = t, t-1, \ldots, 1$ do

$$\hat{x} \leftarrow R(z_s, s) \tag{1}$$

$$z_{s-1} \leftarrow g_t(\hat{x}, s-1)$$

end for

return x_0

Algorithm 2

For a degraded sample z_t

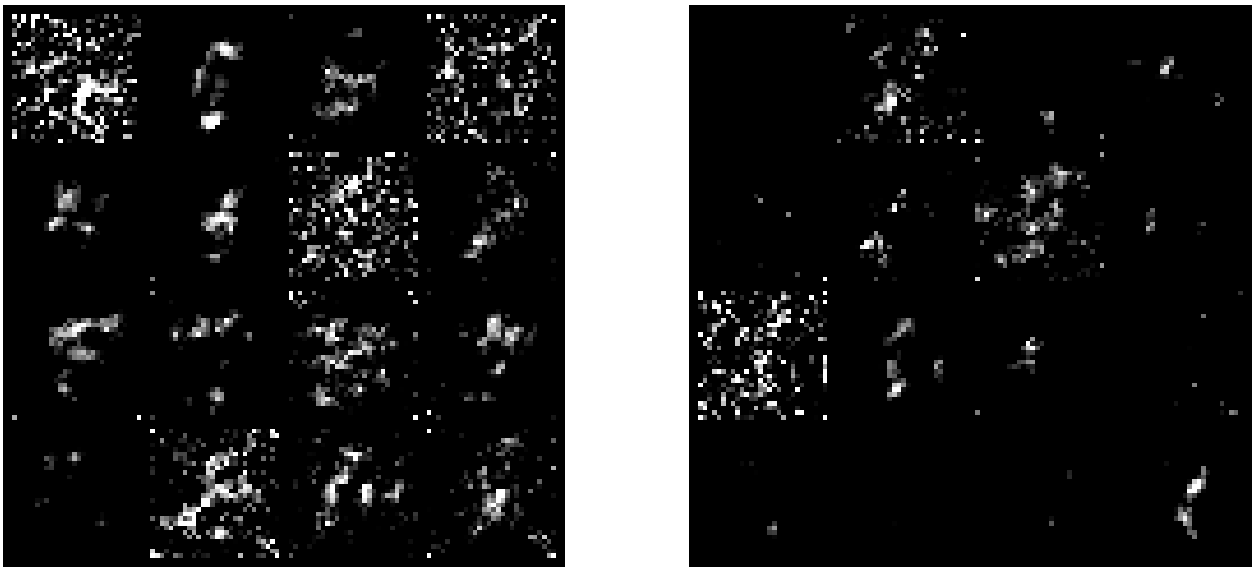
for $s = t, t-1, \ldots, 1$ do

$$\hat{x} \leftarrow R(z_s, s) \tag{2}$$

$$z_{s-1} \leftarrow x_s - g_t(\hat{x}, s) + g_t(\hat{x}, s-1)$$

end for

return x_0



(a) (b)

