COMP3547: Deep Learning Practical 7

Diffusion probabilistic modeling

amir.atapour-abarghouei@durham.ac.uk

1 Overview

Welcome to the seventh deep learning practical. This is the last practical with set material. In this practical, we will be implementing a minimal denoising diffusion-based probabilistic model from scratch. The structure for this practical is as follows:

- The forward process.
- The reverse process.
- U-Net with lazy layers.
- Conditional diffusion.

The solutions in this practical are kept minimal for educational purposes; in practice, for the best results, more expressive architectures and more sophisticated conditioning strategies are used, and they are trained for longer.

2 Setup

It is recommended to use a GPU (either Colab or NCC). If you are unable to allocate a GPU, you will still be able to complete the practical, but you won't have time to see any training results. Before we begin, please use the following setup code:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import matplotlib.pyplot as plt
import einops

from torchvision import transforms
from einops import rearrange

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

assert torch.zeros(32).to(device).device.type=='cuda' # check cuda is working

# helper functions
def cycle(iterable):
    while True:
        for x in iterable:
            yield x

def normalise(x):
    return (x-x.min())/(x.max()-x.min())

# MNIST dataloader (we set MNIST to Bx3x32x32 so our code works the same later for CIFAR10)
train_loader = torch.utils.data.DataLoader(torchvision.datasets.MNIST("./data", train=True,
    download=True,transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize((32,32)), transforms.Lambda(lambda x: x.repeat(3,1,1)),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])), batch_size=16, shuffle=True)

train_iterator = iter(cycle(train_loader))
```

3 The forward process

As described in section two of the 'denoising diffusion probabilist models' paper (Jonathan Ho et al.), the *forward process* also called the *diffusion process* is fixed to a Markov chain that gradually adds noise to the data according to a variance $\beta_1, ..., \beta_T$ schedule of T steps. The network $\boldsymbol{\epsilon}_\theta$ is trained to estimate the noise $\boldsymbol{\epsilon}$ that data $\mathbf{x}$ is corrupted with, from random points on the variance schedule. This is the full algorithm:

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

Exercise

In line 2, sample a batch of data from the data distribution $\mathbf{x} \sim q(\mathbf{x})$. Also, for now, sample labels $\mathbf{c}$ from the dataset $\mathbf{x}, \mathbf{c} \sim q(\mathbf{x}, \mathbf{c})$ as later we will use these in an extension.

```
x,c = next(train_iterator)
x,c = x.to(device), c.to(device)
```

In line 3, specify where we are in time on the variance schedule uniformly randomly (do this for each item in the batch and send the results to the GPU device). Use $T = 400$.

```
T = 400
_ts = torch.randint(1, T+1, (x.shape[0],)).to(device)
```

In line 4, sample a tensor of noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. This noise tensor should be of the same rank and size as $\mathbf{x}$.

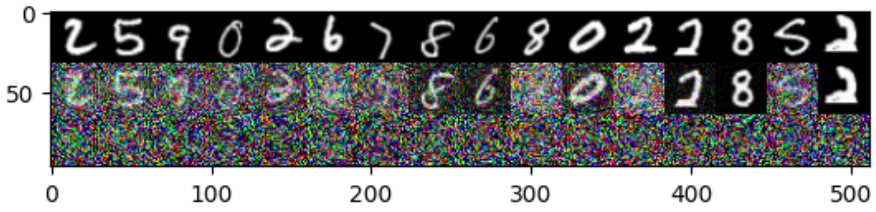
```
eps = torch.randn_like(x)
```

In line 5, you will need the variance schedule $\sqrt{\bar{\alpha}_t}$ and $\sqrt{1-\bar{\alpha}_t}$. The calculation of these schedules, as defined in the paper, is a little involved and not worth going into for now (visualise them with `plt.plot(x.cpu())`, so to keep things simple precompute them as follows:

