312554012 王偉誠 Lab6 : Let's Play DDPM

1. Introduction

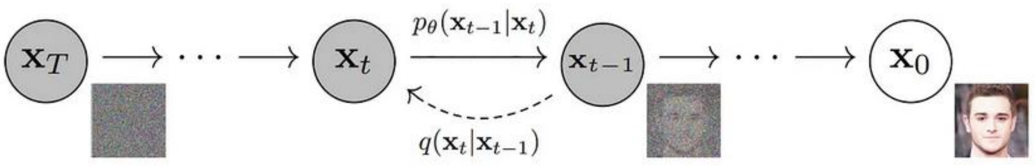
Generating images of three-dimensional geometric shapes using conditional DDPM. The images encompass a repertoire of 24 distinct geometric patterns, comprising three distinct shapes and eight varied colors. Furthermore, each image may encompass between one to three of these geometric entities. Multiple labels are assignable to a single image, thereby facilitating their conversion into one-hot vectors which subsequently undergo embedding within the UNet architecture for training purposes. During the sampling phase, directives delineated within the 'test.json' and 'new_test.json' files dictate the specific geometric configurations and quantities to be synthesized within each image. The resultant images are then subject to evaluation by the provided evaluator, judiciously ascertaining their adherence to stipulated criteria.

2. Implementation details

A. Describe how you implement your model, including your choice of DDPM, UNet architectures, noise schedule, and loss functions.

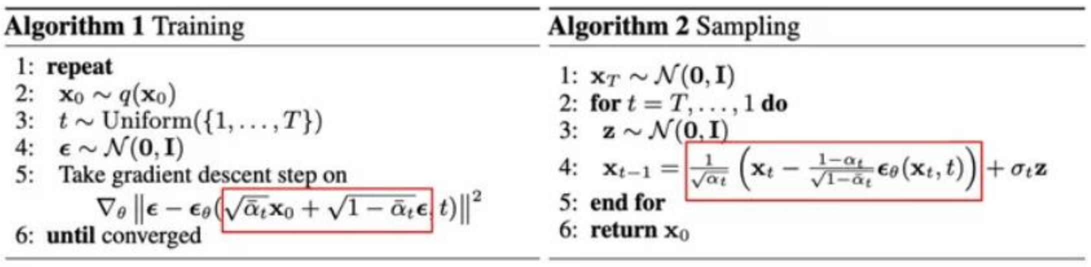
- **Main structure**

The fundamental essence of a Diffusion Model lies in its capacity to acquire knowledge about an incremental denoising process. This process can be delineated by representing each stage of the Diffusion Model as a Markov chain. During training, the integration of minute Gaussian noise is rooted in the application of Gaussian noise, which serves as the source of perturbation. The network's parameter set, denoted as θ, encapsulates the quintessence of denoising strategies, essentially encompassing the techniques for effective noise reduction, as illustrated in the accompanying diagram.


$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_{t-1}$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{x}_0$

Therefore, the optimization objective of DDPM is to align the predicted noise with actual noise, which implies that during training, a random training sample is selected. Then, a value of t is sampled uniformly from the range of 1 to T. Subsequently, random noise is generated and used to compute the current noisy data (illustrated by the red box). This noisy data is fed into the network to predict the noise. The L2 loss between the generated noise and the predicted noise is then calculated. The gradients are computed and subsequently utilized to update the network.

During the sampling process, a novel sample is generated beginning with a randomly initialized noise vector. Leveraging the trained network, the predicted noise is estimated. The mean of the conditional distribution (indicated within the red box) is calculated. This mean, in conjunction with the corresponding standard deviation, is then multiplied by a random noise vector iteratively until t reaches 0, finalizing the generation of the new sample (without adding noise in the last step).



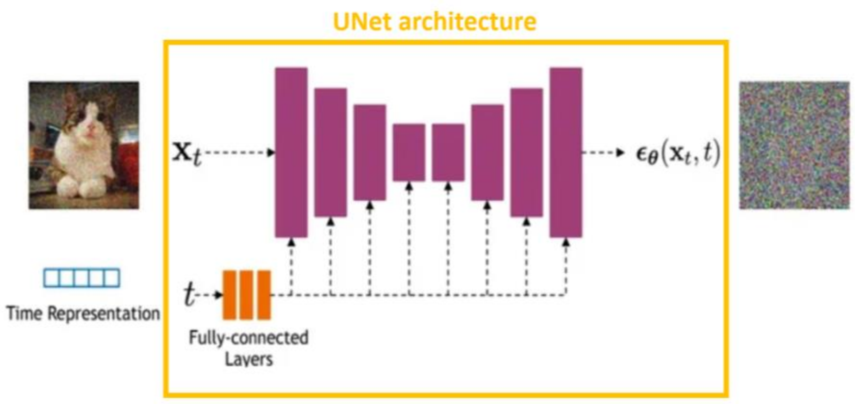
Algorithm 1 Training

```
1: repeat
2:   $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3:   $t \sim \mathrm{Uniform}(\{1, \dots, T\})$
4:   $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5:   Take gradient descent step on
         $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\boxed{\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}}, t) \right\|^2$
6: until converged
```

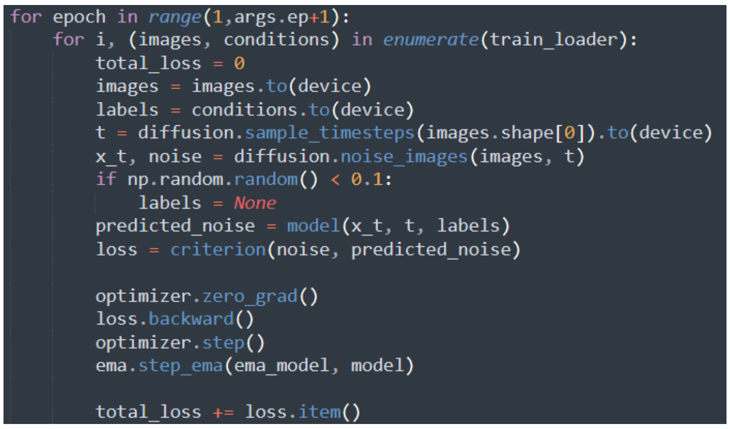
Algorithm 2 Sampling

```
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: for $t = T, \dots, 1$ do
3:   $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
4:   $\mathbf{x}_{t-1} = \boxed{\frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right)} + \sigma_t \mathbf{z}$
5: end for
6: return $\mathbf{x}_0$
```

During the training process, given that the noisy images and original images share the same dimensions, DDPM adopts an architecture akin to an AutoEncoder. As depicted in the diagram, the UNet architecture is employed. Within this architecture, a time embedding mechanism is introduced to encode the timestep information into the network. Moreover, label embedding is incorporated to facilitate the generation of images corresponding to specific conditions. This multifaceted approach enhances the network's ability to effectively generate images conditioned on relevant factors.