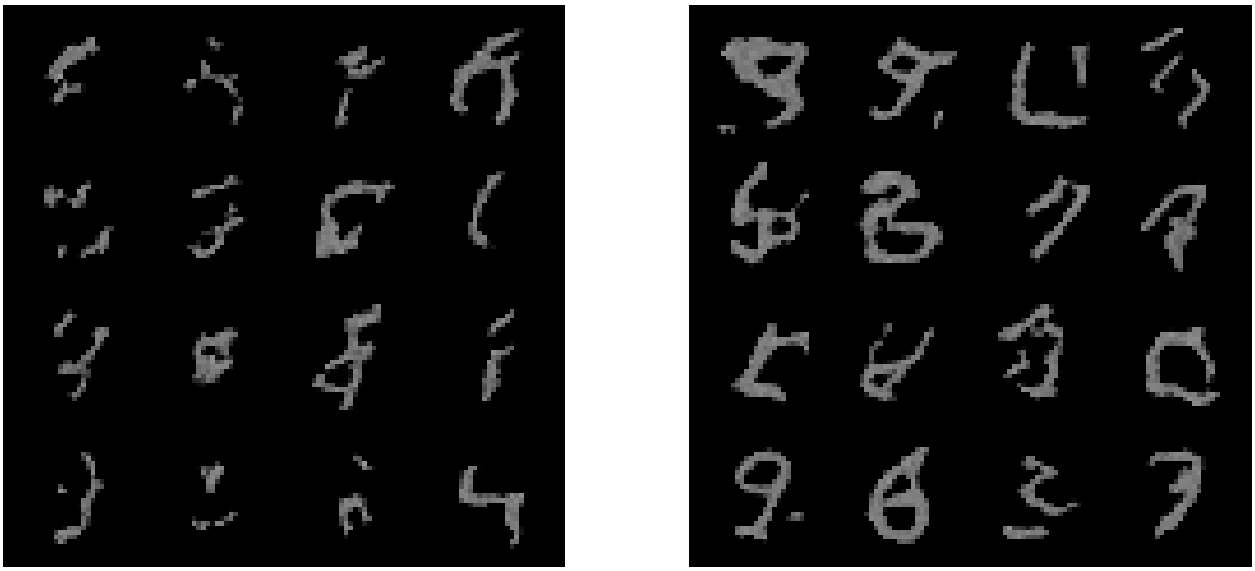
Figure 4: Comparison of 16 samples generated from the sampling method of the DDPM model after epoch 1 with a) the default hyperparameters and b) the extra hidden layer

For the sampling method for the cold diffusion model, algorithm 2 in equation 2 was implemented. The model was set to evaluation mode and the sampling method takes in images from the validation data loader in the 'train2' method in order to be able to blur them as starting images based on the parameter 'n_T' number of iterations for each batch. A loop iterates over each time step and the original image is reconstructed based on the CNN trained in the training method. The blur operation is then applied to the reconstructed image for t timesteps and also to the reconstructed image for t-1 timesteps. The principles of algorithm 2 (2) are applied such the reconstructed image for t timesteps is subtracted from the blurred image and the reconstructed image blurred to t-1 time steps is added back to the image. This is done iteratively, as in algorithm 2 for timesteps t to 1.

The model was trained using two different Gaussian kernels. One with a kernel size of 3 and a standard deviation of 0.1 and another with a kernel size of 13 and a kernel standard deviation of 7.0, hitherto referred to as the first set of parameters and the second set of parameters, respectively. Utilising the first set of parameters, the model was trained and sampled from for 100 epochs and the loss curves generated are plotted in figure 9. Utilising the second set of parameters, the model was again trained and sampled from for 100 epochs and the loss curves generated are plotted in figure 10.

It can be seen from the reconstructed samples from the CNN comparing epoch 1 and epoch 100 for the models with the first set of parameters in figure 11 and second set of parameters in figure 12 that the forward method effectively learned to reconstruct the blurred inputs for both levels of blurring by the 100th epoch. However, as can be seen by the values in figure 9 with the less blurred starting image compared to figure 10 with the more blurred starting image, the losses were substantially lower, as expected, for the model with the less blurred starting image. Figure 13 shows how the sampling method described above was able to effectively generate samples of numbers from the model with the less blurred inputs. An improvement in the quality of the samples generated can be seen from epoch 1 to epoch 100. However, this was not the case for the samples from the model with the second set of parameters, i.e., greater blurring, as shown in figure 14. No digits or even fuzzy symbols were produced and there was also no improvement in the sampling over the 100 epochs. Additionally, both of the models show deterministic sampling with no new numbers being generated.

c) The key difference between the standard and cold diffusion degradation strategies is that the model with the Gaussian blurring degradation resulted in only deterministic sampling and failed to operate from a truly random starting image, unlike the model with the Gaussian noise degradation. There are several

