311551069 余忠旻 Lab5 : Conditional VAE For Video Prediction

1. Introduction

這個 lab 要用 Conditional VAE 來做 video prediction。dataset 為機器人動作影片，每個 sequence 切成 30 個 frame，存成 30 個 64*64 的 png 圖片檔。我們的目標為給前 2 個 frame，並利用 VAE generate 的特性預測出後 10 個 frame。另外，dataset 中對於每個 frame 都有 action 和 endeffector position，分別為 4 個以及 3 個元素(共 7 個元素)，可以拿來當作 condition，以利預測。

2. Derivation of CVAE

We know conditional distribution $p(X|c;\theta)$

$$P(X|c;\theta) = \int p(X|Z,c;\theta)\, p(Z|c)\, dZ$$

To see how the EM works, the chain rule of probability suggest:

$$\log p(X|c;\theta) = \log p(X,Z|c;\theta) - \log p(Z|X,c;\theta)$$

By introducing an arbitary distribution $q(Z|c)$ on both sides and integrate over Z:

$$\log p(X|c;\theta) = \int q(Z|c) \log p(X|c;\theta)\, dZ$$

$$= \int q(Z|c) \log p(X,Z|c;\theta)\, dZ - \int q(Z|c) \log p(Z|X,c;\theta)\, dZ$$

$$= \int q(Z|c) \log p(X,Z|c;\theta)\, dZ - \int q(Z|c) \log q(Z|c)\, dZ$$

$$+ \int q(Z|c) \log q(Z|c)\, dZ - \int q(Z|c) \log p(Z|X,c;\theta)\, dZ$$

$$= L(X,q,\theta|c) + KL(q(Z|c) \,\|\, p(Z|X,c;\theta))$$

where $L(X,q,\theta|c) = \int q(Z|c) \log p(X,Z|c;\theta)\, dZ - \int q(Z|c) \log q(Z|c)\, dZ$

and $KL(q(Z|c) \,\|\, p(Z|X,c;\theta)) = \int q(Z|c) \log \frac{q(Z|c)}{p(Z|X,c;\theta)}\, dZ$

since we know KL-divergence is non-negative,

$$\log p(X|c;\theta) \geq L(X,q,\theta|c)$$

↳ evidence lower bound (ELBO)

We rearrange above equation:

$$L(X, q, \theta | c) = \log p(X|c;\theta) - KL\left(q(z|c) \| p(z|X, c;\theta)\right)$$

Since the equality holds for any choice of $q(z|c)$, we can also introduce a distribution $q(z|X, c; \phi)$ modeled by another neural network (the encoder) with parameter ϕ.

$$L(X, q, \theta | c) = \log p(X|c;\theta) - KL\left(q(z|X, c;\phi) \| p(z|X, c;\phi)\right)$$

$$= E_{z\sim q(z|X,c;\phi)}\left[\log p(X|z, c;\theta) + \log p(z|c) - \log q(z|X, c;\phi)\right]$$

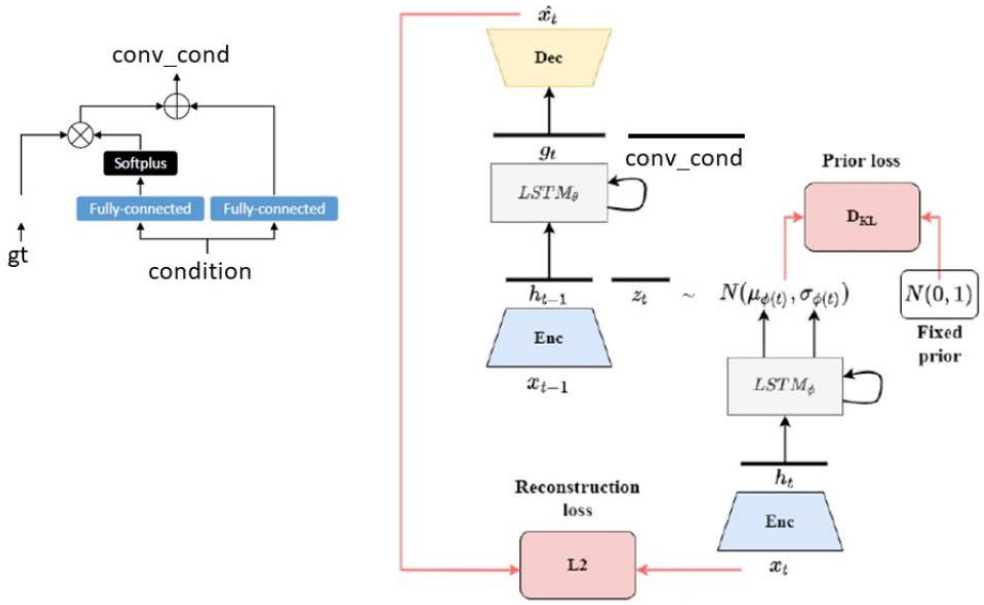
$$= E_{z\sim q(z|X,c;\phi)}\left[\log p(X|z, c;\theta) - KL\left(q(z|X, c;\phi) \| p(z|c)\right)\right]$$

3. Implementation details

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

在這次作業，我實作了 fixed prior 和 learned prior 兩種架構：

- **Main structure (fixed prior)**



1. 首先，會先讓圖片的 frame x_{t-1} 和 frame x_t 都通過 vgg encoder，分別產生 latent vector h_{t-1} 和 h_t。

2. 接著，讓 h_t 通過 posterior，即是上圖右半邊的 gaussian LSTM，產生 mean 和 variance，而利用 reparameterize trick 可以從 mean 和 variance 的資訊 sample 出 z_t。

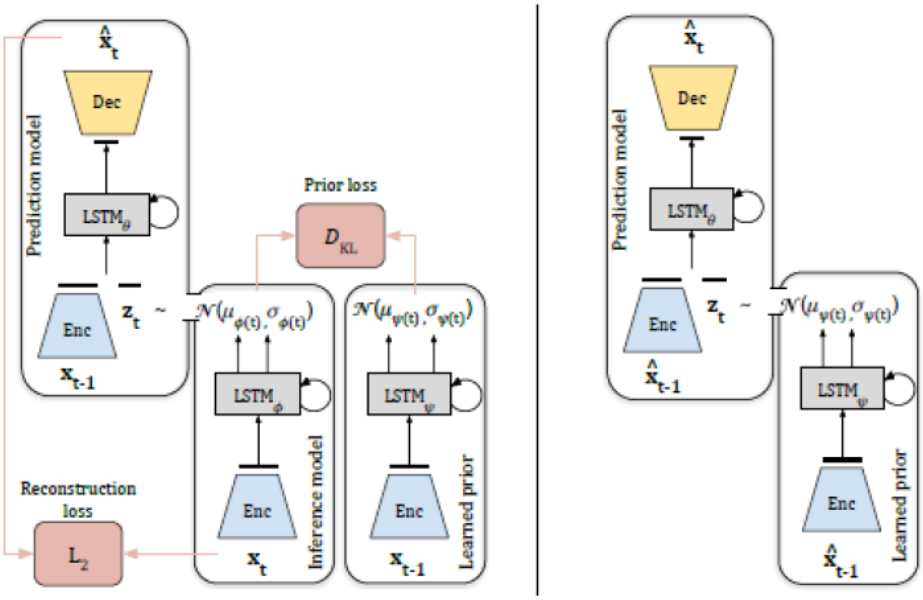
3. sample 出的 z_t 會和 h_{t-1} 一起放入 frame predictor，即是上圖左半邊的 LSTM，試圖 predict 出下一個 frame 的 latent vector g_t (為了避免與右半邊的 h_t 混淆，這裡 predict 出來的 latent vector 以 g_t 表示，而不是 h_t)

