Lecture 12.3: Generating Images with Denoising Diffusion Probabilistic Models

By courtesy of Brian Pulfer

1. Introduction

Denoising Diffusion Probabilistic Models (DDPMs) are generative models based on the idea of reversing a noising process. The idea is fairly simple: Given a dataset, make it more and more noisy with a deterministic process. Then, learn a model that can undo this process.

DDPMs have recently drawn a lot of attention due to their high-quality samples. In this notebook, we will implement the first and most fundamental paper to be familiar with when dealing with DDPMs: *Denoising Diffusion Probabilistic Models* (https://arxiv.org/pdf/2006.11239.pdf) by Ho et. al.

Before starting, we need to install the **einops** package. This will be used to create a nice GIF animation of DDPM.

In [1]:
```
!pip3 install --upgrade pip
!pip3 install einops
```

```
Requirement already satisfied: pip in /usr/local/lib/python3.10/dist-packages (24.1.2)
Collecting pip
  Downloading pip-24.3.1-py3-none-any.whl.metadata (3.7 kB)
Downloading pip-24.3.1-py3-none-any.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 20.3 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 24.1.2
    Uninstalling pip-24.1.2:
      Successfully uninstalled pip-24.1.2
Successfully installed pip-24.3.1
Requirement already satisfied: einops in /usr/local/lib/python3.10/dist-packages (0.8.0)
```

Then, we need to initialize Python and import the required libraries. Run the below cell.

In [2]:
```
# Import of libraries
import random
import imageio
import numpy as np
from argparse import ArgumentParser

from tqdm.auto import tqdm
import matplotlib.pyplot as plt

import einops
import torch
import torch.nn as nn
from torch.optim import Adam
from torch.utils.data import DataLoader

from torchvision.transforms import Compose, ToTensor, Lambda
from torchvision.datasets.mnist import MNIST, FashionMNIST

# Setting reproducibility
SEED = 0
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

# Definitions
```

```
STORE_PATH_MNIST = f"ddpm_model_mnist.pt"
STORE_PATH_FASHION = f"ddpm_model_fashion.pt"
```

2. Implementation

Execution Options

Here's a few options we should set:

- `no_train` specifies whether we want to skip the training loop and just use a pre-trained model. If we haven't trained a model already using this notebook, keep this as `False`. If we want to use a pre-trained model, load it in the colab filesystem.
- `fashion` specifies whether we want to use the Fashion-MNIST dataset (`True`) or not and use the MNIST dataset instead (`False`).
- `batch_size` , `n_epochs` and `lr` are typical training hyper-parameters. Notice that `lr=0.001` is the hyper-parameter used by the authors.

In [3]:
```
no_train = False
fashion = True
batch_size = 128
n_epochs = 20
lr = 0.001
```

In [4]:
```
store_path = "ddpm_fashion.pt" if fashion else "ddpm_mnist.pt"
```

Utility Functions

Following are two utility functions: `show_images` allows us to display images in a square-like pattern with a custom title, while `show_fist_batch` simply shows the images in the first batch of a DataLoader object.

In [5]:
```
def show_images(images, title=""):
    """Shows the provided images as sub-pictures in a square"""

    # Converting images to CPU numpy arrays
    if type(images) is torch.Tensor:
        images = images.detach().cpu().numpy()

    # Defining number of rows and columns
    fig = plt.figure(figsize=(8, 8))
    rows = int(len(images) ** (1 / 2))
    cols = round(len(images) / rows)

    # Populating figure with sub-plots
    idx = 0
    for r in range(rows):
        for c in range(cols):
            fig.add_subplot(rows, cols, idx + 1)

            if idx < len(images):
                plt.imshow(images[idx][0], cmap="gray")
                idx += 1
    fig.suptitle(title, fontsize=30)

    # Showing the figure
    plt.show()
```

In [6]:
```
def show_first_batch(loader):
    for batch in loader:
        show_images(batch[0], "Images in the first batch")
        break
```

Loading Data

We will use the MNIST (or FashionMNIST) dataset and try to generate some new samples from some random gaussian noise.

NOTE: It is important to normalize images in range `[-1,1]` and not `[0,1]` as one might usually do. This is because the DDPM network predicts normally distributed noise throughout the denoising process.

In [7]:
```
# Loading the data (converting each image into a tensor and normalizing between [-1, 1])
transform = Compose([
    ToTensor(),
    Lambda(lambda x: (x - 0.5) * 2)]
)
ds_fn = FashionMNIST if fashion else MNIST
dataset = ds_fn("./datasets", download=True, train=True, transform=transform)
loader = DataLoader(dataset, batch_size, shuffle=True)
```

