312554012 王偉誠 Lab5 : Conditional VAE for Video Prediction

1. Introduction

This lab involves employing a Conditional Variational Autoencoder (VAE) for the purpose of video prediction. The dataset at hand comprises dance movement videos, with each sequence segmented into 630 frames and stored as 30 individual 64x32 PNG image files. Our objective is to take the initial frame as input and utilize the generative capabilities of the VAE to forecast the subsequent 629 frames. Moreover, within the dataset, each frame is associated with a corresponding pose label, which can be utilized as a conditioning factor to enhance the prediction process.

2. Implementation details

i. How do you write your training protocol

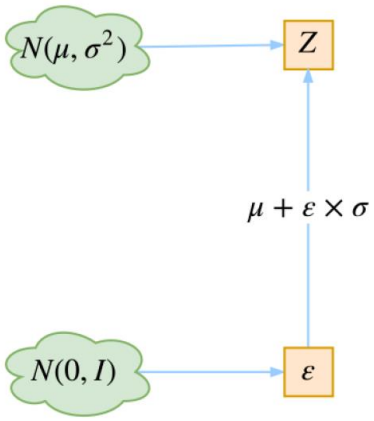
```python
def training_one_step(self, img, label, adapt_TeacherForcing):

    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)

    self.optim.zero_grad()

    h_seq = [self.frame_transformation(img[i]) for i in range(self.train_vi_len)]
    p_seq = [self.label_transformation(label[i]) for i in range(self.train_vi_len)]

    mse = 0
    kld = 0
    for t in range(1, self.train_vi_len):
        h_t = h_seq[t]
        h_t_1 = h_seq[t - 1]
        p_t = p_seq[t]

        z_t, mu, logvar = self.Gaussian_Predictor(h_t, p_t)
        g_t = self.Decoder_Fusion(h_t_1, p_t, z_t)
        img_gen = self.Generator(g_t)

        if not adapt_TeacherForcing:
            h_seq[t] = self.frame_transformation(img_gen)

        mse += self.mse_criterion(img_gen, img[t])
        kld += kl_criterion(mu, logvar, self.batch_size)

    loss = mse + self.kl_annealing.get_beta() * kld  # type: ignore
    loss.backward()  # type: ignore
    self.optimizer_step()

    return (
        loss / self.train_vi_len,
        mse / self.train_vi_len,
        kld / self.train_vi_len,
    )
```

Firstly, both the frame at time step x_t-1 and the frame at time step x_t are passed through a frame encoder, resulting in the generation of latent vectors h_t-1 and h_t respectively. Simultaneously, the pose information p_t is inputted into the pose encoder. Subsequently, h_t undergoes the posterior process, as depicted on the right side of the diagram, to yield the mean and variance parameters. The reparameterization trick is then employed to sample z_t using the information from the mean and variance.

The sampled z_t, in conjunction with h_t-1 and p_t, is fed into the decoder, forming the decoder fusion on the left side of the diagram. This fusion aims to predict the latent vector g_t for the next frame. To avoid confusion with h_t on the right side, the predicted latent vector is denoted as g_t rather than h_t. Finally, introducing g_t into the generator facilitates the image generation process.

ii. How do you implement reparameterization tricks

The Posterior, implemented as a Gaussian Predictor, generates both the mean and variance parameters. Through the utilization of the reparameterization trick, it becomes possible to sample z_t from the information provided by the mean and variance. The reparameterization trick involves sampling Z from a distribution $N(\mu, \sigma^2)$, which is equivalently accomplished by sampling ε from $N(0, I)$, and subsequently computing $Z = \mu + \varepsilon \times \sigma$, as illustrated in the diagram below:



In our implementation, we employ the logarithm of the variance, necessitating a transformation:

$$Z = \mu + \epsilon \times e^{0.5 \times \log \sigma^2} = \mu + \epsilon \times e^{\log \sqrt{\sigma^2}} = \mu + \epsilon \times \sigma$$

```python
class Gaussian_Predictor(nn.Sequential):
    def __init__(self, in_chans=48, out_chans=96):
        super(Gaussian_Predictor, self).__init__(
            ResidualBlock(in_chans, out_chans // 4),
            DepthConvBlock(out_chans // 4, out_chans // 4),
            ResidualBlock(out_chans // 4, out_chans // 2),
            DepthConvBlock(out_chans // 2, out_chans // 2),
            ResidualBlock(out_chans // 2, out_chans),
            nn.LeakyReLU(True),
            nn.Conv2d(out_chans, out_chans * 2, kernel_size=1),
        )

    def reparameterize(self, mu, logvar):
        # TODO

        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, img, label):
        feature = torch.cat([img, label], dim=1)
        parm = super().forward(feature)
        mu, logvar = torch.chunk(parm, 2, dim=1)
        z = self.reparameterize(mu, logvar)

        return z, mu, logvar
```

