Assignment 1 - Diffusion models from scratch

Course: Neural Graphics (3683-8901)
Due: June 19, 2025

Instructions

- In this assignment you will train your own diffusion model on MNIST. Starter code can be found in the provided notebook.
- Start early! This is a challenging assignment, and training models takes time.
- We recommend using GPUs from Colab to finish this project.
- Your final submission should include a zipped folder containing your code (as a `.ipynb` file) and a PDF with your generated plots and typed answers.

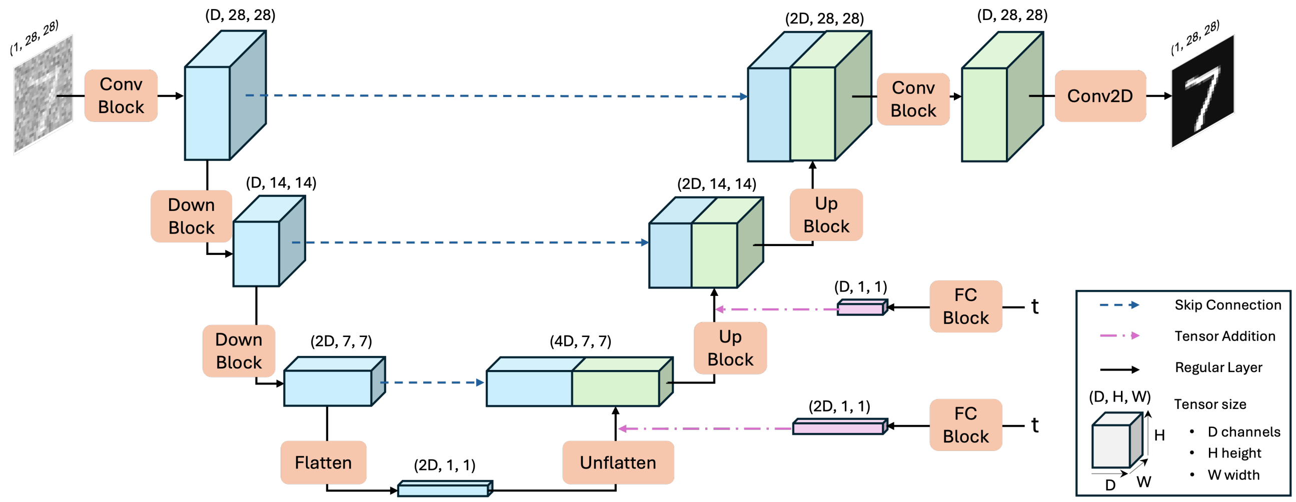

Figure 1: UNet denoiser architecture

1 Basic Ops and UNet Blocks

In this section, you'll implement both basic and composite torch modules that will later serve as building blocks for the denoising network architecture. Detailed documentation for each module is available in the provided notebook — be sure to carefully follow the expected input and output dimensions. Begin by implementing all the basic operations:

1. **Conv** is a convolutional layer that doesn't change the image resolution, only the channel dimension.
2. **DownConv** is a convolutional layer that downsamples the tensor by 2.
3. **UpConv** is a convolutional layer that upsamples the tensor by 2.
4. **Flatten** is an average pooling layer that flattens a 7x7 tensor into a 1x1 tensor. 7 is the resulting height and width after the downsampling operations.
5. **Unflatten** is a convolutional layer that unflattens/upsamples a 1x1 tensor into a 7x7 tensor.
6. **FC** is a fully-connected layer.

After completing the basic operations, proceed to implement the UNet blocks:

8. **ConvBlock** consists of two consecutive Conv ops. Note that it has the same input and output shape as Conv.
9. **DownBlock** consists of DownConv followed by ConvBlock. Note that it has the same input and output shape as DownConv.
10. **UpBlock** consists of UpConv followed by ConvBlock. Note that it has the same input and output shape as UpConv.
11. **FCBlock** consists of FC followed by Linear layer.

2 Unconditional Diffusion Framework

In this section, you will implement and train an end-to-end unconditional diffusion model on MNIST dataset, based on the DDPM framework.