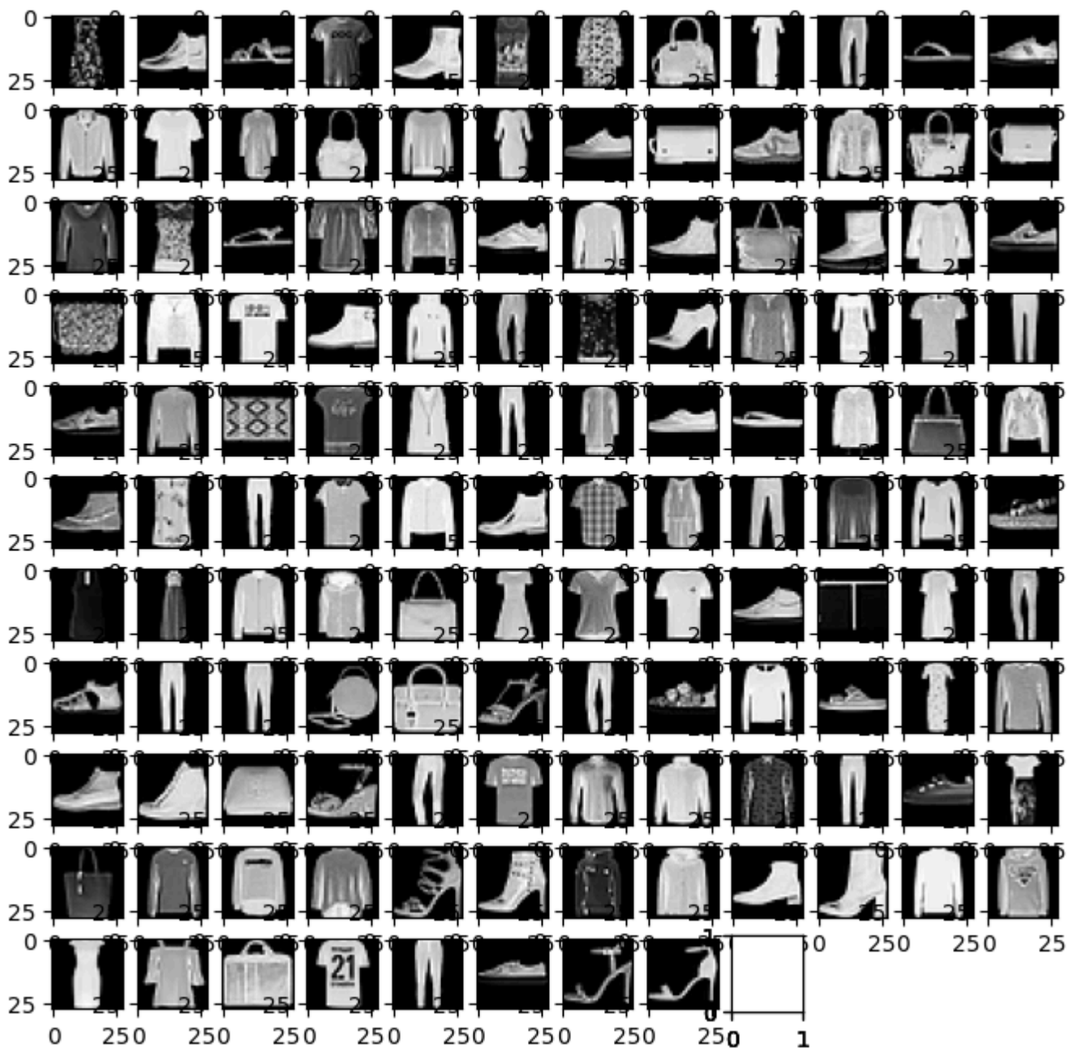
```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./datasets/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26.4M/26.4M [00:02<00:00, 12.5MB/s]
Extracting ./datasets/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./datasets/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./datasets/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29.5k/29.5k [00:00<00:00, 209kB/s]
Extracting ./datasets/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./datasets/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./datasets/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4.42M/4.42M [00:01<00:00, 3.93MB/s]
Extracting ./datasets/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./datasets/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./datasets/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5.15k/5.15k [00:00<00:00, 19.3MB/s]
Extracting ./datasets/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./datasets/FashionMNIST/raw
```

In [8]:
```
# Optionally, show a batch of regular images
show_first_batch(loader)
```



Getting Device

If we are running this codebook from Google Colab, we should make sure that we are using a GPU runtime. Typically, a *Tesla T4* GPU is provided.

In [9]:
```
# Getting device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}\t" + (f"{torch.cuda.get_device_name(0)}" if torch.cuda.is_available() else "CPU"))
```

```
Using device: cuda      Tesla T4
```

Defining the DDPM Module

We now proceed and define a DDPM PyTorch module. Since the DDPM scheme is independent of the model architecture used in each denoising step, we define a high-level model that is constructed using a `network` parameter, as well as:

- `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 onto which the model is run;
- `image_chw` : tuple contining dimensionality of images.

(i) The `forward` process of DDPMs benefits from a nice property: We don't actually need to slowly add noise step-by-step, but we can directly skip to whathever step t we want using coefficients $\bar{\alpha}$:

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

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon} \mid \mathbf{0}, \mathbf{1})$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, and $\alpha_i = 1 - \beta_i$.

(ii) For the `backward` method, we let the network do the job.

Note that in this implementation, t is assumed to be a `(N, 1)` tensor, where `N` is the number of images in tensor `x`. We thus support different time-steps for multiple images.

In [10]:
```
# DDPM class
class MyDDPM(nn.Module):
    def __init__(self, network, n_steps=200, min_beta=10 ** -4, max_beta=0.02, device=None, image_chw=(1, 28, 28)):
        super(MyDDPM, self).__init__()
        self.n_steps = n_steps
        self.device = device
        self.image_chw = image_chw
        self.network = network.to(device)
        self.betas = torch.linspace(min_beta, max_beta, n_steps).to(
            device)  # Number of steps is typically in the order of thousands
        self.alphas = 1 - self.betas
        self.alpha_bars = torch.tensor([torch.prod(self.alphas[:i + 1]) for i in range(len(self.alphas))]).to(device)

    def forward(self, x0, t, eta=None):
        # Make input image more noisy (we can directly skip to the desired step)
        n, c, h, w = x0.shape
        a_bar = self.alpha_bars[t]

        if eta is None:
            eta = torch.randn(n, c, h, w).to(self.device)

        noisy = a_bar.sqrt().reshape(n, 1, 1, 1) * x0 + (1 - a_bar).sqrt().reshape(n, 1, 1, 1) * eta
        return noisy

    def backward(self, x, t):
        # Run each image through the network for each timestep t in the vector t.
        # The network returns its estimation of the noise that was added.
        return self.network(x, t)
```

Functions for Visualizing Forward and Backward Process

Now that we have defined the high-level function of a DDPM, we can define some related utility functions.

In particular, we will show the forward process (which is independent of the denoising network) with the `show_forward` method.

Then, we will run the backward pass and generate new images with the `generate_new_images` method.

Notice that in the paper (https://arxiv.org/pdf/2006.11239.pdf) by Ho et. al., two options are considered for σ_t^2:

- $\sigma_t^2 = \beta_t$
- $\sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$

By default, we choose the first option out of simplicity.

In [11]:
```
def show_forward(ddpm, loader, device):
    # Showing the forward process
    for batch in loader:
        imgs = batch[0]

        show_images(imgs, "Original images")

        for percent in [0.25, 0.5, 0.75, 1]:
            show_images(
                ddpm(imgs.to(device),
                     [int(percent * ddpm.n_steps) - 1 for _ in range(len(imgs))]),
                f"DDPM Noisy images {int(percent * 100)}%"
            )
        break
```

