

Deep Learning Lab7 - Let's Play DDPM Report

智能所 311581006 林子凌

1. Introduction

本次 lab 主題為實作 conditional Denoising Diffusion Probabilistic Models (DDPM)，根據給定的 multi-label 作為 condition，生成對應圖片。DDPM 是最近在 computer vision 領域中，被廣泛應用於圖像風格轉換(style transfer)和圖像生成(image generation)等多種任務的模型，基於擴散過程的概念，在原始圖像中引入 noise 並隨時間進行逐步去噪，進行圖像生成。本次 lab 要求建構一個 DDPM 模型，在給定 multi-label 作為生成條件的情況下，模型要能夠生成帶有對應物體(例如紅色立方體、藍色圓柱體等)的合成圖片。最後，將生成圖片送入 ResNet18 架構的 pretrained object classifier 中，評估圖像中是否包含給定條件的 label。

2. Implementation details

Describe how you implement your model

● Choice of DDPM

本次選擇 pixel-domain DDPM model，因為相較於 latent diffusion 會需要額外的 encoder 和 decoder，做 image pixel 和 latent representation 之間的轉換，pixel-domain DDPM 需要的訓練時間與資源較少。在本次 lab 中，因為最後的目的是根據 label 生成對應合成圖片，因此必須實作 conditional DDPM，把 label 和 time 當作條件加入 DDPM 模型。一種常見將 condition 加入 U-Net 架構的方法，是在 down sampling 和 up sampling 時，將 input 和 condition 做乘加運算或直接 concatenate 作為輸入。在 down sampling 中加入 condition 可以用來控制保留的資訊，而在 up sampling 加入 condition 可以讓生成圖片帶有想要的條件，進而根據 label 生成對應的合成圖像。

● U-Net Architecture

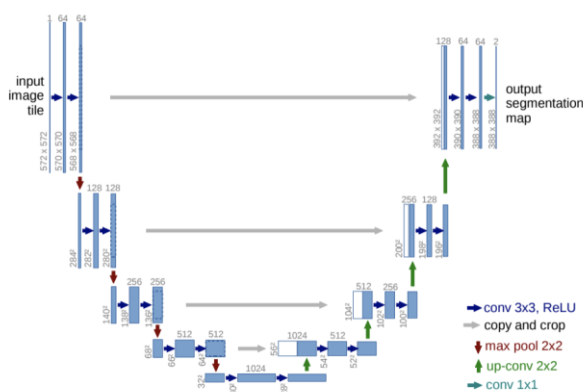


Figure 1 U-Net architecture overview

Figure 1 是 U-Net 架構的示意圖，是 autoencoder 的一種變形體，模型呈現 U 形狀。左半邊由 Down sampling 擔任 encoder 的角色，不斷向下萃取特徵並順便去除雜訊，將 input image 尺寸變小，變成中間 dimension 較小的 hidden representation。右半邊由 Up sampling 擔任 decoder 的角色，根據 hidden representation 一步步放大還原，重建出跟原圖一樣大小的新圖像。另外，為了不要讓 U-Net 在經過 Down sampling 後，遺失重要資訊，因此會在中間增加一些 connection，讓每一段 Down sampling encoder 都可以有路徑連接對面的 Up sampling decoder，。

```

"""Unet model"""
class Unet(nn.Module):
    def __init__(self, in_channels, n_feature=256, n_classes=24): ...

    def initialize_weights(self): ...

    def forward(self, x, cond, time):
        # embed context, time step
        cond_emb1 = self.cond_embed1(cond).view(-1, self.n_feature * 2, 1, 1) # [32,512,1,1]
        time_emb1 = self.time_embed1(time).view(-1, self.n_feature * 2, 1, 1) # [32,512,1,1]
        cond_emb2 = self.cond_embed2(cond).view(-1, self.n_feature, 1, 1) # [32,256,1,1]
        time_emb2 = self.time_embed2(time).view(-1, self.n_feature, 1, 1) # [32,256,1,1]
        # for down
        cond_emb_down1 = self.cond_embed_down1(cond).view(-1, self.n_feature, 1, 1)
        time_emb_down1 = self.time_embed_down1(time).view(-1, self.n_feature, 1, 1)
        cond_emb_down2 = self.cond_embed_down2(cond).view(-1, self.n_feature, 1, 1)
        time_emb_down2 = self.time_embed_down2(time).view(-1, self.n_feature, 1, 1)

        # initial conv
        x = self.initial_conv(x) # [32,256,64,64]

        # down sampling
        down1 = self.down1(cond_emb_down1*x+ time_emb_down1)
        down2 = self.down2(cond_emb_down2*down1+ time_emb_down2)

        # hidden
        hidden = self.hidden(down2)

        # choose to concatenate the embedding at hidden or not
        # hidden = torch.cat((hidden, temb1, cemb1), 1)

        # up sampling
        up1 = self.up0(hidden) # [32,256,64,64]
        up2 = self.up1(cond_emb1*up1+ time_emb1, down2)
        up3 = self.up2(cond_emb2*up2+ time_emb2, down1)

        # output
        out = self.out(torch.cat((up3, x), 1))
        return out

