Programming Assignment 2: Advanced Machine Learning

Advanced Machine Unlearning
Tejas Sharma: 22B0909
Muskaan Jain: 22B1058
Ojas Maheshwari: 22B0965

Contents

1 The Architecture (Denoising Diffusion Probabilistic Model)

1.1 The class `DDPM`: Neural Network $\vec{\epsilon}_{\vec{\theta}}(\vec{x}, t)$

Our model encodes the time t (an integer) into a 16-dimensional vector via *sinusoidal embedding*, which is then passed through a linear layer to form a $t_{dim} = 4$-dimensional vector $\vec{t}_{embed}$.

The Denoising Diffusion Probabilistic Model (DDPM) consists of two main components: a time embedding module and a noise prediction model. The main model takes input a concatenated vector $(\vec{x}, \vec{t}_{embed})$ (input, x_t and time t_{embed}) and passes it through a neural network with 2 hidden layers: 2 ReLU (rectified linear layer) units, sandwiched between 3 linear layers in the neural network.

All intermediate vectors (between layers) have dimension $i_{dim} = 4n_{dim} + 2t_{dim}$, and the final vector has dimension $n_{dim} = dim(\vec{x})$. Both the input $\vec{x}$ and the output $\vec{\epsilon}_{\vec{\theta}}(\vec{x}, t)$ are 2D tensors in the code, with the first dimension being `batch_size`, for parallelization of training the parameters $\vec{\theta}$ and for sampling large number of samples.

1.1.1 Time Embedding

The time embedding module maps a time step $t \in \{1, \dots, N\}$ into an embedding space:

$$\text{Time Embedding:} \quad t \mapsto \mathbf{e}_t$$
$$\mathbf{e}_t = \sigma(W_t \cdot \text{concat}(\sin(\omega t), \cos(\omega t)))$$

where:

- ω is a set of predefined frequencies.
- $W_t \in \mathbb{R}^{2d_{\text{sin}} \times d_{\text{embed}}}$ is a learnable weight matrix.
- d_{sin} is the time dimension.
- d_{embed} is the time embedding dimension.
- σ is a nonlinear activation function (ReLU).

1.1.2 Noise Prediction Model

Given input data $\mathbf{x} \in \mathbb{R}^{d_x}$ and a time embedding $\mathbf{e}_t \in \mathbb{R}^{d_t}$, the model predicts the noise component:

$$\begin{aligned} \mathbf{h}_1 &= \sigma(W_1[\mathbf{x}, \mathbf{e}_t]) && \text{where } W_1 \in \mathbb{R}^{(d_x+d_t)\times d_h} \\ \mathbf{h}_2 &= \sigma(W_2\mathbf{h}_1) && \text{where } W_2 \in \mathbb{R}^{d_h \times d_h} \\ \hat{\boldsymbol{\epsilon}} &= W_3\mathbf{h}_2 && \text{where } W_3 \in \mathbb{R}^{d_h \times d_x} \end{aligned}$$

where:

- $\mathbf{x}$ is the noisy input data.
- $\mathbf{e}_t$ is the time embedding.
- W_1, W_2, W_3 are learnable weight matrices.
- d_h is the hidden dimension of the model.
- σ is an activation function (ReLU).
- $\hat{\boldsymbol{\epsilon}}$ is the predicted noise component.

1.2 Experimental Analysis

1.2.1 Effect of Number of Diffusion Steps (T)

We train five DDPM models with different values of T: {10, 50, 100, 150, 200}, keeping all other hyperparameters fixed. The expected behavior is as follows:

- For small T, the denoising process may be insufficient, leading to lower-quality samples.
- As T increases, sample quality should improve, but the computational cost also increases.

- There might be diminishing returns beyond a certain T, where further increasing T does not significantly enhance performance.
- The optimal choice of T will depend on balancing computational efficiency and generation quality.