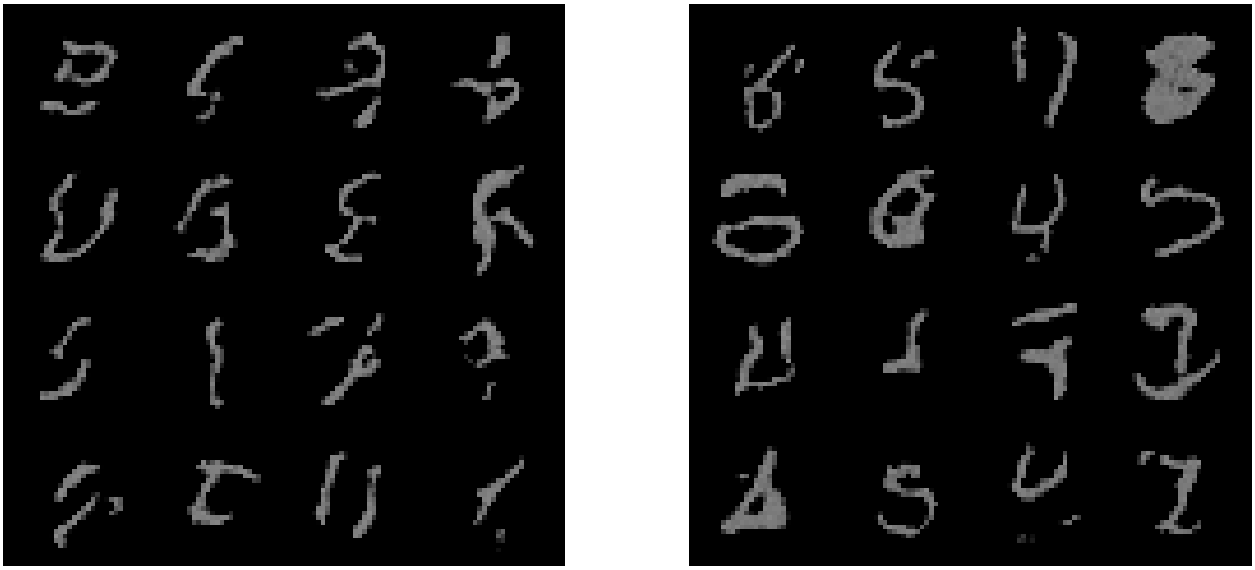


(a) (b)

Figure 5: Comparison of 16 samples generated from the sampling method of the DDPM model after epoch 25 with a) the default hyperparameters and b) the extra hidden layer

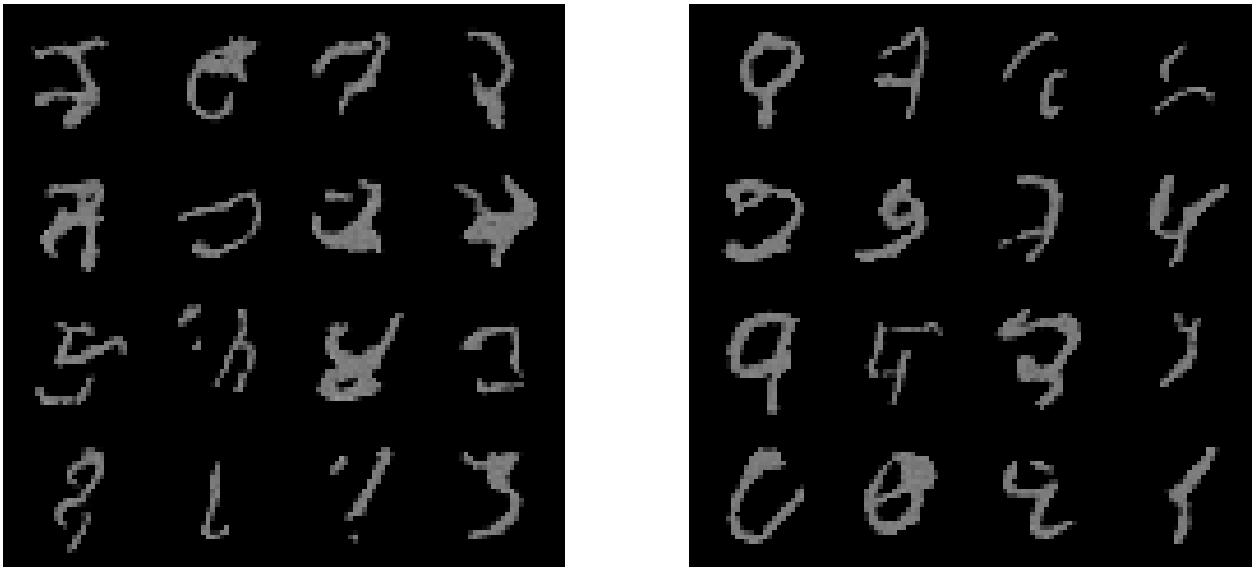
improvements to pursue for the cold diffusion model and additional explorations to probe. One fundamental improvement needed is in the 'blur' method of the forward pass. This should have an additional loop within the loop over each batch to ensure that the Gaussian kernels for each timestep were applied iteratively based on the value of t, rather than just the kernel for the timestep selected. This is a fundamental reason behind why the model only operated deterministically and why the sampling method failed to generate new samples. The method in this coursework doesn't work because there is an error in the code and the Gaussian kernels are not applied iteratively. However, due to time constraints, these changes were not able to be tested, as changes to the code attempting to implement this methodology had an extremely slow runtime when using a local device.

