DLP
LAB 4

謝侑哲

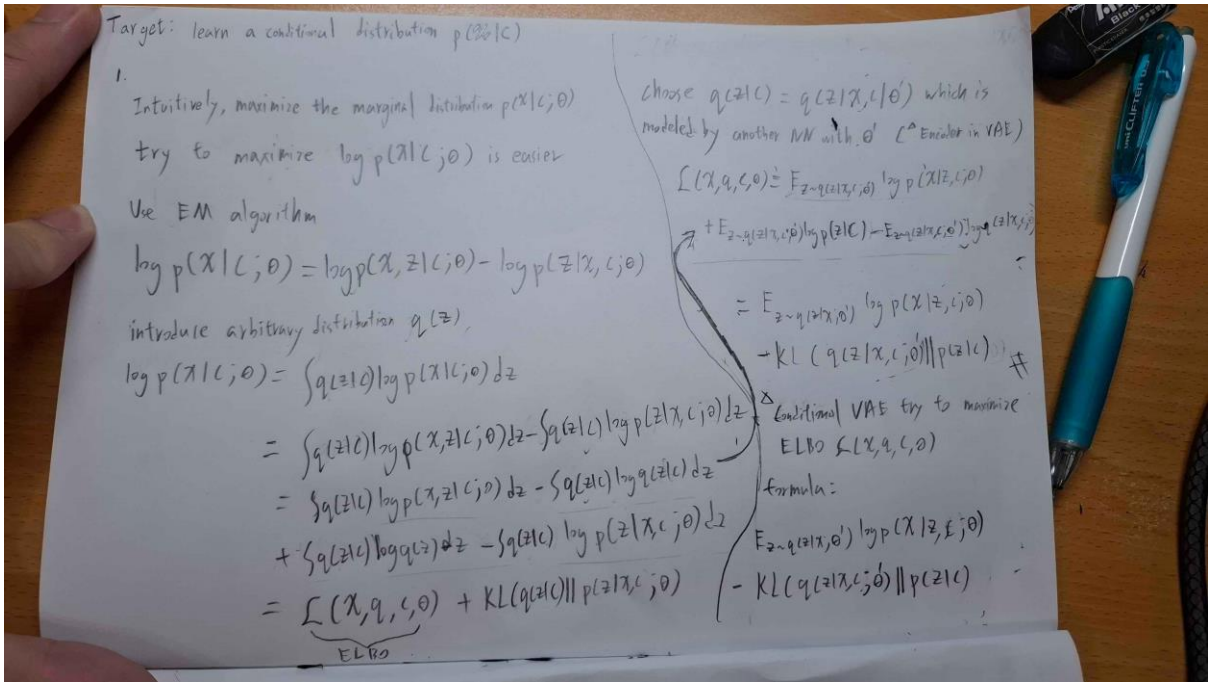
112550069

Github:

https://github.com/youzhe0305/NYCU-DLP

1. Derivate conditional VAE formula

以下是 conditional VAE 的推導，從嘗試最大化 p(X|c)開始，利用 EM algorithm 算出 lower bound (ELBO)，再往後推出 ELBO 的在 VAE 形式的式子



2. Introduction

本次 lab 為，使用 conditional VAE 來做影像生成，使用包含影片及人體骨架的資料集。最終用 1 張影片幀與 630 張骨架圖，生成以第一張圖作為基礎的整個影片。其中採用了許多做法來增強 cVAE，諸如: teacher forcing, kl annealing 等等。試圖增加 psnr 分數。

3. Implementation Details

A. How do you write your training/testing protocol

(1) Training Code

在整體的訓練中，因為助教已經幫忙寫好訓練的迴圈，因此我只實作了訓練一個 batch 的方法。

cVAE 的實作上，很重要的點在於找到對的 condition，在沒有 teacher forcing 的情況下，會先取整的影像序列的第 1 幀作為參考，也將對應要預測的骨架圖做為 condition，之後利用 frame encoder 與 label encoder 分別提取參考幀與骨架圖的特徵。

