CS726 Assignment 2(DDPM)

Kushal Jain Nimay Shah Anay Arora

March 2025

Abstract

This report provides a detailed explanation of the approach and the implementation details of the code written in the assignment. The DDPM model is a generative approach that iteratively refines noisy samples to generate data samples such as images. We discuss the model architecture, training process, and sampling algorithm used in this implementation.

1 Introduction

1.1 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models (DDPMs) are a class of generative models that learn to transform noise into realistic data samples through a sequential denoising process. The model works by first adding noise to an image over several steps and then training a neural network to reverse this process.

1.1.1 Model Architecture and Approach

Our DDPM implementation is a neural network that predicts noise in an input sample at a given timestep. The architecture consists of:

- A **time embedding** module to encode the timestep into a higher-dimensional space. We do this because timesteps are discrete and simply feeding raw values of timestep t does not work since the model may overfit to some specific timestep values. Neural networks are known to struggle with absolute positions and large numerical values. Hence we encode the timestep in a sinusoid. These provide a smooth representation that allows the model to generalize across various timesteps.

- A **noise-prediction network** that takes the noisy input and the time embedding to predict noise. Since we're dealing with lower dimensional data(n_dim = 2) we can use dense fully connected layers instead of something like U-Net architecture which we would have preferred in case of something like image generation. Here fully connected layers are simpler and more efficient. We also use the SiLu activation function instead of ReLu because it is smooth and differentiable everywhere. It prevents dead neurons i.e. neurons stuck as zero.

1.1.2 Time Embedding

The time embedding is implemented using a sinusoidal encoding followed by a feedforward network:

```
def _timestep_embedding(self, t, dim=32, max_period=10000):
    half = dim // 2
    freqs = torch.exp(-math.log(max_period) * torch.arange(
   half, device=t.device) / half)
    args = t[:, None].float() * freqs[None]
    emb = torch.cat([torch.cos(args), torch.sin(args)], dim
   =-1)
    return emb
```

The function determines the embedding dimension dim (default: 32) and splits it into two halves. Then it generates a set of logarithmically spaced frequencies, ensuring that lower indices capture high-frequency variations while higher indices represent low-frequency trends. The input timestep t is then multiplied by these frequencies, producing a tensor of phase values that uniquely encode each timestep. The function then applies sine and cosine transformations to these phase values and concatenates them to create a dimensional embedding.

1.1.3 Noise-Prediction Network

The main neural network consists of fully connected layers with activation functions to process input features:

```
self.model = nn.Sequential(
    nn.Linear(n_dim + 128, 256),
    nn.SiLU(),
    nn.Linear(256, 256),
    nn.SiLU(),
    nn.Linear(256, n_dim)
)
```

The input to the model is a concatenation of the noisy data sample and the time embedding. The model predicts the noise present in the input sample. Initially the input data + the time embedding is the total input dimension size(n_dim + 128). Layer 1 is 256 neurons. It is fully connected with another 256 neuron layer. The final output layer is composed of input dimension 256 and outpput size same as the data n_dim.

1.1.4 Training Process

During training, the model is optimized to predict the noise added at each timestep. The steps are as follows:

1. Randomly select a timestep t.
2. Generate Gaussian noise and add it to the input data.
3. Use the model to predict the noise.
4. Compute loss using mean squared error (MSE):

$$\mathcal{L} = \mathbb{E}_{x,t,\epsilon}\left[||\epsilon - f_\theta(x_t, t)||^2\right] \tag{1}$$

The training loop is implemented as follows:

```
def train(model, noise_scheduler, dataloader, optimizer,
    epochs, run_name):
    for epoch in range(epochs):
        total_loss = 0
        for x, _ in tqdm(dataloader):
            x = x.to(device)
            t = torch.randint(0, noise_scheduler.
    num_timesteps, (x.size(0),), device=x.device)
            eps = torch.randn_like(x)
            x_noisy = sqrt_alpha_bar * x + sqrt_one_minus *
    eps
```

```
            pred_eps = model(x_noisy, t)
            loss = F.mse_loss(pred_eps, eps)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
```

Over here, x_{noisy} represents the noisy sample generated by doing $\sqrt{\alpha_t}x + \sqrt{1-\alpha_t}\epsilon$. We then predict an epsilon using our neural network and train it using the above MSE loss function. We do this for the different number of timesteps and the different settings of noise schedules.

1.1.5 Sampling Process

Once trained, the model can generate samples by reversing the noise process. It starts from pure noise and iteratively denoises it using the learned model:

```
def sample(model, n_samples, noise_scheduler,
   return_intermediate=False):
    x = torch.randn(n_samples, model.n_dim).to(device)
    for t in tqdm(reversed(range(noise_scheduler.
   num_timesteps))):
        t_batch = torch.full((n_samples,), t, device=device,
   dtype=torch.long)
        pred_eps = model(x, t_batch)
        x = (1 / torch.sqrt(alpha_t)) * (x - (beta_t / torch.
   sqrt(1 - alpha_bar_t)) * pred_eps)
        if t > 0:
            x += torch.sqrt(beta_t) * torch.randn_like(x)
```

This process progressively removes noise and reconstructs a clean sample.

