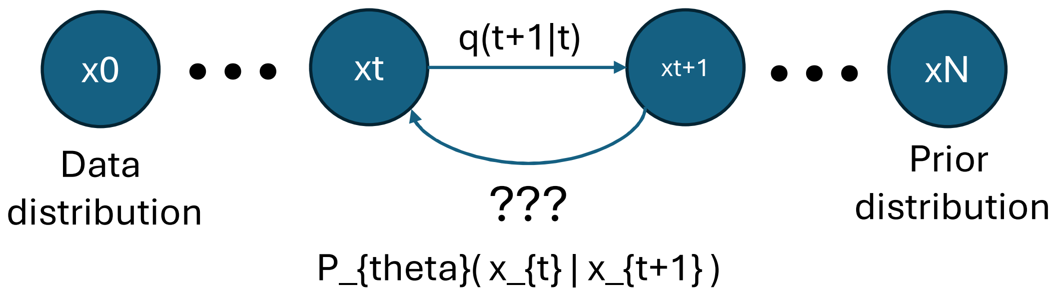
Discretized diffusion process

- DDPM presented a descretized version of the diffusion process which presented in a Markovian framework. Diffusion occurs in steps with the following state transition kernel.

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



- The optimization problem for the reverse problem is given by the maximizing the log-likelihood of the reversed process for all k in 0,…,T. It can be proven, further evaluation gives the "Denoising score matching loss".

$$\nabla \log p_k(x_k) = \mathbb{E}_{p_{0|k}(\cdot|x_k)}[\nabla \log p_{k|0}(x_k|X_0)]$$

$$Y = \mathbb{E}[X\,|U] \text{ if } Y = f(U), \text{ with } f = \arg\min\{\mathbb{E}[\|X - f(U)\|^2] \;:\; f \in \mathrm{L}^2(U)\}.$$

$$\boxed{\nabla \log p_k = \arg\min\{\mathbb{E}[\|f(X_k) - \nabla \log p_{k|0}(X_k|X_0)\|^2] \;:\; f \in \mathrm{L}^2(p_k)\}\;.}$$

Reverse diffusion

Denoising Diffusion Probablistic Model(2019)(DDPM) introduced Langevin Dynamics to solve the reverse diffusion process based on the assumption that it is a markov process. Forward models of diffusion is given as,

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \beta_t\epsilon, \qquad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

Reverse sampling is called ancestral sample and closely resembles Langevin dynamics

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sigma_t\epsilon, \qquad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

Reverse diffusion continued…

Denoising Diffusion Implicit Model(2022)(DDIM) used the same approach for training the model but rejects the Markovian approach to sample and samples only a subsequence $\tau = [\tau_{t_1} \dots \tau_{t_N}]$ with $|\tau|$ selected as a hyper-parameter. Forward models of this diffusion process is same as that of DDPM, reverse sampleing is given by,

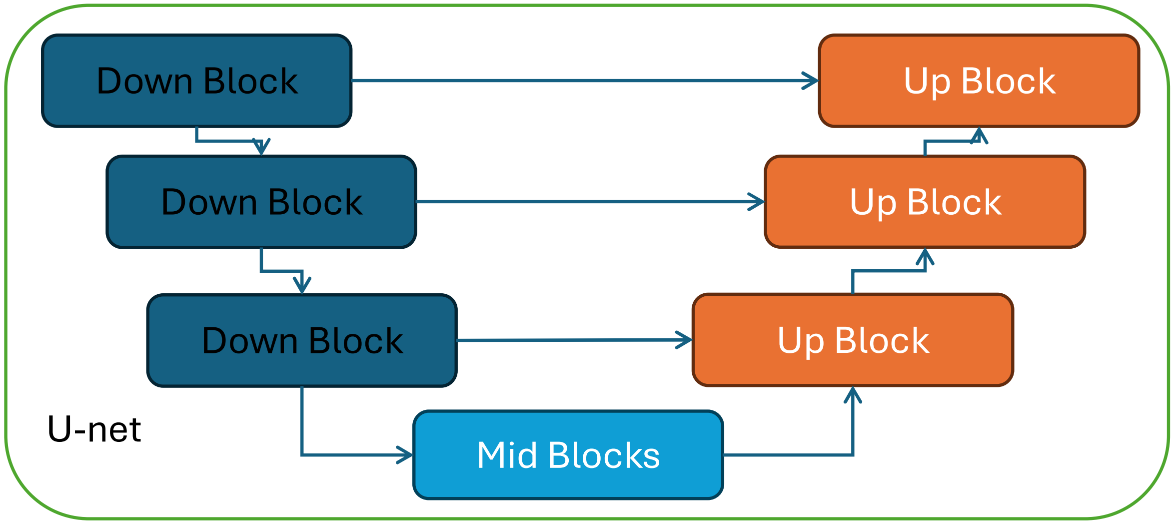
$$x_{\tau_{i-1}} = \frac{\sqrt{\alpha_{\tau_{i-1}}}}{\sqrt{\alpha_{\tau_i}}}(x_{\tau_i} - \sqrt{1-\bar{\alpha}_{\tau_i}}\epsilon_\theta(x_{\tau_i}, \tau_i)) + \sqrt{1-\alpha_{\tau_i} - \sigma_{\tau_i}(\eta)^2}\epsilon_\theta(x_{\tau_i}, \tau_i) + \sigma_{\tau_i}(\eta)\epsilon$$

