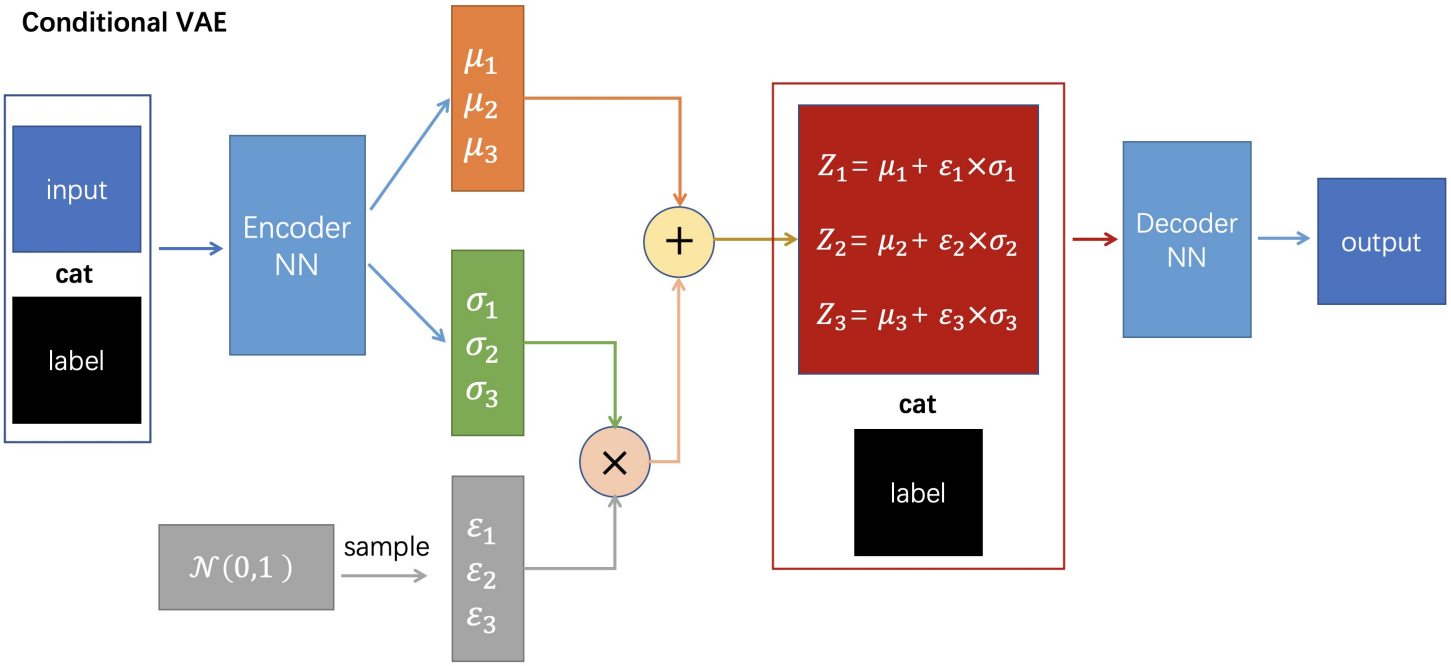
Conditional VAE For Video Prediction

• Introduction (5%)

在這個實驗中，要將CVAE應用於視頻的預測。

CVAE可以在生成數據時通過指定其標籤來生成想生成的數據。 CVAE的結構圖如下所示：

整體結構和VAE差不多，區別是在將數據輸入Encoder時把數據內容與其標籤(label)合併(cat)一起輸入，將編碼(Z)輸入Decoder時把編碼內容與數據標籤(label)合併(cat)一起輸入。

