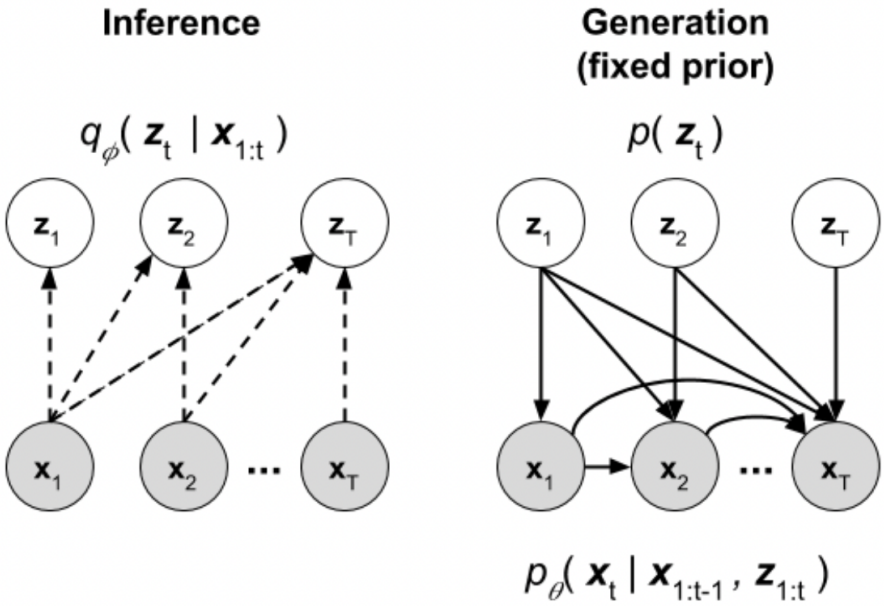
Stochastic video generation

1. Introduction

這次的 LAB 不同於以往的使用 CVAE 單純從一個高斯分佈當中 sample 出一個 latent vector 去產生圖片，而是希望透先給一段影像的前兩張圖片之後，CVAE 模型可以預測接下來會產生的十張圖片，所以整體的架構會點類似 Recursive Neural Network，我們在產生下一張照片 X_{data} 的同時，也會需要前一個時刻資訊讓我們接下來產生的照片能夠符合 X_{data} 的真實分佈。



再把 Decoder 所產生的十張照片整合起來成 GIF 或者是影像檔案，看起來就會像是一段影像的預測或者說是一段影像的產生。

大致上會如下圖所示：

Figure 2: Prediction at each time step

2. Derivation of CVAE

$$p(x, z|c) = p(x|c)\, p(z|x, c)$$

$$\Rightarrow \log p(x, z|c) = \log p(x|c) + \log p(z|x, c)$$

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

左右同乘 $q(z)$ 并积分

$$\int q(z) \log p(x|c)\, dz = \int q(z) \log p(x, z|c)\, dz - \int q(z) \log p(z|x, c)\, dz$$

$$\Rightarrow \log p(x|c) = \int q(z) \log p(x, z|c)\, dz - \underbrace{\int q(z) \log q(z)\, dz}_{\text{extra}}$$

$$+ \underbrace{\int q(z) \log q(z)\, dz}_{\text{extra}} - \int q(z) \log p(z|x, c)\, dz$$

$$\Rightarrow \log p(x|c) = \mathcal{L}(x, q|c) + KL\left(q(z|x, c) \,\|\, p(z|x, c)\right)$$

此处 q 可以是任意、特别好的

$$\mathcal{L}(x, q|z) = \underbrace{\int q(z|x, c) \log p(x, z|c)\, dz}_{\downarrow} - \int q(z|x, c) \log q(z)\, dz$$

$$\text{☆ } p(x, z|c) = p(x|z, c) \cdot p(z|c)$$

$$E_{z \sim q(z|x,c)}\left[\log p(x, z|c)\right] - E_{z \sim q(z|x,c)}\left[\log q(z|x, c)\right]$$

$$\Rightarrow E_{z \sim q(z|x,c;\theta)}\left[\log p(x|z, c; \theta)\right] + \underbrace{E_{z \sim q(z|x,c;\theta')}\left[\log p(z|c) - \log q(z|x, c)\right]}_{\| \; -KL\left(q(z|x,c;\theta') \,\|\, p(z|c)\right)}$$