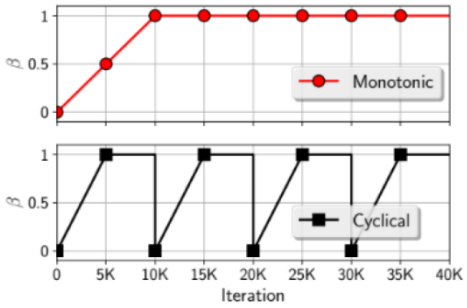
iii. How do you set your teacher forcing strategy

Initially, setting a relatively high value for teacher forcing is crucial to enable the model to steer clear of learning from errors as much as possible. Gradually, this value is diminished, allowing for a more comprehensive learning process. As mentioned earlier, the significance of employing teacher forcing has been highlighted. Without incorporating teacher forcing at all, the model wouldn't converge effectively, while relying entirely on non-teacher forcing at the outset would yield suboptimal early results. For this particular lab, I have maintained a fixed value of tfr_d_step=0.05, implying that after 20 epochs from the tfr_sde, the teacher forcing ratio will decline to 0.

Additionally, I've experimented with adjusting tfr_sde, which determines the number of epochs at the beginning of training when entirely teacher forcing is used. In this instance, I've set it to commence the reduction from the 10th epoch onward. This strategic manipulation aims to strike a balance between harnessing the benefits of teacher forcing and gradually transitioning toward a more autonomous prediction mode as training progresses.

iv. How do you set your kl annealing ratio

The purpose of the KL weight (β) is to guide the model in deciding whether to prioritize minimizing frame prediction errors or fitting the prior distribution. If β is too small, the model becomes overly focused on minimizing frame prediction errors, which could lead to the model merely duplicating the target frame and losing its generative capacity. On the other hand, if β is too large, the opposite effect occurs, resulting in suboptimal prediction outcomes.



To address this, the strategy of KL annealing is employed to adjust the KL weight (β). This strategy can be categorized as either monotonic or cyclical. In the monotonic approach, the weight is set to increase linearly during the initial phase and then remains fixed at 1 during the latter phase.

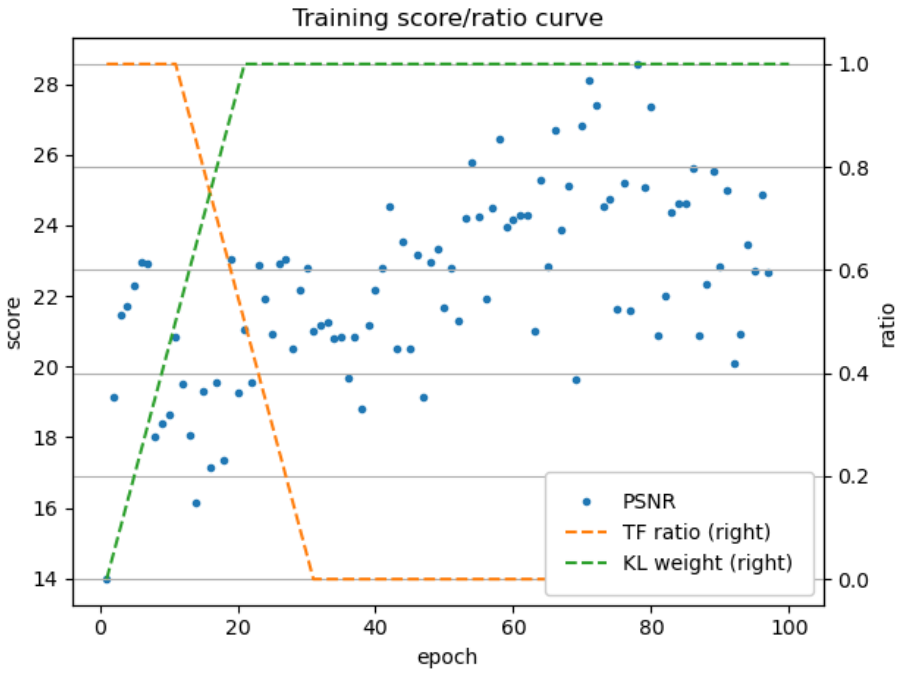
In the cyclical approach, the monotonic pattern is repeated multiple times in succession.

```python
class kl_annealing:
    def __init__(self, args, current_epoch=0):
        super().__init__()
        self.use_cycle = args.kl_anneal_type == "Cyclical"
        self.n_iter = args.num_epoch
        self.n_cycle = args.kl_anneal_cycle
        self.ratio = args.kl_anneal_ratio
        self.period = self.n_iter / self.n_cycle
        self.step = self.ratio / self.period
        self.current_epoch = current_epoch
        self.epoch = 0
        self.v = 0
        self.beta = args.beta

    def update(self):
        self.epoch += 1
        if self.use_cycle:
            if self.epoch % self.period == 0:
                self.beta = 0.0001
            else:
                self.beta += self.step
            if self.beta > 1.0:
                self.beta = 1.0
        else:
            self.v += self.step / 2
            if self.v > 1.0:
                self.v = 1.0
            self.beta = self.v

    def get_beta(self):
        return self.beta
```

