Enhancing Denoising Diffusion Probabilistic Models with Performers

Presentation by Ninniri Matteo (ID: 543873)

October 23, 2022

In this presentation:

We will:

- Briefly discuss the theory behind:
    - DDPMs
    - Performers
- Discuss the implementation alongside the theory
- Compare our implementation against the original one

Table of Contents

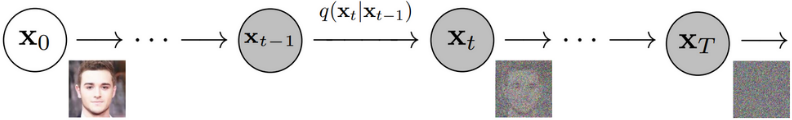
Diffusion models:

- Latest trend in AI research
- Defined in Diffusion.py
- 2 methods:
  - fit() → diffusion process
  - sample() → reverse process

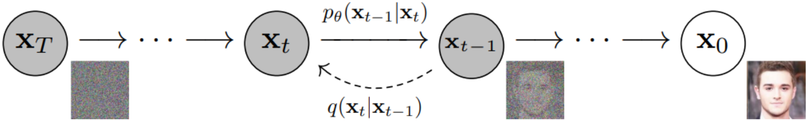
```
class Diffusion(torch.nn.Module):
  def __init__(self, <many parameters>):
    (...)

  def fit(self, x x_vl = None):
    (uses the diffusion process to train a neural network)

  def sample(self, verbose_steps : int = 0):
    (applies the reverse process starting from gaussian noise)
```

The diffusion and the reverse procedures

Diffusion: gradually add noise to $x_0 \sim q(x_0)$ for T steps:



Reverse process: reconstruct x_0 from $x_T \sim \mathcal{N}(0, I)$



We then use p_θ to sample new images!

Training process

x_t can be obtained from x_0 directly:

$$x_t = \sqrt{\overline{\alpha}_t} x_0 + \sqrt{1 - \overline{\alpha}_t} \epsilon \tag{1}$$

- $\epsilon \sim \mathcal{N}(0, I)$
- $\overline{\alpha}_t = \prod_{s=1}^{t} \alpha_s$
- $\alpha_t = 1 - \beta_t$
- β_i is a *variance scheduler* $(\beta_1 \approx 0, \beta_T \approx 1)$

Training:

Idea: we train a neural network to predict ϵ

```
def fit(self, x, x_v = None):
  (dataloader setup)

  for epoch in range(self.n_iters): #for each epoch
    tr_loss = torch.tensor([0])

    enumerator = tqdm(data_loader)
    for i, data in enumerate(enumerator): #for each minibatch
      images, _ = data
      images = images.to(self.device)

      b = images.size(0)

      (see next slide)
```

Training:

```
#sample a timestep and the noise
t = torch.randint(low = 0, high = self.sample_iters,
                  size = (b,))
e = torch.randn_like(images)

#extract the various alphas etc
sqrt_a          = torch.zeros((b, 1, 1, 1))
sqrt_1_minus_a  = torch.zeros((b, 1, 1, 1))

for j in range(b):
  sqrt_a[j, 0, 0, 0]          = self.a_sgn_sqrt[t[j]]
  sqrt_1_minus_a[j, 0, 0, 0]  = self.one_m_a_sgn_sqrt[t[j]]

#samples x_t from x_0, calculates the noise
net_in = sqrt_a*images + sqrt_1_minus_a*e
net_out = self.net(net_in, t)

loss = self.loss(net_out, e)

(the usual Pytorch optimization procedure follows)
```

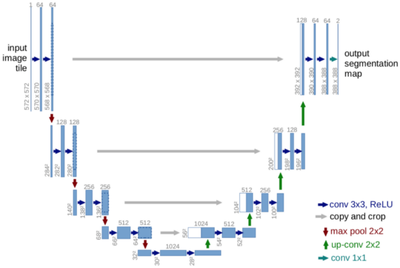
Sampling

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sigma_t z$$

```
def sample(self, verbose_steps : int = 0):
    self.eval()
    transform = transforms.ToPILImage()

    x_T = torch.randn(size = (1, self.n_channels, self.x_sz, self.x_sz))

    for t in reversed(range(self.sample_iters)):
        if(t == 0): z = torch.zeros_like(x_T)
        else:       z = torch.randn_like(x_T)

        model_weight = self.beta[t]/self.one_m_a_sgn_sqrt[t]
        alpha_sqrt = torch.sqrt(self.alphas[t])

        #predicts the noise in x_t
        predicted = self.net(x_T, torch.Tensor([t]).detach()

        #samples x_{t-1} from x_t using the formula shown above
        x_T = (x_T - model_weight*predicted)/alpha_sqrt + z*self.sigma[t]

    self.train()
    return x_T[0, :, :, :]
```

ϵ_θ

ϵ_θ is a Unet featuring ResNet and Attention blocks



The encoder's intermediate blocks are also connected to the decoder's blocks on the same level.

```
class Unet(torch.nn.Module):
    def __init__(
        self,
        x_sz,
        init_dim      : int  = None,
        dim_mults          = (1, 2, 4, 8),
        channels           = 3,
        resnet_block_groups = 8,
        time_dim           = 256,

        h             : int  = 4,
        head_sz       : int  = 32,

        att_type      : str  = 'FAVOR_SDP',
        m             : int  = None,
        redraw_steps  : int  = 1000,
        device             = 'cuda',

        use_original  : bool = False,
    ):
```

It also supports the original implementation's Attention.