4. 最後將 g_t 放入 vgg decoder 解碼就能還原出圖片。在 vgg decoder 中，為了幫助順利生成下一個 frame 的圖片，我有實作 conditional convolution 加到 vgg decoder，也就是將 condition (action 和 endeffector position 所組成) 以及 g_t 當作 input，output 出 conv_cond，再與原來 g_t 串接放入 decoder 解碼。

- **Main structure (learned prior) [1]**

主要架構和 fixed prior 差不多，在 fixed prior 中，我們有兩個 training 目標: 一是讓 generate 出的圖片與真實越接近越好，也就是架構圖中的 reconstruction loss ; 二是讓 posterior 輸出的 mean，variance 越接近 $N(0, I)$ 越好，即架構圖的 prior loss。這麼做的目的是為了讓 frame predictor 接收到這一個 frame 與下一個 frame 相關聯的訊息，也就是 z_t ，讓 z_t 趨向從 $N(0, I)$ 中 sample，也就是預設兩個相鄰的 frame 之間的差距關係是 $N(0, I)$，而這可能會忽略它們真正的 dependencies。

為了解決上述所說的問題，我們引進了 learned prior 來解決：
將原本 fixed prior 的 $N(0, I)$ 取代為另一個 vgg encoder + gaussian LSTM 的架構，並將 x_{t-1} 作為 input，如下圖。這樣能更好的學習出兩相鄰 frame 之間的 distribution 關係，z_t 的產生也會受到 learned prior 影響。



■ Encoder

vgg encoder 做的事就是讓 frame sequence 當作 input，輸出 latent vector，就如上面 Main structure 部分所說，而實作部分，sample code 就已經做好了，我沒有進行修改，因此不加以贅述。

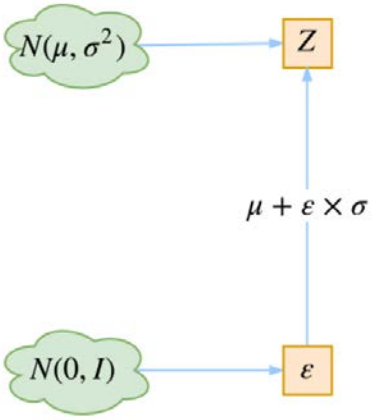
```
class vgg_encoder(nn.Module):
    def __init__(self, dim):
        super(vgg_encoder, self).__init__()
        self.dim = dim
        # 64 x 64
        self.c1 = nn.Sequential(
                vgg_layer(3, 64),
                vgg_layer(64, 64),
                )
        # 32 x 32
        self.c2 = 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),
                )
        # 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]
```

■ Posterior

Posterior(gaussian LSTM) 會產生 mean 和 variance，而利用 reparameterize trick 可以從 mean 和 variance 的資訊 sample 出 z_t。

reparameterize trick 的做法是從 $N(\mu, \sigma^2)$ 中 sample 一個 Z，也就相當於從 $N(0, I)$ 中 sample ε 一個，然後讓 $Z = \mu + \varepsilon \times \sigma$，如下圖所示：



而我們實作時是使用 log variance，所以要進行轉換：

$$Z = \mu + \epsilon \times e^{0.5 \times \log \sigma^2} = \mu + \epsilon \times e^{\log \sqrt{\sigma^2}} = \mu + \epsilon \times \sigma$$

```
class gaussian_lstm(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
        super(gaussian_lstm, self).__init__()
        self.device = device
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.batch_size = batch_size
        self.embed = nn.Linear(input_size, hidden_size)
        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
        self.mu_net = nn.Linear(hidden_size, output_size)
        self.logvar_net = nn.Linear(hidden_size, output_size)
        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

    ## TODO
    def reparameterize(self, mu, logvar):
        logvar = logvar.mul(0.5).exp_()
        eps = Variable(logvar.data.new(logvar.size()).normal_())
        return eps.mul(logvar).add_(mu)

    def forward(self, input):
        embedded = self.embed(input)
        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]
        mu = self.mu_net(h_in)
        logvar = self.logvar_net(h_in)
        z = self.reparameterize(mu, logvar)
        return z, mu, logvar
```

■ Frame Predictor

sample 出的 z_t 會和 h_{t-1} 一起放入 frame predictor(LSTM) 試圖 predict 出下一個 frame 的 latent vector g_t。就如上面 Main structure 部分所說，而實作部分，sample code 就已經做好了，我沒有進行修改，因此不加以贅述。

```
class lstm(nn.Module):
    def __init__(self, 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.embed = nn.Linear(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

    def forward(self, input):
        embedded = self.embed(input)
        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)
```

■ Decoder [2]

vgg decoder 會將 g_t 放入 vgg decoder 進行解碼。在上面有談論到，我實作 conditional convolution 加到 vgg decoder，而在原 paper 的 conditional convolution 架構中，condition 要先進行 one-hot encoding。但我們的 condition 是一個大範圍的實數，不容易進行 one-hot encoding，進行了之後效果也不一定好，因此我選擇不實作 one-hot encoding 的部分。

```
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)
                )
        # 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)

        # conditional convolution
        self.cond_fc1 = nn.Linear(7, 128)
        self.cond_fc2 = nn.Linear(7, 128)
        self.sp = nn.Softplus()
    def forward(self, input, cond):
        # conditional convolution
        vec, skip = input

        cond1 = cond
        cond1 = self.cond_fc1(cond1)
        cond1 = self.sp(cond1)
        cond2 = cond
        cond2 = self.cond_fc2(cond2)

        conv_cond = torch.mul(vec, cond1)
        conv_cond = torch.add(conv_cond, cond2)
        vec = torch.cat([vec, conv_cond], 1) # (12 x 128) cat (12 x 128) = (12 x 256)

        # decoder
        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
```

