Deep Learning Lab4 Report

陳以瑄 (ID: 313551047)

1 Introduction

In this lab, I implemented the CVAE to predict video sequences of a person, starting from the initial frame and subsequent pose labels. To enhance training effectiveness, I employed teacher forcing and the KL annealing trick. As shown in Section 3, I found that using a teacher forcing ratio (TFR) of 0.8 yielded the best validation performance. While applying KL annealing strategies led to higher PSNR compared to not using them. In the end, I trained the final model for the Kaggle competition using a TFR of 0.8 and the cyclical annealing strategy over 500 epochs.

2 Implementation details

2.1 How do you write your training/testing protocol

2.1.1 Training protocol

I follow the workflow shown in Figure 2(a) of the Spec. In the forward function, lines 2 to 4 handle the encoder part, encoding the input frame and pose label, and predicting a Gaussian posterior z. Lines 7 to 9 represent the decoder part, where the decoder fusion uses the reference frame, label, and posterior to generate fused features, which are then fed into the generator to produce the output frame.

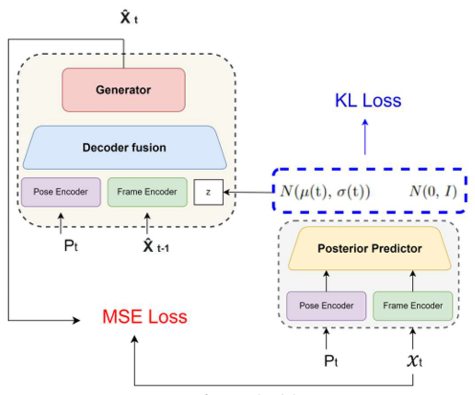


Figure 2. (a)

```
1  def forward(self,img_ref, img, label, mode = None):
2      img = self.frame_transformation(img)
3      label = self.label_transformation(label)
4      z, mu, logvar = self.Gaussian_Predictor(img, label)
5      if mode == "test":
6          z = torch.randn_like(z)  # Inference z~N(0,I)
7      img_ref = self.frame_transformation(img_ref)
8      fusion = self.Decoder_Fusion(img_ref, label, z)
9      output = self.Generator(fusion)
10     return output, mu, logvar
```

For each training video with 16 frames, we start from the first frame and use the forward function to predict the next frame. We then compute the MSE loss between the predicted frame and the ground truth frame, and also calculate the KL divergence between the predicted posterior distribution and a standard normal distribution. The total loss (MSE + KL) is used to perform backpropagation.

```python
def training_one_step(self, img, label, adapt_TeacherForcing):
    loss = 0
    self.optim.zero_grad()
    last = img[:,0]
    for i in range(1, self.train_vi_len):
        output, mu, logvar = self.forward(last, img[:,i], label[:,i])
        if adapt_TeacherForcing:
            # use ground truth as the refernece frame
            last = img[:,i].detach()
        else:
            # use the generated frame as the refernece frame
            last = output.detach()
        mse = self.mse_criterion(output, img[:,i])
        kl = kl_criterion(mu, logvar, self.batch_size) * \
            self.kl_annealing.get_beta()
        loss += mse + kl
    loss.backward()
    self.optimizer_step()
    return loss
```

For validation in the training stage, it start from the first frame of a 630-frame video and predict the next frames one by one. Unlike training, validation only uses the decoder part. So in the forward function, the posterior z is sampled from a standard normal distribution N(0,I) instead of being predicted. Another difference is that we don’t update the network during validation, so I added ”with torch.no_grad():” on line 3 to disable gradient computation.

```python
1  def val_one_step(self, img, label):
2      loss = 0
3      with torch.no_grad():
4          last = img[:,0]
5          for i in range(1, self.val_vi_len):
6              output, mu, logvar = self.forward(last, img[:,i], label[:,i],\
7                                                mode = "test")
8              last = output.detach()
9              mse = self.mse_criterion(output, img[:,i])
10             kl = kl_criterion(mu, logvar, self.batch_size)
11             loss += mse + kl
12         return loss
```

2.1.2 Testing protocol

I follow the workflow shown in Figure 2(b) of the Spec. Similar to validation during training, we only use the decoder part. Although we don’ t need to predict the posterior here, I still call the Gaussian_Predictor (line 4) to get the correct shape for z, then replace it with random values sampled from a standard normal distribution.

