1) Context U-Net

Problem: Implement forward pass of the context U-Net

Algorithm: The input x is an image tensor of shape (batch, n_feat, h, w) and we pass it through the initial convolutional layer to extract its features. The resulting tensor is passed to two downsampling layers to reduce the spacial dimensions of the image but increase the number of channels. This is then passed to the average pooling layer to further reduce the spacial resolution of the image. We start upscaling that result by passing it through the first upsampling block which increases the spacial dimensions of the image. If the input context tensor is empty, we replace it with a tensor of all-zero elements. We then pass the time step tensor into the time embeddings block. For each time embedding, modify the tensor to have the same elements but with a new shape. The first time and context embedding blocks are reshaped to (B, self.n_feat*2, 1, 1). The second time and context embedding blocks are reshaped to (B, self.n_feat, 1, 1). This is so that the shapes align when multiplying the context embeddings with the upsampling block and adding the time embedding. We continue to pass the result to the upsampling blocks to increase resolution size and include the timestep and context embeddings according to the provided image of the context U-Net. The output is a sequential container consisting of a covnet, group norm, relu, and another covnet. This container takes in the concatenation of the previous upsampling block and the initial convolution computed at the beginning of the forward pass. This is also called a skip connection.

2) DDPM noise schedule

Problem: Implement Denoising Diffusion Probabilistic Models noise scheduler

Algorithm: Create a beta tensor to hold beta values that quantify in magnitude how much noise to add to image at each time step where $\beta \in [\beta_1, \beta_2] \subset (0, 1)$. Create an alpha tensor of alpha values each of which signifying the remaining fraction of original image data as a result of noising. $\alpha_t = 1 - \beta_t$ Compute the cumulative product of the alpha values. $\overline{\alpha}_t = \prod_{s=1}^{t} \alpha_s$

3) Training without context

Problem: Corrupt the input image with noise.

Algorithm: We noise the input image x using equation 4 from paper https://arxiv.org/pdf/2006.11239

$q(x_t|x_0) = N(x_t; \sqrt{\overline{\alpha}_t} x_0, (1 - \overline{\alpha}_t) I)$

we know that $N(\mu, \sigma I) \sim x_t = \mu + \sigma N(0, 1)$

hence, our corrupted image is $x_t = \sqrt{\overline{\alpha}_t} x_0 + \sqrt{(1 - \overline{\alpha}_t)}\, \varepsilon$, where $\varepsilon \sim N(0, 1)$

4) Loss no context function

Problem: Implement the loss function without context

Algorithm: Sample random noise from N(0, 1). Let this noise be a tensor epsilon of the same size as image x. Let t be a tensor of random sampled timesteps. Normalize t. Corrupt the input image with noise. Predict that noise from the model and call it epsilon hat. Compute the mean squared loss between that noise and the sampled noise epsilon.

5) Denoise add noise function

Problem: Remove noise from image at a given timestep. This is called the diffusion reverse process.
Algorithm: We follow algorithm 2 from https://arxiv.org/pdf/2006.11239. We want to return $N(\mu, \sigma I) \sim x_t = \mu + \sigma z, \; where \; z \sim N(0,1)$. From the paper, we know that

$\mu = \frac{1}{\alpha_t}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}}}\varepsilon)$ and $\sigma_t^2 = \beta_t$ and $\beta_t = 1 - \alpha_t$

hence, $x_t = \frac{1}{\alpha_t}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}}}\varepsilon) + \sqrt{\beta_t}z$, where $z \sim N(0,1)$

6) **Sample DDPM function**

Problem: Implement the DDPM sampling process. Iteratively apply denoising to generate samples.
Algorithm: Iterating through time steps, get time tensor at time at the current timestep iteration. Sample random noise. Predict noise from model. Denoise and add noise samples using previous denoise add noise function.

7) **Loss with context function**

Problem: Implement the loss function with context
Algorithm: Code is the same as previous loss function except add context to model.

8) **Sample DDPM function with context**

Problem: Implement the DDPM sampling process. Iteratively apply denoising to generate samples.
Algorithm: Same code as previous DDPM function except context is added to the model.

Pytorch Citations:

torch.reshape(input, shape) → Tensor
Parameters:
input (tensor) - the tensor to be reshaped
shape (tuple of integers) - the new shape
Usage: Returns a tensor with the same data and number of elements as input, but with the specified new shape.
Ex:

```
>>> a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> print(a)
tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> print(a.reshape([4, 2]))
tensor([[1, 2],
        [3, 4],
```

[5, 6],
[7, 8]])

Citation: https://pytorch.org/docs/stable/generated/torch.reshape.html

torch.linspace(start, end, steps, device=None) → Tensor
Parameters:
start (float or Tensor) – the starting value for the set of points.
end (float or Tensor) – the ending value for the set of points.
steps (int) – size of the constructed tensor
device (torch.device, optional) – the desired device of returned tensor.
Usage: Creates a one-dimensional tensor of size steps whose values are evenly spaced from start to end, inclusive.
(start, start + (end−start)/(steps−1),,…,start+(steps−2)*(end−start)/(steps−1),end)
Citations: https://pytorch.org/docs/stable/generated/torch.linspace.html

torch.randn_like(input) → Tensor
Parameters:
input (Tensor) – the size of input will determine size of the output tensor.
Usage: Returns a tensor with the same size as input that is filled with random numbers from a normal distribution with mean 0 and variance 1.
Citation: https://pytorch.org/docs/stable/generated/torch.randn_like.html

torch.randint(low=0, high, size) → Tensor
Parameters:
low (int, optional) – Lowest integer to be drawn from the distribution. Default: 0.
high (int) – One above the highest integer to be drawn from the distribution.
size (tuple) – a tuple defining the shape of the output tensor.
Usage: Returns a tensor filled with random integers generated uniformly between low (inclusive) and high (exclusive).
Ex:

```
>>> torch.randint(3, 5, (3,))
tensor([4, 3, 4])
>>> torch.randint(10, (2, 2))
tensor([[0, 2],
        [5, 5]])
>>> torch.randint(3, 10, (2, 2))
tensor([[4, 5],
        [6, 7]])
```

Citation: https://pytorch.org/docs/stable/generated/torch.randint.html