Deep Learning Lab5 (Conditional VAE For Video Prediction)

311553060

周睦鈞

1. Introduction

這次作業要實作 Condition Variational Auto Encoder，我們想要透過前兩個 frame 的資訊來生成後面 10 個 frame 的結果，整體的架構圖如下圖，會由一個 Encoder 和 Decoder 組成，其中在 latent space 中會加入 Condition 的資訊 (Action and Position)

在 Training 的過程中我們會透過調整 Teacher forcing ratio 以及使用不同的 KL weight type 來實驗看看各自的效果對於訓練結果的差異

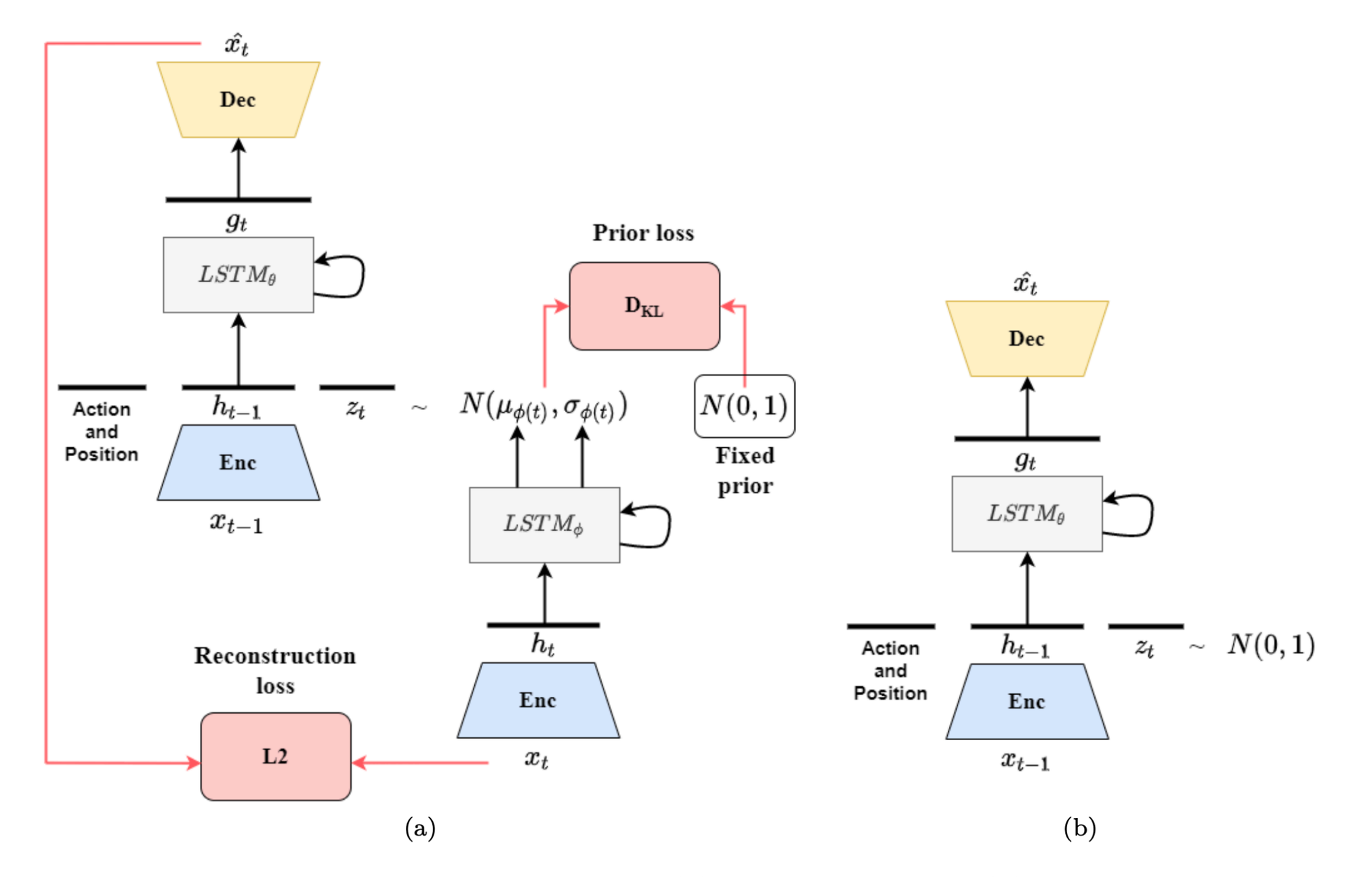


Figure 1: The illustration of overall framework. (a) Training procedure (b) Generation procedure