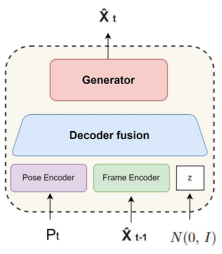


Figure 2. (b)

```python
1 def forward(self, img_ref, label):
2         img_ref = self.frame_transformation(img_ref)
3         label = self.label_transformation(label)
4         z, _, _ = self.Gaussian_Predictor(img_ref, label)
5         z = torch.randn_like(z)
6         fusion = self.Decoder_Fusion(img_ref, label, z)
7         output = self.Generator(fusion)
8         return output
```

In the inference stage, we don’ t have ground truth frames, so we can’ t calculate the loss. Starting from the first frame, we iteratively predict the next frame and collect all the results.

```python
def val_one_step(self, img, label, idx=0):
    img = img.permute(1, 0, 2, 3, 4)
    label = label.permute(1, 0, 2, 3, 4)
    decoded_frame_list = [img[0].cpu()]
    label_list = []

    # TODO
    img_ref = img[0]
    for i in range(1, 630):
        label_now = label[i]
        output = self.forward(img_ref, label_now)
        img_ref = output.detach()
        decoded_frame_list.append(output.cpu())
        label_list.append(output.cpu())
```

2.2 How do you implement reparameterization tricks

Since sampling from a distribution is not differentiable, we use the reparameterization trick to enable end-to-end training.
Using the property that if $X\ N(0, I)$, then $Y = aX + b\ N(b, a^2)$, we can re-write the sampling process in a differentiable way.
Given the mean μ and log-variance $logvar$, we:

1. Compute the standard deviation σ
2. Sample ϵ from a standard normal distribution
3. Calculate $z = \mu + \epsilon * \sigma$

```python
def reparameterize(self, mu, logvar):
    stdvar = torch.exp(logvar*0.5)
    eps = torch.randn_like(stdvar)
    z = mu + eps*stdvar
    return z
```

2.3 How do you set your teacher forcing strategy

At the early stage of training, the predictions of the model are not accurate. If we use the predicted frame as the next reference frame, it may lead to a garbage-in, garbage-out problem. So, during the first few epochs, we apply teacher forcing (lines 4-6), where we use the ground truth frame as the reference.

```python
1 def training_one_step(self, img, label, adapt_TeacherForcing):
2     for i in range(1, self.train_vi_len):
3         output, mu, logvar = self.forward(last, img[:,i], label[:,i])
4         if adapt_TeacherForcing:
5             # use ground truth as the refernece frame
6             last = img[:,i].detach()
7         else:
8             # use the generated frame as the refernece frame
9             last = output.detach()
```

As training progresses and the model becomes better at predicting, we can gradually switch to using the predicted frame as the reference. To do this, we schedule the teacher forcing rate, starting from 1 at the beginning and gradually decreasing it over time.

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

2.4 How do you set your kl annealing ratio

We have two losses during training—the MSE loss between the output frame and the ground truth, and the KL divergence between the posterior distribution and the standard normal. However, in CVAE, we should focus more on the decoder part. Therefore, we can apply a lower weight β to the KL loss.
There are three types of settings: without KL annealing, cyclical KL annealing, and monotonic KL annealing.

In the initialization part, for the KL annealing strategies, β starts from zero (line 7), while in the no-annealing setting, β is initialized and always set to 1 (line 8-9).

```python
1  class kl_annealing():
2      def __init__(self, args, current_epoch=0):
3          self.kl_anneal_type = args.kl_anneal_type
4          self.kl_anneal_cycle = args.kl_anneal_cycle
5          self.kl_anneal_ratio = args.kl_anneal_ratio
6          self.current_epoch = current_epoch
7          self.beta = 0.0
8          if self.kl_anneal_type=="none":
9              self.beta = 1.0
10     def get_beta(self):
11         return self.beta
```

