Report:

● Introduction:

This lab aims at practicing diffusion model with Denoising Diffusion Probabilistic Model (DDPM). Since there is no formatted source code, all the source codes can be implemented freely, there is a briefly introduction about my source code hierarchy.
-data (the data loader file).
-file (the given dataset json file, evaluator file and given model weight).
-model (the diffusion module, UNet module).
-utils (the utility functions including evaluation module and visualization tool).
-config.py (the configuration of parameters).
-test.py (the inference code).
-train.py (the training code).
The DDPM model is implemented with UNet which has the property of skip connections, it helps preserve spatial information throughout the denoising process, which is crucial for generating high-quality images. In addition, the downsampling/upsampling architecture allows the model to capture features at different scales, helping it understand both overall structure and fine details.
I also compare two noise schedules, that is, linear and cosine, to see what are the impact with them respectively.

● Implementation details:

Architecture:
The model architecture is based on UNet, not only because the empirical success in diffusion model, but the components enhance the performance for image generation. In order to have the model better utilize the high dimensional information, time embedding is also introduced in training. The neural network needs rich representations of timesteps, not just scalar values, and the time embedding allows the model to adapt its behavior based on how noisy the image is. As a result, Sinusoidal embedding is used for time embedding.

As for the conditional embedding, it's directly implemented by fully connected layer with GELU as the activation function. The conditional embedding allows the model to understand what content to generate by transferring sparse one-hot labels into rich,

dense representations the network can use. When combined with time embeddings, they guide each denoising step toward the desired content, without them, the model would generate random images with no control over content.
The GELU activation combines aspects of ReLU and sigmoid activations. It multiplies the input by the probability that the input is positive under a standard Gaussian distribution. It's smoother than ReLU at zero, and retains ReLU's ability to mitigate vanishing gradients. It provides better gradient flow and adds beneficial stochastic properties to the network's behavior.

The bottleneck layer is different from traditional UNet architecture, the self-attention layer is introduced for diffusion model. It captures long-range dependencies between distant parts of the image that convolutional layers struggle to model. Self-attention complements the local processing of convolutions by allowing each position to attend to all other positions, helping ensure generated objects have consistent properties across the image.

To sum up, the UNet architecture includes basic upsampling, downsampling and additional sinusoidal time embedding, conditional embedding and self-attention layer in bottleneck.

Diffusion process:
The diffusion process is defined in the Diffusion class in diffusion.py, which includes the methods of the noise schedule, forward diffusion process, loss calculation for training and the sample method for denoising.

There are two noise schedules implemented in my implementation, linear and cosine, the linear noise schedule is defined by one function.
And the noise steps are set to 1000 in both cases, beta_start is 0.0001 and beta_end is 0.02.

```
def linear_beta_schedule(self):
    """Linear noise schedule"""
    return torch.linspace(self.beta_start, self.beta_end, self.noise_steps, device=self.device)
```

It simply creates the equally spaced values from beta_start to beta_end with noise steps.

The cosine noise schedule is a little complicated, it creates a smooth progression of noise levels, which means adding very small noise initially, preserving image structure longer. It creates a natural progression that aligns better with how image details

degrade at different noise levels.

In the forward process (q_sample()), the cumulative product of alpha and the beta are pre-computed in init function, so it just index the alpha value of timestep t in forward process, then apply the noise to the input image. Like the equation here.

$$
\begin{aligned}
q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) &= \mathcal{N}\left(\sqrt{1-\beta_t}\mathbf{x}_{t-1},\ \beta_t\mathbf{I}\right) \\
\mathbf{x}_t &= \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \\
&= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\epsilon \\
&= \dots \\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon
\end{aligned}
$$

Source: UC Berkeley

In the reverse process (p_sample()), use the input noisy image, condition and timestep t to model to do the prediction of the noise, then use the predicted noise to calculate the mean value according to the equation.

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{a_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right)$$

Source: https://medium.com/image-processing-and-ml-note/diffusion-models-b4609ff05ae6

Finally, use the mean value and pre-calculated deterministic variance and noise to reconstruct the previous image. Like the equation below.

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

Source: https://medium.com/image-processing-and-ml-note/diffusion-models-b4609ff05ae6

The variance σ_t is defined by the equation here.

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

Source: UC Berkeley

And the noise z is random variable.