接著，基於第一小節推導出的 cVAE 的公式，要推測 p(z|x,c;theta)，因此使用了一個 gaussian predicter，以上面提到的 condition 與原圖(ground truth)作為輸入，產出圖片與條件的 representation，作為 cVAE 的 Encoder 的部分。

得到 representation 之後，試著利用參考圖、condition，去把 representation 的精簡特徵，去生成出跟原圖(ground truth)盡可能相似的圖片。將 representation 與參考圖、condition 餵進 decoder fusion，提取出其混和、相關聯後的特徵，再將該特徵餵給 generator，用來產生接近原圖的圖。

最後利用得到的預測，以及 representation 的 mean 與 variance，可以分別算出 mse 下與原圖的差距，以及 representation 分布與 standard gaussian distribution 的 KL-Divergence。

```
training_one_step(self, img, label, adapt_TeacherForcing, time_smoothing_rate = 0.3): # return loss
# img, label shape: (batch_size=2, video_frames=16, 3, frams_H=32, frams_W=64)

pred_img = img[:,0,:,:] # predict image, for next frame's reference
video_loss = torch.zeros(1).to(self.device)
beta = self.kl_annealing.get_beta() # beta for this epoch
for idx in range(1,self.train_vi_len): # start predict from img 1(2nd frame)

    if adapt_TeacherForcing == True:# previous image for reference to predict, when teacher forcing, u
        ref_img = self.mix_teacher_forcing(img[:,idx-1,:,:,:], pred_img, self.tfr) # use the mix of gt
    else:
        ref_img = pred_img # previous image for reference to predict

    transformed_ref_img = self.frame_transformation(ref_img) # frame encoder
    transformed_label = self.label_transformation(label[:,idx,:,:,:])
    transformed_gt_img = self.frame_transformation(img[:,idx,:,:,:]) # current image for predict gauss

    z, mu, logvar = self.Gaussian_Predictor(transformed_gt_img, transformed_label)
    fused_features = self.Decoder_Fusion(transformed_ref_img, transformed_label, z)
    prediction = torch.sigmoid(self.Generator(fused_features))
    pred_img = prediction # for next frame reference
    # MSE + KL(N(mean, var) | N(0,1))

    time_smoothing_loss = self.mse_criterion(prediction-ref_img, img[:,idx,:,:,:]-img[:,idx-1,:,:,:])
    video_loss += self.mse_criterion(prediction, img[:,idx,:,:,:]) + beta * kl_criterion(mu, logvar, s
                + time_smoothing_rate * time_smoothing_loss
```

```
avg_loss = video_loss / (self.train_vi_len - 1)
avg_loss.backward()
self.optimizer_step()
self.optim.zero_grad()

return avg_loss
```

(2) Validation Code

同樣因為助教已經幫忙寫好 validation 的迴圈，所以這裡指針對跑一次的 val_one_step 說明。

在實作上，跟 training code 差不多，把影片的生產長度改成 val_vi_len，在本次 lab 中就是 630 張圖片的影片，另外因為 validation 只取 cVAE 的 decoder 部分，因此 latent code 是由 randn 隨機產生的服從於 normal distribution 的 z，而非 gaussian generator 產生的，對於影片的參照，也是完全參照預測的上一個 frame，而沒有使用 groud truth。

```
def val_one_step(self, img, label):
    # TODO
    ref_img = img[:,0,:,:,:] # predict image, for next frame's reference
    video_loss = torch.zeros(1).to(self.device)
    video_psnr = torch.zeros(1).to(self.device)
    psnr_record = []
    frame_idx_record = []
    for idx in range(1,self.val_vi_len): # start predict from img 1(2nd frame)
        frame_idx_record.append(idx+1)
        transformed_ref_img = self.frame_transformation(ref_img) # frame encoder
        transformed_label = self.label_transformation(label[:,idx,:,:,:])

        z = torch.randn(size=(1, self.args.N_dim, self.args.frame_H, self.args.frame_W)).to(self.device)
        fused_features = self.Decoder_Fusion(transformed_ref_img, transformed_label, z)
        prediction = torch.sigmoid(self.Generator(fused_features))
        ref_img = prediction # previous image for reference to predict

        # MSE, no need to consider KL divergence, simply sample from std gaussian distirbution
        video_loss += self.mse_criterion(prediction, img[:,idx,:,:,:])
        psnr = Generate_PSNR(img[:,idx,:,:,:], prediction)
        video_psnr += psnr
        psnr_record.append(psnr.item())

    validation_psnr_plot((frame_idx_record, psnr_record), 'validation_psnr.jpg', 'validation_psnr')
    avg_loss = video_loss / (self.val_vi_len - 1)
    avg_psnr = video_psnr / (self.val_vi_len - 1)

    return avg_loss, avg_psnr
```

