Lab4 : Conditional VAE for Video Prediction

1)Introduction (5%):

This lab's topic is "video prediction", we use conventional VAE and the dataset with videos and human pose skeletons to generate the image. Also, we use a lot of the strategy try to make the result better. Finally, we use the model to generate the video base on a picture with multiple human pose skeletons.

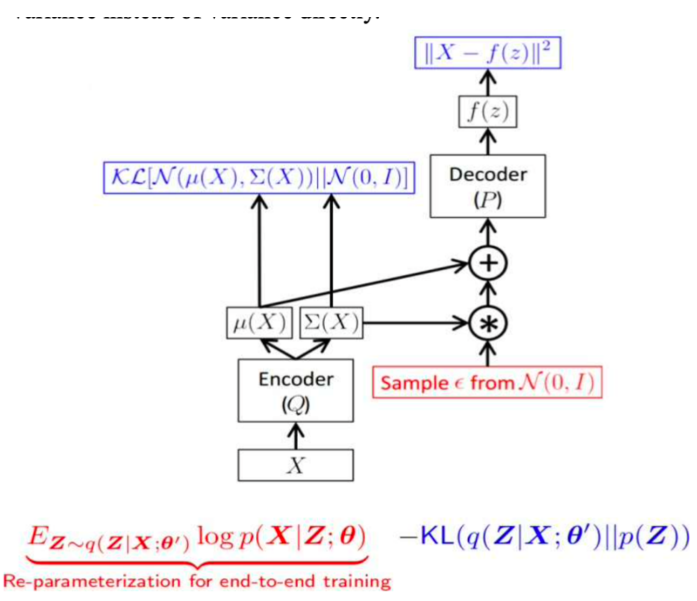
2) Implementation Details (30%):

i. How do you write your training/testing protocol:

cVAE:

Conditional Variational Autoencoder (CVAE) is an extension of the traditional Variational Autoencoder (VAE), which introduces a conditional variable y to enable the model to learn to generate data under specific conditions.

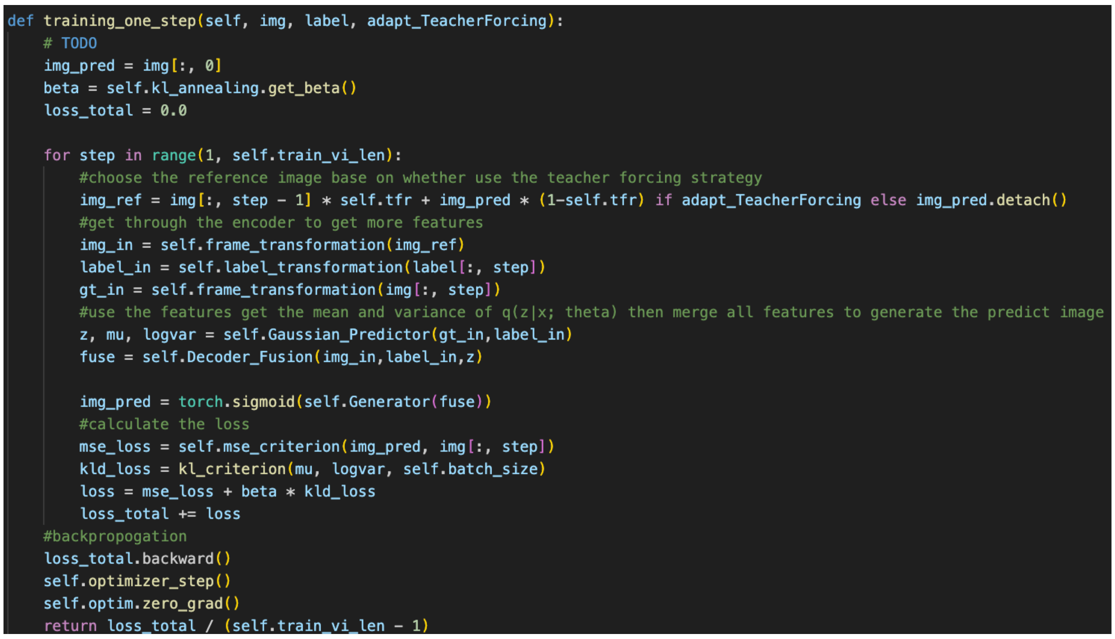
CVAE introduces a latent variable z to model the conditional distribution p(x ∣ y), using an approximate posterior q(z ∣ x, y) to replace the intractable true posterior p(z ∣ x, y). The ELBO is derived as the training objective, combining reconstruction loss and KL divergence to enable data generation conditioned on y.


