IFT6135A – W2025

Assignment 3: Practical Part

Université de Montréal

Felix Wilhelmy

October 4, 2025

Due Date: May 2, 23:00

Instructions

- For all questions that are not graded only on the answer, show your work! Any problem without work shown will get no marks regardless of the correctness of the final answer.
- Please try to use a document preparation system such as LaTeX. *If you write your answers by hand, note that you risk losing marks if your writing is illegible without any possibility of regrade, at the discretion of the grader.*
- Submit your answers electronically via the course GradeScope. *Incorrectly assigned answers can be given 0 automatically at the discretion of the grader.* To assign answers properly, please:
  - Make sure that the top of the first assigned page is the question being graded.
  - Do not include any part of answer to any other questions within the assigned pages.
  - Assigned pages need to be placed in order.
  - For questions with multiple parts, the answers should be written in order of the parts within the question.
- In the code, each part to fill is referenced by a TODO and 'Not Implemented Error'
- Questions requiring written responses should be short and concise when necessary. Unnecessary wordiness will be penalized at the grader's discretion.
- Please sign the agreement below.
- It is your responsibility to follow updates to the assignment after release. All changes will be visible on Overleaf and Piazza.
- Any questions should be directed towards the TAs for this assignment: *Vitória Barin Pacela, Philippe Martin.*

For this assignment, the GitHub link is the following: `https://github.com/philmar1/Teaching_IFT6135---Assignment-3---H25`

I acknowledge I have read the above instructions and will abide by them throughout this assignment. I further acknowledge that any assignment submitted without the following form completed will result in no marks being given for this portion of the assignment.

Name: **Félix Wilhelmy**

UdeM Student ID: **20333575**

Signature: _____

Introduction

All experiments were conducted on Google Colab utilizing T4 GPUs. All models were trained for 20 epochs; however, due to a saving bug, checkpoints for DDPM and CFG models at epoch 20 were not properly saved. Consequently, the results presented for these models correspond to epoch 19.

Question 1 : Variational Autoencoder (VAE)

Question 1.6 : Training & Validation Loss

The loss curves for both training and validation exhibit a steady decline over the 20-epoch run, ultimately converging to final values of 103.8882 (training) and 103.6296 (validation), thereby meeting the prescribed target of 104. As shown in Figure 1, the two curves track closely throughout, indicating stable optimization without significant overfitting. The plateau observed toward the final epochs further confirms consistent learning dynamics across both datasets, with no divergence suggesting under- or overfitting.

