Report – Diffusion models assignment

Part A — Unconditional DDPM on MNIST

I implemented a lightweight DDPM (denoising diffusion probabilistic model) on MNIST. The goal was to learn how a model can reverse a gradual noising process and generate digits from pure noise. I trained a small UNet-style denoiser, saved training loss curves and sample grids, and ran a simple hyperparameter experiment to compare different numbers of diffusion steps.

- ***Forward process (noising):*** We slowly add small amounts of Gaussian noise to a clean image over many steps until it becomes just noise.
- ***Reverse process (denoising):*** A neural network learns to reverse those steps — at each step it removes a bit of noise until we recover an image.
- ***Training objective (simplified):*** Instead of teaching the model to directly predict the cleaned image, I trained it to predict the exact noise that was added. The loss is just mean-squared error between the true and predicted noise — simple and effective.
- ***Role of the denoiser:*** The UNet-style denoiser looks at the noisy image and the current timestep and predicts the noise pattern to remove. Its encoder–decoder structure helps capture both global digit shape and fine strokes...

Implementation:-

Framework: PyTorch, written as a Colab notebook.

Model: Small UNet-like network with sinusoidal timestep embeddings. Kept compact so experiments run quickly on Colab GPU.

Diffusion schedule: Linear beta schedule (easy and works well on MNIST).

Files / notebook cells:

- Utilities: save/display helpers.
- ***BaseUNet*** (denoiser) + timestep embedding.
- ***SimpleSchedule*** for q_sample and sampling loop.
- ***train_ddpm*** training loop that saves checkpoints, ***final_samples.png***, and ***loss_curve.png***.

Design choices: Predicting noise (ε) is stable; normalizing images to [-1, 1] keeps training stable; periodic saved sample grids let me visually track progress.

Experiment:-

I ran a focused experiment to see how the number of diffusion timesteps affects sample quality. Other hyperparameters were kept the same.

- **Settings compared:**
    - T = 50 (few steps)
    - T = 200 (default / balanced)
    - T = 1000 (many steps — run on GPU if time permits)
- Fixed training settings: batch size 128, lr = 2e-4, compact UNet, same number of training epochs (shorter for quick comparisons).

Training configuration (baseline)

- Dataset: MNIST (28×28 grayscale), normalized to `[-1, 1]`
- Model: small UNet (`base_channels=48`)
- Optimizer: Adam, `lr = 2e-4`
- Batch size: 128
- Epochs: 6 (baseline)
- Diffusion timesteps: baseline `T = 200` (I ran experiments with `T = 50` and `T = 1000`)
- Saved outputs: `results/uncond/final_samples.png` and `results/uncond/loss_curve.png`

Observations:-

- **T = 50:** very fast but samples are often blurry and digit shapes are less consistent. Good for quick experiments but low quality.
- **T = 200:** good balance — digits are recognizable and reasonably sharp. Best trade-off for our small model and limited compute.
- **T = 1000:** when trained long enough, outputs can be sharper and more detailed, but this requires much more compute/time. Improvements beyond 200 were noticeable but not massive for this small model.
- **Learning rate note:** lowering lr (e.g., 1e-4) slows learning but can be more stable; raising lr (4e-4) can speed early progress but sometimes destabilizes training.

Part B — Conditional DDPM on MNIST

In Part B, I extended my diffusion model to generate digits in a conditional manner. Unlike Part A where the model freely generated any digit, here I trained the model so it can follow a given class label (0–9). The objective was to see how adding label information changes the training process and whether it gives us control over what digit gets generated.

Idea:-

- In the unconditional setup, the denoiser only sees a noisy image and the timestep. It has to figure out "which digit" to form purely from patterns in the dataset.
- In the conditional setup, I gave the model **an extra hint**: the class label (e.g., "3"). This label is embedded into a vector and combined with the timestep embedding. The model now learns not just "how noisy the input is," but also "which digit it should clean towards."
- Training still uses the same noise prediction objective as Part A. The difference is that now the denoiser is guided by the label at every step.

Implementation summary:-

- **Model:** I built `ConditionalUNet`, a compact UNet-based denoiser similar to Part A, but with a label embedding layer (`nn.Embedding(10, emb_dim)`). The label embedding is added to the timestep embedding before being injected into the UNet.

- **Dataset:** MNIST, normalized to the range [-1, 1].

- **Training details:**

  - Optimizer: Adam, learning rate = 2e-4
  - Batch size = 128
  - Epochs = 6
  - Diffusion steps (T): 200
  - Beta schedule: linear (from 1e-4 to 0.02)

- **Outputs:** At the end of training, I saved conditional sample grids for three classes (0, 3, and 7), the training loss curve, and intermediate checkpoints.

Results:-

Sample grids

Below are examples of digits generated by the conditional model:

- **Class 0:** `<insert results/cond/samples_class_0.png>`
- **Class 3:** `<insert results/cond/samples_class_3.png>`
- **Class 7:** `<insert results/cond/samples_class_7.png>`

(Insert your actual saved images here. Add a short caption under each: e.g., "Digits generated by the conditional model when conditioned on class 3.")

4.2 Training loss

The training loss steadily decreased over epochs, showing that the model successfully learned to denoise under label guidance.

- Training loss curve: `<insert results/cond/loss_curve.png>`

4.3 Observations

- **Alignment with labels:** The conditional samples generally matched the intended digit class. For example, grids conditioned on "3" consistently produced recognizable 3s.
- **Quality vs unconditional:** The sharpness of digits was similar to the unconditional model, but now I had much better control over which digit type was generated.
- **Failure cases:** Occasionally, digits looked distorted or slightly ambiguous (e.g., some "7" samples). This could likely be improved with longer training or a slightly larger model.

Comparison with Part A:-

- In Part A (unconditional), the model produced diverse digits, but I had no control — any digit could appear in a sample grid.
- In Part B (conditional), the label embedding gave explicit control. I could request a specific digit and the model mostly delivered that digit.
- The visual fidelity was roughly the same as unconditional under the same compute budget, but the **usefulness** of the model improved a lot because it allowed controlled generation.

Conclusion:-

The conditional DDPM successfully learned to generate digits based on class labels. The conditioning method (simple label embeddings added to timestep embeddings) was lightweight and worked well on MNIST. The results demonstrate how conditioning transforms a basic generative model into a controllable one, without changing the overall training procedure.

If more training time or capacity were available, I expect the conditional model to produce even sharper and more consistent digits across all classes. This part of the assignment helped me understand how conditioning works in diffusion models and how small architectural changes can give much more control over generation.