Deep Learning

Lab7: Let's Play DDPM

311512064 鄧書桓

1. Intruduction

本次實驗是要實作出 conditional Denoising Diffusion Probabilistic Models(DDPM)，以此來根據多個標籤(條件)來生成出合成圖形。這邊所使用的 dataset 為”clevr dataset”，擁有多張不同物品(包括大小及顏色)的照片。最終的結果為根據生成的照片，放入預先訓練的分類器進行評估。

2. Implementation

ddpm_schedules:

```python
# 用來獲得預先計算的beta和alpha值
def ddpm_schedules(beta1, beta2, T):
    """
    Returns pre-computed schedules for DDPM sampling, training process.
    """
    assert beta1 < beta2 < 1.0, "beta1 and beta2 must be in (0, 1)"

    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_sqrta = 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_sqrta": oneover_sqrta,  # 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": mab_over_sqrtmab_inv,  # (1-\alpha_t)/\sqrt{1-\bar{\alpha_t}}
    }
```

上圖為 ddpm_schedules 的程式碼，功能為利用一些參數來預先算出 betal1、belta2.....等數值，以便 DDPM 使用。

DDPM:

```python
class DDPM(nn.Module):
    def __init__(self, nn_model, betas, n_T, device, drop_prob=0.1):
        super(DDPM, self).__init__()
        self.nn_model = nn_model.to(device)

        # register_buffer 是用來獲得ddpm_schedules中的各種參數
        # self.register_buffer是一种特殊的方法，用于在PyTorch模型的类中注册持久化的缓冲区。
        # 缓冲区是一种在模型中保存持久状态的方式，它不会被视为模型的可训练参数，也不会在反向传播过程中更新
        for k, v in ddpm_schedules(betas[0], betas[1], n_T).items():
            self.register_buffer(k, v)

        self.n_T = n_T
        self.device = device
        self.drop_prob = drop_prob
        self.loss_mse = nn.MSELoss()

    def forward(self, x, c):
        """
        this method is used in training, so samples t and noise randomly
        """
        # 採樣時間(由1~n_T來隨機生成)
        _ts = torch.randint(1, self.n_T+1, (x.shape[0],)).to(self.device)
        # noise為标准正态分布（均值为0，标准差为1）
        noise = torch.randn_like(x)  # eps ~ N(0, 1)

        # x_t 是sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
        # 代表原始照片經過我們network生成的noise
        x_t = (
            self.sqrtab[_ts, None, None, None] * x
            + self.sqrtmab[_ts, None, None, None] * noise
        )
        # loss 是我們預測出來的noise和新增進去的noise(guassin noise)，此使用MSE
        loss = self.loss_mse(noise, self.nn_model(x_t, c, _ts / self.n_T, c))
        return loss
```

上圖為我 DDPM 的實作，這邊的想法是使用給定的採樣時間，再加上標準正態分布的 noise，來藉由自行設計的 model 預測出 noise(這邊的 model 為 ContextUnet，會在下方做詳細說明)，接著藉由預測出來的 noise 和原始的 noise 去做 MSE Loss 的計算，詳細的說明如上圖內容所示。

UNet:

這邊的 Unet 分成 UnetDown 與 UnetUp，且皆有使用 Resnet，這邊會分開做介紹。

Resnet:

```python
class ResidualConvBlock(nn.Module):
    # 如果沒有指定函數的返回值，則默認返回 None

    def __init__(
        self, in_channels: int, out_channels: int, is_res: bool = False
    ) -> None:
        super().__init__()
        '''
        standard ResNet style convolutional block
        '''
        self.same_channels = (in_channels == out_channels)
        self.is_res = is_res
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            # nn.GELU(),
            nn.Tanh(),

        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            # nn.GELU(),
            nn.Tanh(),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # 如果output大小改的話conv會出錯（dim不同）
        if self.is_res:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)
            # 如果過程中使通道數增加，這樣設計才會使殘差是對的
            if self.same_channels:
                out = x + x2
            else:
                out = x1 + x2
            # 輸出前做類似正規化
            return out / 1.414
        else:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)
            return x2
```

上圖為 Resnet 實作的部分，由簡單的兩層 2D convolution 組成，

且各自都有連接 BatchNormalization 與使用 Tanh 來做 active function。其中在 forward 的部分，由於增加文字條件或是 timestep 資料，可能會使通道數發生變化(會導致 conv2d 出現 erro)r，這邊還有對通道輸發生變化時做出應對。