The update rule of monotonic strategy: $\beta \leftarrow \beta + \text{update ratio} \times \frac{\text{current epoch}}{\text{epochs of KL cycle}}$, and the value is bounded by 1.
The update rule of cyclical strategy: $\beta = \text{update ratio} \times \frac{\text{current epoch mod (epochs of KL cycle)}}{\text{epochs of KL cycle}}$, and the value is bounded by 1.

```python
def update(self):
    self.current_epoch+=1
    if self.kl_anneal_type=="Monotonic":
        self.beta = min(1.0,self.beta+ self.kl_anneal_ratio*\
                    self.current_epoch/self.kl_anneal_cycle)
    elif self.kl_anneal_type=="Cyclical":
        self.beta = self.frange_cycle_linear(start=0.0, stop=1.0)

def frange_cycle_linear(self, start=0.0, stop=1.0):
    mod = (self.current_epoch % self.kl_anneal_cycle)
    diff = (stop-start)* mod /self.kl_anneal_cycle
    return min(1.0, start + self.kl_anneal_ratio* diff)
```

3 Analysis & Discussion

3.1 Analysis of teacher forcing ratio

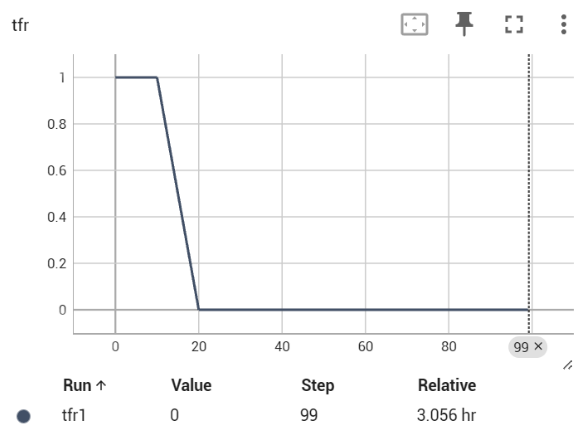
3.1.1 Settings

I tried three different initial teacher forcing rates (tfr): [1.0, 0.8, 0.5]
All other hyperparameters were kept the same:

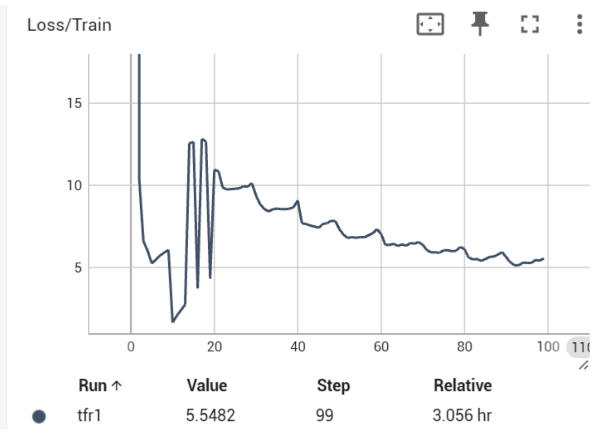
- Total epochs: 100
- Initial learning rate: 0.001
- Learning rate scheduler: MultiStepLR with milestones = [2, 5] and gamma = 0.1

- KL annealing type: Cyclical, with a cycle of 10 and a KL ratio of 1
- TFR scheduling: tfr_sde = 10, tfr_d_step = 0.1
- Random seed: 100

3.1.2 Case1: tfr = 1



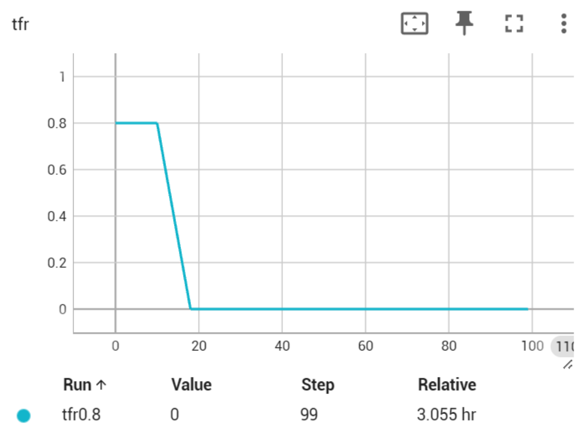
Run ↑	Value	Step	Relative
● tfr1	0	99	3.056 hr



