

# Deep Learning

## Lab5: Conditional VAE For Video Prediction

311512064 鄧書桓

### 1. Introduction:

本次實驗是使用 CVAE 來實作出影片生成的任務，根據過去已知的影片、動作與位置條件，加上 encoder 學習資料之間的分佈關係，再利用 encoder 的輸出、latent vector 以及動作和位置條件作為輸入，生成出未來下一個時間點的影片。

### 2. Derivation of CVAE

Derivation of CVAE:

對於 CVAE 來說，目標是要去 learn 在給定條件  $c$  下的分佈空間，可以表示成  $p(x|c; \theta)$ ，希望此分佈越大越好，其中  $\theta$  是模型要 learn 的參數。

利用聯合機率來表示： $p(x|c; \theta) = \int p(x|z, c; \theta) p(z|c) dz$

其中  $z$  代表取樣  $x$  之間的 latent variable

為了方便計算取 log 變為： $\log p(x|c; \theta) = \log p(x, z|c; \theta) - \log p(z|c; \theta)$

並將要學習的分佈設為任意分佈  $g(z|c)$  代入

$$\begin{aligned}\log p(x|c; \theta) &= \int g(z|c) \log p(x|z, c; \theta) dz \\ &= \int g(z|c) \log p(x, z|c; \theta) dz - \int g(z|c) \log p(z|c; \theta) dz \\ &= \int g(z|c) \log p(x, z|c; \theta) dz - \int g(z|c) \log g(z|c) dz \\ &\quad + \int g(z|c) \log g(z|c) dz - \int g(z|c) \log p(z|c; \theta) dz \\ &= \boxed{L(x, g, \theta|c)} + \boxed{KL(g(z|c) \| p(z|c; \theta))} \quad \text{--- ①} \\ &= \boxed{\int g(z|c) \log p(x, z|c; \theta) dz} - \boxed{\int g(z|c) \log g(z|c) dz} \\ &\quad + \boxed{\int g(z|c) \log \frac{g(z|c)}{p(z|c; \theta)} dz} \quad \text{ELBO} \quad \text{KL}\end{aligned}$$

因為要最大化  $p(x|c; \theta)$ ，又因 KL 是  $g(z|c)$  與  $p(z|c; \theta)$  的分佈一致度，

∴ 只要最大化 ELBO，將①改寫：

$$L(x, g, \theta|c) = \log p(x|c; \theta) - KL(g(z|c) \| p(z|c; \theta))$$

再由 VAE 的 encoder 給出來的分佈  $g(z|c, \theta')$  代入， $\theta'$  為 Encoder 參數

$$\begin{aligned}L(x, g, \theta|c) &= \log p(x|c; \theta) - KL(g(z|c, \theta') \| p(z|c; \theta)) \\ &= E_{g(z|c, \theta')} [\log p(x|z; \theta) + \log p(z|c) - \log g(z|c, \theta')] \\ &= \underbrace{E_{g(z|c, \theta')} [\log p(x|z; \theta) + \log p(z|c)]}_{\text{最大化 reconstruction loss}} - \underbrace{KL(g(z|c, \theta') \| p(z|c))}_{\text{最小化分佈的 KL}}\end{aligned}$$

### 3. Implementation details:

#### A. Encoder

這邊是使用助教提供的 vgg64 encoder 來達成，其中包含了 5 個 convolution layer(c1~c5)，由不同數量的 vgg\_layer 組成。主要功能是把輸入的照片壓縮成較小的向量，也就是逐漸降維，透過不同 convolution layer 保留不同的特徵向量。在每個 convolution layer 都有分別把該層向量存起來，使其可以使用 skip，代表在訓練過程中，跳過一些中間層，以解決梯度消失問題。詳細細節下圖 code 截圖中有介紹。

```
1 class vgg_encoder(nn.Module):
2     def __init__(self, dim):
3         super(vgg_encoder, self).__init__()
4         self.dim = dim
5         # 64 x 64
6         self.c1 = nn.Sequential(
7             vgg_layer(3, 64),
8             vgg_layer(64, 64),
9         )
10        # 32 x 32
11        self.c2 = nn.Sequential(
12            vgg_layer(64, 128),
13            vgg_layer(128, 128),
14        )
15        # 16 x 16
16        self.c3 = nn.Sequential(
17            vgg_layer(128, 256),
18            vgg_layer(256, 256),
19            vgg_layer(256, 256),
20        )
21        # 8 x 8
22        self.c4 = nn.Sequential(
23            vgg_layer(256, 512),
24            vgg_layer(512, 512),
25            vgg_layer(512, 512),
26        )
27        # 4 x 4
28        self.c5 = nn.Sequential(
29            nn.Conv2d(512, dim, 4, 1, 0),
30            nn.BatchNorm2d(dim),
31            nn.Tanh()
32        )
33        # 最大池化層的作用是把輸入的數據區域內的最大值作為輸出，從而實現對輸入數據的下採樣，減少數據的維度和大小。
34        # 這邊根據kernel_size與stride可知維度會少1/2，若兩者不一樣，須額外計算會少多少維度
35        self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
36
37    def forward(self, input):
38        # 接受一個大小為 64 的輸入，通過四個卷積層和池化層進行處理，最終輸出一個大小為 1 的特徵向量
39        # 將輸入進行四次卷積操作，每次將特徵圖的大小減半，然後進行池化操作，將特徵圖的大小再次減半，
40        # 最終通過一個全連接層輸出一個特徵向量
41        h1 = self.c1(input) # 64 -> 32
42        h2 = self.c2(self.mp(h1)) # 32 -> 16
43        h3 = self.c3(self.mp(h2)) # 16 -> 8
44        h4 = self.c4(self.mp(h3)) # 8 -> 4
45        h5 = self.c5(self.mp(h4)) # 4 -> 1
46        return h5.view(-1, self.dim), [h1, h2, h3, h4]
```

## B. Decoder

這邊是使用助教提供的簡化版的 vgg64 decoder，包含了五個 convolution layer(upc1~upc5)，由不同數量的 vgg\_layer 組成。與 encoder 不同的是要把向量逐漸升維，透過不同 convolution layer 放大維度到還原圖片。再還原過程中，同時將對應的 skip 與特徵向量連接起來，利用 skip connections 技巧還原圖片，以保留更多圖片細節。詳細細節下圖 code 截圖中有介紹。

```
1 class vgg_decoder(nn.Module):
2     def __init__(self, dim):
3         super(vgg_decoder, self).__init__()
4         self.dim = dim
5         # 1 x 1 -> 4 x 4
6         self.upc1 = nn.Sequential(
7             # ConvTranspose2d 一個反卷積層，用於將輸入的二維數據進行反卷積操作。
8             # 將輸入的數據進行放大，並且增加通道數，以便進行下一步的處理
9             nn.ConvTranspose2d(dim, 512, 4, 1, 0),
10            nn.BatchNorm2d(512),
11            nn.LeakyReLU(0.2, inplace=True)
12        )
13        # 8 x 8
14        self.upc2 = nn.Sequential(
15            vgg_layer(512*2, 512),
16            vgg_layer(512, 512),
17            vgg_layer(512, 256)
18        )
19        # 16 x 16
20        self.upc3 = nn.Sequential(
21            vgg_layer(256*2, 256),
22            vgg_layer(256, 256),
23            vgg_layer(256, 128)
24        )
25        # 32 x 32
26        self.upc4 = nn.Sequential(
27            vgg_layer(128*2, 128),
28            vgg_layer(128, 64)
29        )
30        # 64 x 64
31        self.upc5 = nn.Sequential(
32            vgg_layer(64*2, 64),
33            nn.ConvTranspose2d(64, 3, 3, 1, 1),
34            nn.Sigmoid()
35        )
36        # 將輸入的 2D 張量進行最近鄰居插值上採樣，將其大小縮放為原來的兩倍。
37        # 換句話說，它將輸入的圖像或特徵圖放大兩倍，並使用最近鄰居插值法填充新的像素值。
38        self.up = nn.UpsamplingNearest2d(scale_factor=2)
39
40    def forward(self, input):
41        vec, skip = input
42        d1 = self.upc1(vec.view(-1, self.dim, 1, 1)) # 1 -> 4
43        up1 = self.up(d1) # 4 -> 8
44        d2 = self.upc2(torch.cat([up1, skip[3]], 1)) # 8 x 8
45        up2 = self.up(d2) # 8 -> 16
46        d3 = self.upc3(torch.cat([up2, skip[2]], 1)) # 16 x 16
47        up3 = self.up(d3) # 16 -> 32
48        d4 = self.upc4(torch.cat([up3, skip[1]], 1)) # 32 x 32
49        up4 = self.up(d4) # 32 -> 64
50        output = self.upc5(torch.cat([up4, skip[0]], 1)) # 64 x 64
51        return output
52
```

### C. LSTM

這邊 LSTM 是用來將 encoder 的輸入學習特徵，並將其作為 decoder 的 input。而 gaussian LSTM 是把 latent variable (z) 與後驗分布一起使用，以生成更符合真實的數據分布。

```
1 class lstm(nn.Module):
2     def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
3         super(lstm, self).__init__()
4         self.device = device
5         self.input_size = input_size
6         self.output_size = output_size
7         self.hidden_size = hidden_size
8         self.batch_size = batch_size
9         self.n_layers = n_layers
10        self.embed = nn.Linear(input_size, hidden_size)
11        #hidden_size 是 LSTM 網路的隱藏層大小，n_layers 是 LSTM 網路的層數，每個 LSTMCell 都是一個單獨的 LSTM 單元
12        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
13        self.output = nn.Sequential(
14            nn.Linear(hidden_size, output_size),
15            nn.BatchNorm1d(output_size),
16            nn.Tanh())
17        self.hidden = self.init_hidden()
18
19    def init_hidden(self):
20        hidden = []
21        for _ in range(self.n_layers):
22            # LSTM 模型中的隱藏狀態就包含了一個細胞狀態和一個隱藏狀態，因此，這邊需要兩個零張量來初始化這兩個部分的隱藏狀態
23            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device)),
24                               Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device))))
25        return hidden
26
27    def forward(self, input):
28        embedded = self.embed(input)
29        h_in = embedded
30        for i in range(self.n_layers):
31            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
32            h_in = self.hidden[i][0]
33
34        return self.output(h_in)
```

```
1 class gaussian_lstm(nn.Module):
2     def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
3         super(gaussian_lstm, self).__init__()
4         self.device = device
5         self.input_size = input_size
6         self.output_size = output_size
7         self.hidden_size = hidden_size
8         self.n_layers = n_layers
9         self.batch_size = batch_size
10        self.embed = nn.Linear(input_size, hidden_size)
11        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
12        self.mu_net = nn.Linear(hidden_size, output_size)
13        self.logvar_net = nn.Linear(hidden_size, output_size)
14        self.hidden = self.init_hidden()
15
16    def init_hidden(self):
17        hidden = []
18        for _ in range(self.n_layers):
19            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device)),
20                               Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device))))
21        return hidden
22
23    def reparameterize(self, mu, logvar):
24        std = torch.exp(0.5*logvar)
25        eps = torch.randn_like(std)
26        return mu + eps*std
27
28    def forward(self, input):
29        embedded = self.embed(input)
30        h_in = embedded
31        for i in range(self.n_layers):
32            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
33            h_in = self.hidden[i][0]
34        mu = self.mu_net(h_in)
35        logvar = self.logvar_net(h_in)
36        z = self.reparameterize(mu, logvar)
37        return z, mu, logvar
```


#### D. Reparameterization Trick

使用原因:

直接從分佈中採樣得到的潛在變量不可導，不能夠直接用於反向傳播計算梯度。

解決辦法:

VAE 在 decoder 抽取 latent variable  $z$  作為輸入時，在模型計算梯度的部分，利用 reparameterization trick 將分佈視為連續的高斯分佈，可以更好計算梯度。先將 log-variance 轉成 sigma，並從高斯分佈抽樣出一個變數與 encoder 生成的分佈(sigma、mu) sigma 相乘並加 mu。



```
1 def reparameterize(self, mu, logvar):
2     std = torch.exp(0.5*logvar)
3     eps = torch.randn_like(std)
4     return mu + eps*std
```

#### E. Data loader

這邊是使用助教提供 data loader 來實現，其中 \_\_init\_\_ 是判斷對哪一種資料集(train、val、test)，以及圖像轉換 transform。\_\_len\_\_ 要計算資料總長度。get\_seq 選擇一個路徑從中讀取圖片的順序並將其轉成向量。get\_csv: 從 csv 對應相對照片來記錄機器人動作和位置(條件)。\_\_getitem\_\_ 將資料對應打包，把圖片的序列和條件作為向量回傳。

```

1 class bair_robot_pushing_dataset(Dataset):
2     def __init__(self, args, mode='train', transform=default_transform):
3         assert mode == 'train' or mode == 'test' or mode == 'validate'
4         self.root = '{}/{}'.format(args.data_root, mode)
5         self.seq_len = max(args.n_past + args.n_future, args.n_eval)
6         self.mode = mode
7         if mode == 'train':
8             self.ordered = False
9         else:
10            self.ordered = True
11
12        self.transform = transform
13        self.dirs = []
14        # 首先，它會使用os.listdir()函數列出指定路徑下的所有目錄，並將它們存儲在dir1變數中。
15        # 接下來，它會使用os.listdir()函數列出dir1目錄下的所有目錄，並將它們存儲在dir2變數中。
16        # 然後，它會使用os.path.join()函數將root、dir1和dir2組合成一個完整的目錄路徑，並將其添加到dirs列表中。
17        # 最終，dirs列表將包含指定路徑下的所有目錄的路徑。
18        for dir1 in os.listdir(self.root):
19            for dir2 in os.listdir(os.path.join(self.root, dir1)):
20                self.dirs.append(os.path.join(self.root, dir1, dir2))
21
22        self.seed_is_set = False
23        self.idx = 0
24        self.cur_dir = self.dirs[0]
25        self.d=0
26
27    def set_seed(self, seed):
28        if not self.seed_is_set:
29            self.seed_is_set = True
30            np.random.seed(seed)
31
32    def __len__(self):
33        return len(self.dirs)
34
35    def get_seq(self):
36        # 如果ordered為真，則會按照順序讀取資料集中的圖片序列
37        if self.ordered:
38            self.cur_dir = self.dirs[self.d]
39            if self.idx == len(self.dirs) - 1:
40                self.idx = 0
41            else:
42                self.idx += 1
43        # 否則，會隨機選取一個圖片序列，接著，會讀取該圖片序列中的每一張圖片
44        else:
45            self.cur_dir = self.dirs[np.random.randint(len(self.dirs))]
46
47        # 接著，會讀取該圖片序列中的每一張圖片，
48        # 並使用 transform 函式將其轉換成 PyTorch 張量的形式，最後，會將所有的圖片張量堆疊成一個張量序列，並回傳。
49        image_seq = []
50        for i in range(self.seq_len):
51            fname = '{}/{}.png'.format(self.cur_dir, i)
52            img = Image.open(fname)
53            image_seq.append(self.transform(img))
54        image_seq = torch.stack(image_seq)
55        return image_seq
56
57    def get_csv(self):
58        # 取 csv 檔案時，每一行的結尾都會有一個換行符號 (newline character)，這個符號在不同的作業系統中可能不同
59        with open('{}actions.csv'.format(self.cur_dir), newline='') as csvfile:
60            rows = csv.reader(csvfile)
61            actions = []
62            for i, row in enumerate(rows):
63                if i == self.seq_len:
64                    break
65                action = [float(value) for value in row]
66                actions.append(torch.tensor(action))
67
68            actions = torch.stack(actions)
69
70        with open('{}endeffector_positions.csv'.format(self.cur_dir), newline='') as csvfile:
71            rows = csv.reader(csvfile)
72            positions = []
73            for i, row in enumerate(rows):
74                if i == self.seq_len:
75                    break
76                position = [float(value) for value in row]
77                positions.append(torch.tensor(position))
78            positions = torch.stack(positions)
79
80        condition = torch.cat((actions, positions), axis=1)
81        return condition
82
83    def __getitem__(self, index):
84        self.set_seed(index)
85        seq = self.get_seq()
86        cond = self.get_csv()
87        return seq, cond

```



## F. Teacher forcing

這部分是用來更新 teacher forcing 的比率，原因是 teacher forcing 用太多會使得訓練模型對訓練資料 over fitting，造成結果不好；但若完全不用，會導致 model 在初期不夠穩定，導致無法收斂。所以用遞減率調整 Teacher forcing 的程度。使模型在訓練隨著訓練 epoch 增加，逐漸減少對 teacher forcing 的依賴，以提產生更好結果。

```
if epoch >= args.tfr_start_decay_epoch:
    ### Update teacher forcing ratio ###
    args.tfr = (args.tfr - args.tfr_decay_step) if args.tfr > 0 else 0
```

### i. Main idea

Teacher forcing 是一個類似於 RNN 的技巧，在訓練時，模型會用真實的數據作為 t-1 的輸入，而不是使用模型預測的輸出。這個方法是為了加速模型的收斂，也可以學到更好的結果。

### ii. Benefits

它可以使收斂速度變快，因為模型會用真實的數據作為 t-1 的輸入，而不是根據模型生成的輸出結果。並且減少錯誤，避免導致預測的十個時間點的誤差使後面序列整個偏掉，提高模型更精確結果。

### iii. Drawbacks

但模型可能會太依賴這些真實輸入，而不是真的學會這些數據之間的關係，導致模型的泛化能力變差。

## 4. Results and discussion

### 一、Show your results of video prediction

#### A. Make videos or gif images for test result

這邊是使用模型參數為 epoch: 30、batch size:20、learning rate: 0.002、tfr start decay epoch 75、kl anneal cyclical: True。

GIF 是在 test 時，以前 2 偵做為模型已知結果，去預測剩餘的 10 張 frames。綠色代表已知，紅色為預測。包含 approximate posterior、最好的 PSNR 結果以及隨機選任 3 個的結果。



## B. Output the prediction at each time step

以下 Ground Truth 是真實結果，Prediction 是在 test 時以前 2 個做為模型已知結果，去預測剩餘的 10 張 frames。

GT:



Pred:



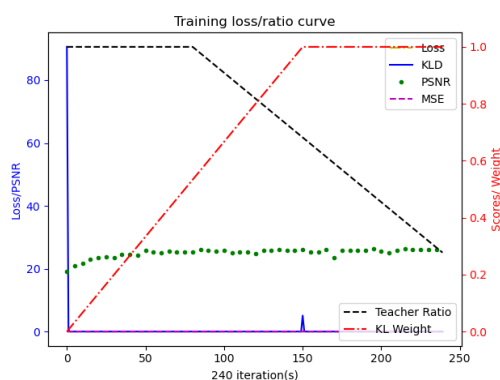
## 二、Plot the KL loss and PSNR curves during training

這邊是使用模型參數為 epoch: 30、batch size:20、learning rate: 0.002、tfr start decay epoch 75。

經過多次測試可知，當使用 Cyclical 時訓練出來的效果較佳，我認為是因為會根據周期去更新 kl annealing 調整模型，可以讓整個模型在訓練過程中去調整 KL loss 對整體的影響，使模型會以 MSE loss 的更新為主。

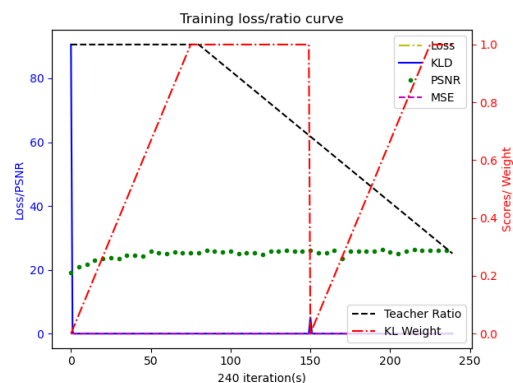
Monotonic

Avg. PSNR:24.568



Cyclical

Avg. PSNR:24.816



## 三、Discuss the results according to your setting of teacher forcing ratio, KL weight, and learning rate

### A. Teacher forcing:

經過多次測試，若一開始就讓 tfr 下降或是完全不使用 tfr，所訓練出來的 model 會很差，且在前期極度不穩定(psnr 約在 16~17 徘徊)，我認為是因為在初期模型學出來結果與真實輸入差太多，導致接下來的預測偏差會越來越大，因此在訓練初期還是先全部以真實輸入來做訓練學習(tfr = 1.0)，等到 model 穩定一點才開始降低 tfr(到 75epochs 才開始)，而這邊要下降的原因是若一直使用真實訓練資料做訓練，會倒置 overfitting(model 太依賴這些真實輸入，而不是真的學會這些數



據之間的關係)，進而讓 val 和 test 的效果變差。若一開始就降低 tfr 所獲得的 test 結果為 22.689，而我所設計的 test 結果為 24.594。

**B. kl weight:**

經過多次測試，使用 kl anneal cyclical 時的訓練結果比較好，不僅是較快收斂，最終 psnr 準確度也較高。我認為是因為其會根據周期去更新 kl weight，進而讓整個模型在訓練過程中去調整 KL loss 對整體的影響，使模型會以 MSE loss 的更新為主(decoder 為主，而非 encoder)，但也會考慮 KL loss，以此達到提高模型泛化能力的目標。

**C. Learning rate:**

經過多次測試，learning rate 對 model 的影響並不大，因此這邊就沒有更改其值，而是選用和助教一樣的 0.002