3. Implementation details

Dataloader

每一個資料夾都代表是一段影片，影片由 30 張圖片組成，而我們只需要用前 2 張照片來預測後面 10 張照片，所以我們只需要將資料夾裡前 12 張照片讀進來，每張照片大小為[3, 64, 64] 讀完 12 張之後維度會變成 [12, 3, 64, 64] 這邊要記得先用 To_Tensor 將圖片轉成 Tensor 的資料型態。

```
def get_seq(self):
    if self.ordered:
        d = self.dirs[self.d]
        if self.d == len(self.dirs) - 1:
            self.d = 0
        else:
            self.d += 1
    else:
        d = self.dirs[np.random.randint(len(self.dirs))]
    image_seq = []
    for i in range(self.seq_len):
        fname = '%s/%d.png' % (d, i)
        img = Image.open(fname)
        img = self.img_Augmentation(img)
        image_seq.append(img)
    # [(3, 64, 64), (3, 64, 64), ..., (3, 64, 64)] -> (12, 3, 64, 64)
    image_seq = torch.stack(image_seq)
    return image_seq
```

再來要將 condition 也讀進來，裡面包含位址跟移動的一些資訊，這邊會直接將整份 CSV 資料讀進來，這邊也記得要將 nprray 的資料型態轉成 Tensor，但我們只需要將 CSV 裡的前 12 row 傳回去主程式就可以了，維度會是 [30, 7] → [12, 7]。

```python
def get_csv(self):
    d = self.dirs[self.d]
    conds = []
    files = ["%s/%s" % (d, "actions.csv"), \
             "%s/%s" % (d, "endeffector_positions.csv")]
    for csv_file in files:
        with open(csv_file) as f_csv:
            rows = csv.reader(f_csv)
            cond_arr = np.array(list(rows)).astype(float)
            cond_tensor = self.csv_Augmentation(cond_arr)
        conds.append(cond_tensor)
    # [(30, 3), (30, 4)] -> (30 ,7)
    conds = torch.cat(conds, dim=2)[0]
    conds = conds[:self.seq_len,:].float()
    return conds
```

Encoder

首先我們會將大小為 [seq_len, epoch_size, 3, 64, 64] (seq_len = n_past + n_future) 的一組資料送進去我們的 encoder，通過五層的轉換之後將最後一層的輸出當成我們的 h 向量（LSTM 的輸入），encoder 剩餘前四層輸出會進一步送進去 decoder 當成 decoder 的輸入。

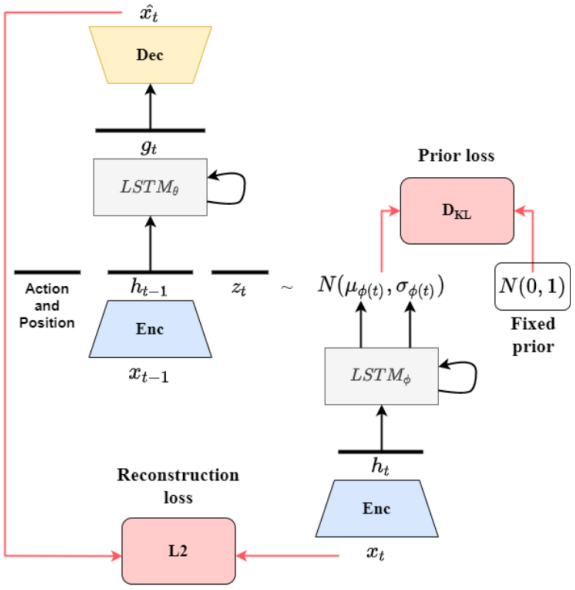
```python
def forward(self, input):
    h1 = self.c1(input) # 64 -> 32
    h2 = self.c2(self.mp(h1)) # 32 -> 16
    h3 = self.c3(self.mp(h2)) # 16 -> 8
    h4 = self.c4(self.mp(h3)) # 8 -> 4
    h5 = self.c5(self.mp(h4)) # 4 -> 1
    return h5.view(-1, self.dim), [h1, h2, h3, h4]
```

