Manga character generation with DDPM

In this exercise, we will learn to use Denoising Diffusion Probabilistic Models (DDPM) to generate manga-like faces.

Diffusion models are generative models that can generate high-quality data (much better than VAEs) without the instability and training complications of GANs. Their main drawback is that the generation process is expensive, requiring multiple steps through a deep neural network.

Diffusion models are the main component of many modern systems for generative AI. Here is a short list of widely known projects using them:

- DALL-E 2 (by OpenAI);
- Stable Diffusion (by Stability AI);
- Imagen (by Google Research);
- Sora (by OpenAI).

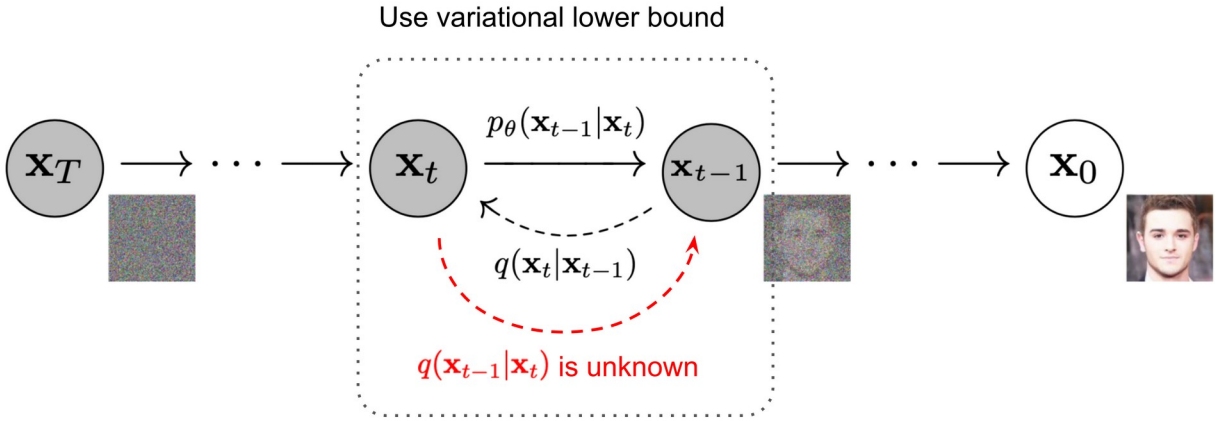
Suggested readings

The following are good introductions to the topic:

- What are diffusion models? by Lilian Wang;
- Understanding Diffusion Models: A Unified Perspective by Calvin Luo;
- Annotated diffusion: a step by step explanation of the source code;
- Denoising Diffusion Probabilistic Models by Ho et al. (original paper on DDPM).

Denoising Diffusion Probabilistic Models (DDPM)

DDPM model a stochastic process in which noise is gradually added to a starting sample x_0. The forward process $x_0 \to x_1 \to \ldots \to x_T$ is modeled exactly. Then, the reverse process $x_T \to x_{T-1} \to \ldots \to x_0$ is approximated thanks to a deep neural network trained for denoising.



Note that, if we take T large enough, x_T can be assumed to be pure noise, and we can take samples of it. The result of the reverse process is that we will be able to sample x_0 as if it was coming from the same unknown distribution represented by the training set.

Details about the two processes will be given during the exercise.

Setup

Let's start by importing the libraries, and by unpacking the training data into the Colab space. Remeber to set the `PROJECT_DIR` so that it points to the folder where you placed the data.

```
import torch
import torch.nn.functional as F
import torchvision
import matplotlib.pyplot as plt
import google.colab
import zipfile

google.colab.drive.mount('/content/drive')
PROJECT_DIR = "/content/drive/MyDrive/"
zip_ref = zipfile.ZipFile(PROJECT_DIR + "manga.zip", 'r')
zip_ref.extractall(".")
zip_ref.close()
print("OK")

DEVICE = ("cuda" if torch.cuda.is_available() else "cpu")
print(DEVICE)
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
OK
cpu
```

Dataset

The training images are a subset of the dataset published here. In total, there are 12000 images, each one already scaled down to 64×64 pixels. The simple `ImageFolder` class in the `torchvision` library can be used to load the dataset. The dataset will pair the images with a useless label, which we can just ignore.