2. Derivation of CVAE

$$p(x|c) = \int_z p(z|c)\, p(x|z,c)\, dz$$

$$\Rightarrow \log p(x|c) = \int_z q(z|x,c) \log p(x|c)\, dz = \int_z q(z|x,c) \log \frac{p(z,x|c)}{p(z|x,c)}\, dz$$

$$= \int_z q(z|x,c) \log \left(\frac{p(z,x|c)}{q(z|x,c)} \frac{q(z|x,c)}{p(z|x,c)} \right) dz$$

$$= \int_z q(z|x,c) \log \frac{p(z,x|c)}{q(z|x,c)}\, dz + \int_z q(z|x,c) \log \frac{q(z|x,c)}{p(z|x,c)}\, dz$$

$$= \int_z q(z|x,c) \log \frac{p(z,x|c)}{q(z|x,c)}\, dz + KL\left(q(z|x,c) \,\|\, p(z|x,c) \right)$$

$$\geq \int_z q(z|x,c) \log \frac{p(z,x|c)}{q(z|x,c)}\, dz = L_b$$

$$\Rightarrow \log p(x|c) = L_b + KL\left(q(z|x,c) \,\|\, p(z|x,c) \right)$$

$$L_b = \int_z q(z|x,c) \log \frac{p(x|z,c)\, p(z|c)}{q(z|x,c)}\, dz$$

$$= \int_z q(z|x,c) \log \frac{p(z|c)}{q(z|x,c)}\, dz + \int_z q(z|x,c) \log p(x|z,c)\, dz$$

$$= -KL\left(q(z|x,c) \,\|\, p(z|c) \right) + E_{q(z|x,c)}\left[\log p(x|z,c) \right]$$

3. Implementation details

3-1. Describe how you implement your model (encoder, decoder, reparameterization trick, dataloader, etc.)

Encoder 和 Decoder 的部分我並沒有做太多修改，使用原本 sample code 提供的 vgg encoder 和 decoder

```python
class vgg_encoder(nn.Module):
    def __init__(self, dim):
        super(vgg_encoder, self).__init__()
        self.dim = dim
        # 64 x 64
        self.c1 = nn.Sequential(
                vgg_layer(3, 64),
                vgg_layer(64, 64),
                )
        # 32 x 32
        self.c2 = nn.Sequential(
                vgg_layer(64, 128),
                vgg_layer(128, 128),
                )
        # 16 x 16
        self.c3 = nn.Sequential(
                vgg_layer(128, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 256),
                )
        # 8 x 8
        self.c4 = nn.Sequential(
                vgg_layer(256, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 512),
                )
```

```python
        # 4 x 4
        self.c5 = nn.Sequential(
                nn.Conv2d(512, dim, 4, 1, 0),
                nn.BatchNorm2d(dim),
                nn.Tanh()
                )
        self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

    def forward(self, input):
        h1 = self.c1(input) # 64 -> 32
        h2 = self.c2(self.mp(h1)) # 32 -> 16
        h3 = self.c3(self.mp(h2)) # 16 -> 8
        h4 = self.c4(self.mp(h3)) # 8 -> 4
        h5 = self.c5(self.mp(h4)) # 4 -> 1
        return h5.view(-1, self.dim), [h1, h2, h3, h4]
```

```python
class vgg_decoder(nn.Module):
    def __init__(self, dim):
        super(vgg_decoder, self).__init__()
        self.dim = dim
        # 1 x 1 -> 4 x 4
        self.upc1 = nn.Sequential(
                nn.ConvTranspose2d(dim, 512, 4, 1, 0),
                nn.BatchNorm2d(512),
                nn.LeakyReLU(0.2, inplace=True)
                )
        # 8 x 8
        self.upc2 = nn.Sequential(
                vgg_layer(512*2, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 256)
                )
        # 16 x 16
        self.upc3 = nn.Sequential(
                vgg_layer(256*2, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 128)
                )
```

```python
        # 32 x 32
        self.upc4 = nn.Sequential(
                vgg_layer(128*2, 128),
                vgg_layer(128, 64)
                )
        # 64 x 64
        self.upc5 = nn.Sequential(
                vgg_layer(64*2, 64),
                nn.ConvTranspose2d(64, 3, 3, 1, 1),
                nn.Sigmoid()
                )
        self.up = nn.UpsamplingNearest2d(scale_factor=2)

    def forward(self, input):
        vec, skip = input
        d1 = self.upc1(vec.view(-1, self.dim, 1, 1)) # 1 -> 4
        up1 = self.up(d1) # 4 -> 8
        d2 = self.upc2(torch.cat([up1, skip[3]], 1)) # 8 x 8
        up2 = self.up(d2) # 8 -> 16
        d3 = self.upc3(torch.cat([up2, skip[2]], 1)) # 16 x 16
        up3 = self.up(d3) # 8 -> 32
        d4 = self.upc4(torch.cat([up3, skip[1]], 1)) # 32 x 32
        up4 = self.up(d4) # 32 -> 64
        output = self.upc5(torch.cat([up4, skip[0]], 1)) # 64 x 64
        return output
```

