Image Generation with Diffusion Models

Márk Somorjai, Zsombor Szommer, Csanád Telbisz

December 2023

This is the documentation of our project work for the Deep Learning course BMEVITMMA19, semester 2023/24/1.

Team

Name: **Lambda**

Members:

Name	Neptun code
Somorjai, Márk	FAFSAG
Szommer, Zsombor	MM5NOT
Telbisz, Csanád	ESV6L2

1 Introduction

The goal of the project is to generate realistic images using diffusion models. We have chosen the *Denoising Diffusion Probabilistic Model* (DDPM) scheme. We have implemented the models based on the following papers and blog posts:

[1] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *CoRR*, abs/2006.11239, 2020. URL https://arxiv.org/abs/2006.11239.

[2] Niels Rogge and Kashif Rasul. The annotated diffusion model, 2022. URL https://huggingface.co/blog/annotated-diffusion.

[3] András Béres. Denoising diffusion implicit models, 2022. URL https://keras.io/examples/generative/ddim/.

[4] Aakash Kumar Nain. Denoising diffusion probabilistic model, 2022. URL https://keras.io/examples/generative/ddpm/.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL http://arxiv.org/abs/1505.04597.

Primarily, we use the Oxford 102 Flower Dataset for the evaluation of our work: we have optimized our model for image generation. On the other hand, we have also tested our implementation with the Oxford-IIIT Pet Dataset. We used GitHub Copilot for code generation in some parts of the implementation.

2 Theoretical Background

The base idea of a diffusion model is to learn how to gradually denoise data step-by-step starting from pure (Gaussian) noise. The model is trained to predict the next step of the denoising process. To achieve this, Gaussian noise is added gradually to the input images, and the neural network learns this noise at each step. That is, the input of the network is a (noisy) image and a noise level, and the network has to predict last added noise. This way, the network can predict the noise component of a given image at each noise level. Thus, if we have T steps such that we end up with (something like) pure noise after adding noise T times, the network can denoise it step-by-step: that is, it can generate an image from pure noise. We use 500 steps in our implementation.

Figure 1 illustrates this process: t represents the noise level, q the noising step, and p_Θ the denoising step (the neural network prediction). The original image is x_0, and its completely noised version is x_T. The goal is to predict x_{t_1} from x_t.

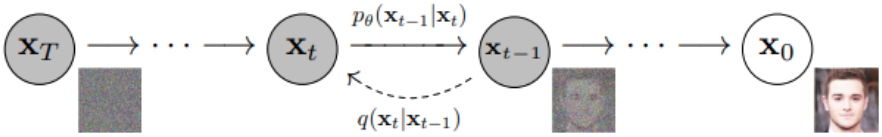

Figure 1: Graphical model of the diffusion process [1]

3 Neural network

The input of the neural network is a noisy image x_t and the noise level t. The output is the predicted noise added at the previous step (from x_{t-1} to x_t). The loss is calculated as the difference between the predicted noise and the actual added noise. Then, the predicted x'_{t-1} is calculated from x_t by subtracting the predicted noise from x_t.

As for the neural network architecture, we have implemented a U-Net [5] which is similar to an *auto-encoder* in the sense it has a "bottleneck". The architecture of the U-Net is illustrated in Figure 2. First, the input data is down-sampled, then up-sampled. At each level, residual blocks are used. The residual blocks are composed of convolutional layers. Furthermore, skip connections are used to connect the down-sampling and up-sampling layers having the same resolution in order to facilitate gradient flow. The skip connections are concatenated with the output of the up-sampling layers.

As the baseline model, we have implemented a simplified version of the U-Net that schematically follows the above architecture but has less convolutional layers. It does not have any optimizations either. Thus, its performance was rather poor.

For the final solution, we have incrementally enhanced the baseline model with the following features:

- We use multiple residual blocks at each level.
- We use group normalization in the residual blocks.
- Sinusoidal time embedding is used to encode the noise level.
- Exponential moving average is used to update the weights of the model.

We have also optimized the hyper-parameters such as the learning rate, the number of epochs, the activation function, etc. To update the model weights using exponential moving average, we create a copy of the neural network (the *EMA network*) and we update its weights after each train step based on its previous weights and the new weights of the trained network.

