311551069 余忠旻 Lab7 : Let's Play DDPM

1. Introduction

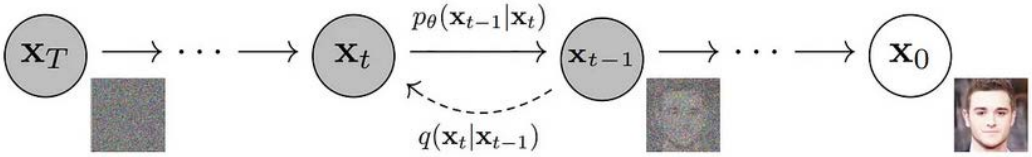
用 conditional DDPM 來生成立體幾何圖形的圖片。圖形包含 3 種形狀和 8 種顏色，總共 24 種幾何圖形。另外，一張圖片可以包含 1 至 3 個圖形。一張圖片可以有多個 labels，因此會把 labels 對應到 one-hot vector 後 embedding 在 UNet architecture 做訓練。Sampling 的時候，會從 test.json 和 new_test.json 中得知每個圖片要生成的幾何圖形的種類和個數，去生成圖片並交給助教所提供的 evaluator 來判斷圖片是否符合要求。

2. Implementation details

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

- **Main structure**

Diffusion model 的核心精神是學習一個逐步 denoise 的過程，可以把 diffusion model 過程的每個影像表示為 Markov chain。而訓練中加入很小的高斯雜訊則是來自 Gaussian noise。而網路 θ 要學的東西就是如何 denoising，如下圖：



因此 DDPM 的優化目標就是，讓網路預測的噪音和真實的噪音一致，也就是在訓練的時候，會隨機選擇一個訓練樣本 -> 從 1-T 中隨機抽樣一個 t -> 隨機產生噪音並計算當前所產生的帶噪音數據(紅色框所示) -> 輸入網路預測噪音 -> 計算產生噪音和預測噪音的 L2 loss -> 計算 gradient 並更新網路。

Sampling 的時候則是，會從一個隨機噪音開始，利用訓練好的網路預測噪音，然後計算條件分布的均值(紅色框部分)，然後用均值和標準差乘以一個隨機噪音，直到 t=0 完成新樣本的生成(最後一步不加噪音)。

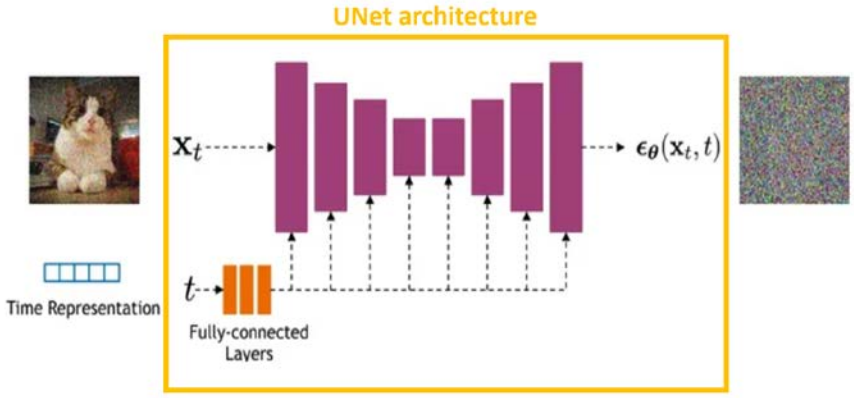
Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
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: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

在訓練的時候，由於噪音圖片和原始圖片是相同維度的，DDPM 會採用 AutoEncoder 的架構，也就是下圖顯示 UNet architecture，當中我們會有 time embedding 來將 timestep 編碼到網路中，並且增加 label embedding，幫助我們生成相對應條件的圖片。



Training process (Algorithm 1)實作部分如下:

```
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)實作部分如下:

```
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**

  我 loss function 用的是 MSELoss，來計算產生噪音和預測噪音的 L2 loss

- **Prediction type**

  根據 Denoising Diffusion Probabilistic Models [1] 的 equation (8)，我們可以用它來做 predict the noisy sample：

