Lab3: Conditional VAE for video prediction

Shyang-En Weng 翁祥恩 413551036

April 14, 2025

1 Introduction

Video prediction refers to forecasting future frames in a video sequence based on past observations. The goal is to model a scene's spatial and temporal dynamics, capturing how objects move and interact over time. These models aim to generate realistic and temporally consistent predictions, handling complex motion patterns and long-term dependencies. In this lab, we will evaluate and show my implementation of Conditional VAE (cVAE) for implementation, input with video frame and detected pose to generate the next consecutive frame, then a video. The highlighted points of the implementation are shown below:

- I implemented the training and testing of conditional VAE for video prediction and successfully generated videos from a single frame with its detected pose, including Training/testing protocol, Reparameterization tricks, Teacher forcing strategy, and KL annealing.
- KL annealing and teacher forcing strategy are discussed and analyzed in this lab, including three different types of annealing schedulers and two different teacher forcing ratios.

2 Implementation Details

In this lab, we want to predict the next frame based on the previous frame and the currently detected pose. Moreover, teacher forcing and KL annealing are utilized to assist the training.

2.1 Training/testing protocol

This lab is built on [1] and [2]. The goal of training the conditional Variational Autoencoder (cVAE) for video prediction is to model the difference between the predicted and current frames, guided by the detected pose information. To begin with, the first frame is excluded from the prediction loop, as it serves as the starting frame and does not require prediction. The model then estimates a latent representation (z) based on the transformation between the previous frame

and the current pose label, effectively capturing the target' s motion in a compressed form. This latent code, along with the previous frame and current pose features, is used to predict the current frame. The training process is supervised using a combination of loss functions: the mean squared error (MSE) between the predicted and ground truth current frames, and the Kullback-Leibler (KL) divergence to regularize the latent space, encouraging it to align with a prior distribution in Gaussian distribution between frames and pose labels. During testing, the latent vector is randomly sampled, allowing the image character to exhibit diverse movements and poses. The training, validation, and testing protocols are shown in Listings 1, 2, and 3, respectively.

Listing 1: Training protocol.

```
def training_one_step(self, img, label, adapt_TeacherForcing):
    # TODO
    # raise NotImplementedError

    total_kl_loss = 0
    total_mse_loss = 0

    x_hat = img[:, 0]
    for i in range(1, self.args.train_vi_len):
        if adapt_TeacherForcing:
            img_i = img[:, i-1]
        else:
            img_i = x_hat.detach()
        label_i = label[:, i]
        img_current = img[:, i]

        frame_feature = self.frame_transformation(img_i)
        current_frame_feature = self.frame_transformation(
            img_current)
        label_feature = self.label_transformation(label_i)

        z, mu, logvar = self.Gaussian_Predictor(
            current_frame_feature, label_feature)
        feature = self.Decoder_Fusion(frame_feature, label_feature, z)
        x_hat = self.Generator(feature)

        kl_loss = kl_criterion(mu, logvar, self.batch_size).to(mu.device)
        kl_loss = kl_loss.mean()

        mse_loss = self.mse_criterion(x_hat, img_current)
        total_mse_loss += mse_loss
        total_kl_loss += kl_loss

    self.optim.zero_grad()
```

```
    loss = total_mse_loss + total_kl_loss * self.kl_annealing.get_beta()
    loss.backward()
    self.optimizer_step()
    return loss
```

Listing 2: Validation protocol.

```
def val_one_step(self, img, label):
    # TODO
    # raise NotImplementedError

    total_kl_loss = 0
    total_mse_loss = 0
    total_psnr = 0

    x_hat = img[:, 0]
    for i in range(1, self.args.train_vi_len):
        img_i = x_hat.detach()
        label_i = label[:, i]
        img_current = img[:, i]

        frame_feature = self.frame_transformation(img_i)
        current_frame_feature = self.frame_transformation(img_current)
        label_feature = self.label_transformation(label_i)

        z, mu, logvar = self.Gaussian_Predictor(
            current_frame_feature, label_feature)

        feature = self.Decoder_Fusion(frame_feature, label_feature, z)
        x_hat = self.Generator(feature)

        kl_loss = kl_criterion(
            mu, logvar, self.batch_size).to(mu.device)

        mse_loss = self.mse_criterion(x_hat, img_current)

        psnr = Generate_PSNR(x_hat, img_current)

        total_psnr += psnr
        total_mse_loss += mse_loss

        total_kl_loss += kl_loss
    total_mse_loss /= (self.args.train_vi_len - 1)
    total_kl_loss /= (self.args.train_vi_len - 1)
    total_psnr /= (self.args.train_vi_len - 1)
```

```
    return total_mse_loss, total_psnr, total_kl_loss
```

