Advanced Deep Learning
Winter 2024/2025
Mathias Öttl, Jonas Utz, Vincent Christlein, Katharina Breininger

Sheet 2, starting from Nov 26th, 2024, due Dec 20th, 2024, 14:00

Exercise 2.1: Basic structure for Diffusion Models

In this exercise, we want to have a closer look at **Diffusion Models** (or more exact: Denoising Diffusion Probabilistic Models - DDPMs) as a type of widely-used and currently extensively researched method. As before, we will provide you with the main structure for this exercise. We aim to keep the structure across the different exercises somewhat consistent; for conciseness this might not always be possible. Therefore, please make yourself familiar with the provided code in `ex2_main.py`.

In the code skeleton, we will provide you with the basic framework for training a diffusion model. This skeleton will also be the basis for the following tasks in this exercise.

Generally: The trainings for these networks may take some time, so be observant of other users on these machines.

In detail, the main code structure encapsulates the following steps:

1. Loading CIFAR10 dataset
2. Creating an Attention-Unet model architecture w.o. pre-trained weights.
3. Defining a Diffusion object which encapsulates the forward and the backward pass for the diffusion process as well as the loss.
4. Defining the optimizer (AdamW). Default learning rate and beta-parameters can be used to get started.
5. Training loop that trains model for a set number of epochs.

Your task: Make yourself familiar with the given code (`ex2_main.py`, `ex2_model.py` and `ex2_diffusion.py`) to be able to implement all subsequent tasks efficiently and successfully. Pay special attention to the positional encoding in the **Unet** model and the general structure of the `Diffusion` class.

Quota & co: We will provide the datasets for this exercise also in the project folder that we use for the environment. This way, the dataset does not (additionally) add to your quota. You can find the CIFAR10 dataset at `/proj/aimi-adl/CIFAR10/` - you can also find this path in the code skeleton.

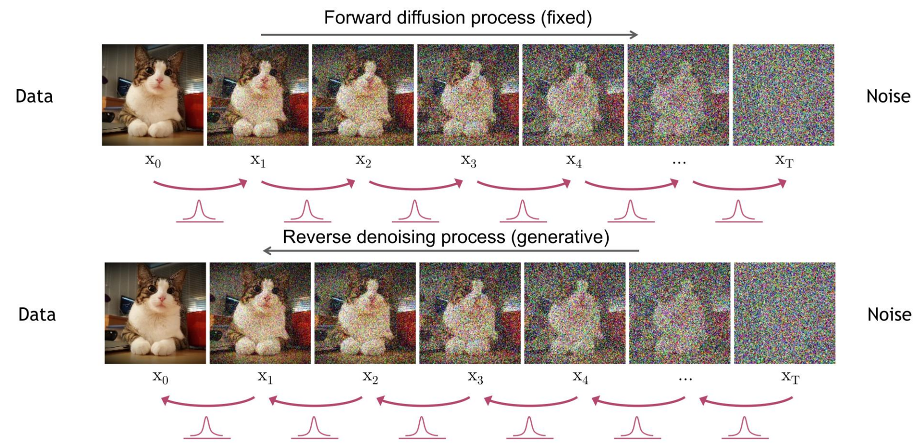
Forward diffusion process (fixed)
Data
Noise
x_0 x_1 x_2 x_3 x_4 ... x_T
Reverse denoising process (generative)
Data
Noise
x_0 x_1 x_2 x_3 x_4 ... x_T

Abbildung 1: Forward (noising) and backward (denoising) process in a DDPM. From `https://cvpr2022-tutorial-diffusion-models.github.io/`.

Exercise 2.2: The Diffusion Process: Forward and Backward

A central aspect for diffusion probabilistic models is the forward and backward process: "Noising" and "Denoising" of the image (see also Fig. 1).

In this exercise task, we want to implement the forward and reverse diffusion process (in a slightly simplified manner compared to the original formulation). For our purposes, we disregard a couple of tricks that make the training more stable and quicker, since we want to achieve a deeper understanding of what the math means for the corresponding implementation.

We will go along the paper by Ho et al.: Denoising Diffusion Probabilistic Models (`https://arxiv.org/abs/2006.11239`). The **core equations** we need to consider in our implementation are the following formulations (we will go into the derivation of the forward and reverse defusion process as well as the loss in detail during the lecture):
Forward diffusion process:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) := \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{1}$$

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{2}$$

The choice of β_t over the different timesteps is called a **beta** or **variance schedule**. The **reparametrization trick** allows us to sample from this distribution as follows:

$$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon_{t-1}, \quad \text{where } \epsilon_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{3}$$

Together with the properties of the Gaussian distribution[1], this allows for a **closed formulation** for sampling at timestep t based on $\mathbf{x}_t$:

$$\mathbf{x}_t = \underbrace{\sqrt{\bar{\alpha}_t}}_{\text{2.2 a)}}\,\mathbf{x}_0 + \underbrace{\sqrt{1-\bar{\alpha}_t}}_{\text{2.2 a)}}\,\epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{4}$$

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_{0}, (1-\bar{\alpha}_t)\mathbf{I}), \tag{5}$$

where $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$. Using this, we are able to sample arbitrary noise levels.

