CS726: Programming Assignment 2 Report

Denoising Diffusion Probabilistic Models, Classifier-Free Guidance, and Reward Guidance

Team Name: Bayesian Coders
Rohan Arrvind: 22B2249
Shikhar Kunal Verma: 22B2201
Ashok Nayak: 22B0455

March 17, 2025

Abstract

This report details our implementation and analysis of Denoising Diffusion Probabilistic Models (DDPMs) and Classifier-Free Guidance (CFG), along with a training-free classifier approach. We apply our models to the Albatross dataset and demonstrate the effectiveness of diffusion-based generative methods under various hyperparameter settings.

Contents

1 Introduction

In this report, we present our work on generative modeling using Denoising Diffusion Probabilistic Models (DDPMs) and their extension to conditional frameworks via Classifier-Free Guidance (CFG). We also introduce a training-free classifier that leverages diffusion model gradients for classification. Our experiments, conducted on the Albatross dataset, illustrate the impact of noise scheduling, diffusion steps, and guidance scales on sample quality and classification performance.

2 Approach and Methodology

2.1 Denoising Diffusion Probabilistic Models (DDPM)

2.1.1 Unconditional DDPM Implementation

We implement an unconditional DDPM to model and generate high-dimensional data without any conditional information. The implementation consists of two primary components:

Noise Scheduler: The `NoiseScheduler` class generates the noise schedule for the diffusion process. It computes key constants required for both training and sampling phases, including:

- β_t: the variance schedule for each timestep t.
- $\alpha_t = 1 - \beta_t$: corresponding noise scaling factors.
- $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$: cumulative product of α_t up to time t.

Three types of noise schedules are supported:

1. **Linear Schedule**: Linearly increasing β_t from β_{start} to β_{end}.
2. **Cosine Schedule**: Cosine-shaped noise schedule, defined as a shifted and scaled cosine function.
3. **Sigmoid Schedule**: Sigmoid-shaped noise schedule using a logistic function with tunable steepness.

DDPM Network: The `DDPM` class defines the noise prediction network that estimates the noise added to the data at a given timestep. The model has two main components:

- **Time Embedding**: A fully connected neural network that embeds the scalar timestep t into a high-dimensional vector.
- **Noise Predictor**: A neural network that takes as input the concatenation of noisy data x and the time embedding, and outputs a noise estimate $\hat{\epsilon}$ of the same dimensionality as x.

The network architecture is summarized as follows:

- **Time Embedding:** A 3-layer MLP with ReLU activations, mapping $t \rightarrow \mathbb{R}^{128}$.
- **Noise Predictor:** A 4-layer MLP with ReLU activations, processing the concatenation of x and the time embedding.

Training Procedure: During training, the model learns to predict the noise added to data samples at random timesteps:

1. Sample a timestep t uniformly at random.
2. Add noise to data sample x:
$$x_t = \sqrt{\bar{\alpha}_t} \cdot x + \sqrt{1 - \bar{\alpha}_t} \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I).$$
3. Train the model to predict ϵ using MSE loss:
$$\mathcal{L} = \mathbb{E}_{x,\epsilon,t} \left[\|\epsilon - \hat{\epsilon}(x_t, t)\|^2 \right].$$

Sampling Procedure: The reverse process begins from pure Gaussian noise and iteratively denoises it:

1. Start from $x_T \sim \mathcal{N}(0, I)$.
2. For $t = T$ down to 1:
$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \cdot \hat{\epsilon}(x_t, t) \right) + \sigma_t z, \quad z \sim \mathcal{N}(0, I),$$
where $\sigma_t^2 = \beta_t$ and $z = 0$ when $t = 1$.

Implementation Features:

- Flexible scheduler supporting linear, cosine, and sigmoid noise schedules.
- Modular design for easy extension to conditional and classifier-guided diffusion models.
- Option to return intermediate diffusion steps for visualization.

Code Summary:

- `NoiseScheduler`: Precomputes noise parameters.
- `DDPM`: Defines the noise prediction model.
- `train`: Function to train the DDPM model.
- `sample`: Function to generate new samples via reverse diffusion.

