Deep Learning – Lab4

Conditional VAE for Video Prediction

I. Introduction

i. Lab Objective:

- Implement a Conditional Variational Autoencoder (CVAE) for video prediction. Utilize the image from the previous time step and the current time step's pose (label) as conditions, then to generate a prediction for the image at the current time step.

ii. VAE:

- VAE (Variational Autoencoder) is an evolution of the Autoencoder model. It aims to address the issue in the original AE model where there was a lack of predictability in the distribution of the encoded latent space, referred to as the 'z' space.
- To overcome this, the traditional encoder, which directly produced 'z', was modified. Instead of directly outputting 'z', the encoder now outputs a pair of parameters, 'mu' and 'sigma'. Subsequently, these parameters are employed to generate 'z' based on Gaussian distribution. The adjustment ensures that the distribution of latent space(z space) follows a Gaussian distribution.
- Consequently, upon completing the training of our model, we can directly perform Gaussian sampling from the latent space distribution to enable the decoder to carry out the prediction process.

iii. Dataset:

- Training dataset
  - **Video with about 16 frames.**
- Validation dataset
  - **Video with 630 frames.**
- Testing dataset
  - **6 video sequences are given. Each video sequence is 630 frames.**

II. Implementation Details

i. How do you write your training protocol

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

    kld = 0
    mse = 0
    out = img[0]

    for i in range(1, self.train_vi_len):
        label_feat = self.label_transformation(label[i])
        human_feat_hat = self.frame_transformation(img[i])
        if adapt_TeacherForcing:
            last_human_feat = self.frame_transformation(img[i - 1]) * 0.3 + self.frame_transformation(out) * 0.7
        else:
            last_human_feat = self.frame_transformation(out)

        z, mu, logvar = self.Gaussian_Predictor(human_feat_hat, label_feat)
        parm = self.Decoder_Fusion(last_human_feat, label_feat, z)
        out = self.Generator(parm)

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

    beta = kl_annealing.get_beta()
    loss = mse + beta*kld

    self.optim.zero_grad()
    loss.backward()
    self.optimizer_step()

    return loss
```

- The provided diagram represents my train_one_step() function.
- Initially, as the DataLoader processes data in the format (S, C, H, W), the generated data type is (B, S, C, H, W), considering a batch size of 'B'. However, since our network employs CNN implementation and CNN's input should be formatted as (B, C, H, W), we decompose the video into individual frames. Consequently, each frame is sequentially inputted into our network. This is the reason why we initiate the input data with a 'permute' operation.
- Throughout the training process, we can view the index within the inner loop as a temporal axis. In each step, we perform label encoding (embedding) on the label corresponding to that time point. Subsequently, depending on whether Teacher Forcing is applied, we use either the ground truth or the predicted image from the previous time step. Then, the image for the current time step is passed through the Encoder, yielding 'mu', 'sigma', and 'z'. This 'z' along with the two conditions obtained earlier are used for predicting the image at the current time step.
- The next remaining portion involves the calculation of the loss, which consists

of two components: KLD (Kullback-Leibler Divergence) and MSE (Mean Squared Error).

- **For the KLD component, the goal is to assess the difference between the distribution generated by our Encoder (mu and sigma) and the Gaussian distribution. This is why the kl_criterion() only takes 'mu' and 'sigma' as inputs, as our label corresponds to the Gaussian distribution's 0 and 1.**
- **Following that is the MSE component, which computes the difference between the images generated by our Decoder and the ground truth images.**
- **Finally, these two losses are added together based on KL Annealing, a technique that adjusts the weighting of the KL divergence loss during training, and then the model is updated accordingly.**

ii. How do you implement reparameterization tricks

```
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):
        sigma = torch.exp(0.5 * logvar)
        eps = Variable(logvar.data.new(logvar.size()).normal_())
        z = mu + eps * sigma
        return z

    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
```

- Reparameterization is a technique used in the VAE architecture. In the original VAE setup, the intention is to generate 'mu' and 'sigma' through the Encoder. Subsequently, 'z' is directly sampled using 'mu' and 'sigma'. This might appear reasonable and function smoothly during the forward process. However, issues arise during the backward process, as we cannot compute the gradient of the sampling operation.
- Here's where the Reparameterization trick comes into play. Initially, we generate a random variable 'r' from a standard Gaussian distribution. Then, we scale 'r' by 'sigma' and add 'mu' to it(The significance of this step lies in the fact that it

allows us to stretch the distribution according to 'sigma' and shift it according to 'mu'). This seemingly simple transformation enables us to perform differentiation without complications.

iii. How do you set your teacher forcing strategy