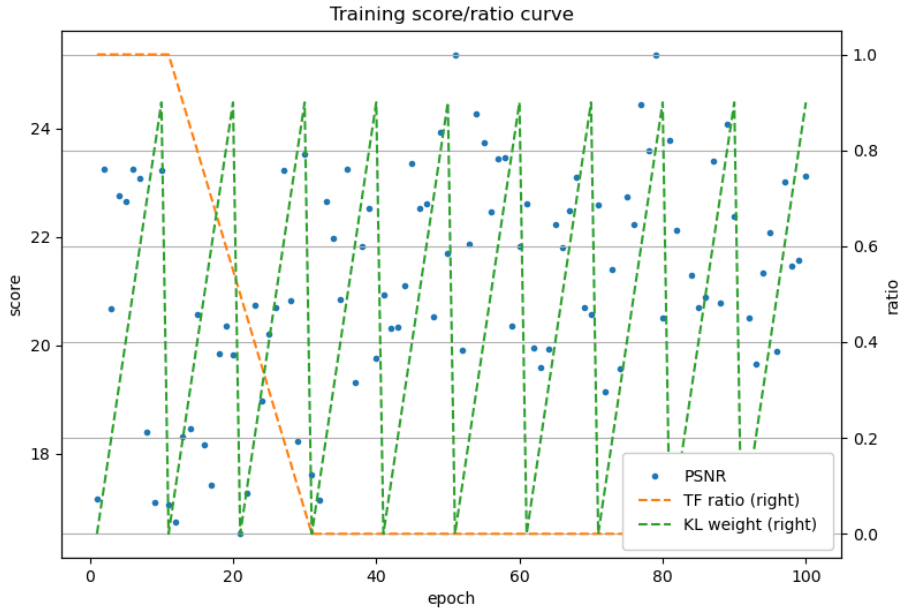
3. Analysis & Discussion

i. Plot Teacher forcing ratio

Monotonic

