Deep Learning Lab 5

311552004 李明倫

1. Introduction

In this lab, you need to implement a conditional VAE for video prediction. Your model should be able to do prediction based on past frames.

For example, when we input frame x_{t-1} to the encoder, it will generate a latent vector h_{t-1}. Then, we will sample z_t from fixed prior distribution. Eventually, we take the output from the encoder (h_{t-1}) and z_t with the condition (action and position) as the input for the decoder and we expect that the output frame should be next frame x_t.

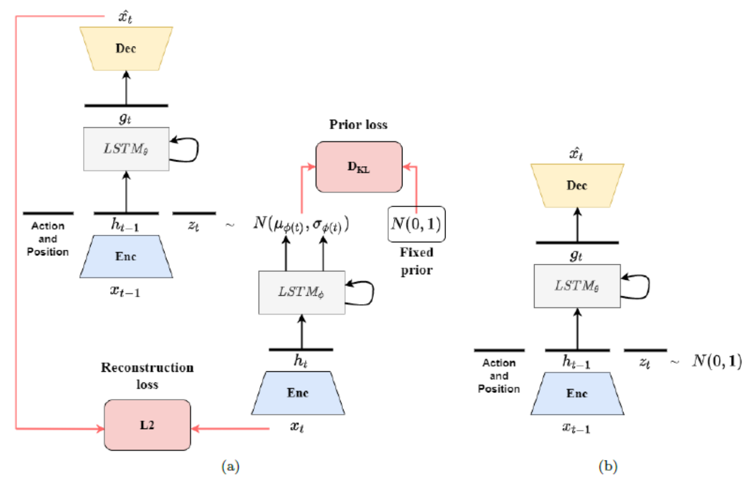


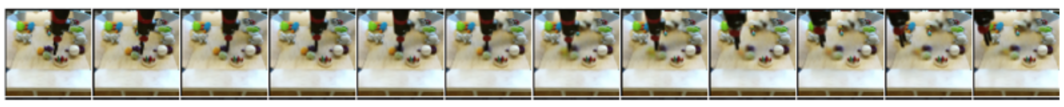
Figure 1: The illustration of overall framework. (a) Training procedure (b) Generation procedure

Requirements

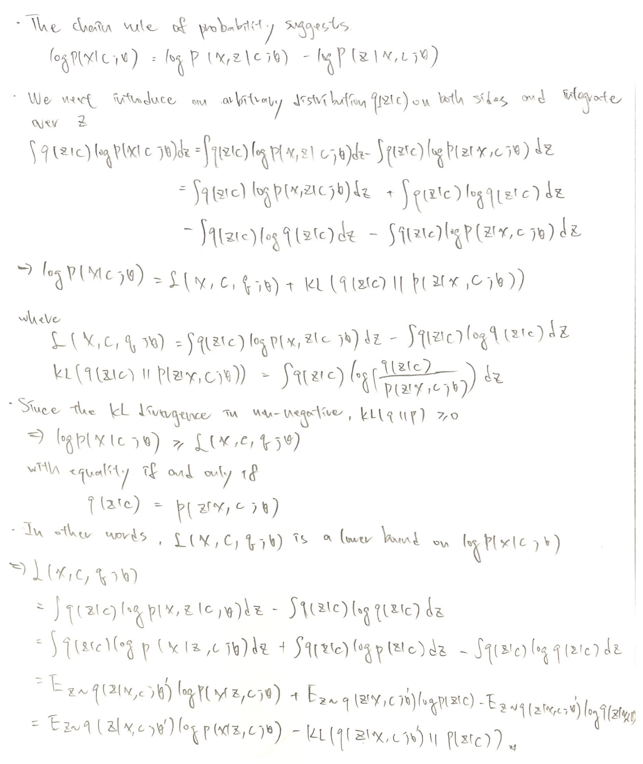
- Implement a conditional VAE model
    - Modify training functions
    - Implement dataloader, teacher forcing, KL annealing, and reparameterization trick
- Plot the training loss and PSNR curves during training
    - Teacher forcing ratio
    - KL annealing schedules (Monotonic / Cyclical)
- Make videos or gif images for test result
- Output the prediction at each time step

Dataset

We use bair robot pushing small dataset to train CVAE. This data set contains roughly 44000 sequences of robot pushing motions, and each sequence include 30 frames. In addition, it also contains action and end-effector position for each time step.