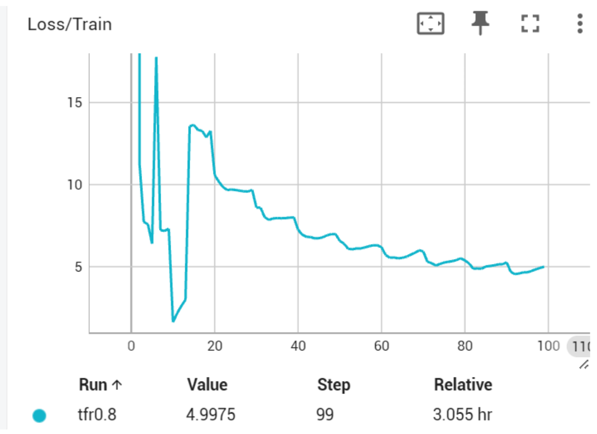
Run ↑	Value	Step	Relative
● tfr1	5.5482	99	3.056 hr

As the plot shows, during the first 10 epochs, when the ground truth frame is always used as the reference frame, the training loss decreases rapidly. However, between epochs 10 and 20, the loss fluctuates significantly. This is likely because the reference frame can be either the ground truth and the predicted frame, and since the predictions are still inaccurate at this stage, the loss varies considerably. After teacher forcing is turned off at epoch 20, the training loss starts to decrease more steadily.

3.1.3 Case2: tfr = 0.8



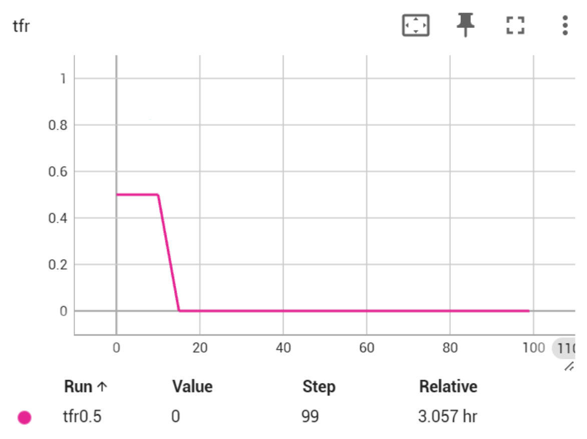
Run ↑	Value	Step	Relative
● tfr0.8	0	99	3.055 hr



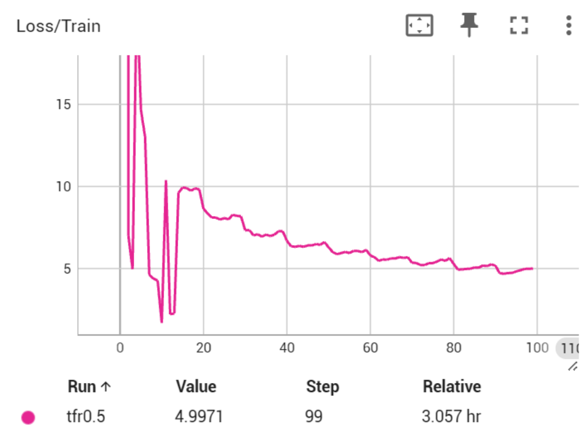
Run ↑	Value	Step	Relative
● tfr0.8	4.9975	99	3.055 hr

When setting the TFR to 0.8 for the first 10 epochs, we observe that the training loss fluctuates significantly. This is because the reference frame has an 80% chance of being the ground truth and a 20% chance of being a potentially inaccurate prediction. Due to this mix, the model occasionally relies on poor predictions, leading to instability in the loss. However, since the initial TFR is slightly lower, the chance of using predicted frames as references increases earlier in training. As a result, the model appears to start relying more on predicted frames around the 14th epoch, then the loss begins to decrease more steadily.