UnetDown:

```
# downsampling的功能
class UnetDown(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UnetDown, self).__init__()
        '''
        process and downscale the image feature maps
        '''
        down_layers = nn.Sequential(ResidualConvBlock(in_channels, out_channels),
                                    nn.MaxPool2d(2),)

        self.model = down_layers

    def forward(self, x):
        return self.model(x)
```

上圖為 UnetDown 的程式碼，使用上述提到的 Resnet(Residual)和一個 maxPool2d 來實現，主要功能為減少輸入的尺寸，這邊的參數設計會讓圖片的尺寸少一半。

UnetUp:

```
class UnetUp(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UnetUp, self).__init__()
        '''
        process and upscale the image feature maps
        '''
        # 來實現一個卷積反轉層，作用是將上一層的特徵映射轉換回輸入圖像的大小，並且保留了更多的資訊
        up_layers = nn.Sequential(nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
                                  ResidualConvBlock(out_channels, out_channels),
                                  ResidualConvBlock(out_channels, out_channels),)

        self.model = up_layers

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

上圖為 UnetUp 的程式碼，使用 ConvTranspose2d(為一個卷積反

轉層，作用是將上一層的特徵映射轉換回輸入圖像的大小，並

且保留了更多的資訊)和兩個上述提到的 Resnet(Residual)。另

外，此 model 會先將輸入與 skip concatenate 在一起，用來恢復

在 Unetdown 中失去的資訊。

EmbedFC:

```
# 用來將各個條件(time或文字條件）壓成特徵向量
class EmbedFC(nn.Module):
    def __init__(self, input_dim, emb_dim):
        super(EmbedFC, self).__init__()
        '''
        generic one layer FC NN for embedding things
        '''
        self.input_dim = input_dim
        Embed_layers = nn.Sequential(nn.Linear(input_dim, emb_dim),
                                    nn.Tanh(),
                                    nn.Linear(emb_dim, emb_dim*2),
                                    nn.Tanh(),
                                    nn.Linear(emb_dim*2, emb_dim),)
        self.model = Embed_layers

    def forward(self, x):
        x = x.view(-1, self.input_dim).float()
        return self.model(x)
```

上述為 EmbedFC 的程式碼，使用三層 linear 和 2 個 Tanh 來當作

active function，此處的作用為將各個條件(time 或文字條件）壓

成特徵向量(大小為 batch_size * emb_dim)。

ContextUnet:

```python
class ContextUnet(nn.Module):
    def __init__(self, in_channels, n_feat=256, n_classes=24):
        super(ContextUnet, self).__init__()

        self.in_channels = in_channels
        self.n_feat = n_feat
        self.n_classes = n_classes

        self.init_conv = ResidualConvBlock(in_channels, n_feat, is_res=True)

        self.down1 = UnetDown(n_feat, n_feat)
        self.down2 = UnetDown(n_feat, 2 * n_feat)
        self.down3 = UnetDown(2 * n_feat, 4 * n_feat)

        """
        Be careful with this part, use nn.AvgPool2d() with the number that can divide current width
        and length with no remain
        """
        self.to_vec = nn.Sequential(nn.AvgPool2d(8), nn.Tanh())
        # 不同層的時間特徵
        self.timeembed0 = EmbedFC(1, 4*n_feat)
        self.timeembed1 = EmbedFC(1, 2*n_feat)
        self.timeembed2 = EmbedFC(1, 1*n_feat)
        # 不同層的condition
        self.contextembed0 = EmbedFC(n_classes, 4*n_feat)
        self.contextembed1 = EmbedFC(n_classes, 2*n_feat)
        self.contextembed2 = EmbedFC(n_classes, 1*n_feat)

