lab5 report

學號：311512049 姓名：陳緯翰

Introduction

本次的任務是使用條件變分自編碼器（conditional VAE）實現影片生成。利用已知的過去影片、動作和位置等條件信息作為輸入，通過編碼器學習數據之間的關係分布，並生成潛變量（latent vector），然後以潛變量和動作、位置等條件作為輸入，生成未來下一個時間點的影片。

Derivation of CVAE

CVAE 就是 VAE 在多給 condition 的條件下能夠較快的學習到資料的分佈 $p(x|c;\theta)$。其中 θ 是 model 需要學習的參數，利用聯合機率表示 $p(x|c;\theta)=\int p(x|z,c;\theta)p(z|c)dz$，$z$ 代表數據 x 中的 latent variable 等等會用計算取 log 變為

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

並將 train 之分佈設為一個任意分佈 $q(z|c)$ 代入

$$\begin{aligned}\log p(x|c;\theta)&=\int q(z|c)\log p(x|c;\theta)dz\\&=\int q(z|c)\log\frac{p(x,z|c;\theta)}{p(z|x,c;\theta)}dz\\&=\int q(z|c)\log\frac{p(x,z|c;\theta)}{p(z|c)}dz-\int q(z|c)\log p(z|x,c;\theta)dz\\&\quad+\int q(z|c)\log p(z|c)dz-\int q(z|c)\log p(z|c)dz\\&=L(x,q,\theta|c)+KL(q(z|c)\|p(z|x,c;\theta))\quad\cdots\ \textcircled{1}\\&=\underbrace{\int q(z|c)\log\frac{p(x,z|c;\theta)}{p(z|c)}dz}_{\text{ELBO}}+\underbrace{\int q(z|c)\log\frac{q(z|c)}{p(z|x,c;\theta)}dz}_{KL\ \text{divergence}}\end{aligned}$$

由於 $\max\ p(x|c;\theta)$，又將 KL 算為 $q(z|c)$，$p(z|x,c;\theta)$ 之分佈 ≥ 0

∴ 只需 $\max$ ELBO 即 ① 式成立

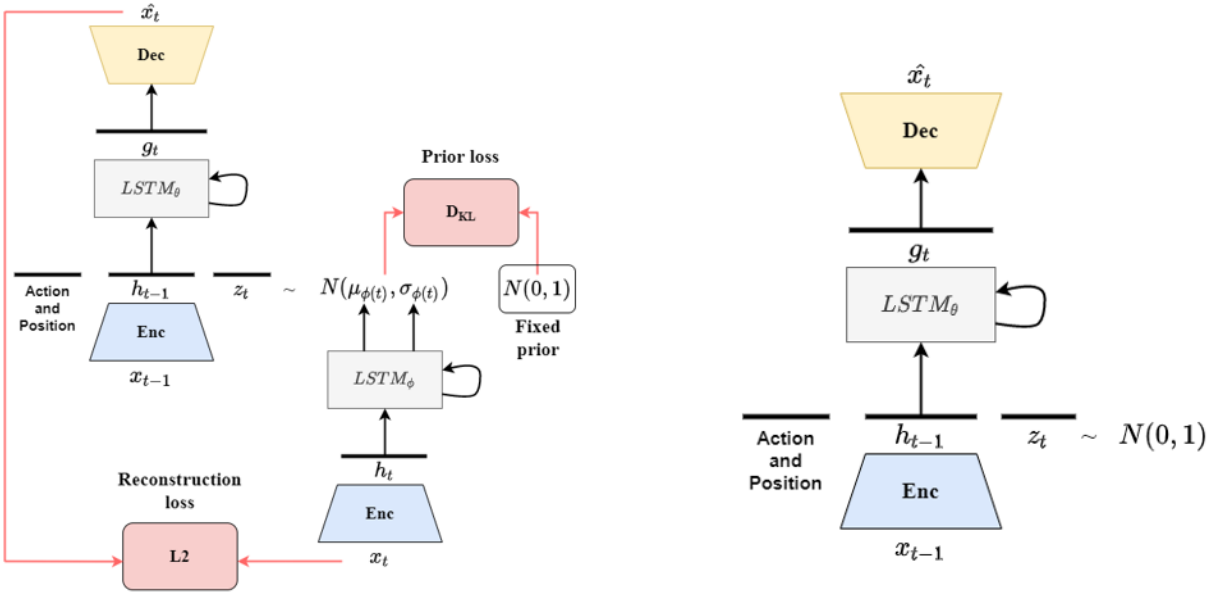
$$L(x,q,\theta|c)=\log p(x|c;\theta)-KL(q(z|c)\|p(z|x,c;\theta))$$