Reparameterization Trick: 將 Network 預測出來的 μ 加上 var * $\mathcal{N}(0,1)$ (normal distribution) 來解決 sample 沒辦法做 gradient 的問題

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

Dataloader 的部分，由於我們最後是想要用前兩個 frame 來預測後面 10 個 frame ，因此我改寫成只會讀取前 12 張 frame 送到 Network 做訓練，另外 condition 的部分，會將 action 和 position concatenate 在一起

```python
    def __len__(self):
        return self.length

    def load_img(self, video, frame):
        img = cv2.imread(self.data_path + video + f'/{frame}.png')
        return self.transform((cv2.cvtColor(img, cv2.COLOR_BGR2RGB))/255.0)

    def get_seq(self, index):
        video = self.videos[index]

        seq = torch.stack([self.load_img(video, i).to(torch.float32) for i in range(self.seq_length)], dim=0

        return seq


    def get_csv(self, index):
        # start = 0
        position = self.positions[index, :self.seq_length]
        action = self.actions[index, :self.seq_length]

        condition = np.hstack((position, action))

        return torch.from_numpy(condition).to(torch.float32)
```

```python
    def __getitem__(self, index):
        self.set_seed(index)
        seq = self.get_seq(index)
        cond =  self.get_csv(index)

        return seq, cond
```

當要使用 Teacher forcing 的時候，我會將上一個 frame 由 model predict 出來的結果來通過 encoder 當作下一層要送到 lstm 前的 latent code

```python
    use_teacher_forcing = True if random.random() < args.tfr else False

    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]

    ## TODO
    for i in range(1, args.n_past + args.n_future):
        if args.last_frame_skip or i < args.n_past:
            h, skip = h_seq[i-1]
        else:
            if use_teacher_forcing:
                h, skip = h_seq[i-1]
            else:
                h = modules['encoder'](x_pred)[0]

        h_target = h_seq[i][0]
        z_t, mu, logvar = modules['posterior'](h_target)
        h_pred = modules['frame_predictor'](cond[i-1], torch.cat([h, z_t], dim = 1))
        x_pred = modules['decoder']([h_pred, skip])

        mse += mse_criterion(x_pred, x[i])
        kld += kl_criterion(mu, logvar, args)

    beta = kl_anneal.get_beta(args, epoch, start_epoch)
    loss = mse + kld * beta
    loss.backward()
```

Condition 的部分，我會將 condition 先經過 nn.Linear 將 dim 從 7 → 2 (Action + Position)，之後將 embedded 的 condition 資訊和 h 以及 z concatenate 再一起之後按到 lstm 的方式得到要送到 decoder 前的 latent code

```python
class lstm(nn.Module):
    def __init__(self, condition_size, input_size, output_size, hidden_size, n_layers, batch_size, device):
        super(lstm, self).__init__()
        self.device = device
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.n_layers = n_layers
        self.cond_embed = nn.Linear(condition_size, 2)
        self.embed = nn.Linear(2 + input_size, hidden_size)
        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
        self.output = nn.Sequential(
                nn.Linear(hidden_size, output_size),
                nn.BatchNorm1d(output_size),
                nn.Tanh())
        self.hidden = self.init_hidden()

    def init_hidden(self):
        hidden = []
        for _ in range(self.n_layers):
            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device)),
                           Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device))))
        return hidden
```

```python
    def forward(self, cond, input):
        cond_embedded = self.cond_embed(cond)
        embedded = self.embed(torch.cat([cond_embedded, input], dim = 1))
        h_in = embedded
        for i in range(self.n_layers):
            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
            h_in = self.hidden[i][0]

        return self.output(h_in)
```

Teacher forcing ratio:

```python
        if epoch >= args.tfr_start_decay_epoch:
            ### Update teacher forcing ratio ###
            ## TODO
            args.tfr_decay_step = 1.0 / (args.tfr_truncate_epoch - args.tfr_start_decay_epoch)
            args.tfr -= args.tfr_decay_step
            args.tfr = max(args.tfr, 0)
```

3-2. Describe the teacher forcing (including main idea, benefits and drawbacks.)