1.1.6 Results

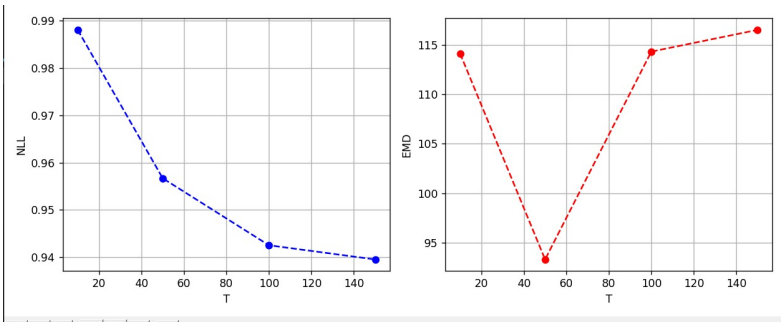

Figure 1: NLL(left) and EMD(right) v/s no. of timesteps

NLL consistently decreases with the number of timesteps. This means the model is better at modelling the data distribution.
EMD initially decreases but then increases after a certain point(T = 50). EMD measures the difference b/w the generated and real data distribution. A lower EMD indicates better quality samples. The minimum EMD occurs at 50 steps indicating that this may be the optimal number of timesteps. We interpret that more timesteps improve likelihood but may not always improve sample quality.
Over here we can see that the very gradual noise schedule has the lowest EMD and NLL. This means the sample quality and the likelihood are both the best when the noise schedule is very gradual i.e. less noise is added per step and the model generalizes well. The original noise schedule had the worst EMD, which indicates that the standard schedule may not be optimal for this dataset. As the diffusion schedule becomes faster (fast, wide_range),

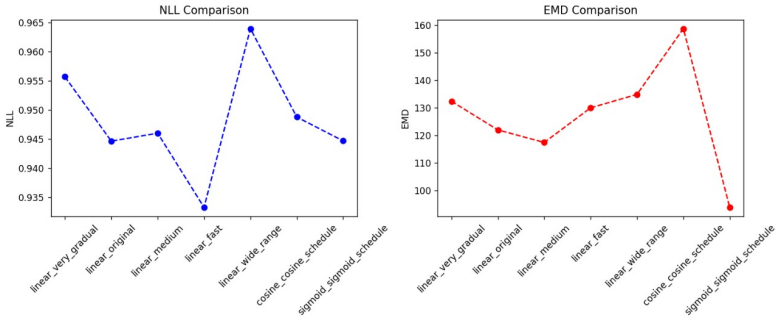

Figure 2: NLL and EMD v/s for different noise schedules including cosine and sigmoid

NLL increases (indicating worse likelihood), and EMD fluctuates. This suggests that removing noise too quickly reduces the model's ability to generate realistic samples. Also we see that cosine and sigmoid are generally good at keeping likelihood low, but sigmoid is able to generate really good quality

1.2 Albatross dataset

The file reproduce.py contains the code to generate the samples based on the albatross dataset.

The following code implements **deterministic sampling** using a trained Denoising Diffusion Probabilistic Model (DDPM). The process involves starting with **noisy prior samples** and iteratively refining them using a trained noise prediction model. Below is a detailed explanation.

1.2.1 Setting Configuration Parameters

```
# Configuration
n_steps = 200  # Total timesteps for diffusion process
lbeta = 1e-4   # Lower bound of noise variance
ubeta = 0.02   # Upper bound of noise variance
n_dim = 64     # Dimensionality of data (e.g., flattened 14
    x14 images)
device = 'cpu' # Runs on CPU for reproducibility
```

The key configuration parameters are as follows:

- `n_steps = 200`: Number of diffusion steps.
- `lbeta` & `ubeta`: Define the linear noise schedule (how much noise is added per step).
- `n_dim = 64`: The data size (a flattened image or vector representation).

1.2.2 Loading the Pretrained DDPM Model

```
model = DDPM(n_dim=n_dim, n_steps=n_steps).to(device)
model.load_state_dict(torch.load("albatross_model.pth",
    map_location=device))
```

A pretrained DDPM model is loaded using the weights from `"albatross_model.pth"`.

```
noise_scheduler = NoiseScheduler(n_steps, "linear",
    beta_start=lbeta, beta_end=ubeta)
```

A `NoiseScheduler` is initialized, which precomputes values for α_t, $\bar{\alpha}_t$, and β_t.

1.2.3 Loading Prior Samples

```
prior_samples = np.load("data/albatross_prior_samples.npy")
samples = torch.from_numpy(prior_samples).float().to(device)
```

Prior samples are loaded from a NumPy file and converted into PyTorch tensors.

