Applied data science coursework

Alberto Plebani
ap2387

Number of words: 1848

1 Training a diffusion model

I started the coursework by training a regular denoising diffusion probabilistic model on MNIST.

The MNIST dataset is a dataset including coloured images of numbers, of which an example can be seen in Figure 1. The goal of this part of the project was to build a denoising diffusion model (DDM). The DDM works by adding noise to the images, with the encoder mapping the input x through a series of latent variables $z_1, \ldots, z_T$. The result of this process is having an image with only noise. Afterwards, the reverse process is learned by a decoder, which passes the data back through the latent variables removing noise at each stage. In the end, if we sample an image from the output model, we expect to see the same images that were used to train the model, without the noise.

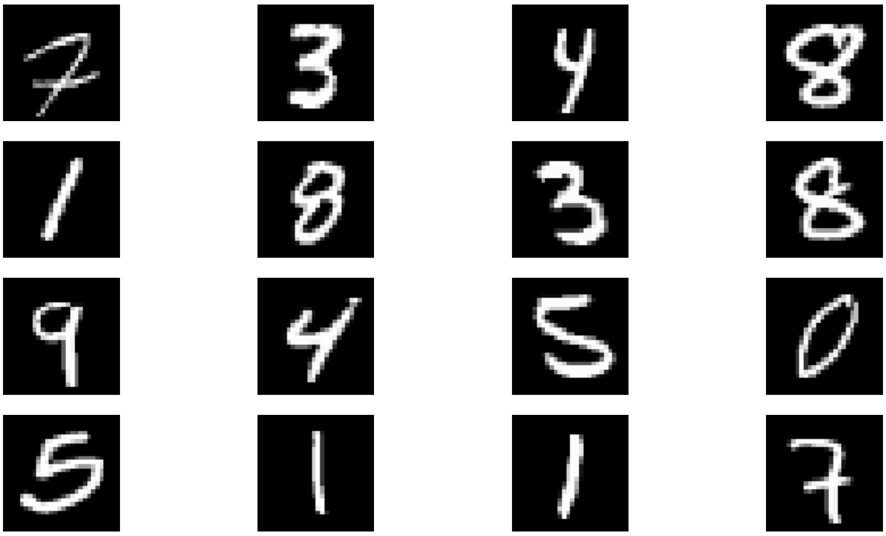


Figure 1: Example MNIST pictures

The training process is iterative, and starts by adding noise to the image at different levels. In the training, both real and noisy images are used, and for each noisy image the model tries to predict the ”true” (not-noisy) image that would have come in the previous step of the diffusion process. The prediction is then compared to the actual clean image, and by doing so the model learns how to remove the noise. This is used in the decoding process, where a noisy image is passed through the latent variables, removing the noise at every stage, eventually generating an image that resembles the original one.

In our case, our model takes a Convolutional Neural Network (CNN) to estimate the diffusion process, with this CNN which is then used to generate the images.

I trained the model using two different sets of hyperparameters, described in Table 1. The results are presented above

Hyperparameter	Set 1	Set 2
Hidden layers	32,64,64,32	32,64,128,64,32
Learning rate	2×10^{-4}	5×10^{-4}
β	$[10^{-4},\ 0.02]$	$[10^{-5},\ 0.01]$
N_T	1000	2000
Epochs	100	100
Patience	15	15
Δ	0.0005	0.0005
Optimiser	Adam	Adam
Activation functions	GELU	GELU
Loss function	MSE	MSE

Table 1: Hyperparameters for the two different models tested.

Set 1

For the first set, I decided to use a simpler model (see Table 1). In this model, the CNN has been built with 4 hidden layers, with 32,64,64, and 32 nodes each. For the learning rate, β, which specifies how noise is added and then removed during the encoding and decoding steps respectively, and the number of diffusion steps N_T, representing how many discrete steps are used to add or remove the noise from the data, the default values provided by the code were used. The NN was allowed to train for up to 100 epochs, with the early stopping technique which was implemented in order to stop the training once it reached convergence. The early stopping was applied once the difference in loss function between epoch i and $i - 15$ was lower than Δ, chosen to be 0.0005. Between the different layers, the Gelu activation function was used, defined as $GELU(x) = x/2 \cdot \big(1 + \mathrm{erf}(x/\sqrt{2})\big)$.