```

Figure 2 U-Net architecture

Figure 2 是本次 lab 中，我設計的 U-Net 架構程式碼實作。可以看到按照順序，由 **initial convolution**, **down sampling**, **hidden representation**, **up sampling**, **output** 這五個區塊搭建而成，以下會分別介紹：

(1) **initial convolution**：整個 U-Net 架構的起始部分，用來初步提取輸入圖片的大致特徵。由一個 ResidualConvBlock 組成，包含兩個 convolution blocks，每個 blocks 裡面由一個 2D CNN, 2D Batch normalization, GELU activation function 搭建而成，具體如下 Figure 3 所示。

```

"""ResNet style convolution block"""
class ResidualConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, is_residual=False):
        super().__init__()

        self.same_channels = (in_channels==out_channels)
        self.is_residual = is_residual

        # conv block
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

```

Figure 3 ResidualConvBlock in U-Net

(2) down sampling：用兩層 down sample block 負責降低輸入圖片的解析度(resolution)，並提取更 high-level 的特徵。具體通過 convolution (用來提取特徵，得到 feature map)和 pooling (用來減少圖片，或者說 feature map 的尺寸)組成，詳細如 Figure 4 中所示，包含一個 ResidualConvBlock (Figure 3) 和一個 Max Pooling layer。

```
"""down sampling image feature maps"""
class DownSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSample, self).__init__()

        self.model = nn.Sequential(
            ResidualConvBlock(in_channels, out_channels),
            nn.MaxPool2d(2)
        )
```

Figure 3 DownSample block in U-Net

(3) hidden representation：U-Net 模型的中間層，位於 U-Net 底部，會再對圖片進行一部更深層的特徵提取，詳細由 Figure 4 所示，包含一個 Average Pooling layer 和 GELU activation function。

```
"""bottom hidden of unet"""
self.hidden = nn.Sequential(nn.AvgPool2d(8), nn.GELU())
```

Figure 4 hidden representation block in U-Net

(4) up sampling：用兩層 up sample block 負責回復輸入圖片的解析度(resolution)，並且根據 high-level 的特徵(hidden representation)，重建圖片。具體通過反向的 convolution block (用來放大 feature map，回復尺寸)和 ResidualConvBlock (如 Figure 3)組成，並加入 connection 到 up sampling 的輸入中，詳細如 Figure 5 中所示，包含一個 2D ConvTranspose block 和兩個 ResidualConvBlock (Figure 3)。

```
"""up sampling image feature maps"""
class UpSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UpSample, self).__init__()

        self.model = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResidualConvBlock(out_channels, out_channels),
            ResidualConvBlock(out_channels, out_channels)
        )

    def forward(self, x, skip):
        out = torch.cat((x, skip), 1)
        out = self.model(out)
        return out
```