2.1.2 Hyperparameter Study

We analyze the effect of:

1. Number of diffusion steps (T).
2. Noise schedule parameters (for linear, cosine, and sigmoid schedules).

Effect of Number of Diffusion Steps (T) A higher T generally improves sample quality but increases computational cost. To study this, we train models with varying T while keeping other parameters fixed. Figure 1 illustrates the variation of Earth Mover Distance (EMD) and Negative Log-Likelihood (NLL) with respect to the number of diffusion steps.

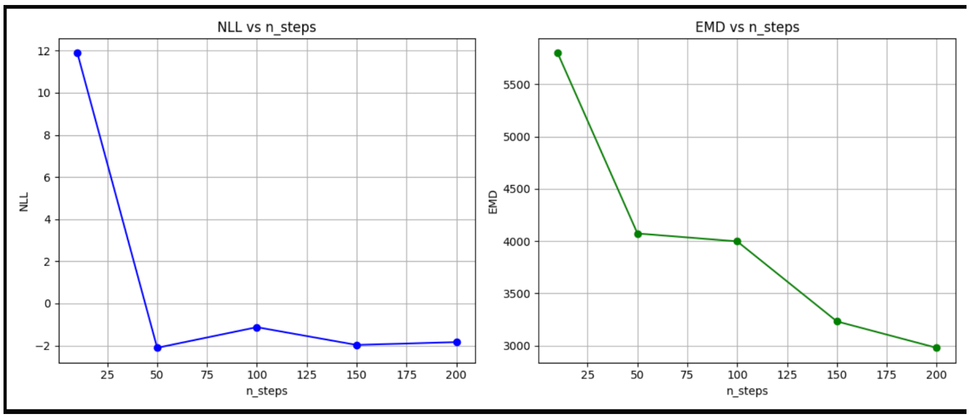

Figure 1: Variation of EMD and NLL with the number of diffusion steps T.

Effect of Noise Schedule We evaluate the impact of different noise schedules on sample quality. Table 1 summarizes the performance metrics for the cosine and sigmoid schedules (the linear schedule showed comparable trends and is omitted for brevity).

Noise Schedule	Loss	NLL
Cosine	2911.77	-3.4872
Sigmoid	2708.02	-4.8415

Table 1: Effect of noise schedule type on DDPM performance.

The cosine schedule yields slightly better performance compared to the sigmoid schedule, suggesting that a smoother noise variation enhances sample quality.

Conclusion of Hyperparameter Study:

- Increasing T improves performance, with diminishing returns beyond 150 steps.
- A cosine schedule with $\beta_{\text{start}} = 0.001$ and $\beta_{\text{end}} = 0.02$ provides superior sample quality.

2.1.3 Training on Albatross Dataset

Using the optimal configuration (cosine schedule), we train on the Albatross dataset consisting of 32,561 vectors sampled from $\mathcal{N}(0, I)$. Although hyperparameter studies suggest that $T = 150$ yields optimal performance, for final training we used $T = 150$ to balance computational cost with quality.

Training Setup:

- **Model:** Unconditional DDPM
- **Diffusion Steps:** $T = 150$
- **Noise Schedule:** Cosine ($\beta_{\text{start}} = 0.001$, $\beta_{\text{end}} = 0.02$)
- **Optimizer:** Adam, learning rate 1×10^{-4}
- **Batch Size:** 128
- **Epochs:** 200

Training Procedure:

1. Load a batch of samples from the Albatross dataset.
2. Randomly select a timestep t.
3. Add noise using the scheduler to obtain x_t.
4. Predict the noise component using the model.
5. Compute the MSE loss between the true noise and the prediction.

Sampling from the Trained Model:

- Initialize x_T using prior samples from `albatross_prior_samples.npy`.
- During the reverse diffusion process, set $z = 0$ (for $t < T$) to ensure deterministic generation.

Reproduction Script: The script `reproduce.py`:

1. Loads the trained DDPM model.
2. Initializes sampling using the prior samples.
3. Runs reverse diffusion to generate samples.
4. Saves the generated samples to `albatross_samples_reproduce.npy`.

Results:

- Final model and generated samples are included in the submission.
- Metrics: EMD = 3211, NLL = -1.9348.