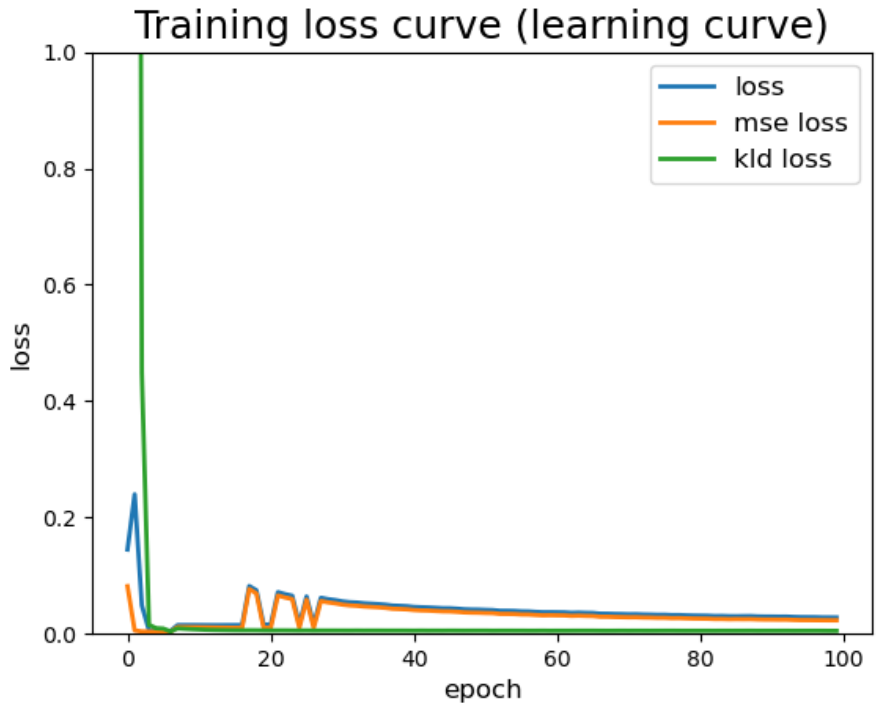


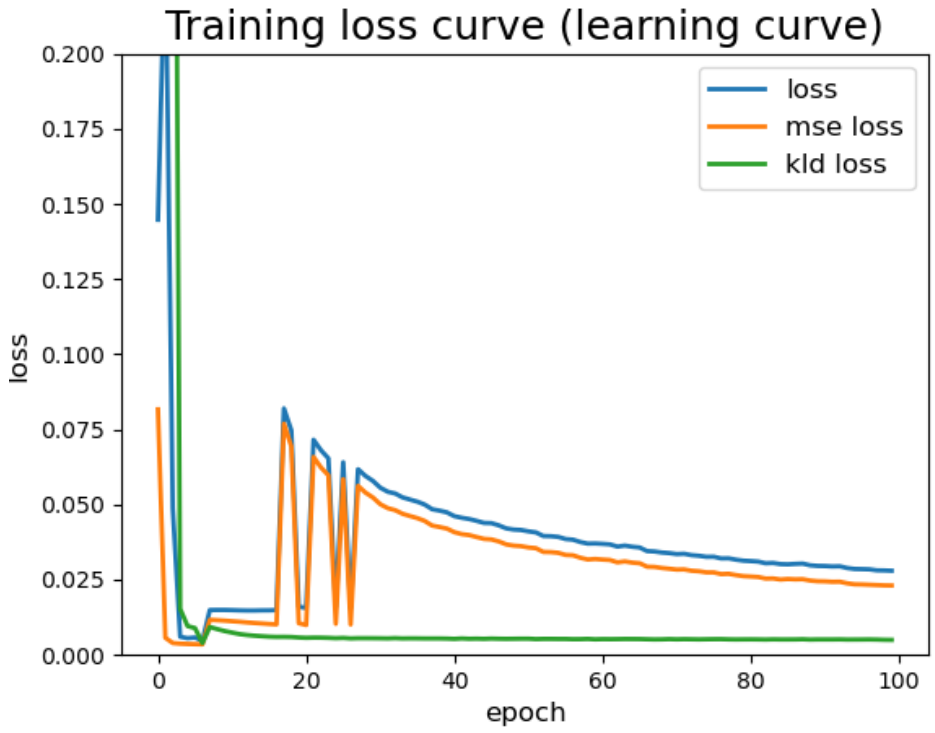
Cyclical



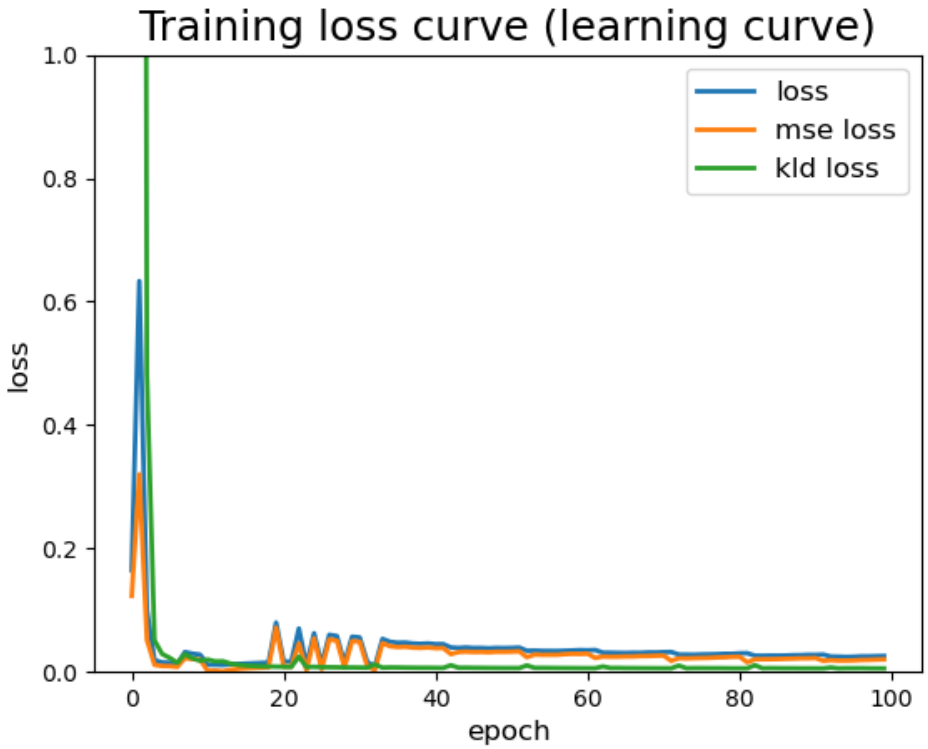
