Lab5 Report

Introduction

這個 lab 要用 CVAE 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 都有一個 action 和 endeffector position 可以拿來當作 condition，以利預測。

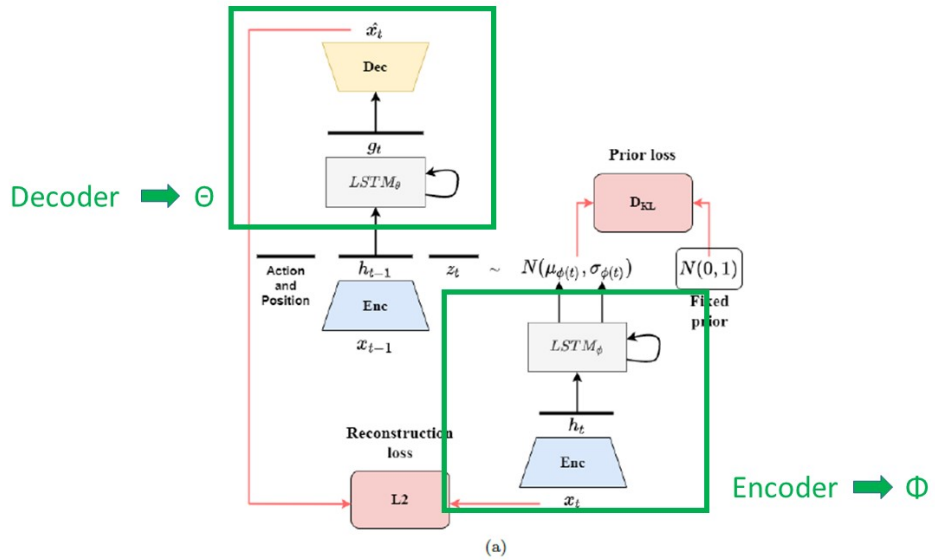
Derivation of CVAE

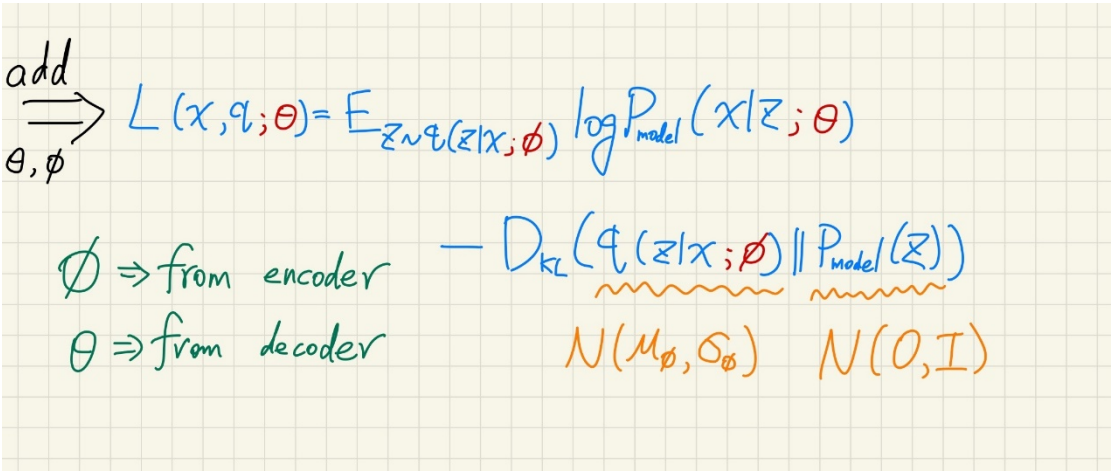
$$L(x,q) = E_{z\sim q(z|x)} \log P_{model}(z|x) + H(q(z|x))$$