Another change given more time would have included connecting the noise scheduler defined in the 'ddpm_schedules' module to the code defined in the 'ddpm2' module in the 'get_kernels' function of the 'ddpm' class in the portion determining the standard deviation increments for the kernels. Lastly, some exploratory changes would have included trying different degradation strategies beyond those in the Bansal et al. paper. An example of other degradations to try would have included motion blur or salt-and-pepper blur, such as utilising the implementation from 'OpenCV' or 'Scikit-Image', respectively.



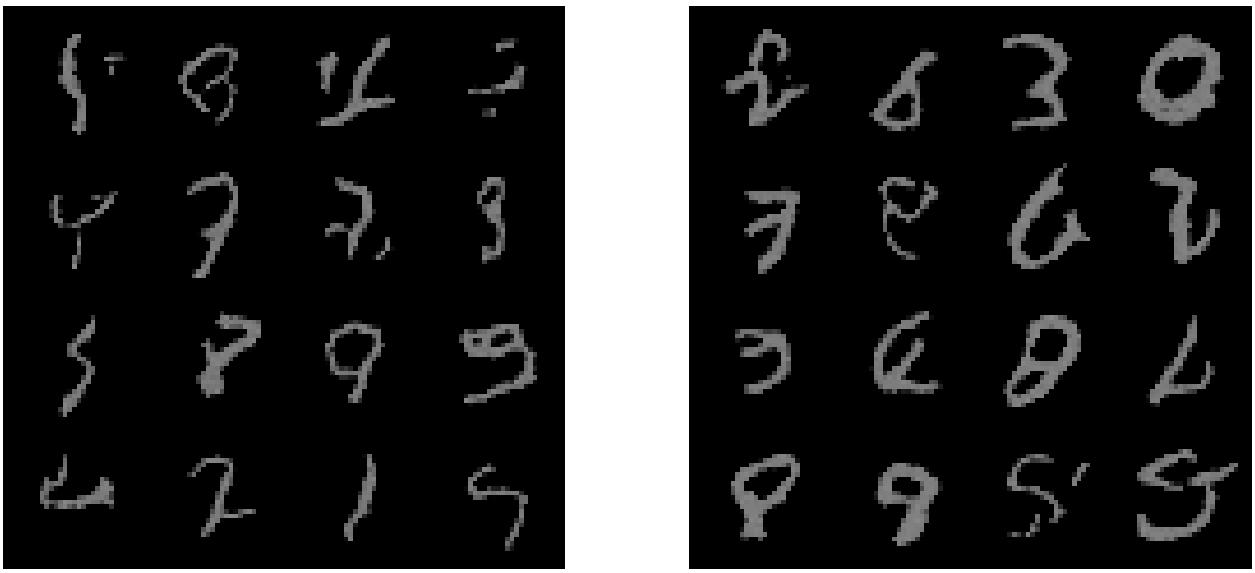
(a) (b)

