Report for exercise 6 from group F

Tasks addressed:	5
Authors:	Qingyu Wang (03792094) Augustin Gaspard Camille Curinier (03784531) Yuxuan Wang (03767260) Shihong Zhang (03764740) Mengshuo Li (03792428)
Last compiled:	2024–07–04

The work on tasks was divided in the following way:

Qingyu Wang (03792094)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Augustin Gaspard Camille Curinier (03784531)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Yuxuan Wang (03767260)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Shihong Zhang (03764740)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Mengshuo Li (03792428)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Report on task 1, Theoretical Analysis of the forward diffusion process in DDPM.

1 Theoretical Analysis of the forward diffusion process in DDPM.

1.1 Introduction

Denoising Diffusion Probabilistic Models (DDPM) are advanced generative models that create high-quality data samples through a reverse noise diffusion process. A key step in these models is the forward diffusion process, which gradually transforms data samples into noise, laying the foundation for the denoising reverse process. This report aims to explain in detail the principles, mathematical derivation, and application of the forward diffusion process in DDPM.

1.2 Overview of the Forward Diffusion Process

The forward diffusion process progressively adds Gaussian noise to an initial data sample until it approximates pure noise. Given an initial data sample x_0, each step of this process can be viewed as a Markov chain, where the noise added at each step depends only on the previous step's result. This process is implemented through a series of conditional Gaussian distributions, ultimately "diffusing" the data sample into noise.

1.3 Mathematical Definition

The forward diffusion process for each step is defined by the following conditional distribution:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)I)$$

where α_t is a parameter controlling the noise level at each step, typically within (0, 1). To intuitively represent the noise level, we define β_t as:

$$\beta_t = 1 - \alpha_t$$

Thus, the conditional distribution can be rewritten as:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

1.4 Mathematical Derivation

Deriving the Distribution from x_0 to x_t

To derive the distribution from the initial sample x_0 to any timestep t, $q(x_t|x_0)$, we use the Markov property, expressing the process as:

$$q(x_t|x_0) = \int q(x_t|x_{t-1})q(x_{t-1}|x_{t-2}) \cdots q(x_1|x_0)\, dx_{t-1} \cdots dx_1$$

Since each step is a linear transformation of Gaussian distributions, the result is also a Gaussian distribution. Using mathematical induction, we obtain:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1 - \bar{\alpha}_t)I)$$

where:

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$$

We define the cumulative noise parameter as:

$$\bar{\beta}_t = 1 - \bar{\alpha}_t$$

Thus, any step x_t in the forward diffusion process can be expressed as:

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$ is standard normal noise.

1.5 Application and Significance

The key to the forward diffusion process lies in its foundation for the reverse denoising process. In DDPM, the model's training objective is to learn how to recover high-quality data from noise samples through a reverse process. The forward process provides the mapping from data to noise, enabling the model to perform symmetric denoising.

Report on task 2, Theoretical Analysis of the reverse diffusion process in DDPM.

2 Theoretical Analysis of the reverse diffusion process in DDPM.

2.1 Introduction

Denoising Diffusion Probabilistic Models (DDPM) are a class of generative models that excel in generating high-quality data samples, particularly in image generation tasks. The forward diffusion process gradually corrupts data with Gaussian noise, transforming it into a near-pure noise state. The reverse diffusion process, which is the focus of this report, aims to revert this noisy data back to the original high-quality samples. This report provides an in-depth explanation of the principles, mathematical derivation, and application of the reverse diffusion process in DDPM.

2.2 Overview of the Reverse Diffusion Process

The reverse diffusion process is the cornerstone of DDPM. It involves learning to reverse the noise added during the forward diffusion process. Given a noisy sample x_T (where T denotes the final timestep in the forward process), the goal is to progressively denoise this sample to recover the original data x_0. This process is achieved through a series of learned reverse transitions.

2.3 Mathematical Definition

The reverse diffusion process is defined by a series of learned conditional distributions that approximate the reverse transitions of the forward diffusion process. Specifically, the reverse process can be formulated as:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

Here, μ_θ and Σ_θ are the mean and variance predicted by a neural network parameterized by θ. These parameters are learned to closely approximate the true reverse transition distributions.

