All things you must know about "Denoising Diffusion Probabilistic Models"

Mehran Tamjidi
Department of Electrical & Computer Science
KN.Toosi University of Technology
mehrant.0611@gmail.com

Abstract

Diffusion models have emerged as a prominent class of generative models, offering a unique and effective approach to data generation. Unlike traditional models such as Variational Autoencoders (VAEs) and flow models, diffusion models utilize a Markov chain of diffusion steps to gradually introduce random noise to data. This process is subsequently reversed through learned noise reduction steps to reconstruct the original data from pure noise, ensuring high-dimensional latent variables that preserve data integrity. In this article, we delve into the theoretical underpinnings of diffusion models, elucidating the forward and reverse diffusion processes. We provide a comprehensive implementation of diffusion models and demonstrate their application on two datasets: the widely recognized MNIST dataset and the Persian digit dataset. Through detailed experiments and analysis, we illustrate the efficacy of diffusion models in generating high-quality data samples, highlighting their potential in various machine learning and data science applications. Our step-by-step guide offers a practical framework for implementing diffusion models, making this powerful generative approach accessible to researchers and practitioners alike.

Contents

List of Figures

1 What is the diffusion model?

Diffusion models have emerged as a powerful class of generative models that are gaining popularity in various fields, particularly in machine learning and data science. Unlike traditional generative models like Variational Autoencoders (VAEs) and flow models, diffusion models operate on a fundamentally different principle. They leverage a Markov chain of diffusion steps to gradually add random noise to the data, and then learn to reverse this process to generate desired data samples from pure noise. This approach not only provides a novel mechanism for data generation but also ensures high-dimensional latent variables, making the model's output closely resemble the original data.

The Diffusion Process

The core idea behind diffusion models is the concept of a diffusion process. This process involves a series of steps in which noise is incrementally added to the data. This gradual noise addition can be thought of as a forward diffusion process, which eventually transforms the data into a completely noisy state. Mathematically, this can be described as a Markov chain, where each state in the chain is a noisier version of the previous one.

Markov Chain and Noise Addition

In a diffusion model, the Markov chain is defined such that at each step, a small amount of Gaussian noise is added to the data. This can be represented as:

$$x_{t+1} = x_t + \sqrt{\beta_t} \cdot \epsilon$$

where x_t is the data at step t, β_t is a noise coefficient, and ϵ is Gaussian noise. Over a large number of steps, this process transforms the original data into pure noise.

Learning the Reverse Process

The innovative aspect of diffusion models is their ability to learn the reverse diffusion process. The goal is to train a neural network to reverse the noise addition steps, effectively denoising the data step-by-step to reconstruct the original data from noise. This reverse process is also a Markov chain, but it involves subtracting the learned noise at each step:

$$x_{t-1} = x_t - \sqrt{\beta_t} \cdot \epsilon_\theta(x_t, t)$$

where ϵ_θ is the learned noise predictor, typically parameterized by a neural network.

High Dimensionality of Latent Variables

Unlike VAEs or flow models, where the latent space is often of lower dimensionality than the data space, diffusion models maintain a high-dimensional latent space that is equal to the original data space. This characteristic is crucial as it ensures that the generative process does not lose information and can produce highly detailed and accurate samples.

Training Diffusion Models

Training diffusion models involves two main objectives:

1. **Forward Process**: Simulate the forward diffusion process to create noisy data at various steps.
2. **Reverse Process**: Train the neural network to predict the noise added at each step of the forward process, thus learning to reverse the diffusion.

The loss function used during training typically measures the discrepancy between the predicted noise and the actual noise added during the forward process. By minimizing this loss, the model learns an accurate denoising function.

2 Prepossessing of Datasets

In this article we use two dataset:Mnist and persian digits and words dataset.

```python
def get_data_loaders(batch_size):

    # we normalize between [-1, 1]
    transform = Compose([
        ToTensor(),
        Lambda(lambda x: (x - 0.5) * 2)]
    )

    train_dataset = MNIST(root='.', train=True, transform=transform, download=True)
    test_dataset = MNIST(root='.', train=False, transform=transform, download=True)

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader
```

Figure 1: dataloaders and transformations

then we can simply show samples of images first from mnist datset.

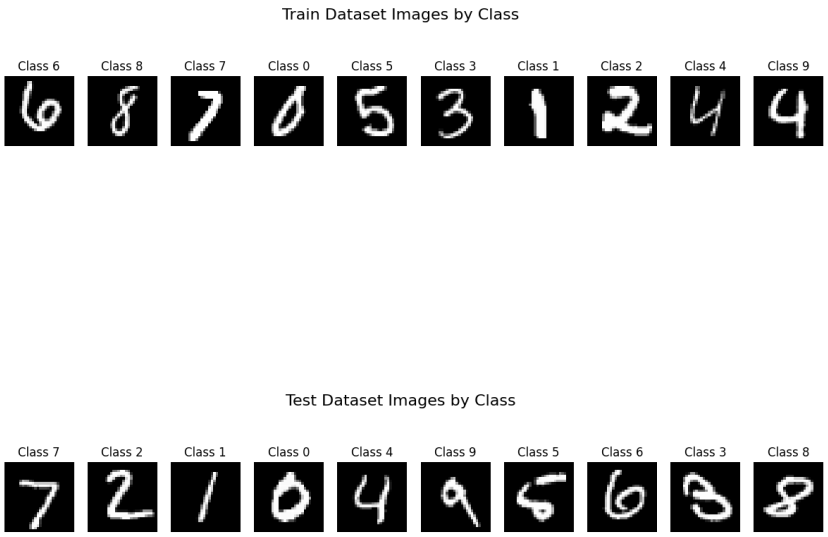


Figure 2: Samples of trainset and testset for the first datset

for the second dataset we need to prepare it and also creating a custom dataset for torch data-loaders
.

```python
file_id = '1bK1gYdxK92jXtDAcuUvITx6pa4CnyPAw'
output_path = '/content/dataset.zip'
extracted_path = '/content/dataset/extracted'
gdown.download(f'https://drive.google.com/uc?export=download&id={file_id}', output_path, quiet=False)

with ZipFile(output_path, 'r') as zip_ref:
    zip_ref.extractall(extracted_path)

# we remove the zip file after extraction
os.remove(output_path)
```