Figure 6: Comparison of 16 samples generated from the sampling method of the DDPM model after epoch 50 with a) the default hyperparameters and b) the extra hidden layer



(a) (b)

Figure 7: Comparison of 16 samples generated from the sampling method of the DDPM model after epoch 75 with a) the default hyperparameters and b) the extra hidden layer



(a) (b)

Figure 8: Comparison of 16 samples generated from the sampling method of the DDPM model after epoch 100 with a) the default hyperparameters and b) the extra hidden layer

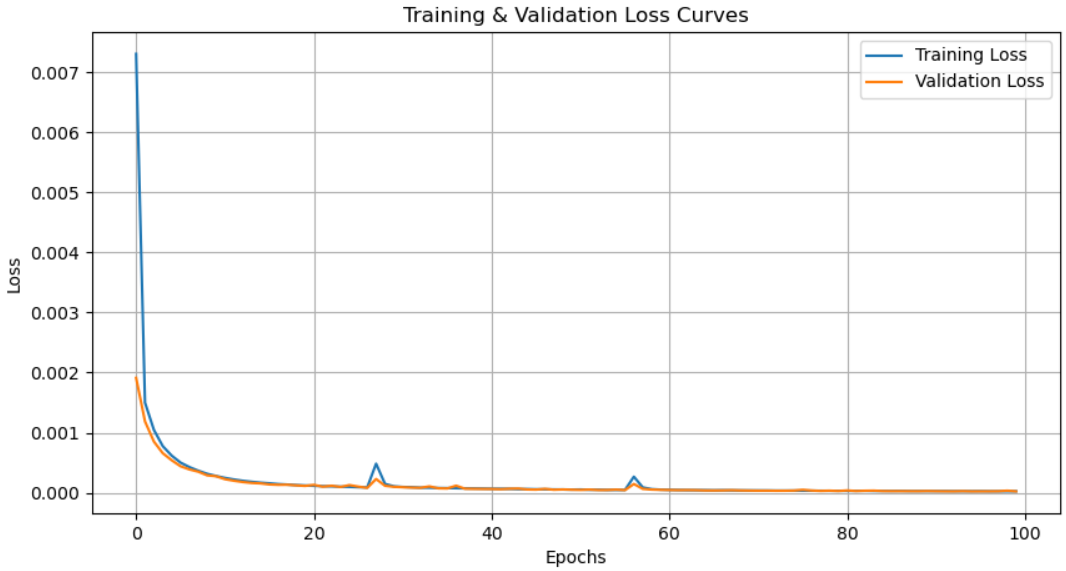


Figure 9: Plot of the training and validation losses for the cold diffusion model trained with a kernel size of 3 and a standard deviation of 0.1