In [12]:
```
def generate_new_images(ddpm, n_samples=16, device=None, frames_per_gif=100, gif_name="sampling.gif", c=1, h=28, w=28):
    """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.n_steps, frames_per_gif).astype(np.uint)
    frames = []

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

        # Starting from random noise
        x = torch.randn(n_samples, c, h, w).to(device)

        for idx, t in enumerate(list(range(ddpm.n_steps))[::-1]):
            # Estimating noise to be removed
            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]

            # Partially denoising the image
            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)

                # Option 1: sigma_t squared = beta_t
                beta_t = ddpm.betas[t]
                sigma_t = beta_t.sqrt()

                # Option 2: sigma_t squared = beta_tilda_t
                # prev_alpha_t_bar = ddpm.alpha_bars[t-1] if t > 0 else ddpm.alphas[0]
                # beta_tilda_t = ((1 - prev_alpha_t_bar)/(1 - alpha_t_bar)) * beta_t
                # sigma_t = beta_tilda_t.sqrt()

                # Adding some more noise like in Langevin Dynamics fashion
                x = x + sigma_t * z

            # Adding frames to the GIF
            if idx in frame_idxs or t == 0:
                # Putting digits in range [0, 255]
                normalized = x.clone()
                for i in range(len(normalized)):
                    normalized[i] -= torch.min(normalized[i])
```

```
                normalized[i] *= 255 / torch.max(normalized[i])

            # Reshaping batch (n, c, h, w) to be a (as much as it gets) square frame
            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)

            # Rendering frame
            frames.append(frame)

    # Storing the gif
    with imageio.get_writer(gif_name, mode="I") as writer:
        for idx, frame in enumerate(frames):
            # print("frame.shape: ", frame.shape)
            # print("frame.ndim: ", frame.ndim)
            writer.append_data(frame[:,:,0])
            if idx == len(frames) - 1:
                for _ in range(frames_per_gif // 3):
                    writer.append_data(frames[-1][:,:,0])
    return x
```

UNet Architecture

Now we define an architecture that will be responsible of denoising. While in principle DDPM scheme is independent of the model architecture, we have to be careful to conditioning our model with the temporal information.

Remember that the only term of the loss function that we really care about is $||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)||^2$, where $\boldsymbol{\epsilon}$ is random noise and $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}$ is the model's prediction of the noise. Now, $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}$ is a function of both $\mathbf{x}_0$ and t and we don't want to have a distinct model for each denoising step (thousands of independent models). Instead we want to use a single model that takes as input the image $\mathbf{x}_0$ and the scalar value indicating the timestep t.

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

NOTE: This UNet architecture is designed to work with 28×28 spatial resolution images.

In [13]:

```
def sinusoidal_embedding(n, d):
    # Returns the standard positional embedding
    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
```

In [14]:
```
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
```

In [15]:
```
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, 28, 28), 1, 10),
            MyBlock((10, 28, 28), 10, 10),
            MyBlock((10, 28, 28), 10, 10)
        )
        self.down1 = nn.Conv2d(10, 10, 4, 2, 1)

        self.te2 = self._make_te(time_emb_dim, 10)
        self.b2 = nn.Sequential(
            MyBlock((10, 14, 14), 10, 20),
            MyBlock((20, 14, 14), 20, 20),
            MyBlock((20, 14, 14), 20, 20)
        )
        self.down2 = nn.Conv2d(20, 20, 4, 2, 1)

        self.te3 = self._make_te(time_emb_dim, 20)
        self.b3 = nn.Sequential(
            MyBlock((20, 7, 7), 20, 40),
            MyBlock((40, 7, 7), 40, 40),
            MyBlock((40, 7, 7), 40, 40)
        )
        self.down3 = nn.Sequential(
            nn.Conv2d(40, 40, 2, 1),
            nn.SiLU(),
            nn.Conv2d(40, 40, 4, 2, 1)
        )

        # Bottleneck
        self.te_mid = self._make_te(time_emb_dim, 40)
        self.b_mid = nn.Sequential(
            MyBlock((40, 3, 3), 40, 20),
            MyBlock((20, 3, 3), 20, 20),
            MyBlock((20, 3, 3), 20, 40)
```

```
        )

        # Second half
        self.up1 = nn.Sequential(
            nn.ConvTranspose2d(40, 40, 4, 2, 1),
            nn.SiLU(),
            nn.ConvTranspose2d(40, 40, 2, 1)
        )

        self.te4 = self._make_te(time_emb_dim, 80)
        self.b4 = nn.Sequential(
            MyBlock((80, 7, 7), 80, 40),
            MyBlock((40, 7, 7), 40, 20),
            MyBlock((20, 7, 7), 20, 20)
        )

        self.up2 = nn.ConvTranspose2d(20, 20, 4, 2, 1)
        self.te5 = self._make_te(time_emb_dim, 40)
        self.b5 = nn.Sequential(
            MyBlock((40, 14, 14), 40, 20),
            MyBlock((20, 14, 14), 20, 10),
            MyBlock((10, 14, 14), 10, 10)
        )

        self.up3 = nn.ConvTranspose2d(10, 10, 4, 2, 1)
        self.te_out = self._make_te(time_emb_dim, 20)
        self.b_out = nn.Sequential(
            MyBlock((20, 28, 28), 20, 10),
            MyBlock((10, 28, 28), 10, 10),
            MyBlock((10, 28, 28), 10, 10, normalize=False)
        )

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

    def forward(self, x, t):
        # x is (N, 2, 28, 28) (image with positional embedding stacked on channel dimension)
        t = self.time_embed(t)
        n = len(x)
        out1 = self.b1(x + self.te1(t).reshape(n, -1, 1, 1))  # (N, 10, 28, 28)
        out2 = self.b2(self.down1(out1) + self.te2(t).reshape(n, -1, 1, 1))  # (N, 20, 14, 14)
        out3 = self.b3(self.down2(out2) + self.te3(t).reshape(n, -1, 1, 1))  # (N, 40, 7, 7)

        out_mid = self.b_mid(self.down3(out3) + self.te_mid(t).reshape(n, -1, 1, 1))  # (N, 40, 3, 3)

        out4 = torch.cat((out3, self.up1(out_mid)), dim=1)  # (N, 80, 7, 7)
        out4 = self.b4(out4 + self.te4(t).reshape(n, -1, 1, 1))  # (N, 20, 7, 7)

        out5 = torch.cat((out2, self.up2(out4)), dim=1)  # (N, 40, 14, 14)
        out5 = self.b5(out5 + self.te5(t).reshape(n, -1, 1, 1))  # (N, 10, 14, 14)

        out = torch.cat((out1, self.up3(out5)), dim=1)  # (N, 20, 28, 28)
        out = self.b_out(out + self.te_out(t).reshape(n, -1, 1, 1))  # (N, 1, 28, 28)

        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)
        )
```

We are finally done! Now we need to instantiate a model, show the forward process, and write the usual code that defines a training loop for our model. When the training is done, we will test its generative capabilities.