Standard Attention

Standard attention (class SDPAttention):

```
def forward(self, Q, K, V, mask = None):
    prod = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.d_k_sqrt

    if mask is not None:
        prod = prod.masked_fill(mask == 0, -1e10)

    att = torch.softmax(prod, dim = -1)
    res = torch.matmul(self.dropout(att), V)

    return res, att
```

Cost: $\mathcal{O}(L^2 d)$ because of the second matmul.
$\Rightarrow$ can we rearrange the multiplications?

Performers ("FAVOR+")

Performers approximates the Attention with:

$$Attention(i,j) = \mathbb{E}[\phi(Q_i)^T \phi(K_j)] \tag{2}$$

ϕ depends on the function we are approximating (softmax):

$$\phi(x) = \frac{h(x)}{\sqrt{m}}(f_1(x\omega_1^T), \ldots, f_1(x\omega_m^T), \ldots, f_l(x\omega_1^T), \ldots, f_l(x\omega_m^T)) \tag{3}$$

- $h : \mathbb{R}^d \to \mathbb{R}$
- $f_1 \ldots f_l$ are l functions of the form $\mathbb{R} \to \mathbb{R}$
- $\omega_1 \ldots \omega_m \sim \mathcal{D} \in \mathcal{P}(\mathbb{R}^d)$ are random orthogonal vectors (FAVORplus.redraw_proj_matrix).

Performers

Softmax's ϕ has the following form:

$$\phi(x) = \frac{1}{\sqrt{m}} exp(-\frac{||x||^2}{2\sqrt{d}})(exp(\frac{x}{\sqrt[4]{d}}\omega_1^T), \ldots, exp(\frac{x}{\sqrt[4]{d}}\omega_m^T)) \quad (4)$$

```
def softmax_kernel(self, x):
    arr = x/self.sq_sq_d

    arr = arr @ self.projection_matrix.permute((1, 0))

    g = x**2
    g = torch.sum(g, dim = -1, keepdim = True)
    g = g/(2*self.sq_d)

    to_return = torch.exp(arr - g + self.num_stabilizer)/self.sq_m

    return to_return
```

Performers

We can also replace softmax completely: here is a ReLU kernel

```
def relu_kernel(self, x):
    arr = x @ self.projection_matrix.permute((1, 0))
    arr = arr/self.sq_m
    arr = torch.nn.functional.relu(arr) + \
        torch.tensor([self.num_stabilizer]).to(self.device)

    return arr
```

Performers

Summarizing:

$$\widehat{Att}_{\leftrightarrow}(Q, K, V) = \widehat{D}^{-1}(Q^{'}((K^{'})^{T}V)) \tag{5}$$

where:

- $Q^{'} = \phi(Q) \in \mathbb{R}^{L \times r}$
- $K^{'} = \phi(K) \in \mathbb{R}^{L \times r}$
- $\widehat{D} = diag(Q^{'}((K^{'T}1_{L}))$

Cost: $\mathcal{O}(Lrd)$

$r = ml$

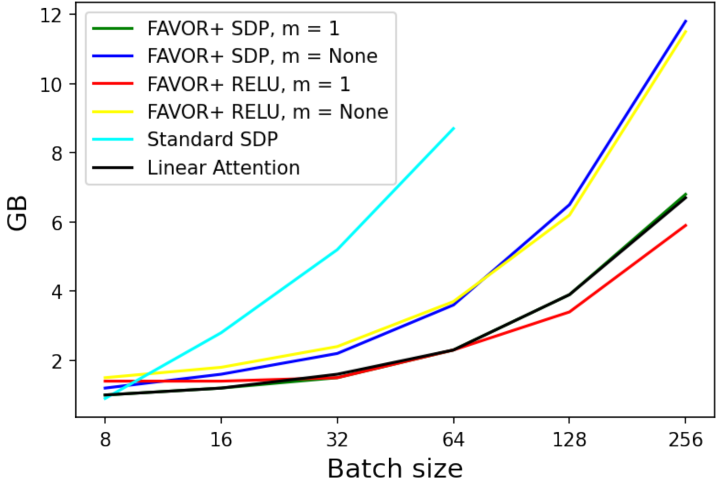
m is optimal when $m = d \log d$

(after calculating phi_q and phi_k):