Decoder

經過 decoder 之後會輸出跟原始輸入一樣大小的矩陣[seq_len, epoch_size, 3, 64, 64]，並且透過 MSE Loss 去計算跟原始輸入的差，此項就稱為 Reconstruction term，透過此 MSE Loss 就可以去執行 Backpropagation 來跟新 decoder 以及 encoder 的網路參數。

```python
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
```

Training Procedure

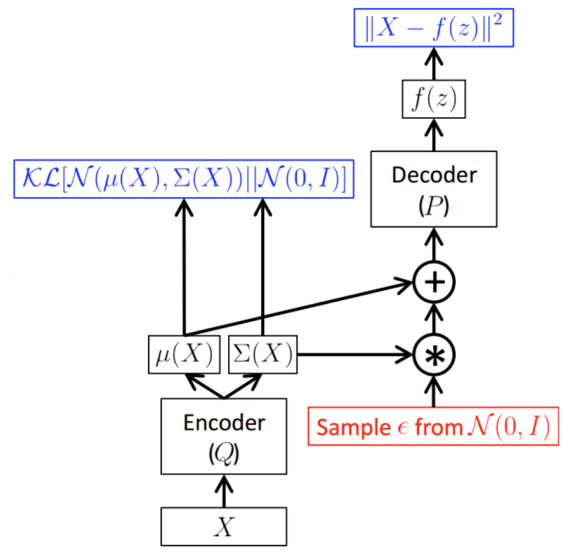


上圖為此次任務整體的 Training 架構，可以直接搭配下方的程式碼進行對比，首先在第 t-1 個時間點時，我們會將 X_{t-1} 跟 X_t 送進去 encoder 當中產生 h_{t-1} 以及 h_t，並將 h_t 送進去 LSTM posterior 產生高斯分佈的兩個參數 mu 跟 logvar，然後再從此分佈當中 sample 出一個 Z_t 與剛剛產生的 h_{t-1} 串接在一起當成 LSTM frame_predictor 當中產生出 g_t，最後在與 encoder 產生的 skip 一起丟進去 decoder 產生我們最後的 X_{pred}，拿 X_{pred} 與我們原始的 X_{t-1} 去計算 MSE Loss（前面提及的 Reconstruction term）。

但目前我們會衍生出一個問題，你剛剛說的 sample 什麼意思？因為在 VAE 這個模型當中我們會希望 X_{data} 是從一個非常簡單的分佈所產生的，這當然可以根據你自己喜愛去設定，但通常我們會假設成是一個 N(0, I)的高斯分佈，所以我們會希望讓

LSTM posterior 所產生的高斯分佈跟 N(0, I)是 compatible 的，可以當成是某種程度的近似，所以這時候我們會去計算兩個分佈之間的 KL-divergence 也就是所謂的 Regularization term。

好了，目前我們已經知道這邊大致上的概念是什麼了，但該怎麼跟新網路的參數呢？剛剛透過 sample 的方式產生 decoder 的輸入顯然是沒辦法做 Backpropagation，這時候我們就會需要透過 Re-parameterize trick，將 sample 出 Z_t 的方式改變成 Z_t = logvar * epsilon + mu , note: epsilon ~ N(0, I)，這樣的好處在於我們將 sample Z_t 變成是網路輸出的可微分函式，這樣就可以透過 Backpropagation 將 decoder 這邊的 loss 傳遞回去以跟新 encoder 的網路參數，因為我們不能夠將兩個網路分別獨立來進行訓練，下面的 objective function 可以看出，我們需要同時考慮兩個網路的 Reconstruction term 以及 Regularization term 才能夠進行 end-to-end 的 Training 過程。