[1] Merging $\mathcal{N}(\mathbf{0}, \sigma_1^2\mathbf{I})$ and $\mathcal{N}(\mathbf{0}, \sigma_2^2\mathbf{I})$ results in $\mathcal{N}(\mathbf{0}, (\sigma_1^2 + \sigma_2^2)\mathbf{I})$

Reverse (”denoising”) process:
As discussed during the lecture, we use the forward diffusion process to generate target data for training a network that performs the reverse diffusion process. This reverse (”denoising”) process looks as follows:

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{6}$$

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) := \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \mathbf{\Sigma}_\theta(\mathbf{x}_t, t)) \tag{7}$$

where $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$.

Following Ho et al., we will set the covariance $\mathbf{\Sigma}_\theta(\mathbf{x}_t, t) = \sigma_t^2 \mathbf{I}$ to a (non-trainable) time-dependent constant where $\sigma_t^2 = \beta_t$. See also [2]. Accordingly, we want to predict only the mean $\mu_\theta(\mathbf{x}_t, t)$.

One core insight: Estimating the noise (and subtracting it from the noisier image $\mathbf{x}_t$) is equivalent to estimating the denoised image directly, but it is easier to learn.

To sample from Eq. 7 and by using the nice property of the Gaussian, we can compute:

$$\mathbf{x}_{t-1} = \underbrace{\frac{1}{\sqrt{\alpha_t}}}_{\text{2.2 a)}} (\mathbf{x}_t - \underbrace{\frac{\beta_t}{\sqrt{1-\bar{\alpha}}}}_{\text{2.2 a)}} \epsilon_\theta(\mathbf{x}_t, t)) + \underbrace{\sqrt{\beta_t}}_{\text{2.2 a)}} \mathbf{z} \ , \tag{8}$$

where $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 0$, else $\mathbf{z} = 0$. This means that the last term $\sqrt{\beta_t}\mathbf{z}$ is set to zero for the first timestep $\mathbf{x}_0$.

We use a neural network to estimate only the noise $\epsilon_\theta(\mathbf{x}_t, t)$, i.e., we provide the network the current noisy image as well as the timestep.

To train this network, we compare the estimated noise $\epsilon_\theta(\mathbf{x}_t, t)$ with the true noise[3] that we added in the forward process.

$$L_{\text{simple}} = \mathbb{E}_{t, \mathbf{x}_0, \epsilon} ||\epsilon - \epsilon_\theta(\mathbf{x}, t)||^2 \tag{9}$$

Your task - Implementation:

a) Complete the `_init_` method of the `Diffusion` class. We recommend that you pre-compute the α_t/α_s etc. to save compute during the forward/reverse step. **Hint:** Have a look at the components in Eq. 4 and Eq. 8 labelled ”2.2 a)”.

b) Implement a full sampling path starting from random noise to data in method `sample`.

c) Implement the forward diffusion process in the method `q_sample` : generate a noisy sample $\mathbf{x}_t$ from a sample $\mathbf{x}_0$ at a timepoint t, using the equations defined above. We provide a unit-test for `q_sample`. If the test fails, please talk to one of the tutors, as this does not necessarily imply incorrect code.

d) Implement the reverse diffusion process and the loss computation:

   – Implement a single reverse step (going from $\mathbf{x}_t$ to $\mathbf{x}_{t-1}$) in `p_sample` based on the equations defined above.

   – Implement the computation of an l2 and an l1 loss based on the noise in the method `p_losses`.

[2]From Nichols and Dhariwal (Improving DDPMs): “One subtlety is that L_{simple} provides no learning signal for $\Sigma_\theta(x_t; t)$. This is irrelevant, however, since Ho et al. (2020) achieved their best results by fixing the variance to $\sigma_t^2 \mathbb{I}$ rather than learning it. They found that they achieve similar sample quality using either $\sigma_t^2 = \beta_t$ or $\sigma_t^2 = \tilde{\beta}_t$, which are the upper and lower bounds on the variance given by $q(x_0)$ being either isotropic Gaussian noise or a delta function, respectively.” Note that Nichols and Dhariwal then show in contrast to this that it is helpful to learn the variance after all.

[3]We use the simplified loss formulation from Ho et al. here as well.

e) Currently, only a training loop is implemented. Implement the test loop in `ex02-main.py` that allows you to quantitatively assess the quality of your images. You can use the loss for a defined test set at specific time-steps as one criterion, but you can also look deeper into possible metrics.

f) Implement visualization helpers that may help you to understand whether the diffusion process works as well as functionality to store the generated images. Optional: Create a couple of animations that show the diffusion / generation process in an animation / gif.
Hint: Please see the provided transformation `reverse_transform` to create an image from the network's output.

Your task - Training the network: Train your network to predict the noise with different settings and compare the results (training convergence, quality of the generated images). Experiment with different parameters (learning rate, number of timesteps, ...). Don't overdo it, as the alternative beta-schedulers can make the training easier. In Figure 2 you can see how your models' generated images may look like.