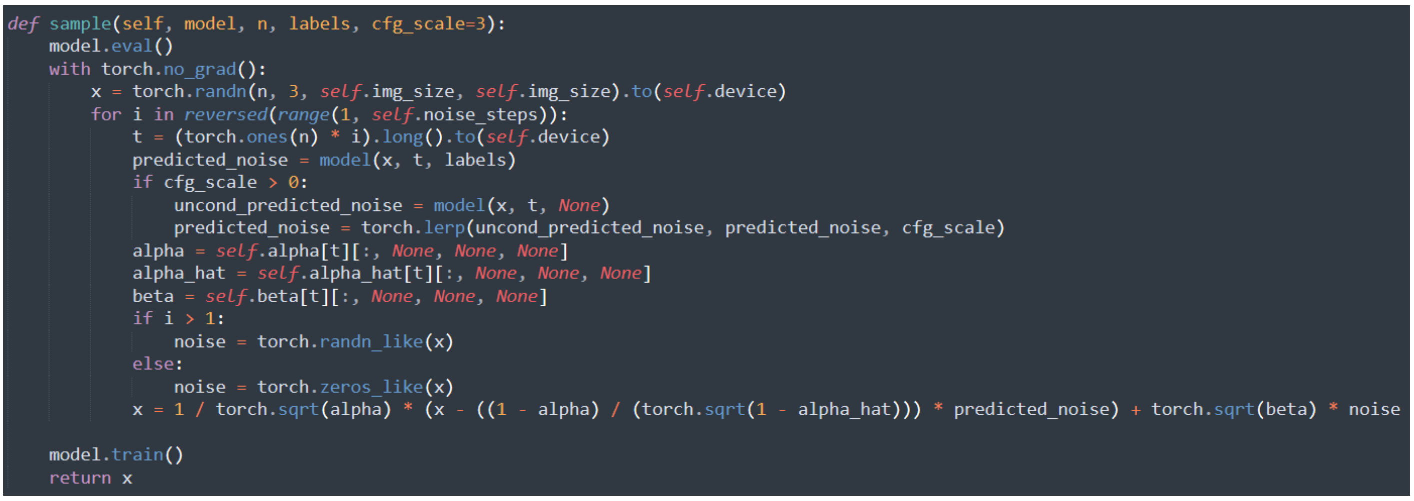


Training process (Algorithm 1) implementation:



```python
for epoch in range(1,args.ep+1):
    for i, (images, conditions) in enumerate(train_loader):
        total_loss = 0
        images = images.to(device)
        labels = conditions.to(device)
        t = diffusion.sample_timesteps(images.shape[0]).to(device)
        x_t, noise = diffusion.noise_images(images, t)
        if np.random.random() < 0.1:
            labels = None
        predicted_noise = model(x_t, t, labels)
        loss = criterion(noise, predicted_noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        ema.step_ema(ema_model, model)

        total_loss += loss.item()
```

Sampling process (Algorithm 2) implementation:



```python
def sample(self, model, n, labels, cfg_scale=3):
    model.eval()
    with torch.no_grad():
        x = torch.randn(n, 3, self.img_size, self.img_size).to(self.device)
        for i in reversed(range(1, self.noise_steps)):
            t = (torch.ones(n) * i).long().to(self.device)
            predicted_noise = model(x, t, labels)
            if cfg_scale > 0:
                uncond_predicted_noise = model(x, t, None)
                predicted_noise = torch.lerp(uncond_predicted_noise, predicted_noise, cfg_scale)
            alpha = self.alpha[t][:, None, None, None]
            alpha_hat = self.alpha_hat[t][:, None, None, None]
            beta = self.beta[t][:, None, None, None]
            if i > 1:
                noise = torch.randn_like(x)
            else:
                noise = torch.zeros_like(x)
            x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 - alpha_hat))) * predicted_noise) + torch.sqrt(beta) * noise

    model.train()
    return x
```

- **Loss function**

  The loss function employed is the Mean Squared Error (MSE) Loss, which is used to compute the L2 loss between the generated noise and the predicted noise.

- **Prediction type**

  Referring to Equation (8) in "Denoising Diffusion Probabilistic Models [1]", we can utilize this equation for predicting the noisy samples.

$$L_{t-1} = \mathbb{E}_q\left[\frac{1}{2\sigma_t^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t, t)\|^2\right] + C \qquad (8)$$

  However, the paper found that omitting the preceding weight term ($\frac{1}{2\sigma_t^2}$) actually assists the network in focusing more on challenging samples.

Consequently, the final loss formulation can be expressed as the "simplified noise predicting" approach.

$$L_{t-1} = \mathbb{E}_{\mathbf{x}_0,\,\epsilon}\left[\left\|\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\, t\right)\right\|_2^2\right]$$

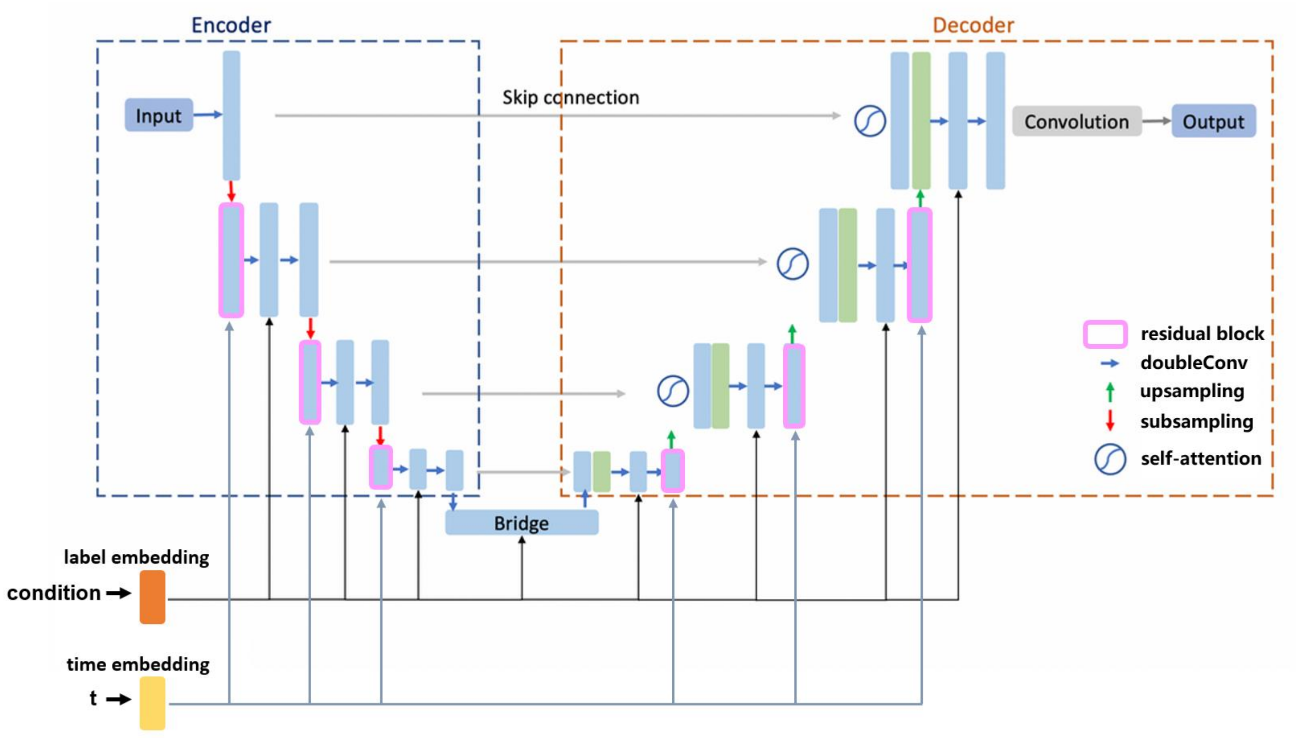
And indeed, the approach I am utilizing is the simplified noise predicting.