2.4 Mathematical Derivation

Deriving the Reverse Conditional Distributions

The forward diffusion process is defined by:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

where $\beta_t = 1-\alpha_t$. Using the property of Gaussian distributions, we can derive the form of the reverse conditional distributions:

$$q(x_{t-1}|x_t, x_0) \propto q(x_t|x_{t-1})q(x_{t-1}|x_0)$$

Since both $q(x_t|x_{t-1})$ and $q(x_{t-1}|x_0)$ are Gaussians, the product of Gaussians is also Gaussian. This allows us to write:

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I)$$

where $\tilde{\mu}_t(x_t, x_0)$ and $\tilde{\beta}_t$ can be derived through standard Gaussian manipulation techniques. For simplicity, the approximations used in DDPM lead to:

$$\tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t$$

and

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

Parameterization and Learning

The parameterization and learning process in Denoising Diffusion Probabilistic Models (DDPMs) begins with the goal of maximizing the log-likelihood of the data. The log-likelihood estimation provides a measure of how well the model explains the observed data. Formally, the log-likelihood can be expressed as:

$$\log p_\theta(x_0)$$

However, directly maximizing this log-likelihood is intractable due to the complex dependencies between the data points. Therefore, an alternative approach is employed by introducing a variational distribution q to approximate the true posterior distribution.

To make the optimization tractable, the problem is reformulated into maximizing the Evidence Lower Bound (ELBO). The ELBO serves as a lower bound to the log-likelihood and is easier to optimize. The ELBO can be derived as follows:

$$\log p_\theta(x_0) \geq \mathbb{E}_{q(x_{1:T}|x_0)} \left[\log p_\theta(x_0|x_{1:T})\right] - \mathrm{KL}(q(x_{1:T}|x_0) \| p_\theta(x_{1:T}))$$

This reformulation decomposes the log-likelihood into the reconstruction error and KL divergence terms.

- **Reconstruction Error Term**:
$$\mathbb{E}_{q(x_{1:T}|x_0)} \left[\log p_\theta(x_0|x_{1:T})\right]$$
This term measures how well the model reconstructs the original data x_0 from the noisy data $x_{1:T}$.
- **KL Divergence Term**:
$$\sum_{t=1}^{T} \mathbb{E}_{q(x_{t-1}|x_t, x_0)} \left[\mathrm{KL}(q(x_t|x_{t-1}) \| p_\theta(x_t|x_{t-1}))\right]$$
This term measures the divergence between the true forward process q and the learned reverse process p_θ.

Based on empirical findings, it is often sufficient to focus on the mean while ignoring the variance in the KL divergence term. This simplification reduces computational complexity and still achieves good performance. The mean is crucial for capturing the primary structure of the data, while the variance, although important, can be approximated or even ignored in certain scenarios.

By simplifying the KL divergence to primarily consider the mean, the ELBO can be further streamlined. This approach leverages the fact that the mean often captures the most significant aspects of the data distribution, making it a practical choice for efficient learning.

Through these simplifications, the final equations for the parameterization and learning in DDPMs are derived. The neural network predicts the noise $\epsilon_\theta(x_t, t)$ added during the forward process, and the mean μ_θ is adjusted accordingly. The resulting prediction for x_{t-1} is given by:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sqrt{\beta_t} z_t$$

where $z_t \sim \mathcal{N}(0, I)$ is sampled noise.

2.5 Training Objective

The training objective for DDPM involves minimizing the difference between the true noise ϵ and the predicted noise ϵ_θ. This is typically done using a simple mean squared error (MSE) loss:

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

where $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$.

2.6 Application and Significance

The reverse diffusion process is crucial for the practical application of DDPMs. By effectively learning to denoise a progressively corrupted sample, the model can generate high-quality samples from pure noise. This approach has demonstrated impressive results in various data generation tasks, including image synthesis and audio generation.

Report on task 3, Code Replication Task

3 Code Replication Task

The code is divided into two main functions: `train` and `eval`, each with several key steps. The `train` function is responsible for setting up the dataset, model, optimizer, and training loop. The `eval` function is used to evaluate the model by generating images from noise and saving them. Here is a detailed explanation in a structured format.

3.1 Importing Modules

```
import os
from typing import Dict

import torch
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import CIFAR10
from torchvision.utils import save_image

from Diffusion import GaussianDiffusionSampler, GaussianDiffusionTrainer
from Diffusion.Model import UNet
from Scheduler import GradualWarmupScheduler
```

These imports include necessary libraries for data handling, model building, training, and evaluation.

3.2 Train Function

3.2.1 Setup Device and Dataset

```
def train(modelConfig: Dict):
    device = torch.device(modelConfig["device"])
    dataset = CIFAR10(
        root='./CIFAR10', train=True, download=True,
        transform=transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]))
    dataloader = DataLoader(
        dataset, batch_size=modelConfig["batch_size"], shuffle=True, num_workers=4,
            drop_last=True, pin_memory=True)
```