Instantiating the Model

In [16]:
```
# Defining model
n_steps, min_beta, max_beta = 1000, 10 ** -4, 0.02  # Originally used by the authors
ddpm = MyDDPM(MyUNet(n_steps), n_steps=n_steps, min_beta=min_beta, max_beta=max_beta, device=device)
```

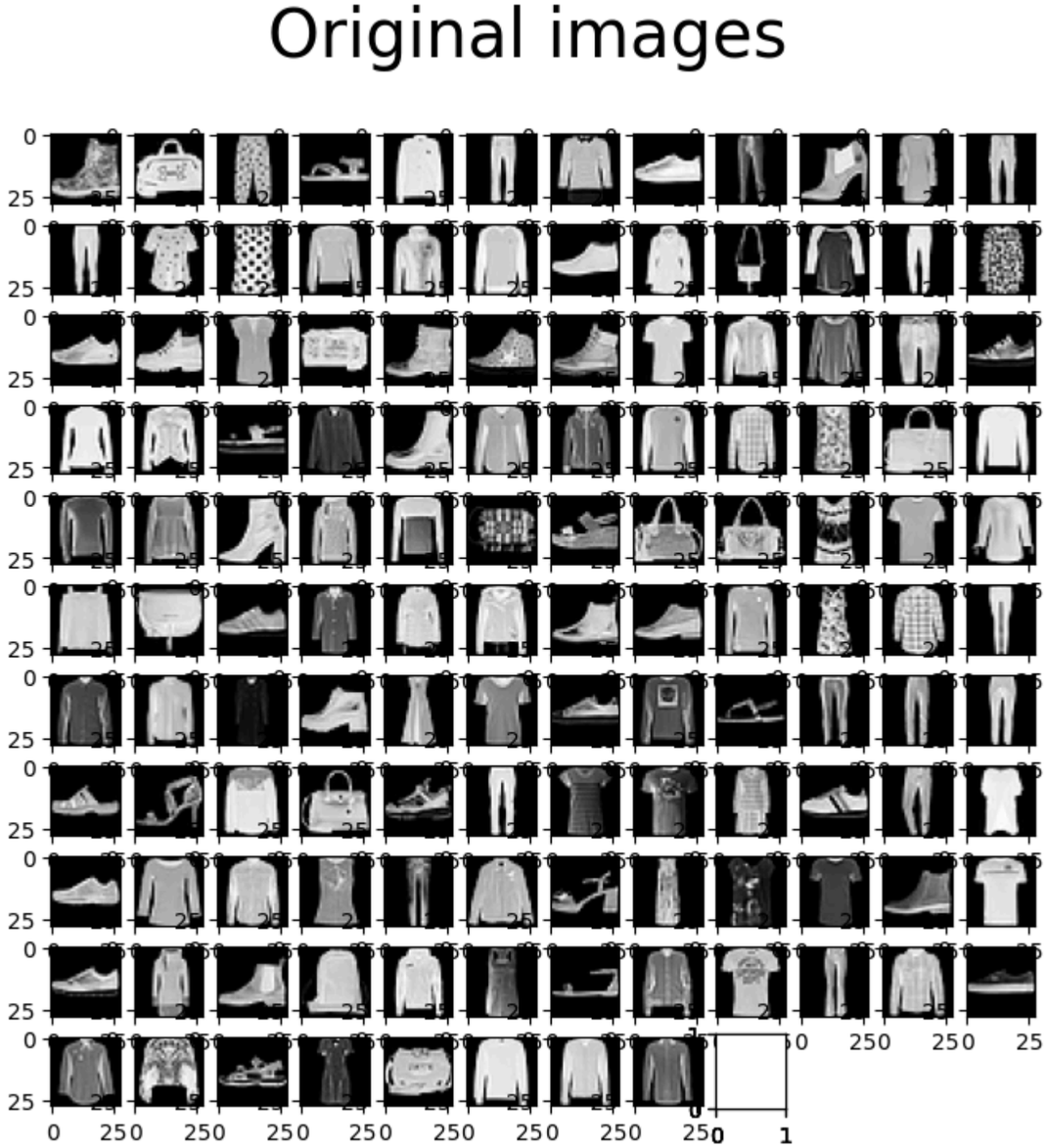
In [17]:
```
sum([p.numel() for p in ddpm.parameters()])
```