ii. Plot the loss curve while training with different settings

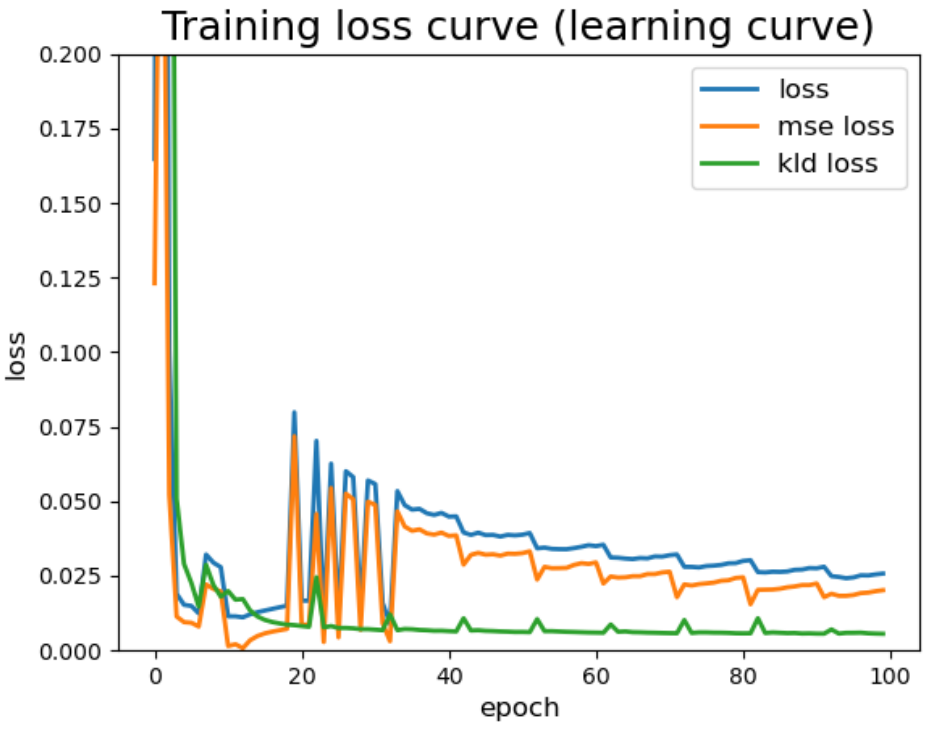
(a) With KL annealing (Monotonic)



