# Exercise 6: Simple Diffusion

**Erik Stolt**

## What you did and how

In this exercise we explore Diffusion Model, a relatively simple architecture that exploits noise for data augmentation and to generate new samples following the same distribution as the training data. In its simplest form, we start by, in time steps, adding noise to a training data point (this is the forward pass), then we train a NN to denoise the the training example in order to reconstruct the image. In the end, the idea is that we can simply start with an image that is complete noise (Gaussian in our case, but it could in theory be another distribution), and then pass it through the network to get a new unique image that does not appear in the data set. In practice we do this by following the algorithm [1]:

**Algorithm 1** **Training**

1: **repeat**
2: $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$
3: $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on

$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta \left( \sqrt{\bar{\alpha}_t} \boldsymbol{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon}, t \right) \right\|^2$$

6: **until** converged

where $\boldsymbol{x}_0$ is an image from the training data set. The advantage of diffusion models is that a single training example can be used multiple times, since in each iteration we add an independent noise vector to it. Although in this case, our image is only a single pixel. Notice that $\boldsymbol{\epsilon}_\theta$ is actually the total noise added to the image over $t$ time steps. This is not what we want, we want the model to predict the noise added to the image in each time step. It turns out, as done in the paper, that one can parametrize the model using a change of variable to obtain this. The end result is given in algorithm 2 [1] where we now have an additional scaling factor in front of $\boldsymbol{\epsilon}_\theta$:

**Algorithm 2** **Sampling**

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

We can now sample from algorithm 2 as man times as we want to end up with a distribution that follows the training data, and which we can input random noise $\boldsymbol{x}_T$ to and receive a new image.

## What results you obtained

One example print-out during training looked like this:

```
Epoch 77/1000| Training loss: 0.35963864673693213 | Validation Loss: 1.7202495336532593
```

And the following plots were obtained

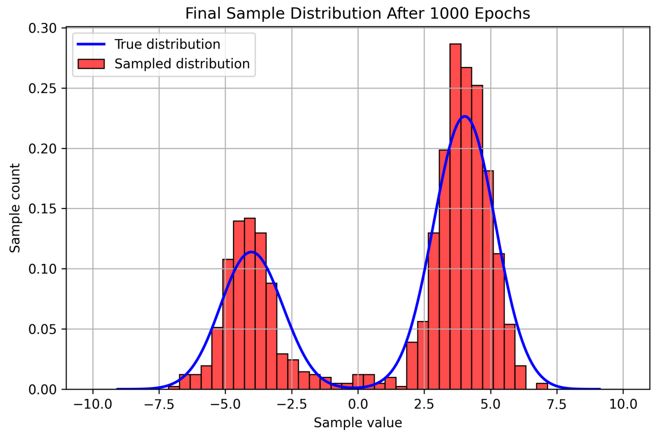


*Figure 1: Posterior distribution of the predicted pixel values (red). Notice that it well aligned with the prior distribution of the training data (blue).*

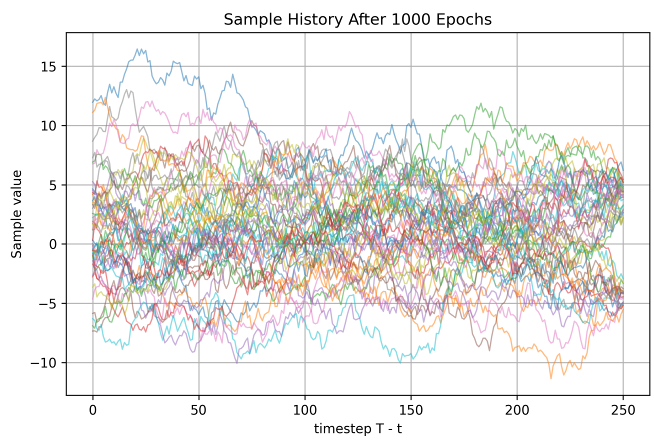


*Figure 2: Evolution of the predicted pixel value as a function of time. Notice that at $t = 250$, the trajectories has converged to the pdf illustrated in Fig.1*

Overall, the results are as expected.

## What challenges you encountered and what could be improved

The main problem I encountered in this exercise was to correctly handle the shapes of all the vectors. I took me a while to correctly implement the algorithms in a way that the code could handle. As this is a relatively simple exercise, there is not much to be improved since a simple network captured all the core features. If that was not the case one would need a better NN architecture. One can always implement this in order to obtain even better results, although, this is not needed here.

## Link to GitHub

Link: `https://github.com/erst6955/Advanced-Applied-Deep-Learning-in-Physics-And-Engineering`

## Code (for backup)

**~\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

```

```
    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)
```

# References

[1] Tom B Brown et al. “Language Models are Few-Shot Learners”. In: *arXiv preprint arXiv:2006.11239* (2020). URL: `https://arxiv.org/abs/2006.11239`.