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

  但論文發現省略前面 weight 項（$\frac{1}{2\sigma_t^2}$），反而可以幫助網路更加集中在較困難的 sample，因此最終 loss 可寫作 simplified noise predicting：

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

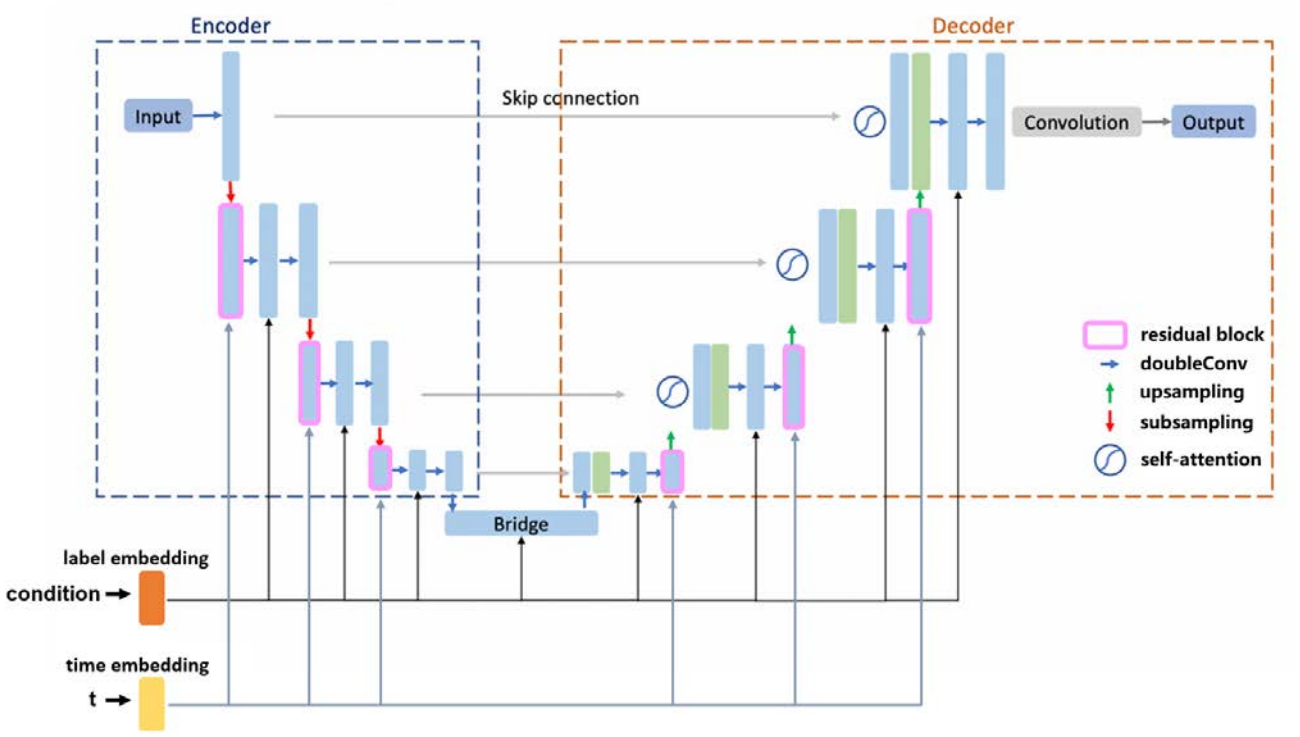
  而我使用的就是 simplified noise predicting。

- **Noise schedule**

  我用的是 sigmoid noise schedule，主要是因為 linear noise schedule 的缺點是資料破壞得太快，而在 Improved Denoising Diffusion Probabilistic Models [2] 這篇論文有提到 cosine noise schedule 能得到較好結果，而經過實驗，我發現我的 diffusion model 架構搭配 sigmoid noise schedule 訓練比較好，因此我使用 sigmoid noise schedule，公式如下：

```
# 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



我的 UNet architecture 如上，encoder 部分是由許多 doubleConv、residualBlock 以及 subsampling 所組成的，主要是透過採樣來降低 feature 的空間大小(H 和 W)，decoder 則是相反，由許多的 doubleConv、residualBlock 和 upsampling 所組成的，主要是將被壓縮的 feature 逐漸恢復。除此之外，還有 self-attention 和 skip connection 來幫助訓練，self-attention 能增加網路的全局建模能力，skip connection 則是 concatenate 了 encoder 中間得到的同維度 feature，有利於網路優化。

而 time embedding 和 label embedding，我分別加在 residualBlock 和 subsampling/upsampling 部分，能提供 UNet 知道 timestamp 和 lable 資訊，這樣在訓練時能更有效率。

■ Dataloader

dataloader 主要分為可以分成兩種模式，一種取 training dataset 的 image 和 condition，一種是取 testing dataset 的 condition。

下圖是取 training dataset 的部分，可以看到我們會先將 object.json 讀進來，當中會有立體幾何圖形所對應的代碼 (0~23)。接著，我們會把 train.json 讀進來，當中會顯示 image 以及圖片所對應的立體幾何圖形，image 部分則會經過前處理，也就是 pad 成正方形，resize 成 32*32 (我實驗環境 64*64 的 image 會 out of memory，所以降低 resolution 來訓練)，最

後轉成 tensor 和 normalize。condition 部分則是將它轉成 one hot vector (也就是圖片有的 label 相對應的 index 為 1，其餘為 0)。

```
