• Introduction

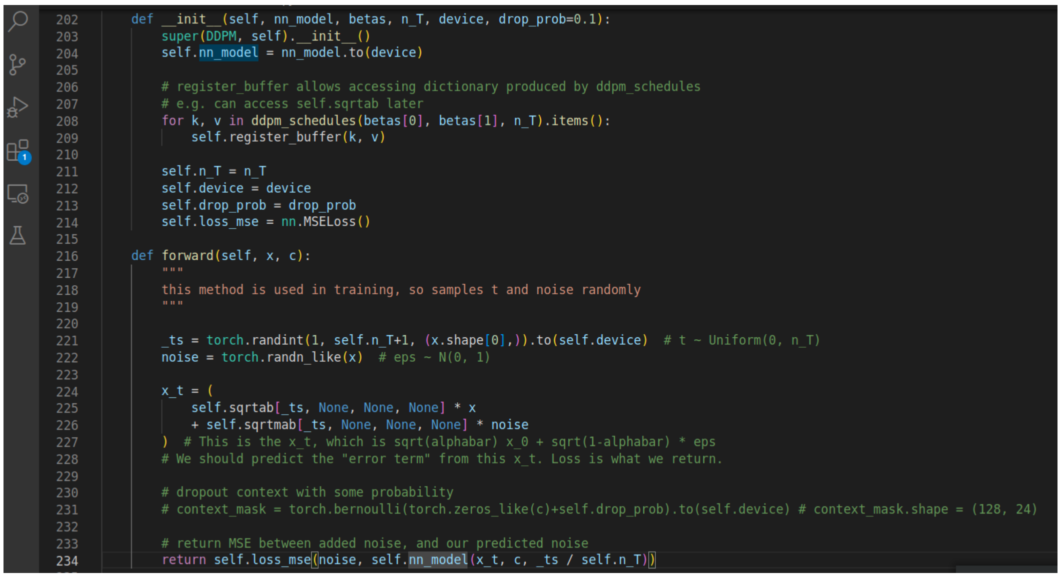
Diffusion model 是一種 generative model。在這次的 lab 中，我們要實作 diffusion model 的 DDPM，input 是一個 noise picture，透過在網路中不斷解噪，最後生成一張解噪後的，乾淨的預測圖片。

• Implementation details

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

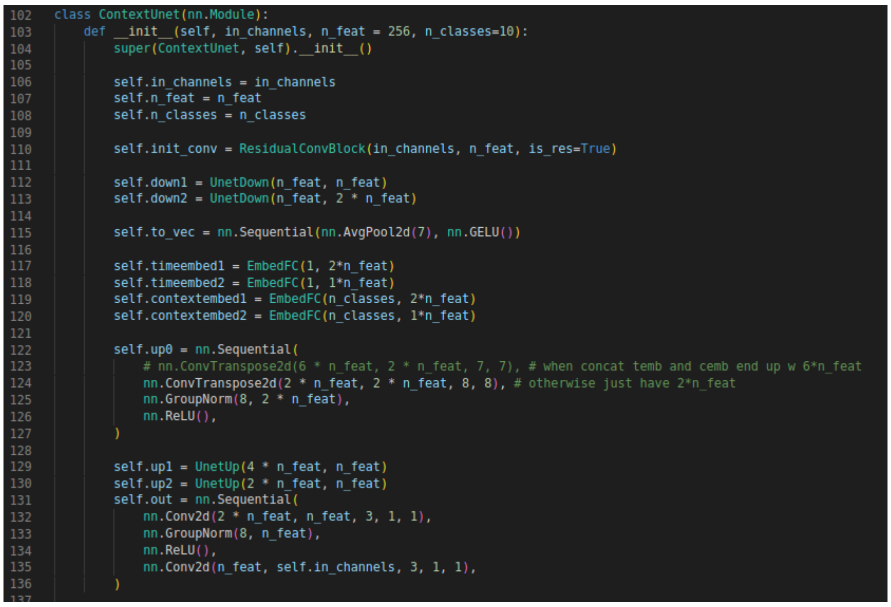
在這次的實作中，我選擇實作的是 pixel domain diffusion model，在網路中加噪解噪的過程是直接對一張圖片作處理，而不是把圖片變成 hidden space 後再作處理，所以是 **pixel domain** diffusion model。

在訓練的過程中，會輸入一張乾淨的照片還有這張照片的 type，在下面的 221-227 行，會根據之前所推導過的 DDPM 裡面的公式對這張照片加上 noise，這張加噪過的圖片表示是原本乾淨的圖片在經過 t 次加噪後的圖片。然後在第 234 行會把這張加噪過的圖片、圖片的 type、加噪幾次的 t，送進 UNet 中(就是圖中第 204 行的 self.nn_model)，去預測原始圖片是加了什麼 noise 才會變成這張加噪的圖片(所以 UNet 預測的是加了多少”noise”)。最後用 mean square error 計算預測的 noise 和真正的 noise 差了多少來做為 noise。