■ Dataloader

dataloader 主要分為四個部分: initialize、get sequence、get condition、get item。

initialize 部分主要是先看是 train mode、validate mode 還是 test mode，來決定要從哪個路徑來取得資料，並將資料路徑存起來方便 get sequence 和 get condition 開啟。

```
def __init__(self, args, mode='train', transform=default_transform):
    assert mode == 'train' or mode == 'test' or mode == 'validate'
    self.root = '{}/{}'.format(args.data_root, mode)
    self.seq_len = max(args.n_past + args.n_future, args.n_eval)
    self.mode = mode
    if mode == 'train':
        self.ordered = False
    else:
        self.ordered = True

    self.transform = transform
    self.dirs = []
    for dir1 in os.listdir(self.root):
        for dir2 in os.listdir(os.path.join(self.root, dir1)):
            self.dirs.append(os.path.join(self.root, dir1, dir2))

    self.seed_is_set = False
    self.idx = 0
    self.cur_dir = self.dirs[0]
```

get sequence 部分(get_seq function)，就是取得一個 12 張圖片檔案的 sequence。圖片用 PIL 的 Image 來開啟再用 torchvision transform 中的 ToTensor 轉換成 tensor 型態。另外，我們會讓 training mode 隨機選擇順序來跑 sequence，但 validate mode 和 test mode 會要求他每次都能固定順序跑過 dataset 中的每一個 sequence。

```
def get_seq(self):
    if self.ordered:
        self.cur_dir = self.dirs[self.idx]
        if self.idx == len(self.dirs) - 1:
            self.idx = 0
        else:
            self.idx += 1
    else:
        self.cur_dir = self.dirs[np.random.randint(len(self.dirs))]

    image_seq = []
    for i in range(self.seq_len):
        fname = '{}/{}.png'.format(self.cur_dir, i)
        img = Image.open(fname)
        image_seq.append(self.transform(img))
    image_seq = torch.stack(image_seq)

    return image_seq
```

get condition 部分(get_csv function)，是開啟 sequence file 資料夾中的 actions.csv 和 endeffector_position.csv 檔。和 get sequence 一樣取得 12 個 frame 對應的 action 和 endeffector position，每個 frame 所對應的 action 和 endeffector position 分別為 4 個以及 3 個元素(共 7 個元素)，將其串聯後回傳。

```
def get_csv(self):
    with open('{}/actions.csv'.format(self.cur_dir), newline='') as csvfile:
        rows = csv.reader(csvfile)
        actions = []
        for i, row in enumerate(rows):
            if i == self.seq_len:
                break
            action = [float(value) for value in row]
            actions.append(torch.tensor(action))

        actions = torch.stack(actions)

    with open('{}/endeffector_positions.csv'.format(self.cur_dir), newline='') as csvfile:
        rows = csv.reader(csvfile)
        positions = []
        for i, row in enumerate(rows):
            if i == self.seq_len:
                break
            position = [float(value) for value in row]
            positions.append(torch.tensor(position))
        positions = torch.stack(positions)

    condition = torch.cat((actions, positions), axis=1)

    return condition
```

最後 get item 整合上述函式，供 pytorch 中的 DataLoader 使用。而使用 DataLoader 時，由於輸入的格式為 get_seq: [batch size, frame num, C, H, W] 和 get_csv: [batch size, frame num, cond num]，因此在拿到資料後，要先轉換格式再開始訓練。

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

```
seq = seq.permute(1, 0, 2, 3, 4).to(device)
cond = cond.permute(1, 0, 2).to(device)
loss, mse, kld = train(seq, cond, modules, mse_criterion, optimizer, kl_anneal, args)
```

■ Training Procedure

將上述 Main structure 所講的實作出來，如下：

```python
def train(x, cond, modules, mse_criterion, optimizer, kl_anneal, args):
    modules['frame_predictor'].zero_grad()
    modules['posterior'].zero_grad()
    modules['encoder'].zero_grad()
    modules['decoder'].zero_grad()

    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    mse = 0
    kld = 0
    use_teacher_forcing = True if random.random() < args.tfr else False

    ## TODO
    # calculate full sequence
    for i in range(1, args.n_past+args.n_future):
        # h_t
        h_target = h_seq[i][0]
        if args.last_frame_skip or i < args.n_past:
            # h_t-1
            h, skip = h_seq[i-1]
        else:
            # h_t-1
            h = h_seq[i-1][0]

        # guassian and latent vector
        z_t, mu, logvar = modules['posterior'](h_target)
        h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1))
        x_pred = modules['decoder']([h_pred, skip], cond[i-1])

        if not use_teacher_forcing:
            h_seq[i] = modules['encoder'](x_pred)

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

    beta = kl_anneal.get_beta()
    loss = mse + kld * beta
    loss.backward()

    optimizer.step()

    return loss.detach().cpu().numpy() / (args.n_past + args.n_future), mse.detac
```

fixed prior

```python
def train(x, cond, modules, mse_criterion, optimizer, kl_anneal, args):
    modules['frame_predictor'].zero_grad()
    modules['posterior'].zero_grad()
    modules['encoder'].zero_grad()
    modules['decoder'].zero_grad()
    modules['prior'].zero_grad()

    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    modules['prior'].hidden = modules['prior'].init_hidden()
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    mse = 0
    kld = 0
    use_teacher_forcing = True if random.random() < args.tfr else False

    ## TODO
    # calculate full sequence
    for i in range(1, args.n_past+args.n_future):
        # h_t
        h_target = h_seq[i][0]
        if args.last_frame_skip or i < args.n_past:
            # h_t-1
            h, skip = h_seq[i-1]
        else:
            # h_t-1
            h = h_seq[i-1][0]

        # guassian and latent vector
        z_t, mu, logvar = modules['posterior'](h_target)
        _, mu_p, logvar_p = modules['prior'](h)
        h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1))
        x_pred = modules['decoder']([h_pred, skip], cond[i-1])

        if not use_teacher_forcing:
            h_seq[i] = modules['encoder'](x_pred)

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

    beta = kl_anneal.get_beta()
    loss = mse + kld * beta
    loss.backward()

    optimizer.step()

    return loss.detach().cpu().numpy() / (args.n_past + args.n_future), mse.detac
```