Listing 3: Testing protocol.

```
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,
    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].cpu()]
    label_list = []

    # TODO
    # raise NotImplementedError

    input_frame = decoded_frame_list[0].to(self.args.device)
    for i in range(1, 630):
        frames = self.frame_transformation(input_frame)
        labels = self.label_transformation(label[i])
        z, _, _ = self.Gaussian_Predictor(frames, labels)
        z = torch.randn_like(z)
        fused_pred = self.Decoder_Fusion(frames, labels, z)
        pred = self.Generator(fused_pred)

        input_frame = pred.detach()
        decoded_frame_list.append(pred.cpu())
        label_list.append(label[i].cpu())


    # 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

    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
```

2.2 Reparameterization tricks

After obtaining the VAE encoder outputs—mean μ and log variance $\log \sigma^2$—we follow the definition of a normal distribution by applying the reparameterization trick:

$$z = \mu + \exp\left(0.5 \cdot \log \sigma^2\right) \cdot \epsilon \tag{1}$$

where $\epsilon \sim \mathcal{N}(0, I)$. This operation maps the latent vector z to follow the target distribution $\mathcal{N}(\mu, \sigma^2)$. The implementation is shown in Listing 4.

Listing 4: Reparameterization tricks.

```
def reparameterize(self, mu, logvar):
    # TODO
    # raise NotImplementedError
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    z = mu + eps * std
    return z
```

2.3 Teacher forcing strategy

The purpose of the teacher forcing strategy is to let the neural network learn the prediction by the previous ground truth frame to the current ground truth frame, which is controlled by a ratio, and off means that the prediction is based on the previous predicted frame. It is controlled by the initial teacher forcing ratio, start to decay epoch and decay step. It started from the start to decay epoch, then linearly decreased the teacher forcing ratio until the value reached zero. The implementation is shown in Listing 5.

Listing 5: Teacher forcing strategy.

```
def teacher_forcing_ratio_update(self):
    # TODO
    # raise NotImplementedError
    if self.current_epoch >= self.tfr_sde:
        self.tfr -= self.tfr_d_step
        if self.tfr < 0.0:
            self.tfr = 0.0
```

2.4 KL annealing ratio

The KL annealing ratio affects the weight of the KL divergence term (β) in the loss function. I implemented three versions of annealing strategies: *Cyclical*,

Monotonic, and *No Annealing*. In the *Cyclical* approach, β periodically increases and resets to encourage exploration during training; in the *Monotonic* strategy, β gradually increases from zero to a fixed value over iterations; and in the *No Annealing* case, β remains constant throughout training. These variations provide insight into how KL regularization influences the learning process and the diversity of generated outputs. The implementation is shown in Listing 6, adapted and modified from the official implementation of [3].

Listing 6: KL annealing ratio.

```
class kl_annealing():
def __init__(self, args, current_epoch=0):
    # TODO
    # raise NotImplementedError
    self.current_epoch = current_epoch
    self.total_epoch = args.num_epoch
    self.kl_anneal_type = args.kl_anneal_type
    self.kl_anneal_cycle = args.kl_anneal_cycle
    self.kl_anneal_ratio = args.kl_anneal_ratio

    assert self.kl_anneal_type in ["Cyclical", "Monotonic", "WithoutKL"], f"
    if self.kl_anneal_type == "Cyclical":
        self.beta = self.frange_cycle_linear(
            self.total_epoch, start=0.0, stop=self.kl_anneal_ratio,
            n_cycle=self.kl_anneal_cycle)
    elif self.kl_anneal_type == 'Monotonic':
        self.beta = self.frange_cycle_linear(
            self.total_epoch, start=0.0,
            stop=self.kl_anneal_ratio, n_cycle=1)
    elif self.kl_anneal_type == "WithoutKL":
        self.beta = self.frange_cycle_linear(
            self.total_epoch, start=0.0,
            stop=self.kl_anneal_ratio, n_cycle=1, without_kl=True)

def update(self):
    # TODO
    # raise NotImplementedError
    self.current_epoch += 1

def get_beta(self):
    # # TODO
    # raise NotImplementedError
    return self.beta[self.current_epoch]

def frange_cycle_linear(self, n_iter, start=0.0,
    stop=1.0,  n_cycle=1, ratio=1, without_kl=False):
    # TODO
```