- **Noise schedule**

  I have opted for a sigmoid noise schedule in my approach. The rationale behind this choice stems from the limitations of the linear noise schedule, where data degradation occurs too rapidly. In the context of "Improved Denoising Diffusion Probabilistic Models [2]," the effectiveness of the cosine noise schedule is highlighted. Through experimentation, I have ascertained that coupling my diffusion model architecture with a sigmoid noise schedule yields superior results. Hence, I have employed the sigmoid noise schedule, defined by the following formula:

```
# sigmoid betas chedule
def prepare_noise_schedule(self):
    betas = torch.linspace(-6, 6, self.noise_steps)
    return torch.sigmoid(betas) * (self.beta_end - self.beta_start) + self.beta_start
```

- **UNet architecture**



My UNet architecture, as described above, incorporates an encoder section composed of several doubleConv layers, residual blocks, and subsampling operations. The primary aim of this configuration is to reduce the spatial dimensions (height and width) of the features via downsampling. The decoder, conversely, includes doubleConv layers, residual blocks, and upsampling operations, facilitating the gradual restoration of the previously compressed features.

Additionally, self-attention mechanisms and skip connections are integrated to augment the training process. Self-attention enhances the network's capacity for global modeling, while skip connections involve the concatenation of intermediate features obtained from the encoder, promoting network optimization.

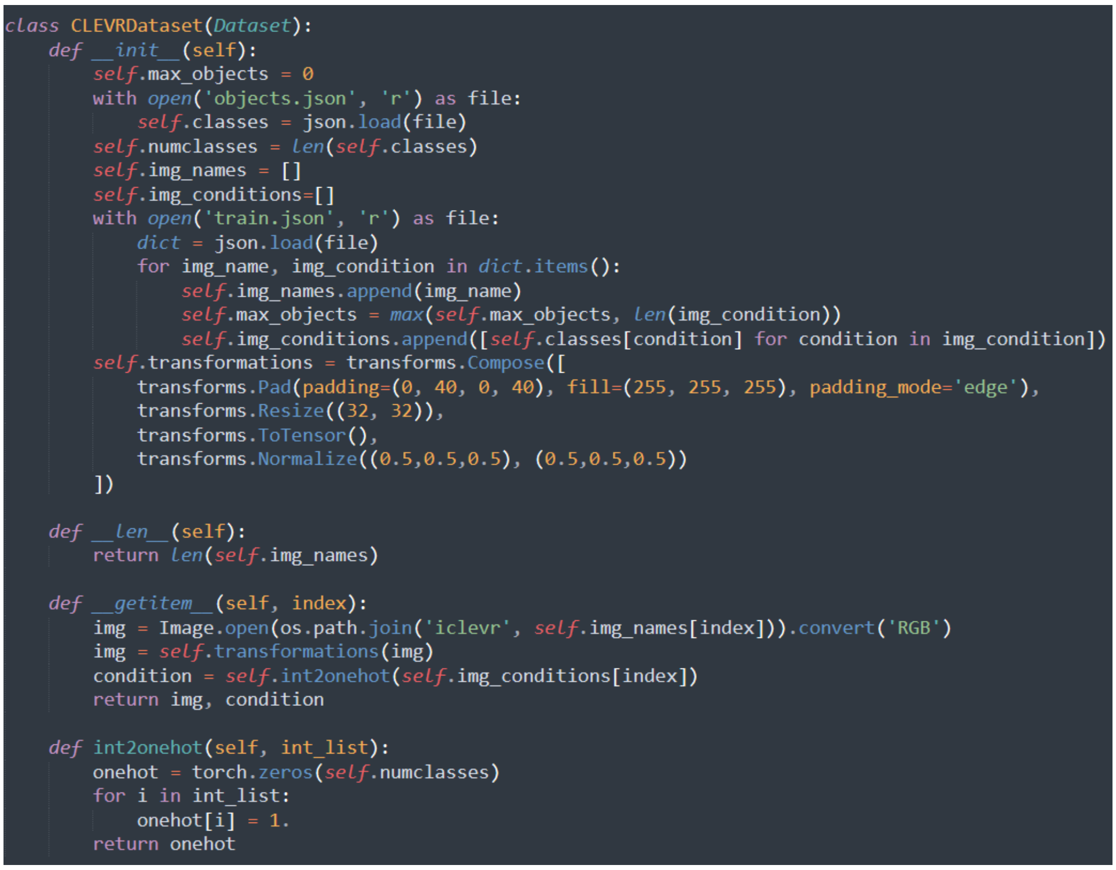
Time embedding and label embedding are strategically inserted within the residual block and subsampling/upsampling sections, respectively. These embeddings serve to provide the UNet architecture with temporal and label-related information, thereby enhancing training efficiency by equipping the network with relevant context during the learning process.

■ Dataloader

The dataloader operates in two distinct modes, catering to both training and testing datasets. The illustration below depicts the segment related to retrieving data from the training dataset.

In this process, the "object.json" file is initially parsed, containing the codes (ranging from 0 to 23) corresponding to the three-dimensional geometric shapes. Subsequently, the "train.json" file is processed, revealing the images along with their corresponding geometric shapes. The images undergo preprocessing, specifically padded to achieve a square aspect ratio and resized to dimensions of 32x32. This resolution reduction is an adjustment made due to memory constraints; I've found that in my experimental environment, 64x64 images lead to out-of-memory issues during training. Finally, these images are transformed into tensors and normalized.

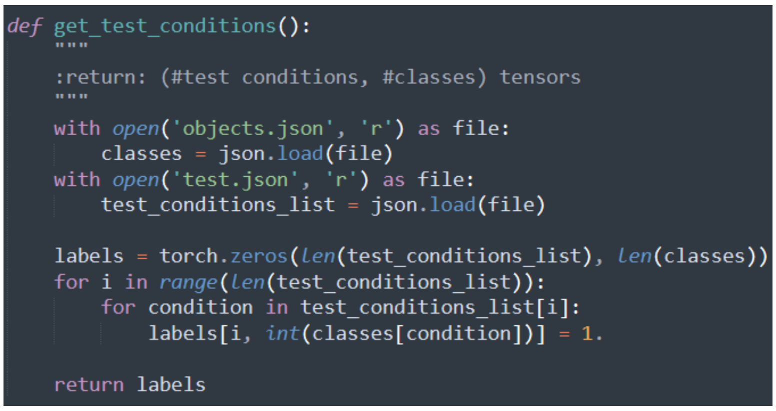
Regarding the conditions, the process involves converting them into one-hot vectors. This entails assigning a value of 1 to the index corresponding to the label present in the image and assigning 0 to the rest of the indices. This manipulation enables effective incorporation of the geometric shape information into the dataset for training.



```python
class CLEVRDataset(Dataset):
    def __init__(self):
        self.max_objects = 0
        with open('objects.json', 'r') as file:
            self.classes = json.load(file)
        self.numclasses = len(self.classes)
        self.img_names = []
        self.img_conditions=[]
        with open('train.json', 'r') as file:
            dict = json.load(file)
            for img_name, img_condition in dict.items():
                self.img_names.append(img_name)
                self.max_objects = max(self.max_objects, len(img_condition))
                self.img_conditions.append([self.classes[condition] for condition in img_condition])
        self.transformations = transforms.Compose([
            transforms.Pad(padding=(0, 40, 0, 40), fill=(255, 255, 255), padding_mode='edge'),
            transforms.Resize((32, 32)),
            transforms.ToTensor(),
            transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))
        ])

    def __len__(self):
        return len(self.img_names)

    def __getitem__(self, index):
        img = Image.open(os.path.join('iclevr', self.img_names[index])).convert('RGB')
        img = self.transformations(img)
        condition = self.int2onehot(self.img_conditions[index])
        return img, condition

    def int2onehot(self, int_list):
        onehot = torch.zeros(self.numclasses)
        for i in int_list:
            onehot[i] = 1.
        return onehot
```

Retrieving data from the testing dataset closely mirrors the process used for the training dataset. The diagram below illustrates this operation.

