Assignment 4: Diffusion Model

In this assignment, you will implement a diffusion model from scratch and train it on the MNIST dataset. Diffusion models are a class of generative models that learn to gradually denoise random noise to generate realistic images. This assignment will guide you through the core components and training process of diffusion models.

Useful links:

1. What are Diffusion Models?
2. Denoising Diffusion Probabilistic Models

Please:

- Fill out the code marked with `TODO` or `Your code here`. You are allowed to split functions or visualizations to different files for more flexibility as long as your output includes what we asked.
- Reuse or modify visualization code from Assignment 2 for creating necessary visualizations.
- Submit the notebook with all original outputs. If the output is included from another file, please include them into your folder.
- Answer questions at the end of the notebook. Write your answere in the notebook.

Please reserve enough time for this assignment given the potential amount of time for training.

In [ ]:
```
import torch
```

Part 1: Implementing the U-Net (30 pt)

In this part, you will implement a U-Net style model that serves as the backbone for the diffusion process. The model takes noisy images and their corresponding timesteps as input and predicts the noise that was added to the original images.

Please fill out the code in `diffusion.DiffusionModel` then run the following code for test. For the time embedding, you can only use one embedding layer and concatenate it with the feature. The attention layer is not enforced given the computation resource.

In [ ]:
```
from diffusion import DiffusionModel

def check_diffusion_model(model_class):
    """Verify that the DiffusionModel class is correctly implemented."""
    try:
```

```
        channels = 1
        image_size = 28
        noise_steps = 1000
        model = model_class(image_size=image_size, channles=channels)

        # Test forward pass with random inputs
        batch_size = 4
        x = torch.randn(batch_size, channels, image_size, image_size)
        t = torch.randint(0, noise_steps, (batch_size,))

        output = model(x, t)

        # Check output shape
        expected_shape = (batch_size, channels, image_size, image_size)
        assert output.shape == expected_shape, f"Expected output shape {expe

        print("DiffusionModel implementation is correct!")
        return True
    except Exception as e:
        print(f"DiffusionModel check failed: {str(e)}")
        return False

check_diffusion_model(DiffusionModel)
```

Part 2: Implementing the Diffusion Process (30 pt)

In this part, you will implement the core diffusion process, including the forward diffusion (adding noise) and the denoising process. This includes setting up the noise schedule and implementing functions for noise addition and sampling.

Please fill out the code in `diffusion.DiffusionProcess` then run the following code for test. Note that this test only tests the correctness of the output format. You need to be careful about the actual math.

In [ ]:
```
from diffusion import DiffusionProcess

def check_diffusion_process(diffusion_class):
    """Verify that the DiffusionProcess class is correctly implemented."""
    try:
        channels = 1
        image_size = 28
        noise_steps = 1000
        diffusion = diffusion_class(image_size=image_size, channels=channels

        # Test add_noise function
        batch_size = 4
        x = torch.randn(batch_size, channels, image_size, image_size)
        t = torch.randint(0, noise_steps, (batch_size,))

        noisy_x, noise = diffusion.add_noise(x, t)
        assert noisy_x.shape == x.shape, f"Expected noisy_x shape {x.shape},
        assert noise.shape == x.shape, f"Expected noise shape {x.shape}, got
```

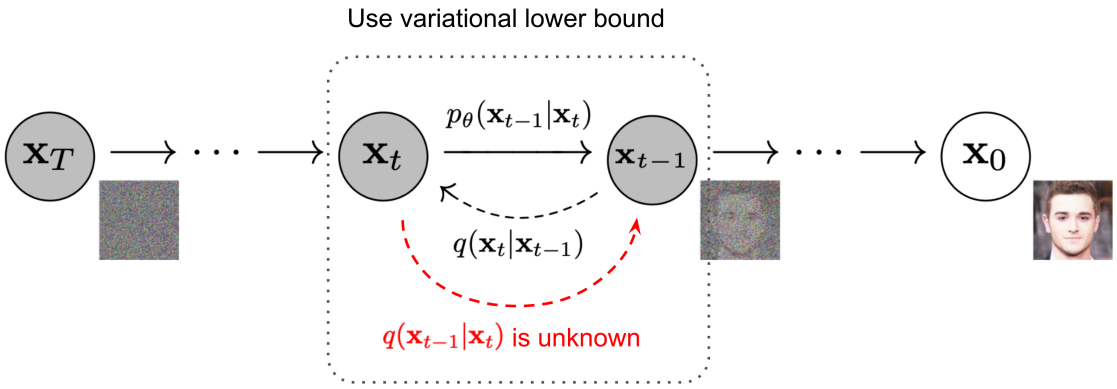
```
        # Test train_step function
        loss = diffusion.train_step(x)
        assert isinstance(loss, float), f"Expected loss to be a float, got {

        print("DiffusionProcess implementation is correct!")
        return True
    except Exception as e:
        print(f"DiffusionProcess check failed: {str(e)}")
        return False

check_diffusion_process(DiffusionProcess)
```

Part 3: Training and Sampling (20 points)

In this part, you will implement the training loop for the diffusion model and the functions for generating and visualizing samples. Please try to follow the assignment you have written and use the `DiffusionModel` and `DiffusionProcess` above for write your training function. You should write your training code in a standalone python file.

Please include the training curves and the sampled results below. You can reuse the visualization code we provided in the GAN assignment.

You can include an image like:



Part 4: Analysis and Visualization (20 points)

Answer the question with your analysis. Most of the questions are open-ended. We are looking for yourown observasion from the experiments you did.

1. How does the choice of noise schedule (beta values) affect the training stability and sample quality? Try at least one alternative to the linear schedule (e.g., cosine or quadratic) and compare the results.

[Answer]:

2. Based on your observations, at which timesteps (early, middle, or late in the diffusion process) does the model seem to struggle the most with accurately predicting the noise (looking into loss)? Why do you think this occurs?

[Answer]:

3. Perform interpolation between two noise vectors and analyze the resulting generated images. Is the transition smooth? What does this tell you about the model's learned latent space?

[Answer]:

4. Recall Assignment 2, we implemented GAN. compare your diffusion model with GANs in terms of:

- Training stability
- Sample quality
- Diversity of samples
- Computational requirements
- Anything else you find interesting

[Answer]:

Extra Credit: Diffusion Model on CIFAR 10 (20 pt)

In this extra credit assignment, you'll extend your Diffusion implementation to handle the more complex CIFAR-10 dataset.

You should design your own network architectures, considering factors like the increased complexity of RGB images, memory efficiency, and training stability. The basic code structure from the MNIST implementation can serve as a reference, but you'll need to modify the network dimensions and potentially add more capacity to handle the increased complexity.

Your submission should include complete implementation code (in a standalone file), training curves, generated imagesamples (inclue below), and a brief analysis (1-2 paragraphs) comparing your CIFAR-10 results with your MNIST implementation

Extra Credit: DDIM Sampler Implementation (20 points)

Implement the Denoising Diffusion Implicit Models (DDIM) sampling method, which allows for faster sampling based on paper Denoising Diffusion Implicit Models.

1. Implement the DDIM sampling algorithm based on the paper.

2. Compare DDIM sampling with the standard DDPM sampling in terms of:

- Sampling speed
- Sample quality
- Number of required steps

3. Experiment with different numbers of DDIM steps and analyze the tradeoff between speed and quality.

Your submission should include complete implementation code (can be in another python file), generated imagesamples (using the same MNIST model you presented above), and a brief analysis (1-2 paragraphs) comparing DDIM and DDPM.