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
```

下圖是取 testing dataset 的部分，跟取 training dataset 差不多，也就是 object.json 讀進來，當中會有立體幾何圖形所對應的代碼 (0~23)，然後將 test condition 讀進來，將它轉成 one hot vector (也就是圖片有的 label 相對應的 index 為 1，其餘為 0)

```
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)

我們訓練時會想要更穩定的訓練，不要震盪太大，而 EMA 實質上就是一種 smoother training，他更新的時候不太容易受到 outliers 的影響，他作法是從 main model 複製一份初始的 model weights，更新的時候會根據 moving average from main model 來更新 EMA model，因此更新的公式如下：

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

程式碼實作部分如下：

```python
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)

在這次作業中，我們會想要同時讓模型保有生成能力，並且能準確生成相對應 condition 的圖片，在 Classifier-Free Diffusion Guidance [3] 這篇論文中，他可以避免 posterior collapse (model ignore conditional information just generate any image)，他做法是捨棄原本外部的 classifier，而是提出一個等價的結構，從而讓 diffusion model 可以成功完成條件生成的任務。

而實際做法是，他會有兩種採樣的輸入，一種是 conditional (random Gaussian noise + label embedding)，一種是 unconditional。兩種輸入都會送到同一個 diffusion model，從而讓其能夠具有 unconditional 和 conditional 的生成能力。

原本的 noise 更新的方式：

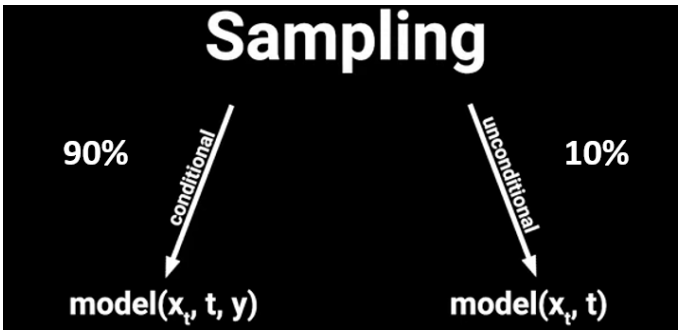
$$\epsilon_\theta(x_t, t) \sim \epsilon_\theta(x_t) - \sqrt{1-\overline{\alpha}_t} \bigtriangledown_{x_t} log p_\phi(y|x_t)$$

而 classifier-free 用另一個近似的等價結構替換了後面那一項：

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

表示 conditional 的輸入，表示 unconditional 的輸入(會將 condition y 設為 NULL)，用這兩項之差乘以一個係數來替換掉原來的那一項。

實作時，這兩種 sampling 比例如下，conditional 和 unconditional 分別是 9:1，並且 predicted noise 會 linear interpolate 從 unconditional predicted noise 逐漸到 conditional predicted noise



training 和 sampling 程式碼實作如下：