3.2.2 Model Initialization and Optimizer Setup

```
    net_model = UNet(T=modelConfig["T"], ch=modelConfig["channel"], ch_mult=modelConfig["
        channel_mult"], attn=modelConfig["attn"],
```

```
                num_res_blocks=modelConfig["num_res_blocks"], dropout=modelConfig["
                    dropout"]).to(device)
    if modelConfig["training_load_weight"] is not None:
        net_model.load_state_dict(torch.load(os.path.join(
            modelConfig["save_weight_dir"], modelConfig["training_load_weight"]),
                map_location=device))
    optimizer = torch.optim.AdamW(
        net_model.parameters(), lr=modelConfig["lr"], weight_decay=1e-4)
    cosineScheduler = optim.lr_scheduler.CosineAnnealingLR(
        optimizer=optimizer, T_max=modelConfig["epoch"], eta_min=0, last_epoch=-1)
    warmUpScheduler = GradualWarmupScheduler(
        optimizer=optimizer, multiplier=modelConfig["multiplier"], warm_epoch=modelConfig["
            epoch"] // 10, after_scheduler=cosineScheduler)
    trainer = GaussianDiffusionTrainer(
        net_model, modelConfig["beta_1"], modelConfig["beta_T"], modelConfig["T"]).to(device
            )
```

3.2.3 Training Loop

```
    for e in range(modelConfig["epoch"]):
        with tqdm(dataloader, dynamic_ncols=True) as tqdmDataLoader:
            for images, labels in tqdmDataLoader:
                optimizer.zero_grad()
                x_0 = images.to(device)
                loss = trainer(x_0).sum() / 1000.
                loss.backward()
                torch.nn.utils.clip_grad_norm_(
                    net_model.parameters(), modelConfig["grad_clip"])
                optimizer.step()
                tqdmDataLoader.set_postfix(ordered_dict={
                    "epoch": e,
                    "loss: ": loss.item(),
                    "img shape: ": x_0.shape,
                    "LR": optimizer.state_dict()['param_groups'][0]["lr"]
                })
        warmUpScheduler.step()
        torch.save(net_model.state_dict(), os.path.join(
            modelConfig["save_weight_dir"], 'ckpt_' + str(e) + "_.pt"))
```

3.3 Eval Function

3.3.1 Model Loading and Evaluation Setup

```
def eval(modelConfig: Dict):
    with torch.no_grad():
        device = torch.device(modelConfig["device"])
        model = UNet(T=modelConfig["T"], ch=modelConfig["channel"], ch_mult=modelConfig["
            channel_mult"], attn=modelConfig["attn"],
                    num_res_blocks=modelConfig["num_res_blocks"], dropout=0.)
        ckpt = torch.load(os.path.join(
            modelConfig["save_weight_dir"], modelConfig["test_load_weight"]), map_location=
                device)
        model.load_state_dict(ckpt)
```

```
print("model load weight done.")
model.eval()
sampler = GaussianDiffusionSampler(
    model, modelConfig["beta_1"], modelConfig["beta_T"], modelConfig["T"]).to(device)
```

3.3.2 Image Sampling and Saving

```
noisyImage = torch.randn(
    size=[modelConfig["batch_size"], 3, 32, 32], device=device)
saveNoisy = torch.clamp(noisyImage * 0.5 + 0.5, 0, 1)
save_image(saveNoisy, os.path.join(
    modelConfig["sampled_dir"], modelConfig["sampledNoisyImgName"]), nrow=modelConfig
        ["nrow"])
sampledImgs = sampler(noisyImage)
sampledImgs = sampledImgs * 0.5 + 0.5 # [0 ~ 1]
save_image(sampledImgs, os.path.join(
    modelConfig["sampled_dir"], modelConfig["sampledImgName"]), nrow=modelConfig["
        nrow"])
```

3.4 Inference and Training Process Analysis

3.4.1 Inference Process Description

The inference process involves loading the trained model, setting up the sampler, and generating images from noise. Below is a detailed description of each step.

1. Model Loading and Evaluation Setup

- **Disable Gradient Calculation**: Gradient calculation is disabled to save memory and improve performance during inference.
- **Device Configuration**: The device (CPU or GPU) is set based on the provided configuration.
- **Model Initialization**: A U-Net model is initialized with parameters specified in the configuration.
- **Loading Pre-trained Weights**: The pre-trained weights are loaded into the model.
- **Model Evaluation Mode**: The model is set to evaluation mode to disable dropout and batch normalization updates.
- **Sampler Initialization**: A Gaussian diffusion sampler is initialized with the model and diffusion parameters.

2. Image Sampling and Saving

- **Noise Image Generation**: A batch of noisy images is sampled from a standard normal distribution.
- **Save Noisy Images**: The noisy images are scaled to the [0, 1] range and saved.
- **Image Generation**: The sampler generates images from the noisy images using the reverse diffusion process.
- **Save Generated Images**: The generated images are scaled to the [0, 1] range and saved.

3.4.2 Training Process Description

The training process involves several key steps, including setting up the device and dataset, initializing the model and optimizer, and running the training loop. Below is a detailed description of each step.