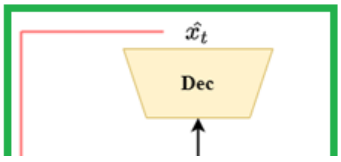


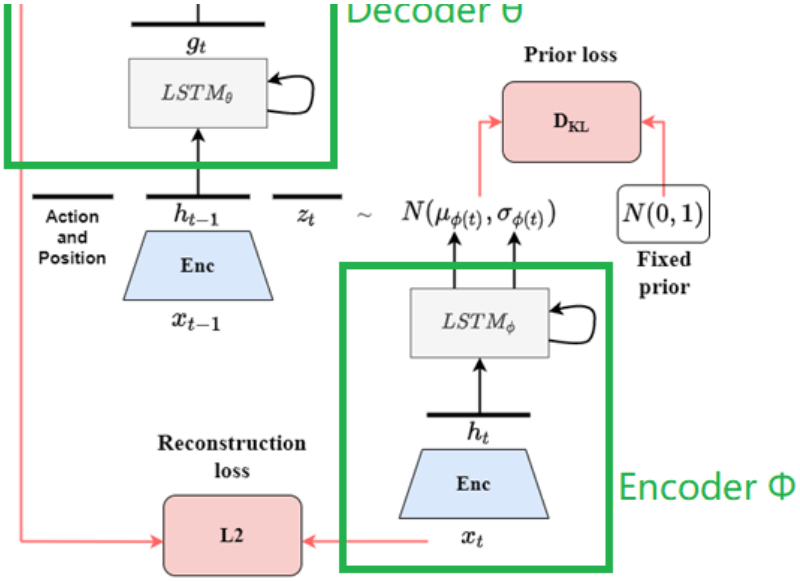
在實驗中，dataset 為機器人動作 影片，每個 sequence 切成 30 個 frame，存成 30 個 64*64 的 png 圖 片檔。我們的目標為給前 2 個 frame，並利用 VAE generate 的特性預測出後 10 個 frame。另外，dataset 中對於每個 frame 都有一個 action 和 endeffector position 可以拿來當作 condition，以利預測。

• Derivation of CVAE (Please use the same notation in Fig.1a)(10%)

$$L(x,q) = E_{z\sim q(z|x)} \log \underbrace{P_{model}(z|x)}_{encoder} + \underbrace{H(q(z|x))}_{\text{熵 雜訊}}$$

$$= E_{z\sim q(z|x)} \log \underbrace{P_{model}(x|z)}_{decoder} - D_{KL}\left(\underbrace{q(z|x)}_{N(\mu,\sigma^2)} \| \underbrace{P_{model}(z)}_{N(0,I)}\right) \le \log P_{model}(x)$$





add θ, ϕ $\Rightarrow$ $L(x, q; \theta) = E_{z \sim q(z|x;\phi)} \log P_{model}(X|z;\theta) - D_{KL}(q(z|x;\phi) \| P_{model}(z))$

$\phi \Rightarrow$ from encoder

$\theta \Rightarrow$ from decoder

$N(\mu_\phi, \sigma_\phi)$ $N(0, I)$

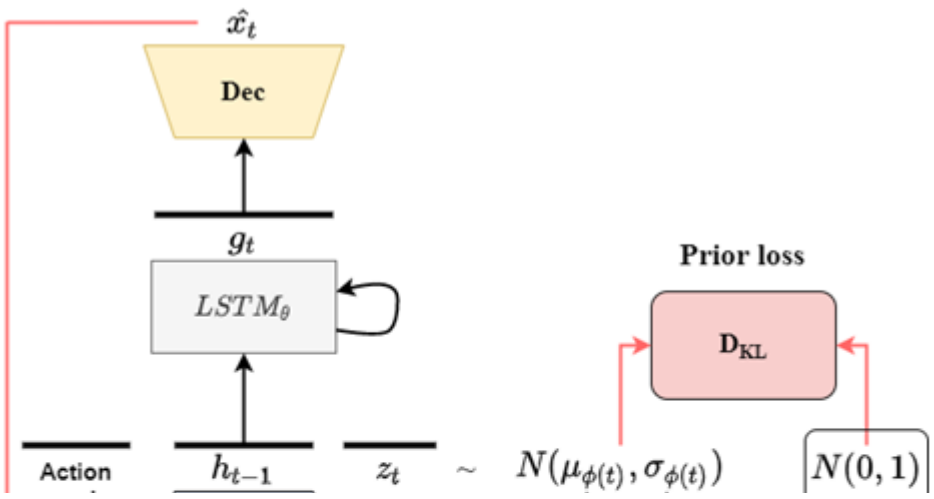
add condition c $L(x, c, q, \theta) = E_{z \sim q(z|x,c;\phi)} \log P_{model}(X|z,c;\theta) - D_{KL}(q(z|x,c;\phi) \| P_{model}(z))$