The loss function is represented in Figure 2. We can see how the training stops after 88 epochs, and the loss function converges to a value of 0.019.

In oreder to evaluate the performance of the diffusion model, I used the Structural Similarity between the generated images and the real ones from the MNIST dataset. The Structural Similarity Index (SSIM) considers luminance, contrast and structural information between the two images. I considered evaluating the score after every epoch, to be

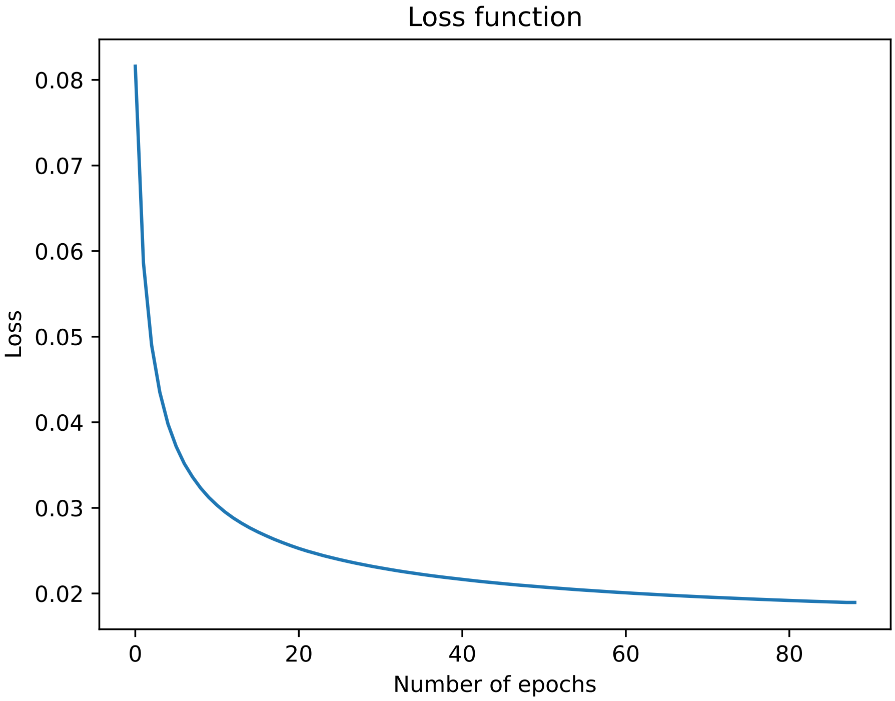


Figure 2: Loss function for Set 1

able to see how much the score improved during the whole training. However, this proved to slow down the code too much, with the time needed for each epoch increasing by more than 3 times. Therefore, I decided to evaluate the score only after the first and the last epoch, to visualise the improvement. For this training, the SSIM score improved from 0.14 after the first epoch to 0.18 after the last one. This proves that the training brought an improvement in the generation of the images. However, the generated images are not really that similar to the original images, as the final score is not really high.

At each step of the training, a set of 16 images was sampled from the distribution. An example of these images is shown in Figure 3. We can see how after 1 epoch we are able to see only . After roughly 10 epochs, it was possible to see symbols consistently, with these symbols looking like numbers only after epoch 22, even though these numbers aren't always clear, as displayed by Figure 3(b). As the number of epochs increased, I was able to identify always more and more numbers, reaching the final epoch (88) where more than half of the numbers sampled are well defined and can be correctly identified. Nevertheless, for some figures it still hard to say which number they are referring to. For instance, the figure in 3rd row and 3rd column could be either a 4 or an 8. Therefore, I decided to use a different set of hyperparameters, to try and improve the performance of the training.

Set 2

In this set, I decided to build a more complex CNN. This CNN has one additional layer of 128 nodes in the middle, resulting in 5 hidden layers, and I also decided to slightly increase the learning rate. Regarding the DDM hyperparameters, I tested a smaller set of values for β. This can lead to a decrease in the convergence time for the NN, but it has the advantage of improving the quality of the generated images. I also doubled the number of diffusion steps N_T, which allows for a more powerful model at the cost of increasing the time needed for training. The other parameters were kept the same as the ones for Set 1.