1. Setup Device and Dataset

- **Device Configuration**: The device (CPU or GPU) is set based on the provided configuration.
- **Dataset Loading**: The CIFAR-10 dataset is loaded with specific transformations:
  - Random horizontal flip for data augmentation.
  - Conversion to tensor format.
  - Normalization to have a mean of 0.5 and a standard deviation of 0.5 for each channel.
- **DataLoader**: A DataLoader is created to iterate over the dataset in batches, with shuffling and multi-threaded data loading.

2. Model Initialization and Optimizer Setup

- **Model Setup**: A U-Net model is initialized with parameters specified in the configuration.
- **Loading Pre-trained Weights**: If pre-trained weights are provided, they are loaded into the model.
- **Optimizer Configuration**: The AdamW optimizer is configured with a specified learning rate and weight decay.
- **Learning Rate Schedulers**: Two schedulers are set up:
  - **CosineAnnealingLR**: For cosine annealing with a minimum learning rate.
  - **GradualWarmupScheduler**: To gradually warm up the learning rate for a fraction of the total epochs, followed by cosine annealing.
- **Diffusion Trainer**: A Gaussian diffusion trainer is initialized with the model and diffusion parameters.

3. Training Loop

- **Epoch Iteration**: The training loop iterates over the specified number of epochs.
- **Batch Processing**: For each batch of data:
  - Gradients are reset to zero.
  - Images are moved to the device.
  - Loss is computed using the diffusion trainer.
  - Backpropagation is performed to compute gradients.
  - Gradients are clipped to prevent exploding gradients.
  - Model parameters are updated using the optimizer.
  - Training progress is displayed using `tqdm`.
- **Scheduler Step**: The warm-up scheduler is updated after each epoch.
- **Model Checkpointing**: Model weights are saved at the end of each epoch.

3.5 Summary

- **Importing Modules**: Import required libraries and custom modules.
- **Train Function**:
  - **Setup Device and Dataset**: Initialize the device and load the CIFAR-10 dataset with transformations.
  - **Model Initialization and Optimizer Setup**: Initialize the U-Net model, optimizer, and learning rate schedulers. Load pre-trained weights if available.
  - **Training Loop**: Iterate over epochs and batches, compute loss, perform backpropagation, update model parameters, and save model weights.
- **Eval Function**:
  - **Model Loading and Evaluation Setup**: Load the model weights, set up the model for evaluation, and initialize the sampler.
  - **Image Sampling and Saving**: Generate images from noise, scale them, and save the noisy and sampled images.

Report on task 4, Training Task

4 Training Task

Through the training process, we obtained results with Epoch values equal to 0, 50, 100, 150, and 199, respectively. The result is shown below.