```
beta1 = 1e-4
beta2 = 0.02
T = 400

beta_t = (beta2 - beta1) * torch.arange(0, T + 1, dtype=torch.float32, device=device) / T + beta1
alpha_t = 1 - beta_t
log_alpha_t = torch.log(alpha_t)
alphabar_t = torch.cumsum(log_alpha_t, dim=0).exp()

sqrtab = torch.sqrt(alphabar_t).view(-1,1,1,1)
sqrtmab = torch.sqrt(1 - alphabar_t).view(-1,1,1,1)
```

In line 5, calculate the inputs for the neural network and show a grid of images with three rows. On the first row show the original data $\mathbf{x}$. On the second row, show the batch of partially diffused data (input to the network). On the last row, show the noise $\boldsymbol{\epsilon}$ that the network tries to predict.

```
# network inputs
x_t = (sqrtab[_ts] * x + sqrtmab[_ts] * eps)
_ts/T

# data is in range -1 to +1 so we have to correct this
img_grid = rearrange(torch.cat((x*0.5+0.5, x_t*0.5+0.5, eps), dim=0),
    '(b1 b2) c h w -> (b1 h) (b2 w) c', b1=3)

plt.imshow((img_grid.cpu()*255).int().numpy())
```



Now create a DDPM class that extends `nn.Module`. Give it a `def forward(self, x)` function that returns the loss `F.mse_loss(eps, self.net(x_t))` (we are not going to also inject t into the network just yet). Initialise it with a very simple neural network `self.net` for now, e.g. `nn.Sequential` with a simple `3,1,1` pattern 2D convolution with 64 output channels, batch normalisation, a `nn.ReLU` layer, then a final `3,1,1` convolution with 3 output channels. This means that the neural network `self.net` must output a tensor of the same rank and dimension as the input data.

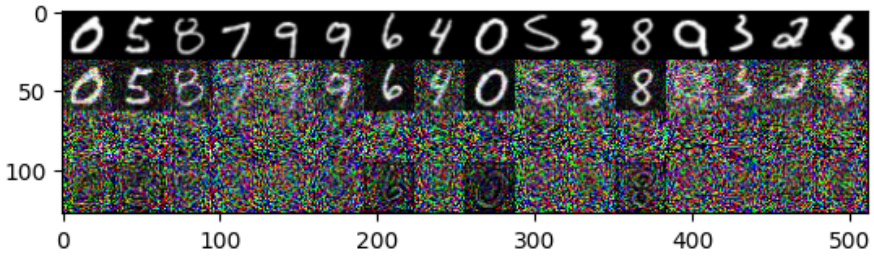
```
class DDPM(nn.Module):
    def __init__(self,):
        super(DDPM, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,stride=1,padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=3, kernel_size=3,stride=1,padding=1)
        )

    # algorithm 1 in DDPM paper
    def forward(self, x):
        _ts = torch.randint(1, T+1, (x.shape[0],)).to(x.device)
        eps = torch.randn_like(x)
        x_t = (sqrtab[_ts] * x + sqrtmab[_ts] * eps)
        return F.mse_loss(eps, self.net(x_t))
```

Lastly, finish writing Algorithm 1 by implementing the outer training loop and optimising the neural network. Use Adam with `lr=1e-4`. Train the algorithm for 2,000 batches of data and print the loss values (better with some smoothing).

```
# train the model
ddpm.train()

while(step < 2000):

    x,c = next(train_iterator)
    x,c = x.to(device), c.to(device)

    loss = ddpm(x)
    optim.zero_grad()
    loss.backward()
    optim.step()

    loss_smooth = loss_smooth + 0.01 * (loss.item() - loss_smooth)
    if step % 100 == 0:
        print('step: {:4d} train loss: {:.3f}'.format(step, loss_smooth))

    step += 1
```

Now display a grid of images as before, but this time add a fourth row displaying the output for your model after training `ddpm.net(x_t)`. What do you expect to see? You may want to compare the outputs before—and after—training accordingly.