Figure 5 UpSample block in U-Net

(5) output：U-Net 模型的最後一層，用來生成最後的輸出圖片。如 Figure 6 所示，由 2D convolution, Group Normalization, ReLU activation function 三個元素組合而成。將經過 Up sampling 得到的 feature map 映射到最後的 output space 中，將 channel 數量回復到 input channels 數。

```
"""output"""
self.out = nn.Sequential(
    nn.Conv2d(2 * n_feature, n_feature, 3, 1, 1),
    nn.GroupNorm(8, n_feature),
    nn.ReLU(),
    nn.Conv2d(n_feature, self.in_channels, 3, 1, 1),
)
```

Figure 6 Output block in U-Net

Condition embedding

Figure 2 中可以看到 condition embedding 宣告並和 feature map 結合，送入 Down sampling 和 Up sampling block 的過程。Time 和 label (cond) 分別建立自己的 condition embedding，因為 feature map 的 channels 可能在 Down sampling 和 Up sampling 過程中改變，所以送入不同 block 的 embedding 都需要重新宣告，不能共用。

```
"""embed time and label condition"""
class Embed(nn.Module):
    def __init__(self, input_dim, emb_dim):
        super(Embed, self).__init__()

        self.input_dim = input_dim
        self.model = nn.Sequential(
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim)
        )
```

Figure 7 Embedding block in U-Net

Figure 7 為 embedding 的建構方法。在 Figure 2 中，作為條件的 time 和 label (cond) 都通過兩層 linear layer，中間加上 GELU activation function 來獲得條件 embedding。在 Down sampling 中加入 condition 的輔助可以幫助 encode 過程留下更重要與條件相關的資訊；在 Up sampling 中加入 condition 則可以在重建圖片時加入條件資訊，形成根據條件合成出的對應圖像。因此在本次 lab 中以乘加方法將 input feature map 和 condition 結合，組合成新的輸入，可以簡單表達如下：

$$x' = c \odot x + t \quad (1)$$

公式(1)中， x 為某個 down sampling block 或 up sampling block 上一個 block 輸出的 feature map，和 label condition embedding c 做 element-wise 相乘後，加上 time condition embedding t ，可以得到 x' 作為輸入 down sampling block 和 up sampling block 的 feature map。

● Noise schedule

在本次 lab 中選擇使用 linear schedule 來實現 DDPM 中的 noise scheduling。在 DDPM 的 forward process，markov chain 中加入雜訊一步步模糊圖片(或稱為 encoding)的步驟可以表達如下：

$$q(x_t|x_{t-1}) := N(x_t; \sqrt{1-\beta_t} x_{t-1}, \beta_t I) \quad (2)$$

在公式(2)中， β 用來決定在圖片上加雜訊的過程快慢。整個 $\beta_i, i = 1, \dots, T$ 是一個 schedule，這次選用的 linear schedule 即如 Figure 8 所示，讓 β vector 在預定好的 β 最小最大值中間呈現線性分布。

$$\beta_t = \min \beta + (t - 1) \times \left(\frac{\max \beta - \min \beta}{T} \right), \forall t \in 1, \dots, T \quad (3)$$

公式(3)中， $\max \beta$ 和 $\min \beta$ 為預先定義好的值，本次設定為 0.02 和 $1e-4$ 。 t 是當前的 diffusion step， T 是總共的 diffusion step。在不同時間點 t 可以按照公式計算出 β_t ，再根據 DDPM 中的公式換算得出對應的 $\sqrt{\beta_t}, \alpha_t, \bar{\alpha}_t, \sqrt{\bar{\alpha}_t}, \dots$ 等，在 DDPM forward 和 reverse process 中，會需要用到的數值。通過這樣的 pre-computed schedule，可以在之後直接從 buffer 取用且避免重複計算。

除了 linear schedule 外，還有 cosine noise schedule 方法，用來改善加入 noise 的速度過快，導致圖片資訊遺失太快的問題。可以用公式表達如下：

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, f(t) = \cos\left(\frac{t/T + s}{1 + s} \cdot \frac{\pi}{2}\right)^2 \quad (4)$$