(3) Test Code

我們自己實作的部分，基本上跟 validation 一樣，只是換了個維度，這裡就不多加贅述。

```python
def val_one_step(self, img, label, idx=0):

    # TODO

    img = img.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    label = label.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    assert label.shape[0] == 630, "Testing pose seqence should be 630"
    assert img.shape[0] == 1, "Testing video seqence should be 1"

    # decoded_frame_list is used to store the predicted frame seq
    # label_list is used to store the label seq
    # Both list will be used to make gif
    decoded_frame_list = [img[0]]
    label_list = []

    for i in range(1, self.val_vi_len):

        ref_img = decoded_frame_list[i-1].to(self.args.device)
        transformed_ref_img = self.frame_transformation(ref_img) # frame encoder
        transformed_label = self.label_transformation(label[i,:,:,:])

        z = torch.randn(size=(1, self.args.N_dim, self.args.frame_H, self.args.frame_W)).to(self.args.dev
        fused_features = self.Decoder_Fusion(transformed_ref_img, transformed_label, z)
        prediction = torch.sigmoid(self.Generator(fused_features))

        decoded_frame_list.append(prediction)
        label_list.append(label[i,:,:,:])
```

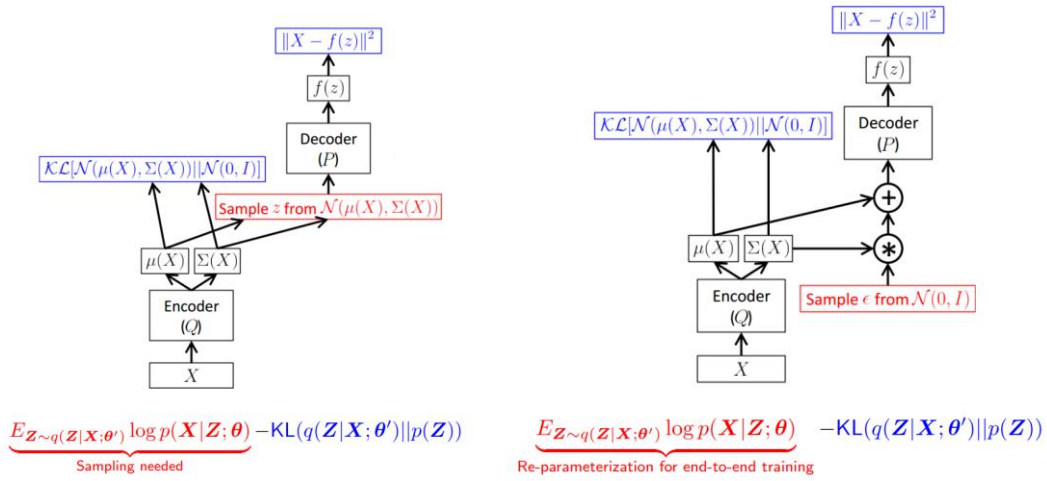
額外的部分就是助教提供的，把資料片成 GIF 的程式

```python
    # Please do not modify this part, it is used for visulization
    generated_frame = stack(decoded_frame_list).permute(1, 0, 2, 3, 4)
    label_frame = stack(label_list).permute(1, 0, 2, 3, 4)

    assert generated_frame.shape == (1, 630, 3, 32, 64), f"The shape of output should be (1, 630

    self.make_gif(generated_frame[0], os.path.join(self.args.save_root, f'pred_seq{idx}.gif'))

    # Reshape the generated frame to (630, 3 * 64 * 32)
    generated_frame = generated_frame.reshape(630, -1)

    return generated_frame
```