Out[17]:
```
606852
```

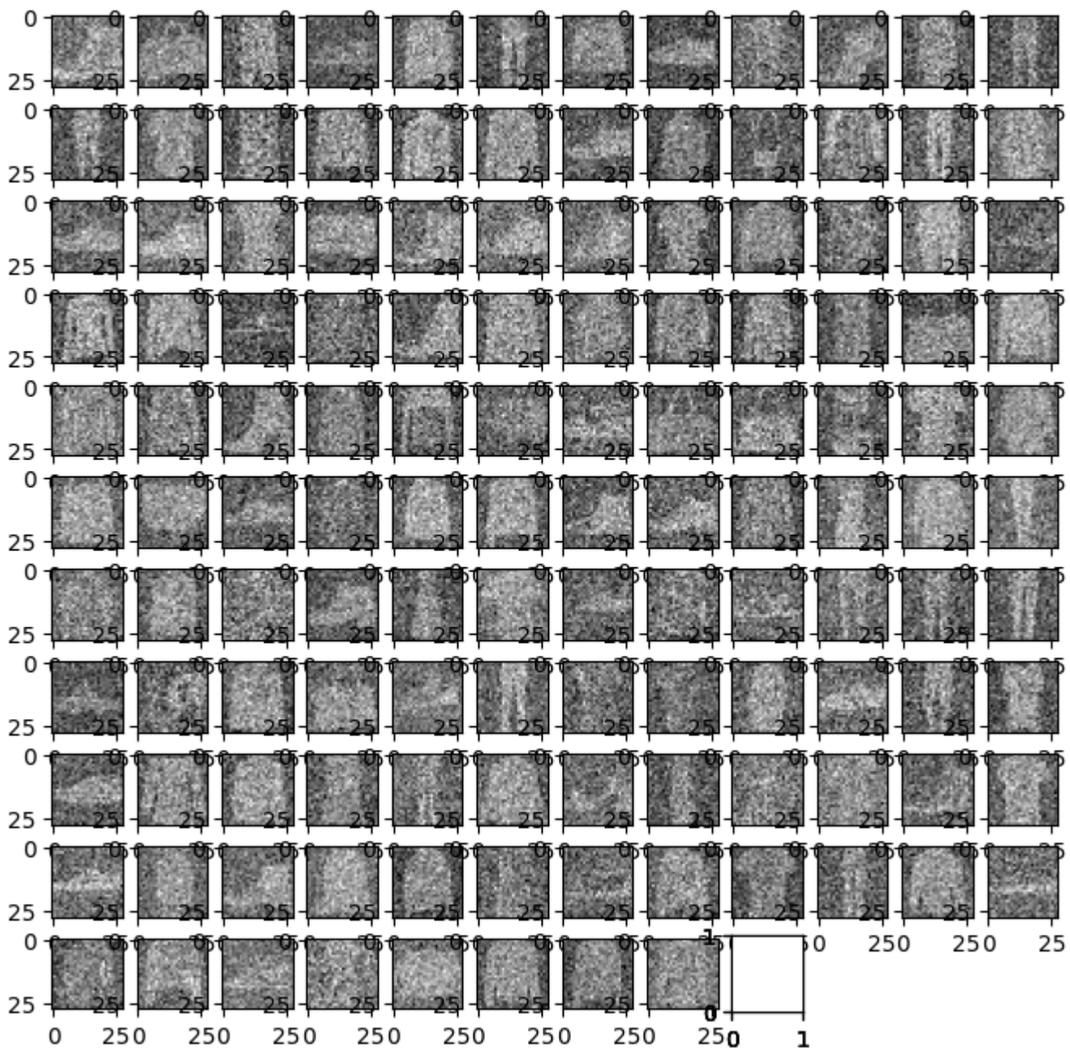
Visualize the Forward Process

In [18]:
```
# Load a pre-trained model that will be further trained
# ddpm.load_state_dict(torch.load(store_path, map_location=device))
```

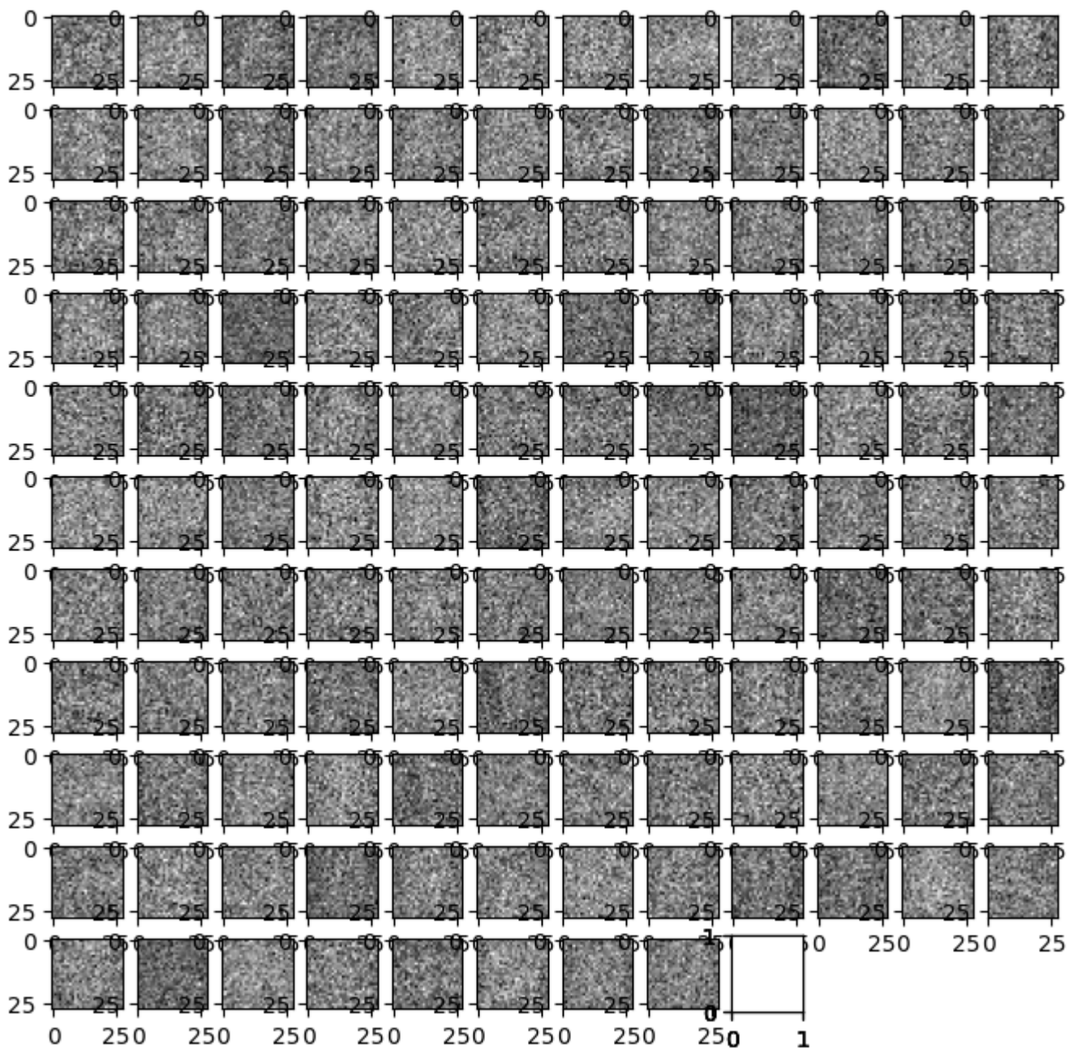
In [19]:
```
# Show the diffusion (forward) process
show_forward(ddpm, loader, device)
```