公式(4)中，t 是當前的 diffusion step，T 是總共的 diffusion step。S 則是一個很小的值，在 DDPM 論文中，設定 $s = \sqrt{\beta_0} = 0.008$ ，目的是用來防止 β_t 過小。將 $\bar{\alpha}_t$ 通過 DDPM 論文中的公式轉換，也可以得到其他需要的數值 ($\beta_t, \sqrt{\beta_t}, \alpha_t, \sqrt{\alpha_t}, \dots$ 等)，但這次 lab 訓練時間較緊迫，因此沒有實作到這部分。Figure 8 為 linear schedule 的實作程式碼，預先計算好所有的數值。Figure 9 則是 forward process 的過程，可以直接取用數值用以在圖片中加入 noise：

```
"""return pre-computed schedules for DDPM sampling in training process"""
def ddpm_schedules(beta1, beta2, T, schedule_type="linear"):

    assert (beta1 < beta2 < 1.0, "beta1 and beta2 must be in (0, 1)")

    if schedule_type == "linear":
        beta_t = (beta2 - beta1) * torch.arange(0, T + 1, dtype=torch.float32) / T + beta1
        sqrt_beta_t = torch.sqrt(beta_t)
        alpha_t = 1 - beta_t
        log_alpha_t = torch.log(alpha_t)
        alphabar_t = torch.cumsum(log_alpha_t, dim=0).exp()

        sqrtab = torch.sqrt(alphabar_t)
        oneover_sqrt_a = 1 / torch.sqrt(alpha_t)

        sqrtmab = torch.sqrt(1 - alphabar_t)
        mab_over_sqrtmab_inv = (1 - alpha_t) / sqrtmab

    return {
        "alpha_t": alpha_t, # alpha_t
        "oneover_sqrt_a": oneover_sqrt_a, # 1/sqrt{alpha_t}
        "sqrt_beta_t": sqrt_beta_t, # sqrt{beta_t}
        "alphabar_t": alphabar_t, # bar{alpha_t}
        "sqrtab": sqrtab, # sqrt{bar{alpha_t}}
        "sqrtmab": sqrtmab, # sqrt{1-bar{alpha_t}}
        "mab_over_sqrtmab_inv": mab_over_sqrtmab_inv, # (1-alpha_t)/sqrt{1-bar{alpha_t}}
    }
```

Figure 8 DDPM linear noise schedule

```
def forward(self, x, cond):
    """training ddpm, sample time and noise randomly (return loss)"""
    # t ~ Uniform(0, n_T)
    timestep = torch.randint(1, self.n_T+1, (x.shape[0],)).to(self.device)
    # eps ~ N(0, 1)
    noise = torch.randn_like(x)

    x_t = (
        self.sqrtab[timestep, None, None, None] * x
        + self.sqrtmab[timestep, None, None, None] * noise
    )

    predict_noise = self.unet_model(x_t, cond, timestep/self.n_T)

    # return MSE Loss between real added noise and predicted noise
    loss = self.mse_loss(noise, predict_noise)
    return loss
```

Figure 9 DDPM forward process

● Loss Function

Loss function 選擇 mean square error (MSE) loss 來計算 predicted noise 和 real encoding noise (gaussian noise)之間的差異。實作程式碼如 Figure 9 所示，real encoding noise 由 normal distribution 抽樣得到，predicted noise 則是選擇 prediction noise type 用 Simplified noise predicting。先從 uniform 分布中抽樣

diffusion time step，計算出對應的 encoded image，將 encoded image, time condition, label condition 輸入 U-Net 模型中，預測出來的 reverse predicted noise。因為訓練目的是讓學習出來的 reverse predicted noise 越接近真實 encode noise (gaussian noise)越好，因此採用 MSE 作為 loss function 計算公式。

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

本次實驗的 learning rate 設定為 $1e-3$ ，訓練中使用 linear decay (min learning rate=0，線性在每個訓練 epoch 降低 learning rate)和 ReduceLROnPlateau (當 test accuracy 和 new test accuracy 都沒有上升超過 5 個 epoch，則將原本的 learning rate 乘上 0.95，min learning rate=0)，利用這兩種 learning rate schedule 方法來遞減 learning rate。另外，Batch size 為 32，epochs 為 500，diffusion noise step 設定為 1000，input 和 output image size 皆為 64，optimizer 為 Adam，embedding size 為 256。