The loss function for this set is displayed in Figure 4. Once again, the early stopping technique was called once convergence was reached, this time after epoch 84. The loss function converged to a value of 0.018, slightly lower than the previous set.

For this set of hyperparameters, the initial SSIM after the first epoch was 0.13, slightly below the one obtained with the previous set. This might be due to the more complexity of the NN, which then requires more epochs to properly learn something. After the training was complete however, we have a score of 0.21, greater than the one obtained with the previous set, representing an improvement in the diffusion model. This improvement can clearly be seen in Figure 5, where we see that the quality of the generated images is better than in the previous case. In the last image (Figure 5(d)) we can identify all the 16 numbers, ensuring that the NN has learned how to correctly decode the noise better than the otehr one. In this case, we start to see clearer numbers after roughly 16 epochs, comparable with what we obtained with the other set.

2 Section 2

In this section, I implemented a different degradation model. I took inspiration from the 'Bansal et al (2022)' paper, where they implemented the `Inpatinting` tecnhique. This involves removing some pixels from the image using a Gaussian mask. I decided to do something similar, selecting a number of pixels with a Bernoulli mask, and then modifying the luminance of the pixels by a random value between 0 and 1. The modified luminance is centred around the original mean luminance of the image, and the range in which the luminance is varied is chosen by the user when running the code.

This method should represent an improvement with respect to the Bansal method of Inpatinting, because it retrieves information also on the pixels which the Inpatinting method would drop. The implementation of the model is presented in HelperFunctions.py file. The code checks if the image is an RGB image or if it is in grayscale, and in the former case it converts it to a single-channel grayscale tensor. Afterwards, noise is generated randomly, sampling from a uniform distribution in the $[0, 1]$ range. A dropout mask is then implemented. This mask returns values following the distribution in Equation (1), where p refers to the probability, d is the dropout and n is the noise.

$$mask = \begin{cases} 1 \text{ with } p = d \cdot n \\ 0 \text{ with } p = 1 - d \cdot n \end{cases} \tag{1}$$

This mask determines to which pixels the degradation is applied and to which ones it isn't. For the ones for which the mask is 1, I defined an adjust factor, which tells how much the luminance of the pixel is changed. This factor is sampled from a uniform distribution in the selected range, which can be provided by the user. Finally, for pixels for which the mask is 1, this adjust factor is rescaled so that it is centred around the mean luminance of the image, and this correction is then summed to the pixel luminance.

Similarly to before, I trained my diffusion model on the MNIST dataset. I decided to use the same set of hyperparameters that provided the best results for the previous degradation model, that is Set 2. Additionally, I set the dropout rate to 0.2, and the range in which the luminance can change to $[-0.1, 0.2]$. I decided to select a non-symmetryc range in order to give more variability to the diffusion model.

The loss function is presented in Figure 6. We can see that convergence is reached after 82 epochs, with a value of 0.02. This value is slightly worse than the one obtained with the standard degradation model.

I evaluated the performance with the SSIM score. The final score was 0.20, compared to a score of 0.11 after the first epoch.