1.2.2 Effect of Noise Schedule

We investigate the effect of the noise schedule by varying its hyperparameters and exploring alternative schedules:

- The linear noise schedule uses two parameters, β_{low} and β_{high}, which determine the range of noise added at each step.
- We test at least five different values of β_{low} and β_{high} to determine their impact on model performance. We realised that for datasets excluding the `albatross`, $\beta_{\text{low}} = 0.001$, $\beta_{\text{high}} = 0.02$ was optimal whereas for `albatross`; $\beta_{\text{low}} = 0.002$, $\beta_{\text{high}} = 0.02$.
- Additionally, we explore alternative noise schedules such as:
  - **Cosine schedule:** Adjusts noise variance in a way that better preserves signal structure.
  - **Sigmoid schedule:** Introduces a non-linearity to control noise variance smoothly.

The expected behavior is:

- The linear schedule provides a straightforward noise increase, but it may not be optimal for all datasets.
- The cosine and sigmoid schedules may lead to better sample quality by preserving structure in early timesteps and gradually reducing noise more effectively.
- There may exist an optimal set of β values that balance learning stability and generation quality.

The below plots are for the following hyperparameter settings: $\beta_{\text{low}} = 0.001$, $\beta_{\text{high}} = 0.02$, `Learning_Rate` $= 0.01$, `Batch_Size` $= 100$:

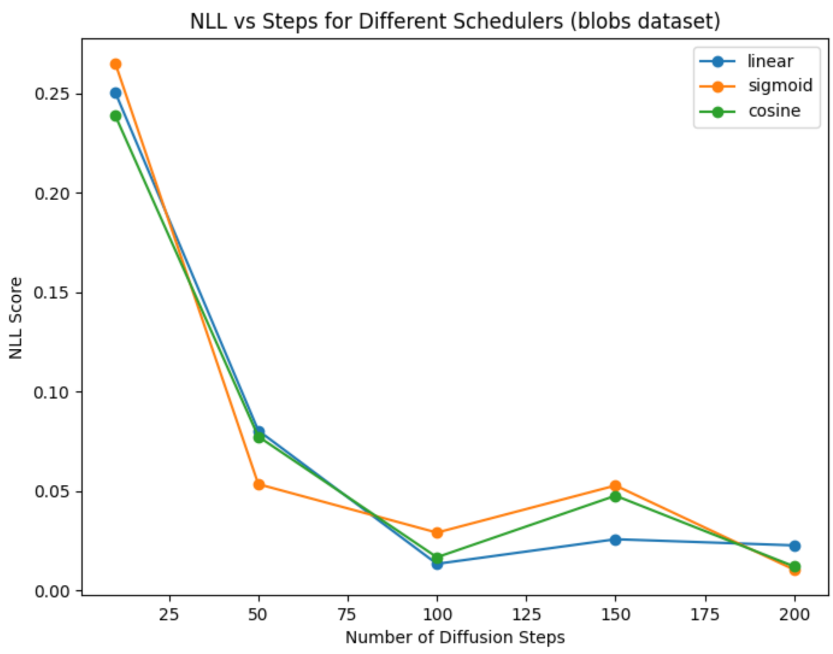

Figure 1: NLL vs Steps for Blobs Dataset

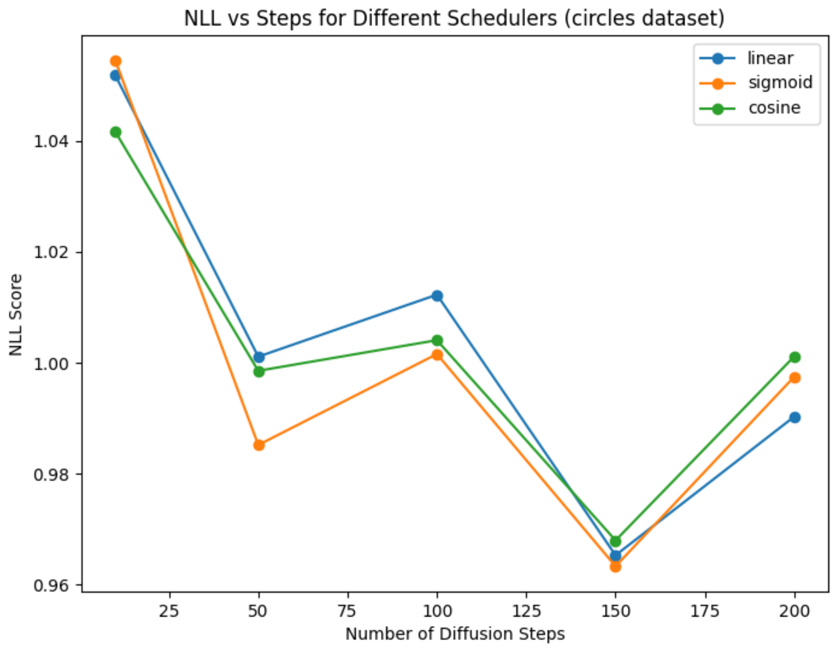

Figure 2: NLL vs Steps for Circles Dataset

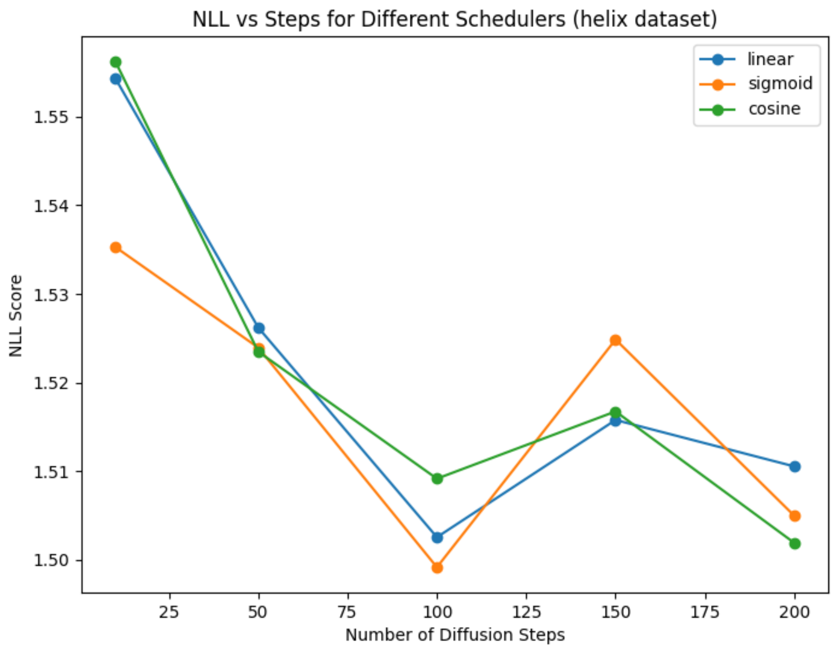

Figure 3: NLL vs Steps for Helix Dataset

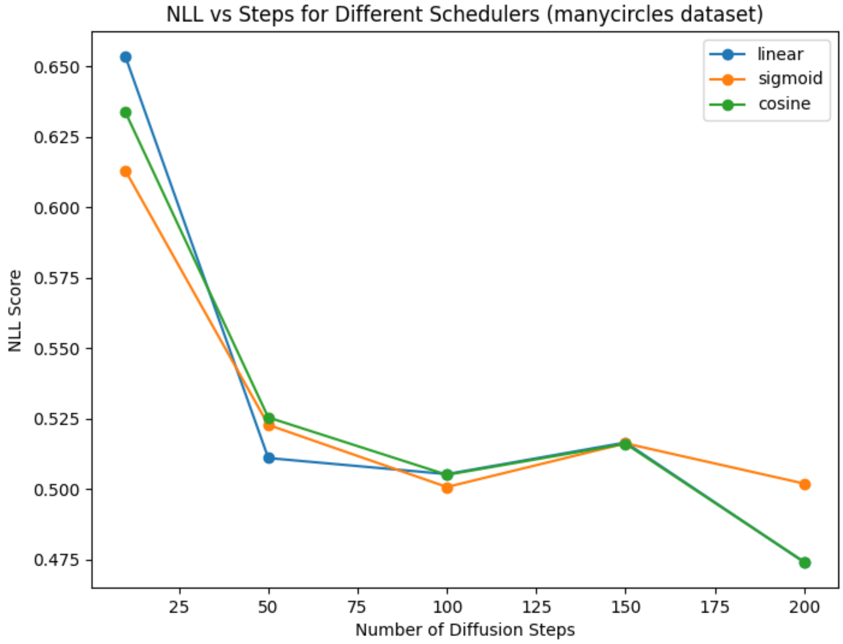

Figure 4: NLL vs Steps for Manycircles Dataset

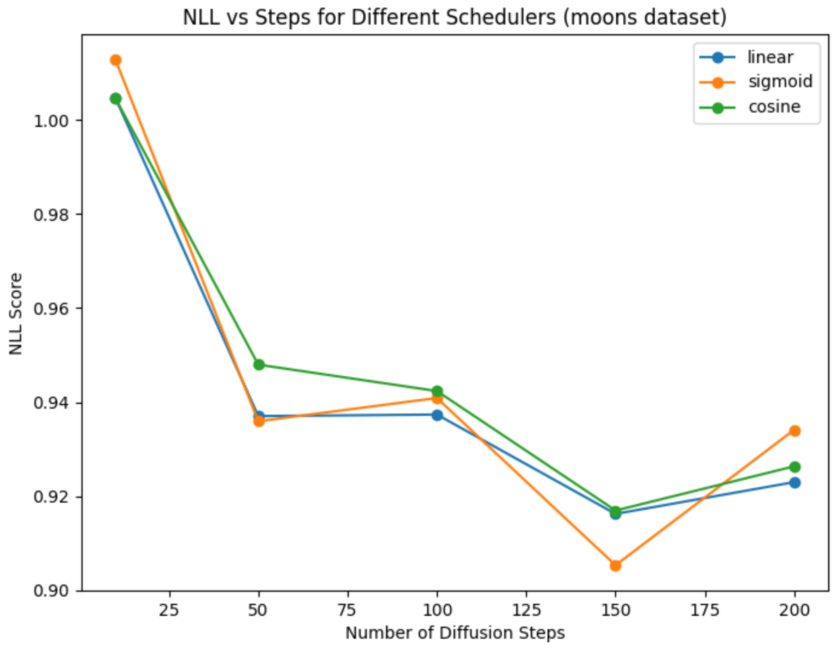

Figure 5: NLL vs Steps for Moons Dataset

Summarising the plots, we observe that it is hard to definitively conclude which scheduler works better overall (general rule for all datasets). The `Linear` Scheduler seems to be the best for the Blobs dataset, whereas `Sigmoid` is best for the Circles and Moons Dataset. The performance of the `Cosine` scheduler is intermediate between the Linear and Sigmoid Schedulers.

We have used only `NLL` as a metric to justify the behaviour of the scheduler. Even with sub-sampling, the computation of `EMD` was taking too much time to run, so we could not include it.

For the `Albatross` Dataset, the following were the hyperparameters that worked best:

- beta range = 0.001 to 0.02
- scheduler = linear
- time steps = 100
- learning rate = 0.01
- batch size = 100
- epochs = 100

The results:

- NLL = 62.9973
- EMD = 1676.230 ± 52.132
  - 0 EMD w.r.t train split: 1664.721
  - 1 EMD w.r.t train split: 1675.890
  - 2 EMD w.r.t train split: 1594.222
  - 3 EMD w.r.t train split: 1757.264
  - 4 EMD w.r.t train split: 1689.052

2 Classifier-free Guidance

2.1 The class `ConditionalDDPM`: Neural Network $\vec{\epsilon}_{\vec{\theta}}(\vec{x}, t, y)$

This model is very similar to the class `DDPM`, except for that it embeds the class as well, and passes the vector $(\vec{x}, \vec{y}_{embed}, \vec{t}_{embed})$, which is a similar concatenated vector, passed as a 2D tensor of appropriate sizes and dimensions.

There is one additional model for embedding the class (y): We assume actual classes vary between `0` and `n_samples - 1`, and keep class `n_samples` for the 'unconditional' class, for unconditional training, which is also needed for the CFG.

The model uses a one-hot encoding to size `n_samples + 1`, followed by a linear layer that maps it to a vector of the same size.