$$\underbrace{E_{\boldsymbol{Z}\sim q(\boldsymbol{Z}|\boldsymbol{X};\boldsymbol{\theta}')} \log p(\boldsymbol{X}|\boldsymbol{Z};\boldsymbol{\theta})}_{\text{Re-parameterization for end-to-end training}} - \mathrm{KL}(q(\boldsymbol{Z}|\boldsymbol{X};\boldsymbol{\theta}')||p(\boldsymbol{Z}))$$

```
h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
mse = 0
kld = 0
use_teacher_forcing = True if random.random() < args.tfr else False
for i in range(1, args.n_past + args.n_future):
    h_target, _ = h_seq[i]
    if args.last_frame_skip or (i < args.n_past):
        h, skip = h_seq[i-1]
    else:
        if use_teacher_forcing:
            h, _ = h_seq[i-1]
        else:
            h, _ = modules['encoder'](x_pred)

    z_t, mu, logvar = modules['posterior'](h_target)
    g_t = modules['frame_predictor'](torch.cat([h, z_t, cond[i-1]], 1))
    x_pred = modules['decoder']([g_t, skip])

    mse = mse + mse_criterion(x[i], x_pred)
    kld = kld + kl_criterion(mu, logvar, args)

loss = mse + kld * kl_anneal.get_beta()
loss.backward()

optimizer.step()
```

Teacher Forcing technique

這個技巧是在 Recursive Neural Network 當中很常使用的到一個技巧，最原始的網路架構在產生第 t 個時間點的輸出時，只會知道網路本身在前一刻時間點 t-1 所產生的輸出，並拿此拿來當成第 t 個時間點的輸入，藉此就能夠循序地往下產生輸出，但原始的這個方法可能會讓模型學到非常不理的特徵，舉例來說，在訓練時如果第 t 個時間點模型所產生的輸出已經是錯誤的結果，這時如果再拿此輸出來當成第 t+1 個時間點的輸入，有可能會讓模型"一步錯、步步錯"。

這時候我們就會需要 Teacher Forcing 的幫助，在訓練時讓網路能夠拿到 Ground Truth 當成模型的輸入，這樣做能夠確保讓網路模型的輸入至少不會太差，而且當第 t 個時間點的網路輸出開始越差越多時，Teacher Forcing 也能夠確保及時糾正，不要讓太差的預測結果又當成下一個時間點 t+1 的網路模型輸入，保證網路模型在訓練時不會導致一步錯、步步錯的結果。

但同時 Teacher Forcing 也有缺點，一直靠老師帶的孩子是走不遠的。

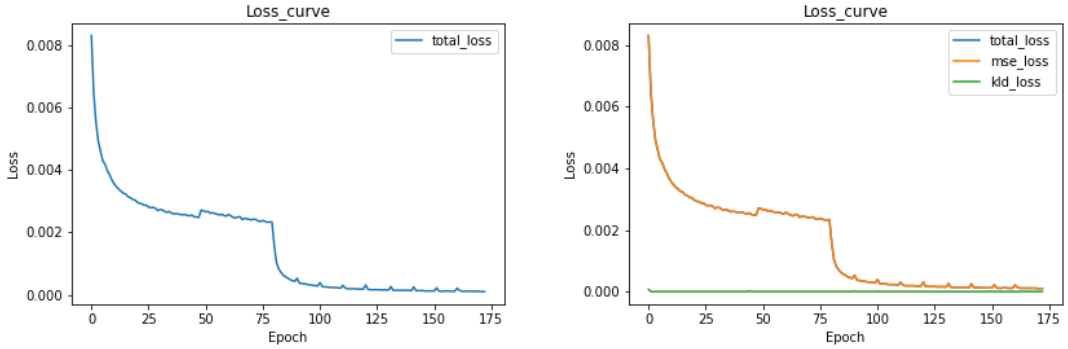
因為依賴標籤數據，在訓練過程中，模型會有較好的效果，但是在測試的時候因為不能得到 ground truth 的支持，所以如果目前生成的序列在訓練過程中有很大不同，