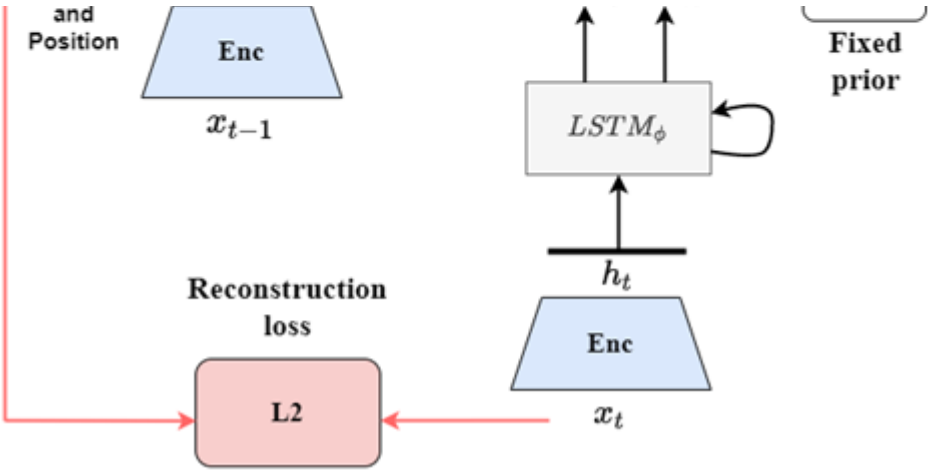
• Implementation details (15%)

escribe how you implement your model (encoder, decoder, reparameterization trick, dataloader, etc.). (10%)

本實驗實作了 fixed prior 和 learned prior 兩種架構。

fixed prior:





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。

encoder

```
encoder = vgg_encoder(args.g_dim)
```

```
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
        h5 = h5.view(-1, self.dim) # size = (12 x 128)

        return h5, [h1, h2, h3, h4]
```

decoder

```
decoder = vgg_decoder(args.g_dim + 128)
```

```
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, 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
```

training function

```
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)
```