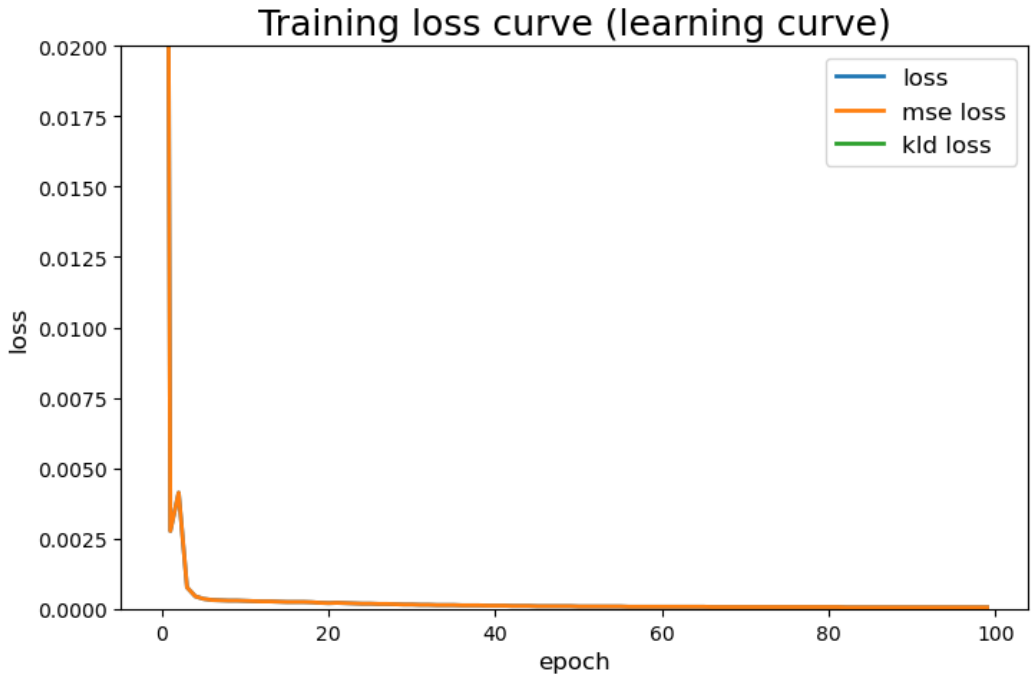


(b) With KL annealing (Cyclical)