Original images

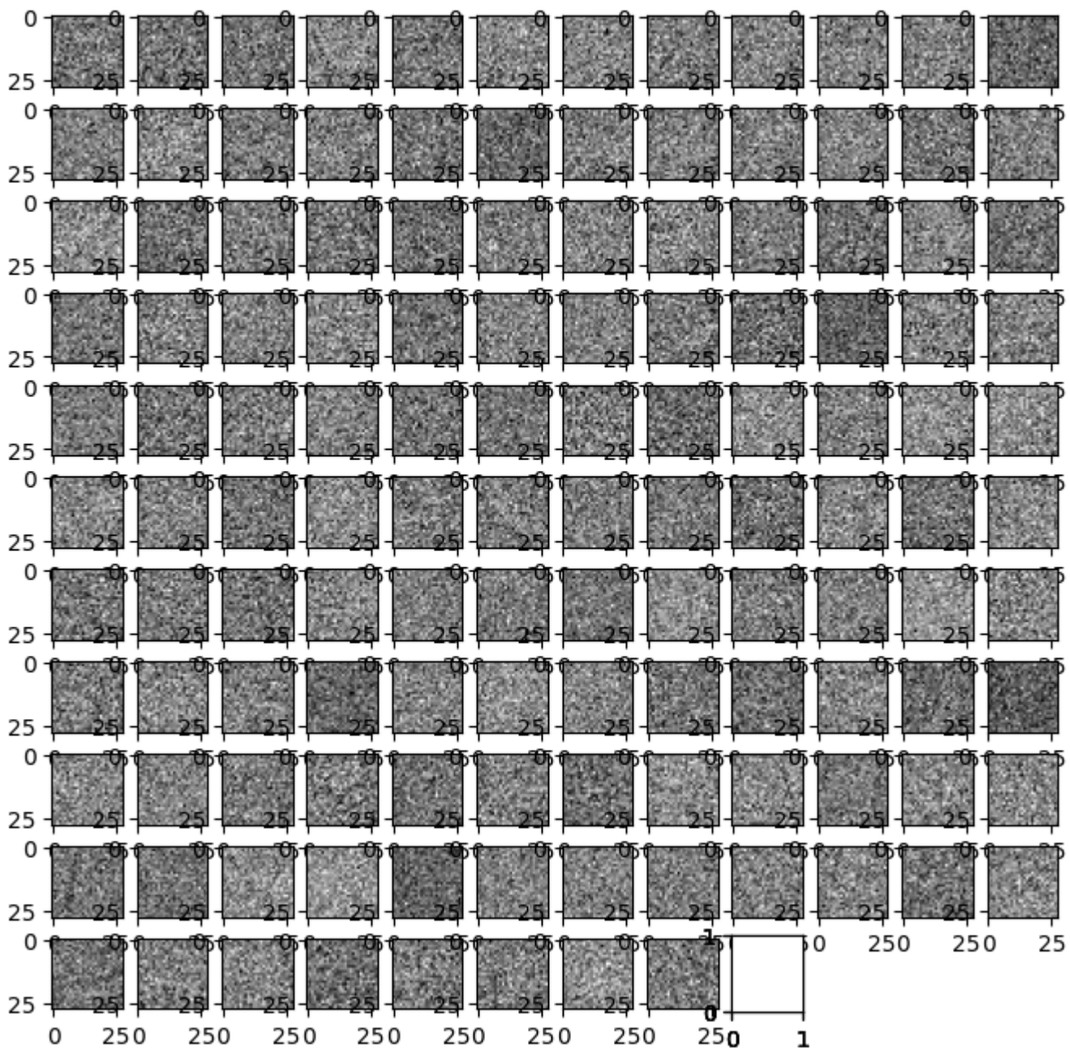
DDPM Noisy images 25%

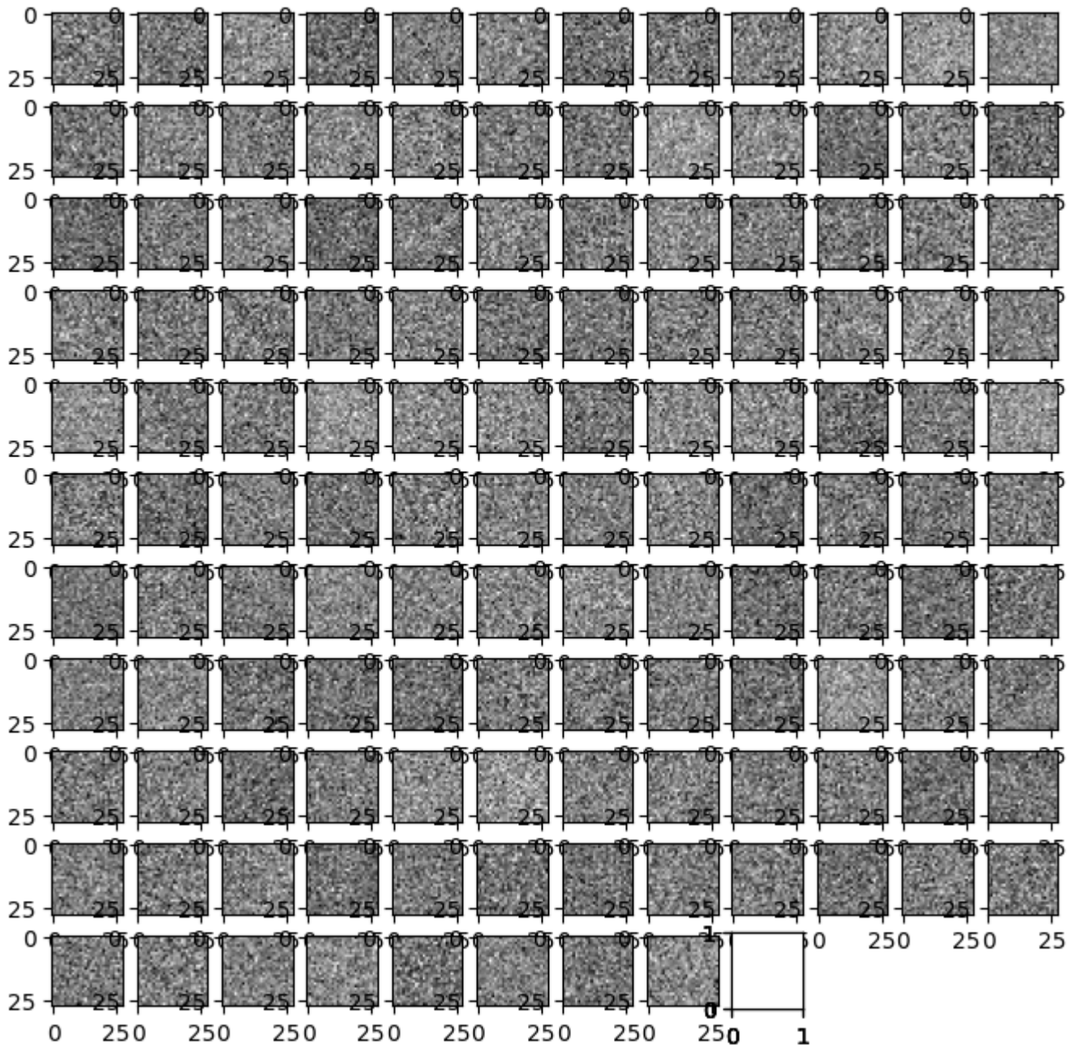


DDPM Noisy images 50%



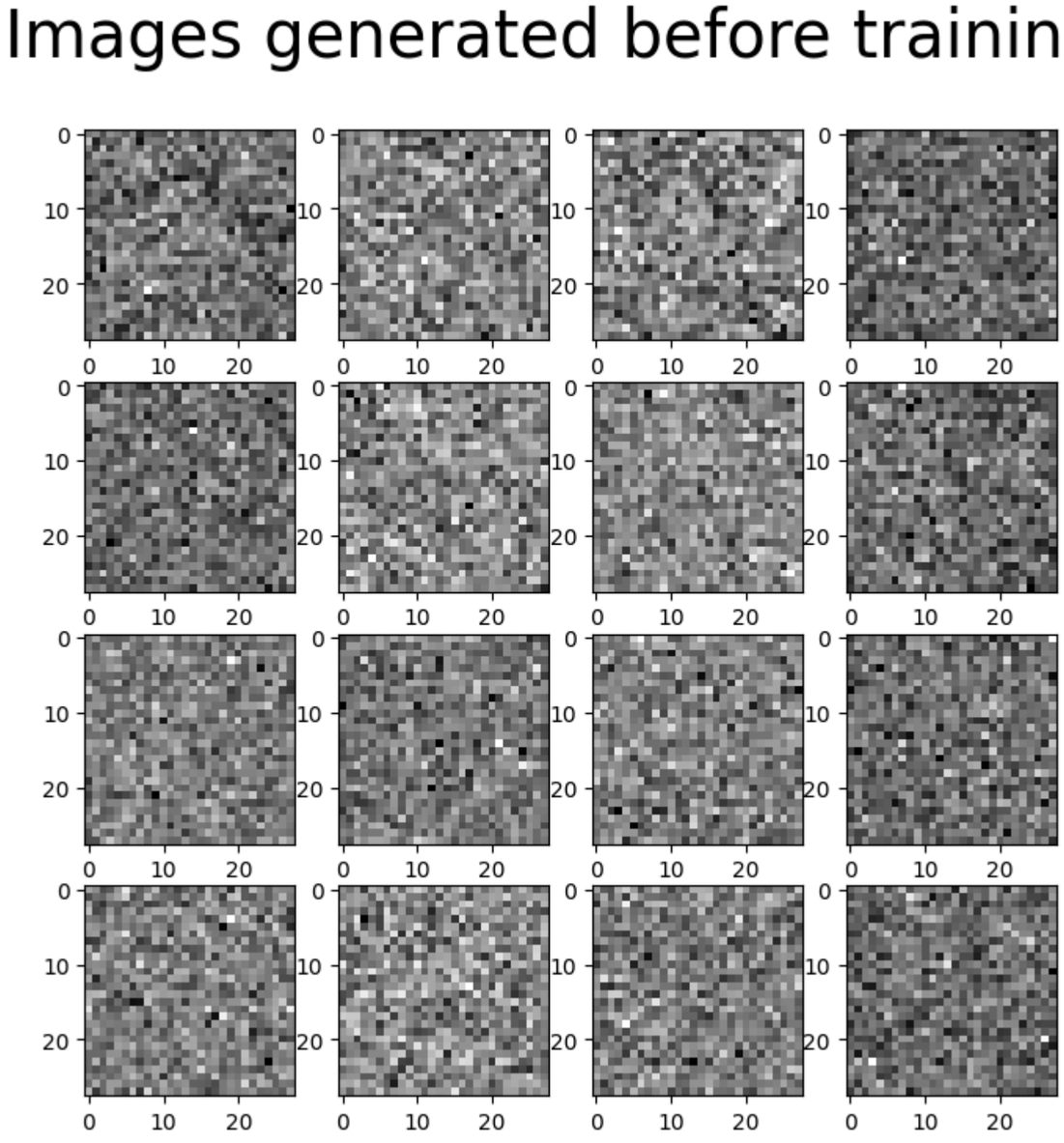
DDPM Noisy images 75%





In [20]:

```
# Show the images generated before training
generated = generate_new_images(ddpm, gif_name="before_training.gif")
show_images(generated, "Images generated before training")
```



Training Loop

The training loop is defined as follows. With each batch of our dataset, we run the forward process on the batch. We use a different timesteps t for each of the `N` images in our `(N, C, H, W)` batch tensor. The added noise is a `(N, C, H, W)` tensor ϵ.

Once we obtained the noisy images, we try to predict ϵ with our network. We optimize with a Mean-Squared Error (MSE) loss.

In [21]:
```
def training_loop(ddpm, loader, n_epochs, optim, device, display=False, store_path="ddpm_model.pt"):
    mse = nn.MSELoss()
    best_loss = float("inf")
    n_steps = ddpm.n_steps

    for epoch in tqdm(range(n_epochs), desc=f"Training progress", colour="#00ff00"):
        epoch_loss = 0.0
        for step, batch in enumerate(tqdm(loader, leave=False, desc=f"Epoch {epoch + 1}/{n_epochs}", colour="#005500")):
```

```
            # Loading data
            x0 = batch[0].to(device)
            n = len(x0)

            # Picking some noise for each of the images in the batch, a timestep and the respective alpha_bars
            eta = torch.randn_like(x0).to(device)
            t = torch.randint(0, n_steps, (n,)).to(device)

            # Computing the noisy image based on x0 and the time-step (forward process)
            noisy_imgs = ddpm(x0, t, eta)

            # Getting model estimation of noise based on the images and the time-step
            eta_theta = ddpm.backward(noisy_imgs, t.reshape(n, -1))

            # Optimizing the MSE between the noise plugged and the predicted noise