```
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
```

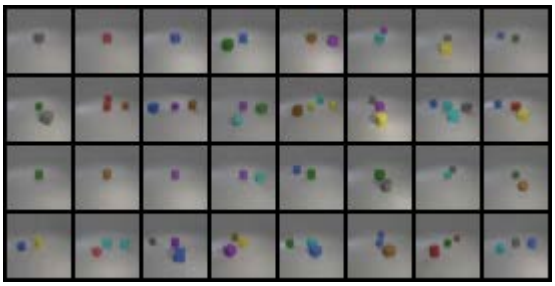
B. Specify the hyperparameters (learning rate, epochs, etc.)

- 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

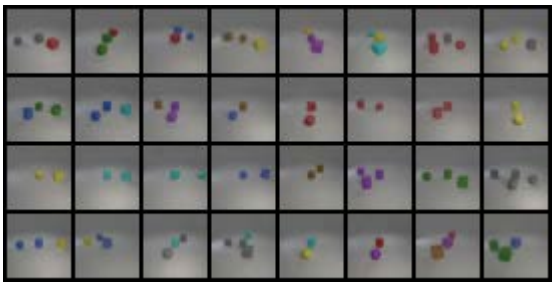
4. Results and discussion

A. Show your results based on the testing data.

test.json (0.72222)



new_test.json(0.75)



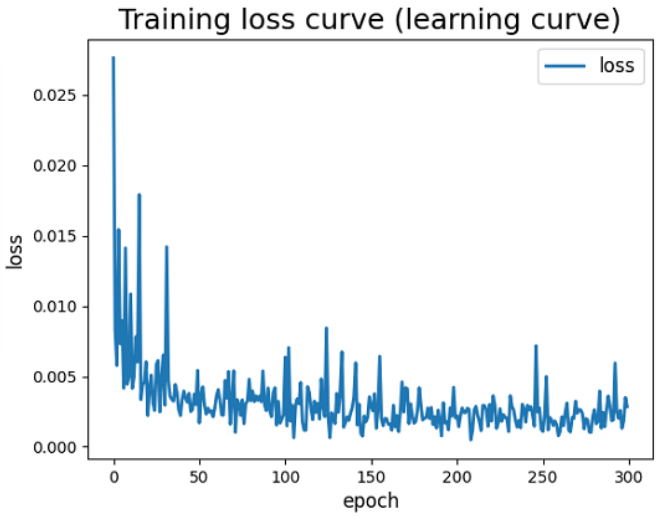
B. Discuss the results of different model architectures.

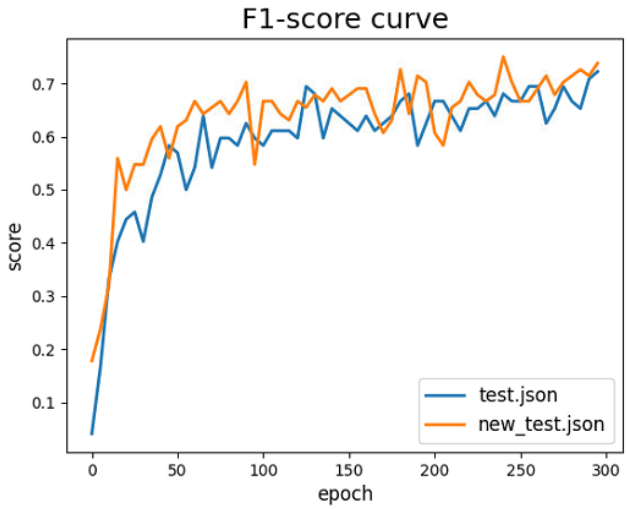
- 我在 lable embedding 部分，實作了兩種做法：

  一種是在上述的 Implementation details 所提到，將 lable embedding 做在 subsampling/upsampling。

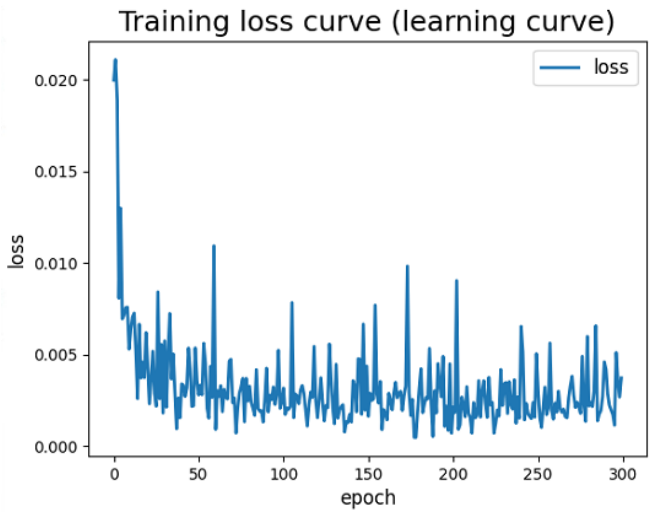
  另一種做法是將 label condition expand 成圖片的 w*h，並將它當作 additional input channels 當作輸入(也就是原本輸入為(bs, 3, 28, 28)，加上 label embedding 變成(bs, 16, 28, 28))。

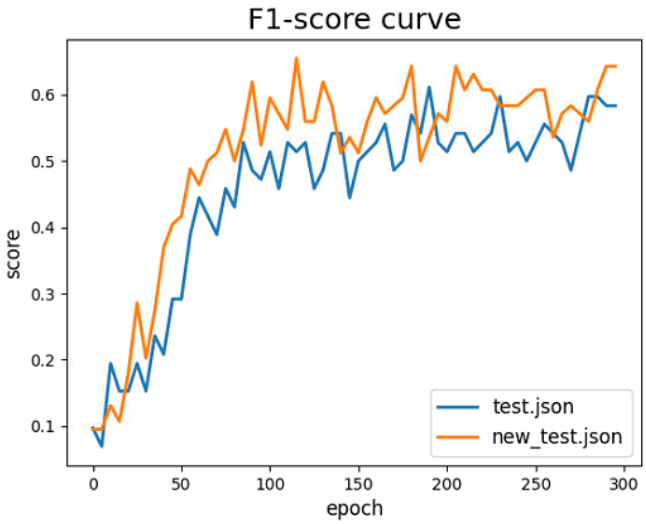
lable embedding on subsampling/upsampling block