        """
        If you change nn.AvgPool2d above, remember to set the last argment in nn.ConvTranspose2d
        carefully, it will upsampling the tensor.
        """
        # 用來將特徵壓更低
        self.bottle_neck = nn.Sequential(
            # 於將輸入圖像的特徵映射到較小的空間中
            nn.ConvTranspose2d(4 * n_feat, 4 * n_feat, 8, 8),  # otherwise just have 2*n_feat
            # 來實現一個分組的批次正規化（Group Normalization）操作
            nn.GroupNorm(8, 4 * n_feat),
            nn.ReLU(),
        )
        self.up0 = UnetUp(8 * n_feat, 2*n_feat)
        self.up1 = UnetUp(4 * n_feat, n_feat)
        self.up2 = UnetUp(2 * n_feat, n_feat)
        self.out = nn.Sequential(
            nn.Conv2d(2 * n_feat, n_feat, 3, 1, 1),
            nn.GroupNorm(8, n_feat),
            nn.ReLU(),
            nn.Conv2d(n_feat, self.in_channels, 3, 1, 1),
        )

    # x是照片, c是文字條件, t 是 timestep
    def forward(self, x, c, t, context_mask=None):
        # down layer
        x = self.init_conv(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        down3 = self.down3(down2)
        hiddenvec = self.to_vec(down3)
        # 各層將條件與time embeded 進來
        cemb0 = self.contextembed0(c).view(-1, self.n_feat * 4, 1, 1)
        temb0 = self.timeembed0(t).view(-1, self.n_feat * 4, 1, 1)
        cemb1 = self.contextembed1(c).view(-1, self.n_feat * 2, 1, 1)
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
        cemb2 = self.contextembed2(c).view(-1, self.n_feat, 1, 1)
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)


        # 最終先層bottle_neck, 並在upsampling每層中加入time與condition
        up0 = self.bottle_neck(hiddenvec)
        up1 = self.up0(cemb0 * up0 + temb0, down3)