Finally, there is a p_losses function which purpose is to calculate the loss during training, it first uses random noise to get the noisy input image, then do the model prediction, calculate the noise loss between the model prediction and the random noise via mean square error. Since we know exactly what noise was added in the forward process, we can train the model to identify and predict that specific noise pattern. The model learns to recognize patterns in how noise transforms images at different timesteps. When given a noisy image, it can identify what noise was likely added, because the corruption process follows consistent mathematical rules.

Training code:
After setting up the hyperparameters from config file, it gets the training data through data loader. In the data loader, it finds the specific input image according to the file name in the json file, and the corresponding labels are also stored in a dictionary. For training convenience, there are two lists to store the transition from label to index and vice versa. The transform function is also defined here, except for the required normalization, random horizontal flipping is introduced to improve generalization by preventing the model from learning orientation-specific features.

Next, it calls the UNet module and diffusion module mentioned above. The optimizer here uses the AdamW which has the weight decay property preventing for large gradient update.

Since a portion of forward and denoising process have been defined in the diffusion process, in the training code, it just uses a batch to compute the loss value with p_losses function in diffusion process, then do the backpropagation.
The batch size use here is 64, the epochs is 2000 and learning rate is 0.0002.

Evaluation code and testing code:
The evaluation inherits the given evaluator, it wraps the evaluator within the evaluator interface which contains the normalization method and evaluate_images function.

The evaluate_images captures a batch to normalize the images, then calls the given evaluator to calculate the accuracy of sampled images from diffusion process. Since the returning accuracy is the average of one batch, there are some calculations to transfer the accuracy to actual accuracy by multiplying batch size.

The main inference code is in the evaluate_model function, there are three tasks in this function. First, define the evaluator interface, second, use the testing labels to generate the output images, finally, calculate the accuracy with evaluate_images function.

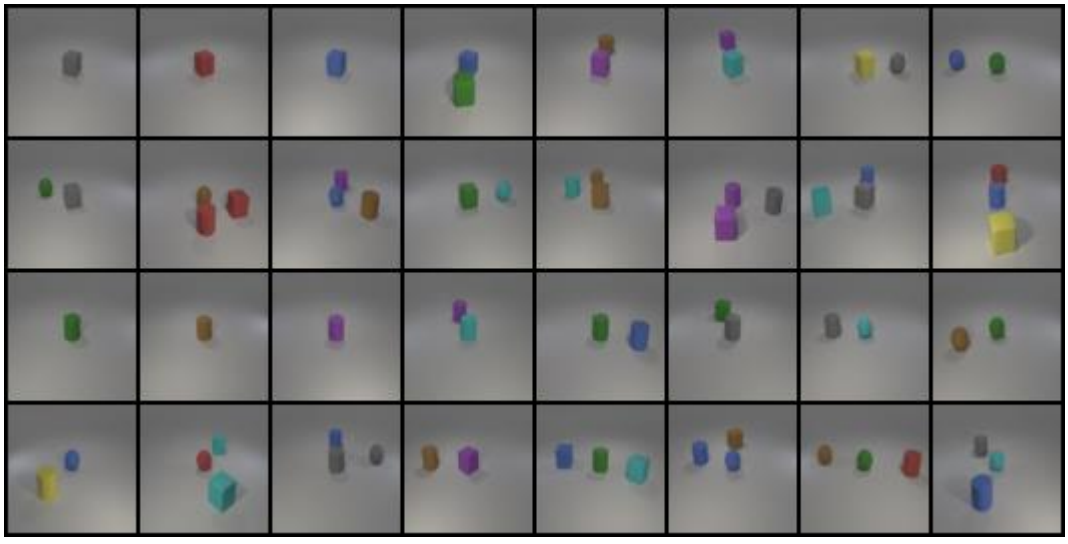
Testing code:
Most of the work in inference has been defined in the evaluation code, so it just defines the testing dataset, new testing dataset, the trained model and diffusion module, then call the evaluate_model function to get the accuracy.
After evaluation completed, it will call the visualization tool on demand.

Visualization code:
The visualize_denoising_process function plots the denoising process with a given specific label testcase, just like the evaluation, it first transfers the label to one-hot vector, then use the diffusion module to denoise the image, it will record the denoise process for 8 timesteps.
The plot_loss_curve function is just plot the training curve.