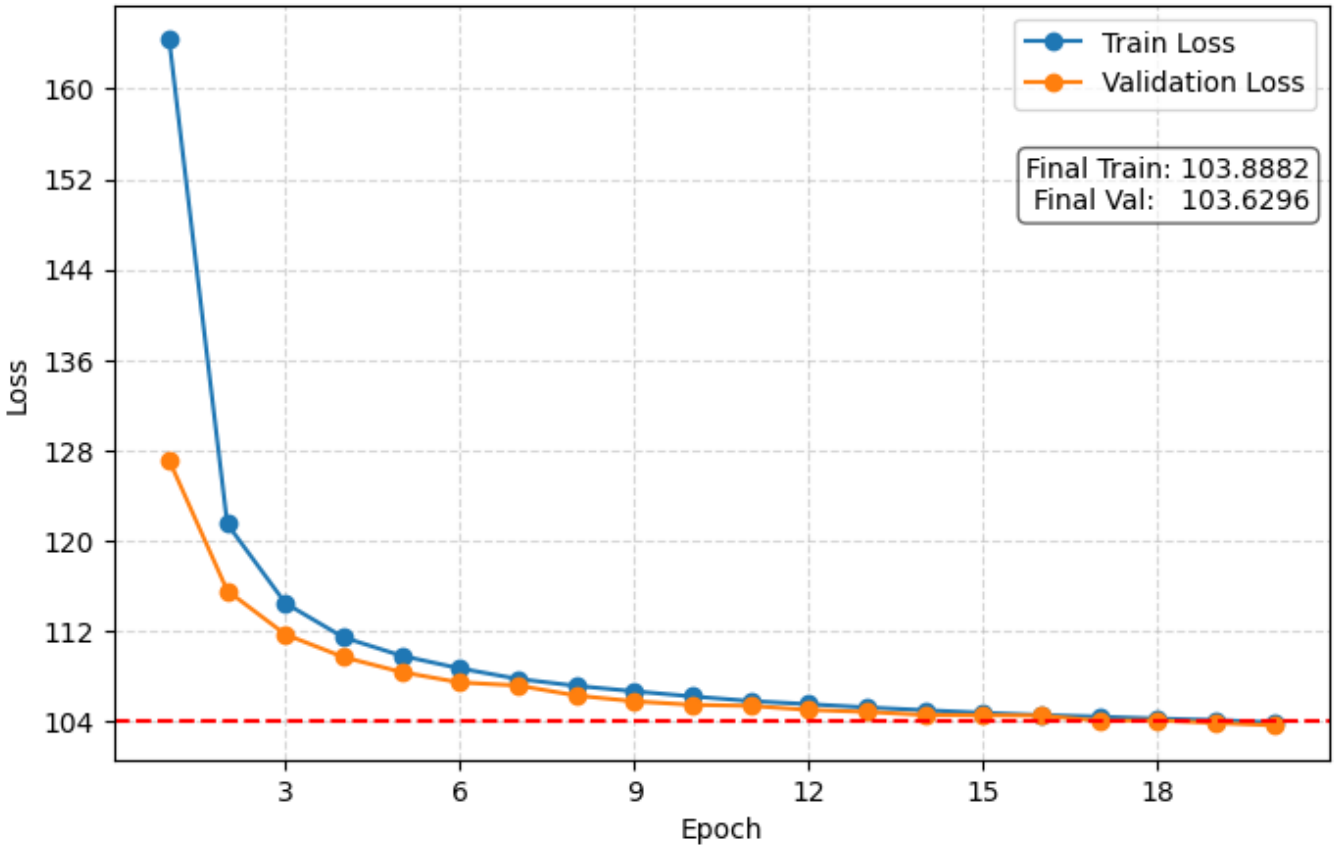

Figure 1: Training and validation loss curves over 20 epochs for the MNIST VAE. The dashed red line marks the target validation loss of 104, and the final loss values are annotated in the top-right corner.

Code for the generation of the figure

```
def plot_loss(history_path: str, save_path: str = None, show: bool = False):
if not os.path.isfile(history_path):
raise FileNotFoundError(f"No file found at {history_path!r}")

with open(history_path, 'r') as fp:
history = json.load(fp)

train_loss = history.get('train_loss', [])
val_loss = history.get('val_loss', [])

num_epochs = len(train_loss)
epochs = range(1, num_epochs + 1)

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(epochs, train_loss, marker='o', label='Train Loss')
ax.plot(epochs, val_loss,   marker='o', label='Validation Loss')

# Horizontal red line at y=104
ax.axhline(y=104, color='red', linestyle='--')

ax.yaxis.set_major_locator(MaxNLocator(integer=True))
ax.set_xticks(range(1, num_epochs + 1)) # Ticks for each epoch
ax.xaxis.set_major_locator(MaxNLocator(integer=True))

# Annote final loss in the top-right corner
final_tr = train_loss[-1]
final_va = val_loss[-1]
text = f"Final Train: {final_tr:.4f}\nFinal Val:   {final_va:.4f}"
ax.text(
0.985, 0.8, text,
transform=ax.transAxes,
fontsize=10,
ha='right', va='top',
bbox=dict(boxstyle='round,pad=0.3', facecolor='white', alpha=0.6)
)

ax.set_xlabel('Epoch')
ax.set_ylabel('Loss')
ax.grid(True, linestyle='--', alpha=0.5)
ax.legend()

if save_path:
plt.savefig(save_path, bbox_inches='tight')
print(f"Saved loss plot to {save_path}")
if show:
plt.show()
plt.close(fig)
```

Question 1.7 : Sample Generation

The trained VAE produces samples that, while somewhat blurred, remain identifiable as MNIST digits and cover a variety of classes, showing no evidence of mode collapse. As illustrated in Figure 2, the digits exhibit thicker or more faded strokes and lack the sharpness of DDPM and CFG outputs. Such blurriness arises from the VAE decoder's implicit averaging under uncertainty, yet the overall coherence and placement of strokes confirm that the model has captured the MNIST manifold and learned to generate plausible handwriting patterns.

Figure 2: An 8×8 grid of 64 images sampled from the trained VAE decoder (epoch 20), demonstrating the model's generative capacity on MNIST.

Despite that, the digits are generally coherent and fall within the manifold of realistic MNIST examples, showing that the VAE has learned to generate plausible handwriting samples (albeit with lower sharpness).

Code for the generation of the figure

```python
def generate_images(
model: VAE,
num_images: int = 64,
device: torch.device = torch.device('cpu'),
show: bool = False,
file_path: Optional[str] = None
) -> torch.Tensor:
model.eval()
model.to(device)

with torch.no_grad():
latent_dim = 20 # Hardcoded for our model
z = torch.randn(num_images, latent_dim, device=device)
x_flat = model.decode(z) # (num_images, 784)
x = x_flat.view(num_images, 1, 28, 28) # (num_images,1,28,28)

if show or file_path:
grid = torchvision.utils.make_grid(x, nrow=8, padding=2)
plt.figure(figsize=(6, 6))
plt.imshow(grid.permute(1, 2, 0).cpu(), cmap='gray')
plt.axis('off')
if file_path:
    plt.savefig(file_path, bbox_inches='tight')
if show:
    plt.show()
plt.close()

return x
```

Question 1.8 : Latent Traversals

To assess disentanglement, we perform two sets of traversals over all 20 latent dimensions: one starting from a random prior sample $z_0 \sim \mathcal{N}(0, I)$, and another from the encoding of an observed MNIST image. For each latent factor $i \in \{0, \ldots, 19\}$, we perturb the base vector by -2ε, $-\varepsilon$, 0, $+\varepsilon$, and $+2\varepsilon$ (with $\varepsilon = 2.0$) and decode the results. Each row in Figures 3–4 corresponds to one latent dimension; columns show increasing offsets.

The results do not reveal strongly disentangled factors in the VAE's learned representation. Varying each latent dimension (while fixing others) causes changes in the generated digit images, but these changes are entangled. For example, adjusting a particular latent dimension might simultaneously alter a digit's stroke thickness and its shape or orientation, rather than isolating one property alone. Nevertheless, a few factors show relatively isolated effects:

- **Factor 15:** Consistently shifts decoded outputs between "0"-like shapes when negatively perturbed and "1"-like shapes when positively perturbed.
- **Factor 13:** Primarily modulates stroke width across digits.
- **Factor 17:** When increased, tends to morph various digits toward the appearance of a "4."