learned prior

■ Testing Procedure

```python
def pred(x, cond, modules, args):
    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    gen_seq = []
    gen_seq.append(x[0])
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    for i in range(1, args.n_past+args.n_future):
        h_target = h_seq[i][0]
        if args.last_frame_skip or i < args.n_past:
            h, skip = h_seq[i-1]
        else:
            h, _ = h_seq[i-1]
        h = h.detach()
        if i < args.n_past:
            z_t, _, _ = modules['posterior'](h_target)
            modules['frame_predictor'](torch.cat([h, z_t], 1))
            gen_seq.append(x[i])
        else:
            z_t = torch.cuda.FloatTensor(args.batch_size, args.z_dim).normal_()
            h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1)).detach()
            x_pred = modules['decoder']([h_pred, skip], cond[i-1]).detach()
            h_seq[i] = modules['encoder'](x_pred)
            gen_seq.append(x_pred)
    return gen_seq
```

fixed prior

```python
def pred_lp(x, cond, modules, args):
    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    modules['prior'].hidden = modules['prior'].init_hidden()
    gen_seq = []
    gen_seq.append(x[0])
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    for i in range(1, args.n_past+args.n_future):
        h_target = h_seq[i][0]
        if args.last_frame_skip or i < args.n_past:
            h, skip = h_seq[i-1]
        else:
            h, _ = h_seq[i-1]
        h = h.detach()
        if i < args.n_past:
            z_t, _, _ = modules['posterior'](h_target)
            modules['frame_predictor'](torch.cat([h, z_t], 1))
            gen_seq.append(x[i])
        else:
            z_t = torch.cuda.FloatTensor(args.batch_size, args.z_dim).normal_()
            h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1)).detach()
            x_pred = modules['decoder']([h_pred, skip], cond[i-1]).detach()
            h_seq[i] = modules['encoder'](x_pred)
            gen_seq.append(x_pred)
    return gen_seq
```

learned prior

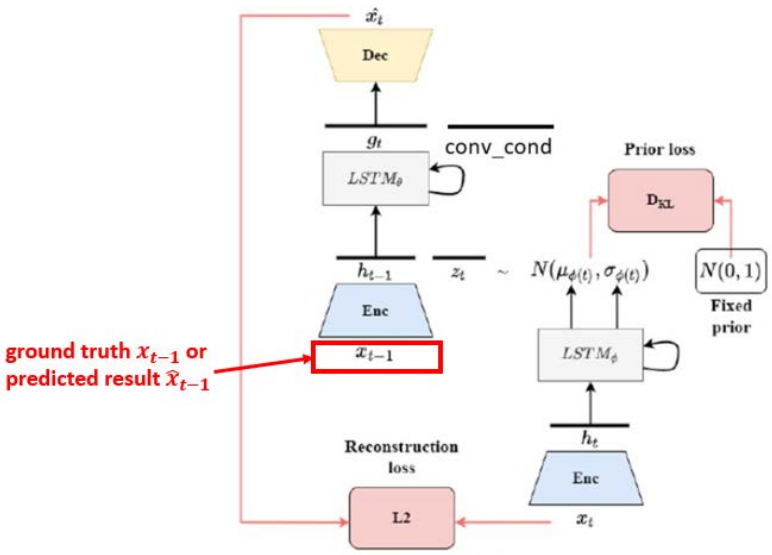
B. Describe the teacher forcing (including main idea, benefits and drawbacks.)

Teacher forcing ratio 是一種快速有效訓練循環神經網路模型(RNN)的方法，訓練的時候 不使用上一個 state 的輸出作爲下一個 state 的輸入，而是直接使用訓練數據的標準答案(ground truth)的對應上一項作爲下一個 state 的輸入。

Teacher-Forcing 可以是一個很好的訓練模式，因為：
(1) Teacher-Forcing 能夠在訓練的時候矯正模型的預測，避免在序列生成的過程中誤差進一步放大。
(2) Teacher-Forcing 能夠極大的加快模型的收斂速度，令模型訓練過程更加快速 以及平穩。

Teacher-Forcing 也存在著一些缺點：
(1) Exposure Bias，也就是 訓練和預測的時候 decode 行為的不一致， 導致預測在訓練和預測的時候是從不同的分佈中推斷出來的。而這種不一致導致訓練模型和預測模型直接的 Gap。
(2) Teacher-Forcing 技術在解碼的時候生成的字符都受到了 ground truth 的約束。而在 testing 過程中，因為無法得到 ground truth 的支持，假如目前生成的序列和 training 時有很大不同，模型就會變得很脆弱。

由於上述的優點以及缺點，訓練剛開始用 Teacher-Forcing 來避免剛開始訓練往錯誤的方向，之後逐漸遞減 teacher forcing ratio，也就是採用 Non-Teacher-Forcing 的機率會逐漸上升，來解決上述所講可能會產生的問題，且隨著次數增加 Non-Teacher-Forcing 的訓練模式也更常被使用。



在 Conditional VAE 架構圖中，如果是 Teacher-Forcing 則使用 ground truth

x_{t-1} 當作 input，經過 encoder 得到 h_{t-1} 去做訓練，反之 Non-Teacher-Forcing 是用 decoder 解碼出來的結果作為下一個 frame 預測的輸入，也就是使用 $\hat{x}_{t-1}$ 當作 input，經過 encoder 得到 h_{t-1} 去做訓練。

實作部分如下，也就是會隨著次數逐漸遞減 teacher forcing ratio：

```
## TODO
if epoch >= args.tfr_start_decay_epoch:
    ### Update teacher forcing ratio ###
    if args.tfr > args.tfr_lower_bound:
        args.tfr = args.tfr - args.tfr_decay_step
```