3.1.4 Case3: tfr = 0.5



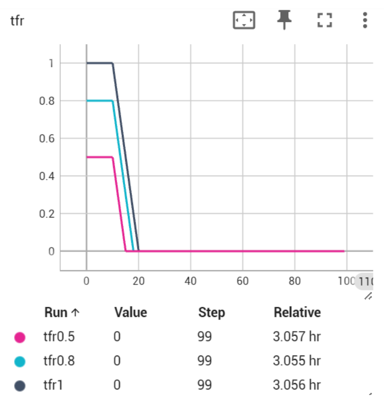
	Run ↑	Value	Step	Relative
●	tfr0.5	0	99	3.057 hr



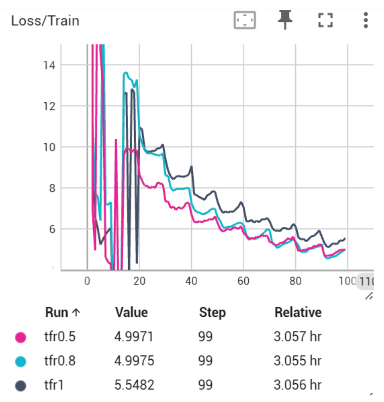
	Run ↑	Value	Step	Relative
●	tfr0.5	4.9971	99	3.057 hr

When setting the initial TFR to 0.5, the training loss fluctuates more during the first 10 epochs compared to the case with TFR = 1. However, since the model uses predicted frames as reference from the very beginning, the loss drops to a slightly lower value—compared to the previous two cases—after the TFR decreases later in training.

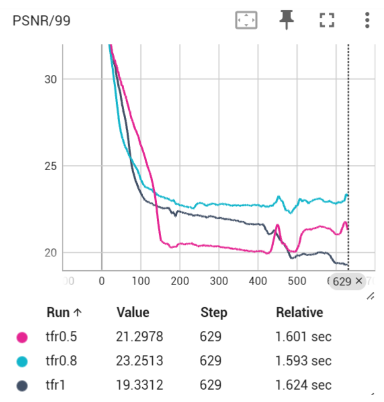
3.1.5 Analysis & Compare with the loss curve



	Run ↑	Value	Step	Relative
●	tfr0.5	0	99	3.057 hr
●	tfr0.8	0	99	3.055 hr
●	tfr1	0	99	3.056 hr



	Run ↑	Value	Step	Relative
●	tfr0.5	4.9971	99	3.057 hr
●	tfr0.8	4.9975	99	3.055 hr
●	tfr1	5.5482	99	3.056 hr



	Run ↑	Value	Step	Relative
●	tfr0.5	21.2978	629	1.601 sec
●	tfr0.8	23.2513	629	1.593 sec
●	tfr1	19.3312	629	1.624 sec

At the end of the teacher forcing stage (epoch 20), the setting with a higher initial TFR has a higher loss, likely because the model is less familiar with using predicted frames as reference. However, in the following epochs, the model with TFR = 0.8 shows a faster decrease in loss compared to the other two settings, and ultimately has the highest average validation PSNR in epoch 100. This may be because the TFR = 0.8 setting strikes a good balance, using ground truth frames helps mitigate the ”garbage in, garbage out” problem, while still exposing the model to its own predictions without letting it overfit or mistakenly trust it’s prediction too early.

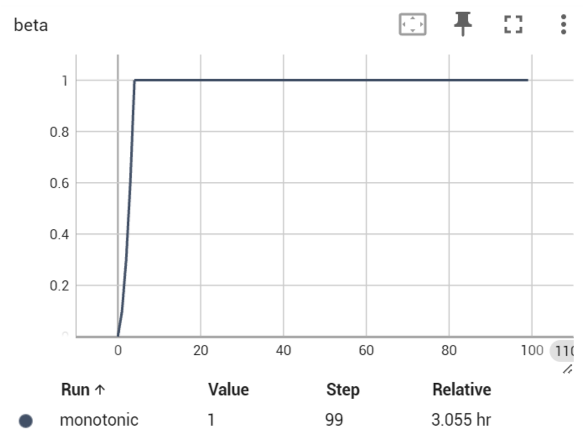
3.2 Analysis of KL annealing

3.2.1 Settings

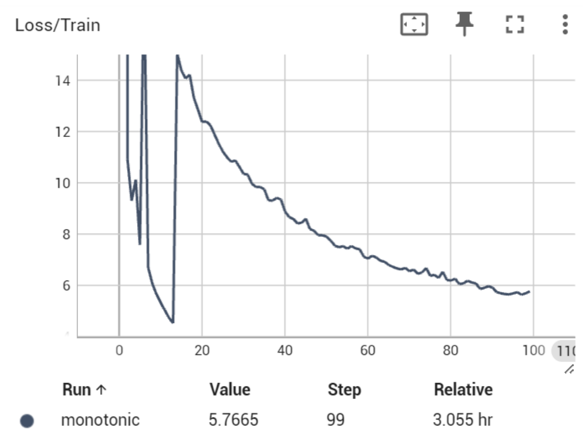
All hyperparameters were kept the same:

- Total epochs: 100
- Initial learning rate: 0.001
- Learning rate scheduler: MultiStepLR with milestones = [2, 5] and gamma = 0.1
- TFR scheduling: tfr = 0.8, tfr_sde = 10, tfr_d_step = 0.1
- Random seed: 42

3.2.2 Case1: With KL annealing (Monotonic), cycle = 10, rate = 1

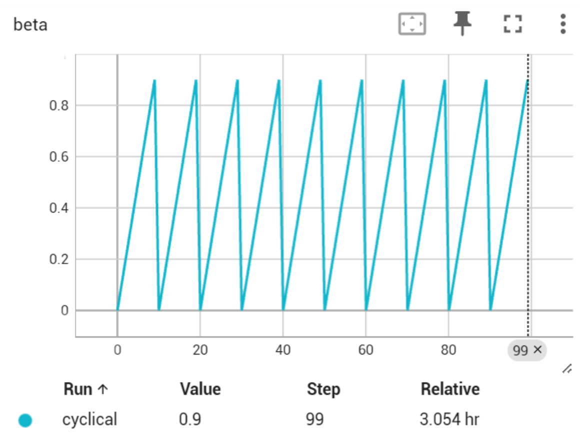


Run ↑	Value	Step	Relative
● monotonic	1	99	3.055 hr

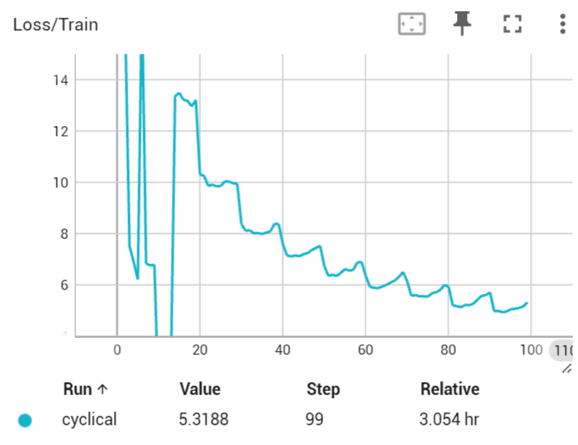


Run ↑	Value	Step	Relative
● monotonic	5.7665	99	3.055 hr

3.2.3 Case2: With KL annealing (Cyclical), cycle = 10, rate = 1

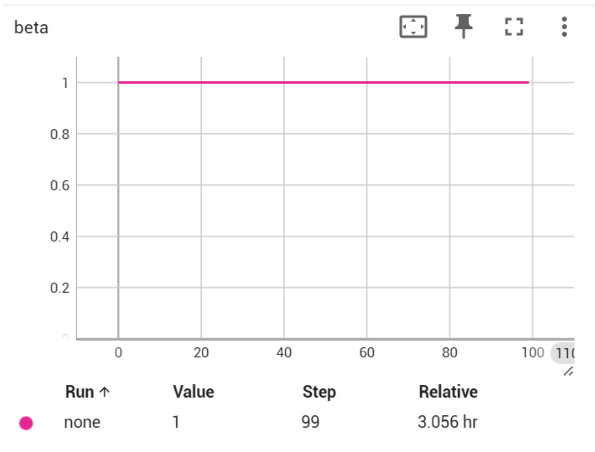


Run ↑	Value	Step	Relative
● cyclical	0.9	99	3.054 hr

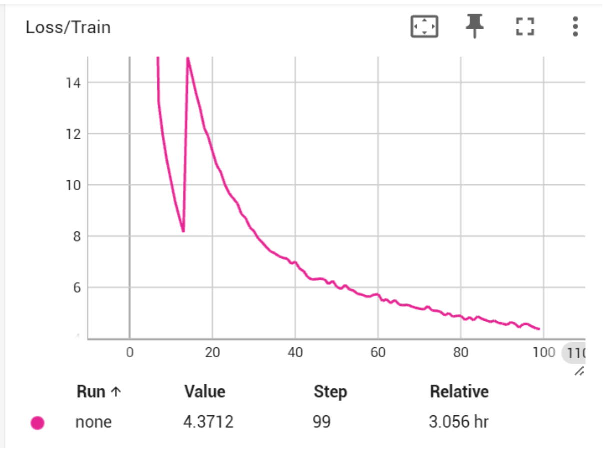


Run ↑	Value	Step	Relative
● cyclical	5.3188	99	3.054 hr

3.2.4 Case3: Without KL annealing

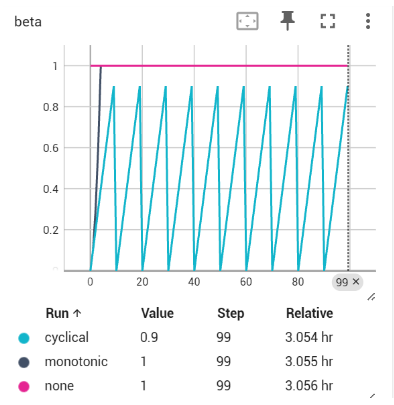


	Run ↑	Value	Step	Relative
●	none	1	99	3.056 hr