$\|X - f(z)\|^2$
$f(z)$
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)]$
Decoder (P)
$+$
$*$
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
Sample ϵ from $\mathcal{N}(0, I)$
X
$E_{Z \sim q(Z|X;\theta')} \log p(X|Z;\theta)$ $-\text{KL}(q(Z|X;\theta')\|p(Z))$
Re-parameterization for end-to-end training

Training:

Reference is a really important part in cVAE, as it guides the generation to stay consistent with the target appearance. We use the label and frame encoder to get the features of the human pose skeleton and the image. Also, we use a gaussian predictor to estimate the parameters (mean and variance) of the latent distribution q(z | x; theta) by the label and frame.

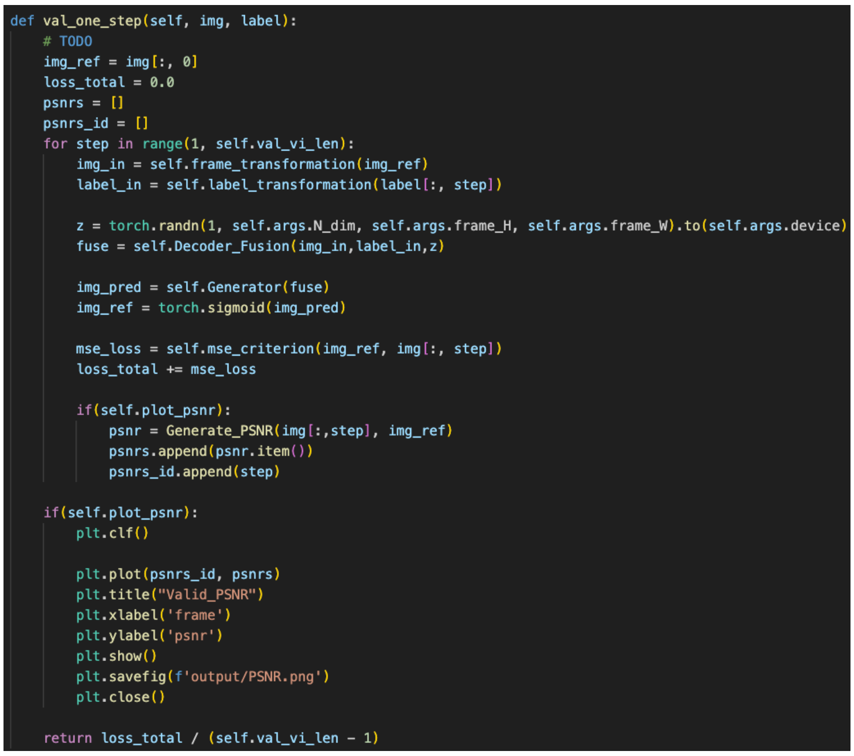
After getting all the component, we put all of them into the fuse decoder to merge all the features, then put it into generator to get the image like the reference. Finally, we get the loss by add the kl and mse criterion of the prediction and reference.



```
def training_one_step(self, img, label, adapt_TeacherForcing):
    # TODO
    img_pred = img[:, 0]
    beta = self.kl_annealing.get_beta()
    loss_total = 0.0

    for step in range(1, self.train_vi_len):
        #choose the reference image base on whether use the teacher forcing strategy
        img_ref = img[:, step - 1] * self.tfr + img_pred * (1-self.tfr) if adapt_TeacherForcing else img_pred.detach()
        #get through the encoder to get more features
        img_in = self.frame_transformation(img_ref)
        label_in = self.label_transformation(label[:, step])
        gt_in = self.frame_transformation(img[:, step])
        #use the features get the mean and variance of q(z|x; theta) then merge all features to generate the predict image
        z, mu, logvar = self.Gaussian_Predictor(gt_in,label_in)
        fuse = self.Decoder_Fusion(img_in,label_in,z)

        img_pred = torch.sigmoid(self.Generator(fuse))
        #calculate the loss
        mse_loss = self.mse_criterion(img_pred, img[:, step])
        kld_loss = kl_criterion(mu, logvar, self.batch_size)
        loss = mse_loss + beta * kld_loss
        loss_total += loss
    #backpropogation
    loss_total.backward()
    self.optimizer_step()
    self.optim.zero_grad()
    return loss_total / (self.train_vi_len - 1)
```