To reduce computation we will further reduce the size of the images to 32×32 pixels. If you have more time, you can try also with the original size.

```
HEIGHT = WIDTH = 32
BATCH_SIZE = 32

transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize((HEIGHT, WIDTH)),
    torchvision.transforms.RandomHorizontalFlip(),
```

```
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(0.5, 0.5)
])

# Define the dataset and the data loader.
train_set = torchvision.datasets.ImageFolder("manga/train",
transform=transform)
loader = torch.utils.data.DataLoader(train_set,
batch_size=BATCH_SIZE,shuffle=True, drop_last=True, num_workers=2)

def show_faces(faces):
    old = plt.rcParams['figure.dpi']
    plt.rcParams['figure.dpi'] = 150
    g = torchvision.utils.make_grid(faces, normalize=True,
value_range=(-1, 1))
    plt.imshow(g.permute(1, 2, 0).detach().cpu())
    plt.axis("off")
    plt.show()
    plt.rcParams['figure.dpi'] = old

batch, labels = next(iter(loader))
show_faces(batch)
```

Forward diffusion process

During training we will use the forward process to add noise to training images. This is done in multiple time steps. At time $t=0$ we have the original image without noise. At the last time step $t=T$ the image is replaced by pure Gaussian noise with zero mean and unit variance. To get to that point we combine the image with a small amount of Gaussian noise:

$$x_t=\sqrt{1-\beta_t}\,x_{t-1}+\sqrt{\beta_t}\epsilon$$

where ϵ is drawn from a Gaussian with zero mean and unit variance. The scalar parameters $\beta_1,\beta_2,\ldots,\beta_{T-1}$ define the "variance schedule" of the process. In fact, adding the same amount of

noise at each step is not optimal. Instead, it is more efficient to add more noise close to the end of the forward process, when the image is already very noisy.

In the following it will be useful to define also:

$$\alpha_t = 1 - \beta_t, \acute{\alpha}_t = \prod_{i=1}^{t} \alpha_t.$$

Therefore,

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon$$

```
TIMESTEPS = 200 #100 #300 #500
beta = torch.linspace(0.0001, 0.02, TIMESTEPS, device=DEVICE)
alpha = 1 - beta
alpha_bar = torch.cumprod(alpha, axis=0)
```

Gaussians are nice distributions

The sum of two Gaussian variables is another Gaussian variable, with the sum of means as mean, and the sum of variances as variance.

$$x_t \sim N\left(\mu_t, \sigma_t^2\right) \Longrightarrow \sum_t x_t \sim N\left(\sum_t \mu_t, \sum_t \sigma_t^2\right)$$

.

Thanks to this property, we can sample x_t by skipping intermediate steps (this will be used for training):

$$\begin{array}{ll} x_t & ¿\sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon \\ ¿ & ¿\ldots \\ ¿ & ¿\sqrt{\acute{\alpha}_t} x_0 + \sqrt{1-\acute{\alpha}_t}\epsilon. \end{array}$$

Note that each step includes its own noise ϵ, but they can be collapsed into a single random variable

```
# Sample xt from x0.  When not given, it must be generated with
`torch.randn_like`.
# t can be an integer, or a vector of indices.
def q_sample(x0, t, noise=None):
    t = t.expand(x0.shape[0])
    if noise is None:
        noise = torch.randn_like(x0)
    # Complete using the formula above
    abar=alpha_bar[t].view(-1,1,1,1)
    xt = torch.sqrt(abar)* x0 + torch.sqrt(1-abar) * noise
    return xt
```

```
x = batch[0:1].to(DEVICE)
t = torch.linspace(0, TIMESTEPS - 1, 64).long()
x_noisy = q_sample(x.repeat(t.shape[0], 1, 1, 1), t)
show_faces(x_noisy)
```