```
_ts = torch.randint(1, T+1, (x.shape[0],)).to(x.device)
eps = torch.randn_like(x)
x_t = (sqrtab[_ts] * x + sqrtmab[_ts] * eps)

img_grid = rearrange(torch.cat((x*0.5+0.5, x_t*0.5+0.5, eps, ddpm.net(x_t)), dim=0),
    '(b1 b2) c h w -> (b1 h) (b2 w) c', b1=4)

plt.imshow((img_grid.cpu()*255).int().numpy())
```



The network learns to estimate the noise. With a network this small, and after such a small amount of training, it won't always be perfect.

4 The reverse process

We have now learned the forward diffusion process. After training, we can reverse this process—starting with noise and gradually denoising—to sample the network (a generative model). This is the algorithm we will implement:

Algorithm 2 Sampling

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

As before, the variance schedule terms are a little involved so they are provided and can be precomputed:

```
oneover_sqrta = 1 / torch.sqrt(alpha_t)
mab_over_sqrtmab_inv = (1 - alpha_t) / sqrtmab.squeeze()
sigma_t = torch.sqrt(beta_t)
```

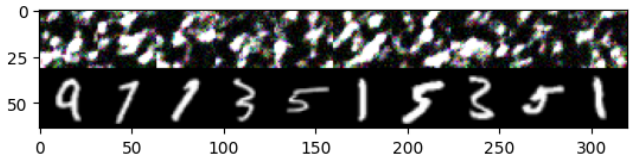
Implement the `sample(self, n_sample, size)` function within your DDPM class. To allow more time for experimentation, here is the implementation:

```
def sample(self, n_sample, size):
    x_i = torch.randn(n_sample, *size).to(device)
    for i in range(T, 0, -1):
        z = torch.randn(n_sample, *size).to(device) if i > 1 else 0
        eps = self.net(x_i)
        x_i = (oneover_sqrta[i] * (x_i - eps * mab_over_sqrtmab_inv[i]) + sigma_t[i] * z)
    return x_i
```

After training, you can therefore sample the model with:

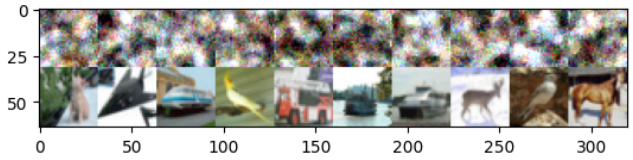
```
# sample the model
ddpm.eval()

with torch.no_grad():
    xh = ddpm.sample(10, (3, 32, 32))
    img_grid = rearrange(torch.cat([(xh*0.5+0.5), (x*0.5+0.5)[:10]], dim=0),
        '(b1 b2) c h w -> (b1 h) (b2 w) c', b1=2)
    plt.imshow((img_grid.cpu()*255).int().numpy())
```

That was a lot of work. So what is the sample quality like?



...I mean if you squint at them, they kind of look like MNIST. No—who am I kidding? Those samples are terrible. Lets swap out the data loader and try with CIFAR10 and see if they are any better:

```
# CIFAR10 dataloader
train_loader = torch.utils.data.DataLoader(torchvision.datasets.CIFAR10("./data",
    train=True,download=True,transform=transforms.Compose([
        transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])), batch_size=16, shuffle=True)