label add as additional input channels





可以觀察到 lable embedding on subsampling/upsampling block 效果比較好，並且訓練也比較穩定，因為在許多層都可以看見 label embedding 的資訊，所以訓練比較好 (跟 time embedding 做法類似)。而另一種 label

add as additional input channels 這種方法，可能因為 label condition 資訊是跟圖片的 RGB channel 一起當輸入的，經過 UNet 許多層之後，condition 的資訊會有所損失，導致訓練比較不好。

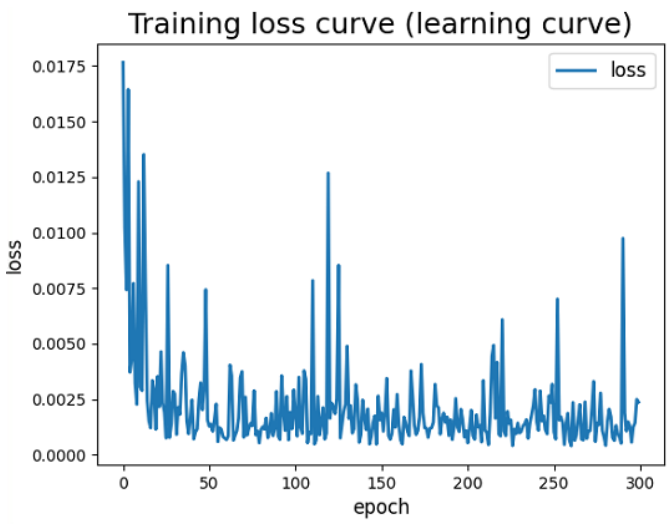
- Noise scheduling

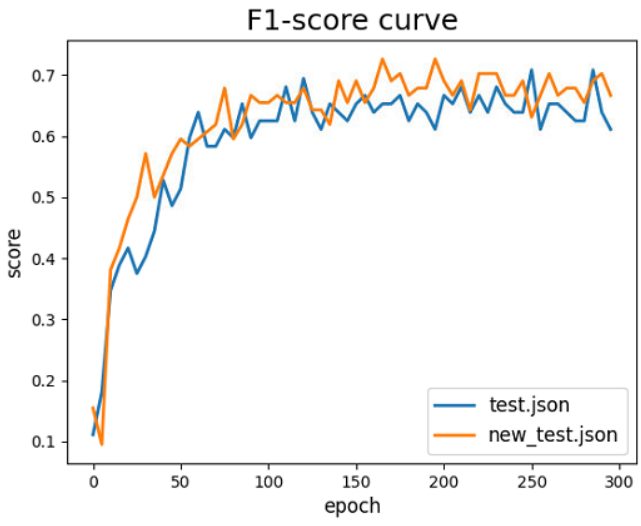
Noise scheduling 會關係 conditional DDPM 資料破壞的速度，這會影響訓練出來的結果，因此我有將不同的 noise scheduling 去做比較：

我總共有比較三種 noise scheduling，分別是 linear、quadratic、sigmoid noise schedule，而 cosine noise scheduling 我也有跑實驗，只是效果很差，可能有寫錯或不適合我的 conditional DDPM 架構，這裡我就不討論 cosine noise schedule。

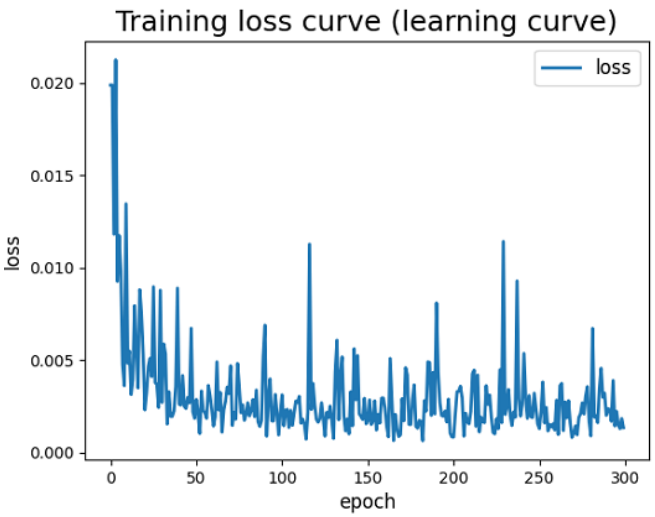
```python
# linear beta schedule
def prepare_noise_schedule(self):
    return torch.linspace(self.beta_start, self.beta_end, self.noise_steps)

