Diffusion Models

Prof. Anna Rohrbach,
Prof. Marcus Rohrbach,
Jonas Grebe

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Semester 2024 – Multimodal Artificial Intelligence
Deadline Wednesday 12th June, 2024
Sheet 3

Denoising Diffusion Probabilistic Models (DDPMs)

This exercise will give us some hands-on experience with **Deep Generative Models**. As you learned in the lecture, many different types of deep generative models (VAEs, GANs, Normalizing Flows) have different strengths and weaknesses. Over the last few years, the paradigm of diffusion models has dominated the landscape.

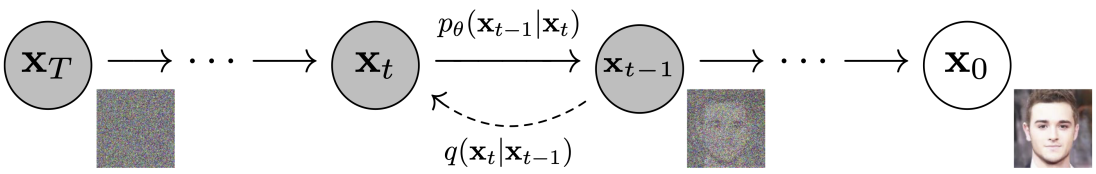


Figure 1: Directed graphical model diagram from the DDPM paper (Figure 2)

This exercise will focus on **Denoising Diffusion Probabilistic Models** (Ho et al. 2020). We have created a Jupyter Notebook for this exercise that you can find on our Multimodal AI Lab Github page and import into your Google Colab. This notebook offers an almost complete implementation for training DDPMs from scratch, with many different hyperparameters for you to explore. There are just a few placeholders to fill out. At the end of this exercise, you will have a solid understanding of how DDPMs work and how they are trained.

Task 3.1: Read the DDPM Paper

We start this exercise by reading the DDPM (Ho et al. 2020) paper. It's a relatively short paper and worth the read. The following exercises will essentially be about implementing the core parts of the DDPM method.

Task 3.2: Data Visualization

This is not a real task that requires coding. Take some time to explore the provided datasets in the notebook. You can already think about which of the datasets you would like to start with for training your DDPM.

Task 3.3: Unconditional Generation

To begin with, we will use the datasets to train DDPMs unconditionally, i.e. without using the class labels. For that, we will have to implement the main components of a DDPM: the forward process, the iterative (denoising) sampling process, and the training loop.

3.3a) Forward Diffusion Process

Implement the forward diffusion process by completing the `sample_xt` function, which takes in the original input x_0, the timestep t, optionally also an already provided noise map ϵ, and an array of $\overline{\alpha}_t$ as arguments, and produces a sample $x_t \sim q(x_t \mid x_0)$.

```python
def sample_xt(x0, t, epsilon, alpha_bars):
    # Your code goes here
    return xt
```