Overall, this indicates that even if some factor seem to have obtained some disentenglement, the latent space of the VAE is largely entangled: the generative factors (such as style, stroke width, or positioning) are mostly distributed across multiple latent dimensions rather than being neatly separated one per dimension.

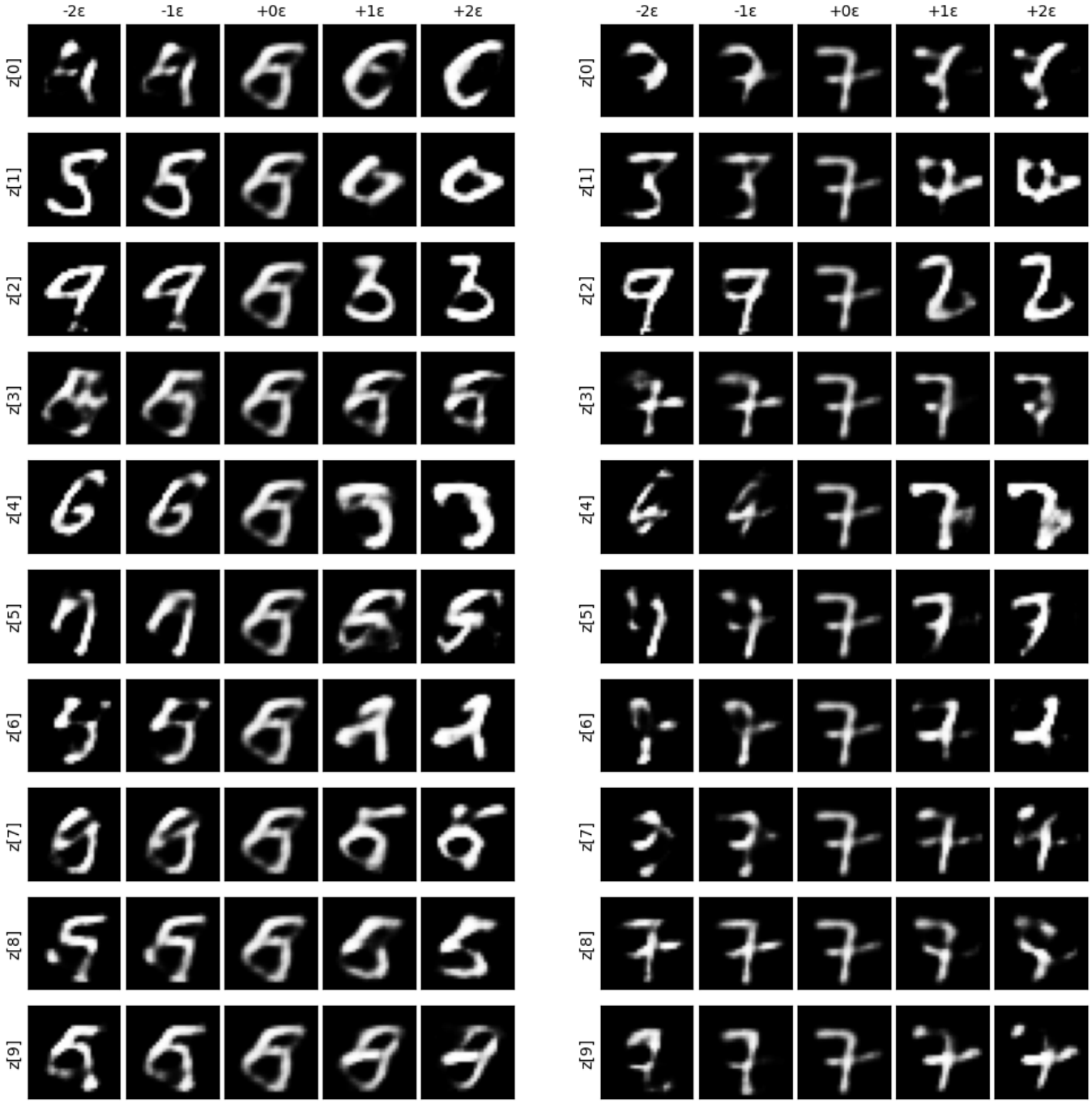

Figure 3: Latent dimensions 0–9. **Left:** traversal from a Gaussian prior sample. **Right:** traversal from an encoded MNIST image. Each row i shows perturbations of z_i by $\{-2, -1, 0, +1, +2\}\varepsilon$.

Figure 4: Latent dimensions 10–19. **Left:** traversal from a Gaussian prior sample. **Right:** traversal from an encoded MNIST image. Each row i shows perturbations of z_i by $\{-2, -1, 0, +1, +2\}\varepsilon$.

Code for the latent traversals

```python
def generate_latent_traversals(
model: VAE,
n_steps: int = 5,
eps: float = 2.0,
device: torch.device = torch.device('cpu'),
data_sample: Optional[torch.Tensor] = None,
) -> torch.Tensor:
model.eval()
model.to(device)

latent_dim = 20 # Hardcoded for our model
if data_sample is not None: # Sampled from X (MNIST)
x_flat = data_sample.to(device).view(1, -1)
mu, logvar = model.encode(x_flat)
z0 = model.reparameterize(mu, logvar)
else: # Sampled from the prior Z = N(0,I)
z0 = torch.randn(1, latent_dim, device=device)

traversals = []
half_span = (n_steps - 1) / 2

with torch.no_grad():
for dim in range(latent_dim):
    row = []
    for step in range(n_steps):
        z = z0.clone()
        # Perturb the dim-th latent variable
        z[0, dim] += (step - half_span) * eps
        x = model.decode(z) # (1, P), P = H*W
        side = int(math.sqrt(x.size(1)))
        row.append(x.view(1, 1, side, side)) # (1, 1, H, W)
    # Stack steps → (n_steps, 1, H, W)
    traversals.append(torch.cat(row, dim=0))

# Stack dims → (n_factors, n_steps, 1, H, W)
return torch.stack(traversals, dim=0).cpu()
```