1.2.4 Deterministic Sampling Process

```
with torch.no_grad():
    for t in reversed(range(n_steps)):  # Reverse diffusion
    process
        t_batch = torch.full((len(samples),), t, device=
    device, dtype=torch.long)
```

The sampling process iterates **backward** from the last timestep to $t = 0$, reversing the diffusion process.

```
        pred_eps = model(samples, t_batch)  # Predict noise
    at timestep t
```

At each timestep, the model predicts the noise ϵ that was originally added to the sample.

```
        alpha_t = noise_scheduler.alphas[t]
        alpha_bar_t = noise_scheduler.alpha_bars[t]
        beta_t = noise_scheduler.betas[t]
```

The necessary noise schedule values are retrieved for the current timestep.

```
        # Reverse process step
        samples = (1 / torch.sqrt(alpha_t)) * (samples - (
    beta_t / torch.sqrt(1 - alpha_bar_t)) * pred_eps)
```

This step applies the **denoising equation** to progressively remove noise from the sample.

```
        if t > 0:
            samples += torch.sqrt(beta_t) * torch.zeros_like(
    samples)
```

In deterministic sampling, instead of adding random noise, $z = 0$ is used to ensure reproducibility.

1.2.5 Saving the Final Generated Samples

```
np.save("albatross_samples_reproduce.npy", samples.cpu().
    numpy())
```

The final denoised samples are saved as a NumPy file.

1.3 Classifier-Free Guidance

1.3.1 Difference b/w guided and conditional sampling

Diffusion models generate samples by iteratively removing noise from an initial random example. In their standard form, they are unconditional, meaning they generate samples without any specific control over the output. However to make the models more controllable, two main approaches exist:

- Guided Sampling: Adjust the sampling process at inference time to push the model towards the desired output.
- Conditional Sampling: Train the model to generate conditioned outputs directly.

Guided Sampling
Classifier guided sampling relies on a pre-trained classifier p(y/x_t) which provides gradients to modify the denoising process. The standard denoising equation is $\tilde{\epsilon}_\theta(x_t, t) = \epsilon_\theta(x_t, t)$ To guide the process toward a class y, we modify it using the classifier's gradient: $\tilde{\epsilon}_\theta(x_t, t, y) = \epsilon_\theta(x_t, t) - w \cdot \nabla_{x_t} \log p(y \mid x_t)$. **w** is the guidance strength, controlling how much the classifier influences the generation.
$\nabla_{x_t} logp(y|x_t)$ is the gradient of the classifier probability with respect to x_t, which pushes the sample toward class y. If the classifier detects that the current sample is not close enough to the target class, it provides a correction signal. The diffusion model adjusts its trajectory to make the sample more likely to belong to the desired category.

Classifier-free guidance removes the need for an external classifier by training the diffusion model to predict both conditional and unconditional noise estimates. The model is trained in two ways: unconditionally learns $\epsilon_\theta(x_t, t)$ and conditionally trained with conditioning information. $\tilde{\epsilon}_\theta(x_t, t, y) = (1 + w) \cdot \epsilon_\theta(x_t, t, y) - w \cdot \epsilon_\theta(x_t, t)$ This equation dictates the final noise prediction during inference. $\epsilon_\theta(x_t, t, y)$ is the conditional noise prediction and $\epsilon_\theta(x_t, t)$ is the unconditional noise prediction.

Conditional Sampling
Conditional sampling trains the diffusion model itself to generate samples based on input conditions without modifying the inference process. The model is trained on paired data (x_0, y) where y is a conditioning signal. During training, the model learns to generate data based on the condition rather than modifying predictions afterward. For e.g. is one we have done in class: The model is trained with labeled data: Input: Image from class y. Output: Model learns to generate images of that class. $p(x_0 \mid y) =$

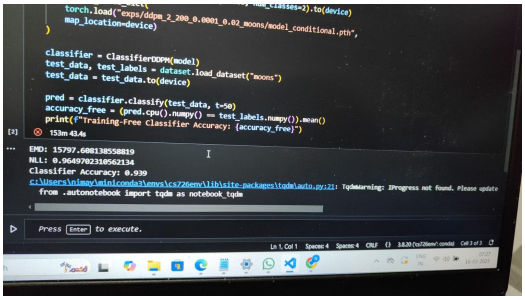

Figure 3: Results obtained in classifier

$\int p(x_T) \prod_{t=1}^{T} p(x_{t-1} \mid x_t, y)\, dx_t$. The model learns a class-conditional noise estimator $\epsilon_\theta(x_t, t, y)$. No need for classifier guidance during inference.

1.3.2 Trained classifier and effect of guidance scale

We can use metrics such as EMD and NLL to determine the effect of guidance scale w on sample quality. EMD quantifies how much "work" is needed to move the generated distribution to align with the real data distribution. Lower EMD means generated samples are closer to real samples. We can compute EMD between generated and real samples for different guidance scales w. NLL measures how well the model predicts real data. Higher NLL suggests that the generated samples deviate from the true distribution. We can compute NLL on real data vs. generated data for different guidance scale and track how NLL changes with stronger guidance.
Accuracy of the classifier obtained: 93.5%
NLL of the classifier obtained: 0.964
EMD of the classifier obtained: 15797

2 Contributions

- Anay Arora: Coding and debugging part 1.1(DDPM Question)
- Nimay Shah: Coding and debuggin part 1.2(CFG)
- Kushal Jain: Making the report and debugging part 1.1 and 1.2 with Nimay and Anay.