2. Derivation of CVAE



- The chain rule of probability suggests

$$\log p(x|c;\theta) = \log p(x,z|c;\theta) - \log p(z|x,c;\theta)$$

- We next introduce an arbitrary distribution $q(z|c)$ on both sides and integrate over z

$$\int q(z|c)\log p(x|c;\theta)dz = \int q(z|c)\log p(x,z|c;\theta)dz - \int q(z|c)\log p(z|x,c;\theta)dz$$

$$= \int q(z|c)\log p(x,z|c;\theta)dz + \int q(z|c)\log q(z|c)dz$$

$$- \int q(z|c)\log q(z|c)dz - \int q(z|c)\log p(z|x,c;\theta)dz$$

$$\Rightarrow \log p(x|c;\theta) = \mathcal{L}(x,c,q;\theta) + KL(q(z|c)\,||\,p(z|x,c;\theta))$$

where

$$\mathcal{L}(x,c,q;\theta) = \int q(z|c)\log p(x,z|c;\theta)dz - \int q(z|c)\log q(z|c)dz$$

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

- Since the KL divergence in non-negative, $KL(q||p) \geq 0$

$$\Rightarrow \log p(x|c;\theta) \geq \mathcal{L}(x,c,q;\theta)$$

with equality if and only if

$$q(z|c) = p(z|x,c;\theta)$$

- In other words, $\mathcal{L}(x,c,q;\theta)$ is a lower bound on $\log p(x|c;\theta)$

$$\Rightarrow \mathcal{L}(x,c,q;\theta)$$

$$= \int q(z|c)\log p(x,z|c,\theta)dz - \int q(z|c)\log q(z|c)dz$$

$$= \int q(z|c)\log p(x|z,c;\theta)dz + \int q(z|c)\log p(z|c)dz - \int q(z|c)\log q(z|c)dz$$

$$= E_{z\sim q(z|x,c;\theta')}\log p(x|z,c;\theta) + E_{z\sim q(z|x,c;\theta')}\log p(z|c) - E_{z\sim q(z|x,c;\theta')}\log q(z|x,c;\theta)$$

$$= E_{z\sim q(z|x,c;\theta')}\log p(x|z,c;\theta) - KL(q(z|x,c;\theta')\,||\,p(z|c))$$

3. Implementation details

Describe how you implement your model

- **DataLoader**

負責讀取 image sequence 與對應的 action, end effector position，在 `__get_item__()` 的時候傳回

```
class bair_robot_pushing_dataset(Dataset):
    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 = args.n_past + args.n_future
        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]

    def set_seed(self, seed):
        if not self.seed_is_set:
            self.seed_is_set = True
            np.random.seed(seed)

    def __len__(self):
        return len(self.dirs)

    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

    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

    def __getitem__(self, index):
        self.set_seed(index)
        seq = self.get_seq()            # 拿到一整個 sequence 的 image
        cond =  self.get_csv()          # 拿到一整個 sequence 的 action 跟 endeffector_position
        return seq, cond
```

- **Encoder / Decoder**

Encoder 跟 Decoder 都採用類似 VGG Model 的架構，encoder 針對輸入影像取特徵，而 decoder 則試著將 latent vector 去生成影像。其中每一層 vgg_layer 由 convolution, Batchnorm 跟 LeakyRelu 組成。比較特別的是，在 forward pass 的時候會順便保留每一個 section 的 activation output，在生成影像的時候會把這些 “intermediate information” 透過 skip connection 的方式傳給 decoder，讓 decoder 在生成影像的時候能夠取得更多不同層面的細節，讓生成出的效果更好。

```
class vgg_layer(nn.Module):
    def __init__(self, nin, nout):
        super(vgg_layer, self).__init__()
        self.main = nn.Sequential(
                nn.Conv2d(nin, nout, 3, 1, 1),
                nn.BatchNorm2d(nout),
                nn.LeakyReLU(0.2, inplace=True)
                )

    def forward(self, input):
        return self.main(input)
```

```
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]
```

```
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)

    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
```

- **Posterior LSTM**

在這個 lstm 的 module 中，會試著將下一個 frame 取完特徵的 h_t 轉成 $\mu_{(t)}$ 跟 $\sigma_{(t)}$，並從 u 跟 σ 的 gaussian distribution $N(\mu_{(t)}, \sigma_{(t)})$ 去 sample z_t 出來，丟回去給 decoder 去生成下一個時間點的 frame。

```
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

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

    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
```