2.1 UNet Architecture

Use the basic ops and blocks to implement the denoiser as a UNet (fill the missing code for **DenoisingUNet**). It consists of a few downsampling and upsampling blocks with skip connections, as depicted in fig. 1. Complete the missing code for the UNet module. The final Conv2D is nn.conv2d(), we leave it to you to decide it's kernel size, stride and padding, as long as input and output dimensions are correct. To implement the skip connections, simply conctenate the tensor along the channel dimenesion. *Tensor addition* refers to element-wise addition of the tensors.

2.2 DDPM Forward and Inverse Process

Now we will guide you through implementing DDPM wrapper for your denoiser. The implementation will follow DDPM training and sampling algorithms (Algorithms 1 and 2 from DDPM paper):

Algorithm 1 Training

```
1: Precompute $\bar{\alpha}$
2: repeat
3:     $\mathbf{x}_0 \sim$ clean image from training set
4:     $t \sim \text{Uniform}(\{1, \ldots, T\})$
5:     $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
6:     $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$
7:     $\hat{\epsilon} = \epsilon_\theta(\mathbf{x}_t, t)$
8:     Take gradient descent step on $\nabla_\theta \|\epsilon - \hat{\epsilon}\|^2$
9: until happy
```

Algorithm 2 Sampling

```
1: Precompute $\beta$, $\alpha$, and $\bar{\alpha}$
2: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
3: for $t$ from $T$ to 1, step size $-1$ do
4:     if $t > 1$ then
5:         $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
6:     else
7:         $\mathbf{z} = \mathbf{0}$
8:     end if
9:     $\hat{\epsilon} = \epsilon_\theta(\mathbf{x}_t, t)$
10:    $\hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\hat{\epsilon}\right)$
11:    $\mathbf{x}_{t-1} = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \sqrt{\beta_t}\mathbf{z}$
12: end for
13: return $\mathbf{x}_0$
```

We provide you with the DDPM recipe to compute $\bar{\alpha}_t$, α and β for $t \in \{1, \cdots, T-1\}$:

- Create a list β of length T, such that $\beta_0 = 0.0001$ and $\beta_T = 0.02$ and all other elements β_t for $t \in \{1, \cdots, T-1\}$ are evenly spaced between the two.
- $\alpha_t = 1 - \beta_t$
- $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$ is a comulative product of α_s for $s \in \{1, \cdots, t\}$

Complete missing code for **ddpm_schedule**, **ddpm_forward** and **ddpm_sample**. We provide you with the wrapper code (DDPM module) which uses those functions. Please follow the documentation, and pay attention to input/output arguments, data types and sizes.

2.3 Train your denoiser

You're almost ready to train your denoiser on the MNIST dataset using the DDPM framework! Fill in the missing parts of the training loop to generate the required deliverables.
We provide a basic training scaffold that handles data loading, optimizer setup and main training loop with gradient updates.
By the end of training, you are expected to submit:

1. **Training loss curves**, showing both batch loss and epoch loss across the full training process. Please specify the axis labels in your plots.
2. **Evaluation plots**, showing sampling results of the denoiser at epochs 1, 5, 10, 15 and 20. Each plot should contain *eval_batch_size* generated samples. It should look something like:

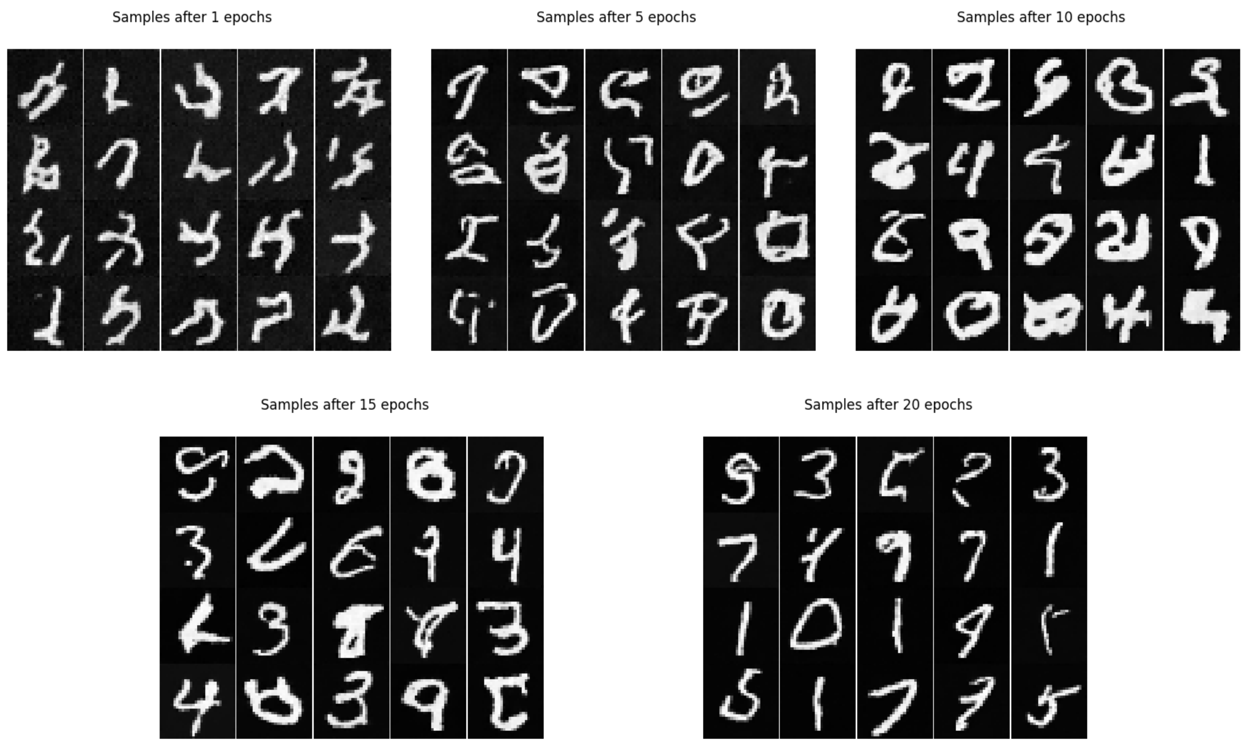

Figure 2: Evaluation plots

Since we are working with MNIST (a simple dataset), you can use fewer diffusion steps - we will set $T = 300$ instead of the full 1000.
You are allowed to modify the training code as you wish. We encourage you to experiment with different hyper parameters and optimizers. However, keep in mind that a model too large is slower to train and might lead to overfitting. Use the default hyperparameters values we provided as a reasonable starting point.
Important Tips:

- As Google Colab GPU time is limited and notebooks runtime tend to reset, we strongly recommend that you checkpoint your model onto your personal Google Drive. This consists of:

  1. Google Drive mounting.
  2. Epoch-wise model & optimizer checkpointing.
  3. Model & optimizer resuming from checkpoints.

  This is not mandatory; it will just make your life easier.

- Track both epoch loss and batch loss — you will need them for your plots.
- Wrap the evaluation code with torch.no_grad() to disable gradient tracking and save memory.
- A full training run should take $\sim$ 30 minutes depending on your network and batch size.
- If your generated samples look like noise after several epochs, stop early and debug your code — something is likely wrong!

3 Implementing class-conditioned diffusion framework with CFG

To improve results better and be able to control generation, we can also optionally condition our UNet on the class of the digit 0-9.
To do so, in this section you will incorporate class-conditioning into the UNet architecture, and then adjust DDPM mechanism to support conditional generation. Finally, you will train your class conditioned UNet and do some fun stuff with it.

3.1 Adding Class-Conditioning to UNet architecture

Extend the UNet architecture in fig. 1 by adding a class-conditioning mechanism (complete the missing code in **ConditionalDenoisingUNet**). Since the denoiser must support both conditional and unconditional generation, it receives two additional inputs at each denoising step: a one-hot class vector and a binary mask indicating whether the label should be used or dropped for each sample in the minibatch. To drop a label, simply replace its one-hot vector with a zero vector, representing the unconditional case. To incorporate the class condition, we recommend the following approach, which involves adding two instances of **FCBlock** to your UNet.:

```
fc1_t = FCBlock(...)
fc2_t = FCBlock(...)
fc1_c = FCBlock(...)
fc2_c = FCBlock(...)

t1 = fc1_t(t)
c1 = fc1_c(c)
t2 = fc2_t(t)
c2 = fc2_c(c)

# Follow diagram to get unflatten (Unflatten block output).
# Replace the original unflatten with modulated unflatten.
unflatten = c1 * unflatten + t1
# Follow diagram to get up1 (First UpBlock output).
...
# Replace the original up1 with modulated up1.
up1 = c2 * up1 + t2
# Follow diagram to get the output.
...
```

You're welcome to explore your own approach for incorporating the class condition, as long as you adhere to the documentation and avoid modifying existing blocks. You may add new blocks if needed.