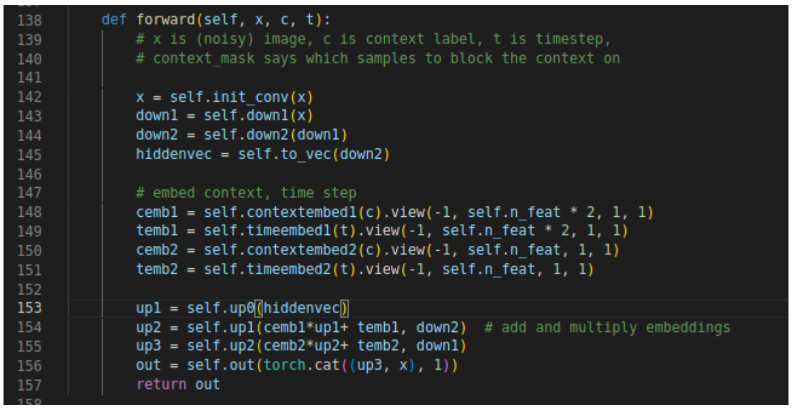


```
202     def __init__(self, nn_model, betas, n_T, device, drop_prob=0.1):
203         super(DDPM, self).__init__()
204         self.nn_model = nn_model.to(device)
205 
206         # register_buffer allows accessing dictionary produced by ddpm_schedules
207         # e.g. can access self.sqrtab later
208         for k, v in ddpm_schedules(betas[0], betas[1], n_T).items():
209             self.register_buffer(k, v)
210 
211         self.n_T = n_T
212         self.device = device
213         self.drop_prob = drop_prob
214         self.loss_mse = nn.MSELoss()
215 
216     def forward(self, x, c):
217         """
218         this method is used in training, so samples t and noise randomly
219         """
220 
221         _ts = torch.randint(1, self.n_T+1, (x.shape[0],)).to(self.device)  # t ~ Uniform(0, n_T)
222         noise = torch.randn_like(x)  # eps ~ N(0, 1)
223 
224         x_t = (
225             self.sqrtab[_ts, None, None, None] * x
226             + self.sqrtmab[_ts, None, None, None] * noise
227         )  # This is the x_t, which is sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
228         # We should predict the "error term" from this x_t. Loss is what we return.
229 
230         # dropout context with some probability
231         # context_mask = torch.bernoulli(torch.zeros_like(c)+self.drop_prob).to(self.device) # context_mask.shape = (128, 24)
232 
233         # return MSE between added noise, and our predicted noise
234         return self.loss_mse(noise, self.nn_model(x_t, c, _ts / self.n_T))
```

以下則是我的 UNet 實作架構。主要包含了兩層由 ResidualConvBlock(由兩層 Convolution 組成)和 MaxPool 所組成的 UNetDown，用意上像是 encode noise picture 到 hidden space。然後也會使用由 fully connected layers 所組成的 EmbedFC 來把傳進來的 t 和圖片的 type 也 encode 到 hidden space。接著會再把 encode 到 hidden space 的 noise picture、t 和圖片的 type 用兩層由 deconvolution block 和 ResidualConvBlock 組成的 UNetUp decode 回圖片的樣子，這個 decode 出來的圖片代表了預測的 apply 到乾淨圖片的 noise。

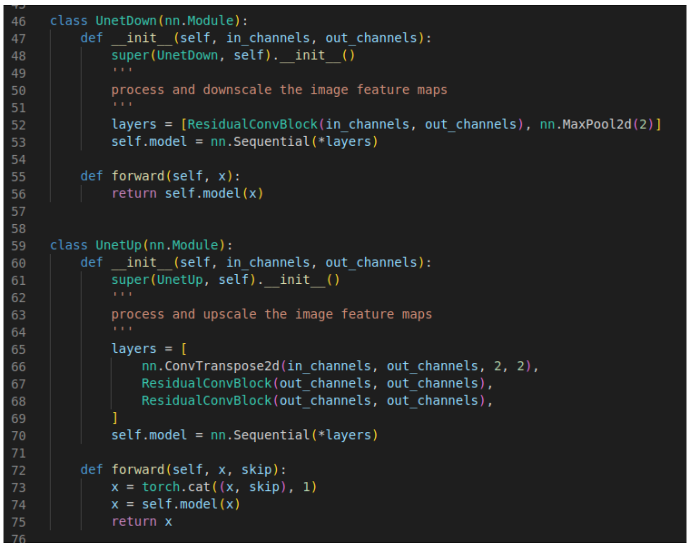


```python
class ContextUnet(nn.Module):
    def __init__(self, in_channels, n_feat = 256, n_classes=10):
        super(ContextUnet, self).__init__()

        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.to_vec = nn.Sequential(nn.AvgPool2d(7), nn.GELU())

        self.timeembed1 = EmbedFC(1, 2*n_feat)
        self.timeembed2 = EmbedFC(1, 1*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(2 * n_feat, 2 * n_feat, 8, 8), # otherwise just have 2*n_feat
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )

        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),
        )
```