Similar to the training dataset, the "object.json" file is read, containing the codes (ranging from 0 to 23) corresponding to the three-dimensional geometric shapes. Subsequently, the "test condition" is read and transformed into a one-hot vector. This conversion involves assigning a value of 1 to the index corresponding to the label present in the image and setting the rest of the indices to 0. This preparation of the testing dataset ensures that the geometric shape information is accurately represented for evaluation purposes.



```python
def get_test_conditions():
    """
    :return: (#test conditions, #classes) tensors
    """
    with open('objects.json', 'r') as file:
        classes = json.load(file)
    with open('test.json', 'r') as file:
        test_conditions_list = json.load(file)

    labels = torch.zeros(len(test_conditions_list), len(classes))
    for i in range(len(test_conditions_list)):
        for condition in test_conditions_list[i]:
            labels[i, int(classes[condition])] = 1.

    return labels
```

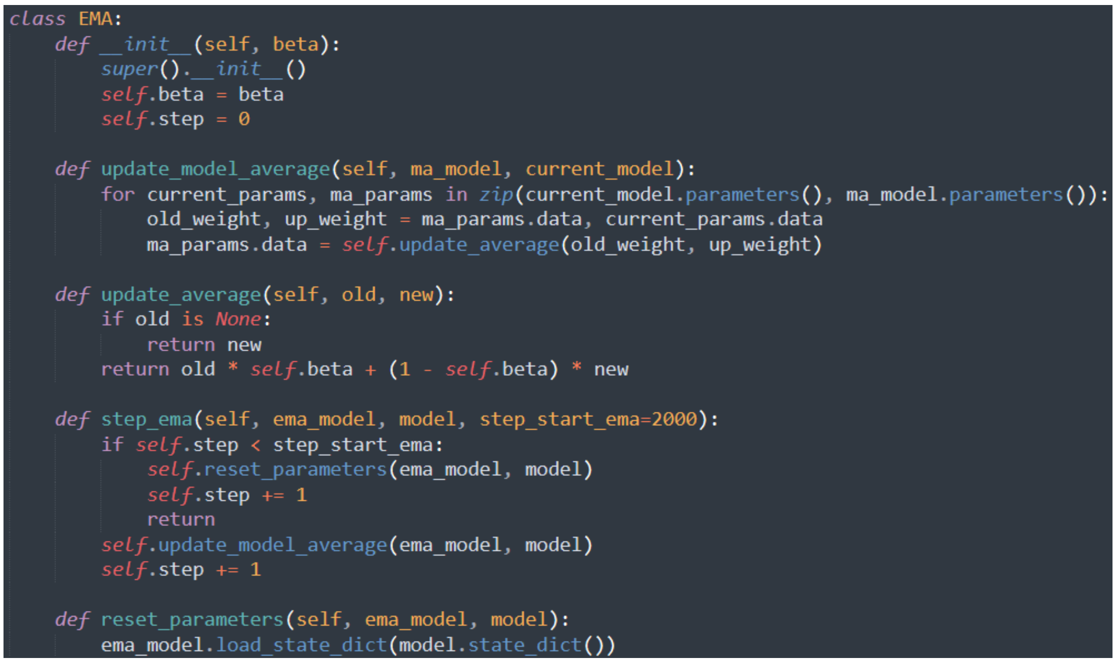
■ Advanced technique

Exponential Moving Average (EMA)

In our pursuit of stable training with reduced oscillations, it is desirable to mitigate excessive fluctuations. The Exponential Moving Average (EMA) technique, essentially, serves as a mechanism for achieving smoother training. EMA operates by creating a replica of the initial model weights from the main model. During updates, it employs a moving average derived from the main model to update the EMA model. Consequently, the update equation is illustrated as depicted in the diagram.

$$w = \beta \cdot w_{old} + (1-\beta) \cdot w_{new} \quad , \quad \beta = 0.995$$

Code implementation:



```
class EMA:
    def __init__(self, beta):
        super().__init__()
        self.beta = beta
        self.step = 0

    def update_model_average(self, ma_model, current_model):
        for current_params, ma_params in zip(current_model.parameters(), ma_model.parameters()):
            old_weight, up_weight = ma_params.data, current_params.data
            ma_params.data = self.update_average(old_weight, up_weight)

    def update_average(self, old, new):
        if old is None:
            return new
        return old * self.beta + (1 - self.beta) * new

    def step_ema(self, ema_model, model, step_start_ema=2000):
        if self.step < step_start_ema:
            self.reset_parameters(ema_model, model)
            self.step += 1
            return
        self.update_model_average(ema_model, model)
        self.step += 1

    def reset_parameters(self, ema_model, model):
        ema_model.load_state_dict(model.state_dict())
```

Classifier Free Guidance (CFG)

In the context of this assignment, our objective is to concurrently retain the model's generative capabilities while ensuring accurate generation of images corresponding to specific conditions. Addressing the concern of posterior collapse, where the model disregards conditional information and generates arbitrary images, the paper "Classifier-Free Diffusion Guidance [3]" introduces an alternative approach to sidestep this issue. This approach involves the abandonment of an external classifier and proposes an equivalent structure. This modification empowers the diffusion model to successfully undertake the task of conditional generation.

In practical terms, the method encompasses two types of input sampling: conditional (involving random Gaussian noise and label embedding) and unconditional. Both types of input are fed into the same diffusion model, enabling it to possess the dual capabilities of unconditional and conditional generation. This mechanism significantly contributes to addressing the challenge of preserving both the model's generative prowess and its fidelity to conditional cues.

The original approach for updating the noise is illustrated as follows:

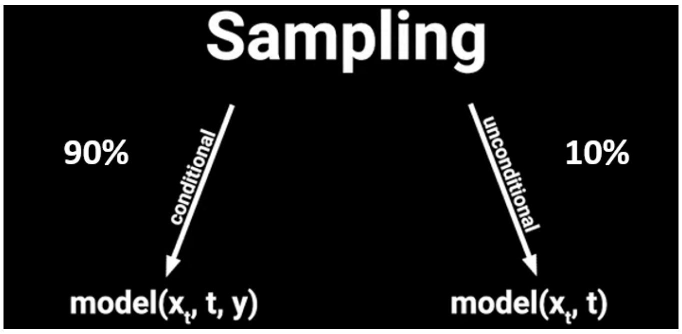
$$\epsilon_\theta(x_t, t) \sim \epsilon_\theta(x_t) - \sqrt{1-\overline{\alpha}_t}\,\nabla_{x_t} logp_\phi(y|x_t)$$

In the classifier-free approach, the posterior update, as illustrated in the diagram, is substituted with another approximate equivalent structure.