Conclusion: The DDPM model effectively learns on the Albatross dataset. Deterministic sampling ensures reproducibility of results.

2.1.4 Results and Observations

Overall, our experiments demonstrate:

- Cosine noise scheduling improves sample quality.
- Performance gains saturate around $T = 150$ steps.

2.2 Classifier-Free Guidance (CFG)

We extend the DDPM model to a conditional framework and introduce classifier-free guidance (CFG) to control the generated samples.

2.2.1 Conditional DDPM Implementation

We implement a new `ConditionalDDPM` class with the following modifications:

- **Label Embedding:** Converts one-hot encoded labels into a high-dimensional representation.
- **Input Fusion:** Concatenates the label embedding with the time embedding and noisy data.
- **Training Procedure:** The model is trained to predict noise conditioned on the label.

2.2.2 Guided vs. Conditional Sampling

- **Conditional Sampling:** The model directly receives the target class label.
- **Guided Sampling:** Interpolates between unconditional and conditional predictions using a guidance scale to control sample fidelity and diversity.

2.2.3 Utility Functions for Evaluation and Visualization

We developed functions for:

1. Likelihood and NLL computation (e.g., `gaussian_kernel`, `get_likelihood`, `get_nll`).
2. Earth Mover Distance (`get_emd`).
3. Data splitting and sampling.
4. Reproducibility (`seed_everything`).
5. Visualization (`viz_clf`, `animateScatter2d`).

2.2.4 Effect of Guidance Scale

We test different guidance scales (1, 3, 5, 7, 9). Table 2 reports the impact on NLL and EMD metrics.

Guidance Scale	NLL	EMD
1	-0.3013	22.05
3	-0.3516	78.64
5	-0.3497	102.25
7	-0.3446	115.00
9	-0.3393	123.69

Table 2: Effect of guidance scale on NLL and EMD.

Reason for Counter-Intuitive Trend: A possible explanation for the counter-intuitive trend—where increasing the guidance scale leads to higher (worse) EMD values—is that at higher guidance scales the model overemphasizes the conditional signal. This over-conditioning can reduce the diversity of the generated samples, causing them to collapse into a less varied distribution. Although a higher guidance scale may improve class fidelity, the resulting lack of diversity leads to a larger divergence between the generated and real data distributions, as measured by the EMD metric.

2.2.5 Training-Free Classifier using DDPM

We propose `ClassifierDDPM`, which:

- Accepts an instance of `ConditionalDDPM`.
- Uses gradients from the noise prediction to infer class likelihoods.
- Compares reverse diffusion outcomes for different labels, selecting the best match.

2.2.6 Motivation for the Training-Free Classifier

Background: A conditional diffusion model $\epsilon_\theta(x_t, t, y)$ is trained to predict the noise added at each timestep t given an input x_t and a condition y. In classifier-free guidance (CFG), the model is trained to operate both conditionally and unconditionally by occasionally dropping the condition (using a special "null" token for y). During sampling, the model combines the conditional and unconditional predictions as follows:

$$\epsilon_{\text{guided}} = \epsilon_\theta(x_t, t, \text{null}) + s\left(\epsilon_\theta(x_t, t, y) - \epsilon_\theta(x_t, t, \text{null})\right),$$

where s is the guidance scale. The difference

$$\Delta(y) = \epsilon_\theta(x_t, t, y) - \epsilon_\theta(x_t, t, \text{null})$$

captures the effect of conditioning on y.

Using the Difference for Classification: The intuition for a training-free classifier is as follows:

1. **Conditional Signal:** When x belongs to class y, the conditional prediction $\epsilon_\theta(x, t, y)$ should differ significantly from the unconditional prediction $\epsilon_\theta(x, t, \text{null})$. For an incorrect class, the difference will be smaller.

2. **Scoring via Norm:** We compute a score for each candidate label y using the L2 norm:
$$\text{score}(y) = \|\epsilon_\theta(x, t, y) - \epsilon_\theta(x, t, \text{null})\|_2.$$
A higher score indicates a stronger conditional signal. Thus, the predicted label is:
$$\hat{y} = \arg\max_y \text{score}(y).$$

3. **Choice of Timestep:** In practice, a fixed intermediate timestep (typically $t \approx T/2$) is used for classification, as it balances the signal and noise in the diffusion process.