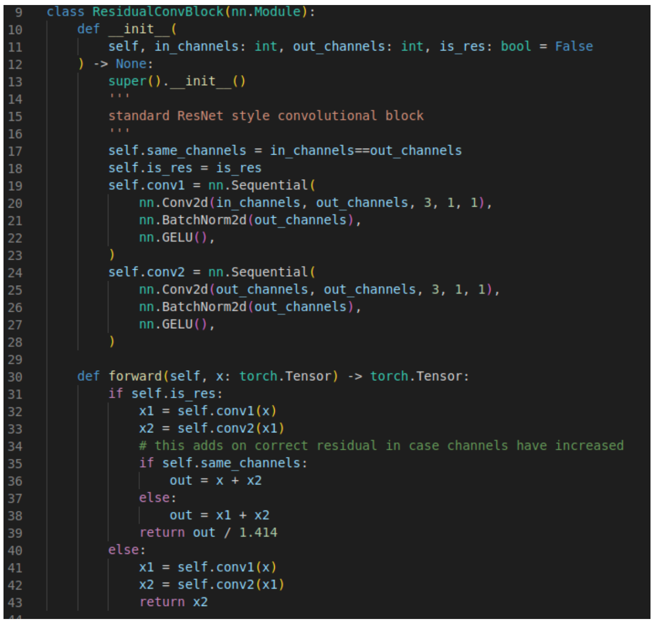
```python
    def forward(self, x, c, t):
        # x is (noisy) image, c is context label, t is timestep,
        # context_mask says which samples to block the context on

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

        # embed context, time step
        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)

        up1 = self.up0(hiddenvec)
        up2 = self.up1(cemb1*up1+ temb1, down2)  # add and multiply embeddings
        up3 = self.up2(cemb2*up2+ temb2, down1)
        out = self.out(torch.cat((up3, x), 1))
        return out
```

以下是前面所提到的 UNetUp 和 UNetDown 的架構。



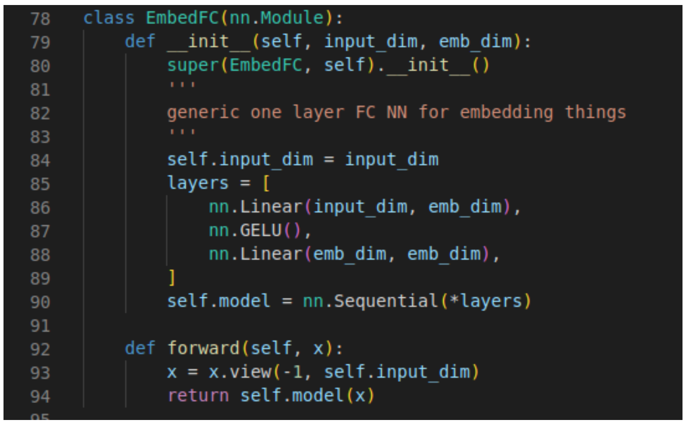
```python
class UnetDown(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UnetDown, self).__init__()
        '''
        process and downscale the image feature maps
        '''
        layers = [ResidualConvBlock(in_channels, out_channels), nn.MaxPool2d(2)]
        self.model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

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


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
```

以下是前面所提到的 ResidualConvBlock 的架構。



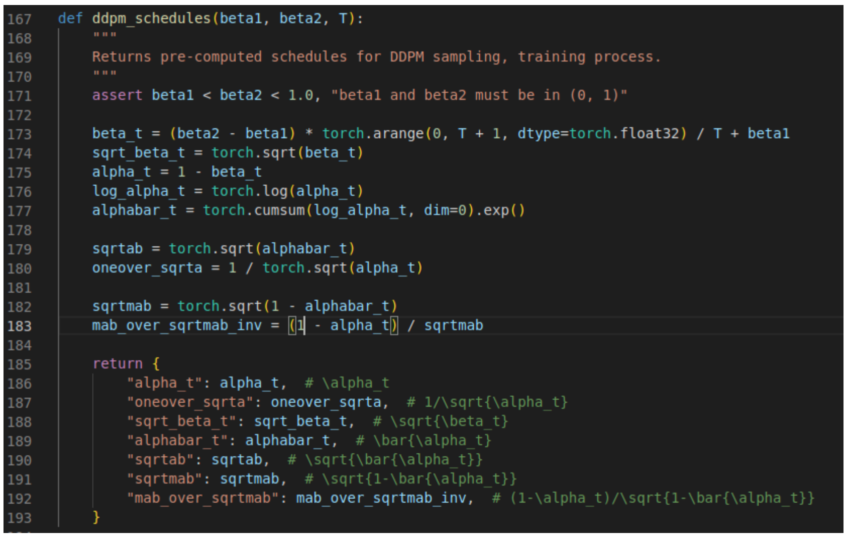
```python
class ResidualConvBlock(nn.Module):
    def __init__(
        self, in_channels: int, out_channels: int, is_res: bool = False
    ) -> None:
        super().__init__()
        '''
        standard ResNet style convolutional block
        '''
        self.same_channels = in_channels==out_channels
        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
```