Figure 3: importing second dataset

```python
class PersianDigitDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_paths = []

        for root, _, files in os.walk(root_dir):
            for file in files:
                if file.endswith(".png"):
                    self.image_paths.append(os.path.join(root, file))
    def __len__(self):
        return len(self.image_paths)
    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert("L")  # we convert image to grayscale
        if self.transform:
            image = self.transform(image)
        # we extract label from file path
        label = int(os.path.basename(img_path).split('.')[0])

        return image, label

def get_data_loaders(batch_size):

    transform = Compose([
        Resize((64, 64)),  # we resize the image to 64x64
        ToTensor(),
        Lambda(lambda x: (x - 0.5) * 2)  # Normalize to [-1, 1]
    ])

    dataset = PersianDigitDataset(root_dir='/content/dataset/extracted', transform=transform)
    train_ratio = 0.95
    test_ratio = 0.05
    train_length = int(train_ratio * len(dataset))
    test_length = len(dataset) - train_length

    train_dataset, test_dataset = random_split(dataset, [train_length, test_length])
    batch_size = 128
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    print(f"Training dataset size: {len(train_dataset)}")
    print(f"Testing dataset size: {len(test_dataset)}")
    for images, labels in test_loader:
        print(images.shape, labels.shape)
        break

    for images, labels in test_loader:
        print(images.shape, labels.shape)
        break

    return train_loader, test_loader
```

Figure 4: creating custom dataset and shuffling

now we can see some samples of the second dataset

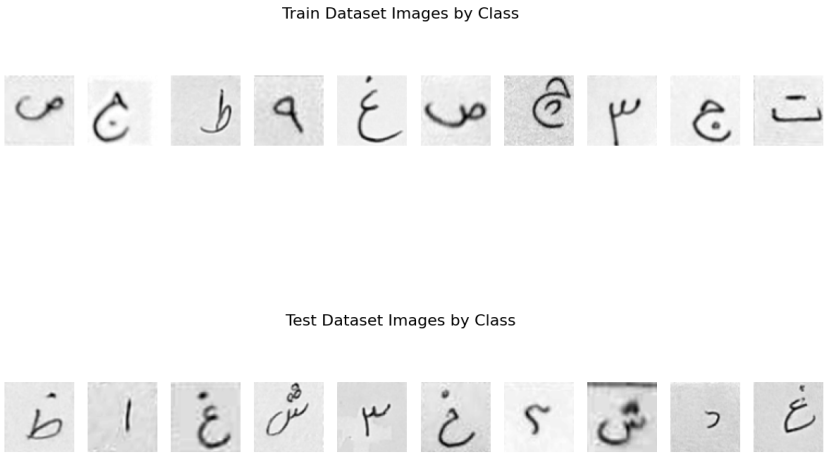


Figure 5: Samples of trainset and testset for the second dataset

3 Forward Process

Mathematical Proof for Diffusion Model Components

In the forward process, Gaussian noise is added to the data over several timesteps. In our case, the data consists of images in our dataset. Initially, we start with a clean image, and at each subsequent step, we add a small amount of Gaussian noise to it. This gradual addition of noise continues until the final step, where the image becomes completely corrupted by Gaussian noise with a mean of 0 and variance of 1.

We can name the noisy images as latent variable and the pure image as observation. in this case , each noisy step only depends upon the previous steps, hence it can make markov chain.

The forward diffusion process is defined as:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}),$$

where

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t|\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}).$$

Transition Distribution $q_\phi(\mathbf{x}_t|\mathbf{x}_{t-1})$

In a denoising diffusion probabilistic model (DDPM), the transition distribution $q_\phi(\mathbf{x}_t|\mathbf{x}_{t-1})$ describes how the data transitions from one timestep to the next in the forward diffusion process. This transition adds a small amount of Gaussian noise to the data at each step, ensuring that the data becomes progressively noisier.

The transition distribution is defined as follows:

$$q_\phi(\mathbf{x}_t|\mathbf{x}_{t-1}) \stackrel{def}{=} \mathcal{N}(\mathbf{x}_t|\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$$

where:

- $\mathcal{N}$ denotes the normal (Gaussian) distribution.
- $\mathbf{x}_t$ is the data at timestep t.
- $\mathbf{x}_{t-1}$ is the data at the previous timestep $t-1$.
- $\sqrt{1-\beta_t}$ is the scaling factor applied to $\mathbf{x}_{t-1}$.
- $\beta_t\mathbf{I}$ is the variance, where $\mathbf{I}$ is the identity matrix.

Detailed Explanation

Transition Distribution

$$q_\phi(\mathbf{x}_t|\mathbf{x}_{t-1})$$

This represents the probability distribution of $\mathbf{x}_t$ given $\mathbf{x}_{t-1}$.

Normal Distribution

$$\mathcal{N}(\mathbf{x}_t|\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$$

Mean

$$\sqrt{1-\beta_t}\mathbf{x}_{t-1}$$

The mean is scaled by $\sqrt{1-\beta_t}$ to control the contribution of $\mathbf{x}_{t-1}$.

Variance

$$\beta_t \mathbf{I}$$

The variance term ensures that the noise added at each step is controlled and does not cause the process to become unstable.

Scaling Factor

$$\sqrt{1-\beta_t}$$

The choice of the scaling factor $\sqrt{1-\beta_t}$ is crucial for maintaining the variance magnitude. This ensures that the variance does not explode or vanish after many iterations, allowing the model to add noise in a controlled manner.

By understanding these equations and their components, we can see how the forward process in DDPMs gradually corrupts the data with controlled Gaussian noise, making it possible to later train a model to reverse this process and denoise the data.

The noisy image at any timestep t can be computed directly using the following formulation:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

where:

- $\mathbf{x}_t$ is the noisy image at timestep t
- $\mathbf{x}_0$ is the original image
- $\bar{\alpha}_t$ is the cumulative product of α_t up to timestep t
- ϵ is Gaussian noise

Implementing Beta Schedule

The DDPM (Denoising Diffusion Probabilistic Models) module is a high-level model constructed using a given `network` parameter. The module is parameterized by:

- `n_steps`: Number of diffusion steps T
- `min_beta`: Value of the first β_t (β_1)
- `max_beta`: Value of the last β_t (β_T)
- `device`: Device on which the model is run
- `image_chw`: Tuple containing the dimensionality of images

The forward process of DDPMs benefits from a property where we can directly skip to any step t using the coefficients $\bar{\alpha}$. The backward process relies on the network to perform the denoising steps.