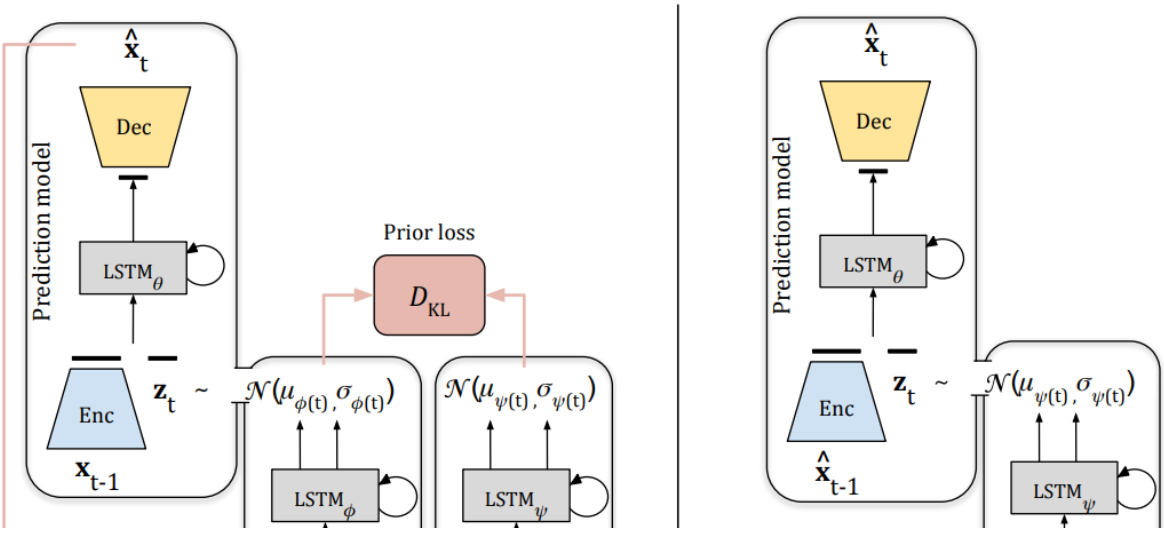
```
        h_pred = modules['frame_predictor'](torch.cat([h, z_t], 1))
        if use_teacher_forcing:
            x_pred = modules['decoder'](h_target, skip, cond[i-1])
        else:
            x_pred = modules['decoder'](h_pred, skip, cond[i-1])
        mse += mse_criterion(x_pred, x[i])
        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().cpu().numpy() /
(args.n_past + args.n_future), kld.detach().cpu().numpy() / (args.n_future + args.n_past)
```

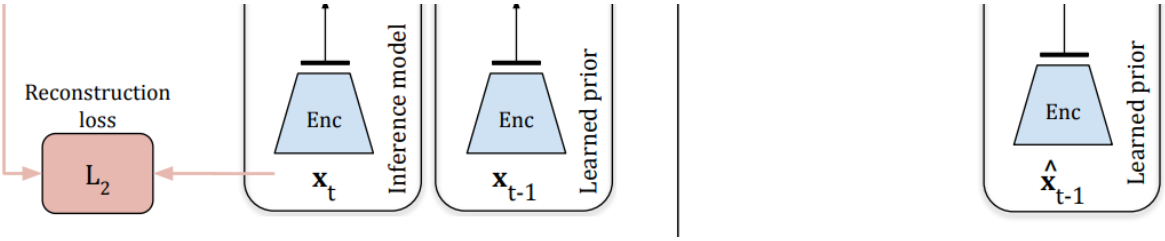
predicting function

```
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:





learned prior 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 output 出的 mean、variance 越接近 N(0, I) 越好，即架構圖的 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，找出他們之間真實dependencies的關係。
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。

training function

```
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().cpu().numpy() /
(args.n_past + args.n_future), kld.detach().cpu().numpy() / (args.n_future + args.n_past)
```

predicting function

```
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
```

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。

initialize