(a) Car (b) Ship (c) Bird (d) Horse

Abbildung 2: Generated images of a diffusion model trained with classifier-free guidance on CIFAR10.

Exercise 2.3: Alternative beta-Schedulers

One essential element in training diffusion models is how we schedule the noise, also known as **beta** or **variance** schedule. While Ho et al. used a linear schedule, it has been shown that other strategies can result in more robust training of diffusion models and improvements in the generated images, especially for small image resolutions (64×64 or 32×32).

Your task:

- Implement a cosine scheduler in the corresponding function `cosine_beta_schedule`. This schedule is described in the Paper by Nichol and Dhariwal (https://arxiv.org/abs/2102.09672).
- Implement a sigmoid scheduler in the corresponding function `sigmoid_beta_schedule` that follows the following formula

$$\beta_t = \beta_{\text{start}} + \sigma(-s_{\text{limit}} + \frac{2t}{T} s_{\text{limit}}) \cdot (\beta_{\text{end}} - \beta_{\text{start}}) \tag{10}$$

  where s_{limit} allows to select a sensible range from the sigmoid function.
- Create a plot of the different beta-schedulers for the final submission.
- Compare the training with these schedulers compared to the standard linear schedule. What differences can you observe?

Exercise 2.4: Classifier-Free Guidance for DDPMs

So far we have worked with an unconditional model that generates images - but we have little control over how they are generated. As discussed in the lecture, there are different ways of pushing a diffusion model in a specific direction - one recent possibility is **Classifier-Free Guidance** by Ho and Salimans (`https://arxiv.org/abs/2207.12598`).

The core idea is to extend the positional (time) conditioning with an additional class conditioning, which can be used by the model. This information, however, is not always provided to the model during training, but is sometimes replaced by a `null` token. This replacement is random, with a probability p_{uncond} typically chosen as 0.1 or 0.2. This means the authors can train and sample from the same diffusion model by using both in "conditional" and "unconditional" mode.

During sampling / inference, the predictions of the conditional and the unconditional part are then combined:

$$\tilde{\epsilon}_\theta(\mathbf{x}_t, \mathbf{c}) = (1 + w)\epsilon_\theta(\mathbf{x}_t, \mathbf{c}) - w\epsilon_\theta(\mathbf{x}_t) \tag{11}$$

where $\mathbf{c}$ is the corresponding class.

Your task: Implement the classifier-free guidance for your diffusion model by refactoring your code accordingly. Note that your changes should be lightweight and should not break existing code.

a) Adapt the `ex02-model.py` and the `UNet` class with an embedding layer for class conditioning as described in the TODOs in the file. Adapt the `__init__` and the `forward` method. For defining a default `null` token, you can experiment with a trainable vector (`nn.Parameter`) or a 0-vector. Note: The pytorch class `nn.Embedding` may be of considerable help here.

b) Adapt the `Diffusion` class accordingly to take and provide class labels to the UNet.

c) Adapt your training accordingly to be able to train a model with classifier guidance.

c) Adapt and train your model using classifier-free guidance to generate images with different guidance factors (w)

Exercise 2.5: Optional: Butterflies everywhere

As CIFAR10 may be boring, you can try to experiment with a Butterfly dataset from the Smithsonian Butterflies Dataset (`https://huggingface.co/datasets/ceyda/smithsonian_butterflies`). This dataset can be loaded using the huggingface datasets library (`https://huggingface.co/docs/datasets/v1.5.0/quicktour.html`). You can use it with the provided `load_dataset` method. Please replace "`<your_IDM>`" with your actual IDM.

```python
from datasets import load_dataset
dataset = load_dataset("huggan/smithsonian_butterflies_subset", cache_dir="/proj/ciptmp/<your_IDM>/
    smithonian_butterflies/", split="train")
```

Abbildung 3: Example images from the Smithsonian Butterfly Dataset. Source: `https://huggingface.co/datasets/ceyda/smithsonian_butterflies` / Smithsonian Education and Outreach collections.

Note that the images have fairly high resolution (2000×1328px), which can make learning very slow to intractable on the available machines. We recommend cropping / padding to a quadratic size and then downsampling (e.g., to 128×128).

Your task: Experiment with this dataset at your own will, using unconditional / conditional generation. Have fun!

Additional information: Using a prepared conda environment

Since the quota on the cip-pool machines is limited and we may need additional packages during the semester, we have prepared a conda environment `adl23_2` that you can use for this exercise. To use this environment, you can do the following:

Option 1: Activate the environment directly:

```
conda activate /proj/aimi-adl/envs/adl23_2
```

Option 2: Adapt your conda config: Check if you have a file `.condarc` in your home directory (`/.condarc`). If not, run

```
conda config
```

If you have it or after running above command, open the file in your favorite editor and add the following line(s) to the file (amend if you already have the `envs_dirs` option):

```
envs_dirs:
- /proj/aimi-adl/envs/
```

You should then be able to activate the environment using

```
conda activate adl23_2
```

Let us know if you encounter any issues!