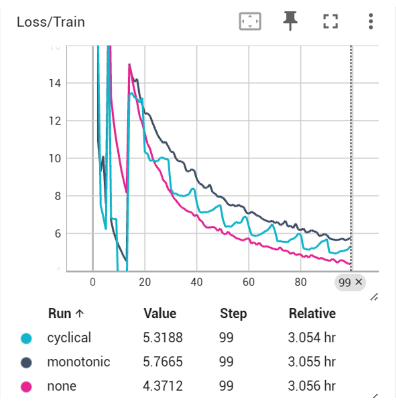


	Run ↑	Value	Step	Relative
●	none	4.3712	99	3.056 hr

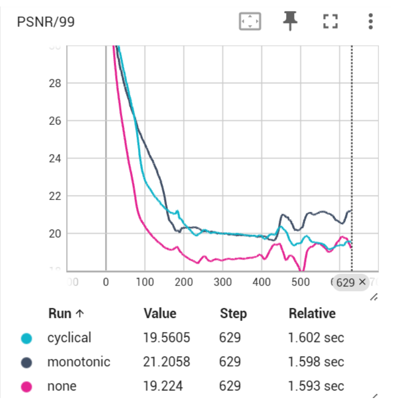
3.2.5 Analysis & Compare with the loss curve



	Run ↑	Value	Step	Relative
●	cyclical	0.9	99	3.054 hr
●	monotonic	1	99	3.055 hr
●	none	1	99	3.056 hr



	Run ↑	Value	Step	Relative
●	cyclical	5.3188	99	3.054 hr
●	monotonic	5.7665	99	3.055 hr
●	none	4.3712	99	3.056 hr



	Run ↑	Value	Step	Relative
●	cyclical	19.5605	629	1.602 sec
●	monotonic	21.2058	629	1.598 sec
●	none	19.224	629	1.593 sec

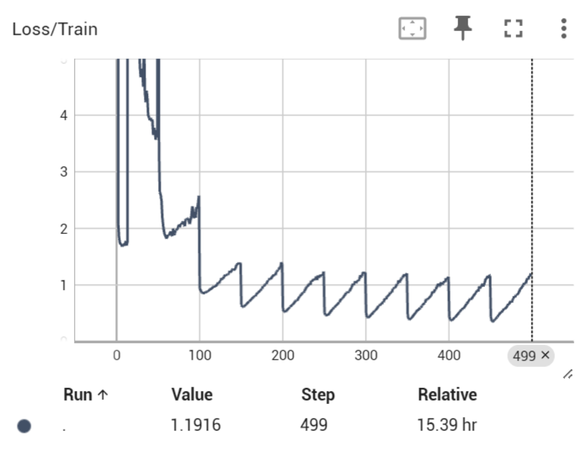
Since the KL annealing strategy affects how the loss is calculated, I think it's more accurate to compare the final validation PSNR. As we can see, the two cases that used KL annealing had higher average PSNR on the validation frames than the one that didn' t, showing that applying KL annealing can improve training performance.