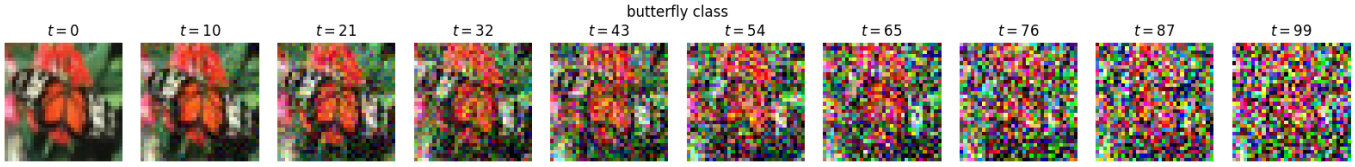
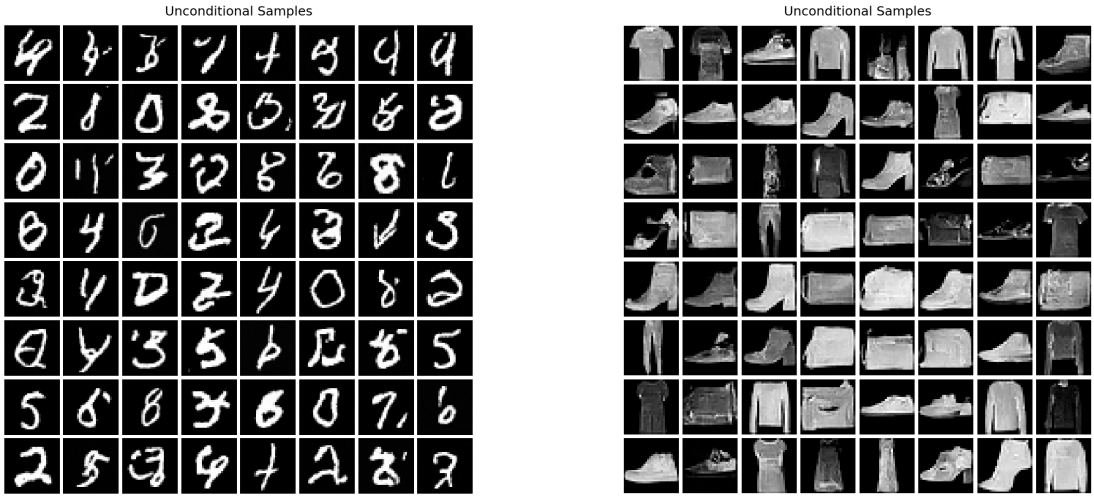


Figure 2: Forward process for a butterfly example image from the CIFAR100 dataset (with $\text{n\_steps} = 100$)

3.3b) Iterative Reverse Sampling

Implement the reverse diffusion process by completing the `sample` function, which takes in our trained DDPM model object and an integer `n_samples` that specifies how many images shall be generated. This function shall implement the iterative sampling procedure with Langevin dynamics described in Algorithm 2 of the DDPM paper.

```python
def sample(ddpm, n_samples):
    # Your code goes here
    return x
```



(a) MNIST dataset

(b) FashionMNIST dataset

Figure 3: Unconditional DDPM samples

3.3c) DDPM Training

Implement the placeholders in the training logic that is already provided. Take a look at Algorithm (1) in the paper for the DDPM training logic.

```python
def train(ddpm, loader, n_epochs, optim, device):
    # Your code goes here
```

Task 3.4: Conditional Generation

For the last task of this exercise, we will slightly change our implementation to make it conditional. Unfortunately, adding natural language text conditions to the notebook is a bit out of scope, especially when training things from scratch. However, every provided dataset is structured into classes, so we can instead use this additional categorical signal to model the reverse process. By conditioning the reverse process on the class information, we can (likely) take control over the generative process after training by simply specifying the class for which we want to generate images.

3.4a) Conditional Sampling

Re-implement the `sample` function for the class-conditional case. This time, we have three parameters: `ddpm`, `n_samples`, and `classes`. Before we only had `n_samples` and simply generated that many images. Now, we want to generate `n_samples * len(classes)` whenever a list of `classes` is provided. It is supposed to be a generalization of the previous sampling function.

```python
def sample(ddpm, n_samples, classes: list[int]=None):
    # Your code goes here
    return x
```

3.4b) Conditional DDPM Training

Re-implement the `train` function for the class-conditional case. Nothing changes despite that the label y is now used from the batch and passed to the DDPM's reverse process.

```python
def train_classconditional(ddpm, loader, n_epochs, optim, device):
    # Your code goes here
```

Task 3.5: Explore!

Explore the hyperparameters and find out what works well for the different datasets. Can you make it work with the more challenging CIFAR100 dataset? You can also try to change the architecture of the ϵ_θ model, i.e. change the parameters of the UNet2D class. Feel also free to share some of your favorite samples in the forum on Moodle.