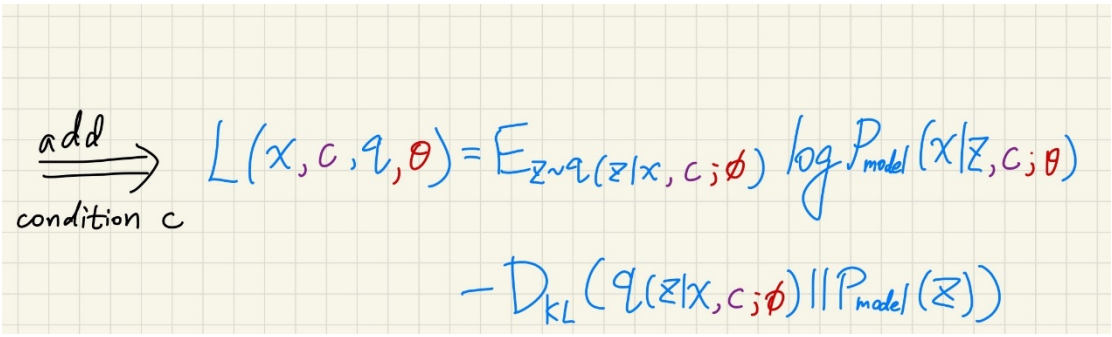
encoder　熵　雜訊

$$= E_{z\sim q(z|x)} \log P_{model}(x|z) - D_{KL}(q(z|x) \| P_{model}(z)) \le \log P_{model}(x)$$

decoder　$N(\mu,\sigma^2)$　$N(0,I)$





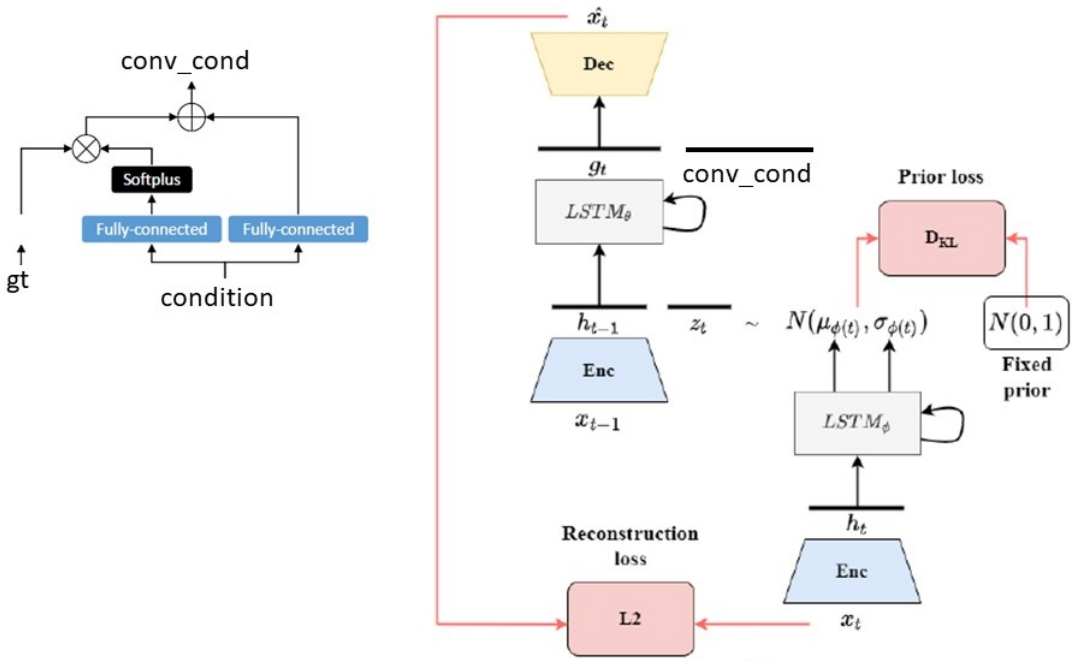


Implementation details

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

Main structure

我實作了 fixed prior 和 learned prior 兩種架構。先介紹 fixed prior 架構，如下圖所示:



首先，先讓這一個 frame (x_t-1)和下一個 frame (x_t)都通過 encoder，分別產生 h_t-1 和 h_t。接著讓 h_t 通過 posterior，即右半邊的 LSTM，產生 mean 和 variance；從這個 mean 和 variance 可以 sample 出 z_t，將 z_t 和 h_t-1 串接，這樣 z_t 就可以帶有下一個 frame 的資訊，並和 h_t-1 一起放入 frame predictor (左半邊 LSTM)，試圖 predict 出下一個 frame 的 latent vector，即 h_t。為了避免與右半邊的 h_t 混亂，這裡 predict 出的 h_t 以 g_t 表示。最後再將 g_t 放入 decoder 解碼還原出圖片。