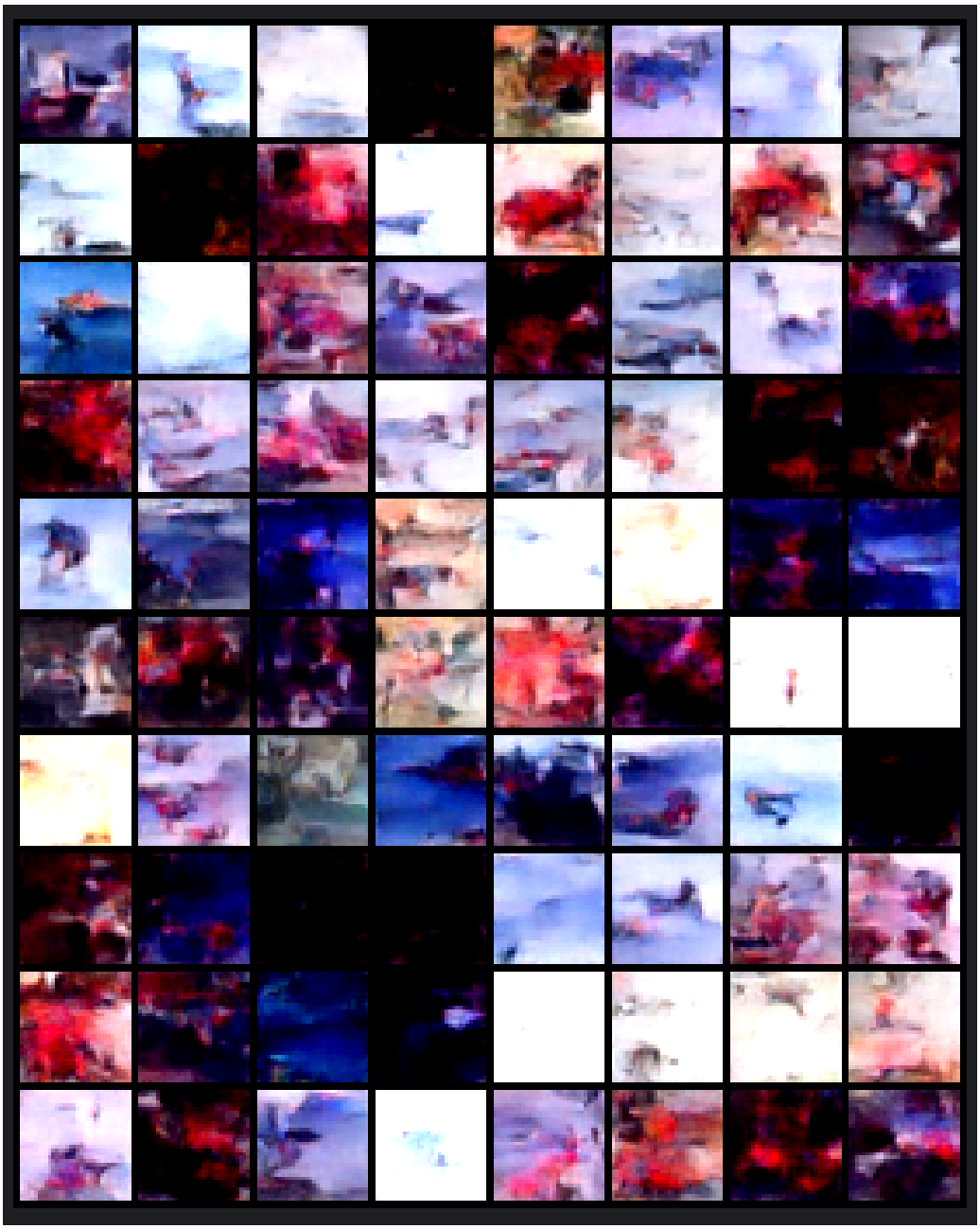

Figure 1: Epoch=0

Initial State: At the beginning of training, the images generated by the model are almost entirely noise, with no discernible objects or shapes. The model has not learned anything yet, and the generated images are indistinguishable from the input noise.