```
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
```

```
            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

```
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

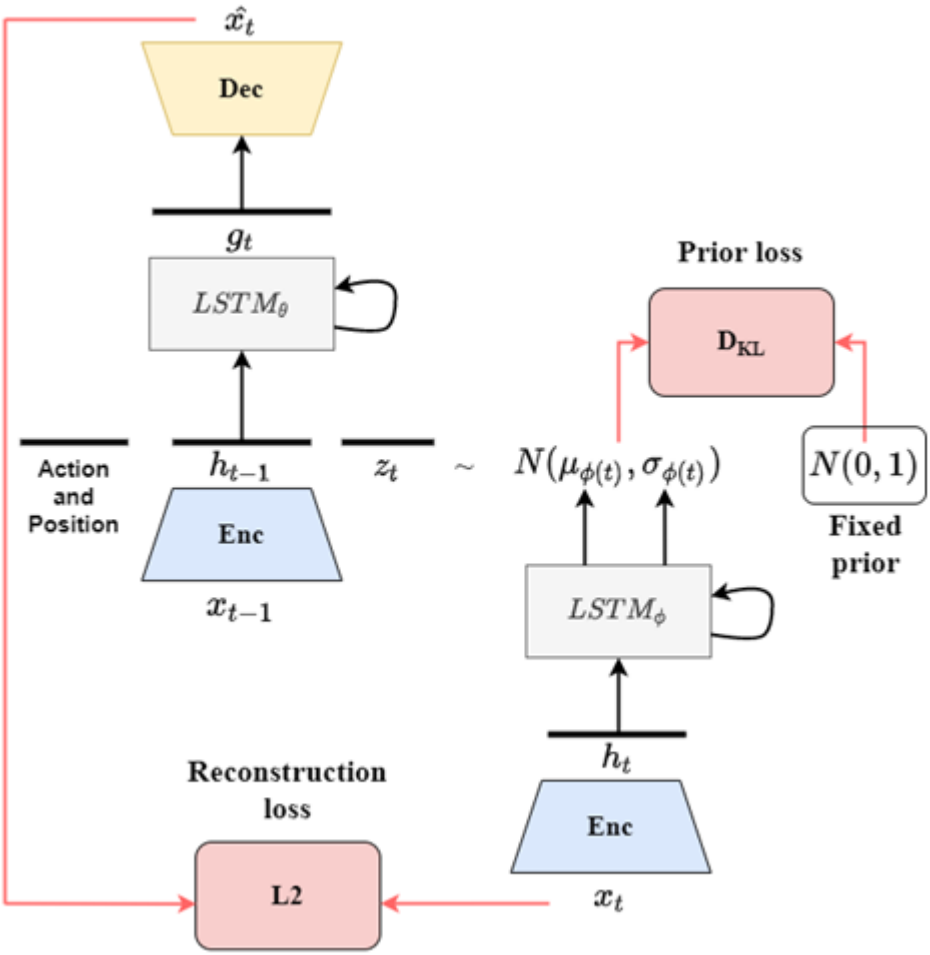
```
def get_csv(self):
        file = open(str(self.d_con) + '/actions.csv', 'r')
        data = loadtxt(file, delimiter=',')
        file2 = open(str(self.d_con) + '/endeffector_positions.csv', 'r')
        data2 = loadtxt(file2, delimiter=',')