- **Reparameterization**

z_t 的 sample 則透過 reparameterization trick 去做 sample。

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

- **Frame predictor LSTM**

在這個部分的 lstm module 中，主要是學習產生下一個 time step frame 的 feature vector。condition (action, end_effector position)、current time step 的 embedding feature h_{t-1} 還有 next time step sample 出來的 latent vector z_t，三個併在一起當作輸入進這個 lstm 中。而因為 lstm 會有記憶的功能，所以在最後產生的 output g_t，都會考慮過往時間點的資訊，會得到比較好的 future frame vector。最後再把 outuput 出的 g_t 送進 decoder 去生成影像。

```
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)
```

- **Training**

在這個部分中，實做了 CVAE 的訓練過程，一開始要先將 image sequence `x` 與 `cond` 做維度上的轉換，轉成 `[seq_len, batch, channel, width, height]` 的形式，讓在處理資料的時候從第一個時間點依序往下做處理，而每個 time step 都是以 batch 的方式在處理。

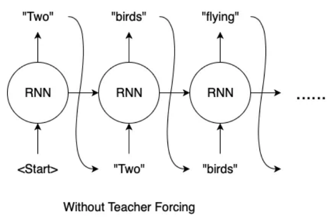
每一個時間點訓練完之後，會檢查是否有 teacher forcing，如果有的話，要把下一個時間點的 ground truth frame 當作輸入，沒有的話則把當前 output 出的 frame 當作下一個時間點的輸入。

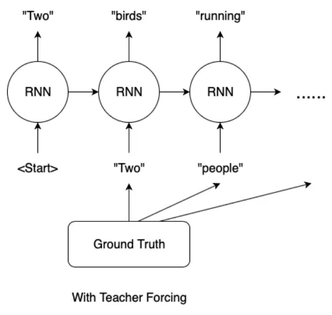
```
def train(x, cond, modules, optimizer, kl_anneal, args):
    modules['frame_predictor'].zero_grad()
    modules['posterior'].zero_grad()
    modules['encoder'].zero_grad()
    modules['decoder'].zero_grad()
    mse_loss = nn.MSELoss()

    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    mse = 0
    kld = 0

    '''
        lstm default: batch_first=False => [seq_len, batch, input_size]
        x.shape: [batch:0, seq_len:1, channel:2, width:3, height:4]
              -> [seq_len:1, batch:0, channel:2, width:3, height:4]

        cond.shape: [batch:0, seq_len:1, action+position:2]
                 -> [seq_len:1, batch:0, action+position:2]
    '''

    x = x.permute(1, 0, 2, 3, 4)
    cond = cond.permute(1, 0, 2)

    use_teacher_forcing = True if random.random() < args.tfr else False
    x_t_1 = x[0]
    for i in range(1, args.n_past + args.n_future):
        x_t = x[i]
        h_t, _ = modules['encoder'](x_t)
        z_t, mu, sigma = modules['posterior'](h_t)

        if args.last_frame_skip or i <= args.n_past:
            h_t_1, skip = modules['encoder'](x_t_1)
        else:
            h_t_1, _ = modules['encoder'](x_t_1)

        cond_t_1 = cond[i-1]

        info = torch.cat((cond_t_1, h_t_1, z_t), axis=1)
        g_t = modules['frame_predictor'](info)
        prediction = modules['decoder']([g_t, skip])

        if use_teacher_forcing:
            x_t_1 = x[i]
        else:
            x_t_1 = prediction

        mse += mse_loss(x_t, prediction)
        kld += kl_criterion(mu, sigma, 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.det
```

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

在訓練 RNN 的時候，我們會需要前一個時間點的 output 當作下一個時間點的 input，去預測下一個時間點的 output。但很有可能我們當前時間點的 output 不一定是好的，那再接著當作下一個時間點的 input 的時候，就會讓下一個時間點的 output 也是不好的，這樣

下去就會讓 model 很難訓練。Teacher forcing 的動機就來自於上述的問題，因此為了避免中間的誤差讓模型難以訓練，因此將正確答案 (ground truth) 當作 input，讓模型知道說在這個正確的輸入底下，應該要生成甚麼樣的資料。