Implementing beta is so simple, we just have to take equal step from β_{start} to β_{end}, which is simple to implementation by linspace functions.

Beta Schedule

The beta values (β_t) are defined linearly between β_{start} and β_{end}. This can be represented mathematically as:

$$\beta_t = \beta_{start} + \frac{t}{T}(\beta_{end} - \beta_{start})$$

where t is the timestep and T is the total number of timesteps.

```
    def get_linear_beta_schedule(self, beta_start, beta_end, timesteps):

    return torch.linspace(beta_start, beta_end, timesteps)

```

Listing 1: Python code for get_linear_beta_schedule method

Alphas

now we computed all betas (simple linespace function). we have to compute alpha and alpha-bars. what are they?

we use

$$\alpha_t = 1 - \beta_t \tag{1}$$

instead of beta.

for simplicity we use Reparameterization Trick like what we hade in VAE's (previously for resolving the derivative problem of sampling).

Given the transition probability, we know that:

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}) \tag{2}$$

$$\mathbf{x}_0 \sim p_0(\mathbf{x})$$

Now, instead of computing noises sequentially from the previous steps, we can compute the noisy image at a specific time step directly from the pure image. Here, we introduce the closed-form implementation and use the parameter alpha-bars for this aim.

Alpha Bars

The conditional distribution $q_\phi(\mathbf{x}_t|\mathbf{x}_0)$ is given by:

$$q_\phi(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t|\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{3}$$

On the other hand, x_t can be written as:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_0 \tag{4}$$

where $\bar{\alpha}_t$ is defined as:

$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i \tag{5}$$

This is the cumulative product of alphas up to timestep t.

These alpha bars are crucial because they allow us to express the variance of the forward process at any timestep. In the PyTorch library, we can implement it using the 'torch.cumprod' function. based on this formula we need only to compute alpha-bars once.

Here is all we require about the forward path:

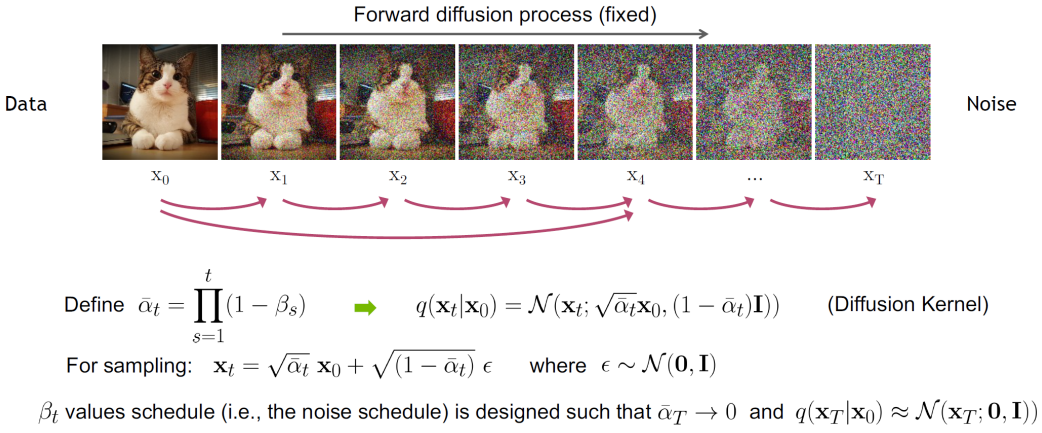


Figure 6: All things about the forward path

Example

Consider our initial discussion: we generate a number of time steps between beta-start and beta-end. We take small steps because we aim to undo these steps in the backward path and recover the image from pure noise. The original paper suggests setting beta-start to 0.0001 and beta-end to 0.02, generating 1000 time steps in between. This was simply constructed using the `get_linear_beta_schedule` function.

Next, we apply the **reparametrization trick** in equations (1) and (2) to compute alpha and subsequently alpha-bar.

In equation (4), we observe that the first term approaches 0 at the last time state, while the second term approaches 1. This is because, in equation (5), the product of many small values tends towards zero. As we proceed, more small values are multiplied, causing the first term to approach zero and the second term to approach 1. If we consider the last time state as T:

$$q_\phi(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t|0, I) \tag{6}$$

Implementation in PyTorch

Here is the implementation of the `DiffusionModel` class in PyTorch:

At the first step we simply compute the get_linear_bet_schedule function

```
class DiffusionModel(nn.Module):
  def __init__(self, backward_process_model, beta_start, beta_end, timesteps=1000, device="cuda"):
    super(DiffusionModel, self).__init__()

    self.beta_start = beta_start
    self.beta_end = beta_end
    self.timesteps = timesteps
    self.device = device
    self.image_chw = (1, 28, 28)
    self.backward_process_model = backward_process_model

    self.betas = self.get_linear_beta_schedule(beta_start, beta_end, timesteps).to(device)
    # we define alphas variable based on q(xt|x0) formula
    self.alphas = 1.0 - self.betas.to(device)

    # we define alpha_bars variable based on q(xt|x0) formula
    self.alpha_bars = torch.cumprod(self.alphas, dim=0).to(device)




  def get_linear_beta_schedule(self, beta_start = 0.0001, beta_end = 0.02, timesteps = 1000):
    return torch.linspace(beta_start, beta_end, timesteps)

  def add_noise(self, x_0, timestep, visualization = False):
    #As we said time step is including 128 random intiger number between 0 and 1000
    # Make input image more noisy (we can directly skip to the desired step)
    # this function get and specific time step and then it computes the corresponding noise and
    n, c, h, w = x_0.shape
    a_bar = self.alpha_bars[timestep]
    eta = torch.randn(n, c, h, w).to(self.device)
    if visualization ==False:
    noisy = a_bar.sqrt().reshape(n, 1, 1, 1) * x_0 + (1 - a_bar).sqrt().reshape(n, 1, 1, 1) * eta
    else:
    noisy = torch.sqrt(a_bar) * x_0 + torch.sqrt(1 - a_bar) * eta
    return noisy, eta

  def backward(self, x, t):
    return self.backward_process_model(x, t)


    def forward(self, x): # size of x0 is almost torch.Size([128, 1, 28, 28]) exept the last epoch  which is torch.Size([96, 1, 28, 28])
    n = len(x) # n is alomost equal to batch size exept the last (128)
```

```
    steps = torch.randint(0, self.timesteps, (n,)).to(self.device) # creating random
     integer number between 0 and the last time state. creating almost 128 number
    noisy_imgs, noise = self.add_noise(x, steps) # going forward to make noisy
    eta_theta = self.backward_process_model(noisy_imgs, steps.reshape(n, -1))  # we
     reshape steps to torch.Size([128, 1]) so steps.reshape(n, -1) size is torch.Size
     ([128, 1])
    return noisy_imgs, noise, eta_theta
```