Here are the sampling results generated via hyperparameter fine-tuning and optimum guidance scale choice:

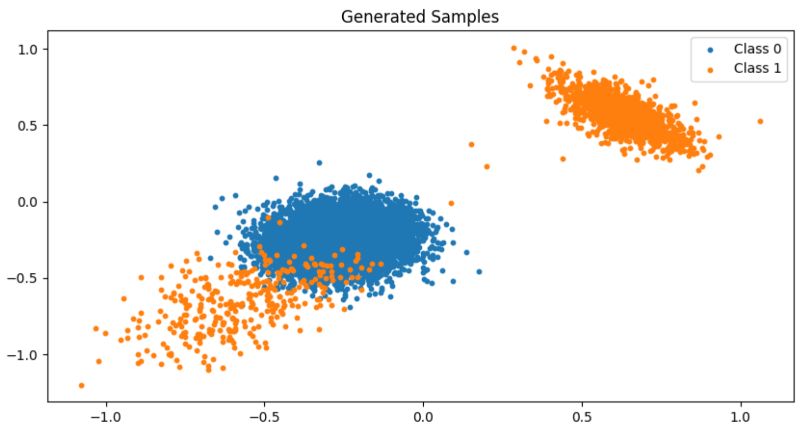

Figure 6: Sampling Scatter Plot for Blobs Dataset: Guidance Scale = 0.5

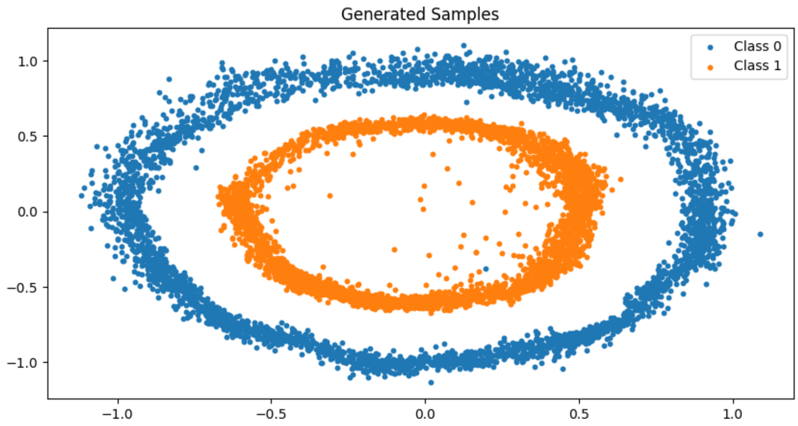

Figure 7: Sampling Scatter Plot for Circles Dataset: Guidance Scale = 1.0

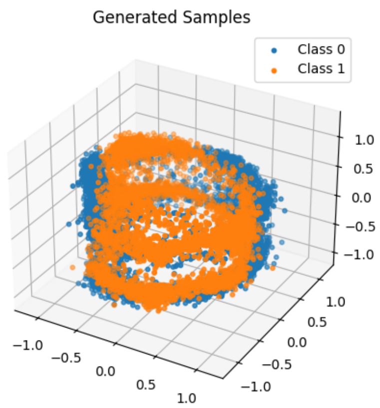

Figure 8: Sampling Scatter Plot for Helix Dataset: Guidance Scale = 1.0

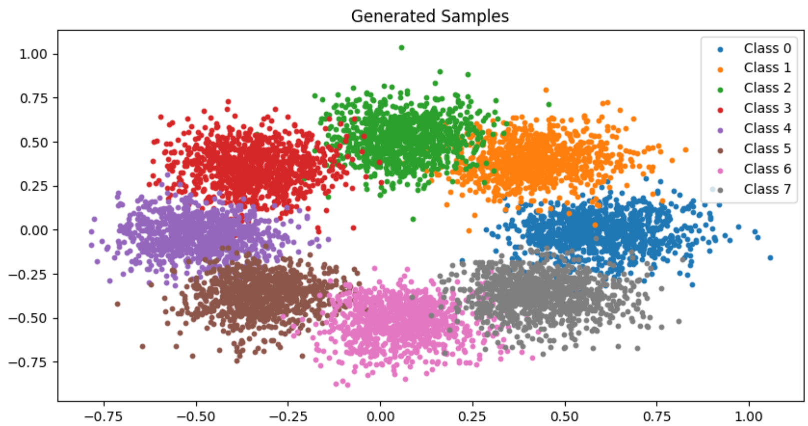

Figure 9: Sampling Scatter Plot for Manycircles Dataset: Guidance Scale = 0.2

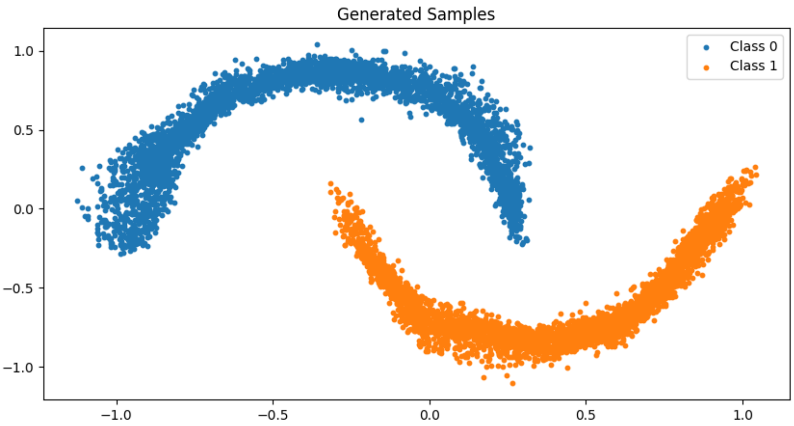

Figure 10: Sampling Scatter Plot for Moons Dataset: Guidance Scale = 1.0

2.2 The class `ClassifierDDPM`

Given a conditional DDPM, this class creates an object (with no additional training) that infers the class of an input vector $\vec{x}$ with 90% accuracy. Its functioning is as follows:

1. Given a vector $\vec{x}$, create a vector $\vec{\Delta} = \vec{0}$ of size `n_classes`.
2. Take $n_t = 10$ random timestamps. Repeat steps 3 to 6 for each timestamp, denoted t.
3. Compute $\vec{x_t} = \sqrt{\overline{\alpha}_t}\vec{x} + \sqrt{1 - \overline{\alpha}_t}\vec{z}$, where $\vec{z} \sim \mathcal{N}(\vec{0}, \mathbf{I})$
4. Compute the model $\vec{\epsilon}_{\vec{\theta}}(\vec{x}_t, t, y)$, for each $y = 0, 1, \ldots$ `n_classes`.
5. Store the results as $\vec{r}_i \quad \forall i \in \{0, 1, \ldots \texttt{n\_classes}\}$.
