1 Basic Part

Implementation details:

- **create_mnist_dataloaders()**: Resize images to image size, convert images to tensor and then use transforms.Normalize() to range [-1, 1]
- **ResidualConvBlock.forward()**: Implement a residual connection, if input/output channels are the same, direct residual by out = x + x2, otherwise residual from first conv by out = x1 + x2. Use residual normalization (/ 1.414) to avoid exploding activations in deep networks and ensure numerical stability.
- **UnetDown()**: Uses ResidualConvBlock() to extract features. Use MaxPool2d(2) to downsample, reducing by half.
- **UnetUp()**: Use transpose convolution nn.ConvTranspose2d() to upsample feature maps. Use torch.cat((x, skip), 1) to concatenate with skip connection, which combines downsample features from the encoder to keep details.
- **SimpleDiffusion.forward()**: Randomly selects a timestamp t for each input sample from 1 to n_T. Compute the noisy image x_t by this equation, which describes the forward process gradually converts a clean image into Gaussian noise, x_0 will multiply by the cumulative product of noise schedule. Then it will dropout conditioning labels by drop_prob to encourage the model to learn both conditional and unconditional sampling. Finally train the model to predict the noise by computing MSE loss:

$$x\_t = \sqrt{\bar{\alpha}_t} x\_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon. \tag{1}$$

- **SimpleDiffusion.sample()**: This function generates new images by reversing the diffusion process. Firstly initialize from pure Gaussian noise. Secondly generates digits 0-9 as class labels and creates 2 versions of the batch (With conditioning: context_mask = 0; Without conditioning: context_mask = 1). Then Reverse the noise process from n_T to 1, using classifier-free guidance to mix conditional and unconditional predictions and using guide_w to increase guidance.This equation describes the forward process gradually converts a clean image into Gaussian noise. Finally return the final generated images and intermediate steps for visualization

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - (1 - \alpha_t)\epsilon_\theta(x_t, t)\right) + \sqrt{\beta_t} z \tag{2}$$

- **generate_samples()**: Train the model and save it as model_19.pth. Load the pre-trained diffusion model and define my student_id = "00297375". For each digit, creates context tensor from digit, initializes context mask and generates random noise as starting point. Then iteratively denoises the image by computing noise prediction at each timestep, applying classifier-free guidance and finally updating image using diffusion equations.

Experiment Result:

guide_w: change the parameter guide_w from [0, 0.5, 2]. Lower guide_w produces more natural but potentially blurry outputs. Higher guide_w creates sharper but potentially more rigid results. This shows how Classifier-Free Guidance effectively controls both generation quality and feature prominence.

$$\hat{\epsilon}_t = (1 + w)\psi(z_t, c) - w\psi(z_t) \tag{3}$$

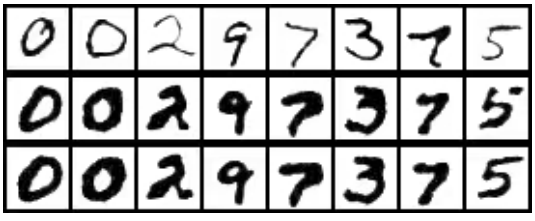

Figure 1: The generated images and guide_w ranges from [0, 0.5, 2] from up to down

2 Puzzle 1: How to generate a two-digit image using an end-to-end diffusion model

Use the 2-digit label space (00-99) instead of single digits (0-9):

- **n_classes=100 in ContextUnet._init_()**: Modify the number of classes from 10 to 100 in ContextUnet().
  def _init_(self, in$_channels, n_feat = 256, n_classes = 100$)
- **n_classes=100 in ContextUnet.forward()**: Convert context to one hot embedding for 100 classes. c = nn.functional.one_hot(c, num_classes=self.n_classes).type(torch.float) context_mask = context_mask.repeat(1, self.n_classes)
- **Result**: It remaps the labels to 0-99 range instead of 0-9. Keep the same architecture but expand the embedding dimension. When generate the images, it can generate a two-digit image once and then concat all the output. It would be an end-to-end diffusion model and get potentially better digit relationships by single forward pass.

3 Puzzle 2: Besides simply sampling the noise from a Gaussian distribution, what are some alternative sampling strategies that could improve the generation process?

Noise selection:

- **Algorithm strategy**: Select the most stable one from K random noises, sample the noise, computes stability score using cosine similarity, and retain the most stable noise
- **quality assurance**: Only selects the most stable noise, avoiding the use of noise that may lead to low-quality generation
- **Automatic screening**: Find the most suitable initial conditions through multiple sampling and evaluation
- **No need to modify the model:**: Can be directly applied to existing pre-trained models with better quality through noise selection

```python
class NoiseStrategyDiffusion(SimpleDiffusion):
    def __init__(self, nn_model, betas, n_T, device, drop_prob=0.1):
        super().__init__(nn_model, betas, n_T, device, drop_prob)

    def compute_inversion_stability(self, noise, c, size):
        x_0 = self.sample_from_noise(noise, c, size)
        inverse_noise = self.inverse_noise(x_0, c)
        return F.cosine_similarity(noise.view(noise.shape[0], -1),
                                   inverse_noise.view(inverse_noise.shape[0], -1),
                                   dim=1)

    def noise_selection(self, K, c, size, device):
        best_noise, best_stability = None, -1
        for _ in range(K):
            noise = torch.randn(1, *size).to(device)
            stability = self.compute_inversion_stability(noise, c, size)
            if stability > best_stability:
                best_stability, best_noise = stability, noise
        return best_noise

    def sample_from_noise(self, noise, c, size):
        x_i = noise
        for i in range(self.n_T, 0, -1):
            t_is = torch.tensor([i / self.n_T]).to(self.device).repeat(noise.shape[0], 1, 1, 1)
            eps = self.nn_model(x_i, c, t_is, torch.zeros_like(c).to(self.device))
            z = torch.randn_like(x_i) if i > 1 else 0
            x_i = (self.oneover_sqrta[i] * (x_i - eps * self.mab_over_sqrtmab[i]) +
                   self.sqrt_beta_t[i] * z)
        return x_i

    def inverse_noise(self, x_0, c):
        x_t = x_0
        for i in range(1, self.n_T + 1):
            t_is = torch.tensor([i / self.n_T]).to(self.device).repeat(x_0.shape[0], 1, 1, 1)
            x_t = (self.sqrtab[i] * x_t + self.sqrtmab[i] * torch.randn_like(x_t))
        return x_t
```