train_iterator = iter(cycle(train_loader))
```



As you can see, the CIFAR10 samples are indeed closer to the CIFAR10 distribution than the MNIST samples are to the MNIST data distribution.

The main problem (besides only training for a very short amount of time) is that the network with two convolutional layers isn't expressive enough.

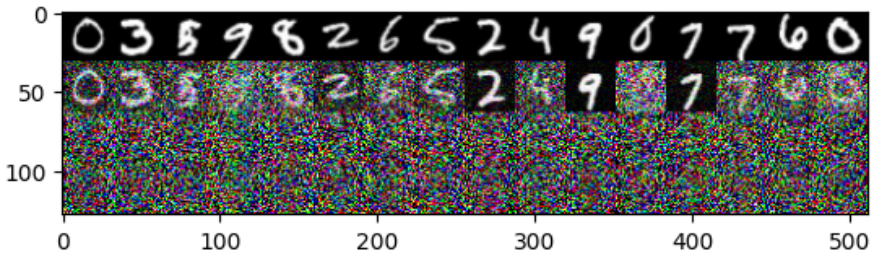
5 U-Net with lazy layers

Inside the DDPM class, replace the pathetic `self.net = nn.Sequential(...)` with `self.net = UNet()`, and create a new `class UNet(nn.Module)`.

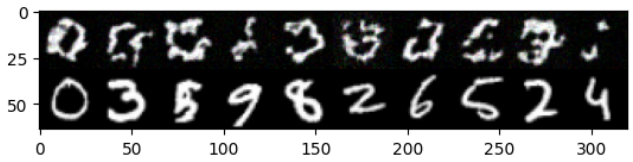
Implement a UNet with `4,2,1` patterns and four skip connections. You could have the number of output channels as [72, 96, 128, 196, 256] in the encoder. Before coding, read about `nn.LazyConv2d` and `nn.LazyTransposeConv2d` and `nn.LazyLinear` from the PyTorch documentation. Essentially, if you use these layers as drop-in replacements for their counterparts, you don't need to specify the number of input channels; it just figures it out. If you have time, try to design this yourself—otherwise copy this version I made:

```
class UNet(nn.Module):
    def __init__(self, out_class=3):
        super().__init__()
        self.dconv_down1 = self.conv_block(72)
        self.dconv_down2 = self.conv_block(96)
        self.dconv_down3 = self.conv_block(128)
        self.dconv_down4 = self.conv_block(196)
        self.dconv_down5 = self.conv_block(256)
        self.bottleneck  = nn.LazyConv2d(256, 3,1,1, bias=False)
        self.dconv_up4 = self.conv_block(256, up=True)
        self.dconv_up3 = self.conv_block(196, up=True)
        self.dconv_up2 = self.conv_block(128, up=True)
        self.dconv_up1 = self.conv_block(96, up=True)
        self.conv_last = nn.LazyConv2d(out_class, 1)

    def conv_block(self, out_channels, up=False):
        ConvType = nn.LazyConvTranspose2d if up else nn.LazyConv2d
        return nn.Sequential(
            ConvType(out_channels, 4, 2, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU())

    def forward(self, x):
        start = x
        conv1 = self.dconv_down1(x)     # skip-connection 1
        conv2 = self.dconv_down2(conv1) # skip-connection 2
        conv3 = self.dconv_down3(conv2) # skip-connection 3
        conv4 = self.dconv_down4(conv3) # skip-connection 4