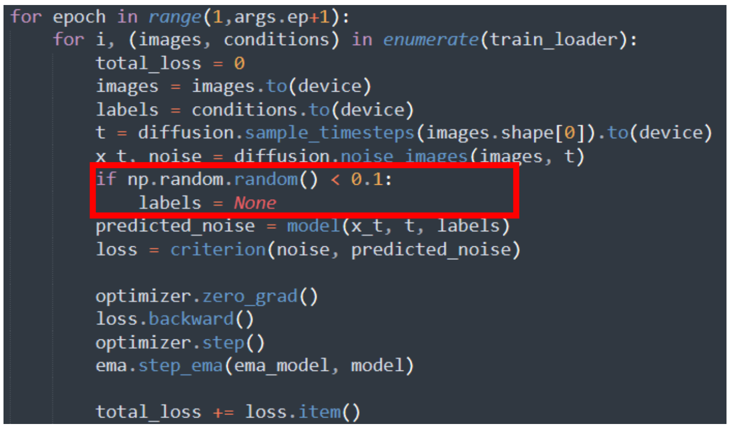
$$\hat{\epsilon}_\theta(x_t|y) = \epsilon_\theta(x_t) + s \cdot (\epsilon_\theta(x_t, y) - \epsilon_\theta(x_t))$$

The conditional input is denoted as described, while the unconditional input (with the condition y set as NULL) is introduced. The difference between these two inputs is then multiplied by a coefficient to replace the original term.

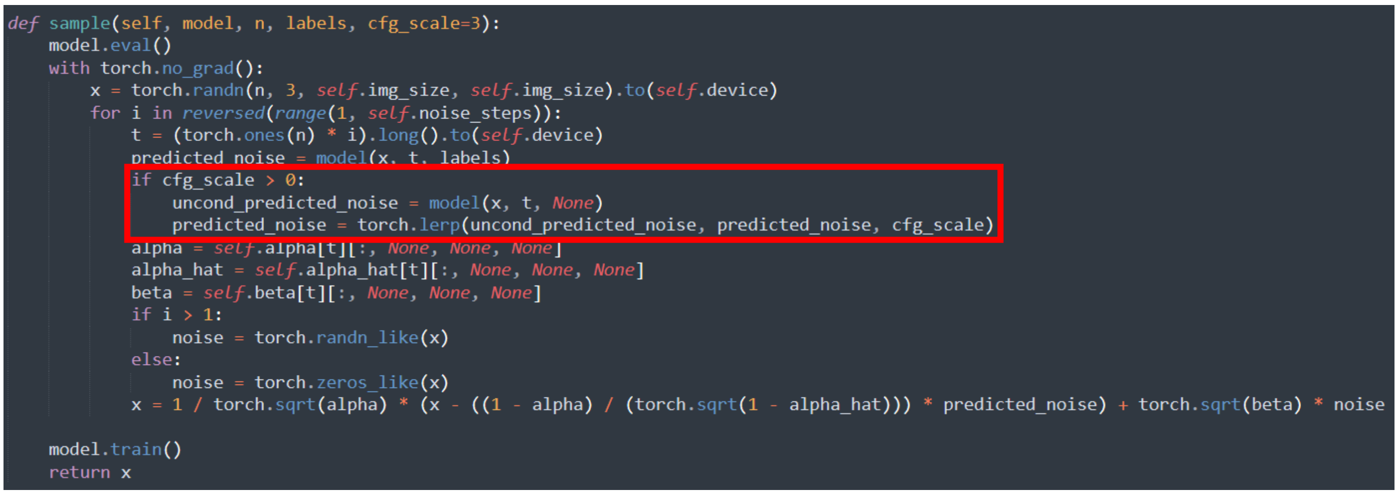
In the practical implementation, the sampling proportions for these two types are distributed as follows: 90% for conditional and 10% for unconditional. Furthermore, the predicted noise is linearly interpolated between the unconditional predicted noise and the conditional predicted noise, contributing to the gradual transition between these two modalities during the training process. This approach, grounded in a 9:1 sampling ratio and a linear interpolation strategy, facilitates effective training while enhancing the diffusion model's ability to balance between unconditional and conditional image generation.



Training and sampling code implementation:



```
for epoch in range(1,args.ep+1):
    for i, (images, conditions) in enumerate(train_loader):
        total_loss = 0
        images = images.to(device)
        labels = conditions.to(device)
        t = diffusion.sample_timesteps(images.shape[0]).to(device)
        x_t, noise = diffusion.noise_images(images, t)
        if np.random.random() < 0.1:
            labels = None
        predicted_noise = model(x_t, t, labels)
        loss = criterion(noise, predicted_noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        ema.step_ema(ema_model, model)

        total_loss += loss.item()
```



```
def sample(self, model, n, labels, cfg_scale=3):
    model.eval()
    with torch.no_grad():
        x = torch.randn(n, 3, self.img_size, self.img_size).to(self.device)
        for i in reversed(range(1, self.noise_steps)):
            t = (torch.ones(n) * i).long().to(self.device)
            predicted_noise = model(x, t, labels)
            if cfg_scale > 0:
                uncond_predicted_noise = model(x, t, None)
                predicted_noise = torch.lerp(uncond_predicted_noise, predicted_noise, cfg_scale)
            alpha = self.alpha[t][:, None, None, None]
            alpha_hat = self.alpha_hat[t][:, None, None, None]
            beta = self.beta[t][:, None, None, None]
            if i > 1:
                noise = torch.randn_like(x)
            else:
                noise = torch.zeros_like(x)
            x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 - alpha_hat))) * predicted_noise) + torch.sqrt(beta) * noise

    model.train()
    return x
```

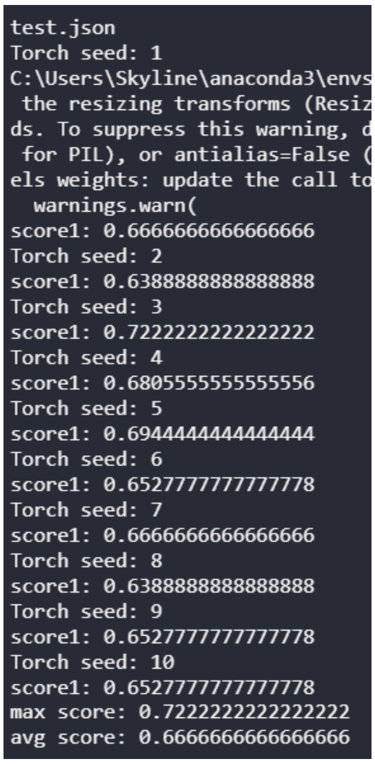
B. Specify the hyperparameters (learning rate, epochs, etc.)