Validation:

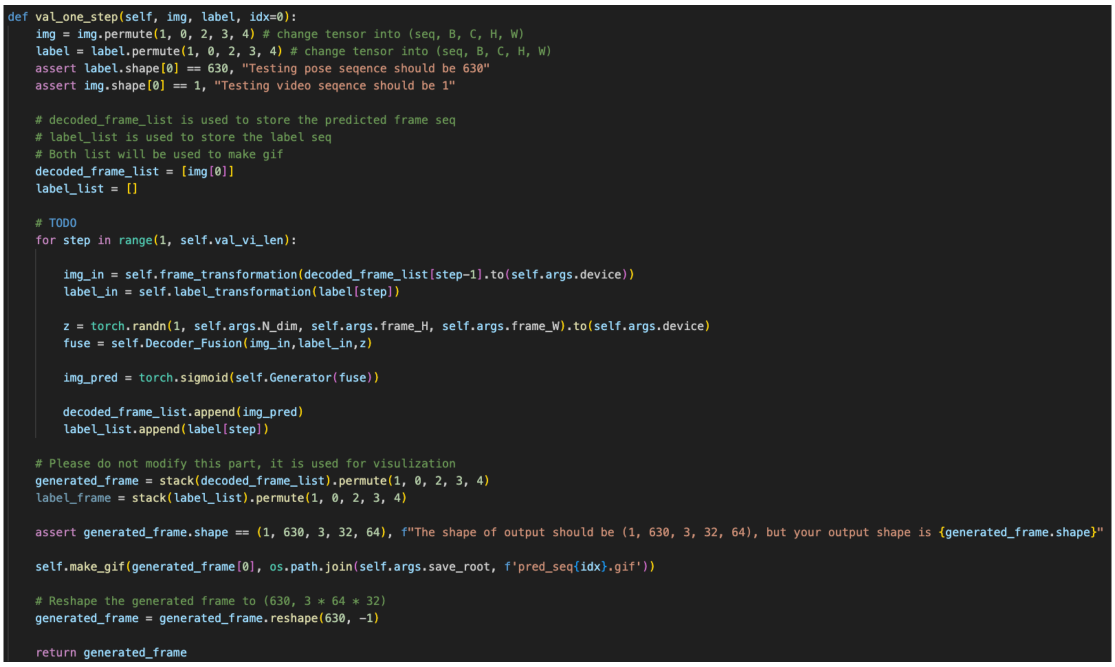
This one is similar to the training process; however, we need to take val_vi_len as the length of the video. Since we only use decoder in validation. So rather than use gaussian, we generate the z from normal distribution ourselves.



```python
def val_one_step(self, img, label):
    # TODO
    img_ref = img[:, 0]
    loss_total = 0.0
    psnrs = []
    psnrs_id = []
    for step in range(1, self.val_vi_len):
        img_in = self.frame_transformation(img_ref)
        label_in = self.label_transformation(label[:, step])

        z = torch.randn(1, self.args.N_dim, self.args.frame_H, self.args.frame_W).to(self.args.device)
        fuse = self.Decoder_Fusion(img_in,label_in,z)

        img_pred = self.Generator(fuse)
        img_ref = torch.sigmoid(img_pred)

        mse_loss = self.mse_criterion(img_ref, img[:, step])
        loss_total += mse_loss

        if(self.plot_psnr):
            psnr = Generate_PSNR(img[:,step], img_ref)
            psnrs.append(psnr.item())
            psnrs_id.append(step)

    if(self.plot_psnr):
        plt.clf()

        plt.plot(psnrs_id, psnrs)
        plt.title("Valid_PSNR")
        plt.xlabel('frame')
        plt.ylabel('psnr')
        plt.show()
        plt.savefig(f'output/PSNR.png')
        plt.close()

    return loss_total / (self.val_vi_len - 1)
```

Testing:

Similar to validation, we change the dimension to validate and turn the image and human pose skeleton into the GIF.