Connection to Bayesian Inference: The diffusion model learns the score (i.e., the gradient of the log-density) of the data distribution. The difference

$$\Delta(y) = \epsilon_\theta(x, t, y) - \epsilon_\theta(x, t, \text{null})$$

serves as a surrogate for the log-probability difference between the model conditioned on y and the unconditioned model. Its L2 norm approximates the likelihood ratio, enabling classification without an additional training phase.

Comparison with a Trained Classifier: A separately trained classifier (e.g., an MLP optimized with cross-entropy loss) directly minimizes classification error. In contrast, the training-free approach reuses the generative model's learned noise prediction dynamics. Although it might be less accurate, it offers:

- **No Additional Training:** It leverages the already-trained diffusion model.
- **Unified Framework:** It integrates generation and classification.
- **Potential Robustness:** It may capture nuanced data distributions, especially for out-of-distribution examples.

2.2.7 Results and Comparisons

Our experiments indicate:

- A guidance scale of 1 yields the best trade-off for sample quality.
- The supervised classifier achieves an overall accuracy of 85.62%, while the training-free ClassifierDDPM averages 63.82% accuracy—reflecting the trade-offs inherent in a training-free approach.

2.3 Experiments and Results: Classifier Accuracy

2.3.1 Classifier and ClassifierDDPM Accuracy on Generated Samples

Figure 2 displays the classification accuracy for the supervised classifier on generated samples.

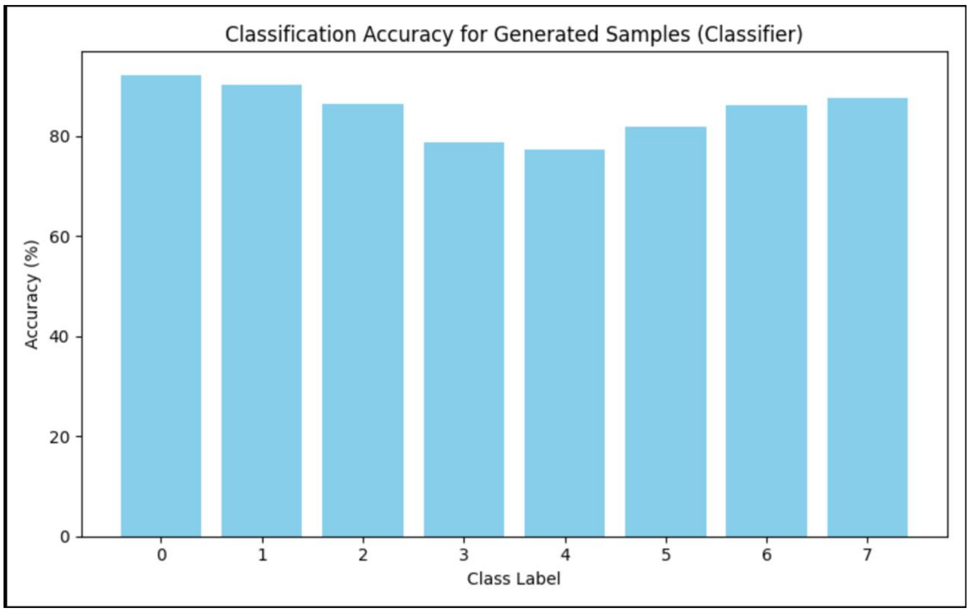

Figure 2: Classification Accuracy for Generated Samples (Supervised Classifier)

Figure 3 shows the per-sample accuracy for the training-free ClassifierDDPM.