            loss = mse(eta_theta, eta)
            optim.zero_grad()
            loss.backward()
            optim.step()

            epoch_loss += loss.item() * len(x0) / len(loader.dataset)

        # Display images generated at this epoch
        if display:
            show_images(generate_new_images(ddpm, device=device), f"Images generated at epoch {epoch + 1}")

        log_string = f"Loss at epoch {epoch + 1}: {epoch_loss:.3f}"

        # Storing the model
        if best_loss > epoch_loss:
            best_loss = epoch_loss
            torch.save(ddpm.state_dict(), store_path)
            log_string += " --> Best model ever (stored)"

        print(log_string)
```

In [22]:

```
# Training
store_path = "ddpm_fashion.pt" if fashion else "ddpm_mnist.pt"
if not no_train:
    training_loop(ddpm, loader, n_epochs, optim=Adam(ddpm.parameters(), lr), device=device, store_path=store_path)
```

```
Loss at epoch 1: 0.204 --> Best model ever (stored)
Loss at epoch 2: 0.077 --> Best model ever (stored)
Loss at epoch 3: 0.067 --> Best model ever (stored)
Loss at epoch 4: 0.063 --> Best model ever (stored)
Loss at epoch 5: 0.059 --> Best model ever (stored)
Loss at epoch 6: 0.057 --> Best model ever (stored)
Loss at epoch 7: 0.056 --> Best model ever (stored)
Loss at epoch 8: 0.055 --> Best model ever (stored)
Loss at epoch 9: 0.054 --> Best model ever (stored)
Loss at epoch 10: 0.053 --> Best model ever (stored)
Loss at epoch 11: 0.052 --> Best model ever (stored)
Loss at epoch 12: 0.051 --> Best model ever (stored)
```

```
Loss at epoch 13: 0.051
Loss at epoch 14: 0.051 --> Best model ever (stored)
Loss at epoch 15: 0.050 --> Best model ever (stored)
Loss at epoch 16: 0.049 --> Best model ever (stored)
Loss at epoch 17: 0.050
Loss at epoch 18: 0.049 --> Best model ever (stored)
Loss at epoch 19: 0.049 --> Best model ever (stored)
Loss at epoch 20: 0.048 --> Best model ever (stored)
```