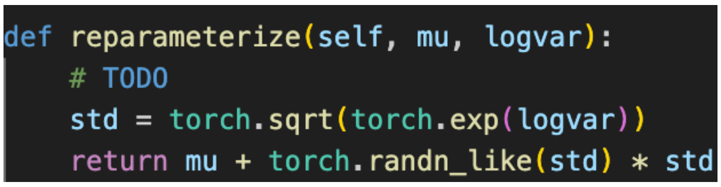


```python
def val_one_step(self, img, label, idx=0):
    img = img.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    label = label.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    assert label.shape[0] == 630, "Testing pose seqence should be 630"
    assert img.shape[0]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

    # TODO
    for step in range(1, self.val_vi_len):

        img_in = self.frame_transformation(decoded_frame_list[step-1].to(self.args.device))
        label_in = self.label_transformation(label[step])

        z = torch.randn(1, self.args.N_dim, self.args.frame_H, self.args.frame_W).to(self.args.device)
        fuse = self.Decoder_Fusion(img_in,label_in,z)

        img_pred = torch.sigmoid(self.Generator(fuse))

        decoded_frame_list.append(img_pred)
        label_list.append(label[step])

    # 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, 3, 32, 64), but your output shape is {generated_frame.shape}"

    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
```

ii. How do you implement reparameterization tricks:

The problem of the standard cVAE is that the process of the cVAE will disturb the backpropagation. Since the features we give to the decoder are sampling from the latent distribution using a non-deterministic operation by using the non-deterministic operation, giving the uncertainty that will make propagation harder. So, we use the reparameterization tricks to solve the problem.

We get the mean of the latent variable and the standard deviation by reverting the log standard deviation from the gaussian distribution to the original standard deviation. Then multiply the standard deviation with a random number, then add it together to achieve the effect of the original distribution.

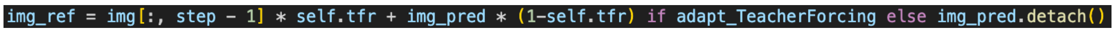


```
def reparameterize(self, mu, logvar):
    # TODO
    std = torch.sqrt(torch.exp(logvar))
    return mu + torch.randn_like(std) * std
```

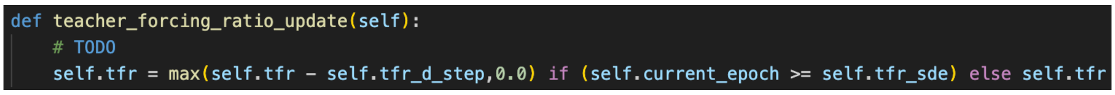
iii. How do you set your teacher forcing strategy:

In this part, I implement the teacher forcing strategy when we use the teacher forcing strategy, when we use the teacher force strategy, instead of use inference to generating frames, it will use the ground truth from the training data as the reference image. So that when the early generated frames are not accurate enough, we can avoid the problem of error propagation and accumulation.

In this part, when we use the teacher forcing strategy, we use the method of mixing strategy to make it gradually decrease to the original prediction-only mode, allowing the model to slowly adapt to relying on its own outputs instead of the ground truth. To achieve this goal, after the self.tfr_sde epoch. I'll decrease the ratio by self.tfr_d_step to gradually back to the original inference.



```
img_ref = img[:, step - 1] * self.tfr + img_pred * (1-self.tfr) if adapt_TeacherForcing else img_pred.detach()
```



```
def teacher_forcing_ratio_update(self):
    # TODO
    self.tfr = max(self.tfr - self.tfr_d_step,0.0) if (self.current_epoch >= self.tfr_sde) else self.tfr
```