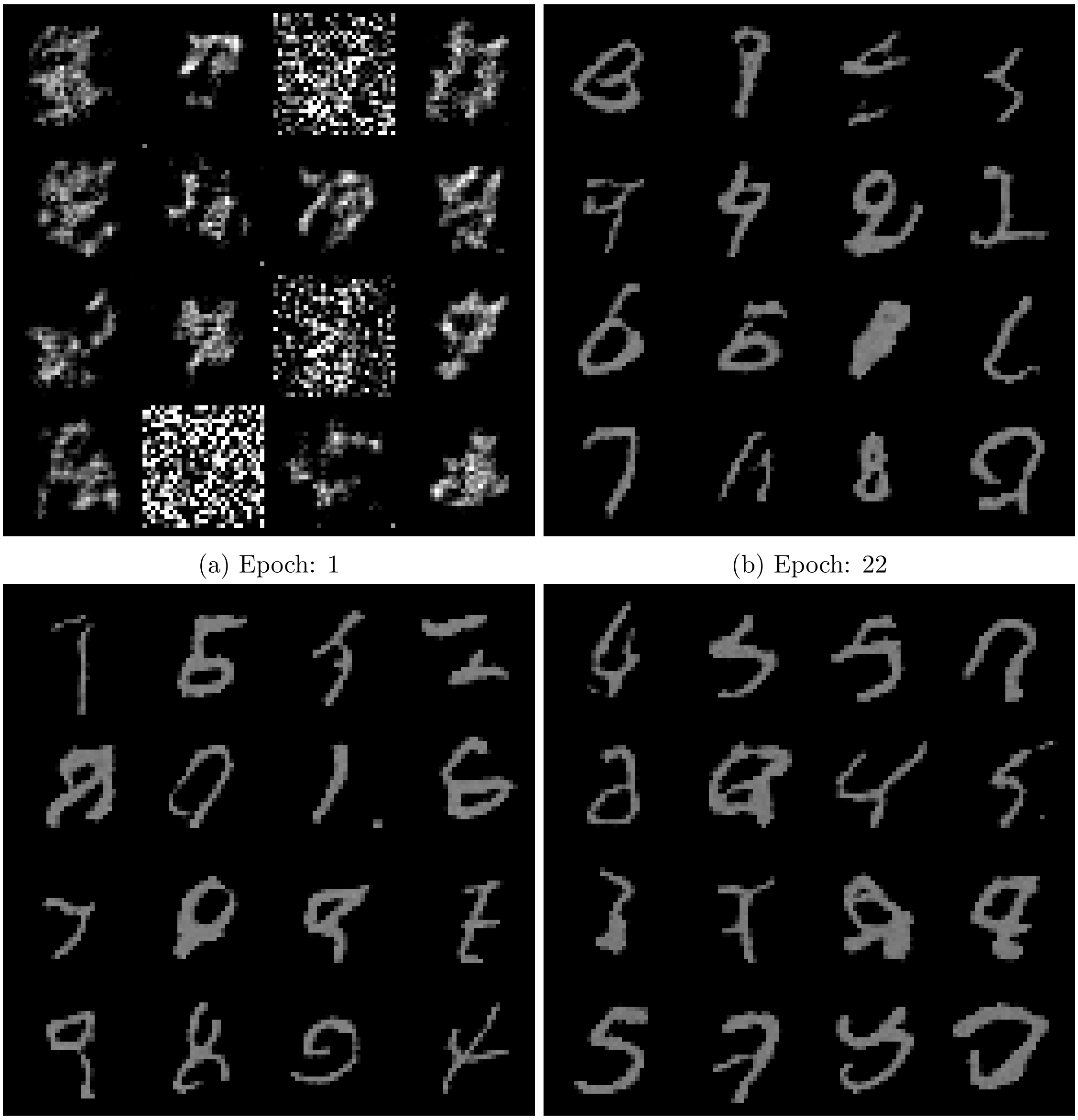
Also for this training, I reported 4 example figures, each containing 16 sampled images. We can see how after the first epoch we see less noise than in the previous two with the standard degradation (Figure 3, 5), and we can also note that there are more bright spots, due to the change in luminance applied by my personal degradation model. In this case it is really hard to identify any image here, even less than it was possible for the other two sets, as represented by the even lower SSIM score. Similarly to the previous two cases, we start seeing clear symbols consistently after roughly 15-20 epochs. However, this time it was harder to identify correctly the numbers in the generated images. We can see how after the last epoch, only 4 of the 16 numbers can be correctly identified, ensuring that this training performed worse than the previous one with the standard degradation model.

Conclusions

In summary, I trained three different models: two using the standard degradation algorithm, changing the hyperparameters in order to have a more complex structure in the second case, and one using my personal degradation model, which works by changing the luminance of some pixels in the image.

I expected the latter model to perform the best, as I have taken inspiration from the 'Bansal et al (2022)' paper, modifying so that information from all the pixels could be retrieved. However, I obtained a worse result with this method, as can be seen in Figure

8. From there, it is clear how the Set 2 is the best model out of the 3, with the personal degradation model performing similar to the Set 1. This is probably due to the need of a more complex NN for the personal degradation, because of the intrinsic complexity of this model. However, I did not have time or CPU resources to train the model with a more complex set of hyperparameters, given that this already took almost half a day to run.