模型就會變得脆弱。也就是說，使用 Teacher Forcing 這種機制的 cross-domain 能力會更差，因為模型在訓練時很少依賴自己產生的輸出結果，如果測試數據集與訓練數據集來自不同的領域，模型的 performance 就會變差。所以如何在使用 Teacher Forcing 以及不使用之間取得一個平衡，也是非常值得探討以及實驗的一部份。

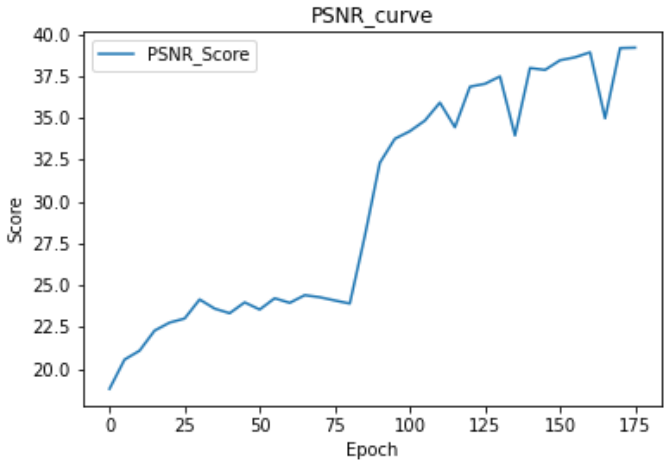
4. Result and discussion

Plot Loss curve

這邊只有將前 175 輪 epoch 的 loss curve 畫出來，再到後面基本上已經趨近於 0 ，畫出來只是一條直線，由於 kld_loss 基本上都很小，所以在右圖看起來會只有兩條 curve，但事實上有三條，所以我特地再畫一張只有 total_loss 的 curve 如左圖。



Plot PSNR curve



Predict Image Sequence

因 pdf 無法顯示 GIF 動畫，所以預測出的 GIF 圖檔會放在資料夾裡。

我選擇了最佳的 psnr 分數的 sequence 來做成 GIF 圖檔如下所示：

下圖則是兩張圖片去預測出的後十張圖片所組成之序列。

→----→----Time series----→----→

首先，我所使用的 Teacher forcing 策略是前 150 輪 epoch 時，都會讓網路模型吃 Ground Truth，以便模型參數可以先收斂至一定程度，後面 150 輪 epoch 則會讓 Teacher forcing rate 慢慢下降，到了最後 50 輪 epoch 時會讓 Teacher forcing rate 歸零，因此此時模型在訓練時只會看到自己前一時刻 X_{pred} 所產生的結果。
我發現如果前期 Teacher forcing 如果 epoch 不夠多，模型參數很可能無法收斂，像我有一次只設定了前面 50 輪 epoch 會開啟 Teacher forcing，但會導致模型此時的參數還不夠收斂，因次模型沒辦法很好的從自己前一個時間點產生的輸出 X_{pred} 當中學到好的特徵並加以收斂，這時候的 psnr 可能會停滯不前，甚至有可能往下掉。

而我所使用的 KL-annealing 方法有兩種策略，一個是在同一輪 epoch 當中，beta 會有三次的起伏循環，但同時我也發現不能夠讓 beta 的值太大，如果調太大會讓網路輸出的 loss 過大，在 Backpropagation 時的 Gradient 會使網路參數變化太大，沒辦法很好的往 Global minimum 進行收斂，另一個策略則是單調遞增，隨著訓練的 epoch 增加，beta 會緩慢地進行遞增，對這個策略同時也必須限制好 beta 的上限，避免 loss 太大導致 Gradient descent 沒辦法很好地讓模型參數進行收斂。