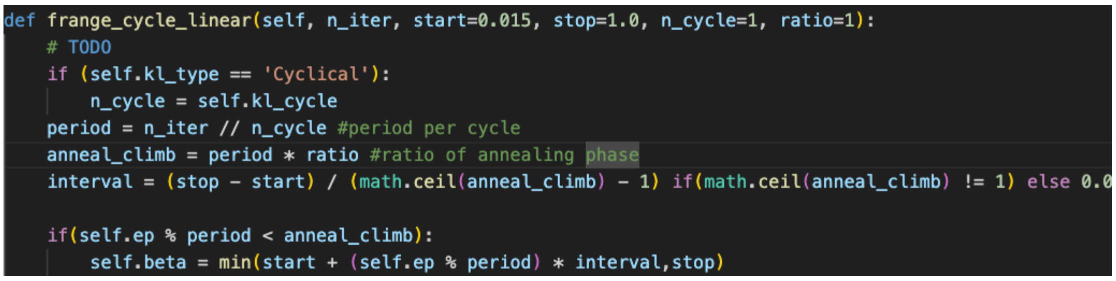
iv. How do you set your kl annealing ratio:

Since the decoder of the cVAE is auto-regressive, it can refer to the previous frame to generate the current output. The KL term may dominate early in training, causing the decoder to ignore the latent variables and resulting in an uninformative latent space. So we use KL annealing to solve the problem by gradually increasing the weight of the KL divergence term during training, the model is encouraged to make meaningful use of the latent space while still focusing on reconstruction in the early stages.

In this part, we first decide the type of the KL-annealing method to decide the cycle of annealing (In fact, Monotonic annealing is a variant of Cyclical annealing which only has one cycle in whole training process). Then we decide the period of a cycle and the ratio of two phases:

1st: Annealing phase: Reset the beta to the start value. Then we will do annealing climbing to climb from the start to 1.0.

2nd: Fixed phase: Beta remains at 1 to ensure that the model fully incorporates the KL divergence, encouraging the latent space to follow the desired prior distribution and improving the generalization ability of the model.



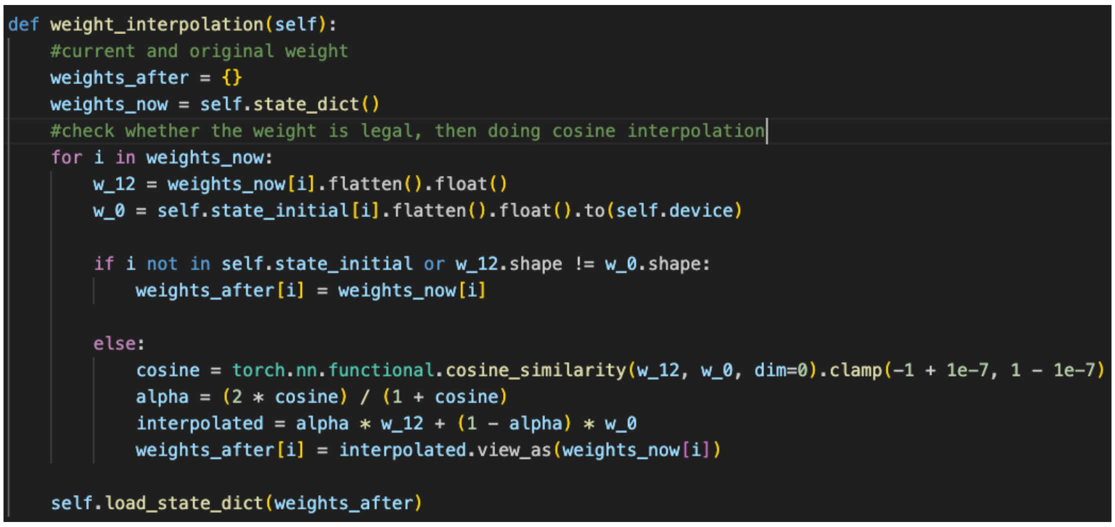
```
def frange_cycle_linear(self, n_iter, start=0.015, stop=1.0, n_cycle=1, ratio=1):
    # TODO
    if (self.kl_type == 'Cyclical'):
        n_cycle = self.kl_cycle
    period = n_iter // n_cycle #period per cycle
    anneal_climb = period * ratio #ratio of annealing phase
    interval = (stop - start) / (math.ceil(anneal_climb) - 1) if(math.ceil(anneal_climb) != 1) else 0.0

    if(self.ep % period < anneal_climb):
        self.beta = min(start + (self.ep % period) * interval,stop)
```

v. cosine weight interpolation:

In the lab I also implement the other strategy. I use the following formula to fine-tune my model’s weight:

$$\mathbf{w}_H = \frac{2\cos\theta}{1+\cos\theta} \cdot \mathbf{w}_{12} + \left(1 - \frac{2\cos\theta}{1+\cos\theta}\right) \cdot \mathbf{w}_0$$

The code will perform a cosine-based interpolation between the model's current weights and the initial weights to stabilize training by balance the learned parameters with the initial ones. If we do not find it in the initial state or the shape is not fit, we will assign the original weight to the current one.



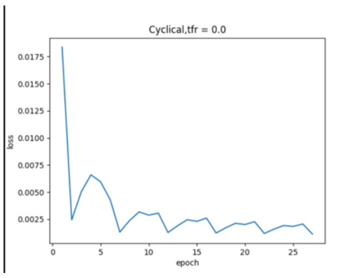
```python
def weight_interpolation(self):
    #current and original weight
    weights_after = {}
    weights_now = self.state_dict()
    #check whether the weight is legal, then doing cosine interpolation
    for i in weights_now:
        w_12 = weights_now[i].flatten().float()
        w_0 = self.state_initial[i].flatten().float().to(self.device)

        if i not in self.state_initial or w_12.shape != w_0.shape:
            weights_after[i] = weights_now[i]

        else:
            cosine = torch.nn.functional.cosine_similarity(w_12, w_0, dim=0).clamp(-1 + 1e-7, 1 - 1e-7)
            alpha = (2 * cosine) / (1 + cosine)
            interpolated = alpha * w_12 + (1 - alpha) * w_0
            weights_after[i] = interpolated.view_as(weights_now[i])

    self.load_state_dict(weights_after)
```

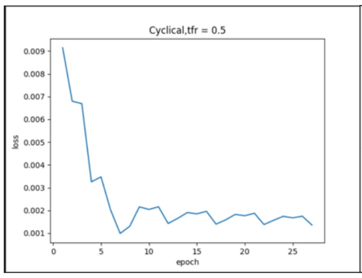
3) Analysis & Discussion (25% + Bonus 10%):

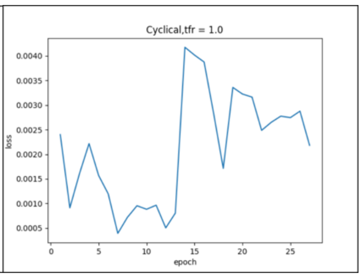
i. Analyzing with different Teacher forcing ratio:

Command: python3 Trainer.py --DR LAB4_Dataset --save_root check_points --fast_train --tfr 0.0 --num_epoch 27 --fast_train_epoch 32 --kl_anneal_ratio 0.6 --kl_anneal_cycle 5 --plot_training_loss

(Change the value of the tfr to get other kind of plots)





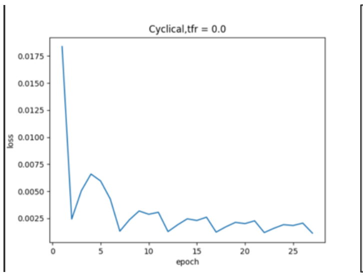


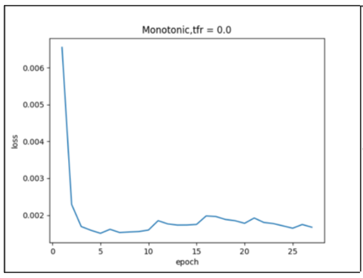
We can see that after we use the teacher force methods. The loss curve will have different degree of oscillation. When tfr = 0.5 we can see that it does not follow the regularity of the kl_annealing at first, but it recovers then. But when the tfr = 1.0, the oscillations of the loss curve start getting more intense. This is because with a higher teacher forcing ratio, the model relies more heavily on ground truth inputs during training, making it less robust to its own prediction errors. However, it seems that the assist of the teacher forcing strategy is not obvious in this lab, probably because that the video we need to generate do not need that much resolution. So that the detail between the different frame is not that much. That’s why the teacher forcing learning do not help that much in this lab.