        # print(cemb1.size())
        # print(up1.size())

        up2 = self.up1(cemb1 * up1 + temb1, down2)
        up3 = self.up2(cemb2 * up2 + temb2, down1)
        out = self.out(torch.cat((up3, x), 1))
        return out
```

這邊的設計十分簡單，利用上述介紹的 Unetdown 和 Unetup 來實現，使用 3 層 Unetdown 來將照片壓縮，中間使用 bottle_neck 再把特徵壓得更小，最後使用 3 層 Unetup 來將圖片的大小恢復原樣。其中的 bottle_neck 如上方程式碼所示，利用 ConvTranspose2d、GroupNorm 和 ReLU 來實現。

Hyperparameters and others:

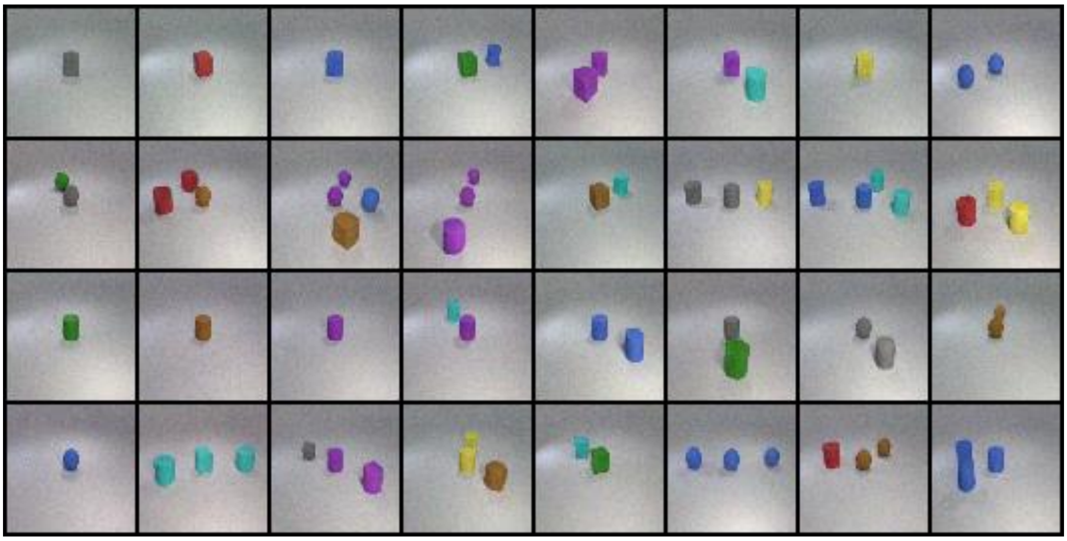
```python
parser = argparse.ArgumentParser()
parser.add_argument("--batch_size", type=int, default=8, help="please input batch_size")
parser.add_argument("--data_train_path", default="./iclevr", help="please input images' location")
parser.add_argument("--train_path", default="./dataset/train.json", help="path for train json file")
parser.add_argument("--test_path", default="./dataset/test.json", help="path for test json file")
parser.add_argument("--new_test_path", default="./dataset/new_test.json", help="path for test json file")
parser.add_argument("--object_path", default="./dataset/objects.json", help="path for object json file")
parser.add_argument("--n_epochs", type=int, default=50, help="number of epochs")
parser.add_argument("--n_objects", type=int, default=24, help="number of objects")
parser.add_argument("--n_feats", type=int, default=512, help="number of feats")
parser.add_argument("--lrate", type=float, default=1e-3, help="learning rate")
parser.add_argument("--n_T", type=float, default=300, help="n_T")
args = parser.parse_args()
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
```

上圖為我超參數的設計，其中最重要的參數我認為是 n_T，他代表最終充滿雜訊的圖與原圖之間的距離(time step)，也就是需要將幾次雜訊逐步加到原圖上，進而影響每次 timestep 需要加入的雜訊量。

3. Results and discussion:

Test.json:

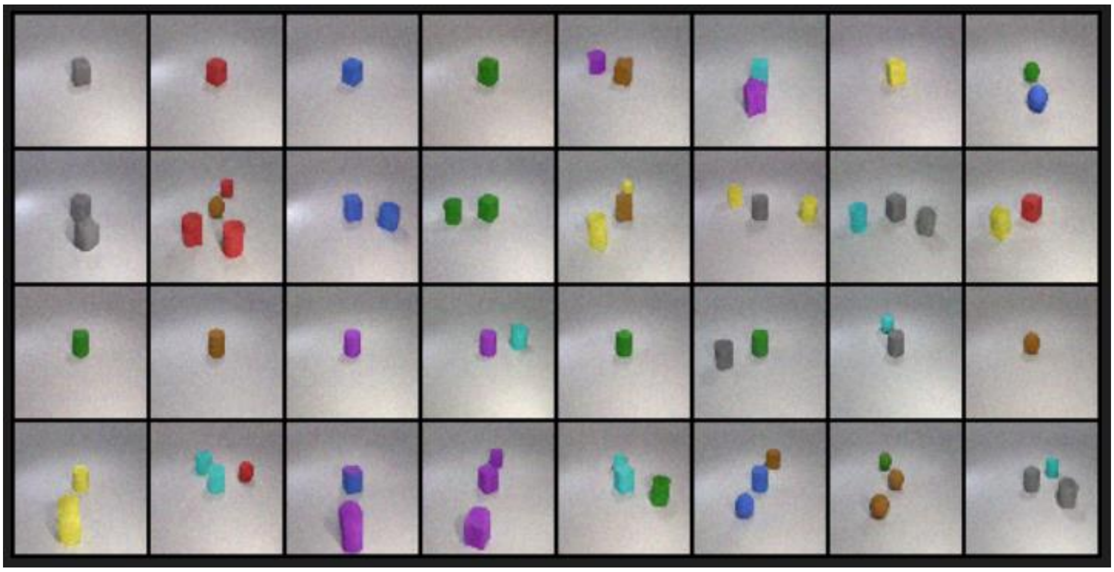
準確率為: 0.5794524448354262



New_test.json:

準確率為: 0.5952380952380952

```
(base) pp037@ec037:~/DL/lab7$ python eval.py
torch.Size([3, 64, 64])
/home/pp037/anaconda3/lib/python3.7/site-pack
weights' instead.
  f"The parameter '{pretrained_param}' is dep
/home/pp037/anaconda3/lib/python3.7/site-pack
moved in the future. The current behavior is
  warnings.warn(msg)
0.5952380952380952
```



Discussion:

我認為在本次的功課並沒有拿到很高的分數，分別只獲得 0.57 和 0.59，我認為是因為沒有將 position 也一併 embeded 進來的緣

故，僅根據 timestep 與文字條件似乎是無法讓 model 學習不同形狀與位置的差異，尤其是在 test 的時候，會生成出錯誤數量的物體，若之後有機會我會將位置資訊也一併考慮進去。另外，由於原先的圖片尺寸太大了(240*320)，我現在設計的三層 Unet(down 和 up)無法處理，但若是將層數條太高，又會讓整體訓練時間過程，因此這邊將原始照片的大小降成 64*64，才勉強能生成出看得懂的照片。