- Benefits

  因為給了模型正確答案當作 input，能夠讓模型不會因為不好的 input 而導致預測的結果更糟，因此可以讓模型加快收斂的速度。

- Drawbacks

  但在實際用來 inference 的時候並不會有 ground truth 的資料，必須輸入上一個 time step 的 output 去預測當前的結果，導致訓練和預測時候是從不同的 distribution 去推論出來的，因此會有 "Exposure Bias" 這種不一致的現象發生。

在前期訓練的時候，先透過 teacher forcing 的方式讓模型能夠快速收斂到某個結果。在訓練一段時間之後，開始讓模型有時候不要用正確答案當作輸入，而是拿前一個時間點的 output 當作輸入，漸漸的把 teacher forcing 的比例調低，以避免 “exposure bias” 的狀況發生，讓模型學著用自己的 output 去預測下一個時間點的結果。

```
slope = (args.tfr_lower_bound - args.tfr) / (args.niter - args.tfr_start_decay_epoch)

if epoch >= args.tfr_start_decay_epoch:         ### Update teacher forcing ratio ###
    tfr = slope * (epoch - args.tfr_start_decay_epoch) + 1
    args.tfr = max(tfr, args.tfr_lower_bound)
```

4. Results and discussion

- test.py

```
test_data = bair_robot_pushing_dataset(args, 'test')
test_loader = DataLoader(test_data,
                        num_workers=4,
                        batch_size=args.batch_size,
                        shuffle=True,
                        drop_last=True,
                        pin_memory=True)

test_psnr = []
with torch.no_grad():
    for i, (seq, cond) in enumerate(test_loader):
        seq, cond = seq.to(device), cond.to(device)
        pred_seq = pred(seq, cond, modules, args, device)

        seq, pred_seq = seq.permute(1, 0, 2, 3, 4), pred_seq.permute(1, 0, 2, 3, 4)
        _, _, psnr = finn_eval_seq(seq[args.n_past:], pred_seq[args.n_past:])
        test_psnr.append(psnr)

avg_psnr = np.mean(test_psnr)
print("Test PSNR:", avg_psnr)
```

- pred()

```
def pred(seq, cond, modules, args, device, posterior=False):
    # reset h, c in lstm cell
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()

    seq = seq.permute(1, 0, 2, 3, 4)
    cond = cond.permute(1, 0, 2)

    pred_seq = [seq[0]]
    with torch.no_grad():
        x_t = seq[0]
        for i in range(1, args.n_past + args.n_future):
            if args.last_frame_skip or i <= args.n_past:
                h_t, skip = modules['encoder'](x_t)
            else:
                h_t, _ = modules['encoder'](x_t)

            cond_t = cond[i-1]

            if not posterior:
```

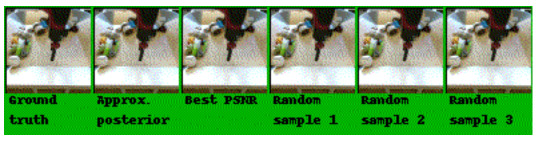
```
            z_t = torch.randn((args.batch_size, args.z_dim)).to(device)
        else:
            z_t, mu, sigma = modules['posterior'](h_t)

        info = torch.cat((cond_t, h_t, z_t), axis=1)
        g_t = modules['frame_predictor'](info)
        prediction = modules['decoder']([g_t, skip])

        if i < args.n_past:
            pred_seq.append(seq[i])
            x_t = seq[i]
        else:
            pred_seq.append(prediction)
            x_t = prediction

    pred_seq = torch.stack(pred_seq)
    return pred_seq.permute(1, 0, 2, 3, 4)
```

Hyperparameters

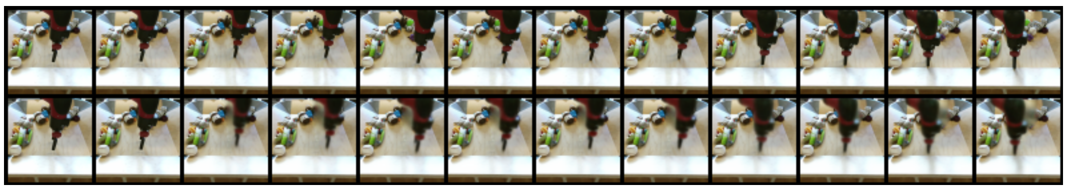
lr	batch_size	niter	epoch_size	tfr_start_decay
0.002	24	200	600	20

KL_Annealing (Monotonic)