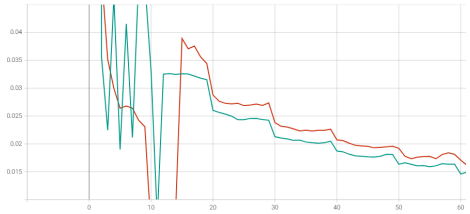

Figure 1: Training loss with different settings of teacher forcing ratios.

```
# raise NotImplementedError
# https://github.com/haofuml/cyclical_annealing
L = np.ones(n_iter) * stop

if without_kl:
    return L

period = n_iter/n_cycle
step = (stop-start)/(period*ratio) # linear schedule

for c in range(n_cycle):
    v, i = start, 0
    while v <= stop and (int(i+c*period) < n_iter):
        L[int(i+c*period)] = v
        v += step
        i += 1
return L
```

3 Analysis & Discussion

3.1 Teacher forcing ratio

In this section, I evaluate the effect of two teacher forcing ratios: 1.0 and 0.5. The corresponding results are presented in Figure **??** and Figure 2. It can be observed that a higher teacher forcing ratio does not necessarily lead to better performance in this task. Additionally, noticeable oscillations occur during the transition from teacher forcing to off mode, indicating instability in the model's predictions when relying less on ground truth inputs. These results show that this is also an important hint which affects the learning.

3.2 Training with different settings of KL annealing

The loss curves are shown in Figure 3, and the weights β are shown in Figure 4. We can observe that cyclical annealing leads to more stable training and improved convergence compared to the other strategies. Additionally, we can

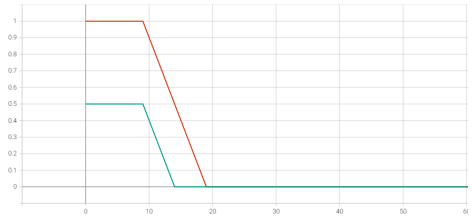

Figure 2: Teacher forcing ratio comparison.

see that using annealing results in better performance, as it allows for a more gradual and adaptive regularization of the latent space, leading to improved generalization and higher-quality outputs.

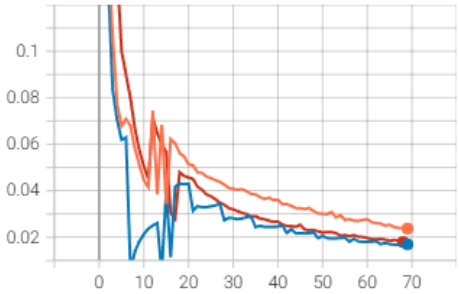

Figure 3: Training loss comparison with different settings of KL annealing. Orange: Monotonic; Blue: Cyclical; Red: No KL Annealing.

3.3 PSNR-per-frame diagram in the validation dataset

The PSNR-per-frame diagram of the validation set is shown in Figure 5. The model is configured with a teacher forcing ratio (TFR) of 1.0, TFR start decay epoch of 10, TFR decay step of 0.1, KL annealing type set to *Cyclical*, KL annealing cycle length of 7, KL annealing ratio of 1.0, and optimized using the Adam optimizer. We observe that the PSNR exhibits oscillations at various time steps, which may correspond to frames with significant character movement, leading to higher prediction difficulty and reduced frame quality.

References

[1] Caroline Chan, Shiry Ginosar, Tinghui Zhou, and Alexei A Efros. Everybody dance now. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 5933–5942, 2019.

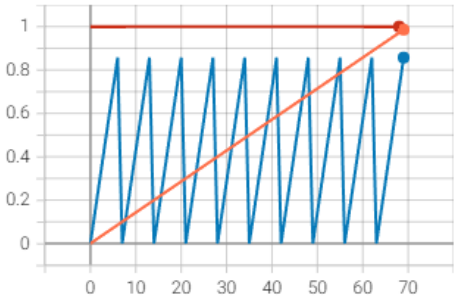

Figure 4: β comparison with different settings of KL annealing. Orange: Monotonic; Blue: Cyclical; Red: No KL Annealing.

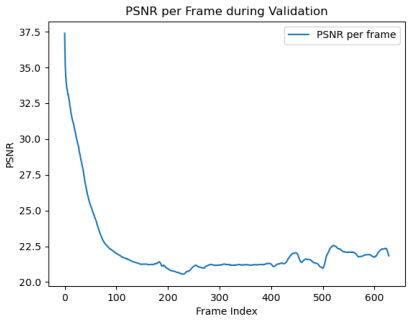

Figure 5: PSNR-per-frame diagram. The y-axis is the PSNR value, and the x-axis is the number of frames.

[2] Emily Denton and Rob Fergus. Stochastic video generation with a learned prior. In *International conference on machine learning*, pages 1174–1183. PMLR, 2018.

[3] Hao Fu, Chunyuan Li, Xiaodong Liu, Jianfeng Gao, Asli Celikyilmaz, and Lawrence Carin. Cyclical annealing schedule: A simple approach to mitigating kl vanishing. *arXiv preprint arXiv:1903.10145*, 2019.