以下是前面所提到的 EmbedFC 的架構。



```python
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)
```

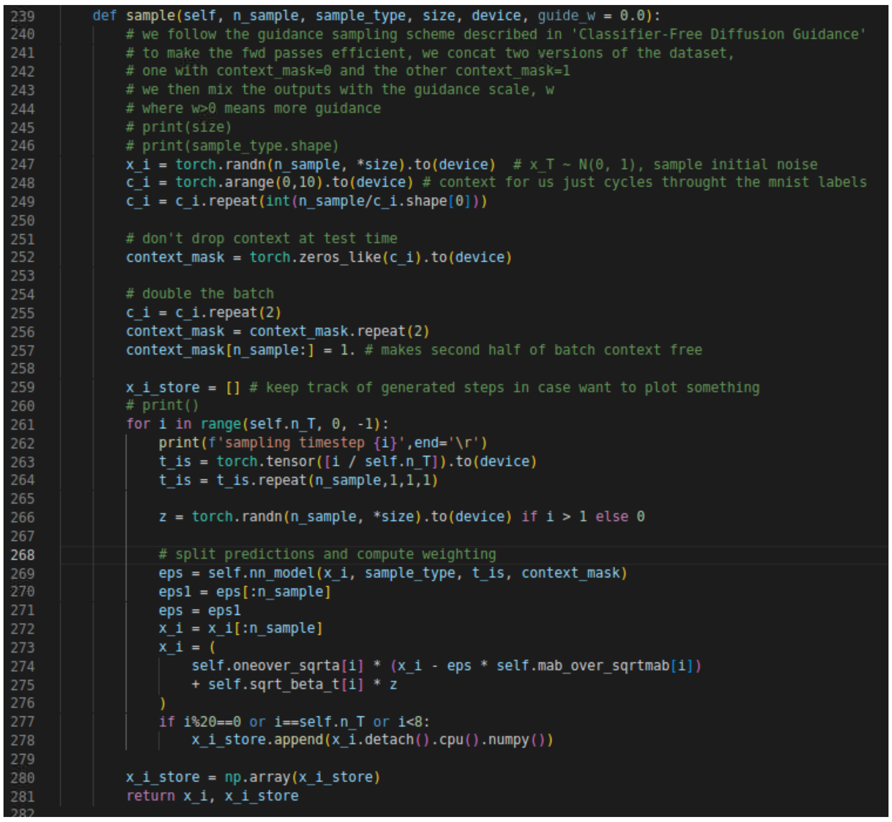
在實作 noise schedule 的部分，我是採用 linear 的方式去計算不同 t 時的 beta。t 從 1 變成 T 的過程中，beta 會從 beta1 線性變大到 beta2。其他 model 中會用到的一些變數，像是 alpha、beta bar 等等，則是根據 DDPM 論文中推導出來的公式所計算的。



```python
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}}
    }
```

在測試的時候，會去 random sample 出 noise 的圖片。這個 noise 圖片會 iteratively 被放進訓練好的 UNet 中，UNet 會預測出這第 t 層的 noise，然後依照公式扣掉第 t 次的 noise，算出第 t-1 層的圖片。這樣一直做下去直到最後，就會產生一張去噪的乾淨圖片。

以下是我實作此部分的 code。



```python
    def sample(self, n_sample, sample_type, size, device, guide_w = 0.0):
        # we follow the guidance sampling scheme described in 'Classifier-Free Diffusion Guidance'
        # to make the fwd passes efficient, we concat two versions of the dataset,
        # one with context_mask=0 and the other context_mask=1
        # we then mix the outputs with the guidance scale, w
        # where w>0 means more guidance
        # print(size)
        # print(sample_type.shape)
        x_i = torch.randn(n_sample, *size).to(device)  # x_T ~ N(0, 1), sample initial noise
        c_i = torch.arange(0,10).to(device) # context for us just cycles throught the mnist labels
        c_i = c_i.repeat(int(n_sample/c_i.shape[0]))

        # don't drop context at test time
        context_mask = torch.zeros_like(c_i).to(device)

        # double the batch
        c_i = c_i.repeat(2)
        context_mask = context_mask.repeat(2)
        context_mask[n_sample:] = 1. # makes second half of batch context free

        x_i_store = [] # keep track of generated steps in case want to plot something
        # print()
        for i in range(self.n_T, 0, -1):
            print(f'sampling timestep {i}',end='\r')
            t_is = torch.tensor([i / self.n_T]).to(device)
            t_is = t_is.repeat(n_sample,1,1,1)

            z = torch.randn(n_sample, *size).to(device) if i > 1 else 0

            # split predictions and compute weighting
            eps = self.nn_model(x_i, sample_type, t_is, context_mask)
            eps1 = eps[:n_sample]
            eps = eps1
            x_i = x_i[:n_sample]
            x_i = (
                self.oneover_sqrta[i] * (x_i - eps * self.mab_over_sqrtmab[i])
                + self.sqrt_beta_t[i] * z
            )
            if i%20==0 or i==self.n_T or i<8:
                x_i_store.append(x_i.detach().cpu().numpy())

        x_i_store = np.array(x_i_store)
        return x_i, x_i_store
```