The neural network

To reverse the process we need to sample x_{t-1} from x_t. This computation is not tractable, and we will approximate it with a neural network. More precisely, the neural network will take x_t and t as input, and will estimate the noise ϵ (last equation of the previous paragraph):

$$\epsilon \sim \epsilon_\theta(x_t, t).$$

We will use the same architecture described in the original paper. It is a Unet-like CNN with residual and skip connections, and with attention mechansim. It is provided in the `unet.py` module.

```
import unet

# Parameters:
# - number of features;
# - multipliers for features at each stage;
# - number of channels.
model = unet.Unet(16, (1, 2, 4, 8), 3)
model.to(DEVICE)
ps = sum(p.numel() for p in model.parameters())
```

```
print(f"{ps:,d} parameters")
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

def save_model(filename):
    data = {
        "model": model.state_dict(),
        "optimizer": optimizer.state_dict()
    }
    torch.save(data, PROJECT_DIR + filename)

def load_model(filename):
    data = torch.load(PROJECT_DIR + filename)
    model.load_state_dict(data["model"])
    optimizer.load_state_dict(data["optimizer"])
```

```
2,430,595 parameters
```

Training algorithm

The training algorithm for DDPM is very simple: we take a batch of training samples, we pick random time indices, we generate the noisy samples, use the network to estimate the noise, and compare the actual and estimated noise.

The derivation of the loss function is cumbersome. Empirically, the following simplified loss works well:

$$L = \| \epsilon - \epsilon_\theta(x_t, t) \|^2.$$

This is the algorithm taken from the original paper.

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

We will introcue the small variation of using the L1 loss instead of L2.

```
EPOCHS = 20

step = 0
for epoch in range(EPOCHS):
    for x0, labels in loader:
        optimizer.zero_grad()
        x0 = x0.to(DEVICE)

        # Algorithm 1 line 3: sample t uniformally for every example
in the batch
        t = torch.randint(0, TIMESTEPS, (BATCH_SIZE,),
device=DEVICE).long()

        # Complete the algorithm, as described above.
        noise=torch.randn_like(x0)
        xt= q_sample(x0,t,noise)
        predicted_noise= model(xt,t)
        loss=F.l1_loss(noise, predicted_noise)
        loss.backward()
        optimizer.step()

        step += 1
        if step % 100 == 0:
            print(f"{step} [{epoch}] {loss.item():.4f}")

    save_model("manga-ddpm-tmp.pt")
save_model("manga-ddpm.pt")
```

```
100 [0] 0.2375
200 [0] 0.2790
300 [0] 0.1836
400 [1] 0.2382
500 [1] 0.2226
600 [1] 0.2171
700 [1] 0.2310
800 [2] 0.1872
900 [2] 0.1794
1000 [2] 0.1958
1100 [2] 0.1748
1200 [3] 0.1906
1300 [3] 0.2094
1400 [3] 0.2066
1500 [3] 0.2113
1600 [4] 0.1595
1700 [4] 0.1796
1800 [4] 0.1772
1900 [5] 0.1875
2000 [5] 0.1916
2100 [5] 0.2136
2200 [5] 0.1741
```

```
2300 [6] 0.1867
2400 [6] 0.1726
2500 [6] 0.2256
2600 [6] 0.1892
2700 [7] 0.1788
2800 [7] 0.1867
2900 [7] 0.1724
3000 [7] 0.1937
3100 [8] 0.2166
3200 [8] 0.1811
3300 [8] 0.1612
3400 [9] 0.1722
3500 [9] 0.1993
3600 [9] 0.1833
3700 [9] 0.1757
3800 [10] 0.2159
3900 [10] 0.1988
4000 [10] 0.1796
4100 [10] 0.2141
4200 [11] 0.2067
4300 [11] 0.1985
4400 [11] 0.1914
4500 [11] 0.1956
4600 [12] 0.1824
4700 [12] 0.2155
4800 [12] 0.1653
4900 [13] 0.1746
5000 [13] 0.1459
5100 [13] 0.1832
5200 [13] 0.2041
5300 [14] 0.2296
5400 [14] 0.1859
```

Sampling

After training, we are ready to generate new images.

We start from x_T sampled from a Gaussian distribution, and sample backward to x_0.