接著由 VAE 的 encoder 學出的分佈 $q(z|x,c,\theta')$ 代 θ' 為 encoder 參數

$$\begin{aligned}\Rightarrow L(x,q,\theta|c)&=\log p(x|c;\theta)-KL(q(z|x,c;\theta')\|p(z|x,c,\theta'))\\&=E_{z\sim q(z|x,c;\theta')}[\log p(x|z,c;\theta)+\log p(z|c)-\log q(z|x,c,\theta')]\\&=\underbrace{E_{z\sim q(z|x,c;\theta')}[\log p(x|z,c;\theta)]}_{\text{reconstruction loss}}-\underbrace{KL(q(z|x,c;\theta')\|p(z|c))}_{\text{min KL}}\end{aligned}$$

Implement details

- 根據下圖來完成整個架構及成的設計



1. Encoder

   我使用助教提供的簡化版vgg64編碼器，它由五個卷積層（c1~c5）組成，每個卷積層由不同數量的vgg_layer組成。主要目的是將輸入的圖片壓縮成較小的向量，逐漸降維並通過不同的卷積層保留不同的特徵向量。在每個卷積層中都有存儲該層向量的機制，以便使用跳躍連接（skip）來跳過一些中間層，在訓練過程中解決梯度消失問題。

```
class vgg_encoder(nn.Module):
    def __init__(self, dim):
        super(vgg_encoder, self).__init__()
        self.dim = dim
        # 64 x 64
        self.c1 = nn.Sequential(
                vgg_layer(3, 64),
                vgg_layer(64, 64),
                )
        # 32 x 32
        self.c2 = nn.Sequential(
                vgg_layer(64, 128),
                vgg_layer(128, 128),
                )
        # 16 x 16
        self.c3 = nn.Sequential(
                vgg_layer(128, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 256),
                )
        # 8 x 8
        self.c4 = nn.Sequential(
                vgg_layer(256, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 512),
                )
        # 4 x 4
```

```
        self.c5 = nn.Sequential(
                nn.Conv2d(512, dim, 4, 1, 0),
                nn.BatchNorm2d(dim),
                nn.Tanh()
                )
        self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

    def forward(self, input):
        h1 = self.c1(input) # 64 -> 32
        h2 = self.c2(self.mp(h1)) # 32 -> 16
        h3 = self.c3(self.mp(h2)) # 16 -> 8
        h4 = self.c4(self.mp(h3)) # 8 -> 4
        h5 = self.c5(self.mp(h4)) # 4 -> 1
        return h5.view(-1, self.dim), [h1, h2, h3, h4]
```