Testing the trained model

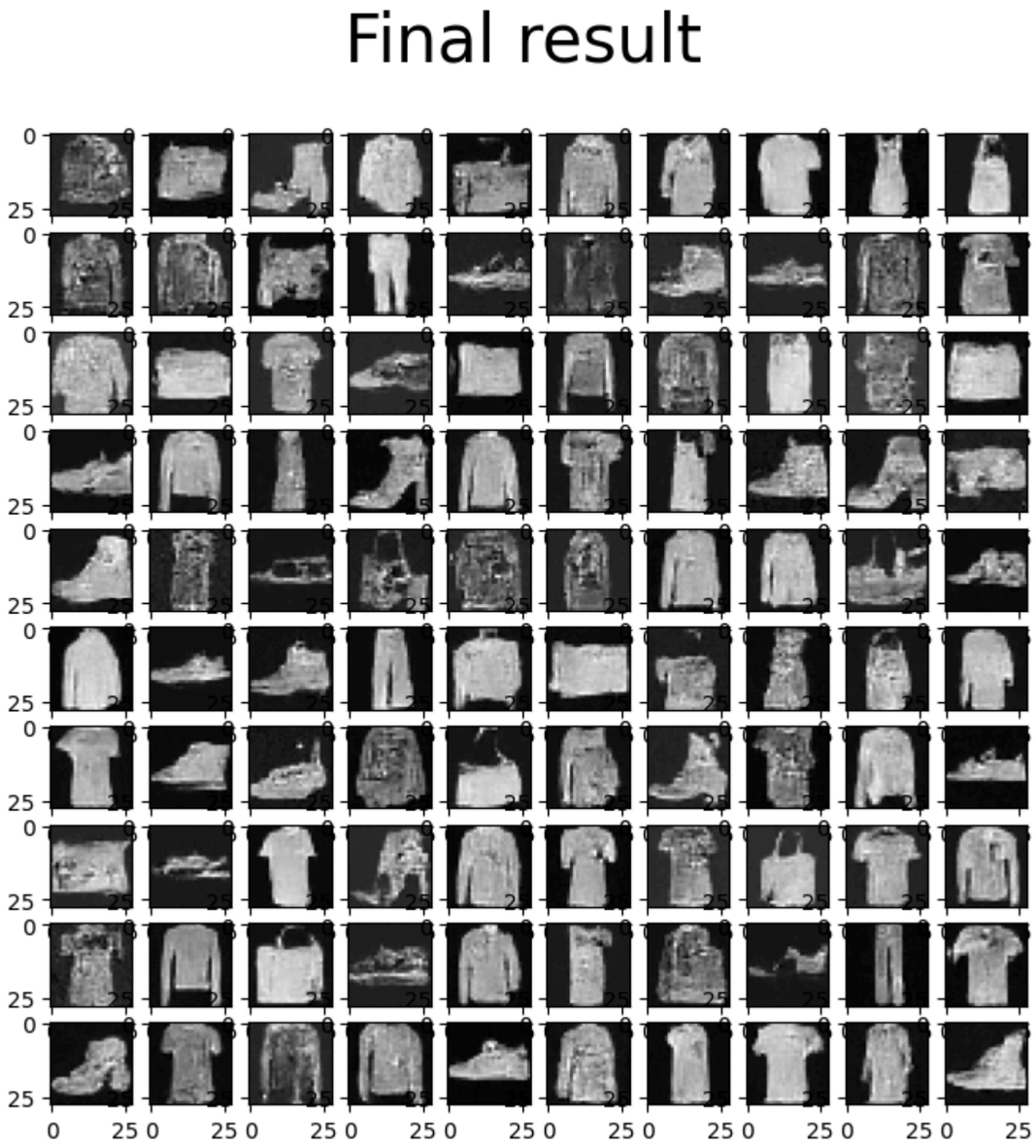
It's time to check how well our model does. We restore the best performing model (according to the training loss) and set it to evaluation mode. Finally, we display a batch of generated images and the relative denoising process.

In [23]:
```
# Loading the trained model
best_model = MyDDPM(MyUNet(), n_steps=n_steps, device=device)
best_model.load_state_dict(torch.load(store_path, map_location=device))
best_model.eval()
print("Model loaded")
```

```
Model loaded
<ipython-input-23-674fc429e284>:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  best_model.load_state_dict(torch.load(store_path, map_location=device))
```

In [24]:
```
print("Generating new images")
generated = generate_new_images(
        best_model,
        n_samples=100,
        device=device,
        gif_name="fashion.gif" if fashion else "mnist.gif"
    )
show_images(generated, "Final result")
```

```
Generating new images
```

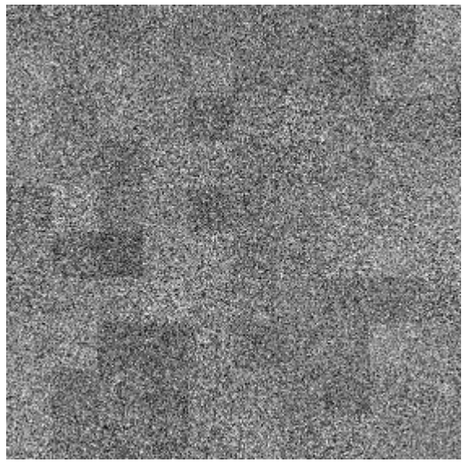


Visualizing the Denoising Process

In [25]:
```
from IPython.display import Image

Image(open('fashion.gif' if fashion else 'mnist.gif','rb').read())
```

Out[25]:



3. Conclusion

In this notebook, we implemented a DDPM from scratch. We used a custom UNet architecture and the sinusoidal positional-embedding technique to condition the denoising process of the network on the particular time-step. We trained the model on the MNIST / Fashion-MNIST dataset. In only 20 training epochs (10 minutes using a Tesla T4 GPU), we were able to generate new samples for these toy datasets.