

# 1. Introduction

In this lab, I would like to implement a **conditional VAE** for video prediction. The dataset we used contains several roughly 44000 sequences (each with 30 frames) of bair robot pushing motions, as well as the action and end-effector position for each time step.

Aside from conditional VAE, I also implement other techniques to help training, including **reparameterization trick**, **KL cost annealing** and **teacher forcing**. I will thoroughly describe each of them, including how I applied them in my code.

## 2. Derivation of Conditional VAE

The objective function of conditional VAE is formulated as:

$$L(x_t, c, q, \theta) = \mathbb{E}_{z_t \sim q(z_t|x_t, c; \phi)} \log p(x_t|x_{t-1}, z_t, c; \theta) - D_{KL}(q(z_t|x_t, c; \phi) || p(z|c))$$

Here,  $c$  is the conditions, namely *actions* and *end-effector positions*.  $q(z_t|x_t, c; \phi)$  denotes the **encoder** as well as the **posterior LSTM** $_{\phi}$ . We simply assume it to follow a *Gaussian* distribution  $N(\mu_{\phi(t)}, \sigma_{\phi(t)})$ . And then,  $p(x_t|x_{t-1}, z_t, c; \theta)$  represents the **decoder** and the **frame predictor LSTM** $_{\theta}$ . Finally, since we basically assume a **fixed prior** in this implementation,  $p(z|c)$  is set to be a **standard normal distribution**,  $N(0, 1)$ .

Since that the prior is the same for all timestamp  $t$ , and that it is independent of the input of last timestamp  $t - 1$ , we can rewrite  $p(z|c)$  as  $p(z_t|x_{t-1}, c; \theta)$ . And then, By the **chain rule of conditional probability** suggests that

$$p(x_t|x_{t-1}, z_t, c; \theta) = \frac{p(x_t, z_t|x_{t-1}, c; \theta)}{p(z_t|x_{t-1}, c; \theta)}$$

After taking the logarithms on both sides, we get

$$\log p(x_t|x_{t-1}, z_t, c; \theta) = \log p(x_t, z_t|x_{t-1}, c; \theta) - \log p(z_t|x_{t-1}, c; \theta)$$

And then, we introduce a distribution  $q(z_t|x_t, c; \theta)$  on both sides and integrate over  $z_t$ , we get

$$\begin{aligned} & \int q(z_t|x_t, c; \theta) \log p(x_t|x_{t-1}, z_t, c; \theta) dz_t \\ &= \int q(z_t|x_t, c; \theta) \log p(x_t, z_t|x_{t-1}, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log p(z_t|x_{t-1}, c; \theta) dz_t \\ &= \int q(z_t|x_t, c; \theta) \log p(x_t, z_t|x_{t-1}, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log p(z_t|x_{t-1}, c; \theta) dz_t \\ & \quad + \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t \\ &= \left( \int q(z_t|x_t, c; \theta) \log p(x_t, z_t|x_{t-1}, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t \right) \\ & \quad + \left( \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log p(z_t|x_{t-1}, c; \theta) dz_t \right) \end{aligned}$$

Thus, we get

$$\begin{aligned} & \int q(z_t|x_t, c; \theta) \log p(x_t, z_t|x_{t-1}, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t \\ &= \int q(z_t|x_t, c; \theta) \log p(x_t|x_{t-1}, z_t, c; \theta) dz_t \\ & \quad - \left( \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log p(z_t|x_{t-1}, c; \theta) dz_t \right) \end{aligned}$$

The left hand side is exactly the objective function, i.e. the lower bound we would like to maximize in the EM algorithm.

$$\begin{aligned} & \int q(z_t|x_t, c; \theta) \log p(x_t, z_t|x_{t-1}, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t \\ &= L(x_t, c, q, \theta) \end{aligned}$$

where the right hand side is composed of two terms:

The first term is the expectation of conditional log-likelihood w.r.t.  $z_t$  using the encoding distribution  $q(z_t|x_t, c; \theta)$ :

$$\begin{aligned} & \int q(z_t|x_t, c; \theta) \log p(x_t|x_{t-1}, z_t, c; \theta) dz_t \\ &= \mathbb{E}_{z_t \sim q(z_t|x_t, c; \theta)} \log p(x_t|x_{t-1}, z_t, c; \theta) \end{aligned}$$

The second term (without the negative sign in front) is the KL divergence between the encoding distribution and the prior.