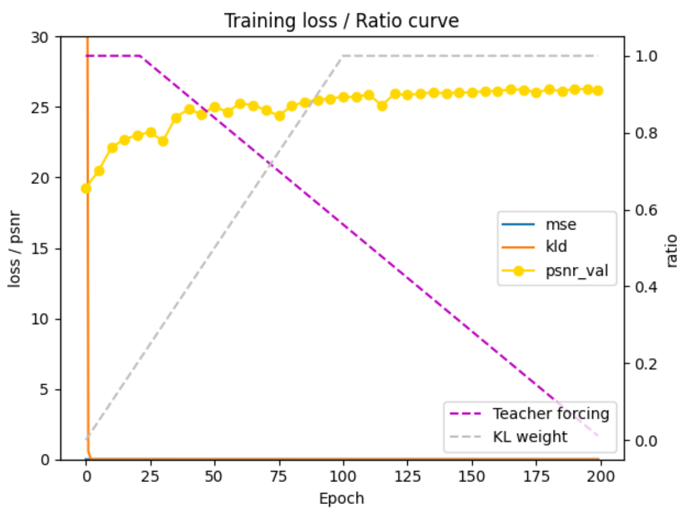
	Validation	Test
PSNR	26.31	25.63 (Best)



Ground Truth & Prediction

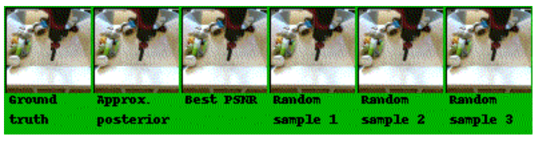


Training Curves

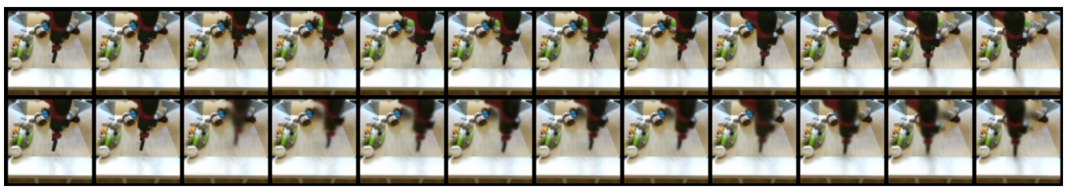


KL_Annealing (Cyclical, cycle=4)

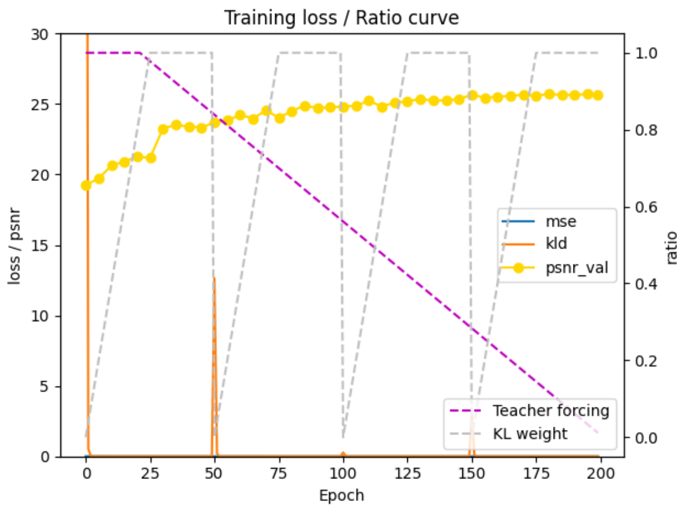
	Validation	Test
PSNR	25.72	25.21



Ground Truth & Prediction



Training Curves



Discuss the results according to your setting

上方呈現的是分別使用 Monotonic 與 Cyclical KL annealing 的實驗結果，兩者的訓練參數都相同，唯一不同的只有 KL weight 的調整方式。Monotonic 只有一次 cycle，而 cyclical 我設成 4 個 cycles。從實驗結果來看，monotonic 的表現比 cyclical 的好一點，但兩個訓練完之後都有過 baseline (psnr >= 25)。