Code for the generation of the figure

```python
def plot_traversals(
traversals: torch.Tensor,
save_path: Optional[str] = None,
show: bool = True,
):
n_factors, n_steps, _, H, W = traversals.shape
```

```
center = n_steps // 2
mid_x = (W - 1) / 2
mid_y = (H - 1) / 2

# Iterate in blocks of 10 latent dims
for block_start in range(0, n_factors, 10):
block_end = min(block_start + 10, n_factors)
block_size = block_end - block_start

fig, axes = plt.subplots(
    block_size, n_steps,
    figsize=(4.8, 10),
    gridspec_kw={'wspace': 0},
    squeeze=False
)

for row, factor in enumerate(range(block_start, block_end)):
    for step in range(n_steps):
        ax = axes[row, step]
        ax.imshow(traversals[factor, step, 0], cmap='gray', vmin=0, vmax=1)

        # Hide ticks and spines
        ax.set_xticks([])
        ax.set_yticks([])

        # Top row x-tick labels
        if row == 0:
            ax.xaxis.set_ticks([mid_x])
            ax.xaxis.set_ticklabels([f'{step - center:+d}'], fontsize=10)
            ax.xaxis.tick_top()
            ax.tick_params(axis='x', length=0)

        # First collumn y-tick labels
        if step == 0:
            ax.yaxis.set_ticks([mid_y])
            ax.yaxis.set_ticklabels([f'z[{factor}]'], fontsize=10, rotation=90, va='center')
            ax.tick_params(axis='y', length=0)

plt.tight_layout()
if save_path:
    # Append block index to filename before extension
    base, ext = os.path.splitext(save_path)
    part_path = f"{base}_part{block_start//10}{ext}"
    os.makedirs(os.path.dirname(part_path), exist_ok=True)
    fig.savefig(part_path, bbox_inches='tight')

if show:
    plt.show()

plt.close(fig)
```

Question 1.9 : Interpolation Comparison

We compare two interpolation schemes between points z_0, z_1 in the VAE's latent space and their corresponding data-space images x_0, x_1. Figure 5 (top row) shows decoded images for $\alpha \in \{0, 0.1, \ldots, 1\}$ via latent-space interpolation $z'_\alpha = \alpha z_0 + (1-\alpha) z_1$. Each intermediate decode is a plausible MNIST digit that transitions smoothly from one class to the next. In contrast, Figure 5 (bottom row) displays pixel-space interpolation $x'_\alpha = \alpha x_0 + (1-\alpha) x_1$, which yields ghosted or blurry superpositions that only resemble valid digits at the endpoints.

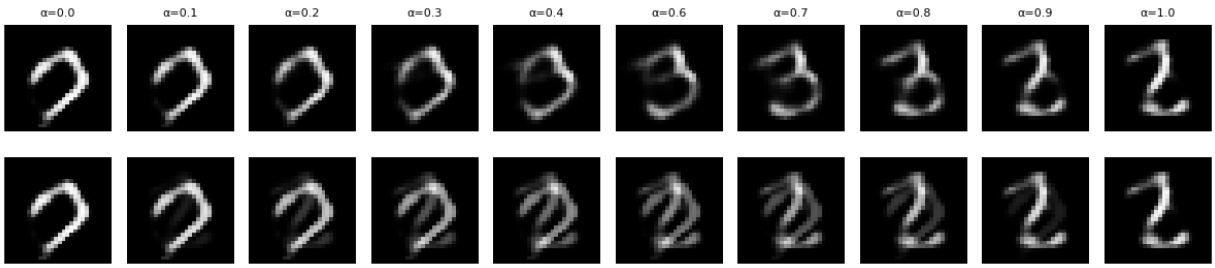

Figure 5: Comparison of interpolation methods between two random latent samples. **Top row:** latent-space interpolation yields coherent digit morphs. **Bottom row:** pixel-space interpolation produces incoherent blends except at $\alpha = 0, 1$.

Thus, the VAE's latent-space interpolation produces far more coherent and meaningful intermediate outputs, highlighting that the latent space has learned a high-level representation of digit features, whereas pixel-space mixing does not respect the data's true structure.

To further illustrate, we perform latent-space interpolations between real MNIST images. Figures 6–8 show smooth transitions learned in latent space, with digit-specific structural changes that respect the data manifold.

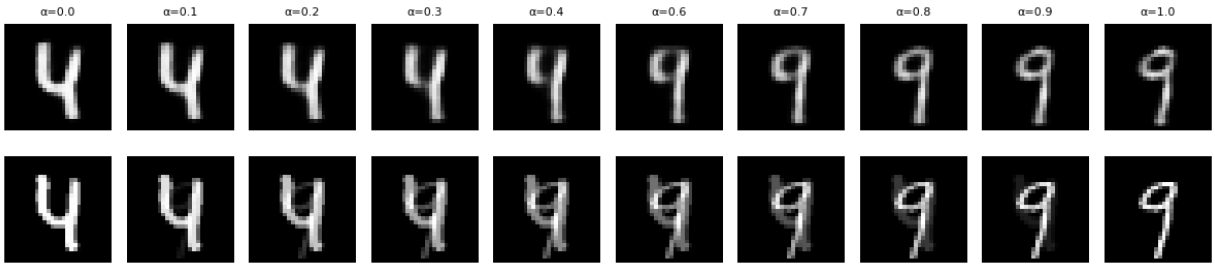

Figure 6: Latent-space interpolation between a real "4" and a real "9." The upper loop of the "4" gradually closes to form the "9."