2. Decoder

   我使用助教提供的簡化版vgg64解碼器，它由五個卷積層（upc1~upc5）組成，每個卷積層由不同數量的vgg_layer組成。與編碼器不同的是，它需要逐步升維，通過不同的卷積層將向量放大到原始圖像的大小。在還原過程中，同時將對應的跳躍連接與特徵向量連接起來，利用跳躍連接技巧還原圖像，以保留更多的圖像細節。

```
class vgg_decoder(nn.Module):
    def __init__(self, dim):
        super(vgg_decoder, self).__init__()
        self.dim = dim
        # 1 x 1 -> 4 x 4
        self.upc1 = nn.Sequential(
                nn.ConvTranspose2d(dim, 512, 4, 1, 0),
                nn.BatchNorm2d(512),
                nn.LeakyReLU(0.2, inplace=True)
                )
        # 8 x 8
        self.upc2 = nn.Sequential(
                vgg_layer(512*2, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 256)
                )
        # 16 x 16
        self.upc3 = nn.Sequential(
                vgg_layer(256*2, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 128)
                )
        # 32 x 32
        self.upc4 = nn.Sequential(
                vgg_layer(128*2, 128),
                vgg_layer(128, 64)
                )
        # 64 x 64
        self.upc5 = nn.Sequential(
                vgg_layer(64*2, 64),
                nn.ConvTranspose2d(64, 3, 3, 1, 1),
                nn.Sigmoid()
                )
        self.up = nn.UpsamplingNearest2d(scale_factor=2)

    def forward(self, input):
        vec, skip = input
```

```
        d1 = self.upc1(vec.view(-1, self.dim, 1, 1)) # 1 -> 4
        up1 = self.up(d1) # 4 -> 8
        d2 = self.upc2(torch.cat([up1, skip[3]], 1)) # 8 x 8
        up2 = self.up(d2) # 8 -> 16
        d3 = self.upc3(torch.cat([up2, skip[2]], 1)) # 16 x 16
        up3 = self.up(d3) # 8 -> 32
        d4 = self.upc4(torch.cat([up3, skip[1]], 1)) # 32 x 32
        up4 = self.up(d4) # 32 -> 64
        output = self.upc5(torch.cat([up4, skip[0]], 1)) # 64 x 64
        return output
```

3. LSTM

LSTM被用於學習 encoder 的輸入特徵，並將其作為 decoder 的輸入。而高斯 LSTM 則是將潛在變量（z）與後驗分布一起使用，以生成更符合真實數據分布的樣本。

```
class lstm(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
        super(lstm, self).__init__()
        self.device = device
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.n_layers = n_layers
        self.embed = nn.Linear(input_size, hidden_size)
        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
        self.output = nn.Sequential(
                nn.Linear(hidden_size, output_size),
                nn.BatchNorm1d(output_size),
                nn.Tanh())
        self.hidden = self.init_hidden()

    def init_hidden(self):
        hidden = []
        for _ in range(self.n_layers):
            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device)),
                           Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device))))
        return hidden

    def forward(self, input):
        embedded = self.embed(input)
        h_in = embedded
        for i in range(self.n_layers):
            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
            h_in = self.hidden[i][0]

        return self.output(h_in)
```

```
class gaussian_lstm(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
        super(gaussian_lstm, self).__init__()
        self.device = device
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers
```

```
        self.batch_size = batch_size
        self.embed = nn.Linear(input_size, hidden_size)
        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
        self.mu_net = nn.Linear(hidden_size, output_size)
        self.logvar_net = nn.Linear(hidden_size, output_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        hidden = []
        for _ in range(self.n_layers):
            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device)),
                           Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device))))
        return hidden

    def reparameterize(self, mu, logvar):
        raise NotImplementedError

    def forward(self, input):
        embedded = self.embed(input)
        h_in = embedded
        for i in range(self.n_layers):
            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
            h_in = self.hidden[i][0]
        mu = self.mu_net(h_in)
        logvar = self.logvar_net(h_in)
        z = self.reparameterize(mu, logvar)
        return z, mu, logvar
```

4. Reparameterization Trick

   在 VAE 中，當 decoder 抽取潛在變量 z 作為輸入時，利用重參數技巧將分布視為連續高斯分布，以更好地計算梯度。首先將 log-variance 轉換為 sigma，然後從高斯分布中抽樣一個變量，再乘以 encoder 生成的分布的 sigma，最後加上 mu。這個函數接受編碼器的輸出mu和logvar作為參數，計算標準差std並從標準正態分佈中生成一個隨機噪聲eps，最終返回重新參數化後的樣本z。

```
def reparameterize(self, mu, logvar):
        # 將mu, logvar轉換成標準常態分佈
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std
```