3. Results and discussion

Show your results based on the testing data

Figure 10 和 Figure 11 分別為 test.json 和 new_test.json 的生成結果。經過 pretrained classifier 計算出的 accuracy 分別為 **0.6528** 和 **0.7262**。

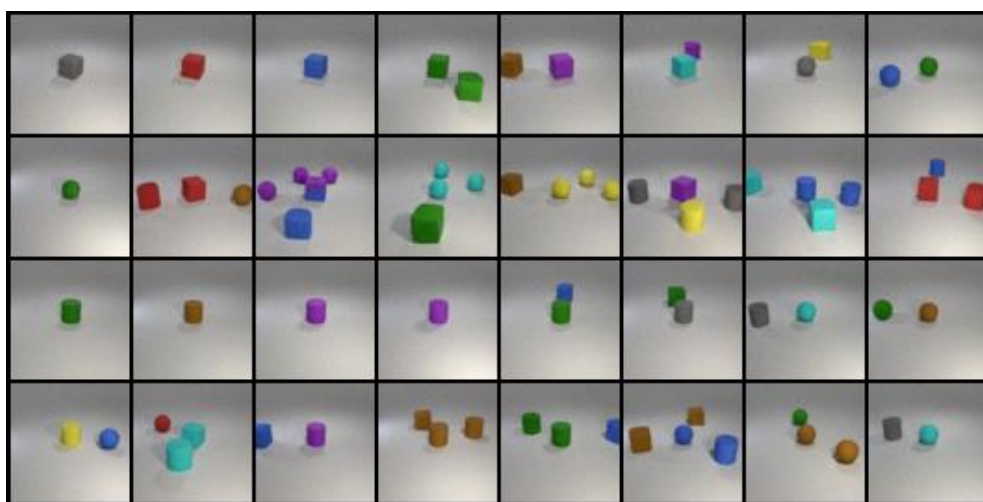


Figure 10 test.json 生成結果，accuracy=0.6528



Figure 11 new_test.json 生成結果，accuracy=0.7262

Discuss the results of different model architectures

● Condition Embedding 的加入位置

一開始我在建構 conditional U-Net 模型時，沒有選擇在 Down Sampling 時加入 condition，得到在 test 和 new test 上的 accuracy 結果分別為 0.6308 和 0.6777，低於在 Down Sampling 加入 condition 的情況約 0.02 和 0.05 (如 Table 1 所示)。可以推測在 Down Sampling 階段加入 condition embedding 作為輔助，可以讓模型學習在 encode 到 hidden representation 時，保留與 condition 相關的重要資訊，進而讓之後根據 condition 重建圖片的效果更好，能夠得到更高的準確度。

Add condition		Result	
Up sampling	Down Sampling	test.json accuracy	new_test.json accuracy
✓	✗	0.6308	0.6777
✓	✓	0.6528	0.7262

Table 1 DDPM 在 Down Sampling 是否加入 condition embedding，對實驗結果的影響

Table 2 顯示在 Up sampling 和 Down sampling 都加入 condition embedding，在不同訓練 epoch 時，模型根據 test.json 和 new_test.json 給定條件的生成結果。可以看到 DDPM 模型確實是從初期生成一堆雜訊，過渡中期產生模糊圖像，再逐漸訓練到可以根據 label 條件生成對應的物體顏色和形狀，且圖片也變得比較清晰。Table 3 是只在 Up sampling 加入 condition embedding，模型的生成結果。相比之下，沒有在 Down sampling 加入 condition 的模型訓練比較快，可以在更早的 epoch 產生出較明確的形狀與顏色，但隨著訓練 epoch 上升，最終生成結果的準確度略低，如 Table 1 中的數據比較所示。