6. For all $i \in \{0, 1, \ldots \texttt{n\_classes} - 1\}$, update $\Delta_i \leftarrow \Delta_i + \|\vec{r}_i - \vec{r}_{\texttt{n\_classes}}\|^2$.
7. Compute $\vec{p} = softmax(\vec{\Delta})$ and that gives us the probabilities that $\vec{x}$ belongs to respective classes.
8. The predicted class is $\arg\max_i(\Delta_i) = \arg\max_i(p_i)$.

The principle behind this is that in case we have an input with y not being the class of $\vec{x}$, then the model would give a suboptimum direction (for drift during denoising), that is closer to that for the unconditional case, than the case for the best y.

We sum the absolute differences across many random timestamps to minimize random errors. All this is done for a batch of vectors, as 2D tensors, that are passed as input.

2.3 The method `sampleCFG`

We are given a parameter 'guidance scale' w. The sampling is very similar to the conditional sampling, except that the update step is

$$\begin{aligned} \vec{x}_{t-1} &= \sqrt{\overline{\alpha}_t}\left(\vec{x}_t + (1+w)\cdot\vec{\epsilon}_{\vec{\theta}}(\vec{t}_t, t, y) - w\cdot\vec{\epsilon}_{\vec{\theta}}(\vec{x}_t, t)\right) + \sqrt{1-\overline{\alpha}_t}\vec{z} \qquad \text{where} \quad \vec{z} \sim \mathcal{N}(\vec{0}, \mathbf{I}) \\ &= (1+w)\cdot\vec{x}_{t-1,\text{conditional}} - w\cdot\vec{x}_{t-1,\text{unconditional}} \end{aligned}$$

3 References

- ChatGPT, for ideation of layer sizes and architecture.
- Lecture slides for the algorithm
- The ipynb colab
- Sinusoidal embeddings