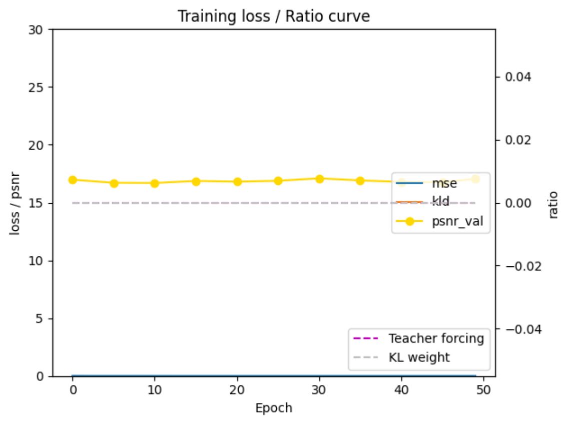
前面有提到大部分跟 RNN 時間序列有關的 task 都會採用 teacher forcing 的方式幫助前期模型的訓練，讓模型一開始不會因為不好的 output 而難以訓練。因此我這邊在前 20 個 epoch 採用完全的 teacher forcing，第 21 個 epoch 開始就會 linear 的調降 teacher forcing 的比例，一路到 0，讓模型能夠慢慢用自己的 output 去做訓練。

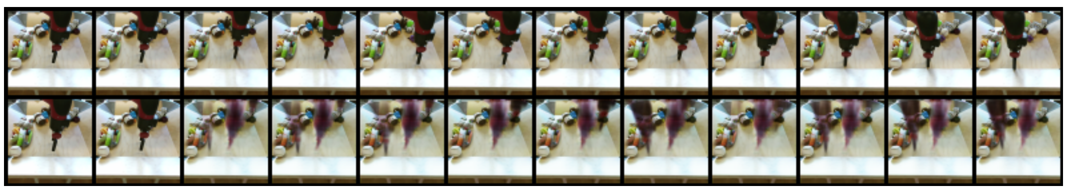
Other observations

Without teacher forcing & Without KL annealing

在這個部分，我測試了如果把 teacher forcing 跟 KL weight 都關掉，讓模型只透過 reconstruction loss 去學習會學得如何。

從實驗結果可以看到，基本上是訓練不起來的，psnr 即使過了 50 個 epoch 還是沒有辦法提高；而從生成的結果來看，模型也沒有辦法生成出正確的結果，甚至影像的基底還是都保持 last ground truth frame 所給的特徵。

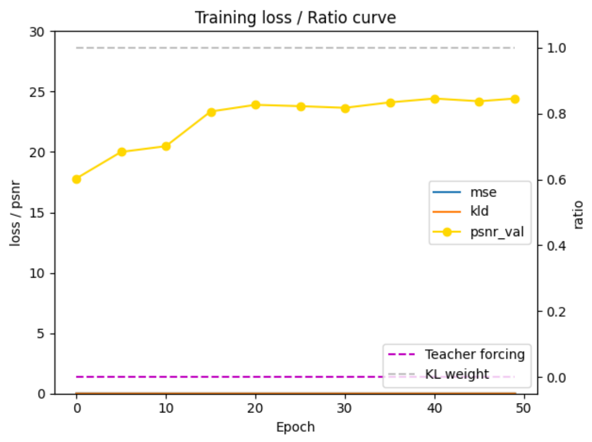


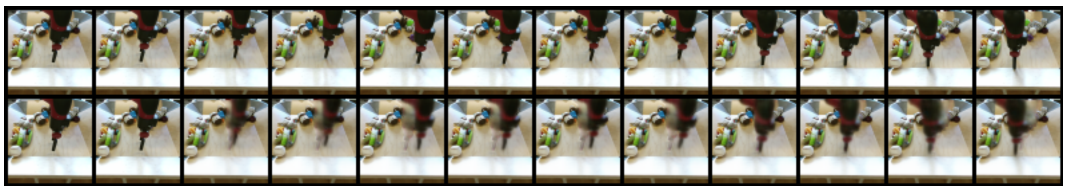


Without teacher forcing & Constant KL weight = 1.0

在這個部分，我加入了 KL weight term，讓模型能夠遵從 VAE 的定義，在訓練的時候也要去 minimize KL divergence，看看效果怎麼樣。

實驗結果有點意外，原本以為即使加入了 KL weight term，但沒有用 teacher forcing 會 train 不起來，但短短跑了 50 個 epoch，validation psnr 就有到 24.66，test psnr 也有到 24.04。雖然從生成的結果來看，還是有些last ground truth frame 的殘影在，但 end effector 的位置其實已經有接近正確答案了，也許再往下訓練一段時間效果會更好。