# 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
```

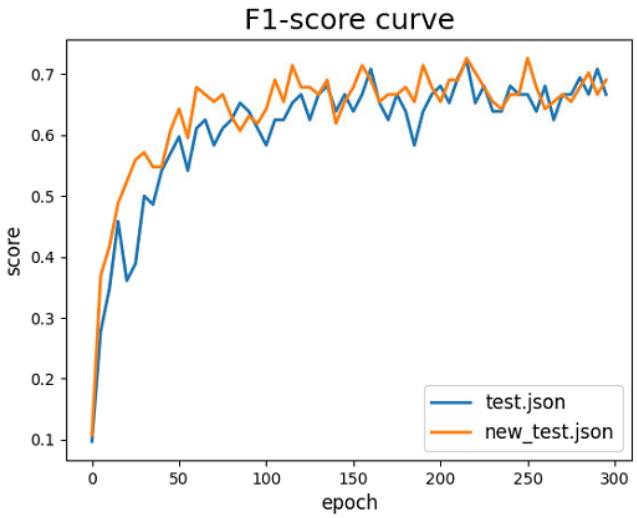
linear noise scheduling



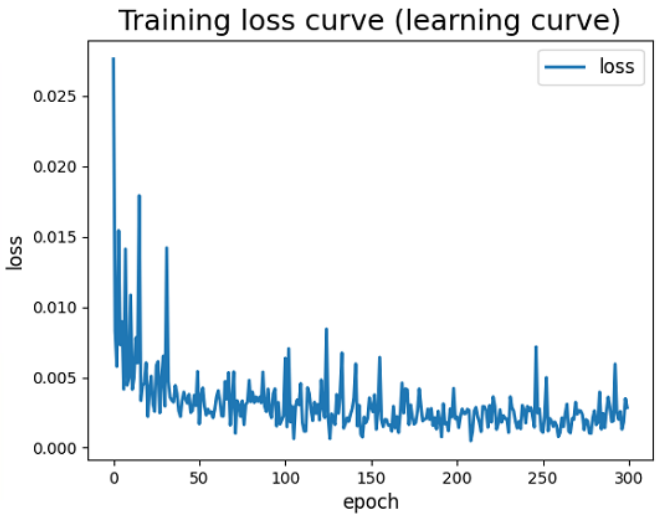


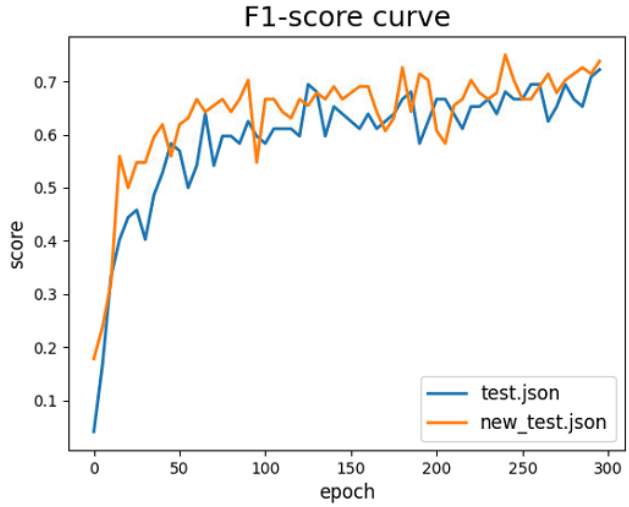
quadratic noise scheduling





sigmoid noise scheduling





可以觀察到 sigmoid noise scheduling 訓練是最穩定的，而 linear noise scheduling 最震盪的，原因是因為 linear noise scheduling 會造成資料破壞得太快，導致訓練結果會稍微差一點。

5. Experimental results