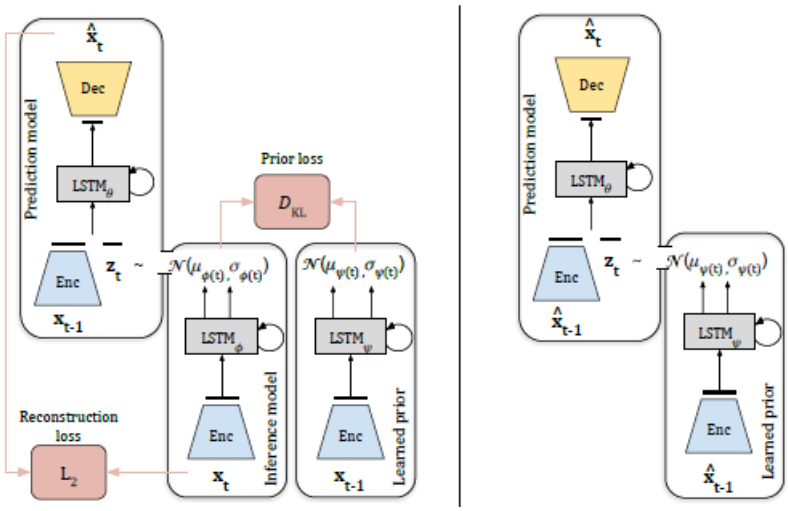
另外，我實作出了 conditional convolution 來幫助 decoder 生成下一個 frame 的圖片。我將 g_t 和 condition 當作 conditional convolution 的 input，output 出 conv_cond，再與原 g_t 串接放入 decoder。

```
        # conditional convolution
        self.cond_fc1 = nn.Linear(7, 128)
        self.cond_fc2 = nn.Linear(7, 128)
        self.sp = nn.Softplus()

    def forward(self, input, skip, cond):
        # conditional convolution
        cond1 = cond
        cond1 = self.cond_fc1(cond1)
        cond1 = self.sp(cond1)
        cond2 = cond
        cond2 = self.cond_fc2(cond2)

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

        # decoder
        d1 = self.upc1(input.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
```

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。encoder 和兩個 LSTM module 都沒有進行修改，就不另外放上截圖。

接下來介紹 learned prior 架構。主要和 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 output 出的 mean、variance 越接近 N(0, I) 越好，即架構途中的 prior loss。這樣做的目的是為了讓 frame predictor 接收到這一

個 frame 與下一個 frame 相關連的訊息，也就是 z_t。而這裡的 z_t 會趨向從 N(0, I) 中 sample，也就是預設兩個相鄰 frame 之間的差距關係是 N(0, I)，而這可能會忽略了他們真正的 dependencies。因此我們可以引入 learned prior 來解決這個問題。以下是從 paper 中擷取的 learned prior 架構圖:



將原本 fixed prior 的 N(0, I) 取代成另一個 encoder + LSTM 架構，並將 x_t-1 作為 input，這樣就能更好的 learned 出兩相鄰 frame 之間的 distribution 關係。在 generation 時，z_t 也從原本拿 N(0, I) 變成由 Learned prior 給定。

綜上所述，我的 training function 和 predicting function 如下:

training:

Fixed prior

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

    mse_criterion = nn.MSELoss()

    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    mse = 0
    kld = 0
    use_teacher_forcing = True if random.random() < args.tfr else False

    # calculate full sequence
    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_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]

        # gaussian and latent vector
        z_t, mu, logvar = modules['posterior'](h_target)
        h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1))

        if use_teacher_forcing:
            x_pred = modules['decoder'](h_target, skip, cond[i-1])
        else:
            x_pred = modules['decoder'](h_pred, skip, cond[i-1])

        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.detach(
```

Learned prior

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

    mse_criterion = nn.MSELoss()

    # 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()
    mse = 0
    kld = 0
    use_teacher_forcing = True if random.random() < args.tfr else False


    # calculate full sequence
    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_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]

        # gaussian 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))

        if use_teacher_forcing:
            x_pred = modules['decoder'](h_target, skip, cond[i-1])
        else:
            x_pred = modules['decoder'](h_pred, skip, cond[i-1])

        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.detach(
```

predicting:

Fixed prior

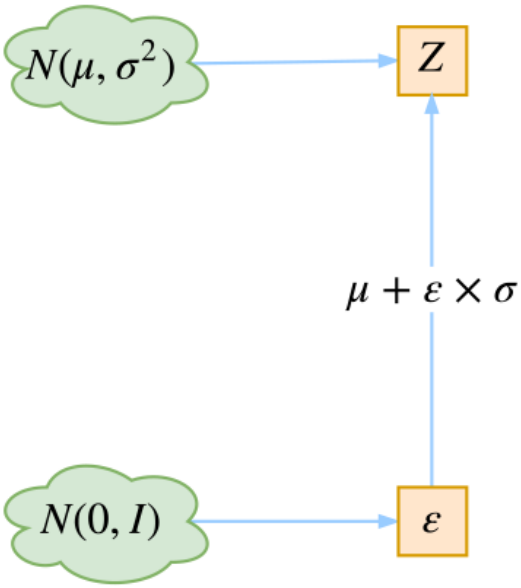
```python
def pred(validate_seq, validate_cond, modules, args, device):
    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()

    pred_seq = []
    for i in range(args.n_past):
        pred_seq.append(validate_seq[i])

    # calculate full given sequence
    h_seq = [modules['encoder'](validate_seq[i]) for i in range(args.n_past)]
    h, skip = h_seq[-1]

    for i in range(args.n_future):
        # h_t
        h = h_seq[-1][0]
        z_t = torch.cuda.FloatTensor(args.batch_size, args.z_dim).normal_()
        h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1))
        x_pred = modules['decoder'](h_pred, skip, validate_cond[args.n_past - 1 + i])
        h_seq.append(modules['encoder'](x_pred))
        pred_seq.append(x_pred)

    return pred_seq
```

Learned prior

```python
def pred(validate_seq, validate_cond, modules, args, device):
    # 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()

    pred_seq = []
    for i in range(args.n_past):
        pred_seq.append(validate_seq[i])


    # calculate full given sequence
    h_seq = [modules['encoder'](validate_seq[i]) for i in range(args.n_past)]
    h, skip = h_seq[-1]

    for i in range(args.n_future):
        # h_t
        h = h_seq[-1][0]
        #z_t = torch.cuda.FloatTensor(args.batch_size, args.z_dim).normal_()
        z_t, _, _ = modules['prior'](h)
        h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1))
        x_pred = modules['decoder'](h_pred, skip, validate_cond[args.n_past - 1 + i])
        h_seq.append(modules['encoder'](x_pred))
        pred_seq.append(x_pred)

    return pred_seq
```

Reparameterization trick

在訓練過程中會涉及到從 N(mean, variance) 中做 sample。從 N(μ, σ^2) 中 sample 一個 Z，相當於從 N(0, I) 中 sample 一個 ε ，然後讓 Z = μ + ε × σ。如下圖所示:



code 如下。要注意這裡傳入的是 log variance，要先進行轉換。

```
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)
```

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。

```python
class dataset(Dataset):
    def __init__(self, args, mode='train', transform=default_transform):
        assert mode == 'train' or mode == 'test' or mode == 'validate'

        self.mode = mode
        if mode == 'train':
            self.data_dir = 'train'
            self.ordered = False
            self.seq_len = args.n_past + args.n_future
        elif mode == 'validate':
            self.data_dir = 'validate'
            self.ordered = True
            #self.seq_len = args.n_eval
            self.seq_len = args.n_past + args.n_future
        else:
            self.data_dir = 'test'
            self.ordered = True
            self.seq_len = args.n_past + args.n_future


        self.dirs = []
        for d1 in os.listdir(self.data_dir):
            for d2 in os.listdir('%s/%s' % (self.data_dir, d1)):
                self.dirs.append('%s/%s/%s' % (self.data_dir, d1, d2))  # ex: train/traj_512_to_767.tfrecords/0

        self.seed_is_set = False
        self.d = 0
        self.d_con = 0
        self.transform = transform
```

get sequence 部分就是取得一個 sequence、12 張圖片的檔案，存成一個 list 並回傳。圖片用 PIL 的 Image 來開啟再用 torchvision transform 中的 ToTensor 轉換成 tensor 型態。另外，我們讓 training mode 隨機選擇 sequence，但 validate mode 和 test mode 會要求他每次都能固定跑過 dataset 中的每一個 sequence。因此需要一個變數儲存當下的 sequence file 路徑。