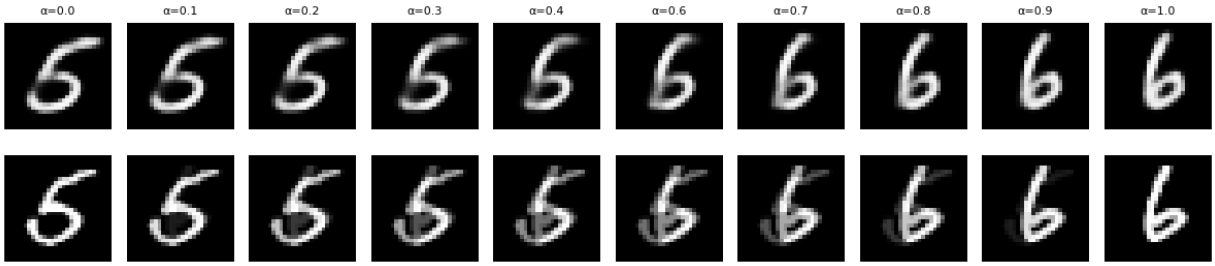

Figure 7: Latent-space interpolation between a real "5" and a real "6." The "5" loop smoothly closes into the "6."

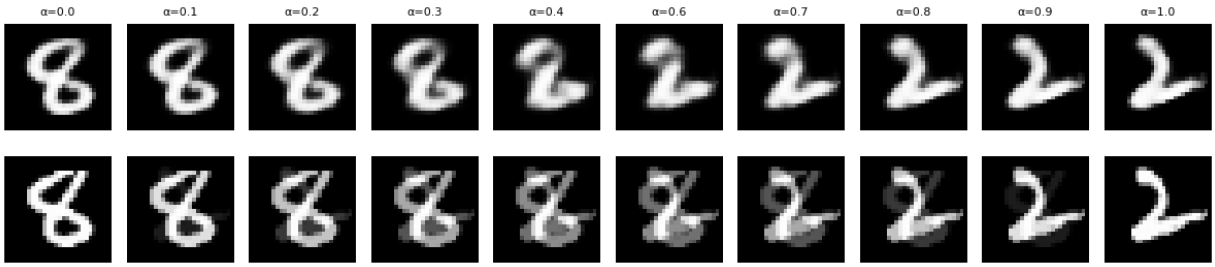

Figure 8: Latent-space interpolation between a real "8" and a real "2." The two loops of the "8" collapse sequentially to form the "2," though with slightly rougher transitions.

Code for the generation of the figure

```python
def plot_interpolation(
model: VAE,
endpoints: Optional[Tuple[torch.Tensor, torch.Tensor]] = None,
num_steps: int = 11,
device: torch.device = torch.device('cpu'),
save_path: Optional[str] = None,
show: bool = True
):
model.eval()
model.to(device)

latent_dim = 20 # Hardcoded for our model

if endpoints is None: # Sample from prior Z = N(0,I)
z0 = torch.randn(1, latent_dim, device=device)
z1 = torch.randn(1, latent_dim, device=device)
with torch.no_grad():
    x0 = model.decode(z0).view(1,1,28,28).cpu()
    x1 = model.decode(z1).view(1,1,28,28).cpu()
else: # Sample from data X (MNIST)
x0, x1 = endpoints
x0 = x0.to(device); x1 = x1.to(device)
flat0, flat1 = x0.view(1,-1), x1.view(1,-1)
with torch.no_grad(): # Encode to latent space
    mu0, logvar0 = model.encode(flat0)
    mu1, logvar1 = model.encode(flat1)
    z0 = model.reparameterize(mu0, logvar0)
    z1 = model.reparameterize(mu1, logvar1)
# Keep CPU copies for pixel interpolation
x0, x1 = x0.cpu(), x1.cpu()

# Interpolation weights
alphas = torch.linspace(0, 1, steps=num_steps, device=device)

# Latent-Space interpolation
latent_interpolation = []
for a in alphas:
z = a*z0 + (1-a)*z1
with torch.no_grad():
    img = model.decode(z).view(1,1,28,28).cpu()
latent_interpolation.append(img)
latent_interpolation = torch.cat(latent_interpolation, dim=0)

# Pixel-Space interpolation
pixel_interpolation = []
for a in alphas:
img = a*x0 + (1-a)*x1
```

```
pixel_interpolation.append(img)
pixel_interpolation = torch.cat(pixel_interpolation, dim=0)

fig, axes = plt.subplots(2, num_steps, figsize=(12, 3), squeeze=False)
for i in range(num_steps):
axes[0,i].imshow(latent_interpolation[i,0], cmap='gray', vmin=0, vmax=1)
axes[0,i].axis('off')
axes[1,i].imshow(pixel_interpolation[i,0], cmap='gray', vmin=0, vmax=1)
axes[1,i].axis('off')
axes[0,i].set_title(f"={alphas[i]:.1f}", fontsize=8)

plt.tight_layout()
if save_path:
plt.savefig(save_path, bbox_inches='tight')
if show:
plt.show()
plt.close(fig)
```

Question 2 : Unconditional Diffusion Model (DDPM)

Question 2.1 : Sample Generation (Across Epochs)

During training, the quality of the DDPM's generated digits improves dramatically. Early checkpoints produce samples that are noisy and often unrecognizable, but by epoch 20 the model yields well-formed MNIST digits—with only minor blurring or small artifacts remaining. As shown in Figure 9, the progressive enhancement of stroke clarity and digit structure is apparent across epochs 5, 9, 15 and 19.