Listing 2: Python code for DiffusionModel class

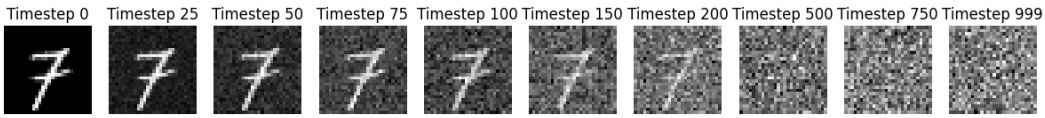


Figure 7: visualization of forward path correspond to each time step for the first dataset

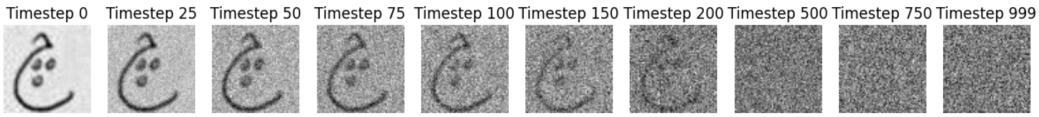
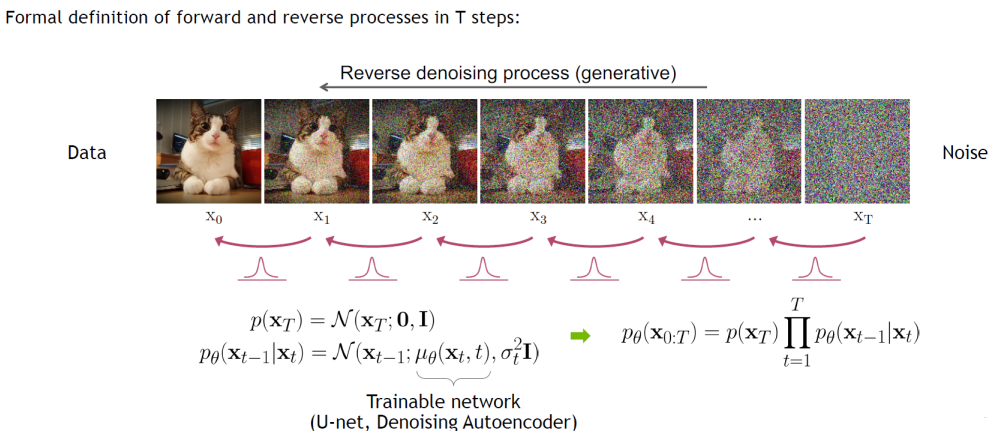


Figure 8: visualization of forward path correspond to each time step for the second dataset

4 Backward Process

For the backward process, the model operates as a Gaussian distribution. The goal is to predict the distribution mean and standard deviation given the noisy image and the time step. In the initial paper on DDPMs, the covariance matrix is kept fixed, so the focus is on predicting the mean of the Gaussian distribution based on the noisy image and the current time step.



4.1 Backward Process Explanation

In the backward process, the objective is to revert to a less noisy image x at timestep $t-1$ using a Gaussian distribution whose mean is predicted by the model. The optimal mean value to be predicted is a function of known terms:

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) \tag{7}$$

Where μ_θ is the predicted mean, x_t is the noisy image at time step t, α_t and β_t are constants derived from the forward process, $\bar{\alpha}_t$ is the cumulative product of α_t up to time step t, and ϵ_θ is the noise predicted by the model.

4.2 Simplifying the Model

Given the more noisy image at step t, the optimal mean value can be simplified by predicting the noise ϵ with a function of the noisy image and the time-step. The model predicts the added noise, which is then used to recover a less noisy image using the time step information.

Loss Function

The loss function is a scaled version of the Mean-Square Error (MSE) between the real noise added to the images and the noise predicted by the model:

$$L(\theta) = \mathbb{E}_{t,x_0,\epsilon}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|^2\right] \tag{8}$$

Here, θ represents the model parameters, t is the time step, x_0 is the original image, x_t is the noisy image at time step t, and ϵ is the actual noise added.

Remember that the only term of the loss function that we really care about is:

$$\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2 \tag{9}$$

where ϵ is some random noise and ϵ_θ is the model's prediction of the noise. Now, ϵ_θ is a function of both x and t. We don't want to have a distinct model for each denoising step (which would require

thousands of independent models). Instead, we use a single model that takes as input the image x and a scalar value indicating the timestep t.

To achieve this, we use a sinusoidal embedding function, `sinusoidal_embedding`, that maps each time-step to a `time_emb_dim` dimension. These time embeddings are further processed with time-embedding MLPs (function `_make_te`) and added to tensors through the network in a channel-wise manner.

4.3 Training and Sampling Algorithms

Once the model is trained using the above loss function (Algorithm 1), it can be employed to generate new images starting from Gaussian noise (Algorithm 2).

finally the backward process in DDPMs involves predicting the mean of a Gaussian distribution to revert a noisy image to its less noisy state. By simplifying the model to predict the added noise, we can efficiently train the model and use it to generate new images. The use of a scaled MSE loss function ensures that the model accurately learns to predict the noise, facilitating effective denoising and sampling processes.

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

at the next step we will explore the concept of positional encoding, with a particular focus on sinusoidal positional encoding, as introduced in the seminal Attention is All You Need paper.

4.4 Understanding Sinusoidal Positional Encoding

Sinusoidal positional encoding is a technique used to integrate positional information into the embeddings at the beginning of encoder and decoder layers in transformer models. This method employs sinusoidal functions at varying frequencies to encode the position of each element within a sequence. Sine and cosine functions are applied to even and odd positions, respectively, ensuring that each position can be uniquely identified by the model.