4. Results and discussion

A. Show your results of video prediction

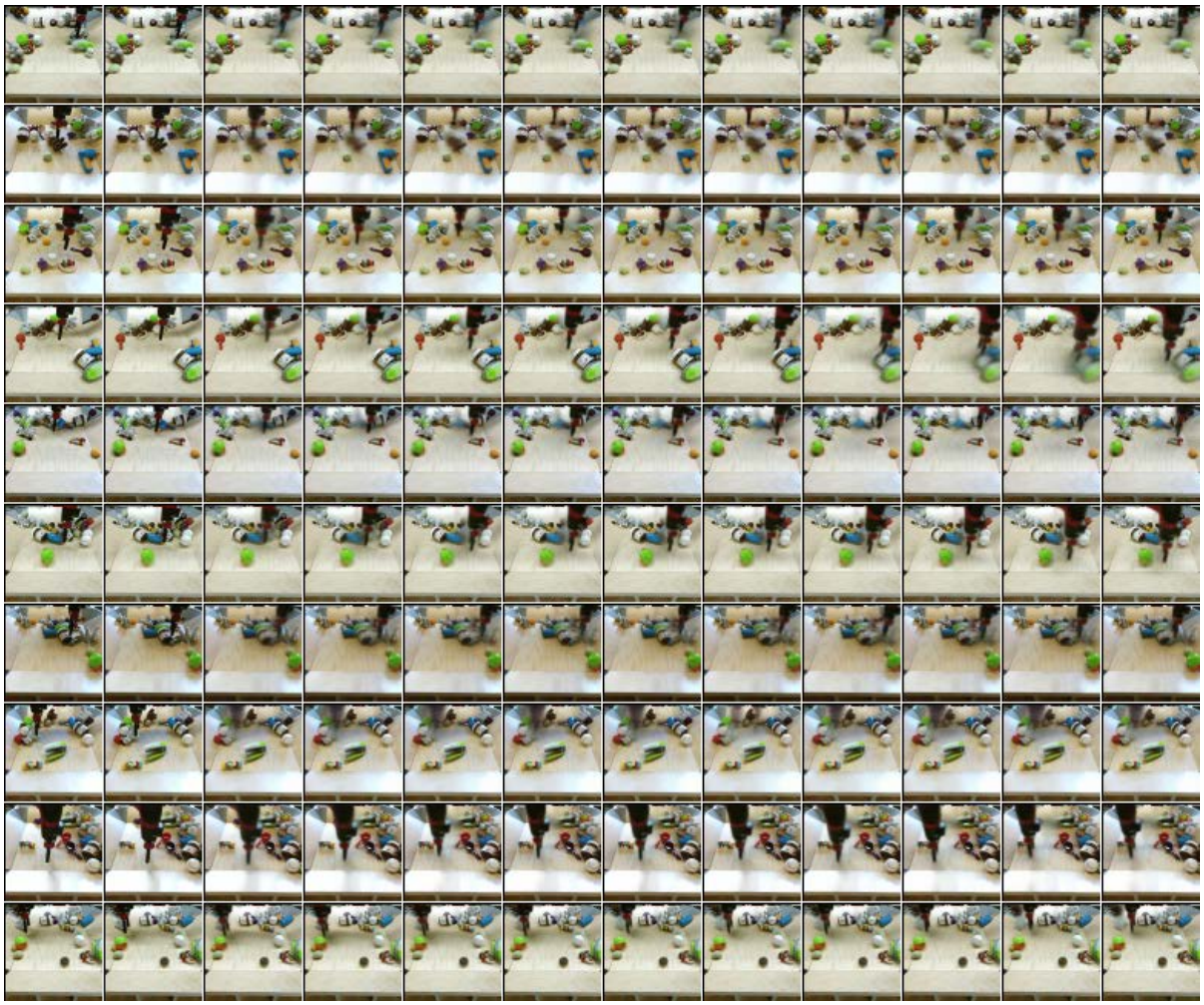
Fixed Prior (Cyclical) : testing PSNR = 25.07

```
pp037@ec037:~/DLP/lab5$ python3 generate_fixed_prior.py
Using pre-trained: best_model/model.pth
Random Seed:  1
Namespace(batch_size=12, beta=0.0001, beta1=0.9, cuda=True, data_root='./data/processed_data', epoch_size=600, g_dim=128, kl_anneal_cycle=4, kl_anneal_cyclical=False, kl_anneal_ratio=2, last_frame_skip=False, log_dir='./best_model', lr=0.002, model_dir='', n_eval=30, n_future=10, n_past=2, niter=160, num_workers=4, optimizer='adam', posterior_rnn_layers=1, predictor_rnn_layers=2, rnn_size=256, seed=1, tfr=1.0, tfr_decay_step=0.01, tfr_lower_bound=0.0, tfr_start_decay_epoch=60, z_dim=64)
100%|| 21/21 [00:29<00:00,  1.12s/it]
avg: 25.077662569248112
100%|| 21/21 [00:37<00:00,  1.80s/it]
```

(a) Make videos or gif images for test result (select one sequence)

([gif link])

(b) Output the prediction at each time step (select one sequence)

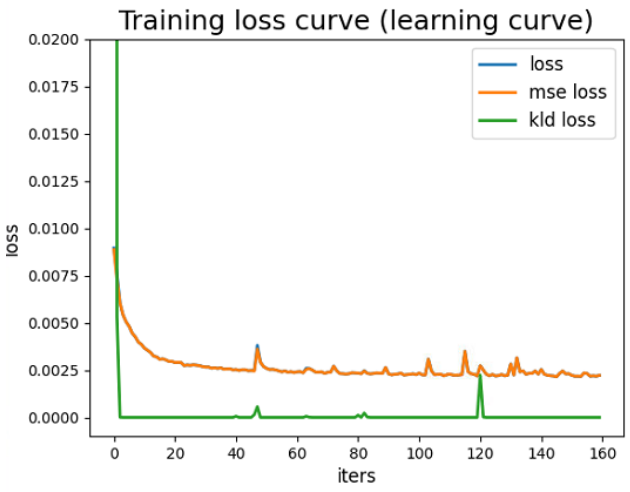


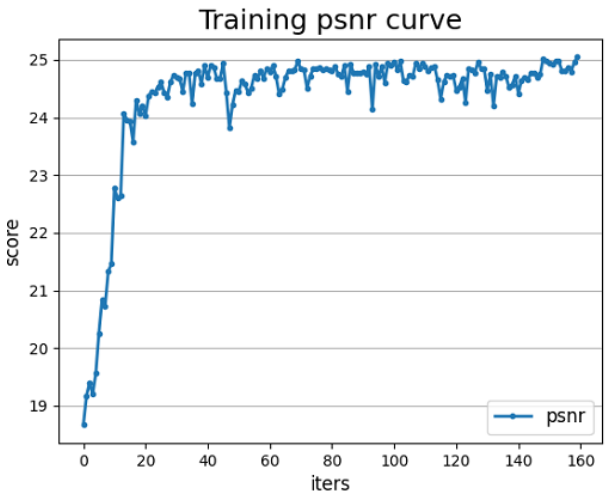
B. Plot the KL loss and PSNR curves during training

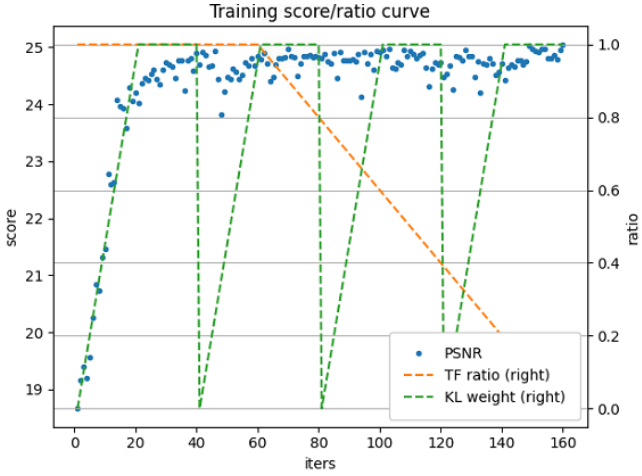
- niter: 160
- epoch size: 600
- batch size: 12
- learning rate: 2e − 3
- optimizer: Adam
- loss function: MSE + KL