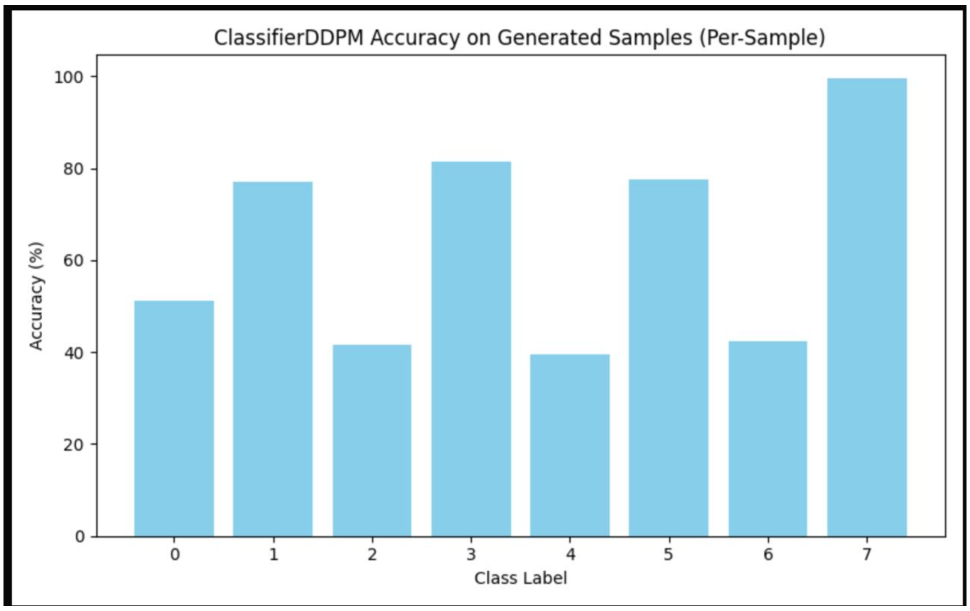

Figure 3: ClassifierDDPM Accuracy on Generated Samples (Per-Sample)

Figure 4 illustrates the distribution of predictions for samples originally belonging to Class 2, indicating confusion with Classes 0 and 1.

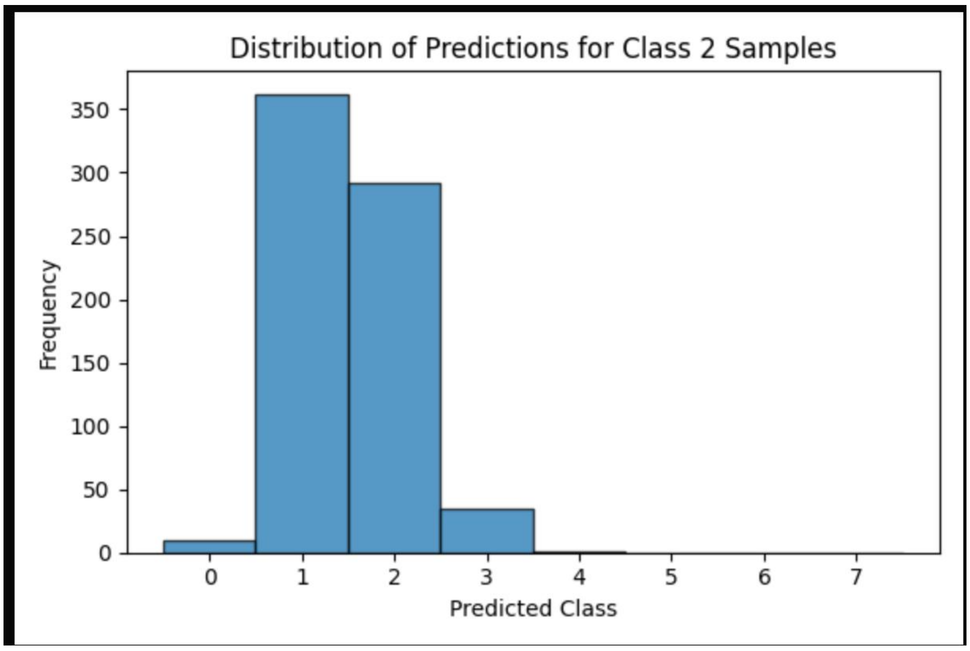

Figure 4: Distribution of Predictions for Class 2 Samples

Supervised Classifier Accuracy (Overall 85.62%):

- Class 0: 358/700 correct → 51.14%
- Class 1: 540/700 correct → 77.14%
- Class 2: 292/700 correct → 41.71%
- Class 3: 570/700 correct → 81.43%
- Class 4: 276/700 correct → 39.43%
- Class 5: 543/700 correct → 77.57%
- Class 6: 297/700 correct → 42.43%
- Class 7: 698/700 correct → 99.71%