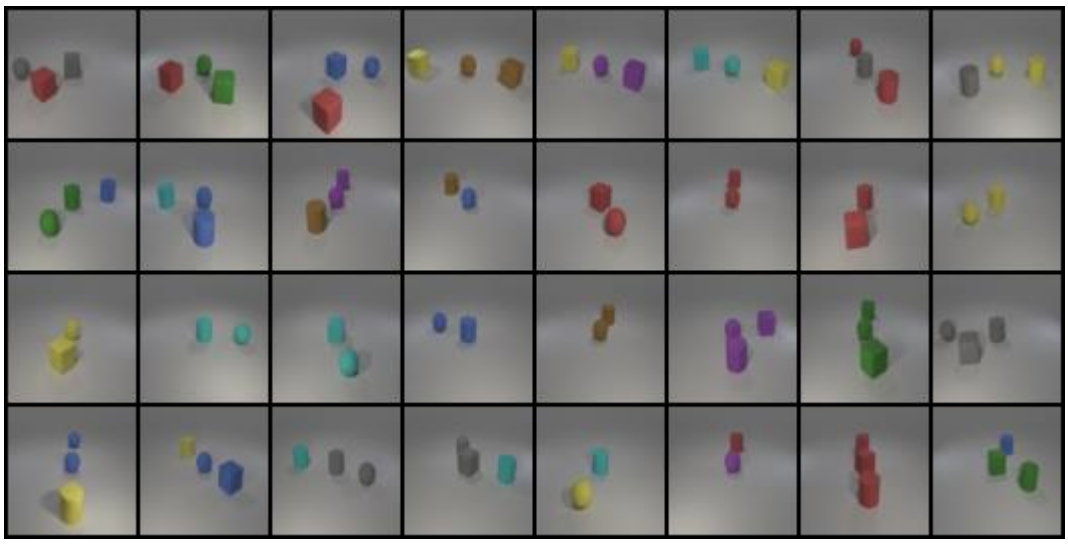
● Results and discussion:

Show your synthetic image grids (total 16%: 8% * 2 testing data) and a denoising process image with the label set ["red sphere", "cyan cylinder", "cyan cube"]:

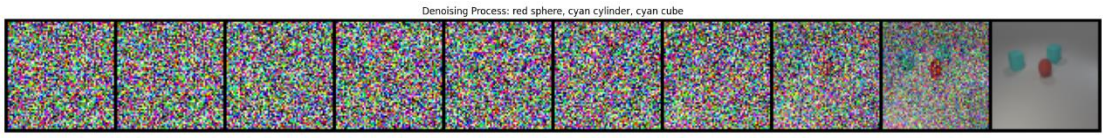
First is the linear noise schedule result:
Testing data grid:



New testing data grid:

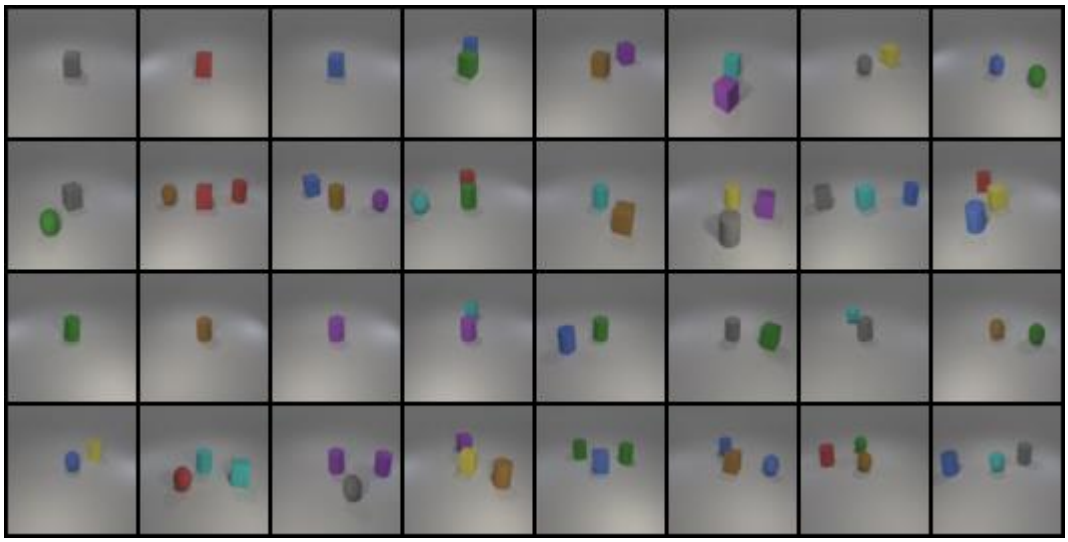


Denoising process for linear schedule:

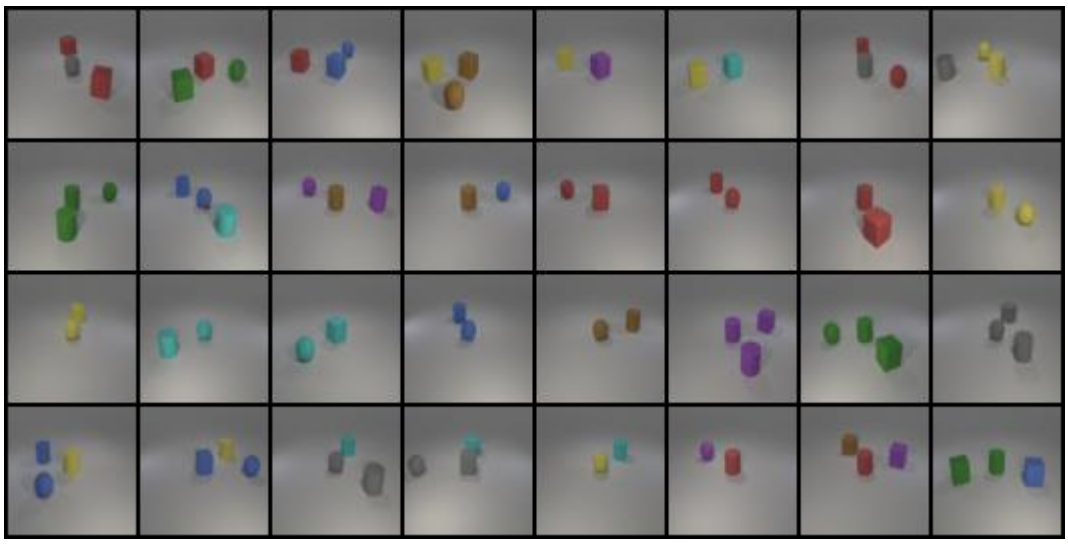


Then here is the result of cosine noise schedule:

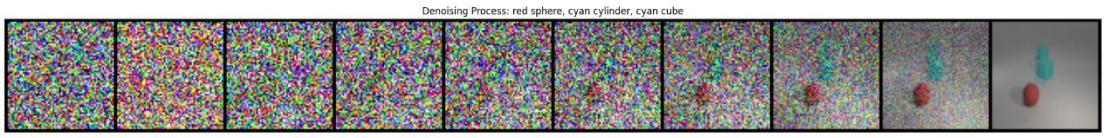
Testing data grid:



New testing data grid:

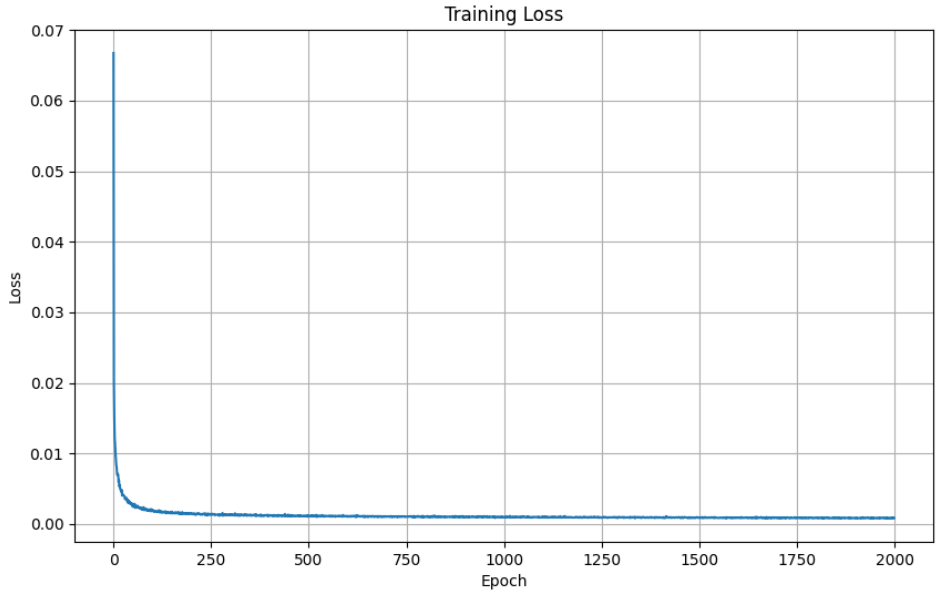


Denoising process for cosine noise schedule:



o Discussion of your extra implementations or experiments (10%):