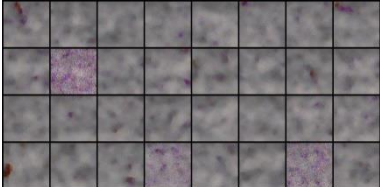

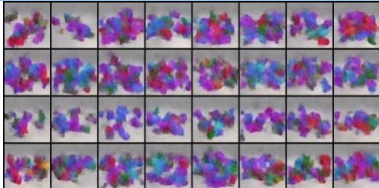
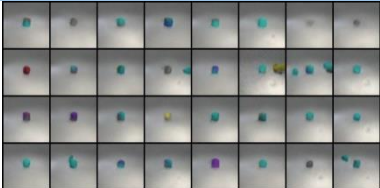

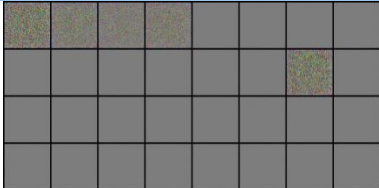
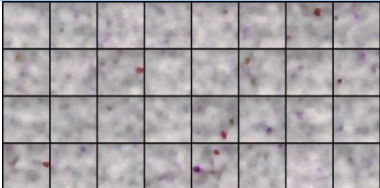
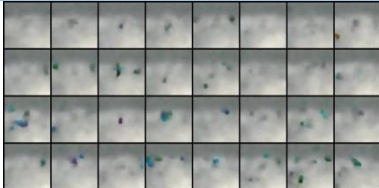
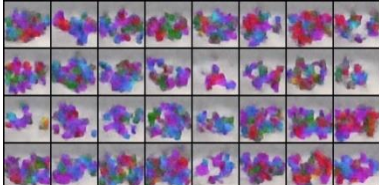


Test.json image generation result																							
Epoch 0								Epoch 5								Epoch 10							
																							
Epoch 15								Epoch 20								Epoch 25							
																							
New_test.json image generation result																							
Epoch 0								Epoch 5								Epoch 10							
																							
Epoch 15								Epoch 20								Epoch 25							
																							

Table 2 DDPM 在不同訓練 epoch 下的生成結果(Down Sampling 時有加入 condition embedding)

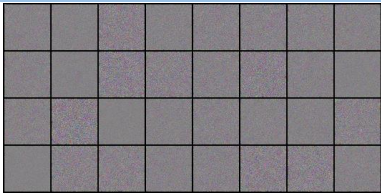
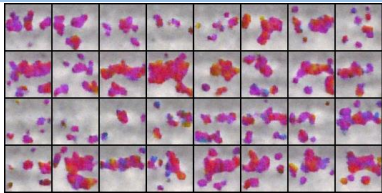
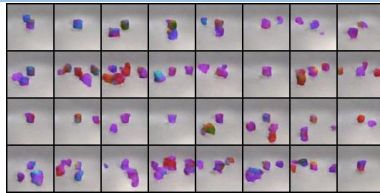
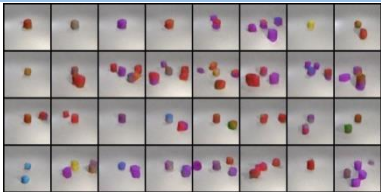


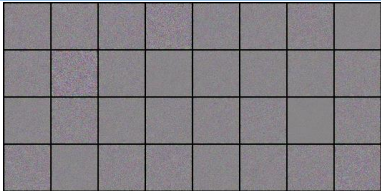
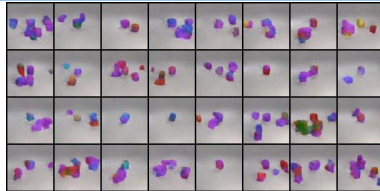



Test.json image generation result																							
Epoch 0								Epoch 5								Epoch 10							
																							
Epoch 15								Epoch 20								Epoch 25							
																							
New_test.json image generation result																							
Epoch 0								Epoch 5								Epoch 10							
																							
Epoch 15								Epoch 20								Epoch 25							
																							

Table 3 DDPM 在不同訓練 epoch 下的生成結果(Down Sampling 時沒有加入 condition embedding)