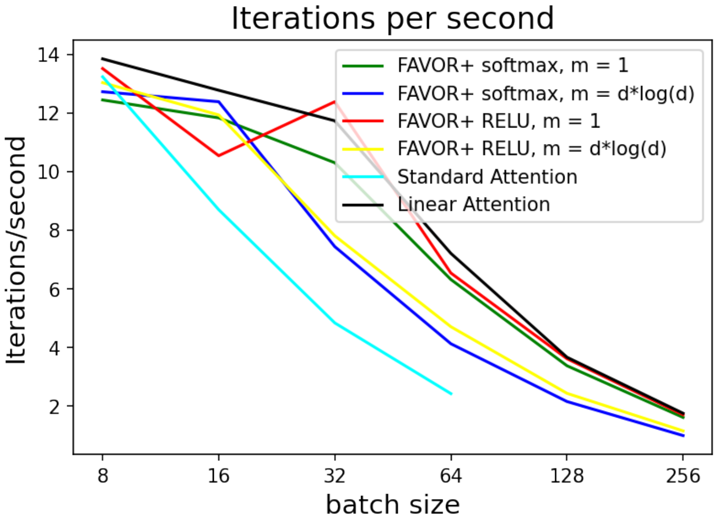
```
phi_k_sum = phi_k.sum(dim = -2).unsqueeze(-1)

#D = Q' @ phi_k_sum
#size: [b, h, L, r]x[b, h, r, 1] = [b, h, L, 1]
D         = phi_q @ phi_k_sum
D_inv     = 1.0/D

#(K'^T) @ V => [b, h, r, L]x[b, h, L, head_sz]
to_return = phi_k.permute((0, 1, 3, 2)) @ v

#phi_q @ to_return => [b, h, L, r]x[b, h, r, he
to_return = phi_q @ to_return

#D^{-1} * to_return => [b, h, L, 1] * [b, h, L,
to_return = D_inv * to_return

return to_return, None
```

Table of Contents

Memory consumption

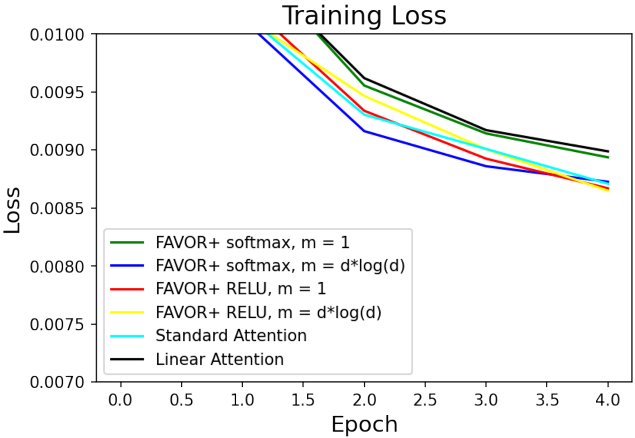


Training times



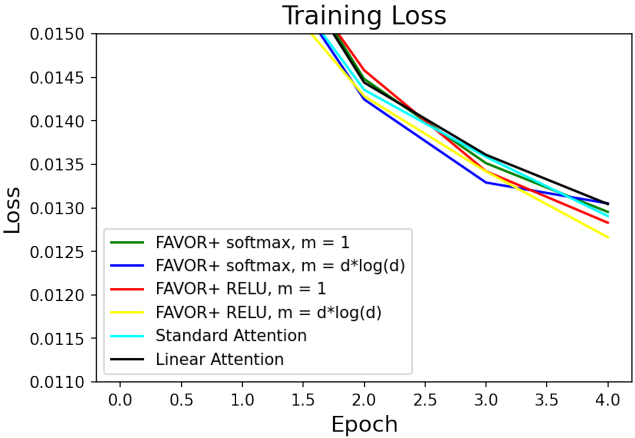
Loss

Zoomed-in loss for MNIST



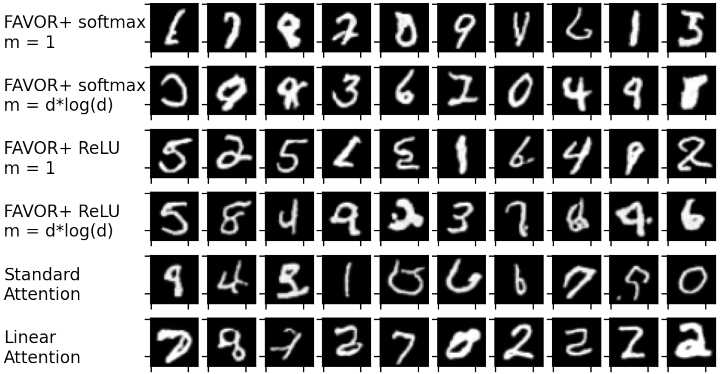
Loss

Zoomed-in loss for FashionMNIST



Generated samples

MNIST



Generated samples

That's all.

Thanks for the attention!