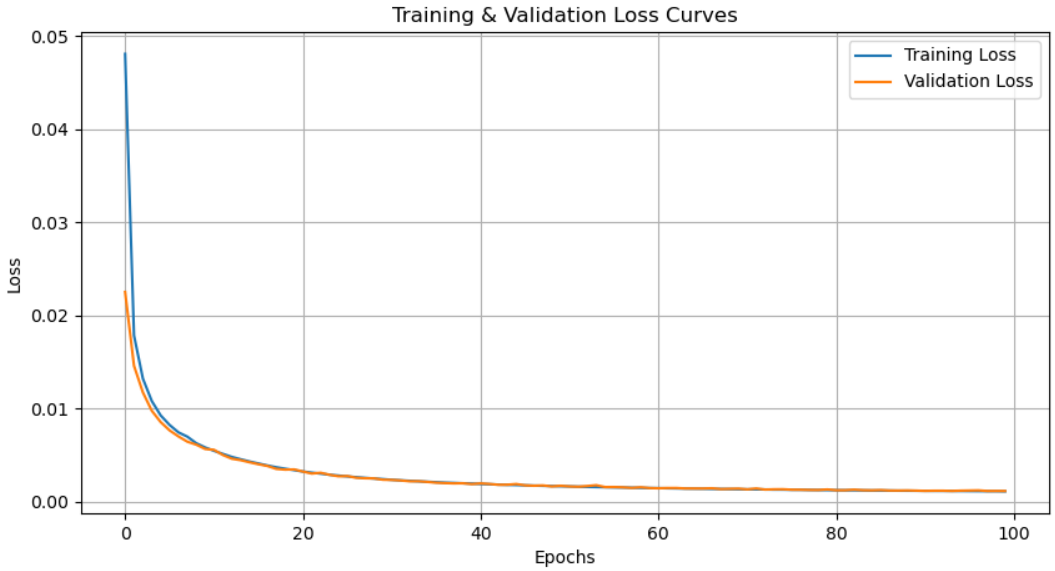
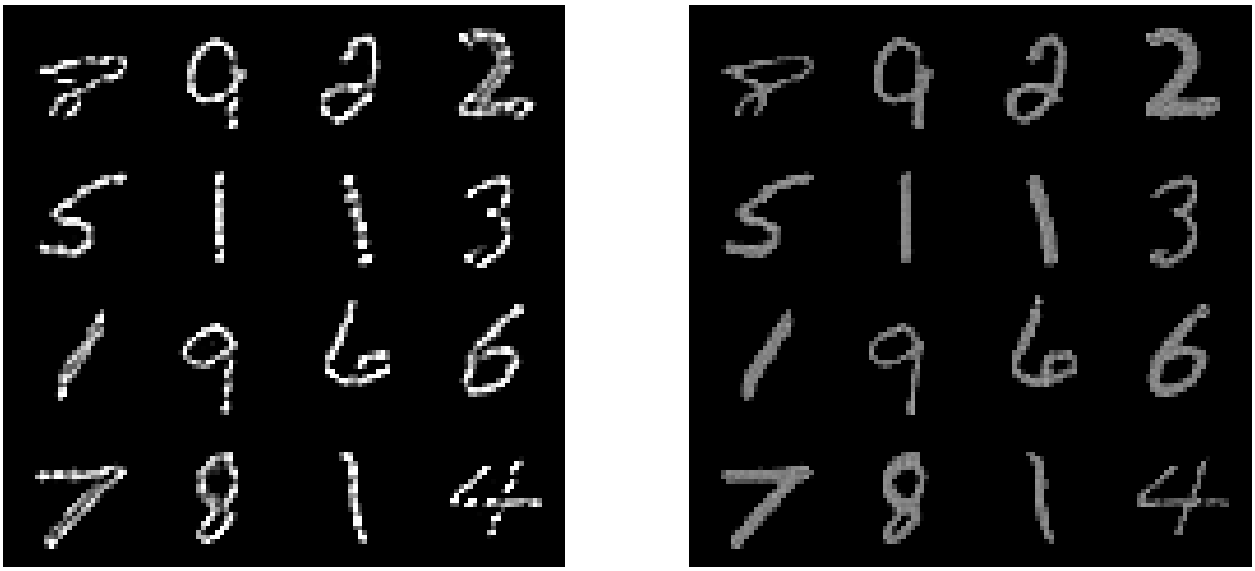
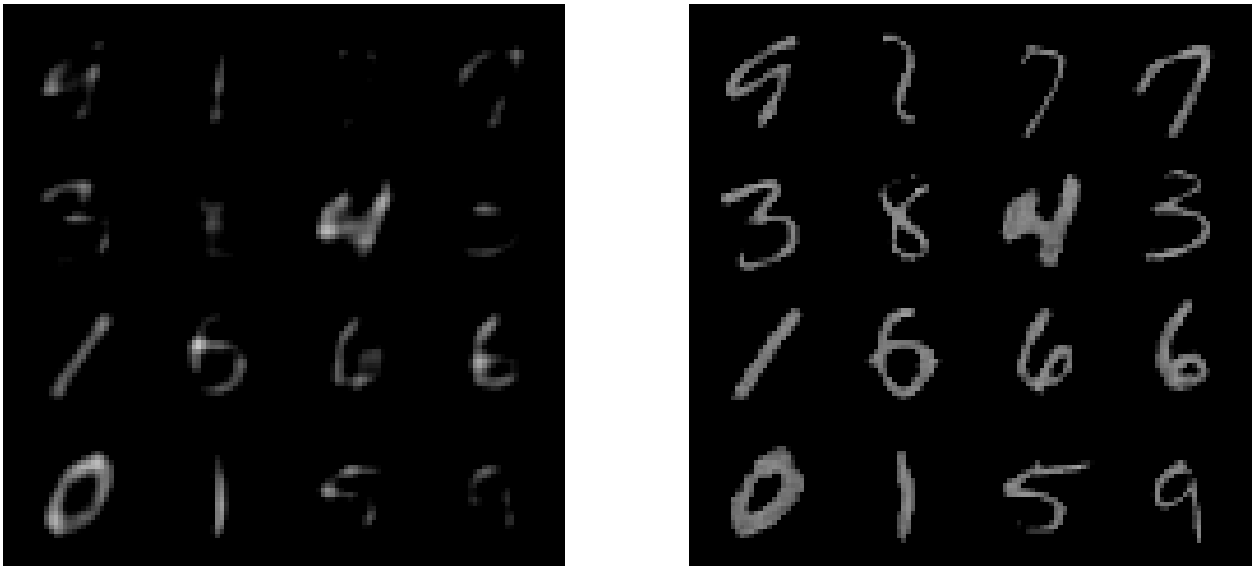