3.3 PSNR-per-frame diagram in the validation dataset

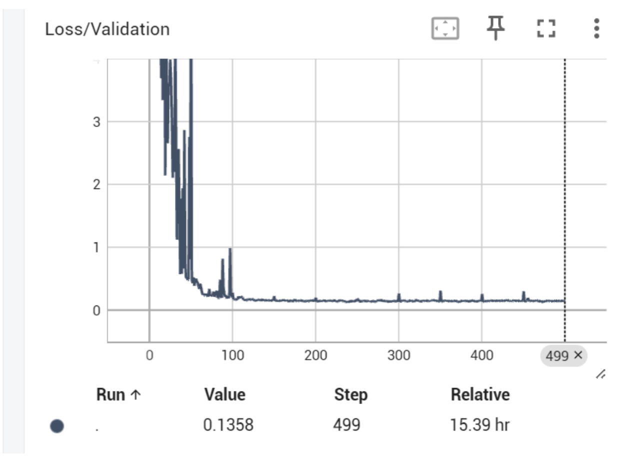
3.3.1 Settings

- Total epochs: 500
- Initial learning rate: 0.001
- LR scheduler: MultiStepLR with milestones = [2, 100] and gamma = 0.15
- TFR scheduling: tfr = 0.8, tfr_sde = 10, tfr_d_step = 0.1
- KL annealing type: Cyclical, with a cycle of 50 and a KL ratio of 1

3.3.2 Training and validation loss



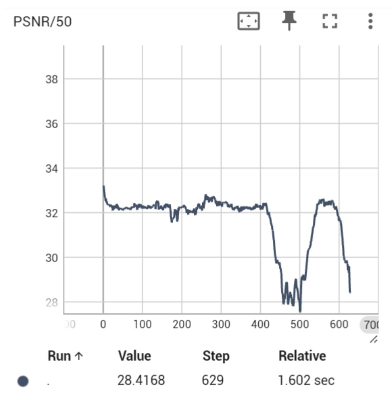
Run ↑	Value	Step	Relative
● .	1.1916	499	15.39 hr



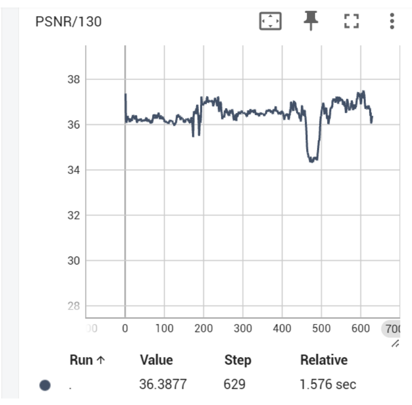
Run ↑	Value	Step	Relative
● .	0.1358	499	15.39 hr

From the figure, we can see that the validation loss drops and converges around 100 epochs. Therefore, I chose to show the PSNR results at epochs 50, 130, and 499.

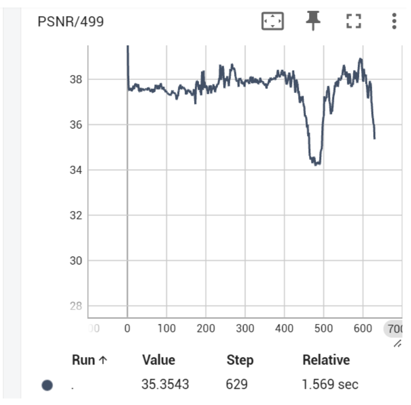
3.3.3 PSNR



Run ↑	Value	Step	Relative
● .	28.4168	629	1.602 sec



Run ↑	Value	Step	Relative
● .	36.3877	629	1.576 sec



Run ↑	Value	Step	Relative
● .	35.3543	629	1.569 sec

Although the values are different, the more training epochs the higher PSNR, the overall trend is quite similar. We can see that the PSNR is lower between frames 450—500 and in the last 10 frames. After checking the validation set, I found that in these two ranges, the person' s hand is raised above the head and partially out of frame. I think this might be the cause of the lower PSNR.