Image generation with diffusion models

Our chosen topic was image generation using diffusion models, where we chose the denoising diffusion model and wanted to generate images using it. For image generation we used Flowers102 and CelebA datasets as training datasets. In the original paper [1], which we observed during the implementation, we used the CelebA and Cifar10 datasets for the same task. We built the diffusion model ourselves, and also created a Gradio interface for it. Our goal was to generate 64x64 images, so the models were built and evaluated in this way.

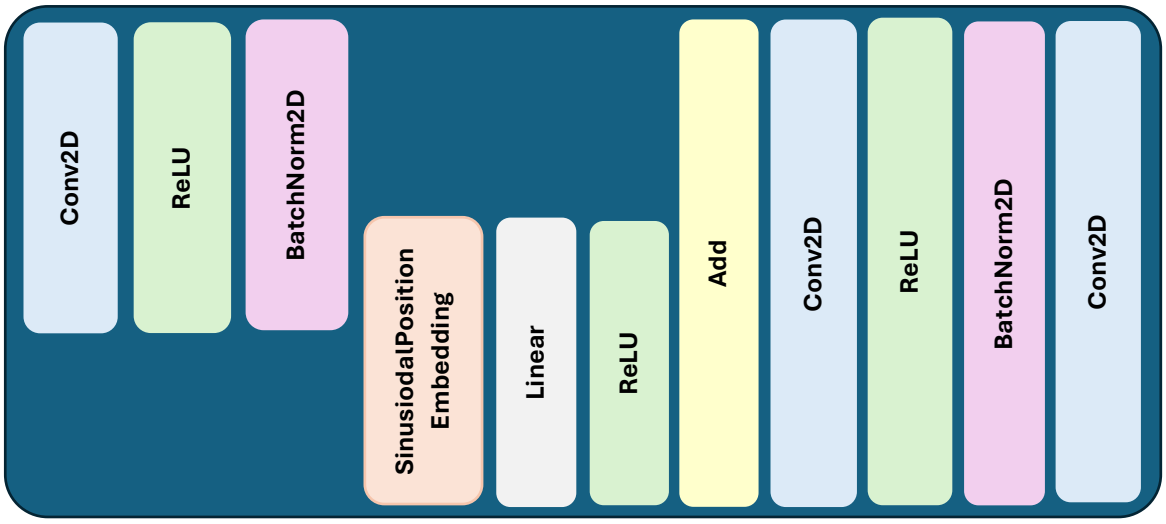
The implementation was based on the article included in the task description, and more specifically on the mathematical model included therein, which defined how the algorithm for noise reduction and sampling is performed for these models. We based our implementation on a YouTube tutorial video [2], where the implementation of the mathematical background on a simple data set was demonstrated in detail, and we also reviewed the Annotated DDPM tutorial [3] in the task description during our implementation.