A key distinction from the VAE samples is the nature of the artifacts: rather than overall blur, the diffusion outputs exhibit "shaky" lines or missing strokes. This suggests that the DDPM's latent denoising process captures sharper, more detailed features, even if fine-grained stability (straight strokes, closed loops) can still be improved.

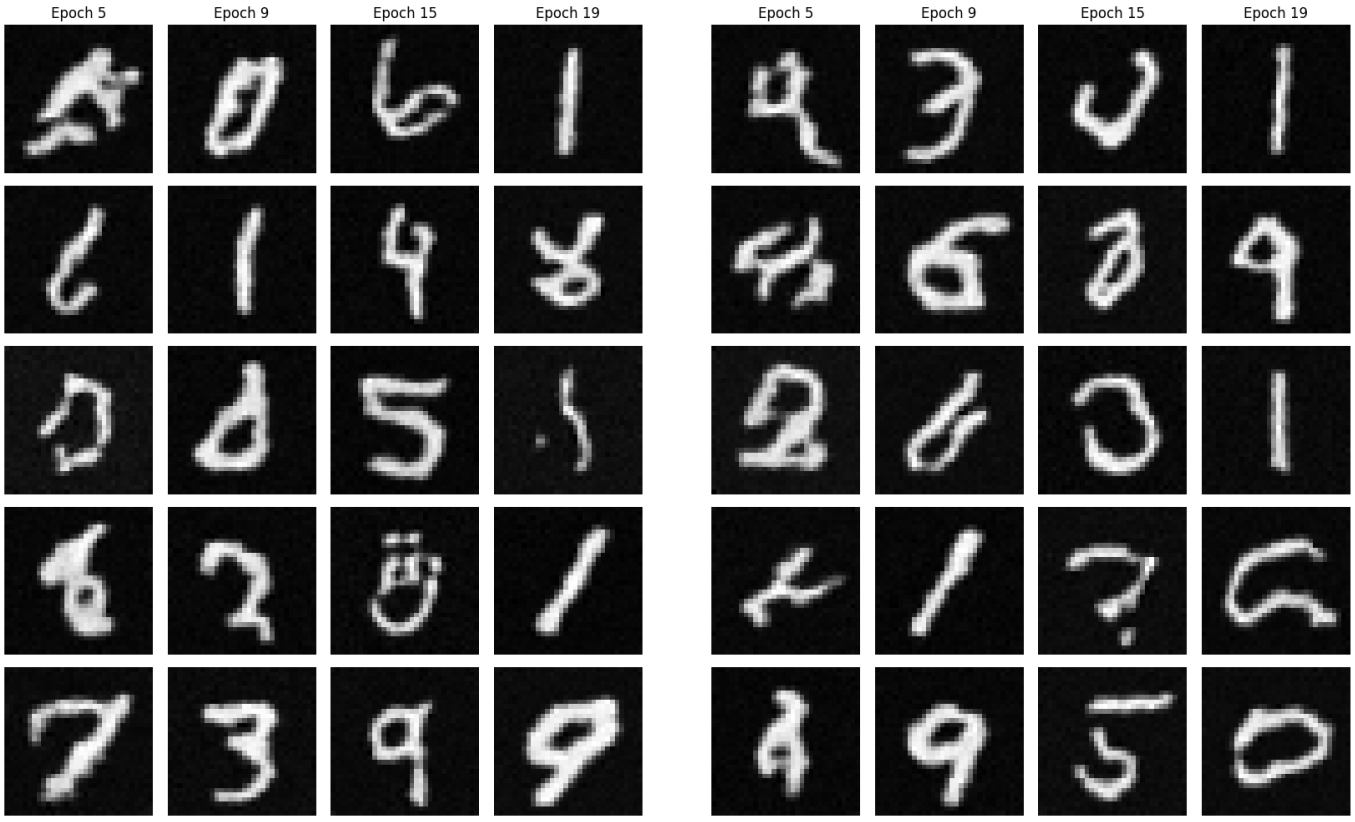

Figure 9: Unconditional DDPM samples at epochs 5, 9, 15, and 19 (columns) for four independent draws (rows). Early epochs produce noisy structures, while later epochs yield clear digit shapes.

Figure 10 presents a larger grid of 64 samples from the final epoch, illustrating the overall diversity and typical fidelity achieved after 20 epochs.

Figure 10: Grid of 64 MNIST images generated by the DDPM after 20 epochs, illustrating the model's typical sample quality and diversity post-training.

To further enhance sample quality, one could:

- **Extended training and/or sampling.** Train for more epochs and/or use additional reverse diffusion steps at inference for sharper outputs.
- **Fix the reverse-process variance to an isotropic constant.** Choosing $\Sigma_t = \beta_t I$ rather than learning a full diagonal covariance could stabilize training and improve quality.
- **Experiment with the forward noise schedule.** Tune the forward diffusion $\{\beta_t\}$ (e.g. linear or cosine schedules) for balanced denoising.
- **Enhance U-Net architecture** Incorporate group normalization and self-attention to capture global structure and reduce artifacts.

Each of these adjustments is expected to reduce residual artifacts and improve the faithfulness of generated digits.