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

以下是我訓練時的 hyperparameter 設定：

Learning rate: 0.0001

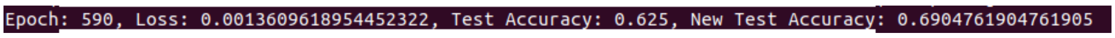
Epochs: 600

Beta1: 0.0001

Beta2: 0.02

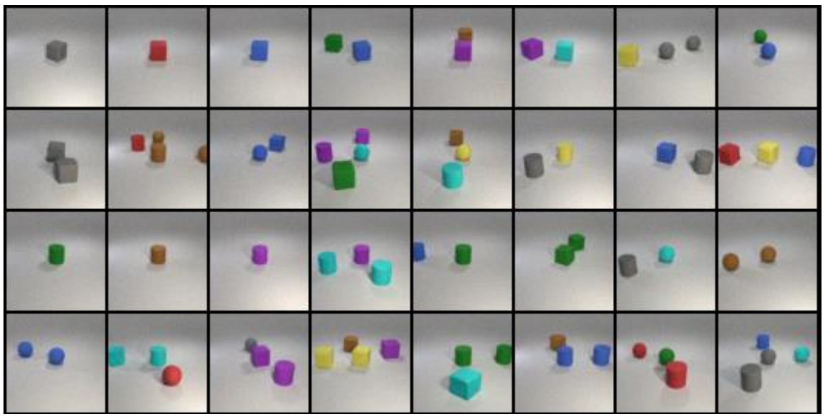
- Results and discussion

– Show your results based on the testing data. (including images)

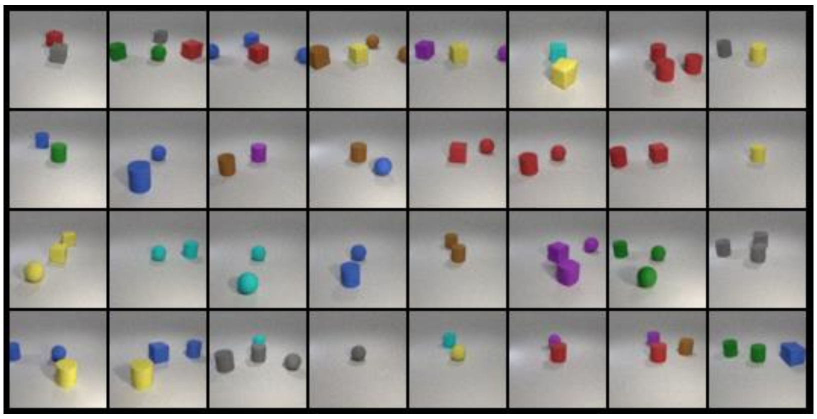


```
Epoch: 590, Loss: 0.0013609618954452322, Test Accuracy: 0.625, New Test Accuracy: 0.6904761904761905
```

用 test.json 產生的結果



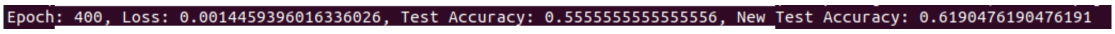
用 new_test.json 產生的結果



– Discuss the results of different model architectures. For example, what is the effect with or without some specific embedding methods, or what kind of prediction type is more effective in this case.

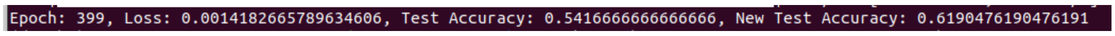
400 個 Epoch 的情況：

原本的實作結果



```
Epoch: 400, Loss: 0.0014459396016336026, Test Accuracy: 0.5555555555555556, New Test Accuracy: 0.6190476190476191
```