Teacher forcing 的想法是我們在訓練序列生成的時候，會想要將 GT 來取代 Decoder 的輸出並且送到下一層的輸入，在訓練的過程中會慢慢減少使用 GT 的方式並且改使用 model 自己生成的輸出來當作下一層的輸入

Benefits: 利用 teacher forcing 的方式可以讓 model 比較好收斂，因為 model 可以在訓練初期學到正確的輸入送到下一層，反之，若不使用 teacher forcing 容易造成 model 預測的和 GT 差距過大造成 loss 容易發散並且不容易收斂

Drawbacks: 由於 model 會過度依賴 GT 的結果，因此當訓練和預測的 dataset 分布差異過大時，model 會因為受到 GT 的影響過渡矯正，造成在預測結果上可能會表現不好

4. Results and discussion

4-1. Show your results of video prediction

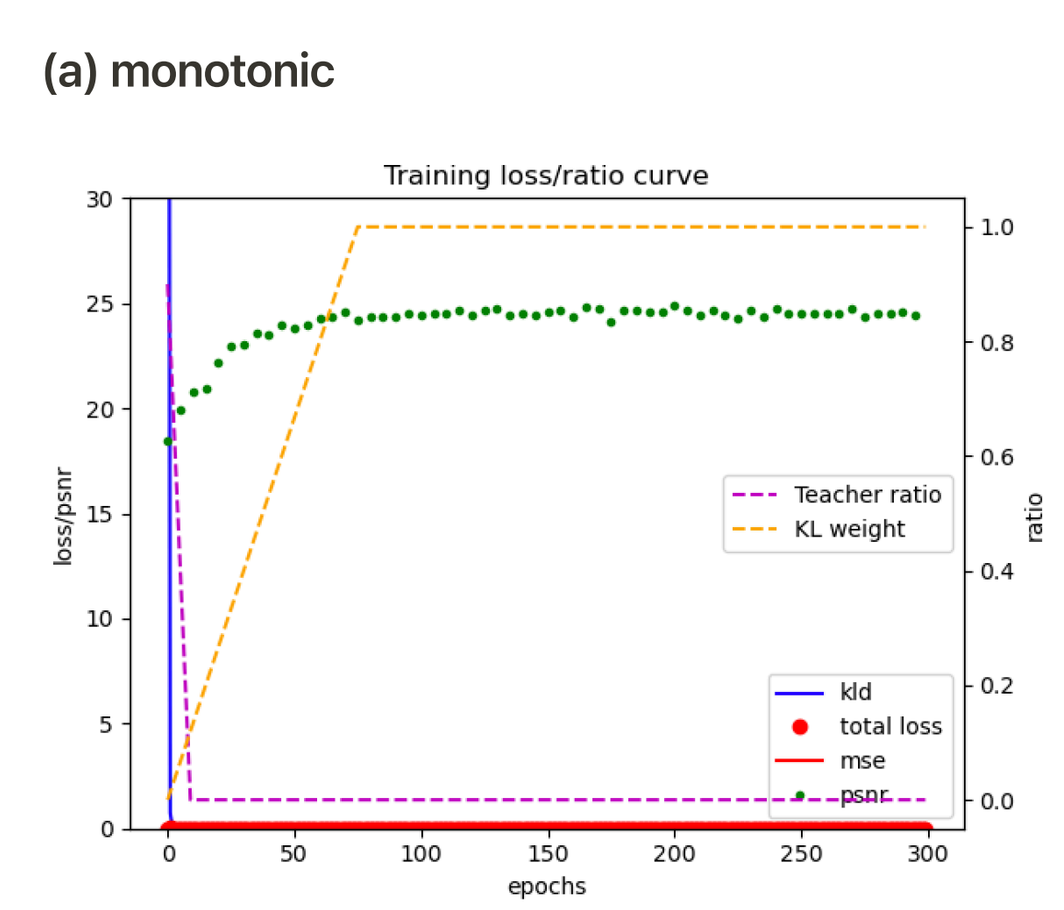
(a) Make videos or gif images for test result