The sampling algorithm is:

Algorithm 2 Sampling

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

Note that $z_\theta(x_t, t)$ is the estimated noise for the full $x_0 \to x_t$ process, and it must be properly scaled to obtain x_{t-1}. The backward process also requires a seprate variance schedule σ_t^2. Here we will use the following:

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

Note that in the last step the process has no noise.

```
load_model("manga-ddpm.pt")

sigma = beta * (1 - alpha_bar / alpha) / (1 - alpha_bar)
sigma[0] = 0

@torch.no_grad()
def p_sample(model, x, t):
    b = beta[t].view(-1, 1, 1, 1)
    a = alpha[t].view(-1, 1, 1, 1)
    abar = alpha_bar[t].view(-1, 1, 1, 1)
    s = sigma[t].view(-1, 1, 1, 1)

    # Implement the formula
    model_mean = (x-b* model(x,t)/ torch.sqrt(1-abar)) / torch.sqrt(a)
    x_prev = model_mean + torch.randn_like(x) * torch.sqrt(s)
    return x_prev

def sample(model, shape):
    b = shape[0]
    img = torch.randn(shape, device=DEVICE)
    imgs = []
```

```
    for i in range(TIMESTEPS - 1, -1, -1):
        t = torch.full((b,), i, device=DEVICE).long()
        img = p_sample(model, img, t)
        imgs.append(img.cpu())
    return imgs

generated = sample(model, (BATCH_SIZE, 3, HEIGHT, WIDTH))
show_faces(generated[199])

#Question 2 version (disabling xT)
load_model("manga-ddpm.pt")

sigma = beta * (1 - alpha_bar / alpha) / (1 - alpha_bar)
sigma[0] = 0

@torch.no_grad()
def p_sample(model, x, t):
    b = beta[t].view(-1, 1, 1, 1)
    a = alpha[t].view(-1, 1, 1, 1)
    abar = alpha_bar[t].view(-1, 1, 1, 1)
    s = sigma[t].view(-1, 1, 1, 1)

    # Implement the formula
    model_mean = (x-b* model(x,t)/ torch.sqrt(1-abar)) / torch.sqrt(a)
    x_prev = model_mean + torch.randn_like(x) * torch.sqrt(s)
    return x_prev
def sample_fixed_xt(model, shape):
    b = shape[0]
    img = torch.zeros(shape, device=DEVICE)  # Fixed initial x_T
    imgs = []

    for i in range(TIMESTEPS - 1, -1, -1):
        t = torch.full((b,), i, device=DEVICE).long()
        img = p_sample(model, img, t)
        imgs.append(img.cpu())
    return imgs

fixed_xt_generated = sample_fixed_xt(model, (BATCH_SIZE, 3, HEIGHT,
WIDTH))
show_faces(fixed_xt_generated[199])


#Question 2 version (disabling step noise z)
load_model("manga-ddpm.pt")

sigma = beta * (1 - alpha_bar / alpha) / (1 - alpha_bar)
sigma[0] = 0

@torch.no_grad()
def p_sample_no_step_noise(model, x, t):
```

```
    b = beta[t].view(-1, 1, 1, 1)
    a = alpha[t].view(-1, 1, 1, 1)
    abar = alpha_bar[t].view(-1, 1, 1, 1)
    # No step noise added
    model_mean = (x - b * model(x, t) / torch.sqrt(1 - abar)) /
torch.sqrt(a)
    return model_mean  # No noise term

def sample_no_step_noise(model, shape):
    b = shape[0]
    img = torch.randn(shape, device=DEVICE)
    imgs = []

    for i in range(TIMESTEPS - 1, -1, -1):
        t = torch.full((b,), i, device=DEVICE).long()
        img = p_sample_no_step_noise(model, img, t)
        imgs.append(img.cpu())
    return imgs

no_step_noise_generated = sample_no_step_noise(model, (BATCH_SIZE, 3,
HEIGHT, WIDTH))
show_faces(no_step_noise_generated[199])
```

Questions

In the report, answer to the following questions:

- Try changing the number of time steps. What happens? Guess advantages and disadvantages of increasing/reducing that number.
- The reverse process has two source of randomness: the initial x_T and the step noise z. Disable them one at a time, and make a hypothesis about their role in the process.