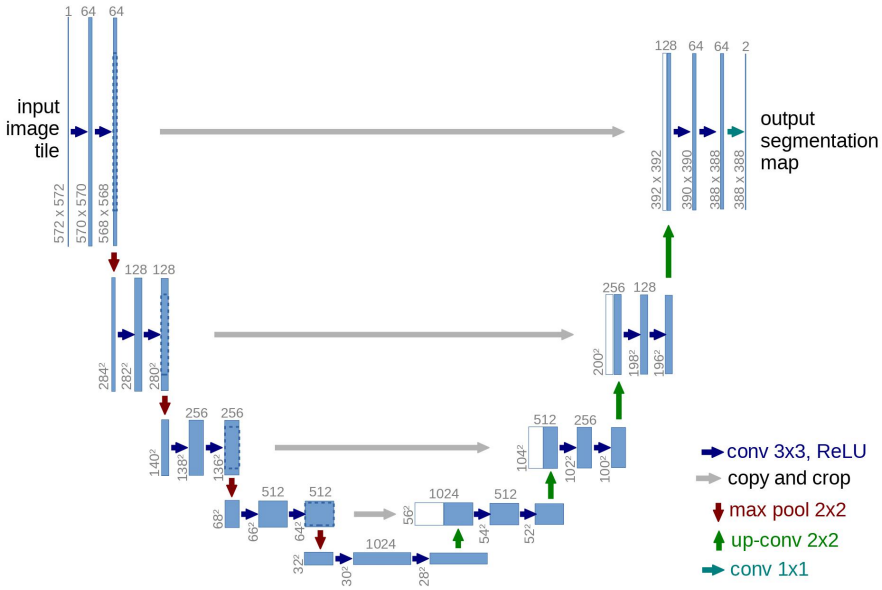

Figure 2: U-Net architecture [5]

4 Implementation

We have implemented the models in Python using TensorFlow. The project is available on GitHub: https://github.com/SPLambda/diffusion. The code is available in the `solution.ipynb` notebook.

4.1 Containerization

We have containerized our solution to make the training process possible to run in a container environment. To build the Docker image, run the following command in your root directory of the repository:

```
docker build -t image-generator .
```

The image build consists of two steps: first, `src/solution.ipynb` is converted to be easily executable from the command line. The resulting Python script is copied to a new image, which is built on top of the `nvcr.io/nvidia/tensorflow:23.09-tf2-py3` image. Additional dependencies are installed from the `requirements_trainer.txt` file. To start the training, run the following command in the root directory of the repository:

```
docker run --gpus all --rm -ti -v $(pwd)/outputs:/app/outputs image-generator
```

Even on relatively powerful hardware, it takes several hours to train the model. The results, including the trained model, will be placed in the `outputs` folder.

4.2 Gradio

In order to test the application, we have also created a web application using Gradio, which can be tested via Hugging Face Spaces. It is important to note that it takes about 90 seconds to generate a single image, so be patient. In return, you can view all the states that occur during generation and adjust them using a slider.

5 Evaluation

The goal of our project is to create a model which is capable of generating images similar to the training dataset. We have planned and carried out the evaluation with this purpose in mind.

5.1 Datasets

Our model and the training process is independent of the used dataset, so we can easily evaluate our work on different datasets. In practice, we have used the Oxford 102 Flower Dataset and the Oxford-IIIT Pet Dataset for evaluation. We have optimized our model on the *flowers* dataset, so we present a detailed evaluation on this dataset. A brief summary of the results on the *pets* dataset is presented in Section 5.6.

Other larger datasets such as the CelebA dataset could be used as well. Unfortunately though, we had limited computing capacity in this project, thus the evaluation on larger datasets has been waived.

The used data is well prepared and cleaned so we did not have much to do with data preparation. We augment the images of the training data by applying a random horizontal flip.

We divide the dataset into a training set and a validation set. We decided not to use a separate test set, because the adjustments we made to the hyperparameters were based on our subjective view of the generated images of the model, i.e., how *flower-like* the generated images looked. Therefore, the hyperparameters were tuned independently from the validation results, hence the various metrics on the validation set are an objective measure of the model's performance. For our smaller, overfit trainings we used 1000-1000 images from the dataset as the training- and test sets, while for our final trainings, we split the dataset 80%-20% as our training- and test sets.