Show at demo

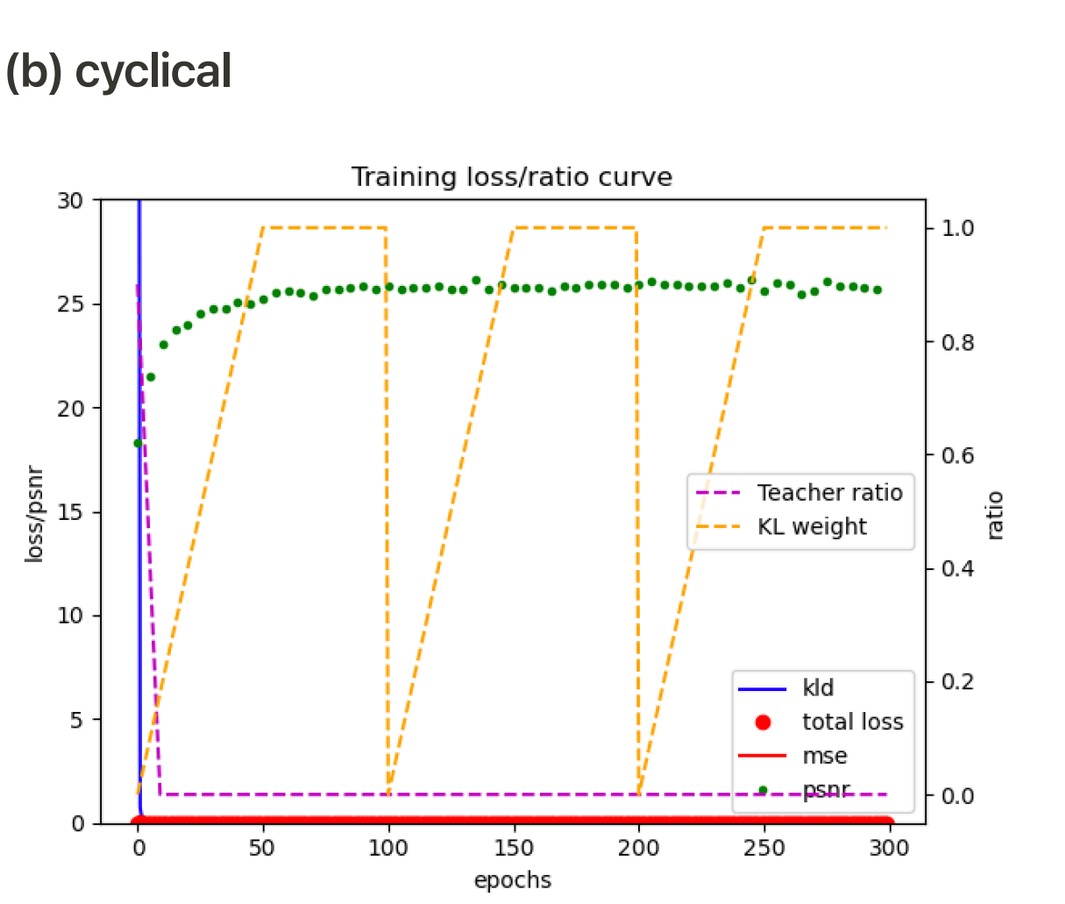
(b) Output the prediction at each time step