$\epsilon \sim \mathcal{N}(0, \mathbf{I})$ and,

$$\sigma_{\tau_i}(\eta) = \eta\sqrt{\frac{1-\alpha_{\tau_{i-1}}}{1-\alpha_{\tau_i}}}\left(\sqrt{1-\frac{\alpha_{\tau_i}}{\alpha_{\tau_{i-1}}}}\right)$$

Deep diffusion model using pytorch

Neural network used:



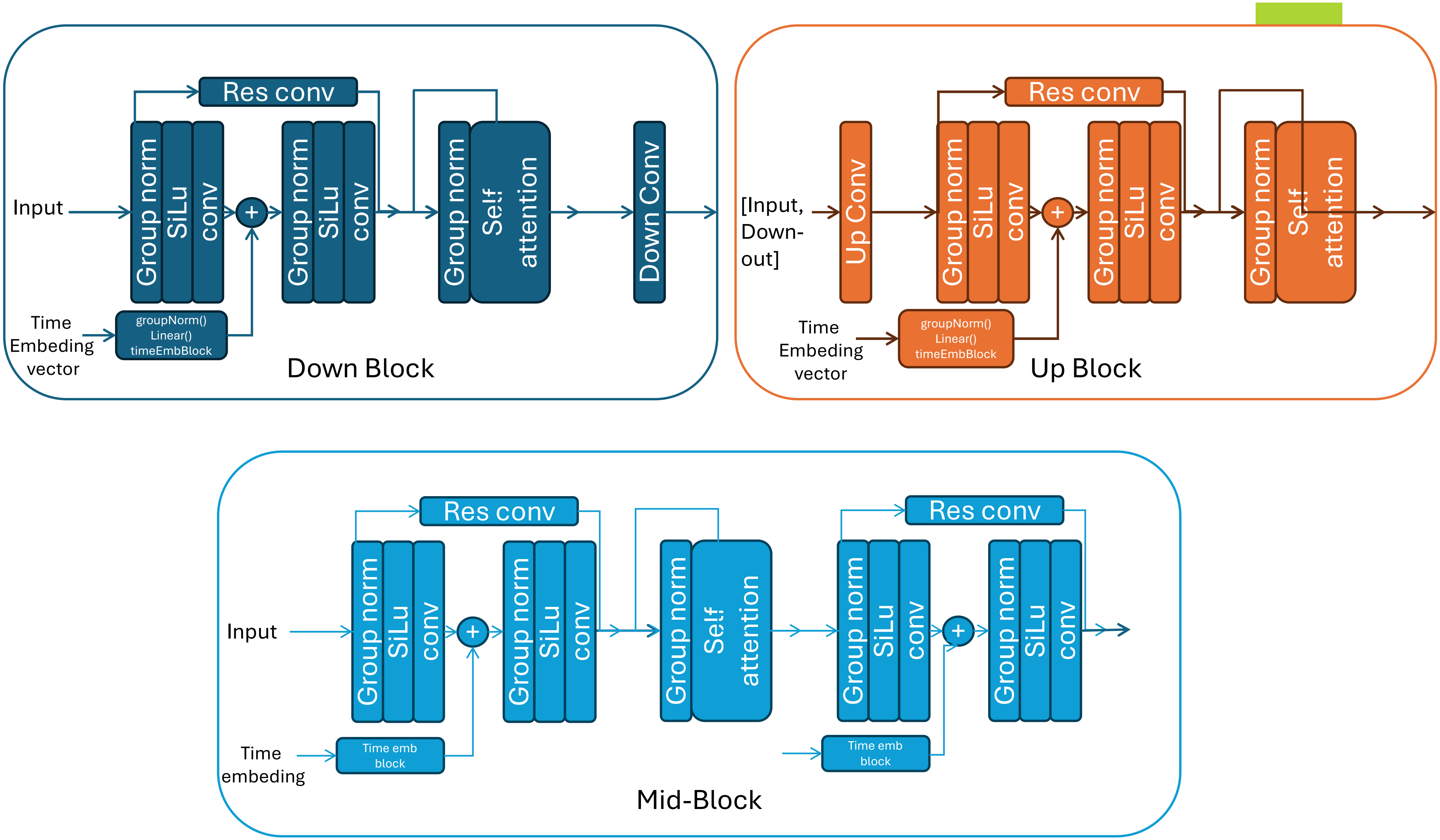
Train step:

```
optimizer.zero_grad()
im = im.float().to(device)

# Sample random noise
noise = torch.randn_like(im).to(device)

# Sample timestep
t = torch.randint(0, diffusion_config['num_timesteps'], (im.shape[0],)).to(device)
noisy_im = scheduler.add_noise(im, noise, t)
noise_pred = model(noisy_im, t)

loss = criteria(noise_pred, noise)
losses.append(loss.item())
loss.backward()
optimizer.step()
```


