~\OneDrive\Desktop\Exercise 6 - Simple Diffusion\Simple diffusion.py

```python
import matplotlib.pyplot as plt
import numpy as np
import torch
import seaborn as sns  # a useful plotting library on top of matplotlib
from tqdm.auto import tqdm # a nice progress bar


def normalize(x, mean, std):
    return (x - mean) / std

def denormalize(x, mean, std):
    return x * std + mean


# generate a dataset of 1D data from a mixture of two Gaussians
# this is a simple example, but you can use any distribution
data_distribution = torch.distributions.mixture_same_family.MixtureSameFamily(
    torch.distributions.Categorical(torch.tensor([1, 2])),
    torch.distributions.Normal(torch.tensor([-4., 4.]), torch.tensor([1., 1.]))
)

dataset = data_distribution.sample(torch.Size([10000]))  # create training data set
dataset_validation = data_distribution.sample(torch.Size([1000])) # create validation data set


mean = dataset.mean()
std = dataset.std()
dataset_norm = normalize(dataset, mean, std)
dataset_validation_norm = normalize(dataset_validation, mean, std)

# ============================ HYPERPARAMETERS ============================

TIME_STEPS = 250
BETA = torch.full((TIME_STEPS,), 0.02)
N_EPOCHS = 1000
BATCH_SIZE = 64
LEARNING_RATE = 0.8e-4

# define the neural network that predicts the amount of noise that was
# added to the data
# the network should have two inputs (the current data and the time step)
# and one output (the predicted noise)
# ================================================= Model ===========================================
class NoisePredictor(torch.nn.Module):
    def __init__(self): # define simple nn with concatenation
        super(NoisePredictor, self).__init__()
        self.fc1 = torch.nn.Linear(2, 128)  # Input layer (data + time step)
        self.fc2 = torch.nn.Linear(128, 128)  # Hidden layer
        self.fc3 = torch.nn.Linear(128, 1)  # Output layer (predicted noise)
        self.tanh = torch.nn.Tanh()
```

```python

    def forward(self, x, t):
        # Concatenate data and time step
        t = t.unsqueeze(1) # [BATCH SIZE, 1]
        x = x.view(x.size(0), -1) # Flatten the input data s.t. [BATCH SIZE, 1]

        input_tensor = torch.cat((x, t.float()), dim=1)  # [BATCH SIZE, 2] e.g.  Sample 1 → x_t = 0.34, timestep t = 5

        x = self.tanh(self.fc1(input_tensor))
        x = self.tanh(self.fc2(x))
        x = self.fc3(x)
        return x


def train_model(g, dataset_norm, dataset_validation_norm):

    epochs = tqdm(range(N_EPOCHS))  # this makes a nice progress bar
    criterion = torch.nn.MSELoss()  # Use Mean Squared Error Loss
    optimizer = torch.optim.Adam(g.parameters(), lr=LEARNING_RATE)

    bar_alpha = torch.cumprod(1 - BETA, dim=0) # Precompute the cumulative product for all time steps
    total_loss = 0
    n_batches = 0

    for e in epochs:  # loop over epochs
        g.train()
        # loop through batches of the dataset, reshuffling it each epoch
        indices = torch.randperm(dataset_norm.shape[0]) # shuffle the dataset
        shuffled_dataset_norm = dataset_norm[indices] # shuffle the dataset

        for i in range(0, shuffled_dataset_norm.shape[0] - BATCH_SIZE, BATCH_SIZE): # loop through the dataset in batches
            x0 = shuffled_dataset_norm[i:i + BATCH_SIZE].view(-1, 1) # sample a batch of data and add dimension [B] --> [B,1] since this is necassary format for the NN

            # here, implement algorithm 1 of the DDPM paper (https://arxiv.org/abs/2006.11239)
            t = torch.randint(0, TIME_STEPS, (BATCH_SIZE,))  # sample uniformly a time step
            noise = torch.randn_like(x0)  # sample the noise
            bar_alpha_t = bar_alpha[t].view(-1, 1)  # compute the product of alphas up to time t and add dimension

            x_t = torch.sqrt(bar_alpha_t) * x0 + torch.sqrt(1 - bar_alpha_t) * noise # --> [B, 1]
            predicted_noise = g(x_t, t.float())  # compute the predicted noise

            # compute the loss (mean squared error between predicted noise and true noise)
            loss = criterion(predicted_noise, noise)

            # backpropagation and loss stuff
            optimizer.zero_grad()
            loss.backward()
```

```
            optimizer.step()

            total_loss += loss.item()