4-2. Plot the KL loss and PSNR curves during training

(a) monotonic

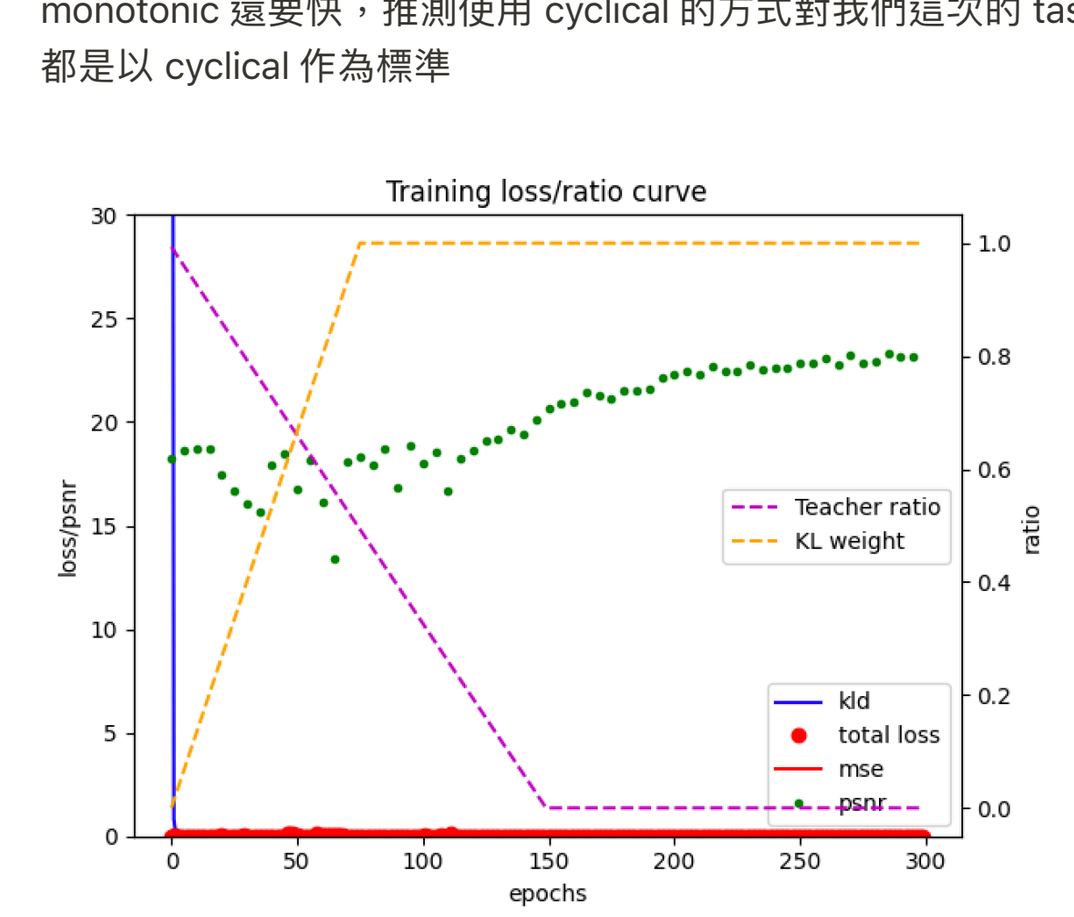


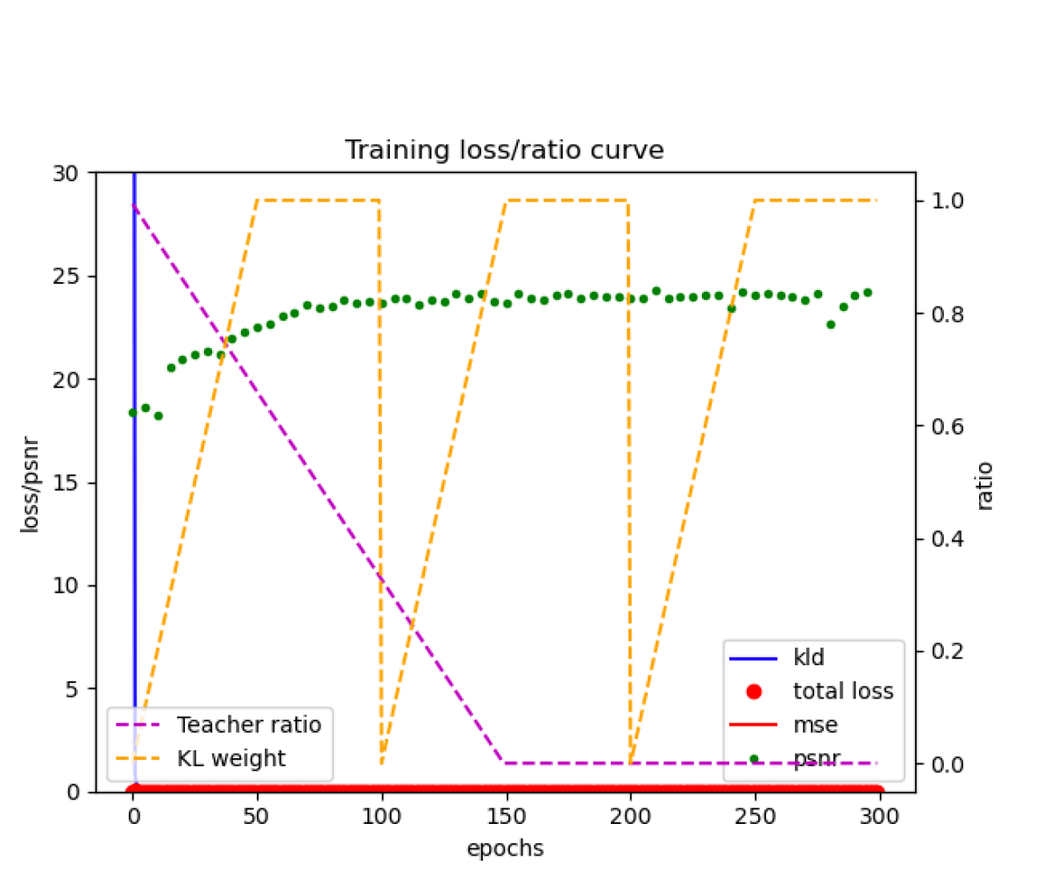
(b) cyclical



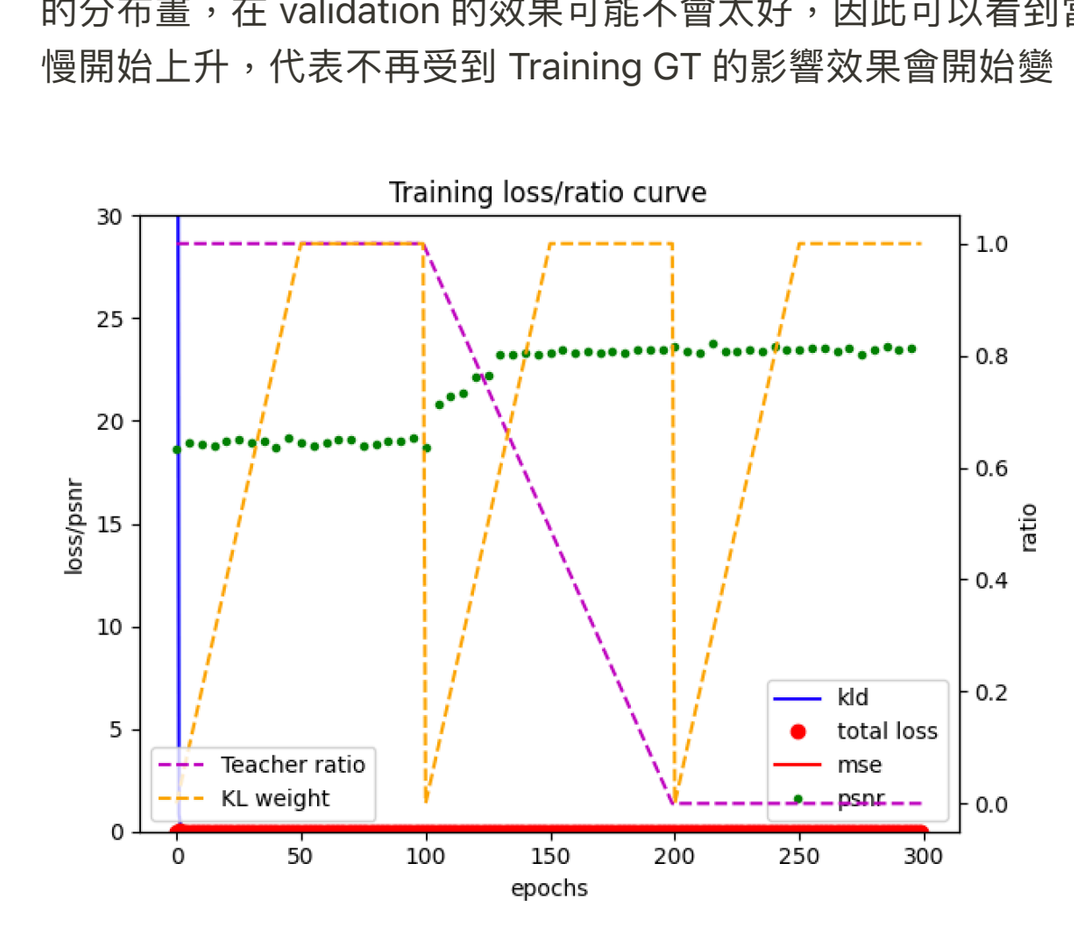
4-3. Discussion

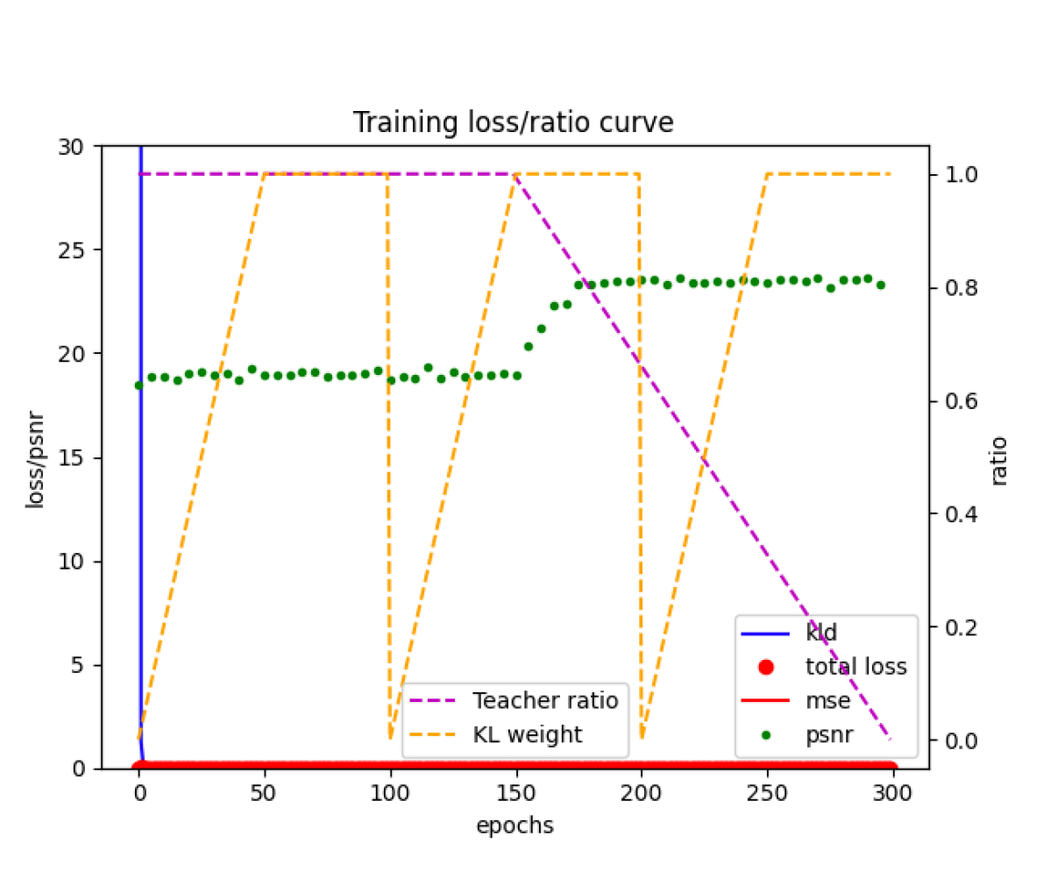
我一開始使用的 teacher forcing ratio 是到 150個 epoch 會線性遞減到 0 , 在這個過程中我發現 cyclical 的收斂速度比 monotonic 還要快，推測使用 cyclical 的方式對我們這次的 task 幫助較大 ，因此後面我嘗試改變不同的 teacher forcing ratio 都是以 cyclical 作為標準





在下面兩張圖，我分別針對 teacher forcing ratio 從 100 → 200 降到 0 以及 150 → 300 降到 0 ，在過程中我發現如果都使用 teacher forcing 來訓練模型，可能會因為 training 和 validation 的資料集分佈有差距，如果 model 被強制學成跟 training 一樣的分布畫，在 validation 的效果可能不會太好，因此可以看到當開始可以使用模型自己的輸出當作下一層輸入時，PSNR 會慢慢開始上升，代表不再受到 Training GT 的影響效果會開始變





有了上述經驗，我開始思考，如果 teacher forcing ratio 提早開始下降是否會加速 PSNR 的收斂速度，因此我讓 teacher forcing ratio 從 0 → 10 個 epoch 會下降到 0 ，發現跟我們想的差不多，PSNR 曲線會開始快速收斂，因此我最後也使用這組參數來當作 demo 的參數