Res conv
Input
Group norm
SiLu
conv
+
Group norm
SiLu
conv
Group norm
Self attention
Down Conv
Time Embeding vector
groupNorm()
Linear()
timeEmbBlock
Down Block
Res conv
[Input, Down-out]
Up Conv
Group norm
SiLu
conv
+
Group norm
SiLu
conv
Group norm
Self attention
Time Embeding vector
groupNorm()
Linear()
timeEmbBlock
Up Block
Res conv
Res conv
Input
Group norm
SiLu
conv
+
Group norm
SiLu
conv
Group norm
Self attention
Group norm
SiLu
conv
+
Group norm
SiLu
conv
Time embeding
Time emb block
Time emb block
Mid-Block

Deep diffusion model using pytorch

Noise Scheduler: Add Noise(DDPM)

```python
def add_noise(self, orignal, noise, t):
    orignal_shape = orignal.shape
    batch_size = orignal_shape[0]

    sqrt_alpha_cum_prod = self.sqrt_alpha_cum_prod[t]
    sqrt_one_minus_alpha_cum_prod = self.sqrt_one_minus_alpha_cum_prod[t]

    for _ in range(len(orignal_shape)-1):
        sqrt_alpha_cum_prod = sqrt_alpha_cum_prod.unsqueeze(-1)
        sqrt_one_minus_alpha_cum_prod = sqrt_one_minus_alpha_cum_prod.unsqueeze(-1)

    return sqrt_alpha_cum_prod*orignal + sqrt_one_minus_alpha_cum_prod*noise
```