B. How do you implement reparameterization tricks

Reparameterization 的實作是為了解決原本的作法無法進行反向傳播的問題。因為原本的 VAE(如下圖左側)，透過從 representation 的 distribution 中 sample 出 feature，再餵給 decoder，但是這樣的作法會阻斷反向傳播，

也就是梯度傳到 sample 的地方就無法繼續傳下去了。因此採用 reparameterization trick，把它變成平均加上標準差成上 standard gaussian distribution 中 sample 出來的浮動值以做到跟直接從 representation 的 distribution sample 一樣的效果。



實作如下，透過 gaussian predictor 取得 representation distribution 的 mean 與 variance，再將這些值透過上述方法，轉成標準差後，計算成等價於直接 sample 的值。

```
def reparameterize(self, mu, logvar):
    # TODO
    var = torch.exp(logvar)
    sd = torch.sqrt(var)
    sample_std_gaussian = torch.randn_like(sd) # shape same as sd
    return mu + sample_std_gaussian * sd
```

C. How do you set your teacher forcing strategy

我實作的 teacher forcing strategy 主要分成三部分，第一部分是 ratio 的應用(下圖一)，當使用 teacher forcing strategy 時，cVAE 生成圖片的參照就不會是 inference 的方法，用生成的前一幀來做參考，而是直接拿 training data 的 ground truth 作為參考圖，避免訓練過程中，用來參考的前一幀預測圖還不夠好，連帶造成後面生成幀的錯誤傳遞及累積。

第二部分是混和策略(圖二)，因為訓練一定程度之後，會需要把 teacher forcing strategy 關掉，讓他回復正常的利用預測幀來做後面的推理，否則會造成 GIF 整個黑掉的結果。但是突然關掉 teacher forcing 會導致變化過於突然，因此我採用混合的方法，將採用 teacher forcing 的機率 self.tfr 作為比率參數，混合 ground truth 與預測圖，讓他可以用更平滑的方式學習。

第三部分是機率的遞減，如上文提及，不能讓他一直保持 teacher forcing strategy，因此設定在 self.tfr_sde 個 epoch 之後，開始逐 epoch 減去 self.tfr_d_step 的機率，讓他以機率性啟用 teacher forcing，直到機率完全歸零。

在設置上，我發現把 teacher forcing 完全關掉的效果反而會更好，可能是因為本次 lab 的影片生成較簡單，解析度也較低，因此比較不會碰到難以學習 latent code 的問題。因此不受干擾的 distribution 反而可以得到更高分

```
if adapt_TeacherForcing == True:# previous image for reference to predict, when
    ref_img = self.mix_teacher_forcing(img[:,idx-1,:,:], pred_img, self.tfr) # u
else:
    ref_img = pred_img # previous image for reference to predict
```

```
def mix_teacher_forcing(self, gt_img, pred_img, ratio):
    return gt_img * ratio + pred_img * (1-ratio)
```

```
def teacher_forcing_ratio_update(self):
    # TODO

    if self.current_epoch >= self.tfr_sde:
        self.tfr -=  self.tfr_d_step
        self.tfr = max(self.tfr, 0)
```

```
adapt_TeacherForcing = True if random.random() < self.tfr else False
```

D. How do you set your kl annealing ratio

Kl annealing 主要的功能是避免本次 lab 的 cVAE 的 decoder 忽視 latent representation 的問題。具體來說，因為 cVAE 的 decoder 是 auto-regressive

