AIL 862

Lecture 25

Figure from "Probabilistic Machine Learning: Advanced Topics". Online version

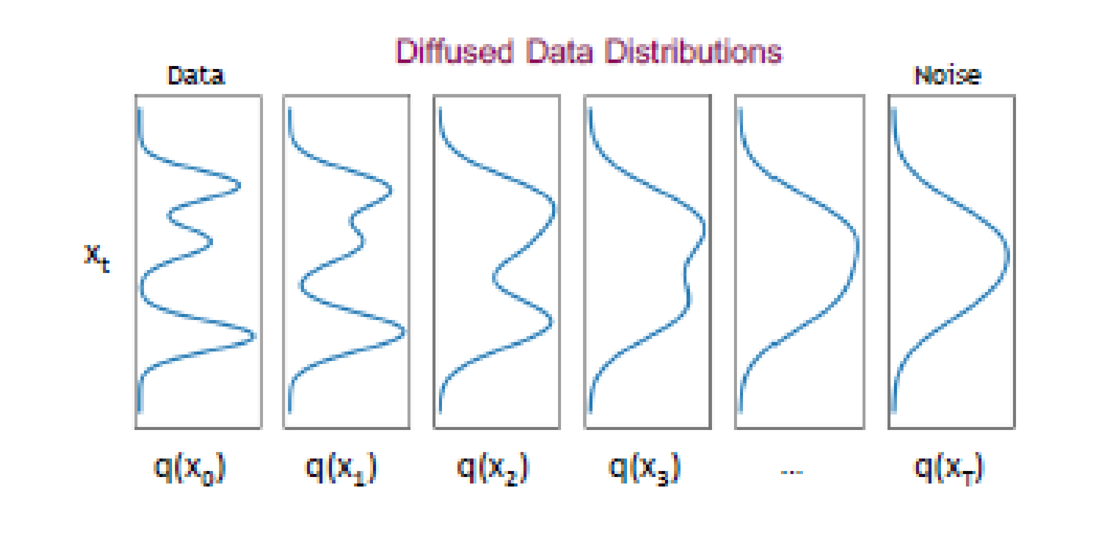

Figure from “Probabilistic Machine Learning: Advanced Topics”. Online version

Forward process

The forwards encoder process is defined to be a simple linear Gaussian model:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t|\sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t\mathbf{I})$$

From “Probabilistic Machine Learning: Advanced Topics”. Online version

Forward process

Since this defines a linear Gaussian Markov chain, we can compute marginals of it in closed form. In particular, we have

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t|\sqrt{\overline{\alpha}_t}\boldsymbol{x}_0, (1-\overline{\alpha}_t)\mathbf{I}) \tag{25.3}$$

where we define

$$\alpha_t \triangleq 1-\beta_t, \overline{\alpha}_t = \prod_{s=1}^{t} \alpha_s \tag{25.4}$$

From “Probabilistic Machine Learning: Advanced Topics”. Online version

```
# 2. Pick up x_0 (shape: [batch_size, 3, 32, 32])
x_0 = data.to(device)

# 3. Pick up random timestep, t .
#    Instead of picking up t=1,2, ... ,T ,
#    here we pick up t=0,1, ... ,T-1 .
#   (i.e, t == 0 means diffused for 1 step)
b = x_0.size(dim=0)
t = torch.randint(T, (b,)).to(device)
```

https://github.com/tsmatz/diffusion-tutorials/blob/master/02-ddpm.ipynb

```
# 2. Pick up x_0 (shape: [batch_size, 3, 32, 32])
x_0 = data.to(device)

# 3. Pick up random timestep, t .
#    Instead of picking up t=1,2, ... ,T ,
#    here we pick up t=0,1, ... ,T-1 .
#   (i.e, t == 0 means diffused for 1 step)
b = x_0.size(dim=0)
t = torch.randint(T, (b,)).to(device)
```

```
eps = torch.randn_like(x_0).to(device)

# 5. Compute x_t = sqrt(alpha_bar_t) x_0 + sqrt(1-alpha_bar_t) epsilon
#    (t == 0 means diffused for 1 step)
x_t = sqrt_alpha_bars_t[t][:,None,None,None].float() * x_0 + sqrt_one_minus_alpha_bars_t[t][:,None,None,None].float() * eps
```

```python
# 2. Pick up x_0 (shape: [batch_size, 3, 32, 32])
x_0 = data.to(device)

# 3. Pick up random timestep, t .
#    Instead of picking up t=1,2, ... ,T ,
#    here we pick up t=0,1, ... ,T-1 .
#   (i.e, t == 0 means diffused for 1 step)
b = x_0.size(dim=0)
t = torch.randint(T, (b,)).to(device)
```

```python
eps = torch.randn_like(x_0).to(device)

# 5. Compute x_t = sqrt(alpha_bar_t) x_0 + sqrt(1-alpha_bar_t) epsilon
#    (t == 0 means diffused for 1 step)
x_t = sqrt_alpha_bars_t[t][:,None,None,None].float() * x_0 + sqrt_one_minus_alpha_bars_t[t][:,None,None,None].float() * eps
```

```python
T = 1000
alphas = torch.linspace(start=0.9999, end=0.98, steps=T, dtype=torch.float64).to(device)
alpha_bars = torch.cumprod(alphas, dim=0)
sqrt_alpha_bars_t = torch.sqrt(alpha_bars)
sqrt_one_minus_alpha_bars_t = torch.sqrt(1.0 - alpha_bars)
```