- epoch size: 300
- batch size: 48
- learning rate: 3e-4
- optimizer: AdamW
- loss function: MSE
- noise steps T: 1000
- beta start β_1: 1e-4
- beta end β_T: 0.02

4. Results and discussion

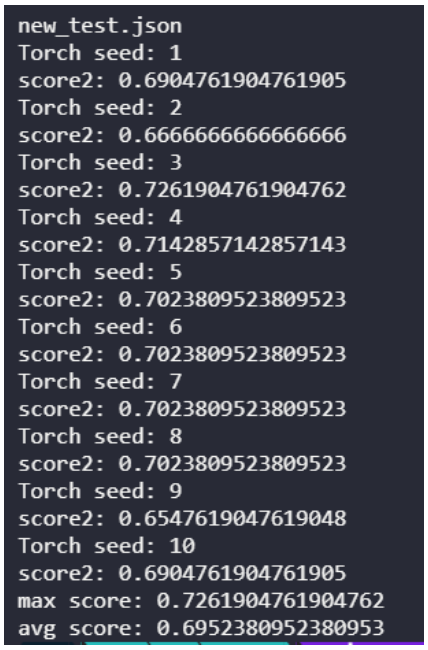
A. Show your accuracy screenshot based on the testing data

test.json (0.72222)



```
test.json
Torch seed: 1
C:\Users\Skyline\anaconda3\envs
 the resizing transforms (Resiz
ds. To suppress this warning, d
 for PIL), or antialias=False (
els weights: update the call to
  warnings.warn(
score1: 0.6666666666666666
Torch seed: 2
score1: 0.6388888888888888
Torch seed: 3
score1: 0.7222222222222222
Torch seed: 4
score1: 0.6805555555555556
Torch seed: 5
score1: 0.6944444444444444
Torch seed: 6
score1: 0.6527777777777778
Torch seed: 7
score1: 0.6666666666666666
Torch seed: 8
score1: 0.6388888888888888
Torch seed: 9
score1: 0.6527777777777778
Torch seed: 10
score1: 0.6527777777777778
max score: 0.7222222222222222
avg score: 0.6666666666666666
```

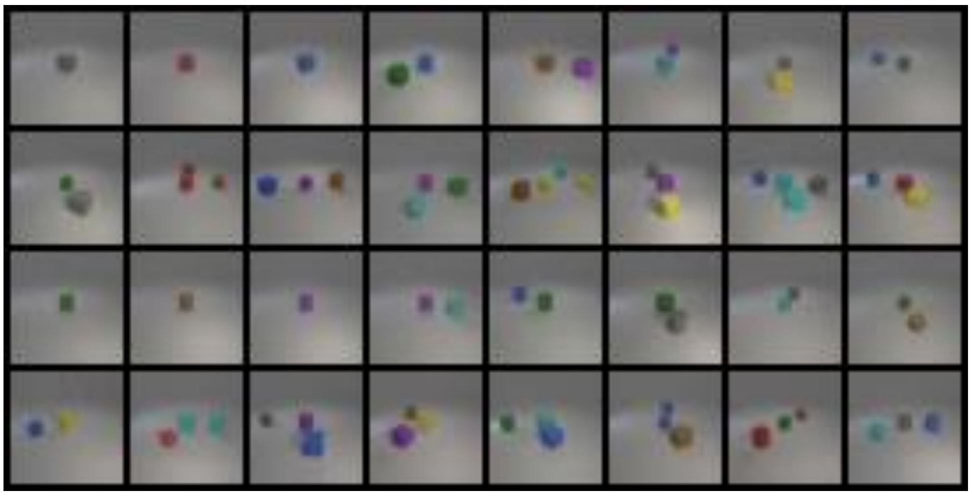
new_test.json(0.726)



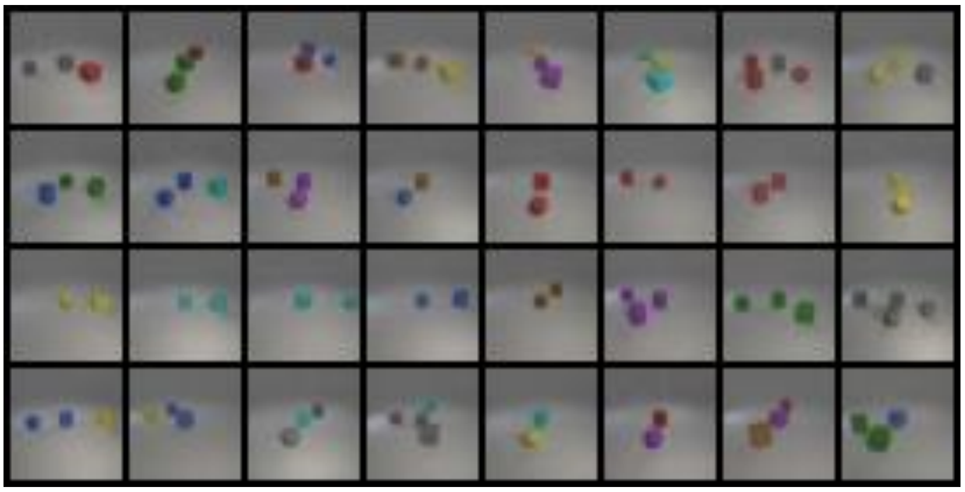
```
new_test.json
Torch seed: 1
score2: 0.6904761904761905
Torch seed: 2
score2: 0.6666666666666666
Torch seed: 3
score2: 0.7261904761904762
Torch seed: 4
score2: 0.7142857142857143
Torch seed: 5
score2: 0.7023809523809523
Torch seed: 6
score2: 0.7023809523809523
Torch seed: 7
score2: 0.7023809523809523
Torch seed: 8
score2: 0.7023809523809523
Torch seed: 9
score2: 0.6547619047619048
Torch seed: 10
score2: 0.6904761904761905
max score: 0.7261904761904762
avg score: 0.6952380952380953
```

B. Show your synthetic image grids and a progressive generation image

test.json (0.72222)



new_test.json(0.726)



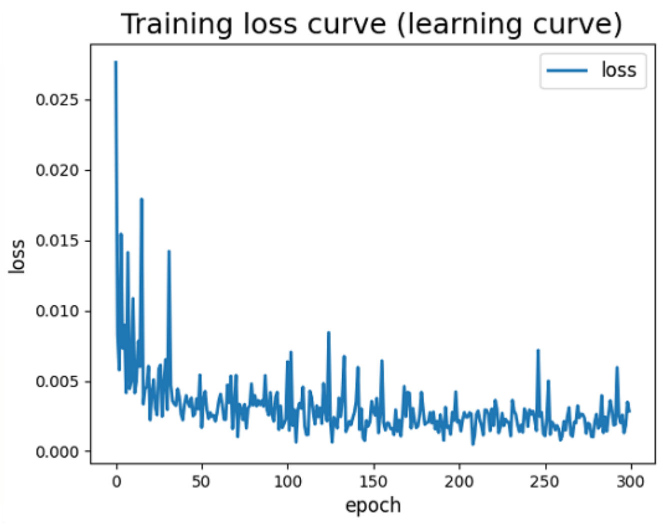
C. Discuss the results of different model architectures.

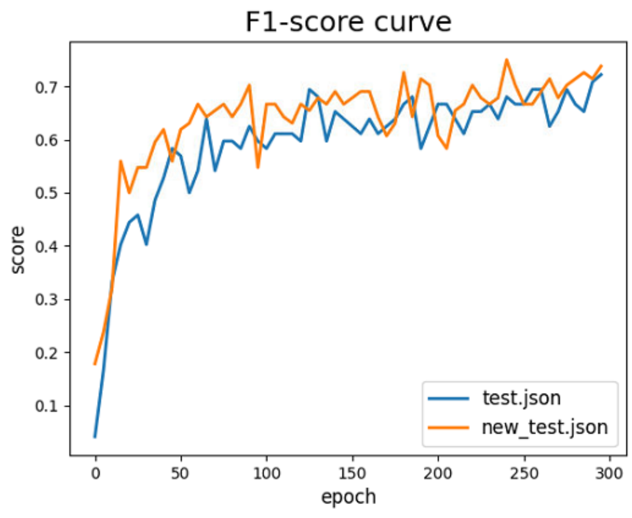
- I've implemented two approaches for label embedding:

  As mentioned in the previously discussed implementation details, label embedding is integrated within the subsampling/upsampling layers.