            n_batches += 1
            avg_loss = total_loss / n_batches

        # compute the loss on the validation set
        g.eval()
        with torch.no_grad():
            x0 = dataset_validation_norm
            t = torch.randint(0, TIME_STEPS, (x0.shape[0],))  # sample a time step for validation
            noise = torch.randn_like(x0)  # sample the noise
            val_bar_alpha_t = bar_alpha[t]  # compute the product of alphas up to time t
            x_t = torch.sqrt(val_bar_alpha_t) * x0 + torch.sqrt(1 - val_bar_alpha_t) * noise  # add noise to the validation data

            predicted_noise = g(x_t, t.float())# Compute the predicted noise

            val_loss = criterion(predicted_noise, noise) # Calculate the validation loss
            print(f" Epoch {e+1}/{N_EPOCHS}| Training loss: {avg_loss} | Validation Loss: {val_loss.item()}")



def sample_and_track(g, count):
    """
    Sample from the model by applying the reverse diffusion process

    Here, implement algorithm 2 of the DDPM paper (https://arxiv.org/abs/2006.11239)

    Parameters
    ----------
    g : torch.nn.Module
        The neural network that predicts the noise added to the data
    count : int
        The number of samples to generate in parallel

    Returns
    -------
    x : torch.Tensor
        The final sample from the model
----------------------------------------------------------------
    Perform reverse diffusion:
    - Return final sampled values for all `count`
    - Track one sample (sample 0) over time using x_batch
    """
    g.eval()
    bar_alpha = torch.cumprod(1 - BETA, dim=0)

    x_batch = torch.randn(count, 1)  # [count, 1]
    tracked_index = 0 # track first index
    history = [x_batch[tracked_index].item()]  # Track first sample

```

```python
    for t in range(TIME_STEPS - 1, -1, -1):
        t_tensor_batch = torch.full((count,), t, dtype=torch.long)

        # Predict noise
        pred_noise_batch = g(x_batch, t_tensor_batch.float()).view(-1, 1)

        # Get scalars
        bar_alpha_t = bar_alpha[t]
        alpha_t = 1 - BETA[t]
        factor = (1 - alpha_t) / torch.sqrt(1 - bar_alpha_t)
        sigma_t = torch.sqrt(BETA[t])

        # Random noise (zero for last step)
        z_batch = torch.randn_like(x_batch) if t > 0 else torch.zeros_like(x_batch)

        # Reverse step
        x_batch = (1 / torch.sqrt(alpha_t)) * (x_batch - factor * pred_noise_batch) + sigma_t * z_batch # Posterior decoded pixel value

        # Track sample 0
        history.append(x_batch[tracked_index].item())

    # Denormalize
    samples = denormalize(x_batch, mean, std).detach().numpy().flatten()
    history = denormalize(torch.tensor(history), mean, std).numpy()

    return samples, history

# ============================ Plots ===========================
def plot_distribution(samples):
    fig, ax = plt.subplots(1, 1, figsize=(8, 5))
    bins = np.linspace(-10, 10, 50)
    sns.kdeplot(dataset.numpy().flatten(), ax=ax, color='blue', label='True distribution', linewidth=2)
    sns.histplot(samples, ax=ax, bins=bins, color='red', label='Sampled distribution', stat='density', alpha=0.7)
    ax.legend()
    ax.set_xlabel('Sample value')
    ax.set_ylabel('Sample count')
    plt.title(f"Final Sample Distribution After {N_EPOCHS} Epochs")
    plt.grid(True)

    plt.savefig("Figures/final_distribution.png", dpi=300)
    plt.close()


def plot_monte_carlo(all_histories):
    plt.figure(figsize=(8, 5))
    for history in all_histories:
        plt.plot(range(TIME_STEPS + 1), history, alpha=0.5, linewidth=1)
    plt.xlabel('timestep T - t')
    plt.ylabel('Sample value')
    plt.title(f'Sample History After {N_EPOCHS} Epochs')
    plt.grid(True)
```

```python

    plt.savefig("Figures/sample_history.png", dpi=300)
    plt.close()


def generate_plots(g, N):
    all_histories = []

    for i in range(N):
        samples, history = sample_and_track(g, 1000)
        # Only save histogram plot for first run
        if i == 0:
            plot_distribution(samples)

        all_histories.append(history)

    plot_monte_carlo(all_histories)


# =========================== RUNNING THE CODE ===========================

g = NoisePredictor()
train_model(g, dataset_norm, dataset_validation_norm)

generate_plots(g, 50)
```