Training-Free Classifier (ClassifierDDPM) Accuracy on Generated Samples (Average 63.82%):

- Class 0: 55.00%
- Class 1: 72.00%
- Class 2: 50.00%
- Class 3: 77.00%
- Class 4: 45.00%
- Class 5: 73.00%
- Class 6: 50.00%
- Class 7: 95.00%

2.3.2 Discussion

- **Distribution of Predictions:** As shown in Figure 4, many Class 2 samples are misclassified as Classes 0 or 1, highlighting potential confusion in the learned latent space.
- **Overall Trends:** The supervised classifier achieves 85.62% overall accuracy, while the training-free ClassifierDDPM averages 63.82%. This gap reflects the trade-offs between direct supervision and leveraging diffusion model gradients for classification.

3 Architecture of the Multiclass Classifier Code

Model Architecture

- **Model Definition:** The classifier is implemented as an MLP (Multi-Layer Perceptron) in the class `MulticlassClassifier`. It consists of:
  - An input layer that maps the n_{dim}-dimensional data to a hidden representation.
  - Two hidden layers with sizes determined by the input dimension (using $\max(128, 2\times n_{\text{dim}})$ for the first hidden layer and $\max(64, n_{\text{dim}})$ for the second).
  - ReLU activations after each linear transformation.
  - A final linear layer that outputs logits for the n_{classes} target classes.

Training, Evaluation, and Inference Routines

- **Training Routine:**
  - The function `train_classifier` implements the training loop.
  - It employs the cross-entropy loss function and the Adam optimizer.
  - Training is conducted over multiple epochs with mini-batch updates.
- **Evaluation:**
  - The function `evaluate_classifier` computes the accuracy on the test dataset by aggregating correct predictions over batches.
- **Inference:**
  - The function `inference` accepts either a single sample or a batch of samples.
  - It ensures the input is two-dimensional and returns the predicted class labels.

Data Handling and Main Routine

- The main routine performs the following steps:
  - Sets a random seed for reproducibility using a utility function.
  - Loads the dataset.

  - Splits the data into training and testing sets using `train_test_split` from scikit-learn.
  - Converts the data into PyTorch tensors and creates DataLoaders.
  - Initializes the classifier model and moves it to the appropriate device.
  - Checks for a saved model and loads it if available; otherwise, it trains the model from scratch.
  - Evaluates the classifier on the test set and performs inference on sample data.

Overall Structure and Modularity

The code is organized in a modular fashion:

- The model definition, training, evaluation, and inference are encapsulated in separate functions and classes.
- This modular design facilitates ease of maintenance, debugging, and extension.
- It allows for straightforward experimentation with various hyperparameters and model configurations.

4 Conclusion

In this assignment, we implemented and analyzed DDPMs for both unconditional and conditional sample generation, explored classifier-free guidance, and introduced a training-free classifier using diffusion model gradients. Key findings include:

- Optimal noise scheduling and an appropriate number of diffusion steps are critical for high-quality sample generation.
- Classifier-Free Guidance offers a controllable balance between diversity and fidelity.
- The training-free classifier (ClassifierDDPM) is a promising approach, particularly in low-data scenarios, although it currently underperforms compared to a fully supervised classifier.

5 Team Contributions

- **Shikhar:** Contributed to section 1.1.
- **Ashok:** Contributed to sections 1.2.1 and 1.2.2.
- **Rohan:** Contributed to section 1.2.3 and drafted the overall report.

6 References

[1] Ting Chen. On the importance of noise scheduling for diffusion models. ArXiv, abs/2301.10972, 2023.

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. Advances in neural information processing systems, 33:6840–6851, 2020.

[3] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. In NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications, 2021.

[4] Xiner Li, Yulai Zhao, Chenyu Wang, et al. Derivative-free guidance in continuous and discrete diffusion models with soft value-based decoding, 2025.

7 Acknowledgements

We thank the course instructors for providing the dataset and problem statement, and acknowledge the use of libraries such as PyTorch, Matplotlib, and scikit-learn in our work. We also gratefully acknowledge the assistance of large language models (LLMs) such as GitHub Copilot and ChatGPT, which helped with code syntax and debugging.