```
pp037@ec037:~/DLP/lab7$ python3 sampling.py
Using device: cuda
/home/pp037/.local/lib/python3.8/site-packages/torchvision/models/_utils.py:208: U
serWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed
 in the future, please use 'weights' instead.
  warnings.warn(
/home/pp037/.local/lib/python3.8/site-packages/torchvision/models/_utils.py:223: U
serWarning: Arguments other than a weight enum or `None` for 'weights' are depreca
ted since 0.13 and may be removed in the future. The current behavior is equivalen
t to passing `weights=None`.
  warnings.warn(msg)

test.json
score1: 0.7083333333333334
score1: 0.7222222222222222
score1: 0.7083333333333334
score1: 0.6944444444444444
score1: 0.6805555555555556
score1: 0.6944444444444444
score1: 0.7083333333333334
score1: 0.7083333333333334
score1: 0.6944444444444444
score1: 0.6944444444444444
max score: 0.7222222222222222
avg score: 0.7013888888888888

new_test.json
score2: 0.7380952380952381
score2: 0.7261904761904762
score2: 0.7142857142857143
score2: 0.7380952380952381
score2: 0.75
score2: 0.7261904761904762
score2: 0.7380952380952381
score2: 0.7380952380952381
score2: 0.7142857142857143
score2: 0.7261904761904762
max score: 0.75
avg score: 0.730952380952381
```

6. Reference

1. [NIPS 2020] Denoising Diffusion Probabilistic Models
2. [PMLR 2021] Improved Denoising Diffusion Probabilistic Models
3. [NeurIPS 2021] Classifier-Free Diffusion Guidance