(a) Epoch: 1

(b) Epoch: 22

(c) Epoch: 65

(d) Epoch: 88 (final)

Figure 3: Examples of sampled images for 4 different epochs, set 1.

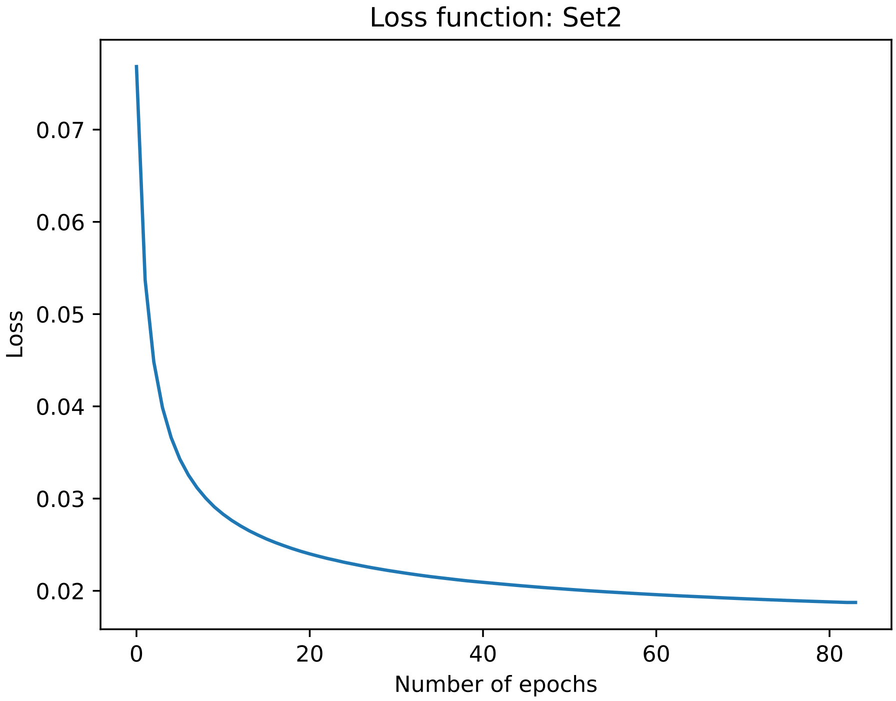
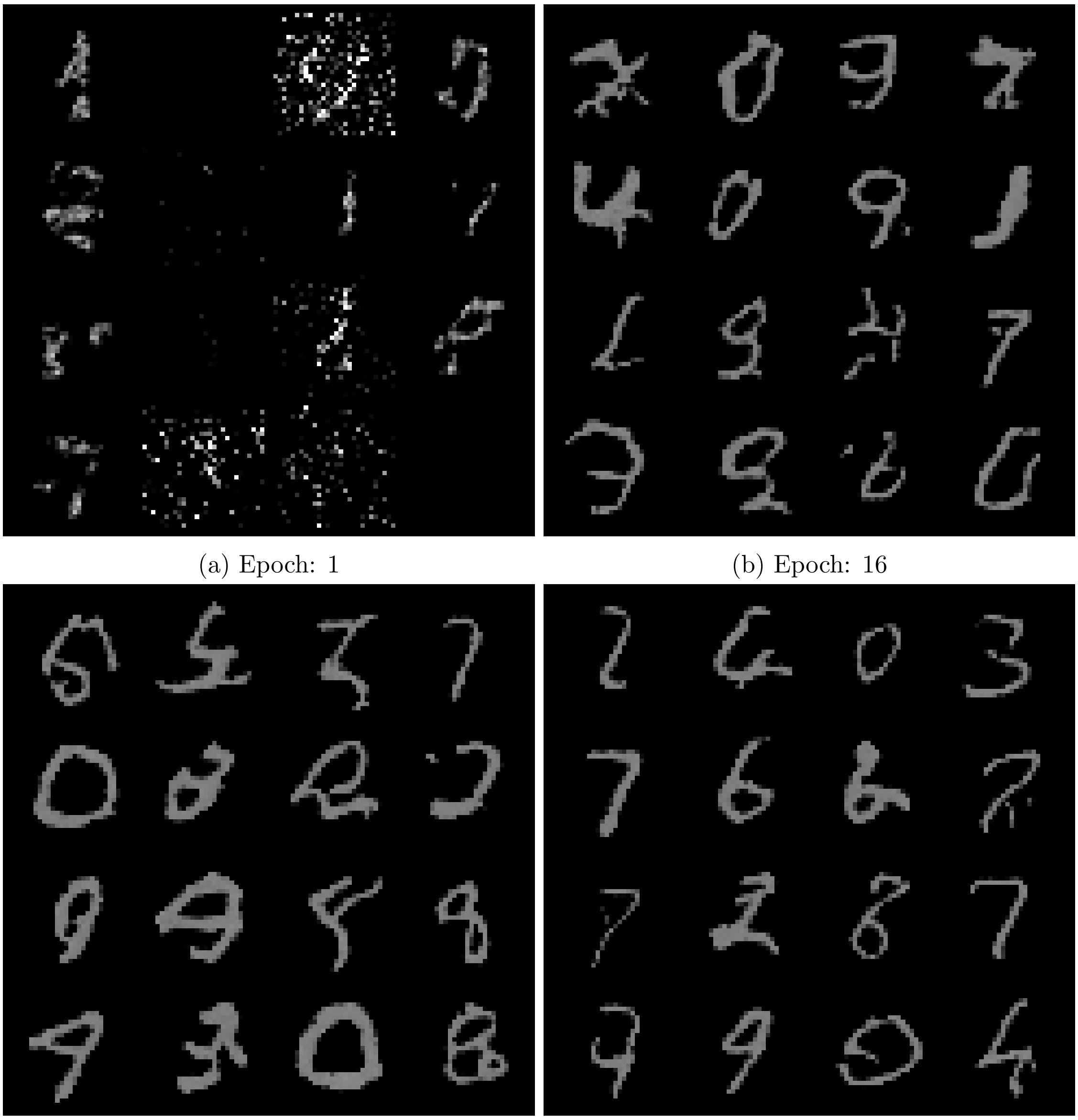