Fixed Prior (Cyclical) : testing PSNR = 25.07

```
pp037@ec037:~/DLP/lab5$ python3 generate_fixed_prior.py
Using pre-trained: best_model/model.pth
Random Seed:  1
Namespace(batch_size=12, beta=0.0001, beta1=0.9, cuda=True, data_root='./data/processed_data', epoch_size=600, g_dim=128, kl_anneal_cycle=4, kl_anneal_cyclical=False, kl_anneal_ratio=2, last_frame_skip=False, log_dir='./best_model', lr=0.002, model_dir='', n_eval=30, n_future=10, n_past=2, niter=160, num_workers=4, optimizer='adam', posterior_rnn_layers=1, predictor_rnn_layers=2, rnn_size=256, seed=1, tfr=1.0, tfr_decay_step=0.01, tfr_lower_bound=0.0, tfr_start_decay_epoch=60, z_dim=64)
100%|| 21/21 [00:29<00:00,  1.12s/it]
avg: 25.077662569248112
100%|| 21/21 [00:37<00:00,  1.80s/it]
```

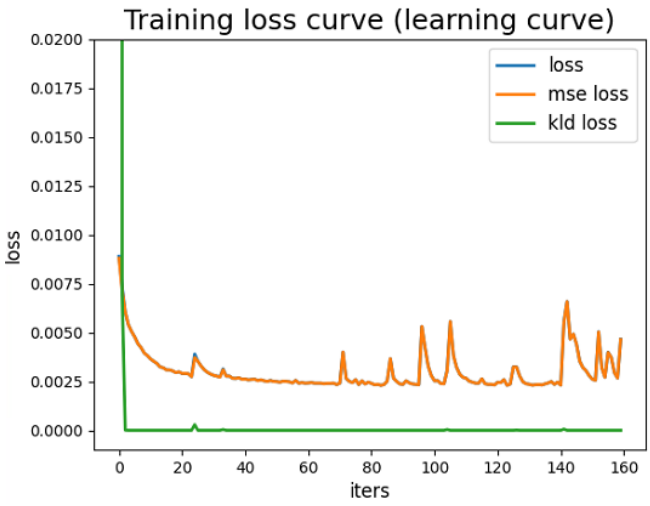


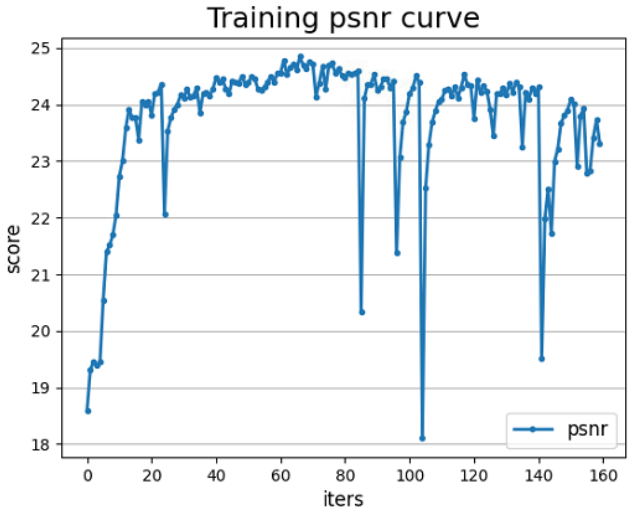


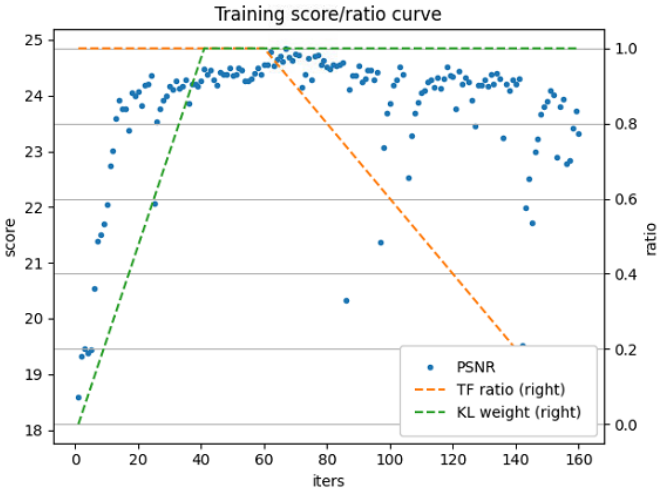


Fixed Prior (Monotonic) : testing PSNR = 24.94

```
pp037@ec037:~/DLP/lab5$ python3 generate_fixed_prior.py
Using pre-trained: best_model/model.pth
Random Seed:  1
Namespace(batch_size=12, beta=0.0001, beta1=0.9, cuda=True, data_root='./data/processed_data', epoch_size=600, g_dim=128, kl_anneal_cycle=4, kl_anneal_cyclical=False, kl_anneal_ratio=2, last_frame_skip=False, log_dir='./best_model', lr=0.002, model_dir='', n_eval=30, n_future=10, n_past=2, niter=160, num_workers=4, optimizer='adam', posterior_rnn_layers=1, predictor_rnn_layers=2, rnn_size=256, seed=1, tfr=1.0, tfr_decay_step=0.01, tfr_lower_bound=0.0, tfr_start_decay_epoch=60, z_dim=64)
100%|| 21/21 [00:30<00:00,  1.14s/it]
 avg: 24.9442791353698
100%|| 21/21 [00:38<00:00,  1.83s/it]
```







上面可以看出 KL annealing 使用 cyclical 方法比 monotonic 方法效果可以得到更好的 PSNR，而且 PSNR curve 比較穩定，而這兩種方法作法及比較會在後面 Discussion 部分詳細討論。

C. Discuss the results according to your setting of teacher forcing ratio, KL weight, and learning rate.

■ **Teacher forcing ratio**

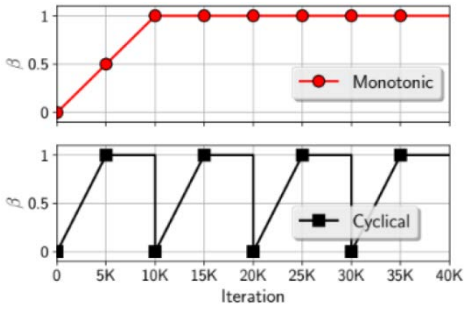
在 teacher forcing 一開始設置較高，讓模型能盡量避免在錯誤中學習，接著逐漸將他調低，才能夠讓學習更為完整。上面也已經講了 teacher forcing 的重要性了，如果都使用 teacher forcing 沒有回歸原本的預測方式，那到後面不會有很好的進步，反之如果都使用 non teacher forcing 剛開始結果就會沒有那麼理想了。