● Condition embedding 方法

除了本次使用的乘加方法(如公式(1)運算)可以將 condition embedding 融入到模型之中，另一種常見的方法是將 input feature map, time condition embedding, label condition embedding 三個元素，利用 concatenate 的方法接到一起，做為下一個 block 的輸入。具體實作如 Figure 2 U-Net architecture 裡面，hidden 改用註解掉的那行。這個方法我認為比較不適用於在 Down sampling 和 Up sampling 都加上 condition embedding，因為 input dimension 會很高，可能只能用在 hidden representation 部分。比較可惜的是這次實驗比較緊迫，運算時間跟資源都不夠跑完實驗，因此沒辦法做結果比較。但我推測只在 hidden representation 加入條件，可能因為沒有在 encode 和 decode 加入條件限制保留資訊與生成方向，所以結果應該會比較差。另外，我嘗試過讓 input channel 數量一樣的 Down sampling 和 Up sampling block 共用 condition embedding，但發現兩者目的不同，一個是想要在縮小 feature map 尺寸時保留與 condition 相關的重要特徵，另一個則關注對生成有輔助效果的 condition 信息。因此共用 embedding 的效果極差，會輸出只包含雜訊的圖片。

● 使用 clamp 對生成圖片做 normalization

在實驗中，我嘗試對輸出 image tensor 的值做 clamp，取值域[-1,1]之間以做 normalization，如下：

```
x = (x.clamp(-1, 1) + 1) / 2
x = (x * 255).type(torch.float)
```

Figure 12 clamp output image tensor

根據 Figure 12 對輸出 image tensor 限制值域，會讓生成結果如以下 Table 4 所示：

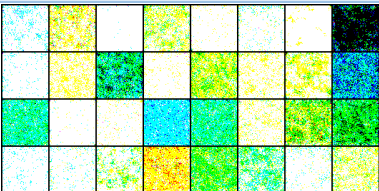
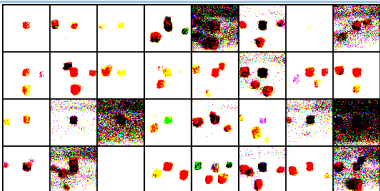
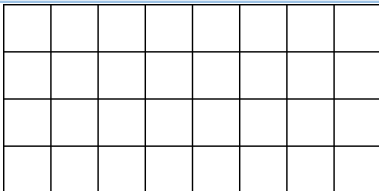
Test.json image generation result																							
Epoch 0								Epoch 15								Epoch 30							
																							

Table 4 用 clamp(-1,1)對每一張生成圖片做 normalization 的生成結果

可以看到輸出圖片會由高對比度的模糊物體變成全白，推測這是因為將值域限制在[-1,1]之間，模型輸出的圖片 tensor value 可能雖然每個 pixel value 具有相對高低之差，但不一定能通過學習 mapping 到範圍內。所以強制做[-1,1]的 clamping 可能會出現這種輸出圖片全白的情況。解決方式後來刪除了對輸出圖片做值域限制，而是在 make_grid 的時候選擇做 normalize (如 Figure 13)。這個選項會根據 image tensor 中的最小值和最大值 shift 到[0,1]的範圍內，使輸出圖片對比度和內容正常如結果所述。

```
grid = make_grid(x_gen, nrow=8, normalize=True)
```

Figure 13 make grid with normalize

● Initialize weight in U-Net model

Figure 14 是對 U-Net 模型中 convolution, batch normalization, linear layer 做參數初始化的實作程式碼。經過初始化可以讓模型初始參數控制在特定分布之中，有助提升訓練效率與最終輸出準確度。

```
def initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight.data)
            if m.bias is not None:
                m.bias.data.zero_()
        elif isinstance(m, nn.BatchNorm2d):
            m.weight.data.fill_(1)
            m.bias.data.zero_()
        elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight.data, 0, 0.01)
            if m.bias is not None:
                m.bias.data.zero_()
```

Figure 14 Initialize U-Net model