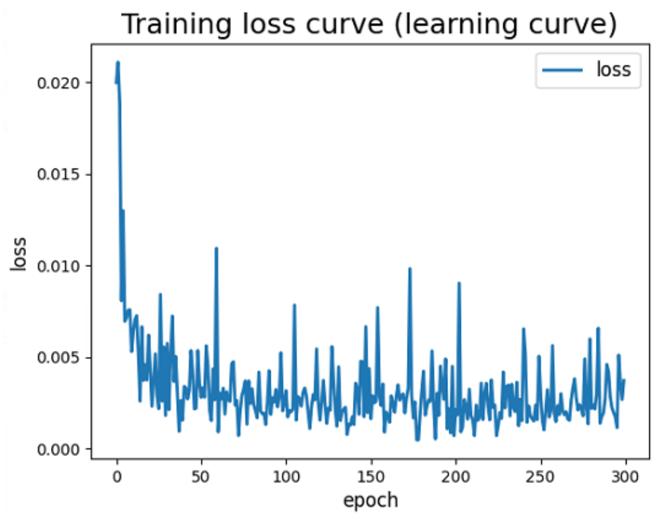
  An alternative approach involves expanding the label condition into dimensions matching the width and height of the image. This expanded label is then introduced as additional input channels during the training process. In other words, the original input shape of (batch_size, 3, 28, 28) is augmented with the label embedding, resulting in an input shape of (batch_size, 16, 28, 28).

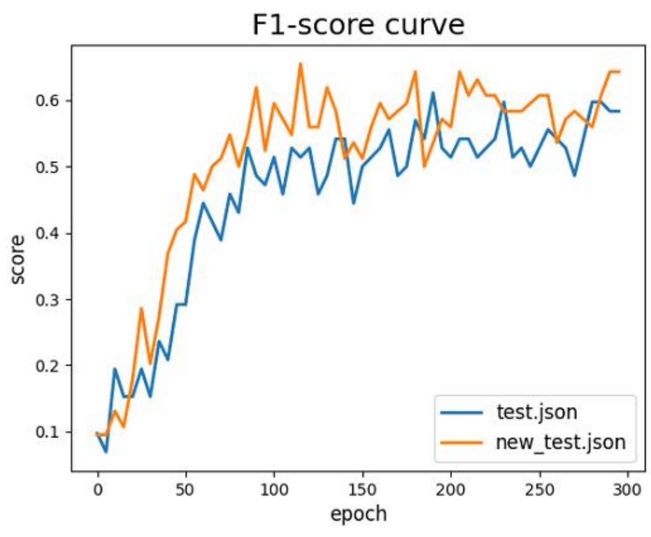
lable embedding on subsampling/upsampling block





label add as additional input channels

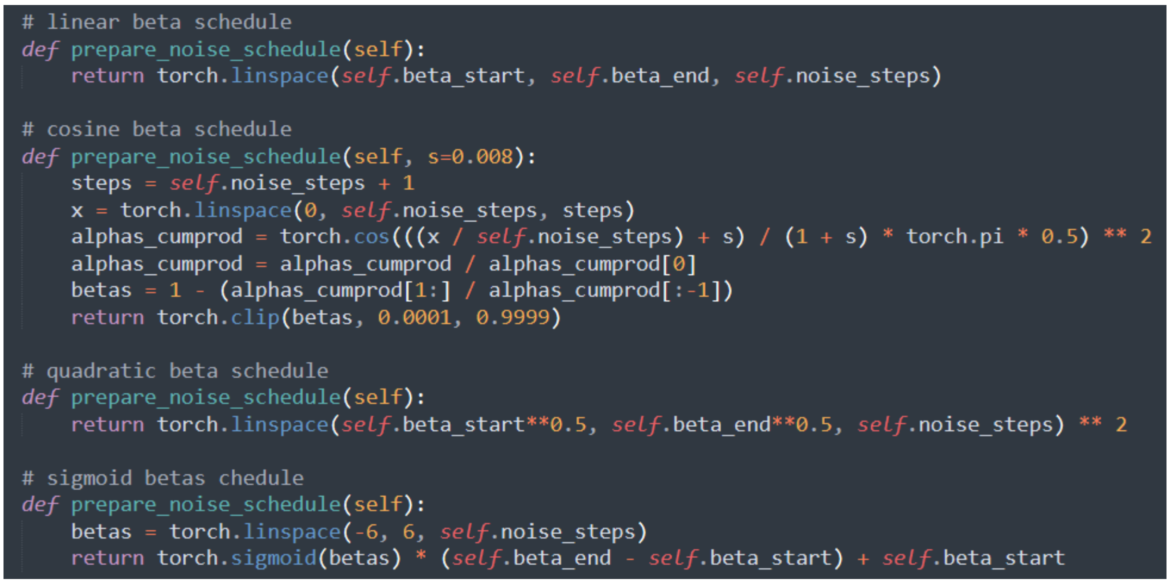




It's evident from observations that incorporating label embedding within the subsampling/upsampling blocks yields better results and contributes to a more stable training process. This is due to the fact that the label embedding information is present across multiple layers, akin to the approach used for time embedding. Conversely, the strategy of adding labels as additional input channels might suffer from information loss as the label condition is combined with the image's RGB channels and propagated through several layers of the UNet. This can lead to less favorable training outcomes due to the degradation of condition-related information over these layers. Your analysis highlights the significance of effectively integrating label information in a manner that aligns with the model's architecture and maintains its pertinence throughout the training process.

