DLP HW5

數據所 311554019 宋沛潔

1. Introduction:

這次的作業是使用 conditional VAE 來實作影片生成的任務。根據已知過去的影片及動作和位置(條件)，利用 encoder 學出來資料之間的關係分布，再利用 encoder 輸出、latent vector 以及動作和位置（條件）做為 input ，去生成未來下一個時間點的影片。

2. Derivation of CVAE:

對於 CVAE 來說，目標是要去學給定數據 X 在條件 (c) 下的分佈空間可以寫成 $p(x|c;\theta)$，希望此分佈愈大愈好，θ 是模型要學習的參數，利用聯合機率表示：$p(x|c;\theta)=\int p(x|z,c;\theta)p(z|c)dz$，其中 z 代表數據 X 之間的潛在關係變量（latent variables）

為了方便計算取 log 變為：

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

並將要學習的分佈設為一個任意分佈 $q(z|c)$ 代入

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

由於最大化 $p(x|c;\theta)$，又用 KL 算 $q(z|c)$、$p(z|x,c;\theta)$ 之分佈一定 ≥ 0，

∴ 只須最大化 ELBO，將①改寫：

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

再由 VAE 的 encoder 學出來的分佈 $q(z|x,c,\theta')$ 代入、θ' 為 Encoder 參數

$$\begin{aligned}L(x,q,\theta|c) &= \log p(x|c;\theta)-KL(q(z|x,c;\theta')\,\|\,p(z|x,c;\theta')) \\ &= E_{z\sim q(z|x,c;\theta')}[\log p(x|z,c;\theta)+\log p(z|c)-\log q(z|x,c;\theta')] \\ &= \underbrace{E_{z\sim q(z|x,c;\theta')}[\log p(x|z,c;\theta)]}_{\text{最大化 reconstruction loss}} - \underbrace{KL(q(z|x,c;\theta')\,\|\,p(z|c))}_{\text{最小化 2分佈的 KL}}\end{aligned}$$

3. Implementation details

1. Encoder

使用助教提供的簡化版的 vgg64 encoder，包含了五個 convolution layer(c1~c5)，由不同數量的 vgg_layer 組成。主要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。

```
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]
```

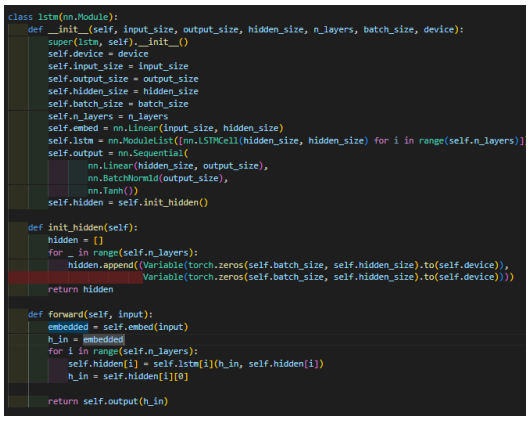
2. Decoder

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。

```python
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

   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。

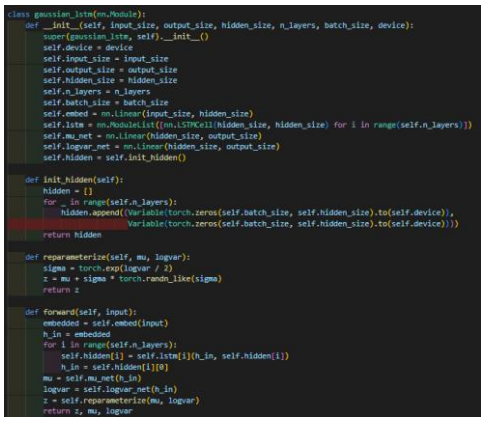


```python
class lstm(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
        super(lstm, self).__init__()
        self.device = device
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size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



```python
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):
        sigma = torch.exp(logvar / 2)
        z = mu + sigma * torch.randn_like(sigma)
        return z

    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

   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 將分布視為連續的高斯分布，可以更好計算梯度。先將 log-variance 轉成 sigma，並從高斯分布抽樣出一個變數與 encoder 生成的分布(sigma、mu) sigma 相乘並加 mu。