Question 2.2 : Intermediate Reverse-Process Samples

During sampling, the DDPM's reverse diffusion gradually transforms noise into coherent digits. At the highest timesteps ($t = 1000$), each output is essentially pure noise; only around the midpoint (e.g. $t = 500$) do faint contours—perhaps the rough shape of a "3" or "7"—begin to appear. It was surprising to observe that recognizable outlines emerge so late in the denoising chain. By $t \leq 100$, most speckles have been removed, yielding solid strokes and well-defined curves (closed loops of an "8", continuous line of a "1"). Finally, at $t = 0$, the model produces a clean, fully formed digit indistinguishable from a real MNIST sample.

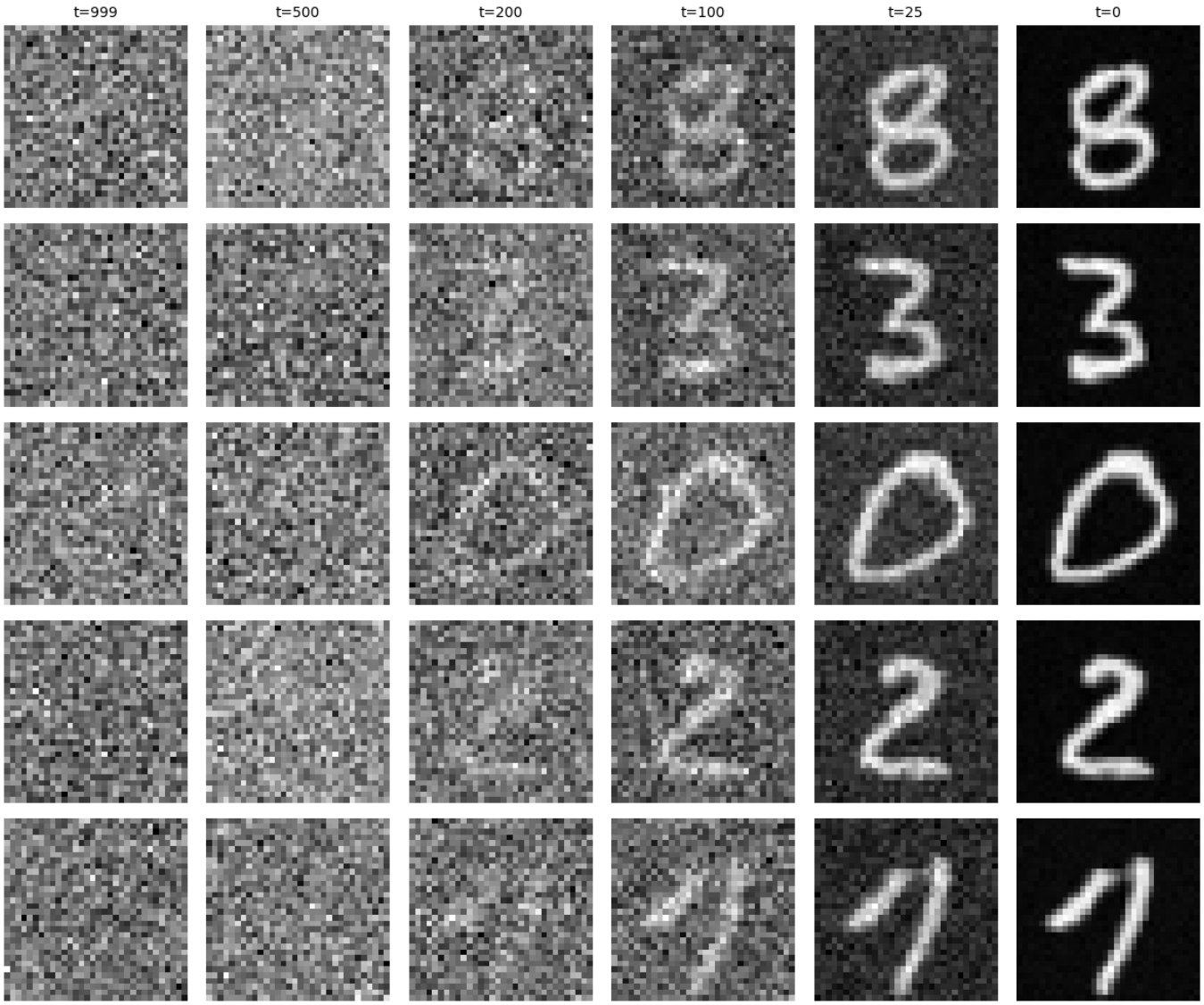

Figure 11: Intermediate reverse diffusion samples for four chains. Columns correspond to timesteps $t = \{999, 500, 200, 100, 25, 0\}$, and rows to different sample chains. Coherent digit shapes emerge gradually as the model removes noise.

This step-by-step refinement confirms the DDPM's expected behavior: high-level digit structures form first, then finer details are filled in, converging to a realistic handwritten digit by the final timestep.

Question 3 : Conditional Diffusion Model (CFG)

Question 3.1: Classifier-Free and Guidance

The term "classifier-free" refers to the fact that this guidance method does not require any separately trained classifier. Instead, we jointly train the diffusion model to predict both a conditional score $\varepsilon_\theta(z_t, c)$ and an unconditional score $\varepsilon_\theta(z_t)$ by sometimes replacing the true label c with a null token during training. At sampling time, these two estimates are linearly combined as

$$\tilde{\varepsilon}_\theta(z_t, c) = (1 + w)\, \varepsilon_\theta(z_t, c) \; - \; w\, \varepsilon_\theta(z_t)$$

where w controls the strength of guidance. This yields the same trade-off between sample fidelity and diversity as traditional classifier gradients—but without ever training or invoking a classifier—hence "classifier-free guidance".