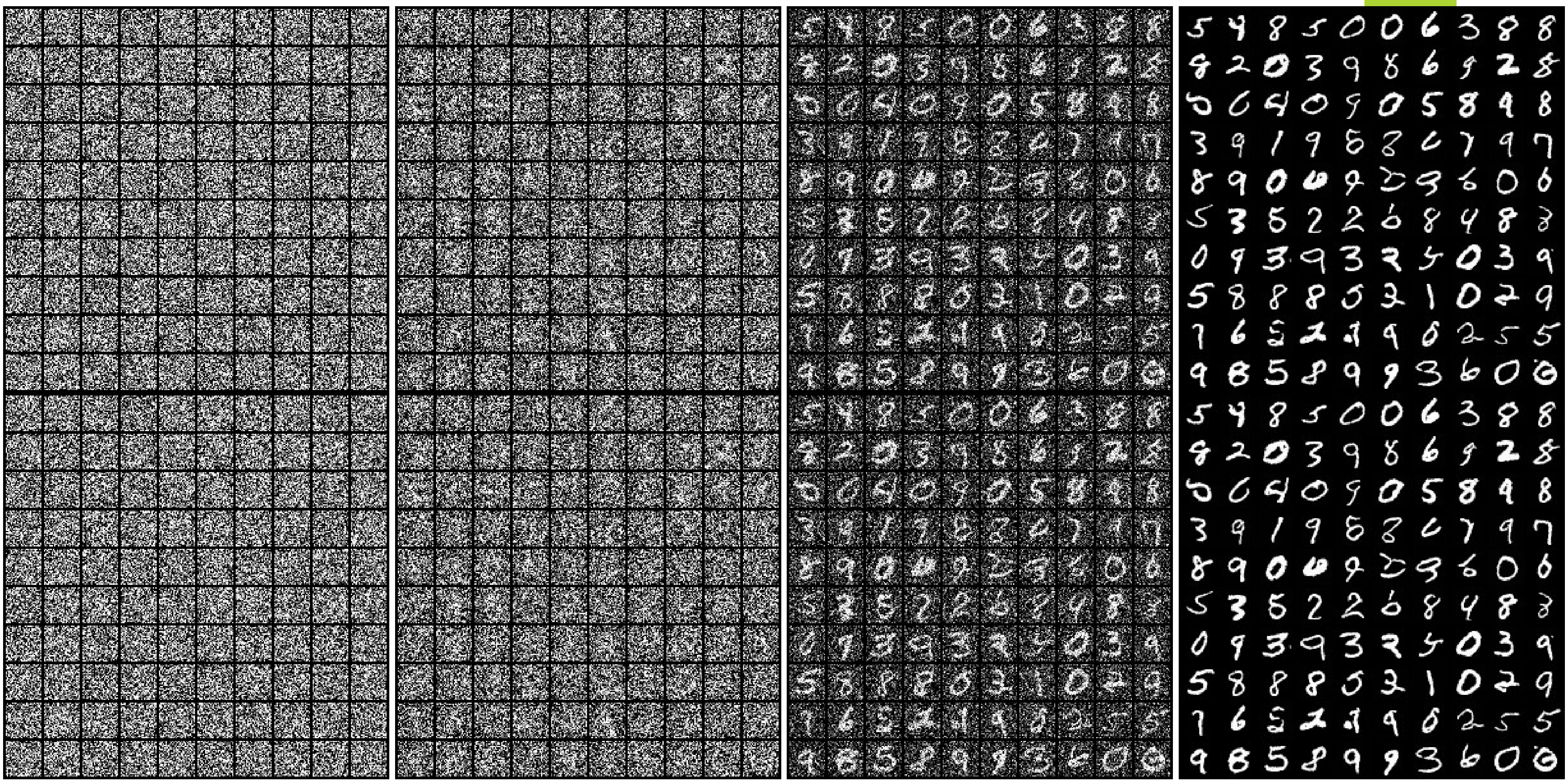
Noise Scheduler: Sample previous step(DDPM)

```python
def sample_prev_timestep(self, xt, noise_pred, t):
    x0 = (xt - (self.sqrt_one_minus_alpha_cum_prod[t] * noise_pred)) / self.sqrt_alpha_cum_prod[t]
    x0 = torch.clamp(x0, -1, 1)

    mean = xt - ((self.betas[t] * noise_pred) / (self.sqrt_one_minus_alpha_cum_prod[t]))
    mean = mean / self.sqrt_alpha_cum_prod[t]

    if t==0:
        return mean, x0
    else:
        variance = ((self.betas[t])*(1 - self.alpha_cum_prod[t-1]))/ (1- self.alpha_cum_prod[t])
        sigma = variance ** 0.5
        z = torch.randn_like(xt).to(xt.device)
        return mean + sigma * z, x0
```

Results:



Step 750 | Step 500 | Step 250 | Step 0