5. Data loader

   資料加載器 (Dataloader) 部分跟之前作業類似，都是從 train / validate / test 資料夾中讀取所有序列 (sequence) 並儲存在 dirs 變數中，因此可以直接利用 dirs 的長度作為 **len** 的回傳值。接著，逐一讀取各個序列中的影格 (frame)，從第 0 個開始依序取出，再將影像轉換為 [C, H, W] 的格式，最後將其 reshape，以方便之後的處理。接著，取出各個序列中的條件資訊 (condition)，並將其連接成 7 個元素的形式，最後再依序取出序列及其條件資訊。

```
class bair_robot_pushing_dataset(Dataset):
    def __init__(self, args, mode='train', transform=default_transform):
        assert mode == 'train' or mode == 'test' or mode == 'validate'
        self.root = '{}/{}'.format(args.data_root, mode)
        self.seq_len = max(args.n_past + args.n_future, args.n_eval)
```

```
        self.mode = mode
        if mode == 'train':
            self.ordered = False
        else:
            self.ordered = True

        self.transform = transform
        self.dirs = []
        for dir1 in os.listdir(self.root):
            for dir2 in os.listdir(os.path.join(self.root, dir1)):
                self.dirs.append(os.path.join(self.root, dir1, dir2))

        self.seed_is_set = False
        self.idx = 0
        self.cur_dir = self.dirs[0]
        self.d = 0

    def set_seed(self, seed):
        if not self.seed_is_set:
            self.seed_is_set = True
            np.random.seed(seed)

    def __len__(self):
        return len(self.dirs)

    def get_seq(self):
        if self.ordered:
            self.cur_dir = self.dirs[self.d]
            if self.idx == len(self.dirs) - 1:
                self.idx = 0
            else:
                self.idx += 1
        else:
            self.cur_dir = self.dirs[np.random.randint(len(self.dirs))]

        image_seq = []
        for i in range(self.seq_len):
            fname = '{}/{}.png'.format(self.cur_dir, i)
            img = Image.open(fname)
            image_seq.append(self.transform(img))
        image_seq = torch.stack(image_seq)

        return image_seq

    def get_csv(self):
        with open('{}/actions.csv'.format(self.cur_dir), newline='') as csvfile:
            rows = csv.reader(csvfile)
            actions = []
            for i, row in enumerate(rows):
                if i == self.seq_len:
                    break
                action = [float(value) for value in row]
                actions.append(torch.tensor(action))

            actions = torch.stack(actions)

        with open('{}/endeffector_positions.csv'.format(self.cur_dir), newline='') as csvfile:
            rows = csv.reader(csvfile)
            positions = []
            for i, row in enumerate(rows):
                if i == self.seq_len:
                    break
                position = [float(value) for value in row]
```

```
                positions.append(torch.tensor(position))
            positions = torch.stack(positions)

        condition = torch.cat((actions, positions), axis=1)

        return condition

    def __getitem__(self, index):
        self.set_seed(index)
        seq = self.get_seq()  # (seq_len, 3, 64, 64)
        cond =  self.get_csv() # (seq_len, 4)
        return seq, cond
```

6. Describe the teacher forcing

   Teacher Forcing是一種快速有效地訓練循環神經網絡模型的方法，該模型使用來自先驗時間步長的輸出作為輸入，並且在訓練的時候使用標準答案（即ground truth）作為上一個時間步的輸入，而不是使用上一個時間步的輸出作為下一個時間步的輸入。