的，因此他可以參照前一個 frame 來做輸出，而此時如果 latent representation 的 path 專注於優化 kl term，就會導致其 distribution 傾向於變成 standard distribution，使得 latent representation 變得無資訊。Kl annealing 就是透過降低 kl term 的權重，來避免這個問題。

我實作 kl annealing 的兩種方法，Cyclic 與 Monotonic，不過 Monotonic 如論文中所述，可以視為 Cyclic 的一種變體，只是改變了參數，因此我先針對 Cyclic 的實作細節介紹。

在 Cyclic 的實作上，會做多次的 kl annealing，讓 KL term 的 beta 係數不斷從 0 爬升到 1，再降回 0，過程稱為一個 cycle。一個 cycle 中，會有兩個階段，annealing 與 fixed。在 annealing 中，會把 beta 歸零，然後線性上升到 1，之後在 fixed 階段保持一陣子，讓他能針對正確的 loss (beta 為 1)去做優化。實作如下圖:

```python
class kl_annealing(): # 用來平衡KL-Divergence與圖片loss的方法，避免KL過大或過小影響結果。透過改變KL的權重

    def update(self):
        if self.type == 'Cyclical':
            self.frange_cycle_linear(n_iter = self.total_epoch, n_cycle = self.cycle, ratio=self.ratio)
        elif self.type == 'Monotonic':
            self.frange_cycle_linear(n_iter = self.total_epoch, n_cycle=1, ratio=self.ratio) # one cycle = Mo
        self.epoch += 1

    def get_beta(self):
        return self.beta

    def frange_cycle_linear(self, n_iter, start=0.0, stop=1.0,  n_cycle=1, ratio=1):
        # start: beta start point, stop: the ceil of beta, ratio: the ratio to increase, otherwise stay in st
        period = np.ceil(n_iter / n_cycle)
        step = (stop-start) / (np.ceil(period*ratio)-1) # compute one step length for one cycle, 1 for 0.0, p

        mod = self.epoch % period
        if mod < period * ratio:
            self.beta = start + step * mod
            if self.beta > stop:
                self.beta = stop
        else:
            pass # do nothing just stay at stop(1)
```

主要有兩個參數 M,R 可以調整，M 代表的是 cycle 的數量，R 為 annealing ratio，也就是 cycle 的 annealing 階段的持續時間比例，也就是 beta 爬升的時間。首先利用 M 參數計算出一個 cycle 是多少 epoch，再根據 R 參數去分配哪些要上升，那些不用，即完成 cyclic annealing。至於 monotonic annealing 則是 cyclic 的特殊形，相當於只有一個 cycle。

至於我在訓練時的參數配置，我的 M 設為 4，R 設為 0.5，遵照論文中提到的參數。

E. Time Smoothing Loss Strategy

除了預設的上述的優化方法外，我還引入的 time smoothing loss 的方法，參考自 Every Dance Now 提到的概念，並使用 time smoothing loss 方法來實行概念。

方法核心為，計算影片 ground truth 的前後兩個 frame 的差，再計算升成的影片的兩個 frame 的差，看看生成的影片在 frame 的差上，是否與 ground truth 吻合，目的是為了讓模型能夠更多的參考影片的連續性。

實作如下圖，照上述方法去對圖片做逐 pixel 的相減，然後再比對兩個相差圖的 MSE，就可以得到設計出的圖片連續性的指標。之後再把他加進去 training loss 裡面，使用 time_smoothing_rate 這個 hyper parameter 調節。

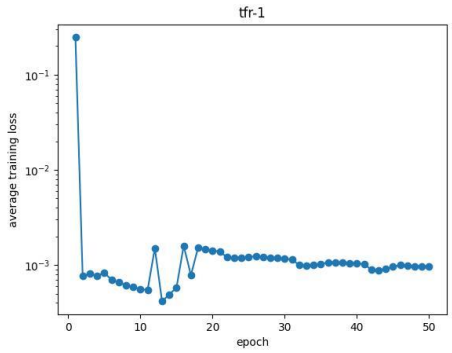
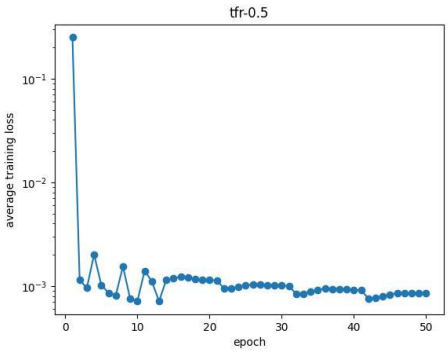
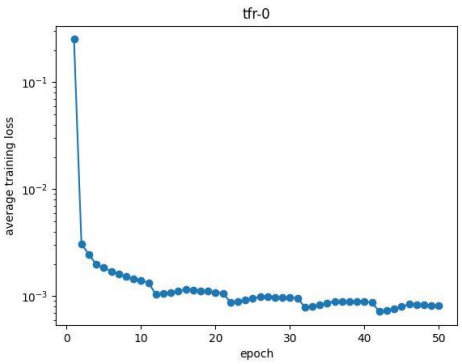
```
time_smoothing_loss = self.mse_criterion(prediction-ref_img, img[:,idx,:,:,:]-img[:,idx-1,:,:,:])
video_loss += self.mse_criterion(prediction, img[:,idx,:,:,:]) + beta * kl_criterion(mu, logvar, self.
            + time_smoothing_rate * time_smoothing_loss
```

4. Analysis & Discussion

A. Plot Teacher forcing ratio

針對 Teacher forcing 的分析，我對於三種初始值分別訓練 50 個 epoch 並觀察其下降的狀況。使用以下指令:

python3 Trainer_for_plot.py --num_epoch 50 --batch_size 2 --DR dataset --save_root check_points --lr 0.001 --kl_anneal_cycle 5 --kl_anneal_ratio 0.5 --tfr $

Tfr 1	Tfr 0.5	Tfr 0
		

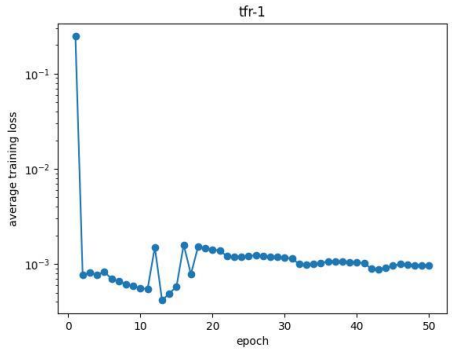
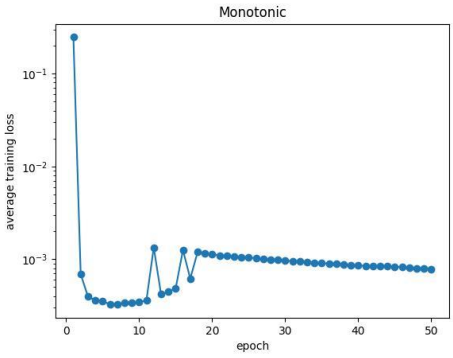
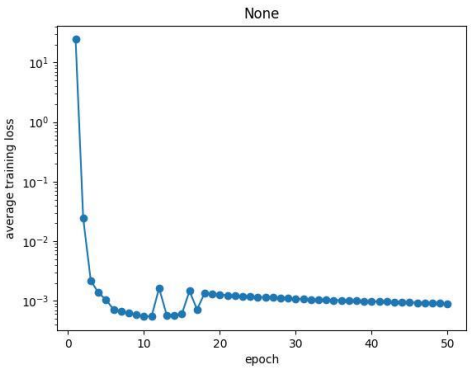
先從左側兩個 tfr 不為 0 的看起，可以看到開啟 teacher forcing 之後，training loss 在最一開始是下降的更快更穩定的，可以讓模型能更好的學到如何產出基本足夠好的 latent code，在 Tfr 為 1 的情況時，到了第 10 個 epoch 才開始出現波動，也就是 Teacher forcing 會被隨機開啟及關閉，Tfr 0.5 則是一開始就在波動。待兩邊的 Tfr 都歸零後，就會變成穩定的下降。而 Tfr 初始為 0 的狀況，則是一開始就是慢慢下降，但是可以看到雖然沒有 Teacher forcing，但他的 loss 下降情況是更好的，在最終的第 50 個 epoch 得到的結果也是比 Tfr 初始為 1 時更好，跟 Tfr 初始為 0.5 時差不多。

從以上觀察可以看出，在本次實驗中，可能因為要生產的影片解析度比較低，影片的內容也比較簡單，因此模型可以比較容易的學到好的 latent code，Teacher forcing 的幫助就比較不明顯，並且可能會反過來因為學習到了不對的 distribution 而造成 loss 下降的效果比較差。在我實驗各種不同參數後，在 kaggle 得到的最高分也是最一開始就不開 teacher forcing 的狀況。

B. Plot the loss curve while training with different settings.

針對不同 kl-annealing 的分析，我對於兩種 kl-annealing 方式以及不使用 kl-annealing 分別訓練 50 個 epoch 並觀察其下降的狀況。使用以下指令:

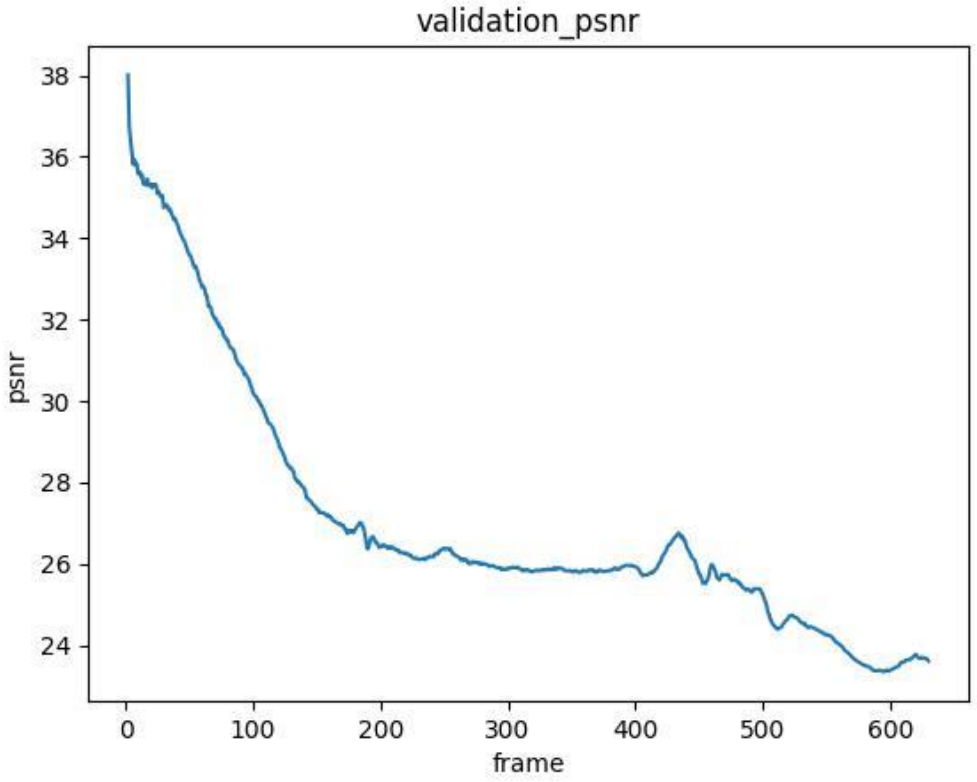
python3 Trainer_for_plot.py --num_epoch 50 --batch_size 2 --DR dataset --save_root check_points --lr 0.001 --tfr 1 --kl_anneal_ratio 0.5 --kl_anneal_type $

Cyclic	Monotonic	None
		

首先看到 Cyclic 與 Monotonic 的主要區別，前面因為 beta 都在變化的關係，所以會有波動的狀況，後面則是只剩 Cyclic 有區段性的波動，也就是 beta 從 1 重新歸 0 的時候會掉一階。可以看到 Cyclic 跟 Monotonic 最終的收斂情況是差不多的，Monotonic 在收斂的速度上也是稍快於 Cyclic，基本上跟論文提到的相同。這裡由於運算資源不足(colab)，因此只實驗 50 個 epoch，可能還沒進到論文中提到的後期 Cyclic 會驟降又升回來的狀況。而 None 的圖中，雖然不明顯，但是 training loss 的斜率看起來是比有使用 kl-annealing 的圖更平的，推測在不使用 kl-annealing 的情況下，loss 的下降結果會因為學不到好的 latent code 而更糟。

C. Plot the PSNR-per frame diagram in validation dataset

這裡對於 validation dataset 中，生成的每一個 frame 計算其 psnr，產出的圖如下:

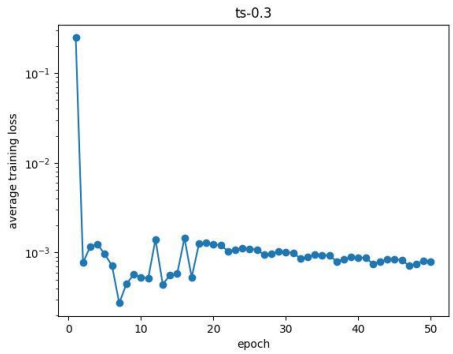
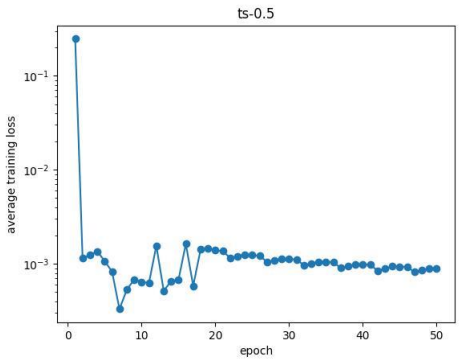
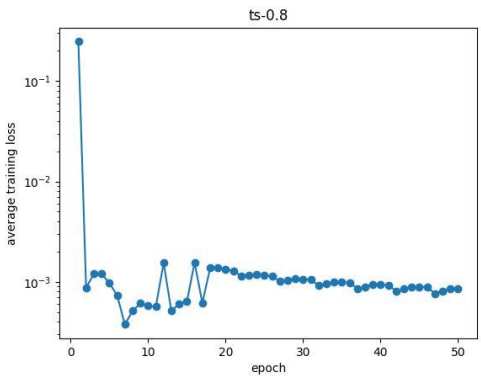


可以看到，越後面的 frame 會產生越小的 psnr，因為在座 inference 時是根據前一個 frame，因此錯誤會在影片生成的過程中傳遞，導致 psnr 的分數會越來越低。

D. Other training strategy analysis

這裡針對我額外添加的 time smoothing loss 的狀況作分析，與上述同樣採用 50 個 epoch 觀察其形況，使用指令如下:

python3 Trainer_for_plot.py --num_epoch 50 --batch_size 2 --DR dataset --save_root check_points --lr 0.001 --tfr 1 --kl_anneal_ratio 0.5 --time_smoothing_rate $

Tsr 0.3	Tsr 0.5	Tsr 0.8
		

基本上這三張圖的差異不太大，甚至可以說他們的波動、圖形快要變成一樣了，不過因為調高 time smoothing rate 本身會讓 training 的 loss 上升一些，所以算回原本的 ELBO，應該是 0.8 的圖得到的 ELBO 最小，不過實際上的差異也並不太多。

因此對於我嘗試採用 time smoothing loss 的方法，來增進圖片間的連續性，效果看起來是有所限制，可能只能對於影片的升成產生小量的品質提升。