```
if adapt_TeacherForcing:
    last_human_feat = self.frame_transformation(img[i - 1])\
    * 0.3 + self.frame_transformation(out) * 0.7
else:
    last_human_feat = self.frame_transformation(out)
```

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

- Regarding the aspect of Teacher Forcing, I've opted for a ratio of 0.3 for ground truth and 0.7 for prediction. This choice is influenced by two considerations.

  - **Firstly, the training data consists of 16 frames, while the testing data contains 630 frames, resulting in a significant difference in sequence lengths. In such a scenario, I believe the model might struggle to predict longer videos. Enabling the model to directly predict with ground truth will exacerbates this issue.(It is because when the model has to predict long videos, it must ensure accuracy for each frame, as a poor prediction early on will affect subsequent frames. However, in the case of using Teacher Forcing, even if some predictions are subpar, it doesn't heavily impact the overall loss, as later frames still rely on the ground truth.) Thus, to avoid excessive reliance on ground truth, I've made this adjustment.**

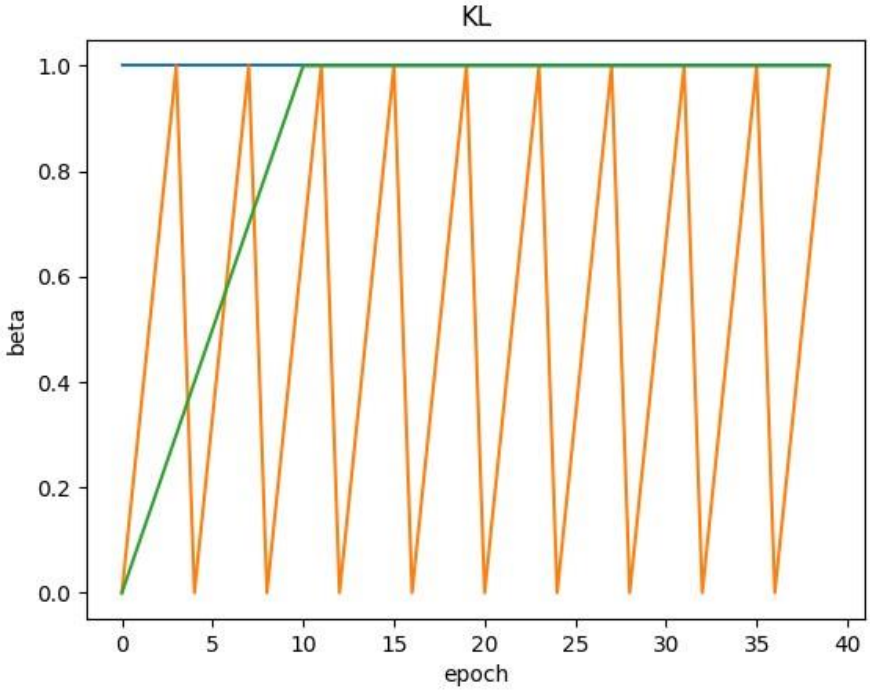
  - **Secondly, the code includes a default MultiStepLR scheduler, which rapidly decreases the learning rate in exponential steps. Often, Teacher Forcing hasn't concluded when the learning rate has already dropped significantly. Consequently, when Teacher Forcing ends and the model truly needs to rely on its own predictions, the learning rate has already plummeted to a level where learning becomes challenging.**

iv. How do you set your kl annealing ratio

```python
class kl_annealing():
    def __init__(self, args, current_epoch = 0):
        self.args = args
        self.current_epoch = current_epoch
        self.step = 0
        self.n_per_cycle, self.beta_rate = self.frange_cycle_linear\
        (cycle = self.args.kl_anneal_cycle, ratio = self.args.kl_anneal_ratio)
        self.kl_type = args.kl_anneal_type

    def update(self):
        self.current_epoch += 1
        self.step = self.current_epoch % self.n_per_cycle

    def get_beta(self):
        if self.kl_type == "None":
            return 1.0
        elif self.kl_type == "Cyclical":
            return min(1.0, self.step * self.beta_rate)
        elif self.kl_type == "Monotonic":
            return self.monotonic()

    def monotonic(self, max_beta = 1.0, beta_rate = 0.1):
        return min(max_beta, beta_rate * self.current_epoch)