這次 lab 中，我固定了 tfr_decay_step = 0.01
(從 start_decay_epoch 後過 100 epcohs，teacher forcing ratio 會下降為 0)
然後我嘗試調整 tfr_start_decay_epoch，也就是決定訓練初期有多少個 epoch 是完全使用 teacher forcing 訓練，我發現如果一開始就將 teacher forcing ratio 逐漸遞減 (有一定機率會採用 Non-Teacher-Forcing，而且機率會逐漸上升)，也就是 tfr_start_decay_epoch = 0，最後 PSNR 結果會比較差。

	TF strategy	KL strategy	Avg. PSNR score
Fixed Prior	Decay after 60 epochs	Cyclical	25.077662569248112
Fixed Prior	Decay from start	Cyclical	24.862151730020167

■ **KL weight**

KL weight (beta) 的目的是為了讓 model 選擇要 minimize frame prediction error 還是要 fitting the prior。若 beta 太小，model 會過於專注於 minimize frame prediction error，model 就會傾向於直接複製 target frame 而失去生成能力；若 beta 太大則會造成相反效果，導致 prediction 結果不佳。

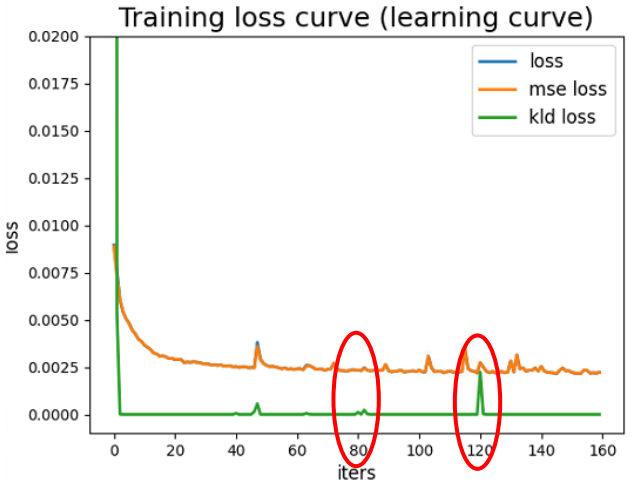


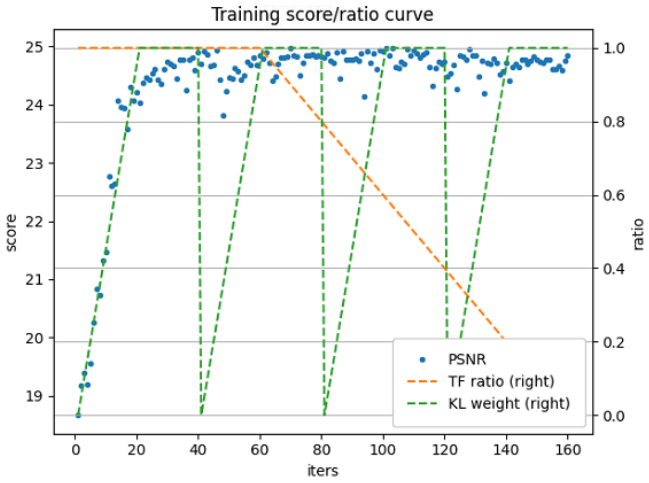
而調整 KL weight (beta) 策略 KL annealing，可分為 monotonic 和 cyclical：

monotonic 設定為前半部分斜直線上升、後半部分固定為 1
cyclical 就是多次 monotonic 的重複。

在此次 lab 中，KL annealing 使用 cyclical 方法比 monotonic 方法效果可以得到更好的 PSNR，因為當模型訓練較穩定時，KL 會隨著訓練消失趨近於 0，這樣學習到的特徵將不再能夠表達觀測到的數據，透過 cyclical 能解決 KL 消失問題，讓訓練中表現更好。

另外，我發現雖然 kld loss 會影響訓練，但 total loss 受 mse loss 影響比較多，幾乎是和 mse loss 重合。只有 cyclical 當 beta 從 1 回到 0 時，kld loss 會出現波動，如下面 Fixed Prior (Cyclical)的圖，而由於公式關係
loss = mse + kld * beta (此時 beta = 0)，kld loss 並不會對 total loss 產生影響。





■ **Learning rate**

learning rate 一直以來都是訓練很大的關鍵因素， 因為 learning rate 太低，loss function 的變化速度就越慢，容易過擬合，雖然使用低 learning rate 可以確保我們不會錯過任何局部極小值，但也意味著我們將花費更長的時間來進行收斂，特別是在被困在局部最優點的時候。而 learning rate 過高容易發生梯度爆炸， loss 振動幅度較大，模型難以收斂。

而這次 lab 我主要是嘗試不同 teaching forcing ratio 和不同 KL annealing 方法，learning rate 則都是用預設的 0.002 做訓練，來不及嘗試用不同的 learning rate 對訓練有多大影響。