Figure 10: Plot of the training and validation losses for the cold diffusion model trained with a kernel size of 13 and a standard deviation of 7.0



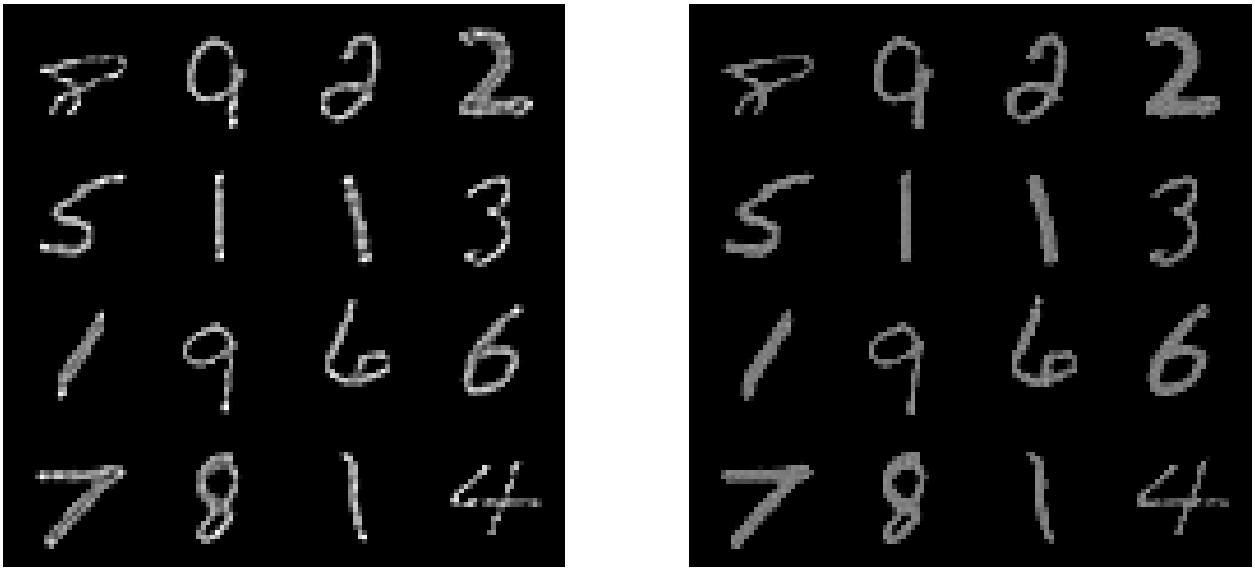
(a) (b)