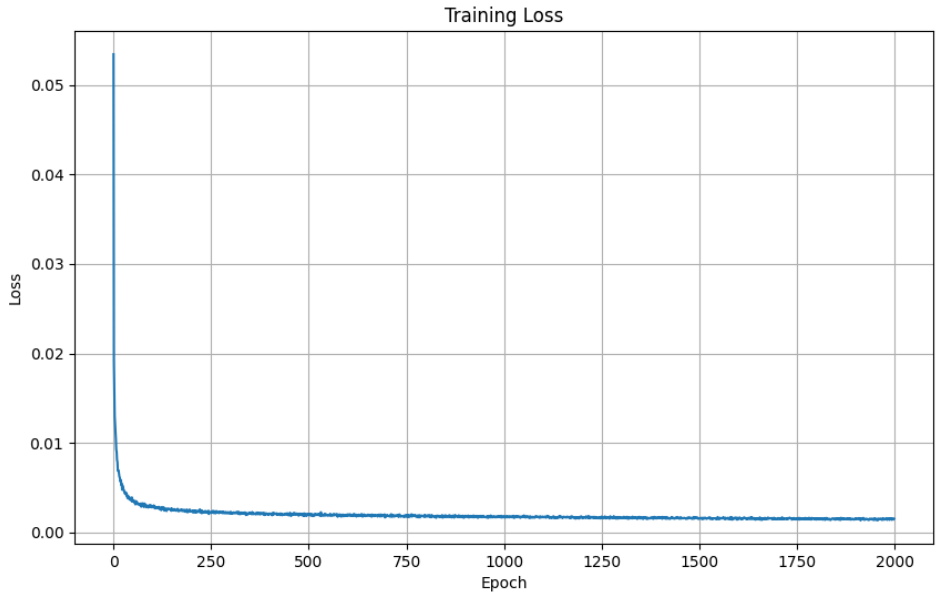
Here is the first training strategy, that is, the linear noise schedule.



It shows that the loss value drops rapidly, it seems that the diffusion model is easier to train than all the other models in previous labs.

The model converges at epoch 1000, it seems that training for 2000 epochs is too long.

And there is the cosine noise schedule.



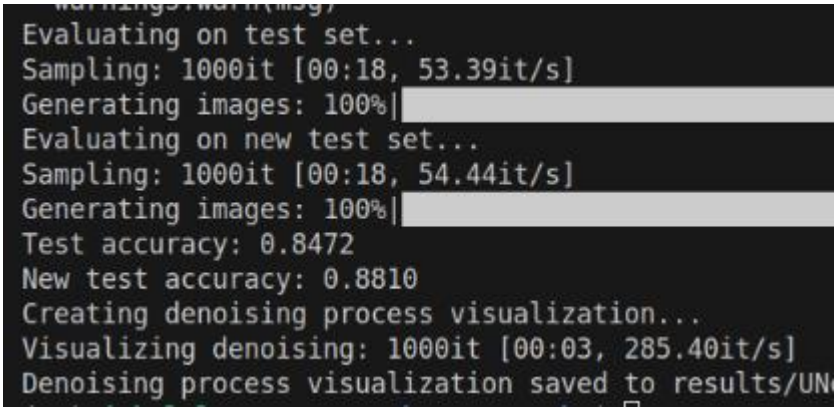
It shows a similar result compared to the linear noise schedule, but the loss value converges at a little higher value than the linear. Besides, the speed of convergence is also slower than the linear, this implies that the cosine noise schedule has smoother noise scheduling to make the model learn slowly at the first few steps and achieve more stable theoretically.

Experimental results:

- Classification accuracy on test.json and new test.json:

- Show your accuracy screenshots:

  The accuracies of test.json and new test.json for linear noise schedule:



And here is the accuracy of cosine noise schedule: