1. Derivate conditional VAE formula (5%)

$$\log\big(p_\theta(x|c)\big) = \log\left(\int_z p_\theta(x,z|c)dz\right) = \log\left(\int_z \frac{q_\phi(z|x,c)p_\theta(x,z|c)}{q_\phi(z|x,c)}dz\right)$$

$$= \log\left(\mathbb{E}_{z\sim q_\phi(z|x,c)}\left[\frac{p_\theta(x,z|c)}{q_\phi(z|x,c)}\right]\right)$$

$$\geq \mathbb{E}_{z\sim q_\phi(z|x,c)}\left[\log\left(\frac{p_\theta(x,z|c)}{q_\phi(z|x,c)}\right)\right]$$

$$= \mathbb{E}_{z\sim q_\phi(z|x,c)}\left[\log\left(\frac{p_\theta(x|z,c)p_\theta(z|c)}{q_\phi(z|x,c)}\right)\right]$$

$$= \mathbb{E}_{z\sim q_\phi(z|x,c)}\big[\log\big(p_\theta(x|z,c)\big)\big]$$

$$- \mathbb{E}_{z\sim q_\phi(z|x,c)}\left[\log\left(\frac{q_\phi(z|x,c)}{p_\theta(z|c)}\right)\right]$$

$$= \mathbb{E}_{z\sim q_\phi(z|x,c)}\big[\log\big(p_\theta(x|z,c)\big)\big] - KL(q_\phi(z|x,c)||p_\theta(z|c))$$

Define $L(x,z,c,\theta) = \mathbb{E}_{z\sim q_\phi(z|x,c)}\big[\log\big(p_\theta(x|z,c)\big)\big] - KL(q_\phi(z|x,c)||p_\theta(z|c))$

So, instead of maximize log likelihood $\log\big(p_\theta(x|c)\big)$, we maximize the variational lower bound $L(x,z,c,\theta)$.

2. Introduction (5%)

In this lab, we aim to train a video predictor by implementing a VAE-based model. Given an initial image and a sequence of poses, the model can predict a video that is similar to the image with the same poses in the sequence.

3. Implementation details (25%)

- How do you write your training protocol (10%)

```python
def training_one_step(self, img, label, adapt_TeacherForcing):
    # permute the size of img and label from [batch_size, frame_length,channel, W, H]
    # to [frame_length, batch_size,channel, W, H]
    img = img.permute(1, 0, 2, 3, 4)
    label = label.permute(1, 0, 2, 3, 4)
    MSELoss = 0
    KLLoss = 0
    out = img[0]
    for frame_idx in range(1, self.args.train_vi_len):
        img_encode = self.frame_transformation(img[frame_idx])
        label_encode = self.label_transformation(label[frame_idx])
        z, mu, logvar = self.Gaussian_Predictor(img_encode, label_encode)

        img_encode = self.frame_transformation(out)
        label_encode = self.label_transformation(label[frame_idx])
        out = self.Generator(self.Decoder_Fusion(img_encode,label_encode,z))

        KLLoss +=  kl_criterion(mu, logvar, self.batch_size)
        MSELoss += self.mse_criterion(out,img[frame_idx])
        if adapt_TeacherForcing:
            out = img[frame_idx]

    beta = self.kl_annealing.get_beta()
    Loss = MSELoss + beta * KLLoss
    self.optim.zero_grad()
    Loss.backward()
    self.optim.step()
    #self.optimizer_step()
```

First of all, we permute the shape of img and label from [batch_size, frame_length, channel, W, H] to [frame_length, batch_size, channel, W, H]. Next, for each frame, we forward the image frame and label to the transformation and Gaussian_Predictor to get the z, mu, and logvar. We compute the KL Loss between N(mu, e^(logvar*0.5)) and N(0,I). Then, we forward the z, out (either the last iteration's output or the last iteration's ground truth according to the adapt_TeacherForcing), and label to the transformation decoder_fusion and generator to obtain the predicted image. After that, we compute the MSE Loss between the predicted image and the ground truth. Lastly, we compute the Total Loss and update the network parameters.

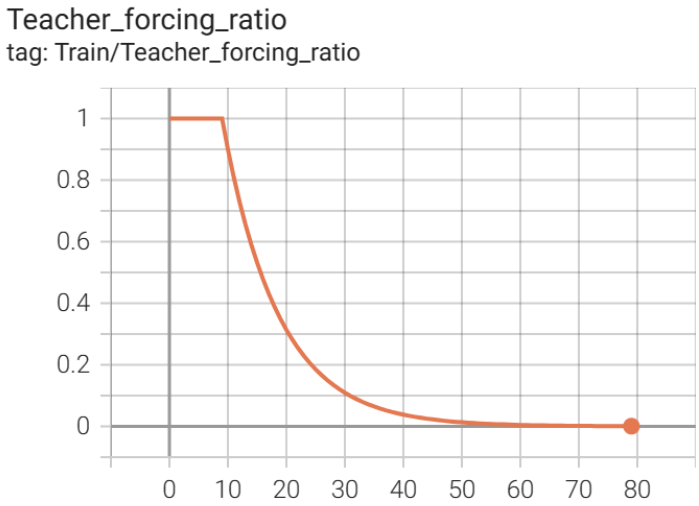
- How do you implement reparameterization tricks (5%)

```python
def reparameterize(self, mu, logvar):
    # TODO
    std = torch.exp(logvar* 0.5)
    epsilon = torch.randn_like(std)
    return mu + std * epsilon
```

Sample $\epsilon \sim N(0, I)$ and compute z = $\mu + \epsilon \circ \sigma$, where $\circ$ means the entrywise product and $\sigma = e^{logvar*0.5}$

- How do you set your teacher forcing strategy (5%)

  For the first self.tfr_sde (default: 10) epochs, we set the self.tfr (teaching force ratio) to 1. After that, we decrease it each epoch by multiplying by 0.95.



- How do you set your kl annealing ratio (5%)

```python
class kl_annealing():
    def __init__(self, args, current_epoch=0):
        self.kl_anneal_type = args.kl_anneal_type
        self.kl_anneal_cycle = args.kl_anneal_cycle
        self.kl_anneal_ratio = args.kl_anneal_ratio
        self.current_steps = current_epoch + 1

    def update(self):
        self.current_steps += 1

    def get_beta(self):
        if self.kl_anneal_type == 'Monotonic':
            if self.current_steps < 10:
                return self.current_steps/10
            else:
                return 1.0
        elif self.kl_anneal_type == 'Cyclic':
            steps = self.current_steps % 10
            if steps < 5:
                return steps/5.0
            else:
                return 1.0
        else:
            return 1.0

    def frange_cycle_linear(self, n_iter, start=0.0, stop=1.0,  n_cycle=1, ratio=1):
        return None
```

For each epoch, we call .update() to update the current_steps and use .get_beta() to get the value of kl annealing ratio.

The rule for .get_beta() is as follows:

- Cyclic:

We set the value of kl annealing ratio iterate over [0.2, 0.4, 0.6, 0.8, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0].

- Monotonic:

We set the value of kl annealing ratio to 0.1+k*0.1, $k \in [0,8]$ for the first 9 epochs. After that, we set the kl annealing ratio to 1.0.

- None:

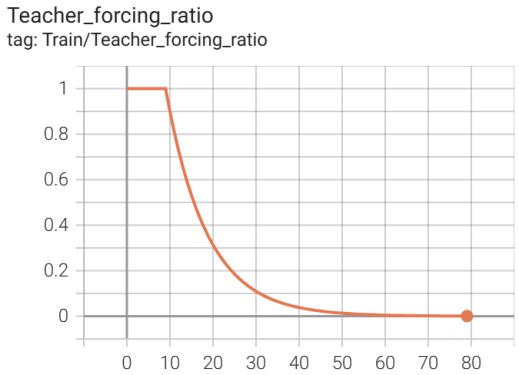
We set the value of kl annealing ratio to 1.0 for every epochs.

4. Analysis & Discussion (25%)

The hyperparameters is set as follows:

- Initial Learning rate: 1e-3
- MultiStepLR(milestones=[60], gamma=0.1)
- Optimizer: Adamax
- num_epoch= 80
- tfr_d_step=0.9
- Others remain as default.

- Plot Teacher forcing ratio (5%)

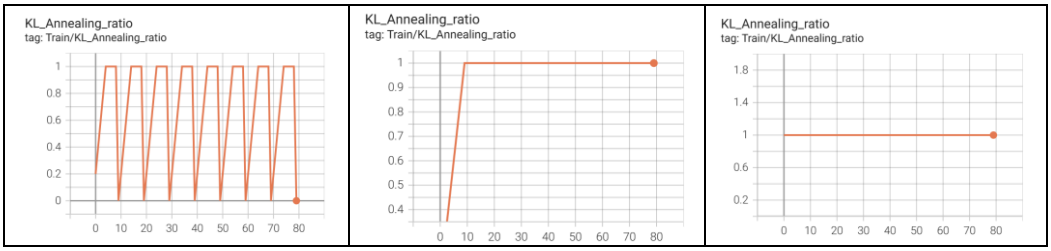


Analyze/Observation:

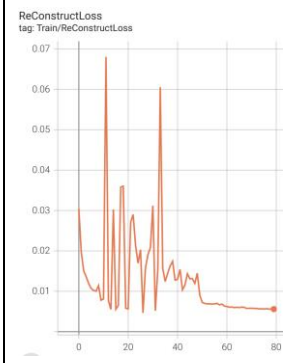
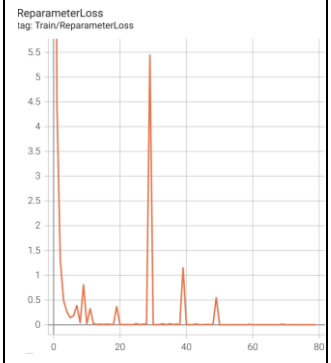
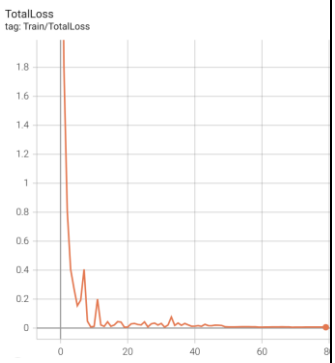
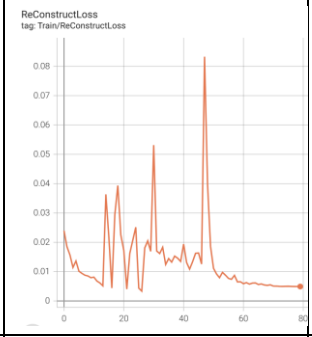
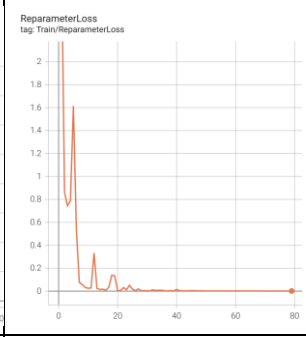
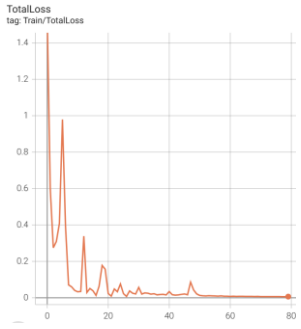
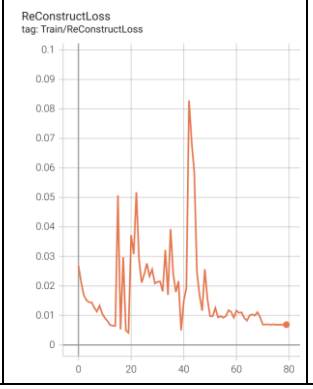
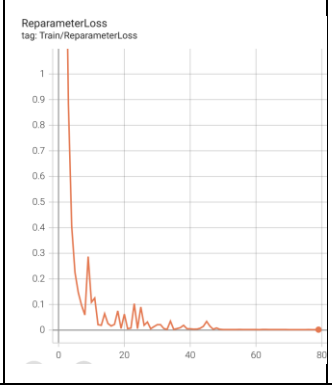
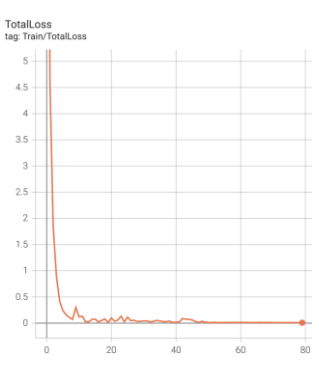
Loss curve is in the next part. We can observe that in the first 10 epochs, the ReConstructLoss shows a decreasing trend because the Teacher forcing ratio is set to 1. However, afterwards, the ReConstructLoss begins to fluctuate. This is because the Teacher forcing ratio is less than 1, meaning that for some epochs, Teacher_forcing=false. As a result, the model has to rely on its own predictions for the remaining sequence, which poses a significant challenge during the early stages of model training.

- Plot the loss curve while training with different settings. Analyze the difference between them(10%)

Cyclic	Monotonic	None



TotalLoss= ReConstructLoss + β* ReparameterLoss, where β is Teacher forcing ratio.

	ReConstructLoss	ReparameterLoss	TotalLoss
Cyclic			
Monotonic			
None			

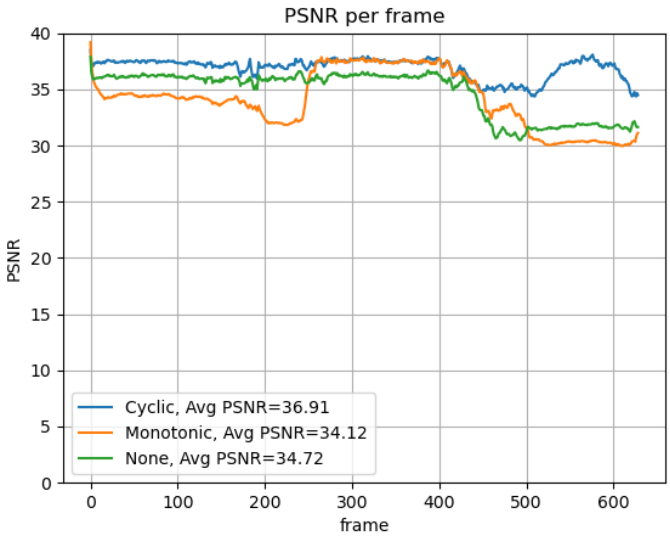
Analyze/Observation:

- We can observe that the loss of Monotonic and None cases are similar after first 10 epochs, since they optimize same loss function.
- The ReparameterLoss of Monotonic and None Cases shows a decreasing trend, but for the Cyclic Case, it occasionally spikes, mostly corresponding to β=0.

- All three cases have a ReConstructLoss significantly greater than the ReparameterLoss, indicating that the ReparameterLoss is relatively better optimized compared to ReConstructLoss.

- Plot the PSNR-per frame diagram in validation dataset (5%)

  We plot the validation the PSNR-per frame of the last epoch.



  Analyze/Observation:

  We can observe that the cyclic case provides the best results, with no decrease in PSNR as the frame length increases. In other cases, we can observe that the PSNR score decreases as the frame length increases.

- Other training strategy analysis (Bonus) (5%)
  - We add the Sigmoid() at the last layer of Generator.

```python
class Generator(nn.Sequential):
    def __init__(self, input_nc, output_nc):
        super(Generator, self).__init__(
            DepthConvBlock(input_nc, input_nc),
            ResidualBlock(input_nc, input_nc//2),
            DepthConvBlock(input_nc//2, input_nc//2),
            ResidualBlock(input_nc//2, input_nc//4),
            DepthConvBlock(input_nc//4, input_nc//4),
            ResidualBlock(input_nc//4, input_nc//8),
            DepthConvBlock(input_nc//8, input_nc//8),
            nn.Conv2d(input_nc//8, 3, 1),
            nn.Sigmoid() #New Add
        )

    def forward(self, input):
        return super().forward(input)
```

  - Change optimizer

  We took Section 4 - Cyclic Case, where the Validation Score is 36.91. Uploading this model to Kaggle resulted in a score of 31.90. We then used

load_checkpoint() to load this model and employed the ASGD optimizer to converge the model. And Then, we obtained results with a Validation Score of 37.75 (+1.84) and a Kaggle score of 34.63 (+2.73).