(c) Without KL annealing



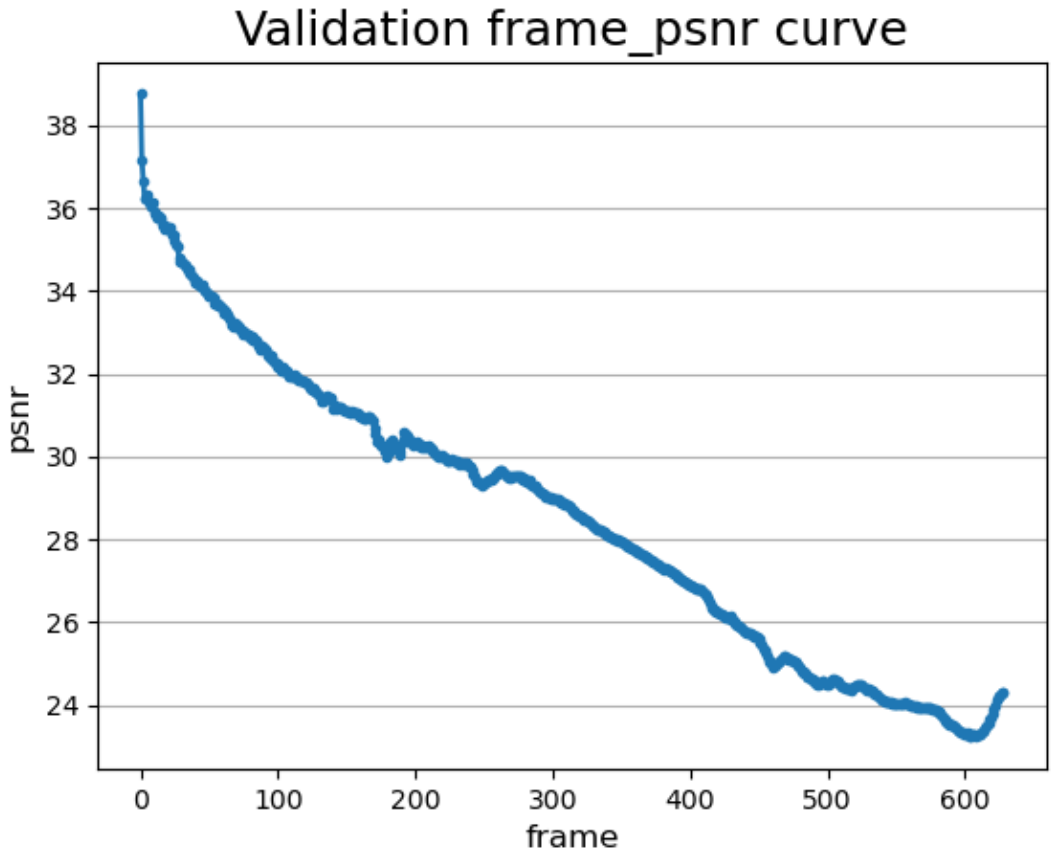
(d) Discuss the results

In this specific lab scenario, employing the cyclical approach for KL annealing yields better results in terms of Peak Signal-to-Noise Ratio (PSNR) compared to the monotonic approach. This is attributed to the fact that as the model's training stabilizes, the KL term tends to vanish towards zero, causing the learned features to lose their ability to effectively represent observed data. The cyclical method effectively addresses the issue of KL vanishing, enhancing performance during training.

Moreover, I have observed that although the KL divergence (KLD) loss does impact the training process, the total loss is significantly influenced by the Mean Squared Error (MSE) loss. The two losses practically coincide, with the exception of the cyclical approach when the beta parameter returns from 1 to 0. However, due to the formulation of the loss as loss = mse + kld * β (where in this case, β = 0), the KLD loss does not substantially affect the total loss.

In the absence of KL annealing, as the model's training progresses and gradually stabilizes, the loss value tends to decrease over time. However, this continuous reduction in loss can eventually lead to a situation where the gradients become very small, making it challenging to effectively update the model's weights using gradient-based optimization methods.

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



iv. Derivate conditional VAE formula

We know conditional distribution $p(X|c;\theta)$

$$p(X|c;\theta) = \int p(X|Z,c;\theta)\, p(Z|c)\, dZ$$

To see how the EM works, the chain rule of probability suggest:

$$\log p(X|c;\theta) = \log p(X,Z|c;\theta) - \log p(Z|X,c;\theta)$$

By introducing an arbitary distribution $q(Z|c)$ on both sides and integrate over Z:

$$\log p(X|c;\theta) = \int q(Z|c) \log p(X|c;\theta)\, dZ$$

$$= \int q(Z|c) \log p(X,Z|c;\theta)\, dZ - \int q(Z|c) \log p(Z|X,c;\theta)\, dZ$$

$$= \int q(Z|c) \log p(X,Z|c;\theta)\, dZ - \int q(Z|c) \log q(Z|c)\, dZ$$

$$+ \int q(Z|c) \log q(Z|c)\, dZ - \int q(Z|c) \log p(Z|X,c;\theta)\, dZ$$

$$= L(X,q,\theta|c) + KL(q(Z|c) \,\|\, p(Z|X,c;\theta))$$

where $L(X,q,\theta|c) = \int q(Z|c) \log p(X,Z|c;\theta)\, dZ - \int q(Z|c) \log q(Z|c)\, dZ$

and $KL(q(Z|c) \,\|\, p(Z|X,c;\theta)) = \int q(Z|c) \log \frac{q(Z|c)}{p(Z|X,c;\theta)}\, dZ$

since we know KL-divergence is non-negative,

$$\log p(X|c;\theta) \geq L(X,q,\theta|c)$$

↳ evidence lower bound (ELBO)

We rearrange above equation:

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

Since the equality holds for any choice of $q(Z|c)$, we can also introduce a distribution $q(Z|X,c;\phi)$ modeled by another neural network (the encoder) with parameter ϕ.

$$L(X,q,\theta|c) = \log p(X|c;\theta) - KL(q(Z|X,c;\phi) \,\|\, p(Z|X,c;\phi))$$

$$= E_{Z\sim q(Z|X,c;\phi)}[\log p(X|Z,c;\theta) + \log p(Z|c) - \log q(Z|X,c;\phi)]$$

$$= E_{Z\sim q(Z|X,c;\phi)}[\log p(X|Z,c;\theta) - KL(q(Z|X,c;\phi) \,\|\, p(Z|c))]$$