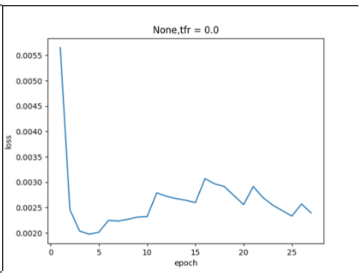
ii. Analyzing with different kl annealing methods:

Command: python3 Trainer.py --DR LAB4_Dataset --save_root check_points --fast_train --tfr 0.0 --num_epoch 27 --fast_train_epoch 32 --kl_anneal_ratio 0.6 --kl_anneal_cycle 5 --plot_training_loss --kl_anneal_type Monotonic

(Change the kl_anneal_type to get other kind of plots)





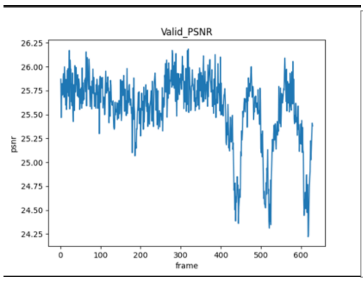


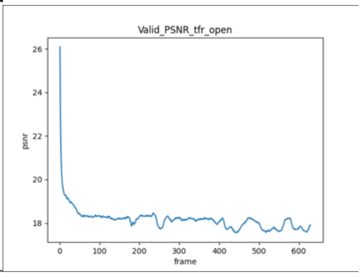
We can see that without kl_annealing, the loss curve is not stable and have greater loss value just as above mention. Then we see the cyclical and monotonic loss curve we can see that when the beta turn from 1 to 0 it would show segmented fluctuations. Also the monotonic curve seems converge faster than the cyclical one.

iii. Analyzing the PSNR:

Command: python3 Trainer.py --DR LAB4_Dataset --save_root check_points --fast_train --tfr 0.0 --num_epoch 27 --fast_train_epoch 32 --kl_anneal_ratio 0.6 --kl_anneal_cycle 5 --plot_psnr --fast_partial 0.0025

(Change the tfr to get other kind of plots)



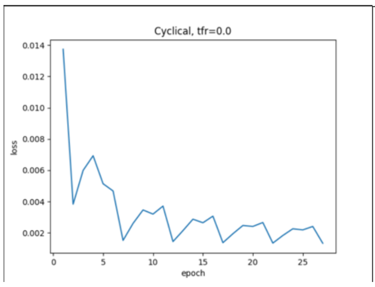


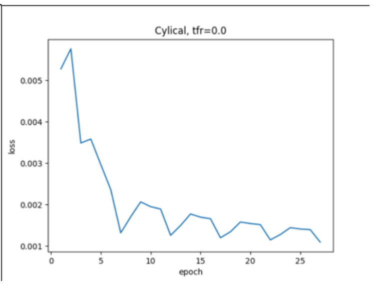
It is really interesting that we can observe the psnr curve with teacher forcing strategy is smoother than without teacher forcing strategy and without a lot of oscillation. However, we can see that the average PSNR score is lower than the one without teacher forcing strategy. This is because when teacher forcing is enabled, the model is fed ground truth (or a mixture) at each step. It means that the error will accumulate during prediction, causing smoother PSNR curves. However this will affect the model to learn how to recover from its own prediction errors, it could also make model struggling to generate accurate predictions based on its own previous outputs. That’s why the PSNR curve is lower.

iv. Analyzing the effect of cosine weight interpolation:

Command:
```
python3 Trainer.py --DR LAB4_Dataset --save_root check_points --fast_train --tfr
0.0 --num_epoch 27 --fast_train_epoch 32 --kl_anneal_ratio 0.6 --kl_anneal_cycle 5 --
plot_training_loss --no_weight_interpolation
```

(remove --no_weight_interpolation to perform fine tuning)





You can see when the segmented fluctuations when it approaches to five (we interpolate the weight when epoch % 5 == 0), however when it comes to training loss, it seems that it doesn’t have much effect on it. Seems like this method does not work on data like videos and pictures.

Reference:

[1] Mitchell Wortsman “Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time ICML” 2022