5.2 Evaluation Criteria

For the evaluation of our model, we use both numerical metrics and a subjective rating of the generated images.

We calculate training and validation loss during the training process for the predicted noise of a noising step as described in Section 3. We also track the Kernel Inception Distance (KID) of the model's generated images and the validation dataset, as an objective measure of the quality of the generated images. We chose KID over Frechet Inception Distance (FID) because it is computationally lighter and is less sensitive to the number of samples it is calculated on.

Though the KID metric intends to measure the quality of the generated images, the most obvious way to assess the quality of the generated images is human exploration. Therefore, we include several generated samples in this documentation (see Section 5.4).

5.3 Numerical Results

Figure 3 plots the train and validation loss. The train loss quickly reaches low values while the validation loss takes some epochs to decrease. One reason for the slower convergence of the validation loss is that it is calculated on the *EMA network* whose weights do not change so quickly. The lowest value of validation loss is 0.01702.

Figure 4 displays the Kernel Inception Distance of the model's generated images and the validation set. Though KID is less computationally intensive than FID, the generation of a single batch of images still took about 2 minutes. Thus, we chose to compare only a single batch of generated images to every batch of the validation dataset, and limited the calculation of KID to every 10th epoch. KID takes a bit longer to reach low values than the validation loss, the lowest value being 0.06957.

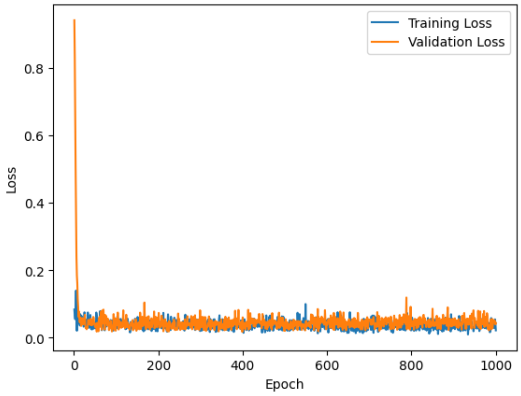

Figure 3: Train & validation loss

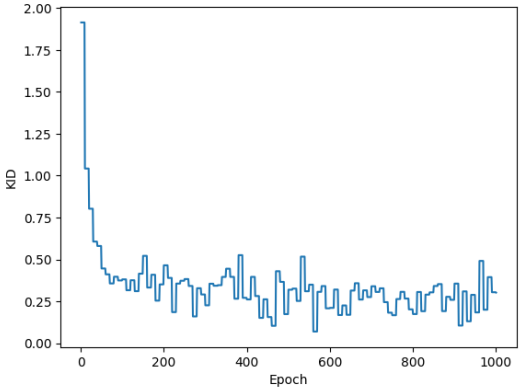

Figure 4: Kernel inception distance (KID)

5.4 Generated images

For generating images, we sample random noise and use our network to gradually denoise it. Figure 5 visualizes the generation process by displaying different steps of the denoising process. The denoising process is available as an animation at link to the gif.

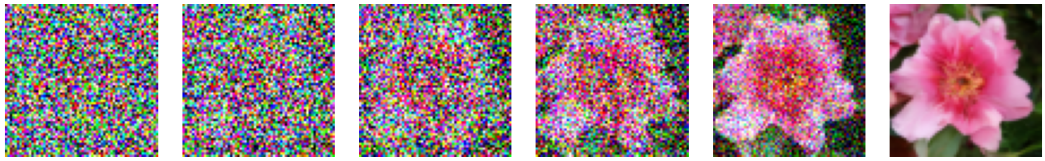

Figure 5: Denoising process

Figure 6 shows a set of generated images. Though not all samples are perfect flowers, the overall quality of the generated flowers are satisfactory.

Figure 6: Generated images

It is also interesting to investigate the generated samples of an overfitted model. The quality of the images in Figure 7 is apparently worse compared to Figure 6. Generated samples are often blurred and the range of generated flowers is less diverse.