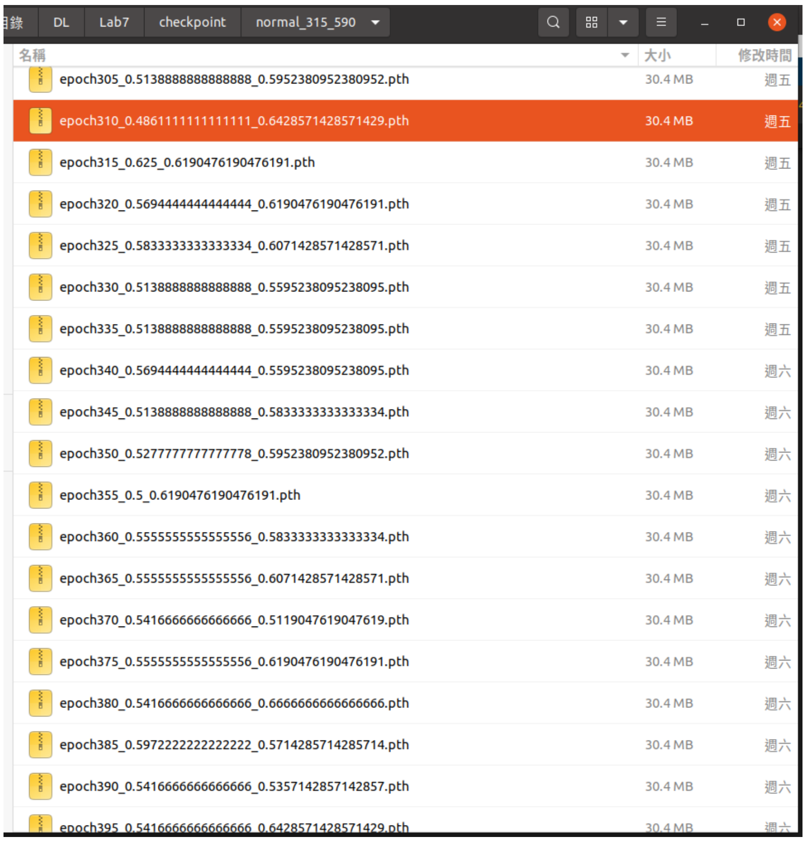
在實作 UNet 的架構時，我所找到的參考架構所使用的 activation function 是 GELU (Gaussian Error Linerar Unit)。我想說大家一般比較常用的 activation fuction 是 ReLU，因此就試試看將 model 的 activation function 替換成 ReLU，以下是使用 ReLU 的 model train 到第 400 個 epoch 的結果。



```
Epoch: 399, Loss: 0.0014182665789634606, Test Accuracy: 0.5416666666666666, New Test Accuracy: 0.6190476190476191
```

單純以第 400 個 epoch 的結果來說，我覺得使用 ReLU 的結果和使用 GELU 的結果不論在 loss 還是 accuracy 上都沒有很顯著的不同。不過如果是訓練過程中來說的話，用 GELU 訓練的結果感覺稍微比 ReLU 的好一點，在大約 300~400 個 epoch 時，普遍用 GELU 訓練的 test.json 結果幾乎都大於 50，不過用 ReLU 的訓練結果則出現較多 40 幾的準確率。

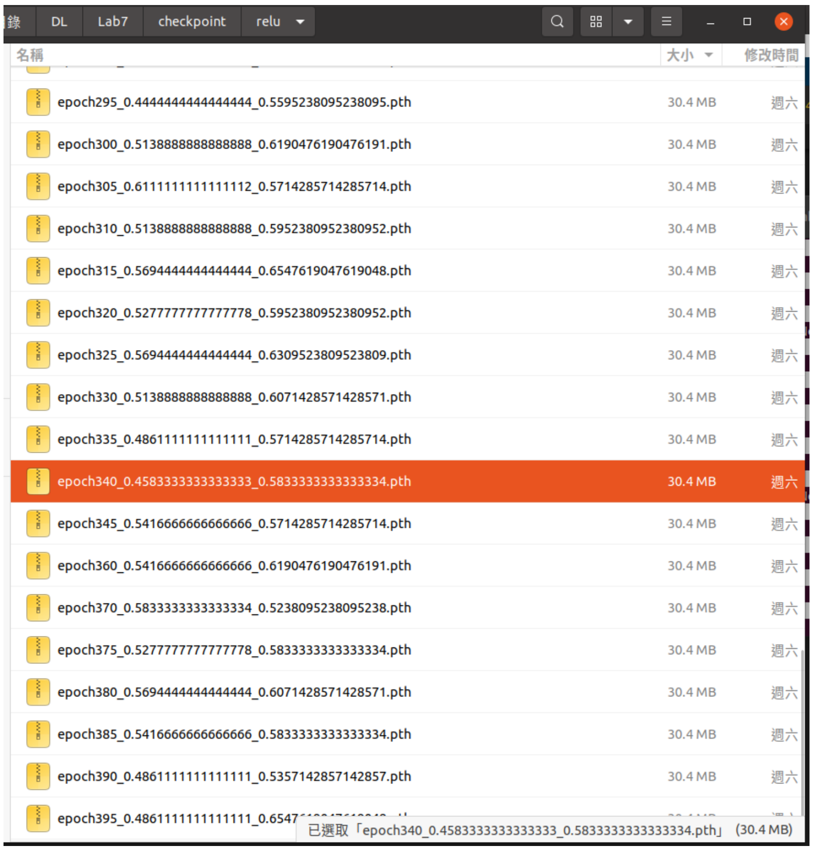
下圖為用 GELU 訓練的過程，檔名的兩個數字分別代表用 test.json 和 new_test.json 的訓練結果。