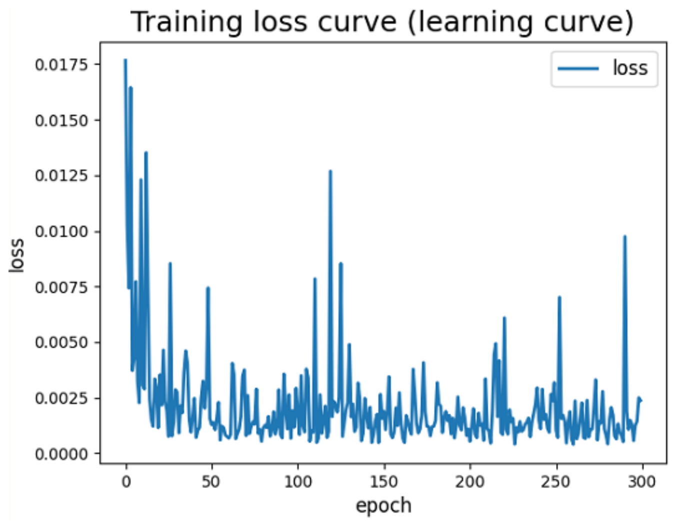
- Noise scheduling

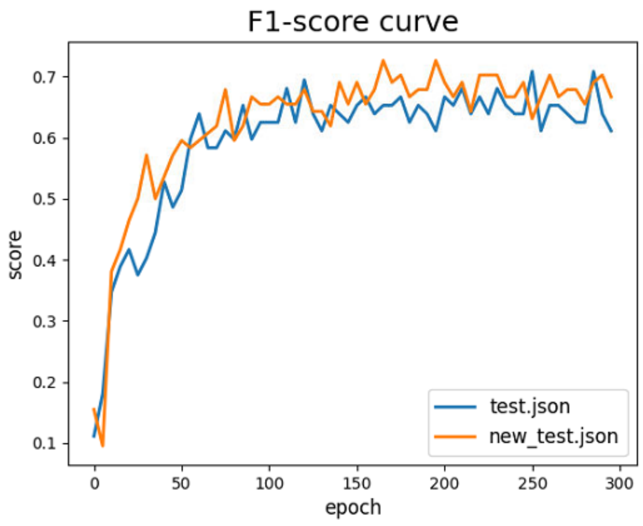
Noise scheduling significantly impacts the pace at which conditional DDPM data degradation occurs, which in turn affects the outcomes of the training process. As a result, I have conducted comparisons among different noise scheduling techniques. Specifically, I have examined three distinct approaches: linear, quadratic, and sigmoid noise schedules. Additionally, I experimented with cosine noise scheduling; however, its performance was subpar, possibly due to incorrect implementation or its unsuitability for my conditional DDPM architecture. Given these considerations, I will refrain from discussing the details of the cosine noise schedule.



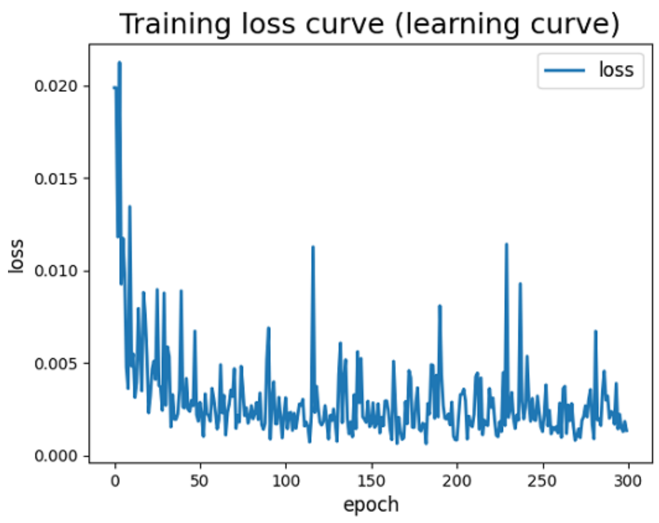
```python
# linear beta schedule
def prepare_noise_schedule(self):
    return torch.linspace(self.beta_start, self.beta_end, self.noise_steps)

# cosine beta schedule
def prepare_noise_schedule(self, s=0.008):
    steps = self.noise_steps + 1
    x = torch.linspace(0, self.noise_steps, steps)
    alphas_cumprod = torch.cos(((x / self.noise_steps) + s) / (1 + s) * torch.pi * 0.5) ** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return torch.clip(betas, 0.0001, 0.9999)

# quadratic beta schedule
def prepare_noise_schedule(self):
    return torch.linspace(self.beta_start**0.5, self.beta_end**0.5, self.noise_steps) ** 2

# sigmoid betas chedule
def prepare_noise_schedule(self):
    betas = torch.linspace(-6, 6, self.noise_steps)
    return torch.sigmoid(betas) * (self.beta_end - self.beta_start) + self.beta_start
```

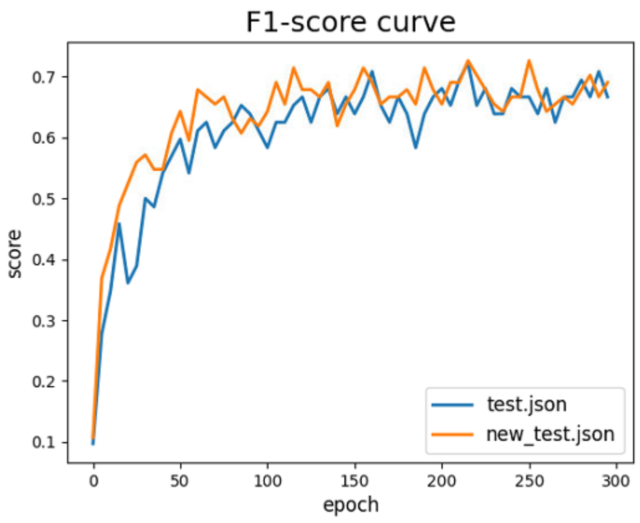
linear noise scheduling



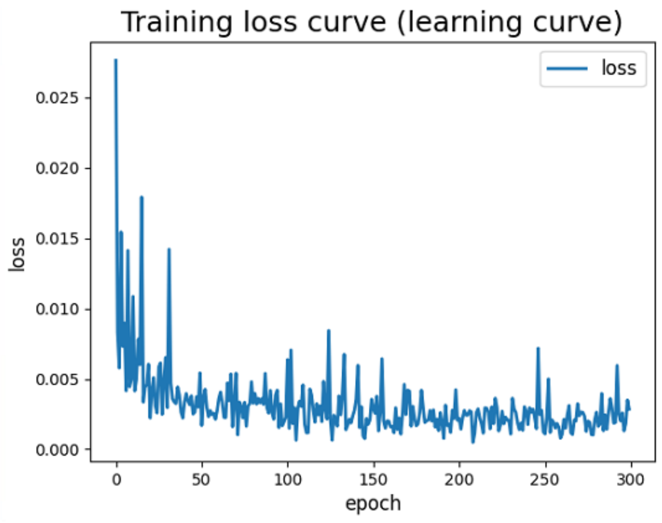


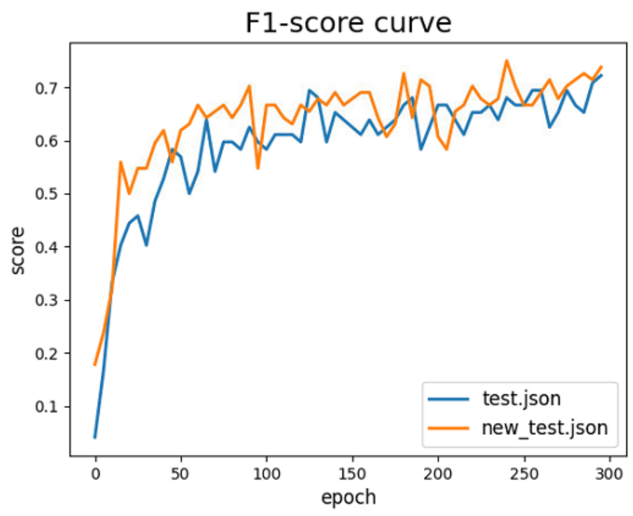
quadratic noise scheduling





sigmoid noise scheduling





It is evident from the observations that training with sigmoid noise scheduling yields the most stable results, while linear noise scheduling leads to the highest degree of oscillation. This discrepancy arises because linear noise scheduling accelerates data degradation too rapidly, resulting in slightly diminished training outcomes.

6. Reference

1. [NIPS 2020] Denoising Diffusion Probabilistic Models

2. [PMLR 2021] Improved Denoising Diffusion Probabilistic Models

3. [NeurIPS 2021] Classier-Free Diffusion Guidance