About datasets

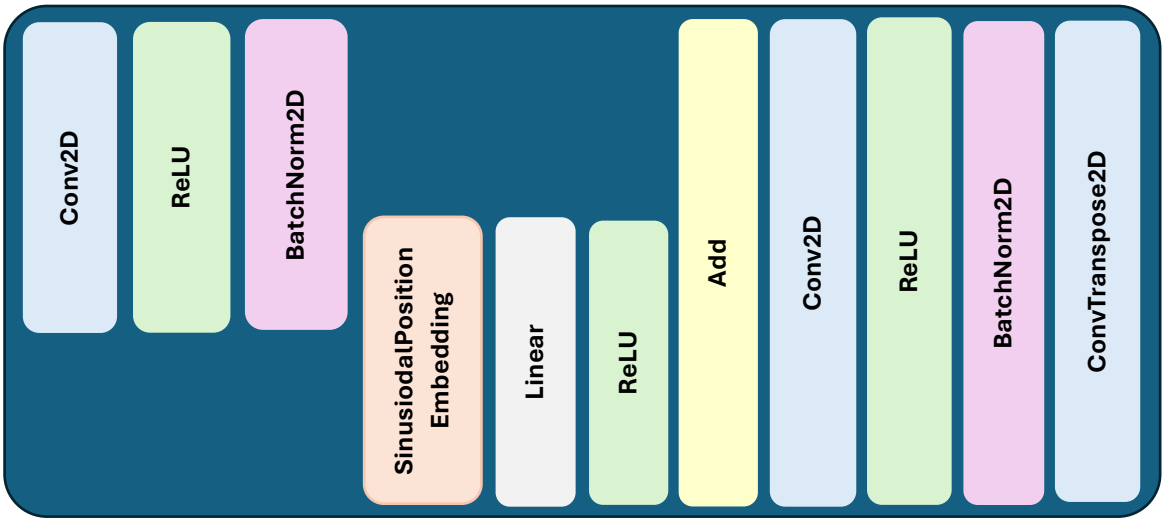
We had trouble collecting the data, as the original CelebA resource is hosted via Google Drive, so it can only be used in limited numbers with the dataset manager in Pytorch (there is a daily limit on downloads, which runs out quickly), instead we had previously downloaded it manually and created a minimal script to unpack the images and copy them into the appropriate folder structure. However, we had to move the dataset several times during the training, so we decided to get these images from elsewhere to avoid unnecessary data movement. Finally, we found a huggingface dataset with the CelebA images [4], which we downloaded from there and used for the next steps.

About architecture

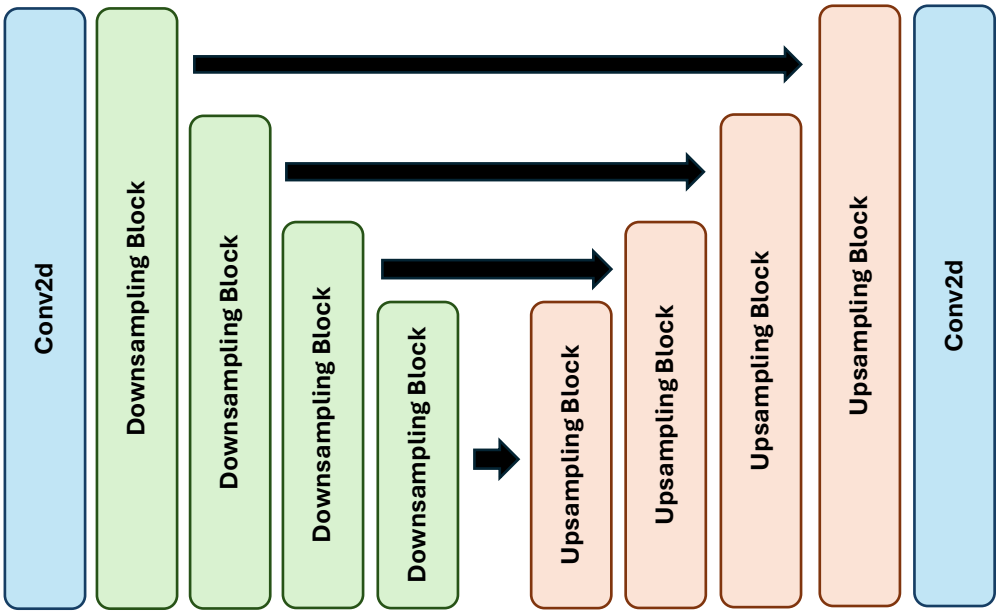
Our implemented model is built from downsampling and upsampling blocks, we didn't use attention blocks separately (we wanted to by default, but for some reason the model didn't learn and we couldn't find the bug, so we kept this simpler solution, as it worked well in principle). The downsampling and upsampling blocks are connected residually, as this guarantees that the deeper layers can learn from a deeper net. The model uses Sinusoidal Position Embeddings to encode temporality, as is commonly used for this model type. The size of the embedding vector was tested during the hyperparameter optimization and the maximum time step was found under it.



1. FIGURE STRUCTURE OF A DOWNSAMPLING BLOCK



2. FIGURE STRUCTURE OF A UPSAMPLING BLOCK



3. FIGURE THE BIG PICTURE OF THE ARCHITECTURE

In the previous figures we have tried to show the architecture we have implemented. The diagrams simplify the architecture and don't show everything in detail, but they give you an overview of how we built our architecture. As we wrote before, we didn't use Attention layers, so we couldn't reproduce the results from the articles, but overall this architecture proved to be sufficient for generating images, as we managed to generate quite nice images.

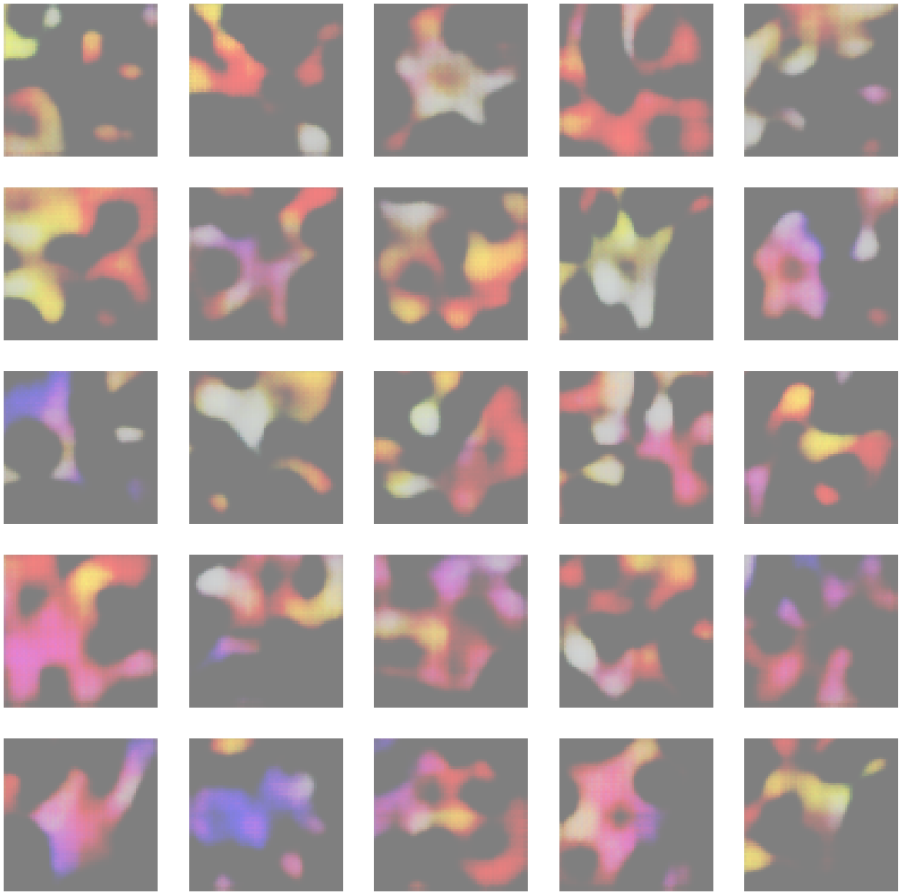
Baseline model

We have chosen the variational autoencoder (VAE) as our baseline model, as we have already implemented it in our previous semester for generating handwritten numbers based on the MNIST database. In the papers we have read, this construct has been overshadowed by diffusion models, so we also thought we would investigate how much difference there will be between the images generated by the two approaches.

The VAE model will be implemented by us based on our existing knowledge, but we do not aim to create a fully optimised one. We want to train the model on the CelebA and Flowers102 datasets we use. We will not implement hyperparameter optimization during the training, but we will use the best model based on the reconstruction loss for the comparison. During training we will use EarlyStopping to prevent over-

learning, as we will be testing on a completely independent subset, so we do not want to be able to map only the images that are included in the training to the latent space.

The VAE model was measured using the basic Inception V3 model to calculate the FID and Inception Score. No hyperparameter optimization was run on the model and it was run for only 10 epochs, during which the model with the best val_loss result was saved.



4. FIGURE THE GENERATED IMAGES FROM LATENT-SPACE (FLOWERS102)

We generated 25 images using random vectors from the latent space, shown in the following figure. You can see that the images are quite blurred, with some flower shapes visible in a few of the images. The images are quite faint. The quality of the images may be due to not running the net long enough to learn the distribution of the images well. It is important to add that for the Flowers102 dataset, we swapped the train and body datasets, since the body dataset contained many more images, so it was better to train the models on it.

5. FIGURE THE GENERATED IMAGES FROM LATENT-SPACE (CELEBA)

Also 25 images were generated from the latent space by selecting random vectors. The result was better than for the flowers, but this can be explained by the fact that we had many more images to train on. Here again, the images are mostly blurred, but we can make out facial shapes. The result is more spectacular than with flowers, but far from perfect, partly due to the small number of epochs.

As a result of short-time learning, there may also be inaccuracies in the metrics for comparison, as the model does not appear to have learned much from the distribution of each image (we did not consider this to be a fault, as there was no expectation to tune this model). But that said, we present the measured values:

Model	Dataset	Loss	FID	Inception mean	Inception std
VAE-baseline	Flowers102	0.2779	8.9862	3.2937	0.1748
VAE-baseline	CelebA	0.2353	4.5525	3.0732	0.0394

Overall, what can be said about the baseline model is that the images do not look very natural (at least the ones generated from the latent space). The FID and IS values were calculated using the image submitted to the model and the predicted image obtained from the model (which, since we are randomly sampling from the latent space between the encoder and decoder layers, can be considered as a generated image at some level).

Methods of training our DDPM model

In preparing the data, simple transformations were implemented using the transforms module functions. Since we didn't want to distort the images, we used CenterCrop on the images (using our knowledge of what the original dimensions of the images were, so we cropped them optimally) and resized

them to the 64x64 size we used. We used color images (already used for the baseline model), so this had 3 channel inputs. After that we did a simple transformation, where we took the values between 0…1 and -1…1 (of course we did the inverse operation for this).

For the training pipeline we used the Pytorch LightningModule [4] for both models, where we created the steps by overlaying the appropriate methods. We basically used the MSE loss for the training optimization. The reason for this was that it was easy to calculate, both for the baseline and the DDPM model. For the former, we computed the KL divergence and the reconstruction loss (MSE), for the latter we computed the MSE only to calculate the difference between the real and the predicted noise.

We also used EarlyStopping, which was adapted to the validation loss (we tried to minimise it). We also used ModelCheckpoint to save the best model during the training. We made the epoch number parameterizable, but basically set it to 50, and in the end, for the DDPM model, we mostly used an epoch of 100. Except for the baseline model, where we only ran it for 10 epochs and since the patience value was higher (20), EarlyStopping was not activated, only the ModelCheckpoint.

6. FIGURE EVOLUTION OF TRAIN LOSS FOR THE FLOWERS102 DATASET

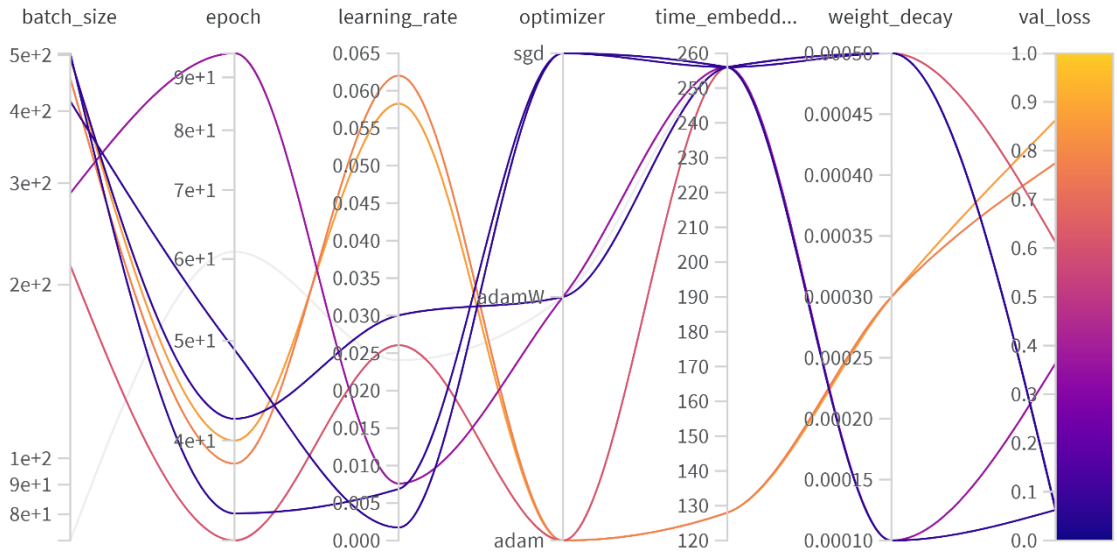
The figure above shows how the value of the train loss evolved when several parameters were changed (hyperparameter optimized using Wandb). Overall, the models converged relatively quickly, but there were some interesting cases where the training stopped at a much worse level, and there were also cases where for some reason the train loss suddenly deteriorated towards the end of a longer training. It is likely that the optimizer tried to overshoot and look for another minimum, but failed, or simply because the noise was random, a situation occurred that the model had not experienced before.

The training was basically done in two places: for the baseline model, we ran the 10 epoch training in Colab, which did not cause any problems there. However, due to the size of our DDPM model and the limits of Colab, it could no longer run there, so we moved it to Komondor, where we had access for another project. On Komondor we trained and performed hyperparameter optimization with only one GPU.

Hyperparameter optimization

The hyperparameter optimization was performed in two ways. We used Wandb to optimize the models based on validation loss, by selecting parameters that we thought might be worth optimizing. But we performed the FID and IS based optimization manually, as these were very lengthy operations, so we only ran it for cases that we found worthwhile to measure from the generated images. However, we tried to be

traceable here and kept an Excel spreadsheet of the parameters and their corresponding results and images.



7. FIGURE PARALLEL DIAGRAM FOR ONE SWEEP WITH OPTIMIZED PARAMETERS

In Wandb, hyperparameter optimization was performed on a Bayesian basis. During the runs, we noticed that the model is sensitive to the size of the embedding vector and that it makes a big difference how large the batches are (this was also important at run-time). For example, the graph shows that in this run, the SGD and AdamW optimizers performed best in terms of validation loss. And it can also be seen that the 256 embedding vector produced better results than the previous 128. The epoch number only affected the result when it was very low, otherwise EarlyStopping and ModelCheckpoint compensated for this.

The same basis was used as a model for both models, but optimisation was performed separately. The results of each run were then used to generate a series of example images. Then the ones that performed nicely at a glance were run through a FID and IS calculation, where the real images and the same number of completely randomly generated images were compared against these metrics.

model	batch size	num of epochs	optimizer	time embedding size	time steps	FID	IS mean	IS std	loss
1	32	100	AdamW	256	500	60,68	1,328	0,021	0,057
2	32	100	AdamW	256	300	19,33	1,527	0,026	0,059
3	32	100	AdamW	256	300	100,41	2,485	0,123	0,055
4	128	95	AdamW	256	300	17,958	1,579	0,054	0,062
5	256	100	AdamW	256	300	14,353	1,738	0,063	0,065
6	256	65	Adam	256	300	15,57	1,562	0,029	0,077
7	128	65	Adam	256	300	8,477	1,628	0,048	0,067
8	128	65	Adam	256	500	25,804	1,7	0,048	0,073

The table above shows the result of the manual evaluation for the Flowers102 dataset. For each result, 25 generated images have been created and uploaded to our repository where they can be viewed. The table shows that we managed to lower the FID value quite a bit, but the IS value remained low, which is not so good. It can also be seen that the loss was not necessarily the best parameter to optimise, as there were cases where it was lower but the FID was worse. We think it is better to compare with the baseline model

in the generated images, as the difference is more visible there. We were able to generate very good flower images and the images are visibly sharper than the VAE.