Figure 7: Generated images of overfit model

5.5 Reconstructing Noised Images

Even though the goal of our project was not to reconstruct noised images, it is worth examining how well our model is able to reconstruct. For this task, we do not start the denoising process from pure noise rather take an image from the dataset and apply a certain amount of noising steps on it. Then, the model denoises this spoiled version of the image. We can compare the result with the original image. Figure 8 show the noisy, the reconstructed and the original image.

It is also worth taking a look at the reconstructed images of an overfit model on Figure 9. Note, how the reconstructed flower of the right column significantly differs from the original one, presumably because it was the closest image to the original in the training dataset.

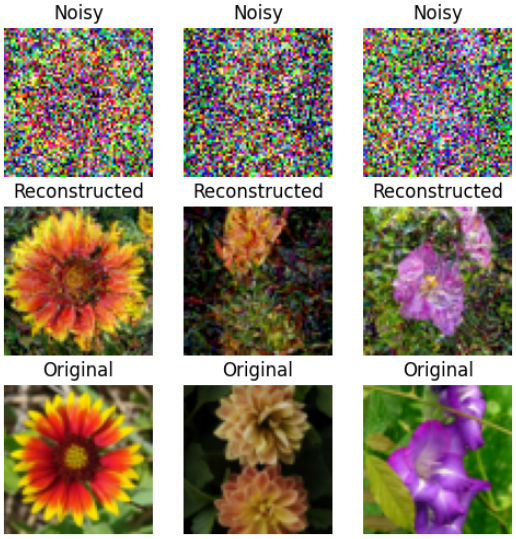

Figure 8: Reconstructed images

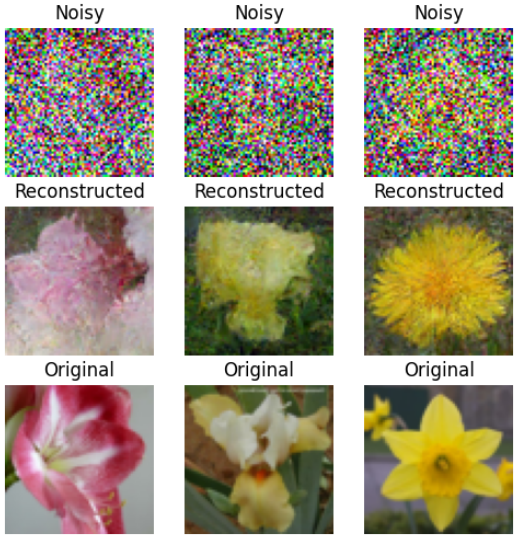

Figure 9: Reconstructed images of overfit model

5.6 Evaluation on the Pets Dataset

We have also evaluated our model on the Oxford-IIIT Pet Dataset. The images in this dataset are visually more complex (e.g., dogs must have exactly two eyes, one mouth, and so on, while a flower

can have many petals, various colors and shapes). Thus, the generated images are less realistic in this case.

Figure 10: Generated images

6 Conclusion

Based on the evaluation results, we can state that we have achieved the goal of our project as our model is able to generate realistic images - at least on the flowers dataset. We briefly summarize the important aspects which helped us optimizing the model and understanding its performance.

6.1 Lessons learned

- In the baseline version, we have chosen a wrong activation function (`tanh`) for the output layer. This prevented the model from reaching low loss values (it could not go much below 0.2). Removing this activation greatly improved the performance.
- Another considerable improvement was achieved by adding group normalization between the convolutional layers of residual blocks.
- Smaller optimizations have also improved our results: e.g., choosing the right number of epochs, using the AdamW optimizer, or augmenting the training images.

6.2 Ideas to improve the model

The performance of the model could be further enhanced by several improvements in the architecture of the neural network or by the optimization of some hyper-parameters based what we have read in the referenced papers and blog posts. We list some examples:

- Others use attention in their U-Net [1, 2, 4]: this would presumably improve our network.
- We add noise linearly, that is we add noise with the same variance at each noise level. Some suggest that using a more complex schedule for determining the amount of noise added at the noise levels can lead to better results [1, 2].
- Some implementations use kernel initializers for convolutional layers [4]. This could speed up the training.