        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
```

get item

```
def __getitem__(self, index):
        self.set_seed(index)
        seq = self.get_seq()
        cond = self.get_csv()
        if self.mode == 'test':
            return seq, cond, self.d_con
        else:
            return seq, cond
```

Describe the teacher forcing (including main idea, benefits and drawbacks.) (5%)

Teacher forcing ratio (簡稱 tfr) 是一種用來加速訓練 decoder 的方法。



decoder 的 input g_t 理應是上一個 frame 的 h_{t-1} 經過 frame predictor r (左半邊 LSTM)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 (30%)

– Show your results of video prediction (10%)

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

truth:

https://drive.google.com/file/d/1G5W3fGrtpO6RKZpX3HlFfWEsCjK080Lu/view?usp=sharing

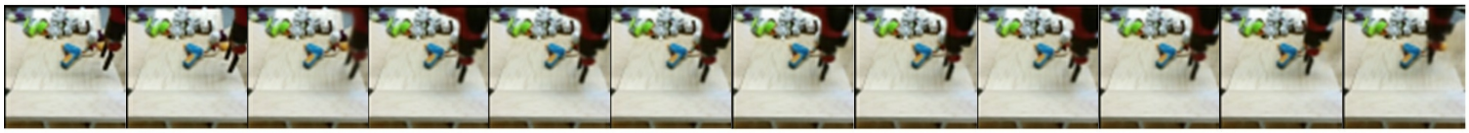
prediction:

https://drive.google.com/file/d/1CwfQE0mA_pEYpD_sj-m2Orgk4vPeuNtl/view?usp=sharing

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

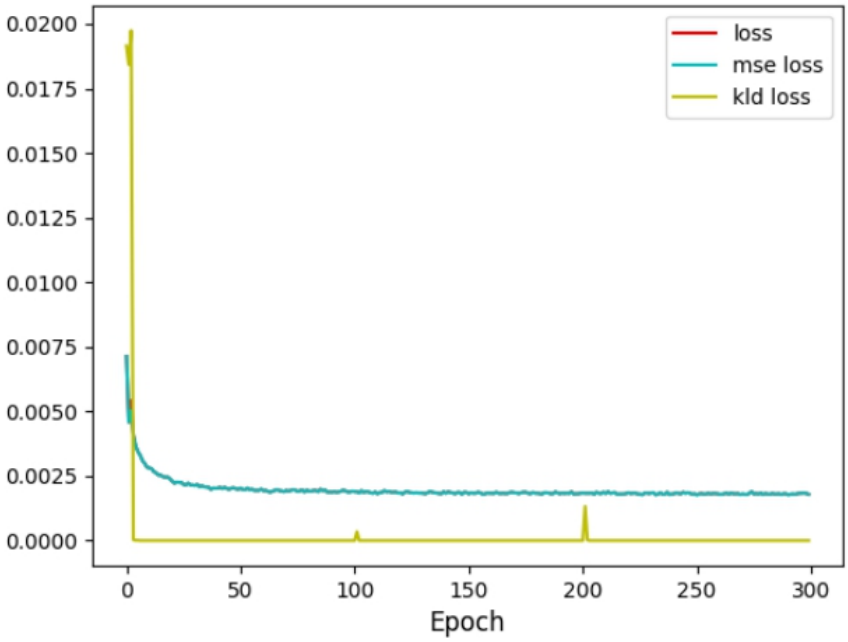
truth:

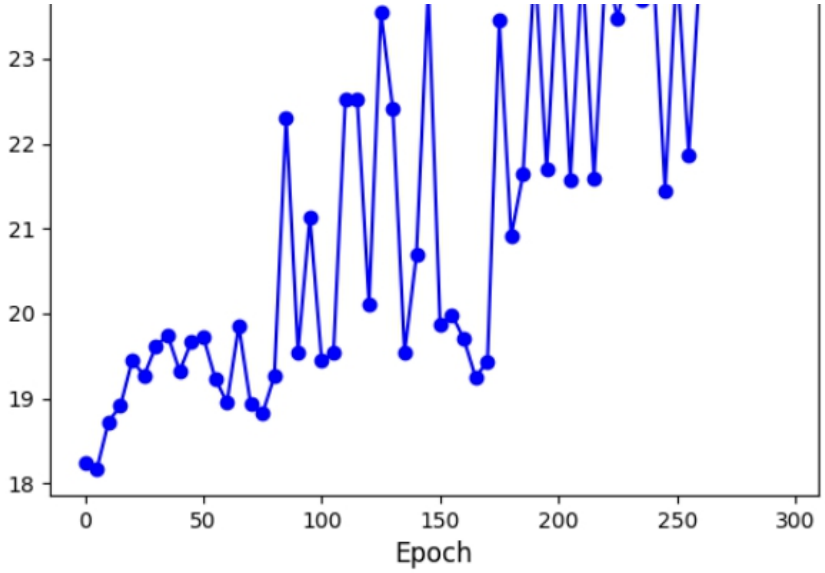
prediction:



– Plot the KL loss and PSNR curves during training (5%)

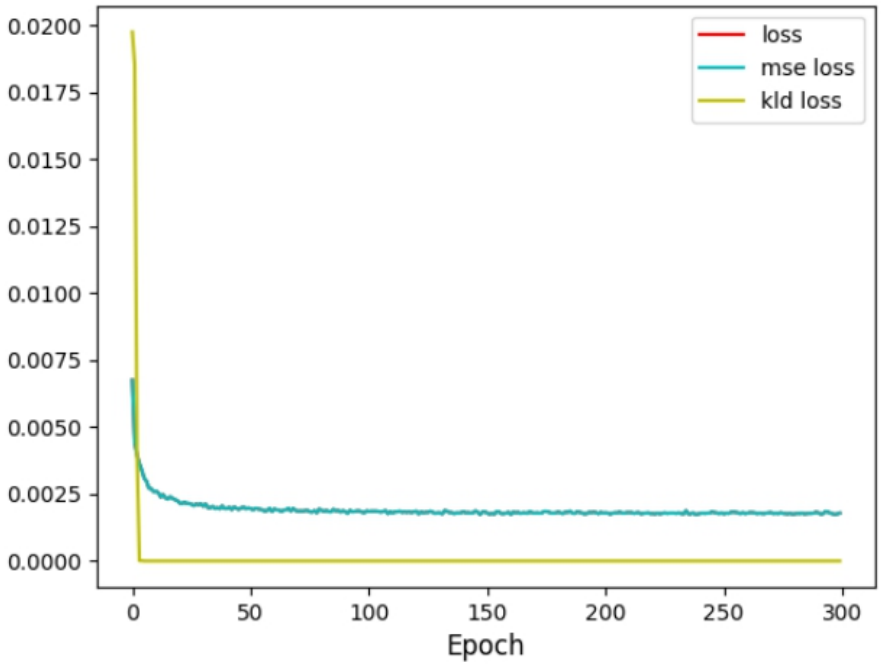
Learned Prior with KL cyclical annealing:

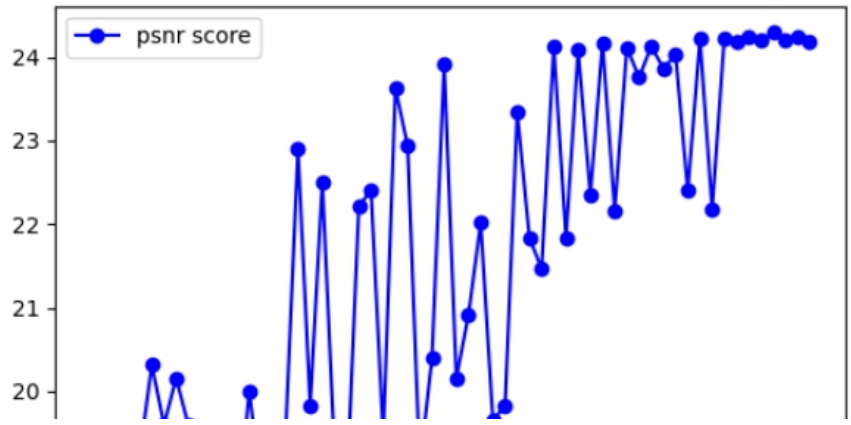