    def frange_cycle_linear(self, min_beta = 0.0, max_beta = 1.0, cycle = 10, ratio = 1.0):
        beta_range = max_beta - min_beta
        n_per_cycle = self.args.num_epoch/cycle
        beta_rate = beta_range/((n_per_cycle - 1) * ratio)
        return n_per_cycle, beta_rate
```

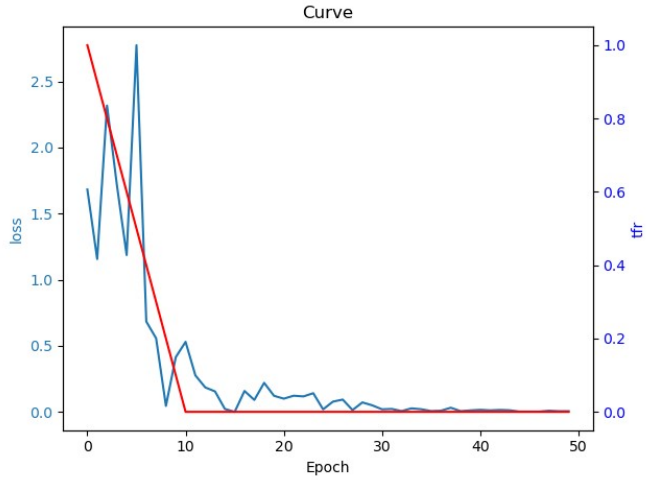
- Regarding KL Annealing, the approach involves different modes: 'None', 'Monotonic', and 'Cyclical'.
  - **For 'None', it simply returns 1, implying no adjustment to the beta parameter.**
  - **In the 'Monotonic' mode, the beta parameter is linearly increased over epochs.**
  - **The 'Cyclical' mode is a bit more complex.**
    - **First, I calculate how many epochs should be in each cycle (n_per_cycle) based on the total number of cycles and total epochs. Then, using the provided minimum and maximum values for beta, n_per_cycle, and the ratio, I compute the rate at which beta should increase (beta_rate) for each step within a cycle.**
    - **Next, I calculate the current step within the cycle using current_epoch % total_epoch. By multiplying the step with beta_rate, I can determine the current beta value for the step. It's important to note that due to the influence of the ratio, I need to ensure that beta doesn't exceed 1.**

- **Consequently, the final value returned is min(1.0, self.step * self.beta_rate).**



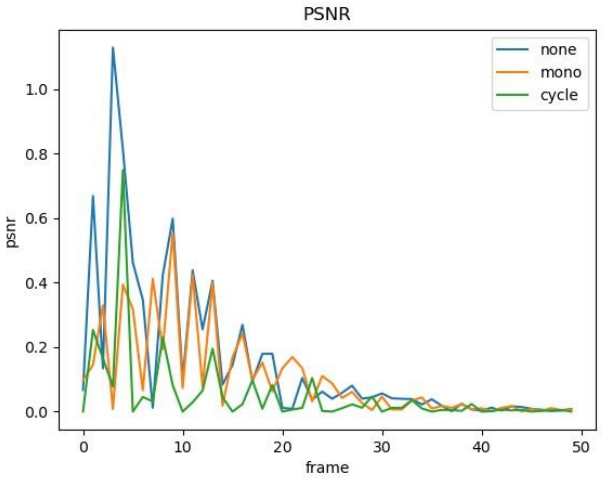
III. Analysis & Discussion

i. Plot Teacher forcing ratio



- In cases where there is a significant spike in the loss within the graph, it is mostly observed when Teacher Forcing (TF) is set to True. It's evident that the loss stabilizes more after the 10th epoch. This is because after the 10th epoch, the probability of TF being true is 0, meaning it is no longer being executed.

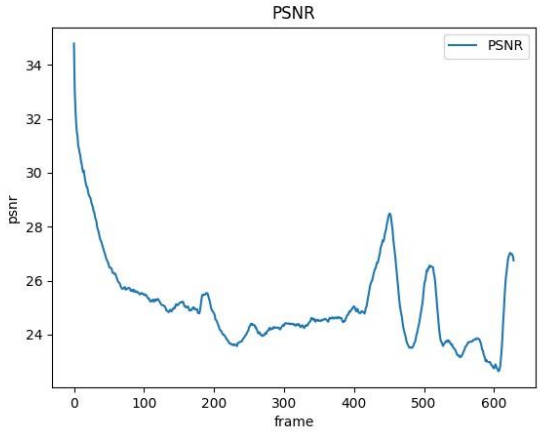
ii. Plot the loss curve while training with different settings.



- None: Since beta is always 1, the initial loss is the highest. Additionally, the learning process may exhibit more oscillations compared to the other two methods.
- Monotonic: Beta steadily increases until it reaches 1. As a result, the initial loss might appear to rise, but it will eventually stabilize and decrease.
- Cyclical: I've set a cycle every 5 epochs. Consequently, every 5 epochs, We can

observe a rise in loss. However, this approach can lead to a more stable convergence in the end.

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



- Initially, the PSNR (Peak Signal-to-Noise Ratio) appears high because the first image prediction is based on the ground truth [0]. Subsequently, as the prediction process continues, the accuracy gradually declines. When the prediction for a previous image isn't accurate, it affects the prediction for the subsequent one as well. As a result, the overall trend tends to show a decline in PSNR over time.