8. FIGURE GENERATED IMAGES BY DDPM ON FLOWERS102

model	batch size	num of epochs	optimizer	time embedding size	time steps	FID	IS mean	IS std	loss
1	256	34	AdamW	256	300	26,072	1,568	0,006	0,04
2	256	39	AdamW	256	500	18,794	1,186	0,003	0,037
3	256	41	Adam	256	500	16,499	1,605	0,008	0,041
4	256	15	AdamW	256	1000	16,39	1,374	0,0048	0,039
5	512	11	AdamW	512	500	16,606	1,349	0,003	0,038

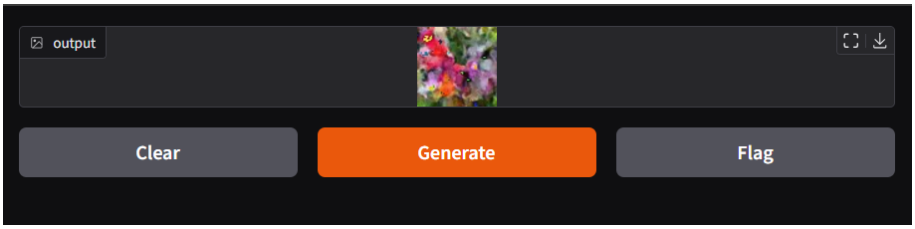
The CelebA dataset had many more images available than Flowers102, but the details of the human faces still make the generated images look weirder. Many of the generated images are quite amorphous and don't really resemble human faces, but there are some quite good ones. Here it might have helped a lot if we had used Attention layers, as they might have brought out the finer details of the human face better. For the flowers, the situation was much simpler (since they are simpler shapes and there are not as many constraints on the shapes as for human faces), so the difference in the results obtained is not a significant difference for the two datasets. Since the same architecture was used for both datasets, it was expected that what would perform acceptably on one could easily bleed on the more complex one.

For CelebA, the FID and IS values obtained do not reach the best result for the Flowers102 dataset. This is also reflected in the realism of the generated images, which look more like pictures painted with some new painting style than photos. This suggests that the current architecture is not sufficient to obtain meaningful results for CelebA. It could probably be remedied by adding layers of Attention or rearranging the architecture of the model.

9. FIGURE GENERATED IMAGES BY DDPM ON CELEBA

AI service with Gradio

Using Gradio, we created a simple interface to generate an image using our best diffusion model. (We don't have a proper GPU and it takes about 5-6 minutes to generate an image on a notebook GTX video card.) The resolution of the generated image is still 64x64, so it's not that spectacular, but we managed to make a working AI service out of the model.



10. FIGURE GENERATED IMAGE IN GRADIO CLIENT

Conclusion

Overall, we have managed to implement a DDPM model that is capable of generating an image, even if in many cases it does not look very realistic. However, during the project work we have learned a lot about generative modelling, as we have not dealt with such problems before. For us, the task was very interesting despite the difficulties. We believe that if we had had more time and resources, we could have improved the results further by incorporating the Attention layer. Perhaps another interesting option would have been to take advantage of the fact that we had tagged our data, or say added image descriptions to the model (our model itself supported it, but we ignored it in the tutorial). We created a Gradio interface where we also made sense of our modeling with an usable image generation service.

References

[1] A. J. P. A. Jonathan Ho, „Denoising Diffusion Probabilistic Models," 2020.

[2] dtransposed, „Diffusion Models - Live Coding Tutorial," YouTube, 2023.

[3] N. R. Kashif Rasul, „The Annotated Diffusion Model," 7 June 2022. [Online]. Available: https://huggingface.co/blog/annotated-diffusion.

[4] „Welcome to ⚡ PyTorch Lightning," [Online]. Available: https://lightning.ai/docs/pytorch/stable/ . [Access date: 1 December 2024].

[5] „Hugging Face - CelebA dataset," [Online]. Available: https://huggingface.co/datasets/nielsr/CelebA-faces. [Access date: 1 December 2024].

An English version of this documentation was produced using DeepL.