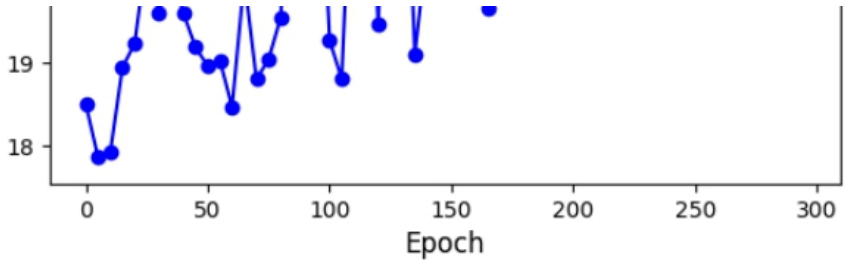


Max psnr score = 24.30334

Learned Prior with KL monotonic annealing:



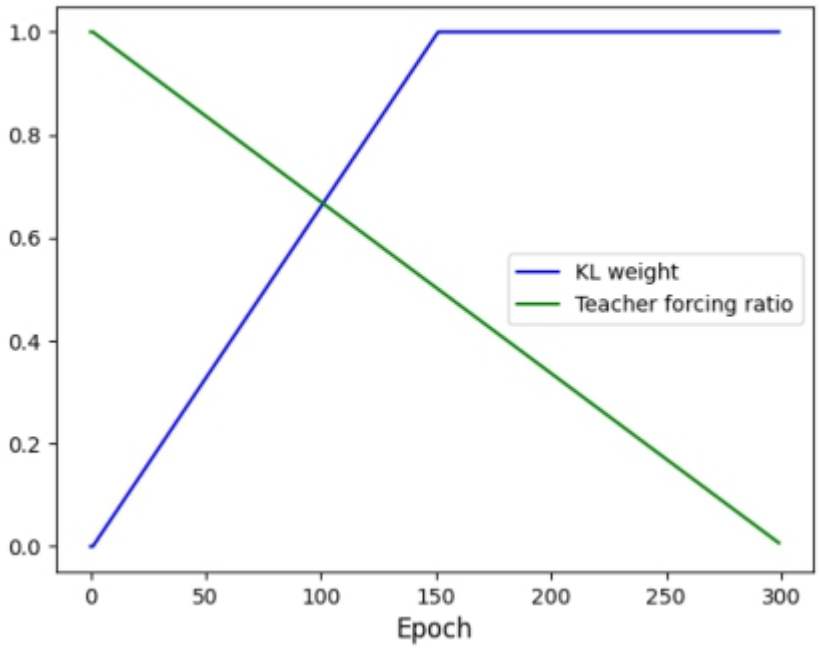




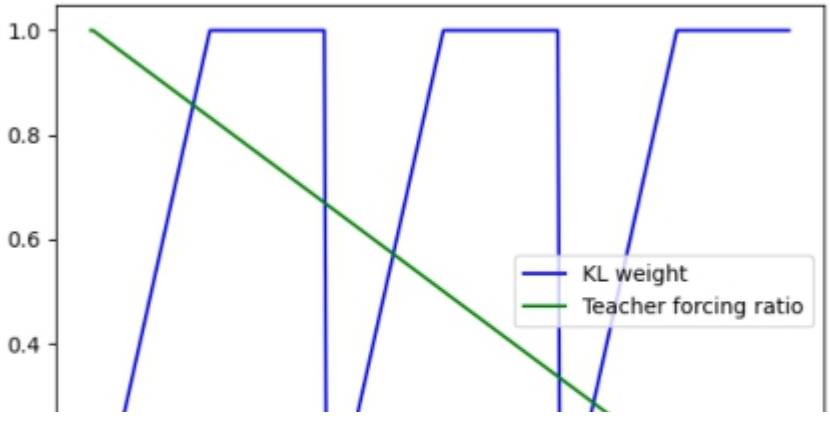
Max psnr score = 24.29078

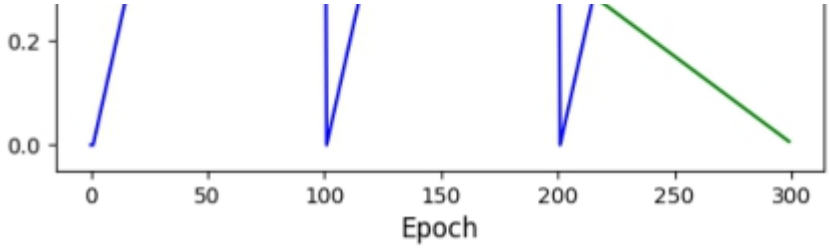
– Discuss the results according to your setting of teacher forcing ratio, KL weight, and learning rate. Note that this part mainly focuses on your discussion, if you simply just paste your results, you will get a low score. (15%)

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。
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。