The key advantage of using sinusoidal functions is their periodic nature, which makes it straightforward for the model to infer positional information. The encoding is derived from a geometric progression of frequencies, allowing the model to maintain relative positional information easily. Thus, for any fixed offset k, the positional encoding at position $pos + k$ can be expressed as a linear function of the encoding at position pos.

Formula for Positional Encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{10}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{11}$$

4.4.1 Why we often use 10000 in Sinusoidal Positional Encoding

The use of the value 10000 in the sinusoidal positional encoding formula is crucial for defining the wavelengths of the sinusoidal functions, which range from 2π to $10000 \cdot 2\pi$. This choice ensures that the wavelengths grow exponentially with the position index, providing a unique representation for each position in long sequences.

The selection of 10000 is designed to balance the ease of learning for the model with the need for a distinct representation of each position. By scaling the frequencies appropriately, the model can efficiently encode positional information across varying sequence lengths.

4.4.2 Detailed Mathematical Explanation

The denominator in the sinusoidal positional encoding formula is given by

$$10000^{\frac{2i}{\text{dim}}} \tag{12}$$

where $2i$ denotes the specific dimension in the embedding, dim represents the overall embedding dimension, and 10000 is the constant used for scaling. This setup forms an arithmetic progression for the power term $\frac{2i}{\text{dim}}$, which is then transformed into a geometric progression through exponentiation with base 10000. This geometric progression, when combined with positional indices, generates a sequence of values used as inputs for the sine and cosine functions.

4.4.3 Characteristics of Sinusoidal Positional Encoding

- **Normalized Value Range:** The sine and cosine functions ensure that the encoded values stay within the range $[-1, 1]$, facilitating easier learning by the model.
- **Unique Encodings:** Each position in the sequence receives a unique encoding, allowing the model to differentiate between positions effectively and to handle sequences of virtually unlimited length.
- **Preservation of Relative Distances:** Sinusoidal positional encoding maintains consistent relative distances between positions, such that the difference between the encodings of consecutive positions remains constant.

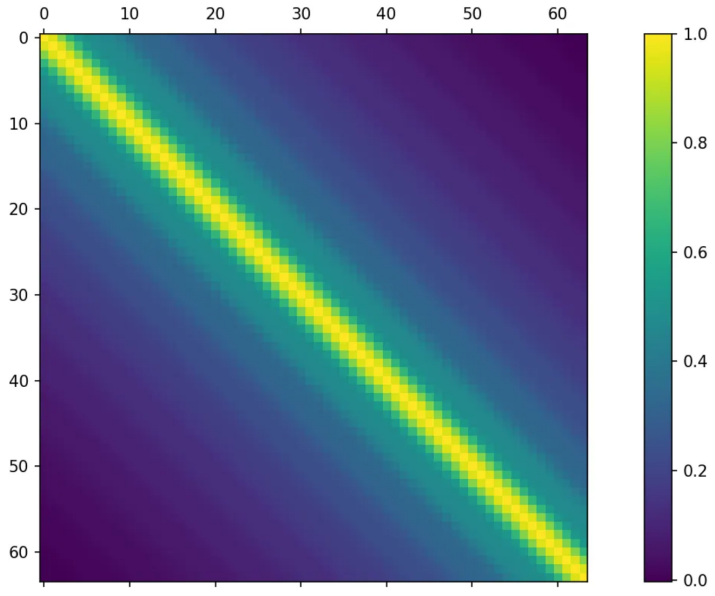


Figure 9: relative distance matrix for 64 sequence length positional encoding

Positional encoding (which also is a fundamental component of transformer models) enabling models to comprehend the sequential order of tokens (here is the time steps). Sinusoidal positional encoding, with its thoughtful design involving the value 10000, offers a robust mechanism for embedding positional information. This method strikes a balance between the simplicity of learning and the distinctiveness of positional representations.

we can easily implement it in two way:

```python
def sinusoidal_embedding(n, d):
    embedding = torch.zeros(n, d)
    wk = torch.tensor([1 / 10_000 ** (2 * j / d) for j in range(d)])
    wk = wk.reshape((1, d))
    t = torch.arange(n).reshape((n, 1))
    embedding[:,::2] = torch.sin(t * wk[:,::2])
    embedding[:,1::2] = torch.cos(t * wk[:,::2])

    return embedding

# second way of implementing

def sinusoidal_embedding(n, d):
    embedding = torch.zeros(n, d)
    wk = torch.tensor([1 / 10_000 ** (2 * j / d) for j in range(d)])
    wk = wk.reshape((1, d))
    t = torch.arange(n).reshape((n, 1))
    embedding = torch.sin(t * wk)
    return embedding
```

Figure 10: Implementation of sinusoidal positional embedding

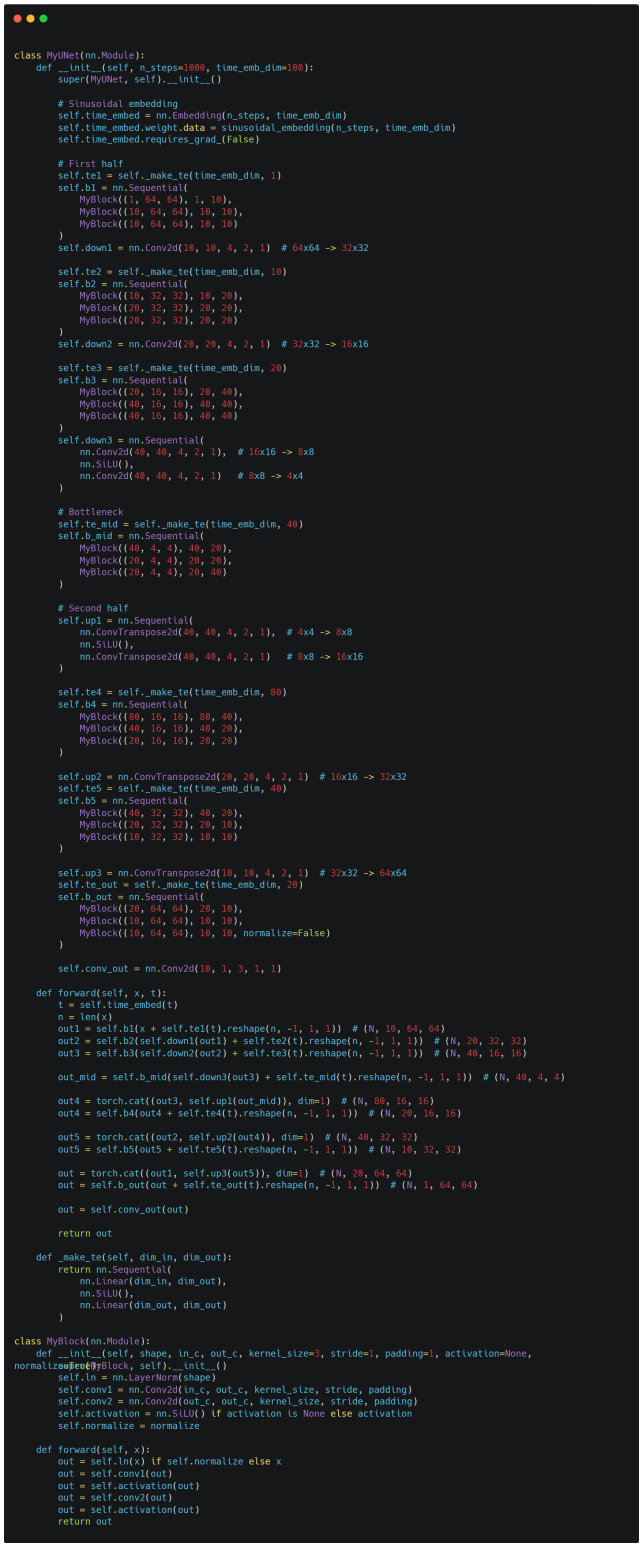
After implementing the sinusoidal embedding we have to implement the modified UNET. for this aim we add some modification to the original UNET architecture.