   A. 主要觀念
   Teacher forcing是一種類似於RNN的技巧，在訓練模型時，使用真實數據作為t-1的輸入，而不是使用模型預測的輸出。這個技巧的目的是加速模型的收斂，並且可以獲得更好的結果。

   B. 優點
   Teacher forcing可以使模型收斂速度更快，因為模型使用真實數據作為t-1的輸入，而不是依據模型生成的輸出。這樣可以減少錯誤，避免預測一個時間點的誤差導致整個序列偏掉，提高模型的準確性。

   C. 缺點
   然而，使用Teacher forcing可能會讓模型過度依賴真實輸入，而沒有真正學習數據之間的關係，進而降低模型的泛化能力。

   而在訓練初期使用teacher forcing(有就是使用ground truth)可以使model 較容易收斂，但如果使用太多的 teacher forcing 會使得模型過於依賴已知的答案，導致產生的結果不佳，所以我在中間漸漸將teacher forcing的rate降低去更多的使用predict出來的當作輸入來增加整個model的強健性。

```
if epoch >= args.tfr_start_decay_epoch:
    # TODO Update teacher forcing ratio
    args.tfr -= args.tfr_decay_step
    if args.tfr <= args.tfr_lower_bound:
        args.tfr = 0
```

Results and discussion

Make videos or gif images for test result

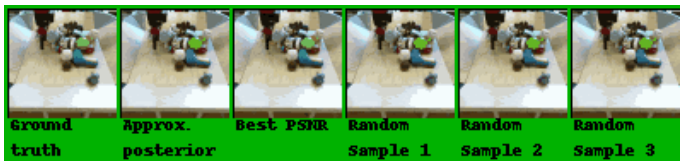
使用的模型參數為
epoch: 300、 batch size:20、 learning rate: 0.002、 tfr start decay epoch 100、 kl anneal cyclical: True。

GIF是在 test時用前2個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，及隨機選任 3個的結果。

Output the prediction at each time step

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。
Ground Truth：

Prediction：

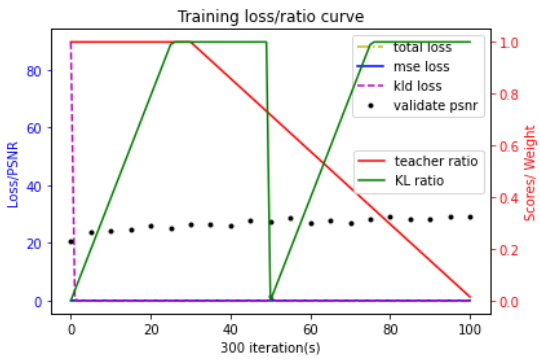


Plot the KL loss and PSNR curves

1. KL weight：
   epoch: 100、 batch size:20、 learning rate: 0.002、 tfr start decay epoch 30。
   當設置 kl anneal cyclical 為 True 時，訓練結果往往更好且更穩定，這是因為模型會根據週期進行 kl annealing 的更新，從而調整 KL 損失對整個模型的影響。這樣做可以讓模型在訓練過程中主要基於 MSE 損失進行更新，同時也考慮 KL 損失以提高模型的泛化能力。

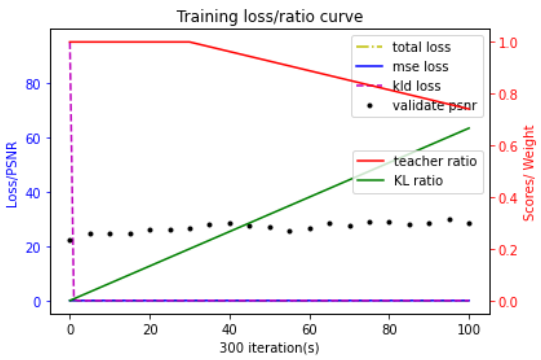
- Cyclical

  best_val_psnr: 29.347229729044187



- Monotonic mode

  best_val_psnr: 28.958629075549116



2. Teacher forcing：

   在訓練模型時，一開始利用teacher forcing的時候，我沒有將tfr start decay epoch設為0的原因是，如果模型在一開始學習時與真實輸入差距太大，接下來的預測結果會越來越偏差。為了防止這種情況發生，我選擇在訓練的前30個epochs中保持使用teacher forcing，之後再逐漸降低使用teacher forcing的比例。我之所以選擇這樣的遞減策略，是為了避免模型過度依賴真實輸入而無法真正學習數據之間的關係，這可能會導致模型在其他類似但未見過的情況下表現不佳。

3. Learning rate：

   調整learning rate並沒有差距太多因此設為跟助教預設一樣的0.002