錄 DL Lab7 checkpoint normal_315_590

名稱	大小	修改時間
epoch305_0.5138888888888888_0.5952380952380952.pth	30.4 MB	週五
epoch310_0.4861111111111111_0.6428571428571429.pth	30.4 MB	週五
epoch315_0.625_0.6190476190476191.pth	30.4 MB	週五
epoch320_0.5694444444444444_0.6190476190476191.pth	30.4 MB	週五
epoch325_0.5833333333333334_0.6071428571428571.pth	30.4 MB	週五
epoch330_0.5138888888888888_0.5595238095238095.pth	30.4 MB	週五
epoch335_0.5138888888888888_0.5595238095238095.pth	30.4 MB	週五
epoch340_0.5694444444444444_0.5595238095238095.pth	30.4 MB	週六
epoch345_0.5138888888888888_0.5833333333333334.pth	30.4 MB	週六
epoch350_0.5277777777777778_0.5952380952380952.pth	30.4 MB	週六
epoch355_0.5_0.6190476190476191.pth	30.4 MB	週六
epoch360_0.5555555555555556_0.5833333333333334.pth	30.4 MB	週六
epoch365_0.5555555555555556_0.6071428571428571.pth	30.4 MB	週六
epoch370_0.5416666666666666_0.5119047619047619.pth	30.4 MB	週六
epoch375_0.5555555555555556_0.6190476190476191.pth	30.4 MB	週六
epoch380_0.5416666666666666_0.6666666666666666.pth	30.4 MB	週六
epoch385_0.5972222222222222_0.5714285714285714.pth	30.4 MB	週六
epoch390_0.5416666666666666_0.5357142857142857.pth	30.4 MB	週六
epoch395_0.5416666666666666_0.6428571428571429.pth	30.4 MB	週六

下圖為用 ReLU 訓練的過程，檔名的兩個數字分別代表用 test.json 和 new_test.json 的訓練結果。



錄 DL Lab7 checkpoint relu

名稱	大小	修改時間
epoch295_0.4444444444444444_0.5595238095238095.pth	30.4 MB	週六
epoch300_0.5138888888888888_0.6190476190476191.pth	30.4 MB	週六
epoch305_0.6111111111111112_0.5714285714285714.pth	30.4 MB	週六
epoch310_0.5138888888888888_0.5952380952380952.pth	30.4 MB	週六
epoch315_0.5694444444444444_0.6547619047619048.pth	30.4 MB	週六
epoch320_0.5277777777777778_0.5952380952380952.pth	30.4 MB	週六
epoch325_0.5694444444444444_0.6309523809523809.pth	30.4 MB	週六
epoch330_0.5138888888888888_0.6071428571428571.pth	30.4 MB	週六
epoch335_0.4861111111111111_0.5714285714285714.pth	30.4 MB	週六
epoch340_0.4583333333333333_0.5833333333333334.pth	30.4 MB	週六
epoch345_0.5416666666666666_0.5714285714285714.pth	30.4 MB	週六
epoch360_0.5416666666666666_0.6190476190476191.pth	30.4 MB	週六
epoch370_0.5833333333333334_0.5238095238095238.pth	30.4 MB	週六
epoch375_0.5277777777777778_0.5833333333333334.pth	30.4 MB	週六
epoch380_0.5694444444444444_0.6071428571428571.pth	30.4 MB	週六
epoch385_0.5416666666666666_0.5833333333333334.pth	30.4 MB	週六
epoch390_0.4861111111111111_0.5357142857142857.pth	30.4 MB	週六
epoch395_0.4861111111111111_0.6547		

已選取「epoch340_0.4583333333333333_0.5833333333333334.pth」 (30.4 MB)

根據網路上搜尋的資料所說，GELU 比起 ReLU，可以有更好的 performance 和較快的收斂速度。同時他也是連續的函數。

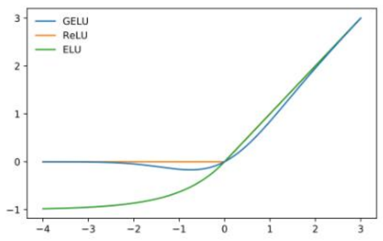
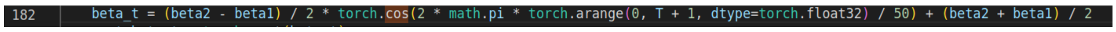


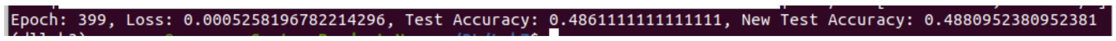
Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

另一個我所嘗試的是，是在計算 noise schedule 時，將原本線性上升的 beta 值，改為用 cos 的方式，在 beta1 和 beta2 之間週期變動。因為我的 T 最大是 400 個，我就取整數讓 cos 的變化週期是 50。



```
beta_t = (beta2 - beta1) / 2 * torch.cos(2 * math.pi * torch.arange(0, T + 1, dtype=torch.float32) / 50) + (beta2 + beta1) / 2
```