the first thing is adding time embedding because we want our model be shared through time!

original UNET is consisted double conv layer and three down samples and three up sample blocks. for getting better result we almost using three double block as MyBlock layer which uses as a feature extractor. Then we need to add positional embedding to each feature map. therefore, instead of giving pure feature map to each block we add corresponding time embedding to each feature map. For matching the channel size and positional embedding we use two linear layer. these two layers are in _make_te function which project the time embedding dimension to the channel size correspond to each stage in Unet.

so after down sample and reconstruction, the network can learn the added noise.

4.5 Implementing model



```
class MyUNet(nn.Module):
    def __init__(self, n_steps=1000, time_emb_dim=100):
        super(MyUNet, self).__init__()

        # Sinusoidal embedding
        self.time_embed = nn.Embedding(n_steps, time_emb_dim)
        self.time_embed.weight.data = sinusoidal_embedding(n_steps, time_emb_dim)
        self.time_embed.requires_grad_(False)

        # First half
        self.te1 = self._make_te(time_emb_dim, 1)
        self.b1 = nn.Sequential(
            MyBlock((1, 64, 64), 1, 10),
            MyBlock((10, 64, 64), 10, 10),
            MyBlock((10, 64, 64), 10, 10)
        )
        self.down1 = nn.Conv2d(10, 10, 4, 2, 1)  # 64x64 -> 32x32

        self.te2 = self._make_te(time_emb_dim, 10)
        self.b2 = nn.Sequential(
            MyBlock((10, 32, 32), 10, 20),
            MyBlock((20, 32, 32), 20, 20),
            MyBlock((20, 32, 32), 20, 20)
        )
        self.down2 = nn.Conv2d(20, 20, 4, 2, 1)  # 32x32 -> 16x16

        self.te3 = self._make_te(time_emb_dim, 20)
        self.b3 = nn.Sequential(
            MyBlock((20, 16, 16), 20, 40),
            MyBlock((40, 16, 16), 40, 40),
            MyBlock((40, 16, 16), 40, 40)
        )
        self.down3 = nn.Sequential(
            nn.Conv2d(40, 40, 4, 2, 1),  # 16x16 -> 8x8
            nn.SiLU(),
            nn.Conv2d(40, 40, 4, 2, 1)   # 8x8 -> 4x4
        )

        # Bottleneck
        self.te_mid = self._make_te(time_emb_dim, 40)
        self.b_mid = nn.Sequential(
            MyBlock((40, 4, 4), 40, 20),
            MyBlock((20, 4, 4), 20, 20),
            MyBlock((20, 4, 4), 20, 40)
        )

        # Second half
        self.up1 = nn.Sequential(
            nn.ConvTranspose2d(40, 40, 4, 2, 1),  # 4x4 -> 8x8
            nn.SiLU(),
            nn.ConvTranspose2d(40, 40, 4, 2, 1)   # 8x8 -> 16x16
        )

        self.te4 = self._make_te(time_emb_dim, 80)
        self.b4 = nn.Sequential(
            MyBlock((80, 16, 16), 80, 40),
            MyBlock((40, 16, 16), 40, 20),
            MyBlock((20, 16, 16), 20, 20)
        )

        self.up2 = nn.ConvTranspose2d(20, 20, 4, 2, 1)  # 16x16 -> 32x32
        self.te5 = self._make_te(time_emb_dim, 40)
        self.b5 = nn.Sequential(
            MyBlock((40, 32, 32), 40, 20),
            MyBlock((20, 32, 32), 20, 10),
            MyBlock((10, 32, 32), 10, 10)
        )

        self.up3 = nn.ConvTranspose2d(10, 10, 4, 2, 1)  # 32x32 -> 64x64
        self.te_out = self._make_te(time_emb_dim, 20)
        self.b_out = nn.Sequential(
            MyBlock((20, 64, 64), 20, 10),
            MyBlock((10, 64, 64), 10, 10),
            MyBlock((10, 64, 64), 10, 10, normalize=False)
        )

        self.conv_out = nn.Conv2d(10, 1, 3, 1, 1)

    def forward(self, x, t):
        t = self.time_embed(t)
        n = len(x)
        out1 = self.b1(x + self.te1(t).reshape(n, -1, 1, 1))  # (N, 10, 64, 64)
        out2 = self.b2(self.down1(out1) + self.te2(t).reshape(n, -1, 1, 1))  # (N, 20, 32, 32)
        out3 = self.b3(self.down2(out2) + self.te3(t).reshape(n, -1, 1, 1))  # (N, 40, 16, 16)

        out_mid = self.b_mid(self.down3(out3) + self.te_mid(t).reshape(n, -1, 1, 1))  # (N, 40, 4, 4)

        out4 = torch.cat((out3, self.up1(out_mid)), dim=1)  # (N, 80, 16, 16)
        out4 = self.b4(out4 + self.te4(t).reshape(n, -1, 1, 1))  # (N, 20, 16, 16)

        out5 = torch.cat((out2, self.up2(out4)), dim=1)  # (N, 40, 32, 32)
        out5 = self.b5(out5 + self.te5(t).reshape(n, -1, 1, 1))  # (N, 10, 32, 32)

        out = torch.cat((out1, self.up3(out5)), dim=1)  # (N, 20, 64, 64)
        out = self.b_out(out + self.te_out(t).reshape(n, -1, 1, 1))  # (N, 1, 64, 64)

        out = self.conv_out(out)

        return out

    def _make_te(self, dim_in, dim_out):
        return nn.Sequential(
            nn.Linear(dim_in, dim_out),
            nn.SiLU(),
            nn.Linear(dim_out, dim_out)
        )

class MyBlock(nn.Module):
    def __init__(self, shape, in_c, out_c, kernel_size=3, stride=1, padding=1, activation=None,
normalize=True):
        super(MyBlock, self).__init__()
        self.ln = nn.LayerNorm(shape)
        self.conv1 = nn.Conv2d(in_c, out_c, kernel_size, stride, padding)
        self.conv2 = nn.Conv2d(out_c, out_c, kernel_size, stride, padding)
        self.activation = nn.SiLU() if activation is None else activation
        self.normalize = normalize

    def forward(self, x):
        out = self.ln(x) if self.normalize else x
        out = self.conv1(out)
        out = self.activation(out)
        out = self.conv2(out)
        out = self.activation(out)
        return out
```