Reverse diffusion (decoder)

- Implemented with deep networks

```
# 6. Get loss and apply gradient (update)
model_out = unet(x_t, t)
loss = F.mse_loss(model_out, eps, reduction="mean")
loss.backward()
opt.step()
scheduler.step()
```

```python
def forward(self, x, t_emb):
    """
    Parameters
    ----------
    x : torch.tensor((batch_size, in_channel, width, height), dtype=float)
        input x
    t_emb : torch.tensor((batch_size, base_channel_dim * 4), dtype=float)
        timestep embeddings
    """

    # Apply conv
    out = self.norm1(x)
    out = F.silu(out)
    out = self.conv1(out)

    # Add timestep encoding
    pos = F.silu(t_emb)
    pos = self.linear_pos(pos)
    pos = pos[:, :, None, None]
    out = out + pos

    # apply dropout + conv
    out = self.norm2(out)
    out = F.silu(out)
    out = F.dropout(out, p=0.1, training=self.training)
    out = self.conv2(out)

    # apply residual
    if self.linear_src is not None:
        x_trans = x.permute(0, 2, 3, 1)       # (N,C,H,W) --> (N,H,W,C)
        x_trans = self.linear_src(x_trans)
        x_trans = x_trans.permute(0, 3, 1, 2) # (N,H,W,C) --> (N,C,H,W)
        out = out + x_trans
    else:
        out = out + x

    return out
```

Training algorithm

Algorithm 25.1: Training a DDPM model with L_{simple}.

while *not converged* **do**
- $\boldsymbol{x}_0 \sim q_0(\boldsymbol{x}_0)$
- $t \sim \text{Unif}(\{1, \ldots, T\})$
- $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
- Take gradient descent step on $\nabla_{\boldsymbol{\theta}} ||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}} \left(\sqrt{\overline{\alpha}_t} \boldsymbol{x}_0 + \sqrt{1 - \overline{\alpha}_t} \boldsymbol{\epsilon}, t \right) ||^2$

Sampling

Sampling

Algorithm 25.2: Sampling from a DDPM model.

$\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

foreach $t = T, \ldots, 1$ **do**

$\quad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

$\quad \boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}} \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \right) + \sigma_t \boldsymbol{\epsilon}_t$

Return $\boldsymbol{x}_0$

```python
import tqdm

def run_inference(unet, num_images):
    unet.eval()
    # 0. generate sigma_t
    alpha_bars_prev = torch.cat((torch.ones(1).to(device), alpha_bars[:-1]))
    sigma_t_squared = (1.0 - alphas) * (1.0 - alpha_bars_prev) / (1.0 - alpha_bars)
    sigma_t = torch.sqrt(sigma_t_squared)
    # 1. make white noise
    x = torch.randn(num_images, 3, 32, 32).to(device)
    # 2. loop
    #   (t == 0 means diffused for 1 step)
    with torch.no_grad():
        for t in tqdm.tqdm(reversed(range(T)), total=T):
            if t > 0:
                z = torch.randn_like(x).to(device)
            else:
                z = torch.zeros_like(x).to(device)
            t_batch = (torch.tensor(t).to(device)).repeat(num_images)
            epsilon = unet(x, t_batch)
            x = (1.0 / torch.sqrt(alphas[t])).float() * (x - ((1.0 - alphas[t]) / torch.sqrt(1.0 - alpha_bars[t])).float() * epsilon) + \
                sigma_t[t].float() * z

    # reshape to channels-last : (N,C,H,W) --> (N,H,W,C)
    x = x.permute(0, 2, 3, 1)

    # clip
    x = torch.clamp(x, min=0.0, max=1.0)

    return x

# initialize
num_row = 10
num_col = 10
# generate images
x = run_inference(unet, num_row*num_col)
# draw
fig, axes = plt.subplots(num_row, num_col, figsize=(5,5))
for i in range(num_row*num_col):
    image = x[i].cpu().numpy()
    row = i//num_col
    col = i%num_col
    ax = axes[row, col]
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(image)
```