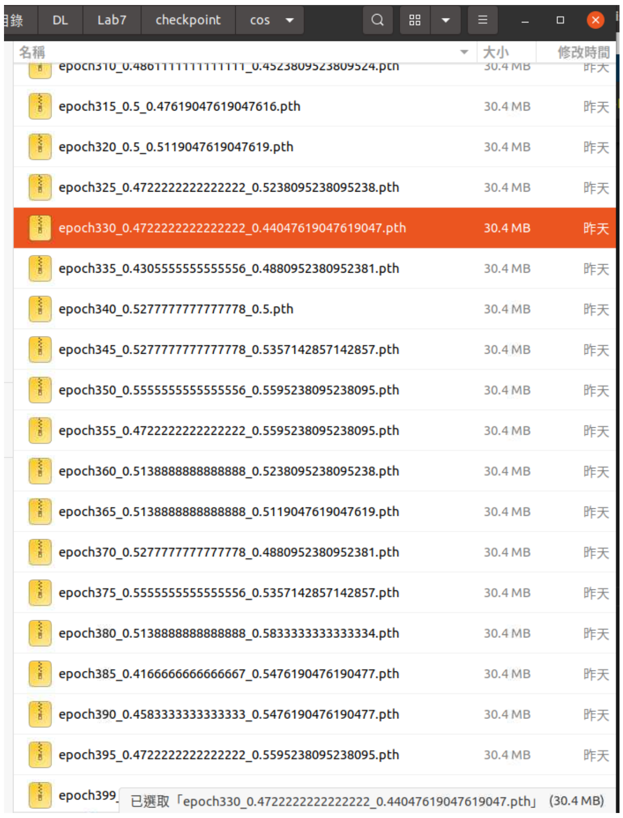
Cos:



```
Epoch: 399, Loss: 0.0005258196782214296, Test Accuracy: 0.4861111111111111, New Test Accuracy: 0.4880952380952381
```

在 train 至第 400 個 epoch 的最終結果上，test 的 accuracy 明顯是原本的 linear 的結果比較好(過程也是)。我覺得因為原本線性的做法是越到後面的 t，beta 越大，相當於是越到後面的 t 加的 noise 越多。我覺得這樣的好處是，在 test 去噪的時候，因為是從最後一個 T 開始，一步一步去噪，這樣可以讓它一開始除掉那些很明顯的 noise，然後慢慢的影像變乾淨的過程中，再去除掉那些很細微的 noise，讓影像 quality 變得更好。

下圖為用 cos 訓練的過程，檔名的兩個數字分別代表用 test.json 和 new_test.json 的訓練結果。



```
名稱                                                   大小      修改時間
epoch310_0.4861111111111111_0.4523809523809524.pth    30.4 MB   昨天
epoch315_0.5_0.47619047619047616.pth                  30.4 MB   昨天
epoch320_0.5_0.5119047619047619.pth                   30.4 MB   昨天
epoch325_0.4722222222222222_0.5238095238095238.pth    30.4 MB   昨天
epoch330_0.4722222222222222_0.44047619047619047.pth   30.4 MB   昨天
epoch335_0.4305555555555556_0.4880952380952381.pth    30.4 MB   昨天
epoch340_0.5277777777777778_0.5.pth                   30.4 MB   昨天
epoch345_0.5277777777777778_0.5357142857142857.pth    30.4 MB   昨天
epoch350_0.5555555555555556_0.5595238095238095.pth    30.4 MB   昨天
epoch355_0.4722222222222222_0.5595238095238095.pth    30.4 MB   昨天
epoch360_0.5138888888888888_0.5238095238095238.pth    30.4 MB   昨天
epoch365_0.5138888888888888_0.5119047619047619.pth    30.4 MB   昨天
epoch370_0.5277777777777778_0.4880952380952381.pth    30.4 MB   昨天
epoch375_0.5555555555555556_0.5357142857142857.pth    30.4 MB   昨天
epoch380_0.5138888888888888_0.5833333333333334.pth    30.4 MB   昨天
epoch385_0.4166666666666667_0.5476190476190477.pth    30.4 MB   昨天
epoch390_0.4583333333333333_0.5476190476190477.pth    30.4 MB   昨天
epoch395_0.4722222222222222_0.5595238095238095.pth    30.4 MB   昨天
epoch399_   已選取「epoch330_0.4722222222222222_0.44047619047619047.pth」 (30.4 MB)
```

另外再 cos 的實驗結果中，可以發現 train 到第 400 個 epoch 的時候，loss 比前面用 GELU 或是 ReLU 的都還要小。這個原因應該不是調整架構的關係，是因為我再 optimize 的地方加上了 learning rate decay 的東西。從這個 loss 的結果也可以看出，在這次的 lab 中即使 loss 比較小，也不一定 accuracy 就會比較好。

```
# linear lrate decay
optimizer.param_groups[0]['lr'] = args.lr*(1-epoch/args.epochs)
```