Figure 11: implementing modified Unet model

4.6 training loop

Here is our training loop for learning the model

```python
def get_loss(noise, noise_pred):

    return torch.mean((noise - noise_pred) ** 2)


def training_loop(ddpm, train_loader, val_loader, n_epochs, optim, device, display=False, store_path="ddpm_model.pt"):

    best_loss = float("inf")
    n_steps = ddpm.timesteps
    scheduler = StepLR(optim, step_size=10, gamma=0.5)
    train_losses = []
    val_losses = []

    for epoch in tqdm(range(n_epochs), desc="Training progress", colour="#00ff00"):
        epoch_train_loss = 0.0
        epoch_val_loss = 0.0

        ddpm.train()
        for batch in tqdm(train_loader, leave=False, desc=f"Epoch {epoch + 1}/{n_epochs} - Training", colour="#005500"):
            x0 = batch[0].to(device)
            a, eta, eta_theta = ddpm(x0)
            loss = get_loss(eta_theta, eta)
            optim.zero_grad()
            loss.backward()
            optim.step()

            epoch_train_loss += loss.item() * len(x0) / len(train_loader.dataset)

        ddpm.eval()
        with torch.no_grad():
            for batch in tqdm(val_loader, leave=False, desc=f"Epoch {epoch + 1}/{n_epochs} - Validation", colour="#550000"):
                x0 = batch[0].to(device)
                a, eta, eta_theta = ddpm(x0)
                loss = get_loss(eta_theta, eta)

                epoch_val_loss += loss.item() * len(x0) / len(val_loader.dataset)

        scheduler.step()

        train_losses.append(epoch_train_loss)
        val_losses.append(epoch_val_loss)

        lr = scheduler.get_last_lr()[0]
        log_string = f"Epoch {epoch + 1}/{n_epochs} - Training Loss: {epoch_train_loss:.3f}, Validation Loss: {epoch_val_loss:.3f}, Learning Rate: {lr:.6f}"

        if best_loss > epoch_train_loss:
            best_loss = epoch_train_loss
            torch.save(ddpm.state_dict(), store_path)
            log_string += " --> saving best model"
        print(log_string)

    plt.figure()
    plt.plot(range(1, n_epochs + 1), train_losses, marker='.', linestyle='-', color='b', label='Training Loss')
    plt.plot(range(1, n_epochs + 1), val_losses, marker='.', linestyle='-', color='r', label='Validation Loss')
    plt.title("Loss per Epoch")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid(True)
    plt.show()
```

Figure 12: training loop

We use learning rate =0.001 using StepLR for each 100 epoch with gamma = 0.5 and Adam optimizer.

4.7 Implementing of evaluation function and sampling function

for evaluation we only need to check the MSE metric for the predicted noise and added noise.

```python
def evaluate(diffusion_model, test_loader, device="cuda"):
    diffusion_model.eval()
    total_loss = 0.0
    total_samples = 0

    with torch.no_grad():
        for data, target in test_loader:
            data = data.to(device)
            n = len(data)
            n_steps = torch.randint(0, diffusion_model.timesteps, (n,)).to(device)
            noisy_imgs, noise = diffusion_model.add_noise(data, n_steps)
            eta_theta = diffusion_model.backward_process_model(noisy_imgs, n_steps.reshape(n, -1))

            loss = get_loss(noise, eta_theta)
            total_loss += loss.item() * n
            total_samples += n

    average_loss = total_loss / total_samples
    return average_loss
```

Figure 13: Evaluate function

we can exactly implement the sampling function according to provided formula and algorithm.

```python
def sample(ddpm, n_samples=16, device=None, frames_per_gif=100, gif_name="sampling.gif", c=1, h=64, w=64):
    """Given a DDPM model, a number of samples to be generated and a device, returns some newly generated samples"""
    frame_idxs = np.linspace(0, ddpm.timesteps, frames_per_gif).astype(np.uint)
    frames = []

    with torch.no_grad():
        if device is None:
            device = ddpm.device

        x = torch.randn(n_samples, c, h, w).to(device)

        for idx, t in enumerate(list(range(ddpm.timesteps))[::-1]):
            time_tensor = (torch.ones(n_samples, 1) * t).to(device).long()
            eta_theta = ddpm.backward(x, time_tensor)

            alpha_t = ddpm.alphas[t]
            alpha_t_bar = ddpm.alpha_bars[t]


            x = (1 / alpha_t.sqrt()) * (x - (1 - alpha_t) / (1 - alpha_t_bar).sqrt() * eta_theta)

            if t > 0:
                z = torch.randn(n_samples, c, h, w).to(device)

                beta_t = ddpm.betas[t]
                sigma_t = beta_t.sqrt()
                x = x + sigma_t * z

            if idx in frame_idxs or t == 0:

                normalized = x.clone()
                for i in range(len(normalized)):
                    normalized[i] -= torch.min(normalized[i])
                    normalized[i] *= 255 / torch.max(normalized[i])


                frame = einops.rearrange(normalized, "(b1 b2) c h w -> (b1 h) (b2 w) c", b1=int(n_samples ** 0.5))
                frame = frame.cpu().numpy().astype(np.uint8)


                frames.append(frame)


    with imageio.get_writer(gif_name, mode="I") as writer:
        for idx, frame in enumerate(frames):
            rgb_frame = np.repeat(frame, 3, axis=2)
            writer.append_data(rgb_frame)

            if idx == len(frames) - 1:
                last_rgb_frame = np.repeat(frames[-1], 3, axis=2)
                for _ in range(frames_per_gif // 3):
                    writer.append_data(last_rgb_frame)
    return
```