Figure 4: Loss function for set 2



(a) Epoch: 1

(b) Epoch: 16

(c) Epoch: 65

(d) Epoch: 83 (final)

Figure 5: Examples of sampled images for 4 different epochs, set 2.

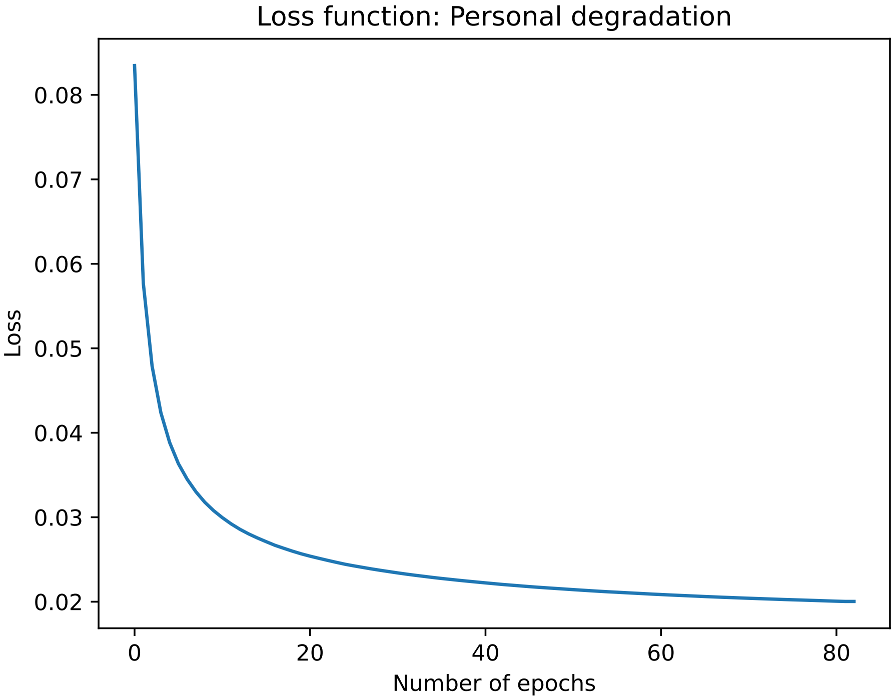
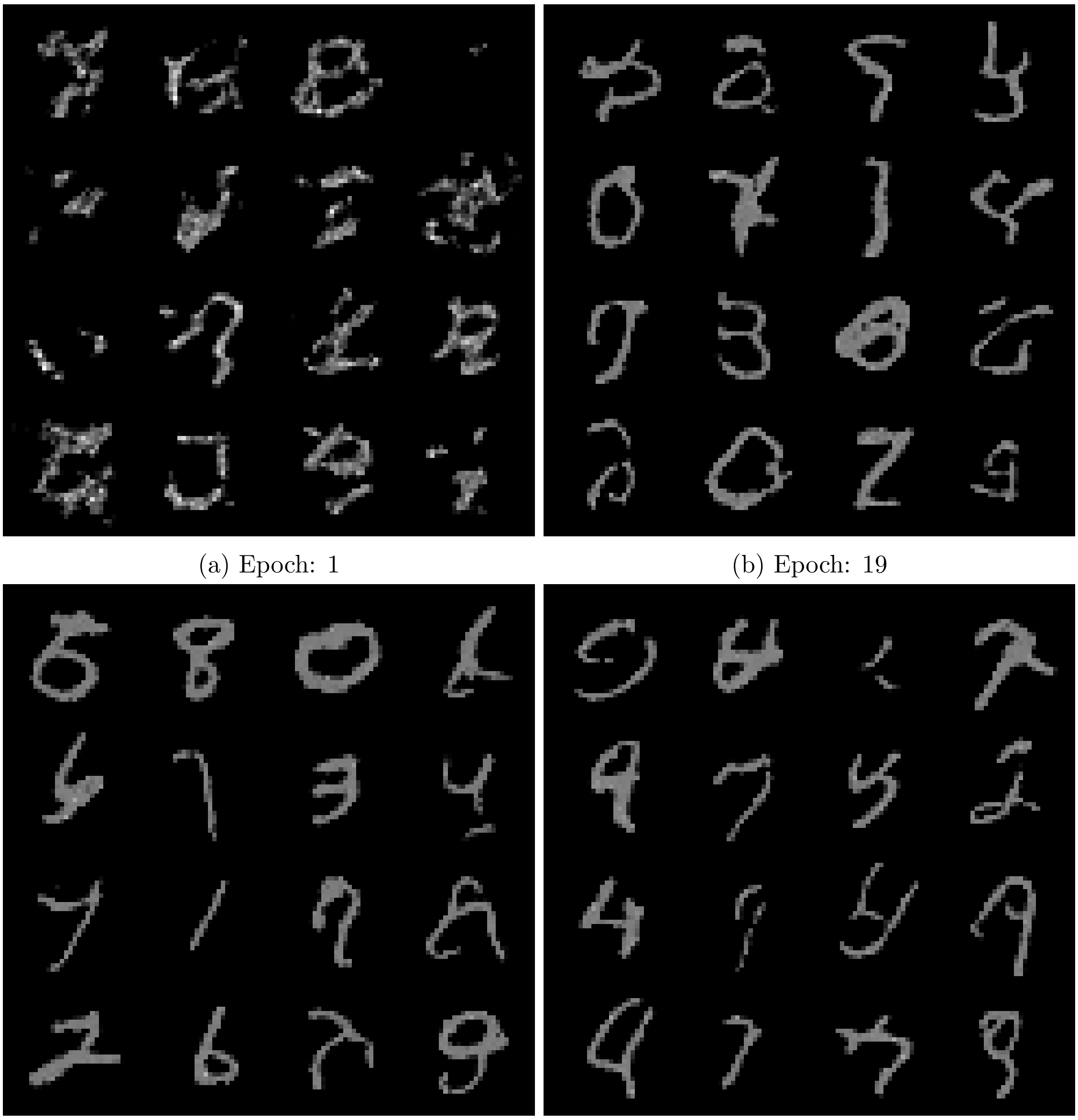