```python
def reparameterize(self, mu, logvar):
    sigma = torch.exp(logvar / 2)
    z = mu + sigma * torch.randn_like(sigma)
    return z
```

5. Data loader

   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。

```python
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 = args.n_past + args.n_future
        # if  mode == "test":
        #     self.seq_len = 30
        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]

    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.idx]
            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
```

```python
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()
    cond = self.get_csv()
    return seq, cond
```

6. 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硬記，造成結果不好。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:
    decay_epochs = (args.niter - args.tfr_start_decay_epoch)
    tfr_decay_rate = (1.0 - args.tfr_lower_bound) / decay_epochs
    upper_bound = max(args.tfr_lower_bound, 1.0 - (epoch - args.tfr_start_decay_epoch) * tfr_decay_rate)
    args.tfr = min(1.0, upper_bound)
```

A. 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。

B. 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，導致預測十一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。

C. Drawbacks

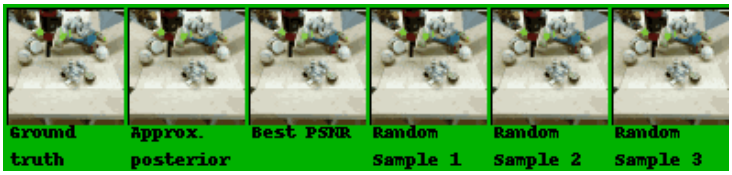
但模型可能會太依賴這些真實輸入，而不是真的學會這些數據之間的關係，導致模型的泛化能力變差。

4. Results and discussion

A. 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，以前 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。



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。

Ground Truth:

Prediction:

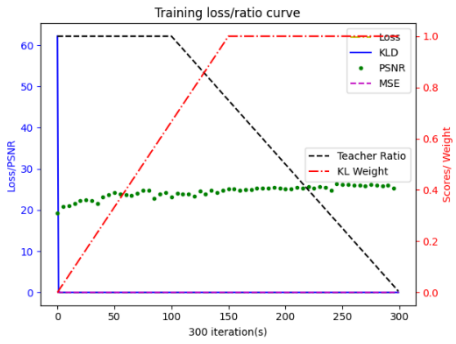
C. Plot the KL loss and PSNR curves

1. kl anneal cyclical:

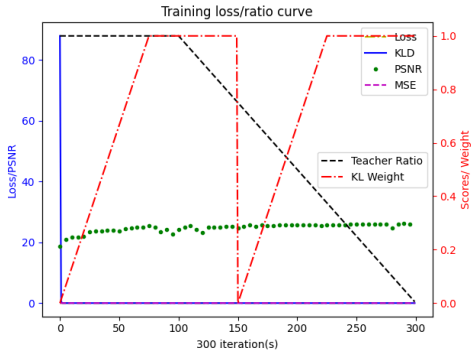
   使用的模型參數為 epoch: 300、batch size:20、learning rate: 0.002、tfr start decay epoch 100。

   當 kl anneal cyclical 設為 True 的訓練結果比較好，也較穩定，那是因為在 kl anneal cyclical 設為 True 代表會根據周期去更新 kl annealing 調整模型。這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，但也可以考慮 KL loss，以提高模型的泛化能力。

   A. kl anneal cyclical: False
   Avg. PSNR: 24.618

   B. kl anneal cyclical: True
   Avg. PSNR:24.855

2. Teacher forcing:

   在訓練模型時，直接在一開始利用 teacher forcing 時的 tfr start decay epoch 不設為 0，原因是模型學出來與真實輸入差太多，接下來的預測偏差會越來越大，因此我從 100 epochs 才開始遞減，遞減原因是防止模型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。則預測結果 test 有達 24.855，但是如果將 tfr start decay epoch 設較低則 test 結果為 23.~ 左右。

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