Figure 14: sampling function

5 Results and Metrics

here we provides both results for both dataset Mnist and persian words.

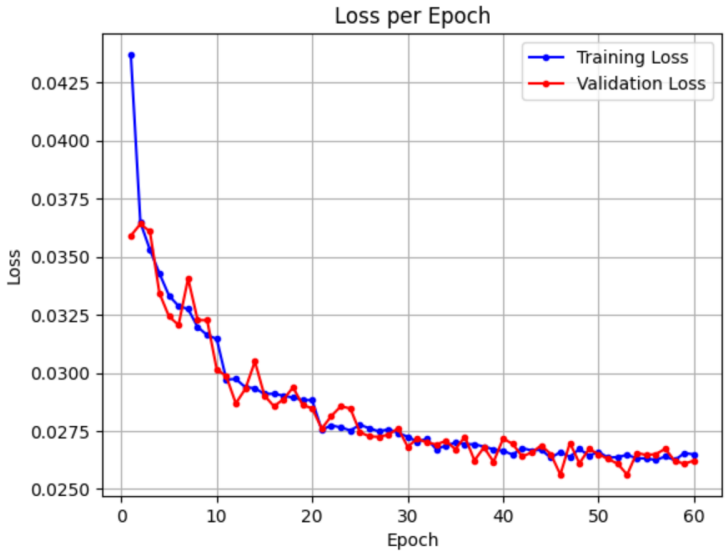


Figure 15: train and validation loss curves

Then we generate new sample from mnist datset as follows: we used the mentioned formula in original paper for generating samples.

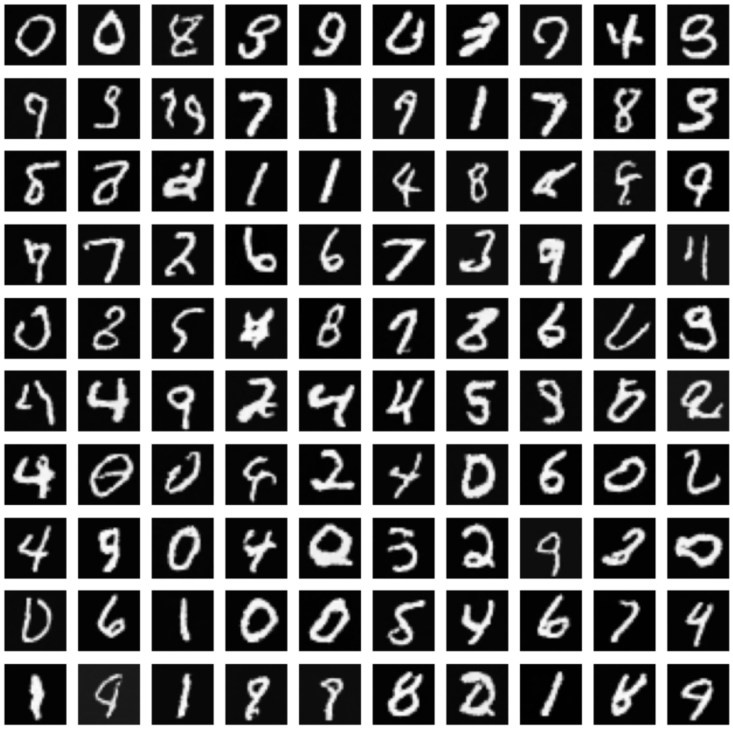


Figure 16: 100 sample generated form mnist dataset

For evaluation we can use the first and second images we can use the sampling equation for generating new noised image(we use alpha and beta in the forward process to complete this formula and using backward model for computing added noise or eta_theta).

And also after training for the second dataset we have:

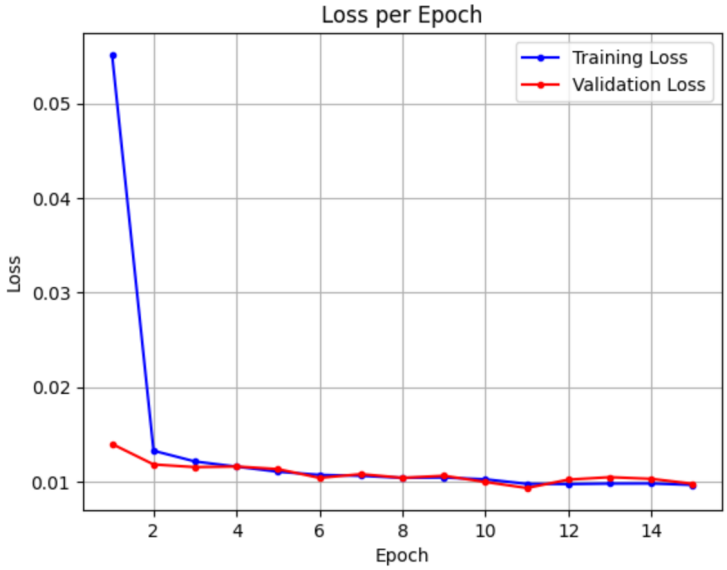


Figure 17: loss curves for validation and train sets during training

Figure 18: generated images for the second dataset

Figure 19: gif generated for the second dataset

This document outlines the implementation of a Denoising Diffusion Probabilistic Model (DDPM). The following Python script includes that all necessary files are present before proceeding with the implementation.

Required Files

The implementation requires the following files:

- `main.py`
- `Dataset.py`
- `utils.py`
- `diffusion_model.py`
- `unet.py`
- `requirement.txt`