3.2 DDPM Forward and Inverse Process with CFG

The DDPM forward and reverse processes for class-conditioned setting using CFG are similar to the unconditional case, with a few key differences:

Algorithm 3 Class-conditioned Training

```
1: Precompute $\bar{\alpha}$
2: repeat
3:     $\mathbf{x}_0, c \sim$ clean image & label from train set
4:     Make $c$ into a one-hot vector $\vec{c}$
5:     Set $\vec{c}$ to $\vec{0}$ with probability $p_{uncond}$
6:     $t \sim$ Uniform($\{1, \ldots, T\}$)
7:     $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
8:     $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$
9:     $\hat{\epsilon} = \epsilon_\theta(\mathbf{x}_t, t, \vec{c})$
10:    Take gradient descent step on $\nabla_\theta \|\epsilon - \hat{\epsilon}\|^2$
11: until happy
```

Algorithm 4 Class-conditioned Sampling

```
1: Input: Class condition $c$ and guidance scale $\gamma$
2: Make $c$ into a one-hot vector $\vec{c}$
3: Precompute $\beta$, $\alpha$, and $\bar{\alpha}$
4: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: for $t$ from $T$ to 1, step size $-1$ do
6:     if $t > 1$ then
7:         $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
8:     else
9:         $\mathbf{z} = \mathbf{0}$
10:    end if
11:    $\epsilon_{cond} = \epsilon_\theta(\mathbf{x}_t, t, \vec{c})$
12:    $\epsilon_{uncond} = \epsilon_\theta(\mathbf{x}_t, t, \vec{0})$
13:    $\hat{\epsilon} = \epsilon_{uncond} + \gamma(\epsilon_{cond} - \epsilon_{uncond})$
14:    $\hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\hat{\epsilon}\right)$
15:    $\mathbf{x}_{t-1} = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \sqrt{\beta_t}\mathbf{z}$
16: end for
17: return $\mathbf{x}_0$
```

Complete missing code for **ddpm_forward** and **ddpm_cfg_sample**. For training algorithm, drop class labels from each minibatch with probability of $p_{uncond} = 0.1$. Please follow the documentation and pay attention to input/output arguments, data types, and sizes. As in section 2.2, we have provided the DDPM wrapper code for you.

3.3 Train your class-conditioned denoiser

Implement the training loop for your conditional diffusion model. Unlike before, no starter code is provided this time. Begin with the training loop you used for the unconditional denoiser, and adapt it to incorporate class conditioning. At the end of the training process, you are expected to submit:

1. **Training loss curves**, showing both batch loss and epoch loss across the full training process. Please specify the axis labels in your plots.

2. **Evaluation plots**, showing sampling results of the denoiser at epochs 1, 5, 10, 15 and 20. For evaluation plots, use guidance scale $\gamma = 5.0$. Each plot should contain *eval_batch_size* generated samples of class-labels minibatches **drawn from MNIST test set**. Present each sample next to it's class-label. It should look something like:

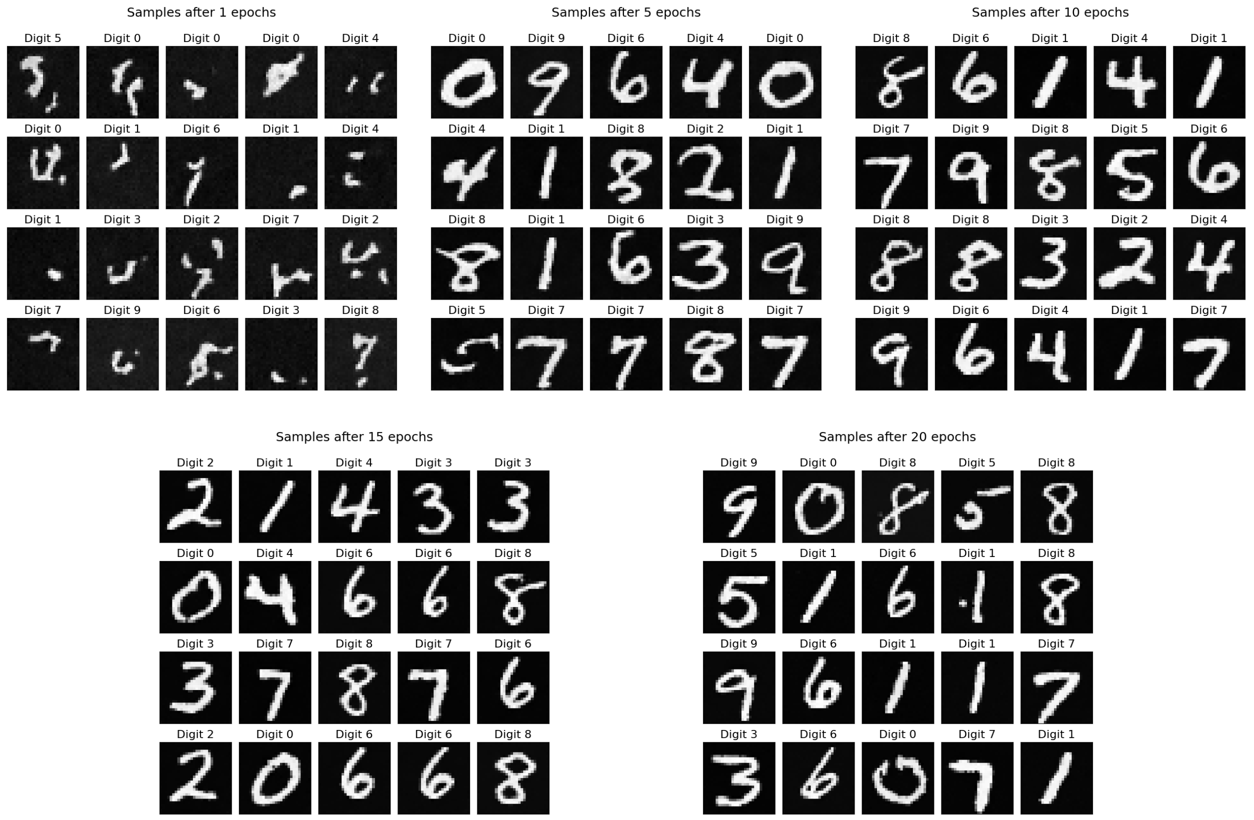
Samples after 1 epochs
Digit 5 Digit 0 Digit 0 Digit 0 Digit 4
Digit 0 Digit 1 Digit 6 Digit 1 Digit 4
Digit 1 Digit 3 Digit 2 Digit 7 Digit 2
Digit 7 Digit 9 Digit 6 Digit 3 Digit 8
Samples after 5 epochs
Digit 0 Digit 9 Digit 6 Digit 4 Digit 0
Digit 4 Digit 1 Digit 8 Digit 2 Digit 1
Digit 8 Digit 1 Digit 6 Digit 3 Digit 9
Digit 5 Digit 7 Digit 7 Digit 8 Digit 7
Samples after 10 epochs
Digit 8 Digit 6 Digit 1 Digit 4 Digit 1
Digit 7 Digit 9 Digit 8 Digit 5 Digit 6
Digit 8 Digit 8 Digit 3 Digit 2 Digit 4
Digit 9 Digit 6 Digit 4 Digit 1 Digit 7
Samples after 15 epochs
Digit 2 Digit 1 Digit 4 Digit 3 Digit 3
Digit 0 Digit 4 Digit 6 Digit 6 Digit 8
Digit 3 Digit 7 Digit 8 Digit 7 Digit 6
Digit 2 Digit 0 Digit 6 Digit 6 Digit 8
Samples after 20 epochs
Digit 9 Digit 0 Digit 8 Digit 5 Digit 8
Digit 5 Digit 1 Digit 6 Digit 1 Digit 8
Digit 9 Digit 6 Digit 1 Digit 1 Digit 7
Digit 3 Digit 6 Digit 0 Digit 7 Digit 1

Figure 3: Evaluation plots of class-conditioned denoiser

All the tips from section 2.3 are relevant for the conditional case as well.

3.4 Experimenting with different guidance scales

In this section, you will explore how varying the guidance scale impacts the generated images. Use your trained model to sample digits 0–9 with guidance scales of [0, 1, 5, 7, 10, 15]. Analyze and describe the visual effects of using guidance values that are too low or too high, and explain your observations based on the concepts discussed in class. Include both the generated images and your written analysis in your final submission.