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

        image_seq = []
        for i in range(self.seq_len):
            fname = '%s/%d.png' % (d, i)
            img = Image.open(fname)
            img = self.transform(img)
            image_seq.append(img)

        self.d_con = d
        return image_seq
```

get condition 是開啟 sequence file 資料夾中的 actions.csv 和 endeffector_position.csv 檔，要開啟的 sequence file 路徑由前面 get sequence 儲存的路徑決定。和 get sequence 一樣取得 12 個 frame 對應的 action 和 position，存成一個 list 並回傳。

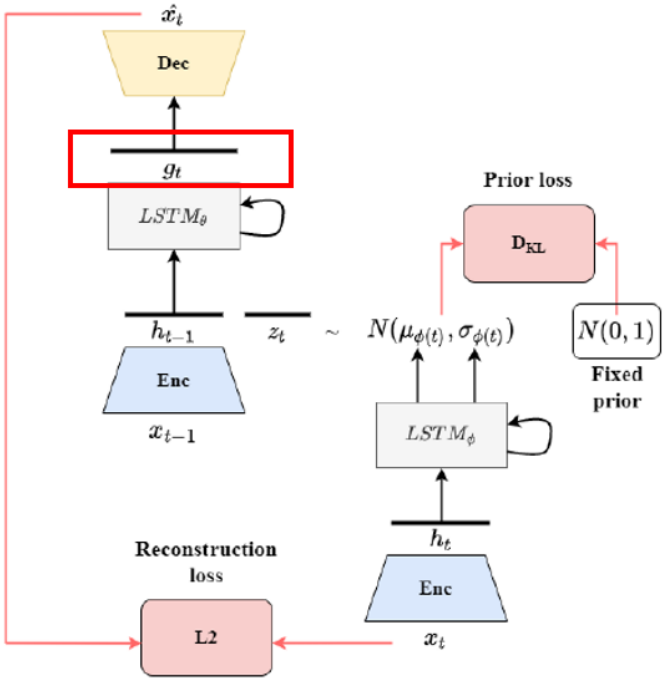
```
def get_csv(self):
    file = open(str(self.d_con) + '/actions.csv', 'r')
    data = loadtxt(file, delimiter=',')

    file2 = open(str(self.d_con) + '/endeffector_positions.csv', 'r')
    data2 = loadtxt(file2, delimiter=',')

    condition = []
    for i in range(self.seq_len):
        act_and_pos = np.concatenate((data[i], data2[i]), axis=None)
        act_and_pos = torch.FloatTensor(act_and_pos)
        condition.append(act_and_pos)

    return condition
```

最後 get item 整合上述函式，供 pytorch 中的 DataLoader 使用。

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

    return seq, cond
```

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

Teacher forcing ratio (簡稱 tfr) 是一種用來加速訓練 decoder 的方法。先注意下圖紅色框框部分，也就是 decoder 的 input。



以架構上來說，decoder 的 input g_t 理應是上一個 frame 的 h_t-1 經過 frame predictor 的 output，而 frame predictor 的 output 就是試圖預測下一個 frame，也就是**仿 h_t**。但我們提供一定的機率讓 decoder 直接拿**真 h_t** 來當作 input。這是為了在訓練前期能先確保 decoder 依 latent vector (即 h) 生成圖片的能力。因此，我們會讓 tfr 從訓練初期的高、隨著訓練進行而慢慢下降。

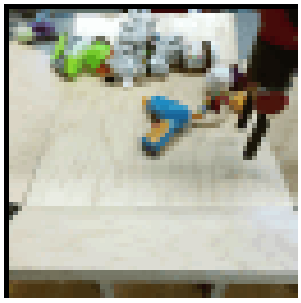
tfr 的優點如上述，會先確保 decoder 的生成能力，讓中後期更容易訓練，對整體的訓練也有加速效果。缺點就是前期會比較難訓練到 decoder 以外的 module。

Results and discussion

Show results of video prediction

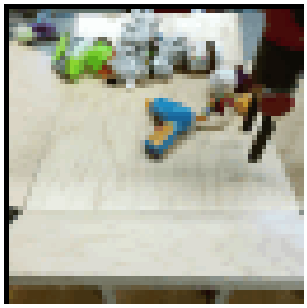
(a) Make videos or gif images for test result (select one sequence)

truth:



雲端連結

prediction:



雲端連結

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

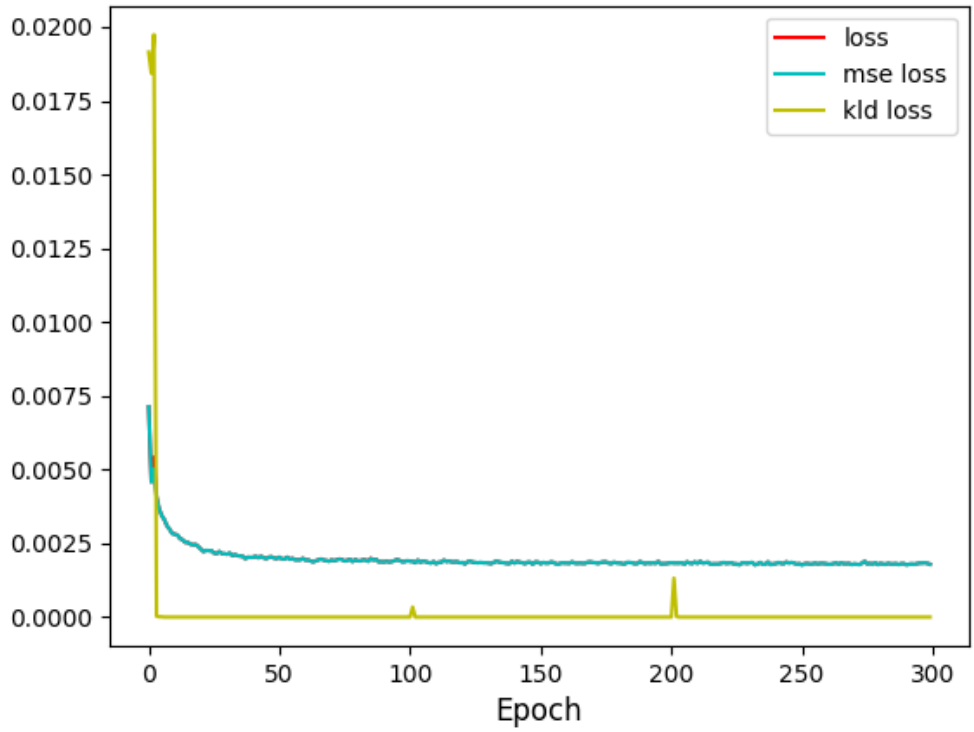
Seq	1	2	3	4	5	6
Truth						
Predict						

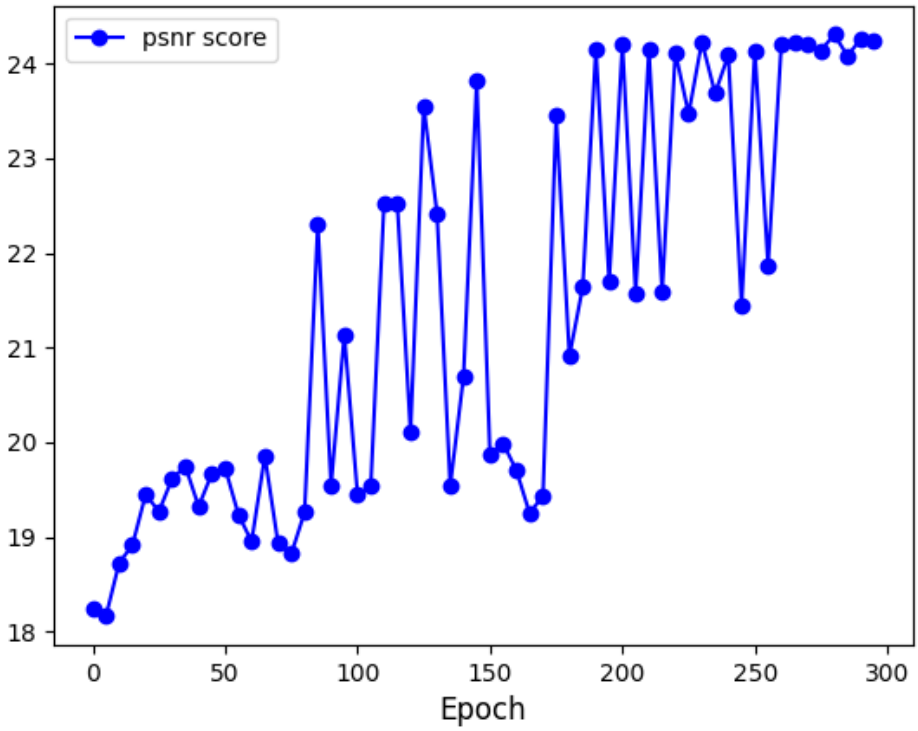
Seq	7	8	9	10	11	12
Truth						
Predict						

P.S. seq1 和 seq2 為給定的 input，因此 ground truth 和 prediction 會長得一樣。從 seq3 以後 prediction 的機器手臂邊界就明顯比 truth 模糊。