Figure 11: Comparison of 16 samples generated from the output of the CNN from the cold diffusion model trained with a Gaussian kernel size of 3 and standard deviation of 0.1 at a) epoch 0 and b) epoch 100



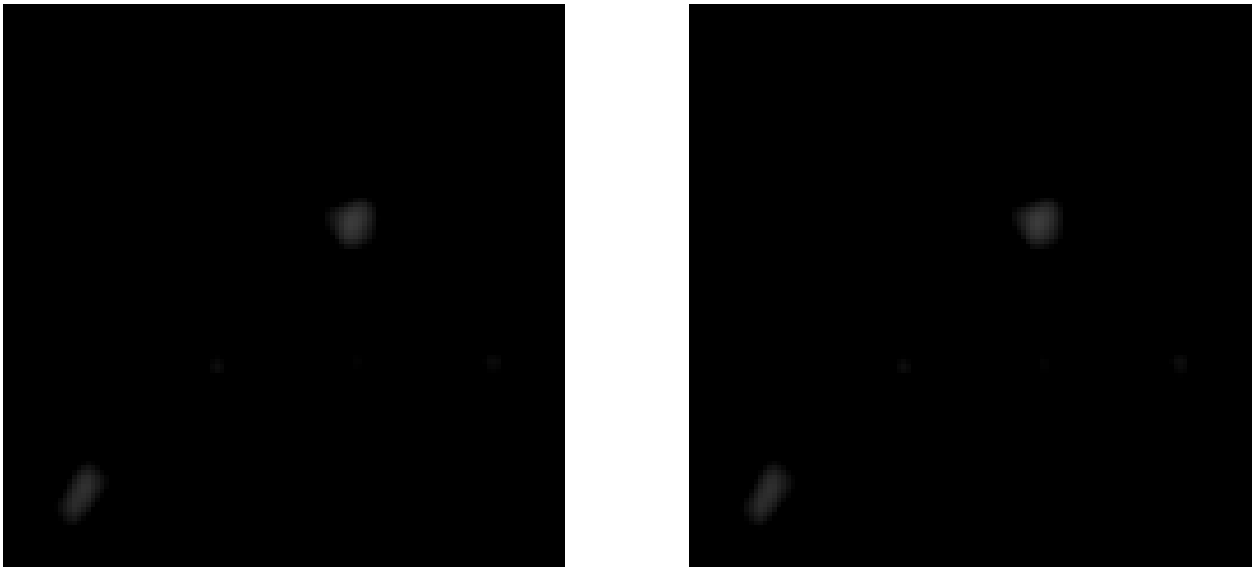
(a) (b)

Figure 12: Comparison of 16 samples generated from the output of the CNN from the cold diffusion model trained with a Gaussian kernel size of 13 and standard deviation of 7.0 at a) epoch 0 and b) epoch 100



(a) (b)

Figure 13: Comparison of 16 samples generated from the sampling method of the cold diffusion model trained with a Gaussian kernel size of 3 and standard deviation of 0.1 after a) 1 epoch and b) 100 epochs



(a) (b)

Figure 14: Comparison of 16 samples generated from the sampling method of the cold diffusion model trained with a Gaussian kernel size of 13 and standard deviation of 7.0 after a) 1 epoch and b) 100 epochs

References

[1] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023. URL: http://udlbook.com.

[2] Arpit Bansal et al. *Cold Diffusion: Inverting Arbitrary Image Transforms Without Noise*. 2022. DOI: 10.48550/arXiv.2208.09392. arXiv: 2208.09392.

[3] Arpit Bansal et al. *Cold Diffusion: Inverting Arbitrary Image Transforms Without Noise*. https://github.com/arpitbansal297/Cold-Diffusion-Models. 2022.

Auto-generation tool citations

GitHub Copilot was used to help write documentation for Doxygen and comments within the code.