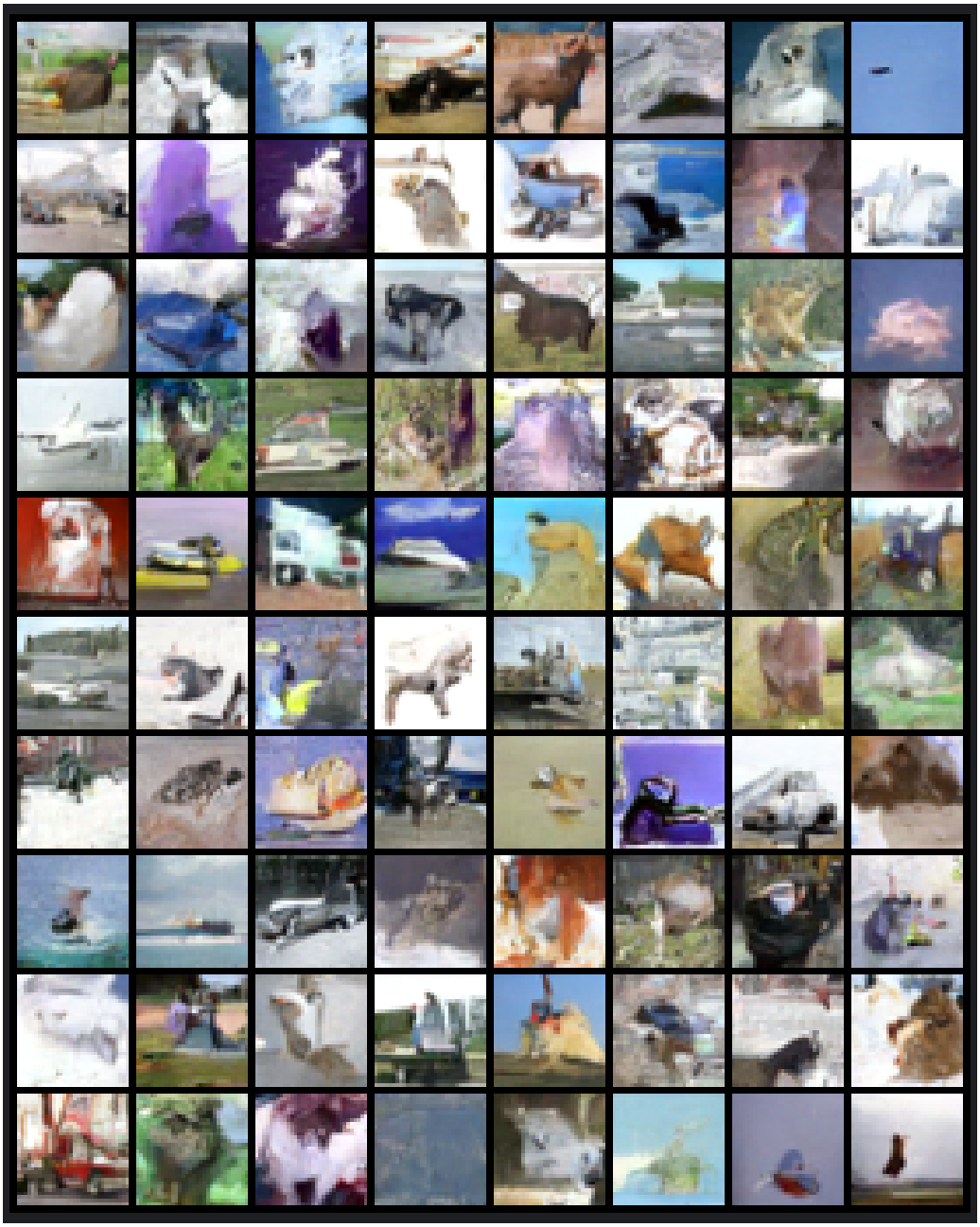

Figure 2: Epoch=50

Early Training: After 50 epochs of training, the generated images start to show some blurry shapes and colors, but still lack clear structure or objects.The model is beginning to learn some basic features of the data, but more training is needed to improve the quality of the generated images.

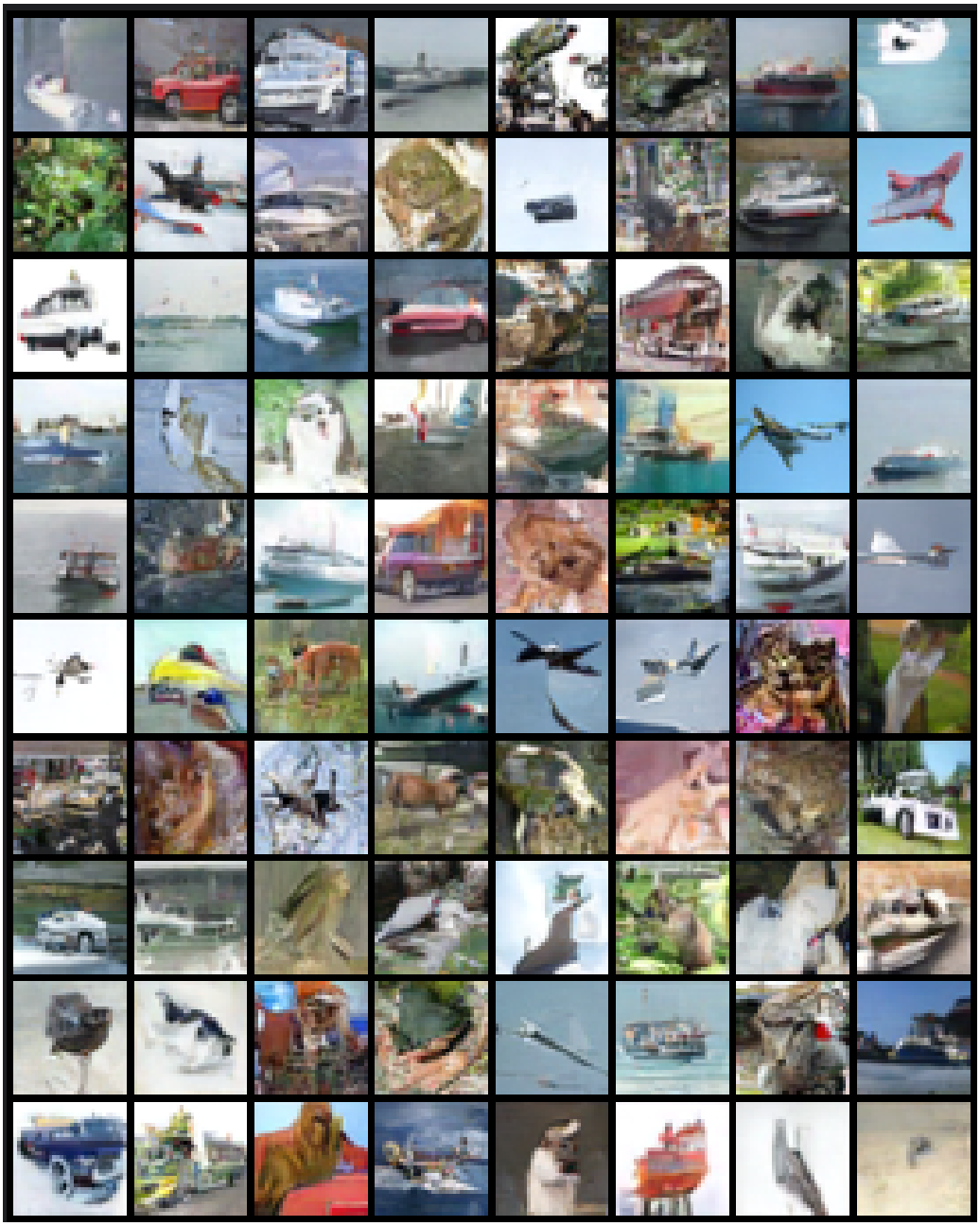

Figure 3: Epoch=100

Mid Training: After 100 epochs, the generated images start to show more distinct shapes and objects, though there is still noticeable blurriness and distortion.The model has learned more features of the data, and the quality of the generated images has improved, but further training is required to produce clearer images.

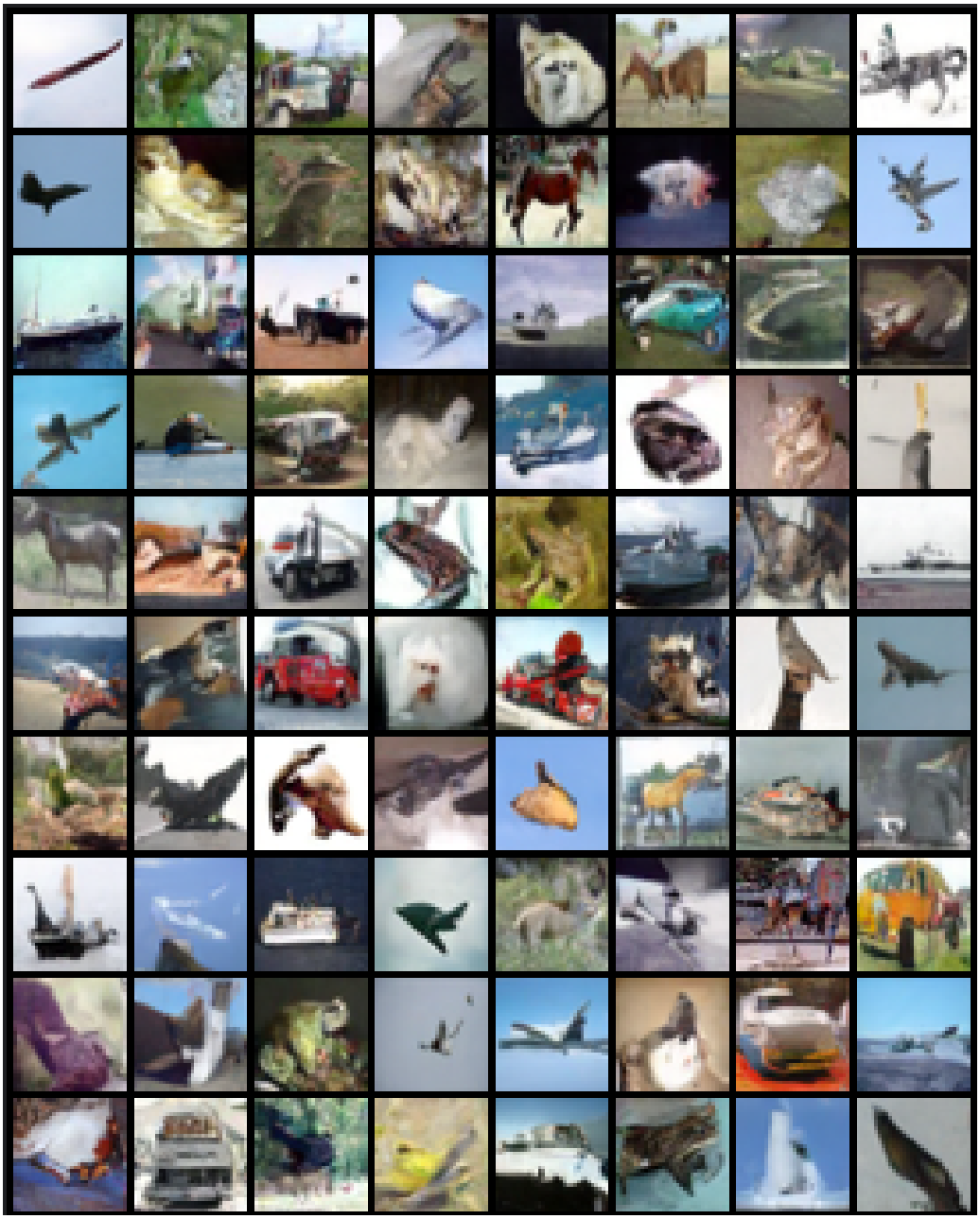

Figure 4: Epoch=150

Late Training: After 150 epochs, the generated images become clearer, with more discernible details in the shapes and objects.The model has continued to improve during training and is now capable of generating fairly clear and realistic images, though some details still need refinement.

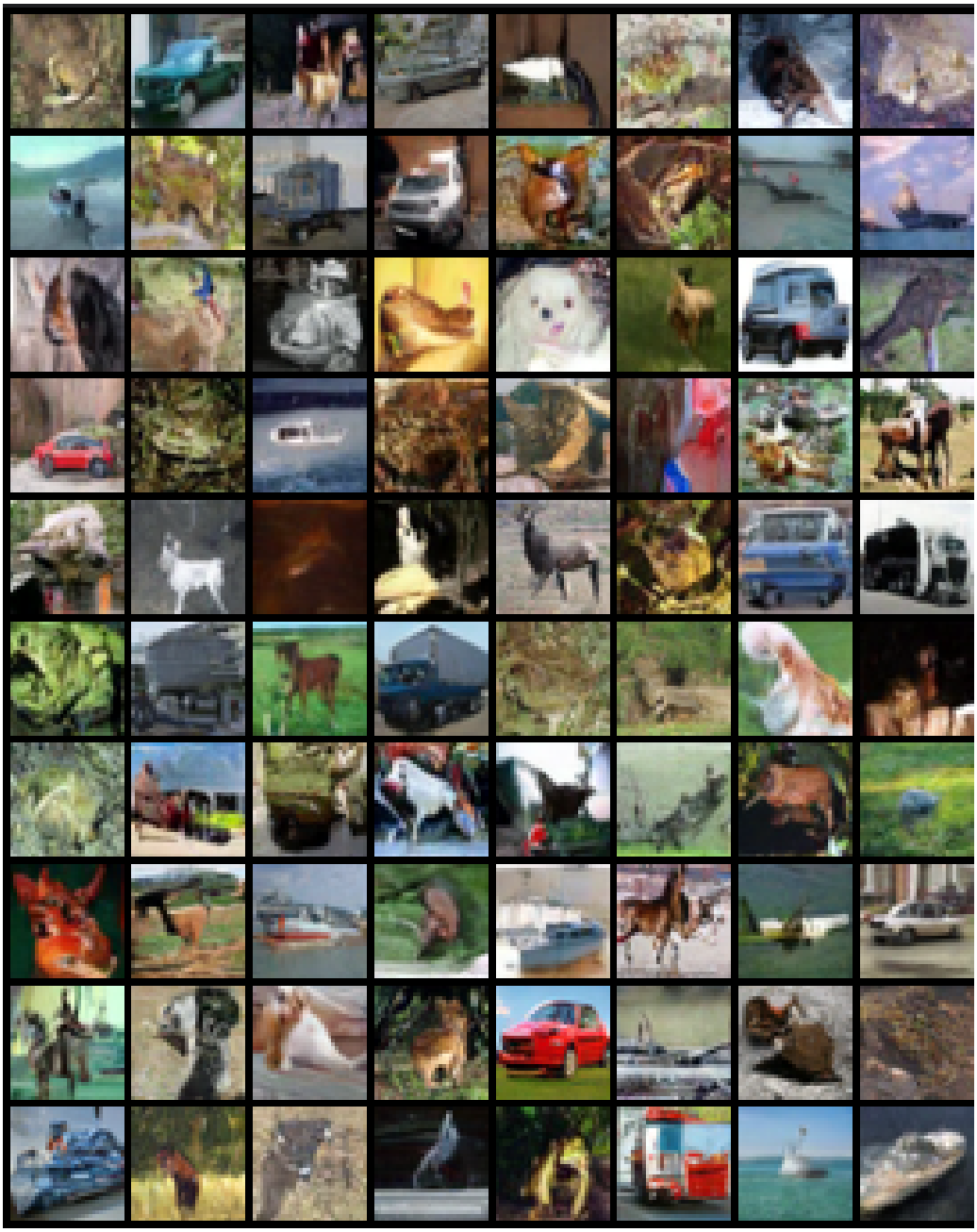

Figure 5: Epoch=199

Final State: After 199 epochs, the quality of the generated images has further improved, with more precise details and overall image quality approaching that of real images.After sufficient training, the model can generate high-quality images, nearly achieving the desired level of realism.

From these images at different epoch values, we can see that the quality of the generated images gradually improves with increased training. The model starts from complete noise, gradually learning the features of the data, and eventually producing clearer and more realistic images. This indicates that the model continuously improves during training, eventually mastering the ability to generate high-quality images.

Report on task 5, Results Analysis Task

REPORT TEXT.

References

- Ho, J., Jain, A., Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. In Advances in Neural Information Processing Systems (NeurIPS).
- Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., Ganguli, S. (2015). Deep Unsupervised Learning using Nonequilibrium Thermodynamics. In Proceedings of the 32nd International Conference on Machine Learning (ICML).
- Kingma, D. P., Welling, M. (2013). Auto-Encoding Variational Bayes. arXiv preprint arXiv:1312.6114.