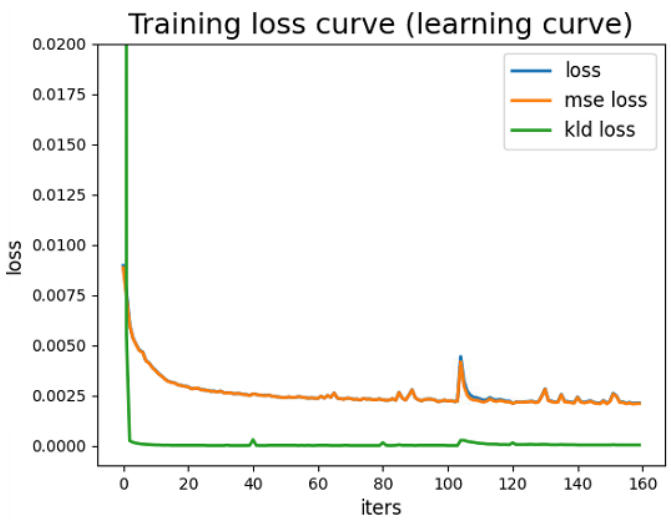
5. Extra

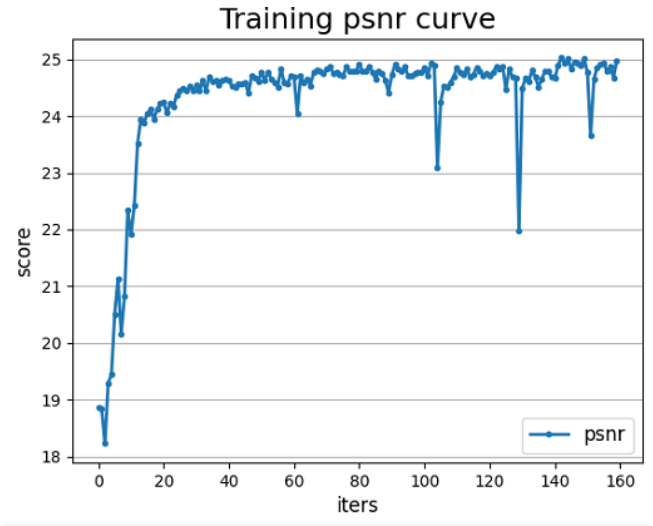
- **Learned prior [1]**

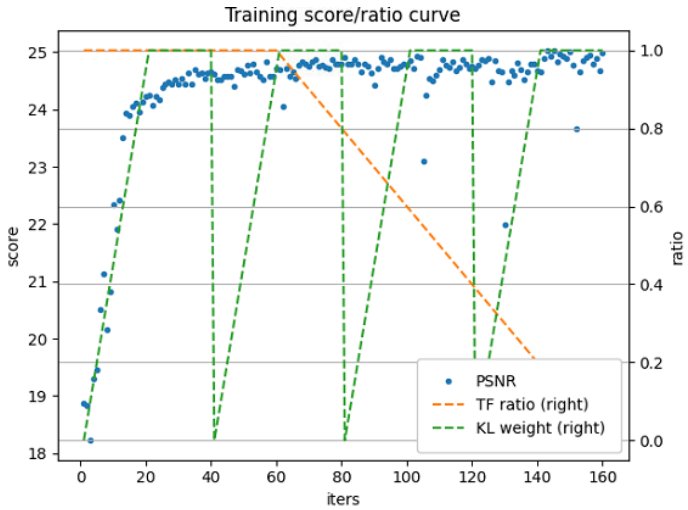
learned prior 的架構在上述的 Main structure (learned prior)有所介紹，而我跑出來的實驗結果和 fixed prior 比較如下：

Learned Prior (Cyclical) : testing PSNR = 25.11

```
pp037@ec037:~/DLP/Lab5$ python3 generate_learned_prior.py
Using pre-trained: best_model/model.pth
Random Seed:  1
Namespace(batch_size=12, beta=0.0001, beta1=0.9, cuda=True, data_root='./data/processed_data', epoch_size=600, g_dim=128, kl_anneal_cycle=4, kl_anneal_cyclical=False, kl_anneal_ratio=2, last_frame_skip=False, log_dir='./best_model', lr=0.002, model_dir='', n_eval=30, n_future=10, n_past=2, niter=160, num_workers=4, optimizer='adam', posterior_rnn_layers=1, predictor_rnn_layers=2, rnn_size=256, seed=1, tfr=1.0, tfr_decay_step=0.01, tfr_lower_bound=0.0, tfr_start_decay_epoch=60, z_dim=64)
100%|                                                                                | 21/21 [00:26<00:00,  1.13s/it]
 avg: 25.117067808522467
100%|                                                                                | 21/21 [00:34<00:00,  1.63s/it]
```

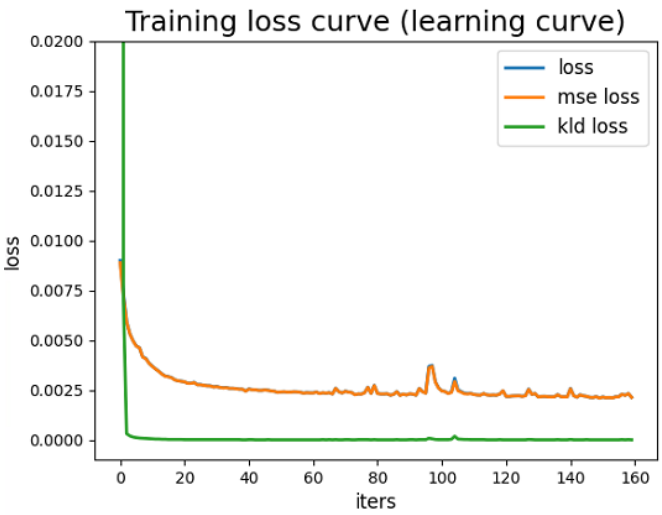


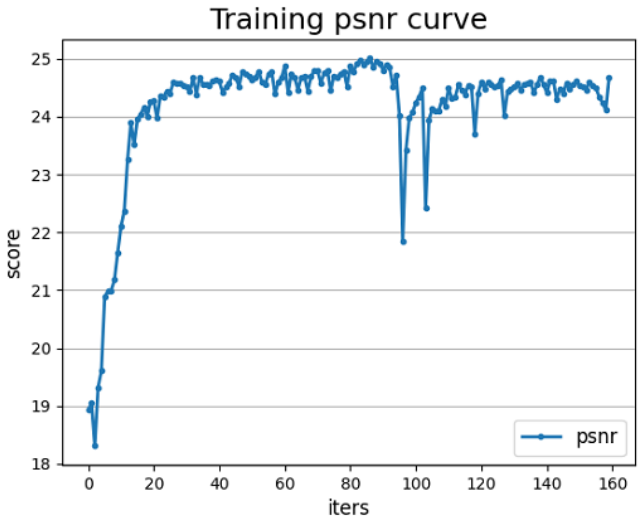




Learned Prior (Monotonic) : testing PSNR = 25.08

```
pp037@ec037:~/DLP/lab5$ python3 generate_learned_prior.py
Using pre-trained: best_model/model.pth
Random Seed:  1
Namespace(batch_size=12, beta=0.0001, beta1=0.9, cuda=True, data_root='./data/processed_data', epoch_size=600, g_dim=128, kl_anneal_cycle=4, kl_anneal_cyclical=False, kl_anneal_ratio=2, last_frame_skip=False, log_dir='./best_model', lr=0.002, model_dir='', n_eval=30, n_future=10, n_past=2, niter=160, num_workers=4, optimizer='adam', posterior_rnn_layers=1, predictor_rnn_layers=2, rnn_size=256, seed=1, tfr=1.0, tfr_decay_step=0.01, tfr_lower_bound=0.0, tfr_start_decay_epoch=60, z_dim=64)
100%|| 21/21 [00:26<00:00,  1.14s/it]
 avg: 25.083296791531343
100%|| 21/21 [00:33<00:00,  1.62s/it]
```





可以發現 learned prior 的 PSNR 比 fixed prior 好，因為 learned prior 有考慮到相連 2 個 frames 的 dependencies，而不是直接假設 prior 為 $N(0, I)$，所以訓練效果會比較好。

■ Conditional convolution [2]

我實作 conditional convolution 加到 vgg decoder，在上面 Implementation details 有詳細介紹，這裡就不加以贅述。