Figure 6: Loss function for personal degradation model



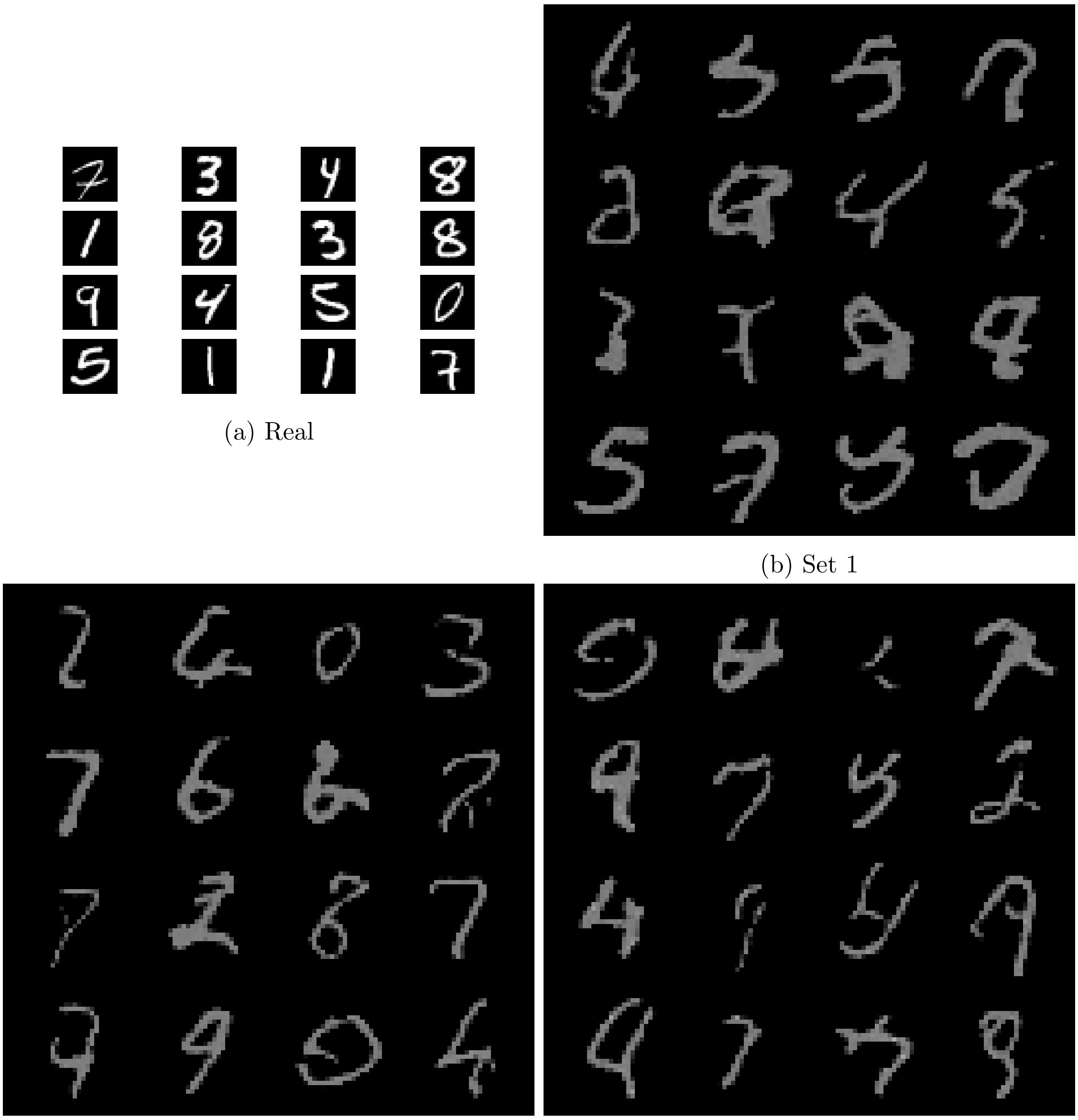
(a) Epoch: 1

(b) Epoch: 19

(c) Epoch: 68

(d) Epoch: 82 (final)

Figure 7: Examples of sampled images for 4 different epochs, personal degradation.



(a) Real

(b) Set 1

(c) Set 2

(d) Personal

Figure 8: Comparison between real (top left) and sampled images at the end of each training for the three different models.