        x = self.bottleneck(conv4)
        x = torch.cat([x, conv4], dim=1)
        x = self.dconv_up4(x)
        x = torch.cat([x, conv3], dim=1)
        x = self.dconv_up3(x)
        x = torch.cat([x, conv2], dim=1)
        x = self.dconv_up2(x)
        x = torch.cat([x, conv1], dim=1)
        x = self.dconv_up1(x)
        x = torch.cat([x, start], dim=1)
        return self.conv_last(x)
```

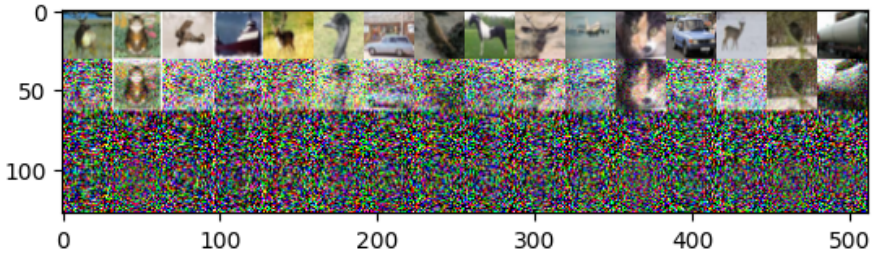
Evaluate your architecture for 10,000 steps on MNIST. You should see much better convergence; the loss on MNIST should be less than 0.1 after 2500 steps and less than 0.05 after 10,000 steps. This means it is now much better at recovering ϵ, with the forward process outputs:



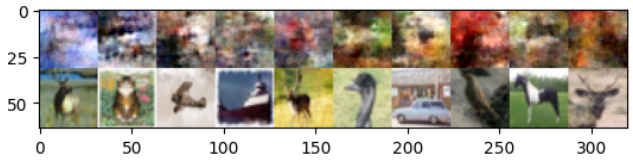
And similarly this produces better (much room for improvement) samples:



For CIFAR10, the loss should be less than 0.12 after 2500 steps and around 0.08 after 10,000 steps. Clearly, the distribution of the CIFAR10 data is more difficult to model than MNIST:



And similarly this also produces better (much room for improvement) samples:



6 Conditional diffusion

In Algorithm 1 (forward) and Algorithm 2 (sample) of the original paper, the authors also inject the schedule times t into the network for some improvement. This extra 'conditional' information helps the network generate a better estimate. In other words, it becomes a conditional generative model.

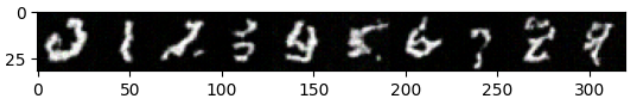
The general wisdom is that conditional (prior) information nearly always helps the generated outputs. With MNIST and CIFAR10 we have access to some rather excellent prior knowledge about the images; the class label $\mathbf{c}$ that they belong to. This could be expressed as a one-hot encoding (a vector of zeros with a single 1 at the index of the class):

```
ch = F.one_hot(c, num_classes=10).float().to(device)
```

We need to find a way to inject this prior class information into our U-Net architecture during training. Then, during sampling, we could inject `torch.arange(n_sample)` to generate class conditional images. Try to implement this yourself, otherwise if you are short on time here is a very simple way example of this:

```python
class UNet(nn.Module):

    ...

    def forward(self, x, c):
        start = x
        conv1 = self.dconv_down1(x)     # skip-connection 1
        conv2 = self.dconv_down2(conv1) # skip-connection 2
        conv3 = self.dconv_down3(conv2) # skip-connection 3
        conv4 = self.dconv_down4(conv3) # skip-connection 4

        # simple inject of the conditionals into the network
        ch = F.one_hot(c, num_classes=10).float().to(device).view(-1,10,1,1)

        x = self.bottleneck(conv4)
        x = torch.cat([x, conv4, ch.repeat(1,1,x.size(2),x.size(3))], dim=1) # injection 1
        x = self.dconv_up4(x)
        x = torch.cat([x, conv3, ch.repeat(1,1,x.size(2),x.size(3))], dim=1) # injection 2
        x = self.dconv_up3(x)
        x = torch.cat([x, conv2], dim=1)
        x = self.dconv_up2(x)
        x = torch.cat([x, conv1], dim=1)
        x = self.dconv_up1(x)
        x = torch.cat([x, start], dim=1)

        return self.conv_last(x)

class DDPM(nn.Module):
    ...

    # algorithm 1 in DDPM paper (with conditionals)
    def forward(self, x, c):
        ...
        return F.mse_loss(eps, self.net(x_t, c))

    # algorithm 2 in DDPM paper with torch.arange(n_sample) conditionals
    def sample(self, n_sample, size):
        ...
            eps = self.net(x_i, torch.arange(n_sample))
        ...

while(step < 10000):

    x,c = next(train_iterator)
    x,c = x.to(device), c.to(device)

    loss = ddpm(x, c) # modified with conditionals

    optim.zero_grad()
    loss.backward()
    optim.step()

    ...
```

After 25,000 steps this gives:



As you can see, it is far from perfect—but you can certainly make out the conditional information. There are much better ways to inject conditionals and better UNet architectures than the minimal implementation we have explored today. State-of-the-art approaches use deep residual architectures, better sampling strategies, attention layers, SIREN time embeddings, *much longer* training times, and other more expressive functions.
