report

學號：311512049 姓名：陳緯翰

Introduction

- 使用的dataset是clevr dataset，有不同顏色跟形狀的物品的圖片。condition diffusion model的實作如下，利用UNET去學習雜訊的生成，並且加上one hot的condition，也就是用來表示顏色及方塊種類以及個數，來去訓練出一個更好的conditional diffusion model.

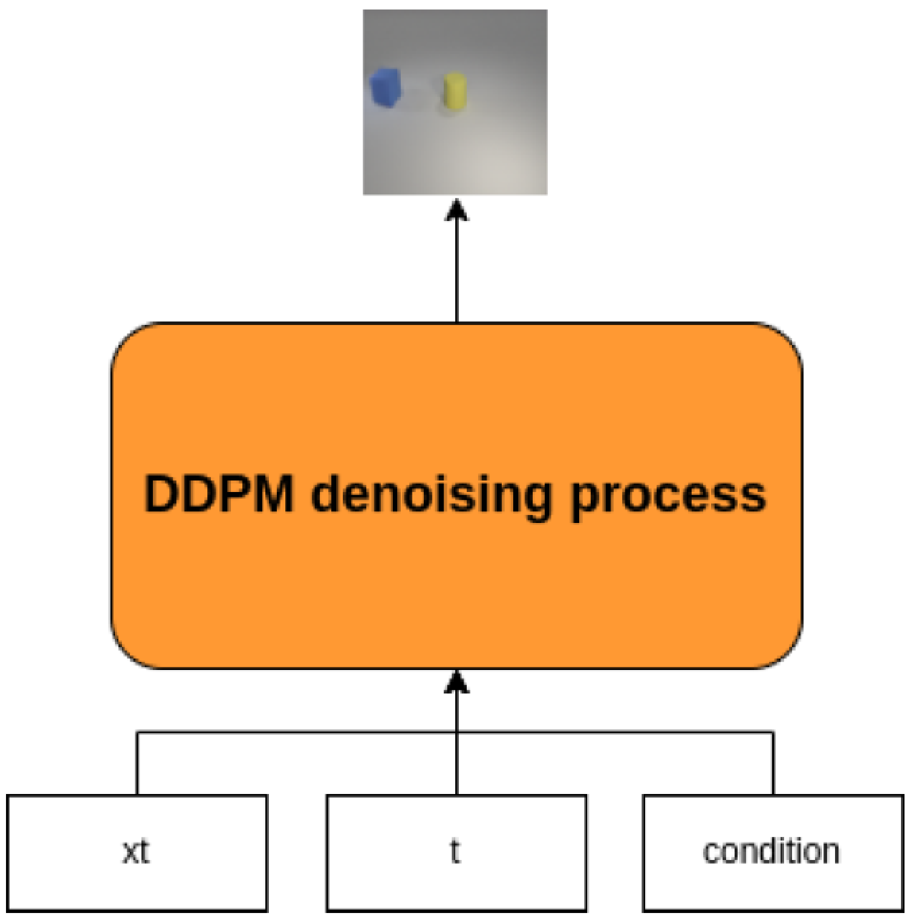


Figure 1: The illustration of conditional DDPM

Implement details

resnet

```
class ResidualConvBlock(nn.Module):
    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(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        if self.is_res:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)
            # this adds on correct residual in case channels have increased
            if self.same_channels:
                out = x + x2
            else:
                out = x1 + x2
            return out / 1.414
        else:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)
            return x2
```

Unet Downsampling

- 利用MaxPool2d將size減少兩倍

```
class UnetDown(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UnetDown, self).__init__()
        '''
        process and downscale the image feature maps
        '''
        layers = [ResidualConvBlock(in_channels, out_channels), nn.MaxPool2d(2)]
        self.model = nn.Sequential(*layers)

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

Unet Upsampling

- 利用ConvTranspose2d+兩層residual block將size增加兩倍，且輸入model的input由當前的x和之前的skip組成

```
class UnetUp(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UnetUp, self).__init__()
        '''
        process and upscale the image feature maps
        '''
        layers = [
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResidualConvBlock(out_channels, out_channels),
            ResidualConvBlock(out_channels, out_channels),
        ]
        self.model = nn.Sequential(*layers)

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

Embed Fully Convolution

```
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
        layers = [
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
        ]
        self.model = nn.Sequential(*layers)

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

Unet