Skip Connections Between Encoder and Decoder

另外一個發現是，作者在文獻中有提到 “*For all datasets we add skip connections from the encoder at the last ground truth frame to the decoder at t, enabling the model to easliy generate static background features.*” 我原本在實作上並沒有注意到那個 skip 的用途，發現不管怎麼 train，psnr 都大約卡在 21、22 沒有辦法再往上。後來才有去注意到作者有特別說明，加了 skip connections 可以讓 decoder 在生成影像的時候，能夠有原本 encoder 中每一層的細節資訊當作參考，讓他在生成影像的效果上提高很多，也可以捕捉到很多細節。

Extra

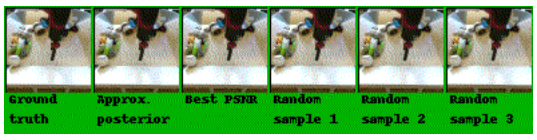
Implement learned prior

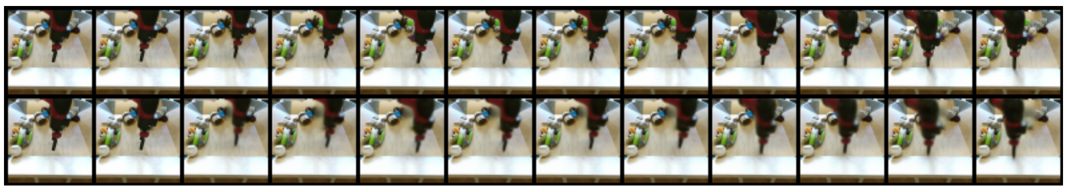
在實作中，把原本要從 normal distribution 中去 sample 換成從 learnable 的 prior distribution 中做 sample。

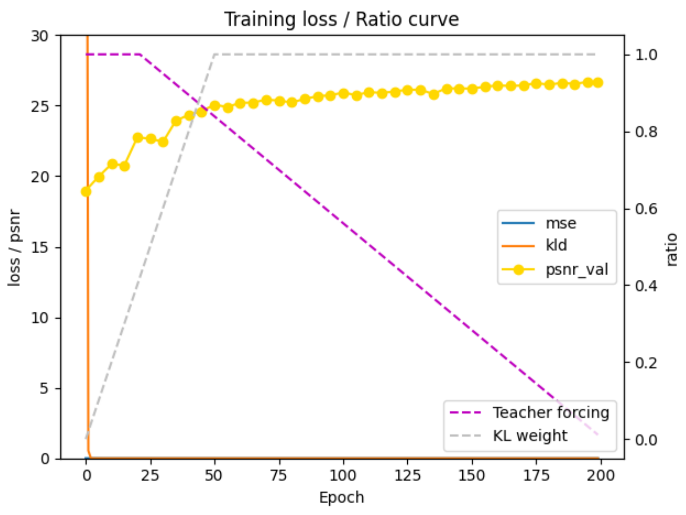
```
prior = gaussian_lstm(args.g_dim, args.z_dim, args.rnn_size, args.prior_rnn_layers, args.batch_size, device)

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

    x = x.permute(1, 0, 2, 3, 4)
    cond = cond.permute(1, 0, 2)

    use_teacher_forcing = True if random.random() < args.tfr else False
    x_t_1 = x[0]
    for i in range(1, args.n_past + args.n_future):
        x_t = x[i]
        h_t, _ = modules['encoder'](x_t)
        z_t, mu, sigma = modules['posterior'](h_t)

        if args.last_frame_skip or i <= args.n_past:
            h_t_1, skip = modules['encoder'](x_t_1)
        else:
            h_t_1, _ = modules['encoder'](x_t_1)

        z_prior, mu_prior, sigma_prior = modules['prior'](h_t_1)

        cond_t_1 = cond[i-1]

        info = torch.cat((cond_t_1, h_t_1, z_t), axis=1)
        g_t = modules['frame_predictor'](info)
        prediction = modules['decoder']([g_t, skip])

        if use_teacher_forcing:
            x_t_1 = x[i]
        else:
            x_t_1 = prediction

        mse += mse_loss(x_t, prediction)
        kld += kl_divergence(mu, sigma, mu_prior, sigma_prior, 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.det
```

Training Curves and Results

	Validation	Test
PSNR	26.68	25.99






Training loss / Ratio curve
loss / psnr
ratio
Epoch
mse
kld
psnr_val
Teacher forcing
KL weight
0
5
10
15
20
25
30
0.0
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200