Plot the KL loss and PSNR curves during training

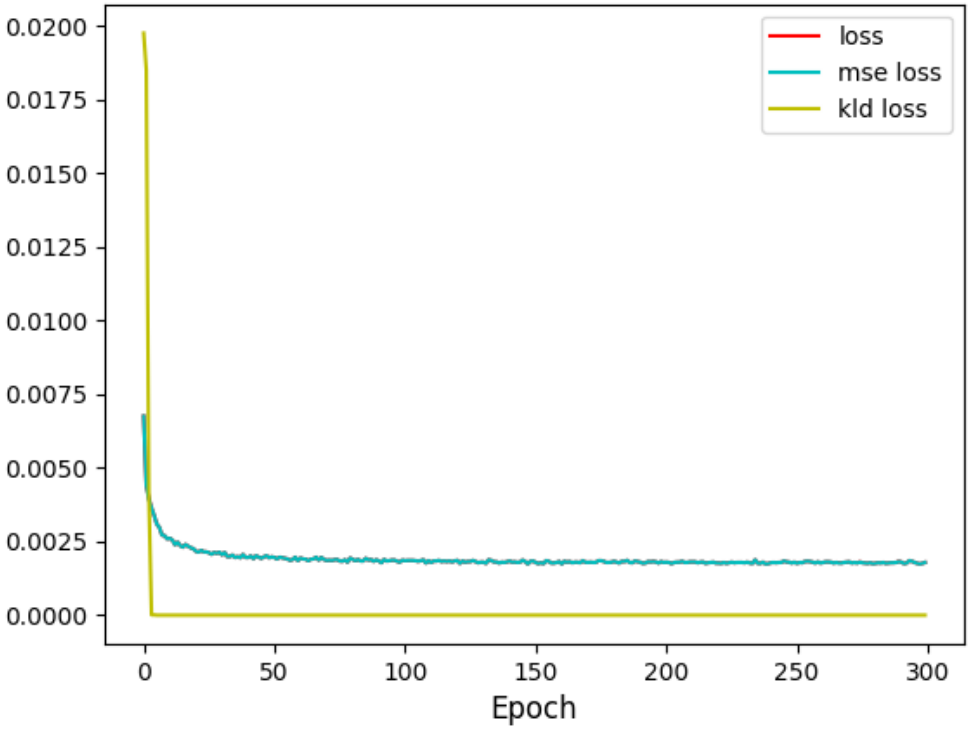
Learned Prior with KL cyclical annealing:

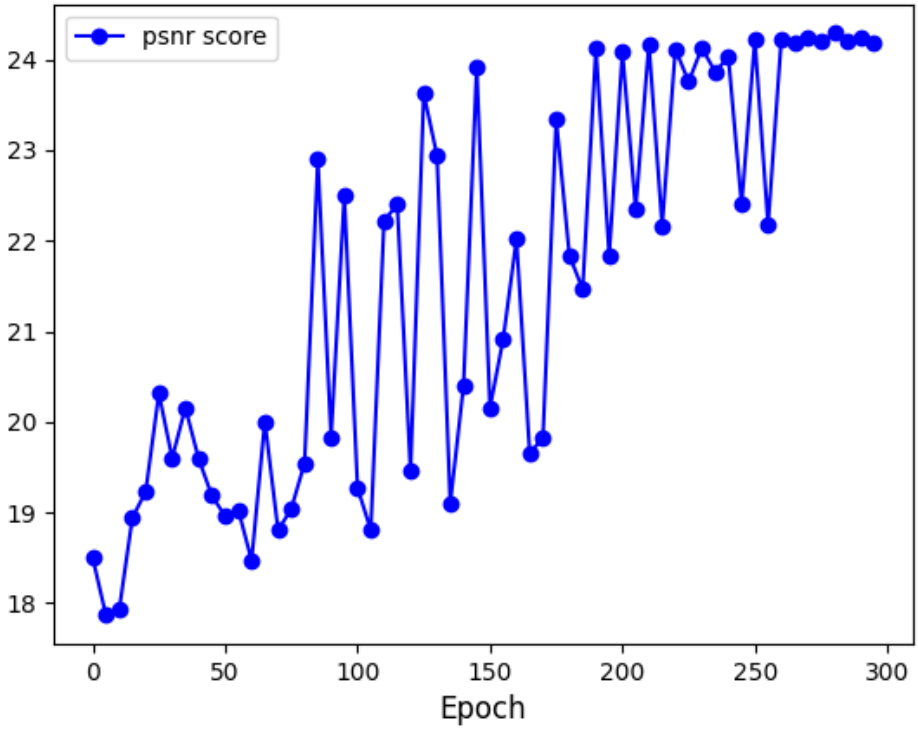




Max psnr score = 24.30334

Learned Prior with KL monotonic annealing:

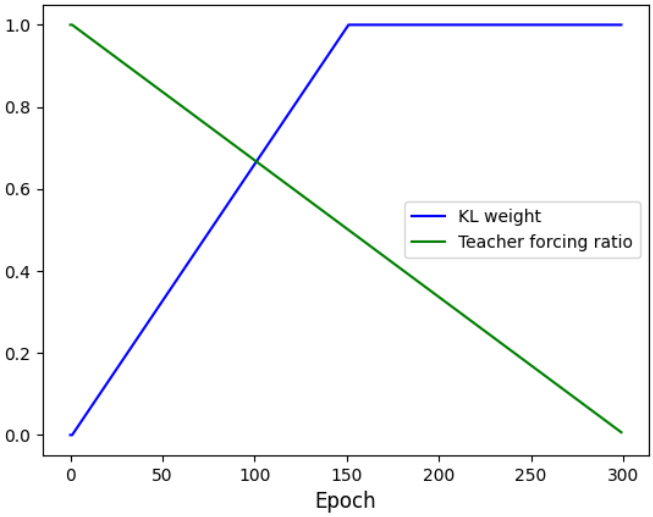




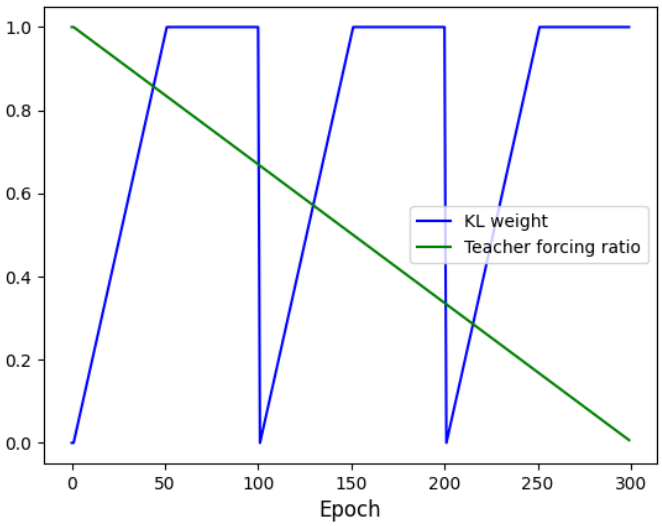
Max psnr score = 24.29078

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

KL monotonic annealing:



KL cyclical annealing:



根據上一個部分所述，tfr 的設定必須隨著訓練進行由大到小。因此我直接設定為一條過 $(0, 1.0)$、$(300, 0.0)$ 的斜直線。KL annealing 則是分為 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，如上圖所示。

我們可以從 loss curve 和 psnr curve 中看出，兩種 KL annealing 對我的結果影響並不大，基本上圖形都在非常相似的狀態，最好的 model 的 psnr score 一個為 24.30、一個為 24.29，也相差無幾。

原本 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。但在此 lab 中，我發現 KL loss 過了前幾個 epoch 後就變成 0 了，導致 total loss 幾乎從頭到尾都和 mse loss 重合，如上圖所示。只有 cyclical 當 beta 從 1 回到 0 時，KL loss 會出現小幅波動，如下圖。我認為原因和 pytorch 中的 loss back propagate 的計算有關。