```
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)

        self.to_vec = nn.Sequential(nn.AvgPool2d(8), nn.GELU())
```

```
        self.timeembed0 = EmbedFC(1, 4*n_feat)
        self.timeembed1 = EmbedFC(1, 2*n_feat)
        self.timeembed2 = EmbedFC(1, 1*n_feat)
        self.contextembed0 = EmbedFC(n_classes, 4*n_feat)
        self.contextembed1 = EmbedFC(n_classes, 2*n_feat)
        self.contextembed2 = EmbedFC(n_classes, 1*n_feat)

        self.up0 = nn.Sequential(
            # nn.ConvTranspose2d(6 * n_feat, 2 * n_feat, 7, 7), # when concat temb and cemb end up w 6*n_feat
            nn.ConvTranspose2d(4 * n_feat, 4 * n_feat, 8, 8), # otherwise just have 2*n_feat
            nn.GroupNorm(8, 4 * n_feat),
            nn.ReLU(),
        )

        self.up1 = UnetUp(8 * n_feat, 2*n_feat)   #!!!!!!!!!!!!!!!!!!!!!!!!!
        self.up2 = UnetUp(4 * n_feat, n_feat)
        self.up3 = 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),
        )

    def forward(self, x, c, t, context_mask = None):
        # x is (noisy) image, c is context label, t is timestep,
        # context_mask says which samples to block the context on

        x = self.init_conv(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        down3 = self.down3(down2)
        hiddenvec = self.to_vec(down3)

        # embed context, time step
        cemb1 = self.contextembed0(c).view(-1, self.n_feat * 4, 1, 1)  # torch.Size([128, 256, 1, 1])
        temb1 = self.timeembed0(t).view(-1, self.n_feat * 4, 1, 1)   # torch.Size([128, 256, 1, 1])
        cemb2 = self.contextembed1(c).view(-1, self.n_feat * 2, 1, 1)  # torch.Size([128, 128, 1, 1])
        temb2 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1) # torch.Size([128, 128, 1, 1])
        cemb3 = self.contextembed2(c).view(-1, self.n_feat, 1, 1) # torch.Size([128, 64, 1, 1])
        temb3 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)  # torch.Size([128, 64, 1, 1])

        up1 = self.up0(hiddenvec)   # torch.Size([128, 256, 8, 8])
        up2 = self.up1(cemb1*up1+ temb1, down3)  # add and multiply embeddings  # torch.Size([128, 256, 16, 16]
        up3 = self.up2(cemb2*up2+ temb2, down2)  # add and multiply embeddings  # torch.Size([128, 128, 32, 32])
        up4 = self.up3(cemb3*up3+ temb3, down1)
        out = self.out(torch.cat((up4, x), 1))
        return out
```

DDPM schedules

```
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}}
    }
```

DDPM

```
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 allows accessing dictionary produced by ddpm_schedules
        # e.g. can access self.sqrtab later
        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
        """

        _ts = torch.randint(1, self.n_T+1, (x.shape[0],)).to(self.device)  # t ~ Uniform(0, n_T)
        noise = torch.randn_like(x)  # eps ~ N(0, 1)

        x_t = (
            self.sqrtab[_ts, None, None, None] * x
            + self.sqrtmab[_ts, None, None, None] * noise
        )  # This is the x_t, which is sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
        # We should predict the "error term" from this x_t. Loss is what we return.

        # return MSE between added noise, and our predicted noise
        return self.loss_mse(noise, self.nn_model(x_t, c, _ts / self.n_T, c))
```

Hyperparameters

```
n_epoch = 150
batch_size = 64
n_T = 400
device = "cuda:0"
```

```
n_classes = 24
n_feat = 64 # 128, 256 better (but slower)
lrate = 1e-4
save_model = True
save_dir = './data/
```

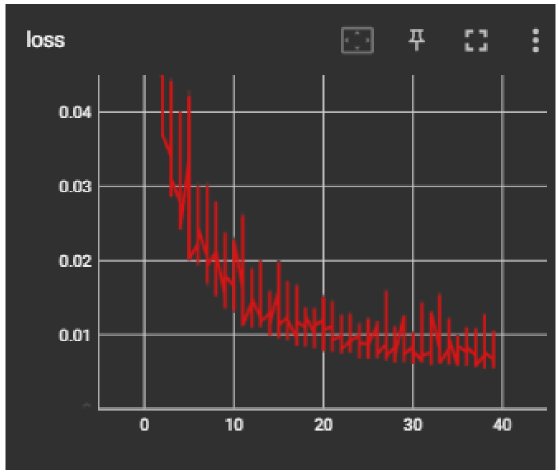
Results

- test.json (Score：0.6094444444444444)

- new_test.json (Score：0.6428571428571429)

- 修改github 開源的Conditional Diffusion MNIST，將其原本只有兩層的Unet變成3層來完成此任務，未來可以經由把resnet更改成self attention layer，或引入更多層的unet來完成此項任務。
- training loss

  當epoch設定為40到後段可以看到loss已經沒有再多做下降，可見此model的能力已經到極限，要竟由上面所提到的更改才有可能將performance作提升。


loss
0.04
0.03
0.02
0.01
0
10
20
30
40