Question 3.2: Alternative to Classifier-Free Guidance

The paper points out an alternative that avoids learning an unconditional branch altogether: if the class prior $p(c)$ is known and the number of classes is small, one can recover the marginal (unconditional) score by

$$\nabla_{z_t} \log p(z_t) = \nabla_{z_t} \log\Big(\sum_c p(z_t \mid c)\, p(c)\Big) = \sum_c p(c \mid z_t)\, \nabla_{z_t} \log p(z_t \mid c)$$

thus obtaining $\nabla_{z_t} \log p(z_t)$ from the conditional scores alone. Under this scheme, the training loss reduces to the original DDPM denoising objective applied only to the conditional model:

$$L(\theta) = \mathbb{E}\big\|\varepsilon_\theta(z_t, c) - \epsilon\big\|_2^2, \quad z_t = \alpha_t x + \sigma_t \epsilon.$$

In contrast, classifier-free guidance uses a joint loss that alternates between conditional and unconditional updates, explicitly learning both $\varepsilon_\theta(z_t, c)$ and $\varepsilon_\theta(z_t)$ during training.

Question 3.3 : Sample Generation (Across Epochs)

Introducing classifier-free guidance significantly sharpens and class-faithful digit generation as training progresses. Even in early epochs, the guided model produces noticeably clearer outputs than the VAE or unguided DDPM: although some blurring remains, each sample already bears a strong resemblance to its target class. By epochs 15–20, the digits become crisp and unambiguous, with strokes and loops well defined.

The benefits of conditioning and guidance are evident: the images are less noisy as we can see in Figure **??**. looks more consistently like the intended digits showing improved fidelity to the desired digit class compared to an unguided model.

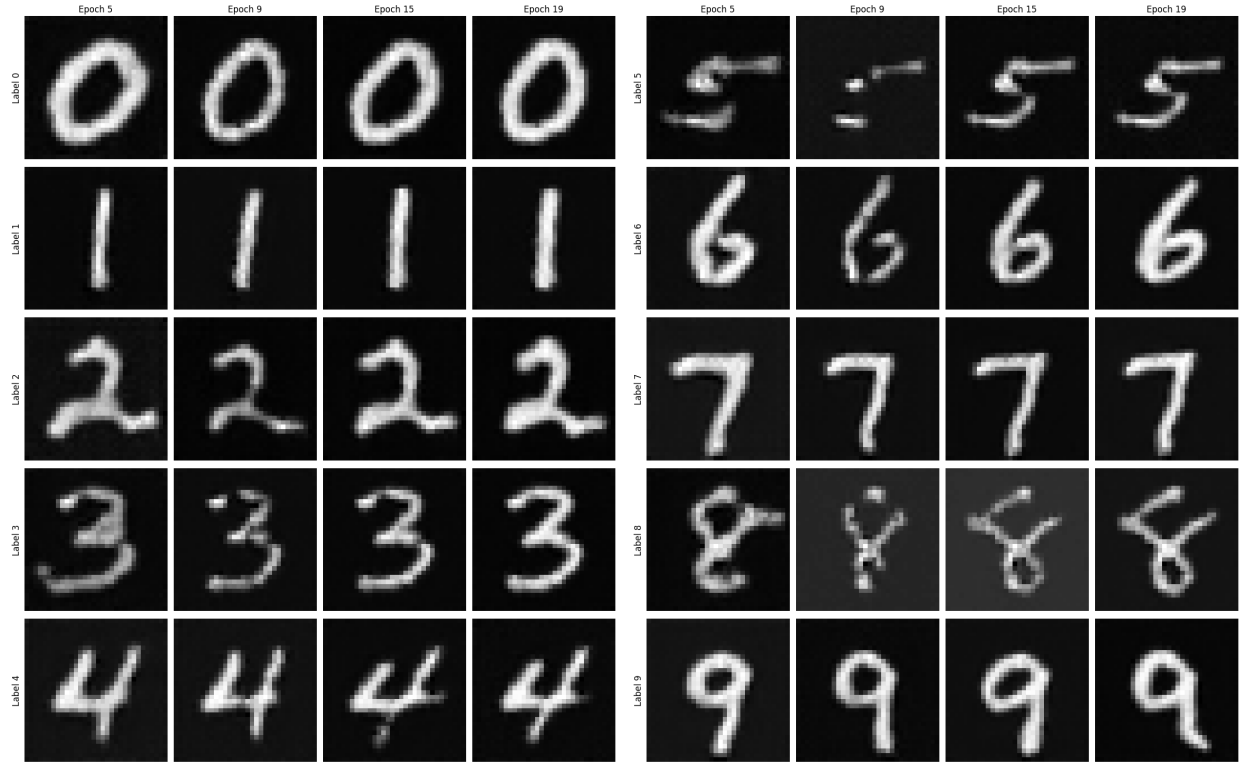

Figure 12: CFG diffusion model samples conditioned on labels 0–9 across epochs 5, 9, 15, and 19 (left panel: labels 0–4; right panel: labels 5–9). Within each panel, columns denote epochs and rows fixed digit classes, illustrating how classifier-free guidance sharpens and refines digit fidelity over training.

In summary, conditioning steers the generation process to produce well-defined digits of the requested class, and this effect becomes stronger over training: the longer the model trains, the more it can translate label information into sharper, more reliable samples for each digit category.