$$\begin{aligned} & \int q(z_t|x_t, c; \theta) \log q(z_t|x_t, c; \theta) dz_t - \int q(z_t|x_t, c; \theta) \log p(z_t|x_{t-1}, c; \theta) dz_t \\ &= \int q(z_t|x_t, c; \theta) \log \frac{q(z_t|x_t, c; \theta)}{p(z_t|x_{t-1}, c; \theta)} dz_t = D_{KL}(q(z_t|x_t, c; \theta) || p(z_t|x_{t-1}, c; \theta)) \end{aligned}$$

Therefore, putting them together, we get:

$$\begin{aligned} L(x_t, c, q, \theta) &= \mathbb{E}_{z_t \sim q(z_t|x_t, c; \theta)} \log p(x_t|x_{t-1}, z_t, c; \theta) \\ &\quad - D_{KL}(q(z_t|x_t, c; \theta) || p(z_t|x_{t-1}, c; \theta)) \end{aligned}$$

### 3. Implementation Details

#### Dataset and Dataloader

 This part is given in the sample code and left unchanged.

Since the pattern is the same, I will use training data as an example.

```
1 train_data = bair_robot_pushing_dataset(args, 'train')
2 train_loader = DataLoader(train_data,
3                             num_workers=args.num_workers,
4                             batch_size=args.batch_size,
5                             shuffle=True,
6                             drop_last=True,
7                             pin_memory=True)
8 train_iterator = iter(train_loader)
```

Firstly, we use the pre-defined class

`bair_robot_pushing_dataset` to retrieve the data from the specific directory and gather them as a dataset. The `bair_robot_pushing_dataset` class is inherited from `torch.utils.data.Dataset` class. While the magic method

`__getitem__()` is called, the dataset class will return the **image tensor** as well as the **conditions** (action and end-effector position concatenated). Both of them are returned as tensors.

Then, we will call the `DataLoader` from `torch.utils.data` to wrap the dataset into a dataloader with mini-batches. The default batch size `args.batch_size` is **12** and the multi-process data loading `args.num_workers` contains **4** subprocesses by default. If run into any RAM or GPU memory problem, one can set smaller values for these 2. Finally, we will turn the dataloader into an iterator object for fetching mini-batches one-by-one.

## Encoder

---



This part is given in the sample code and left unchanged.

```
1 class vgg_layer(nn.Module):
2     def __init__(self, nin, nout):
3         super(vgg_layer, self).__init__()
4         self.main = nn.Sequential(
5             nn.Conv2d(nin, nout, 3, 1, 1),
6             nn.BatchNorm2d(nout),
7             nn.LeakyReLU(0.2, inplace=True)
8         )
9
10    def forward(self, input):
11        return self.main(input)
```

This is the structure of the **convolution blocks** used in **VGG net**. It consists of 3 layers:

1. A **convolution** layer with  $3 \times 3$  kernels and **same padding** (meaning that the output feature maps do not shrink).
2. A **batch normalization** layer
3. A **Leaky ReLU** activation function layer, with the slope for negative input being **0.2**.

```

1 class vgg_encoder(nn.Module):
2     def __init__(self, dim):
3         super(vgg_encoder, self).__init__()
4         self.dim = dim
5         # 64 x 64
6         self.c1 = nn.Sequential(
7             vgg_layer(3, 64),
8             vgg_layer(64, 64),
9         )
10        # 32 x 32
11        self.c2 = nn.Sequential(
12            vgg_layer(64, 128),
13            vgg_layer(128, 128),
14        )
15        # 16 x 16
16        self.c3 = nn.Sequential(
17            vgg_layer(128, 256),
18            vgg_layer(256, 256),
19            vgg_layer(256, 256),
20        )
21        # 8 x 8
22        self.c4 = nn.Sequential(
23            vgg_layer(256, 512),
24            vgg_layer(512, 512),
25            vgg_layer(512, 512),
26        )
27        # 4 x 4
28        self.c5 = nn.Sequential(
29            nn.Conv2d(512, dim, 4, 1, 0),
30            nn.BatchNorm2d(dim),
31            nn.Tanh()
32        )
33        self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
34
35    def forward(self, input):
36        h1 = self.c1(input) # 64 -> 32
37        h2 = self.c2(self.mp(h1)) # 32 -> 16
38        h3 = self.c3(self.mp(h2)) # 16 -> 8
39        h4 = self.c4(self.mp(h3)) # 8 -> 4
40        h5 = self.c5(self.mp(h4)) # 4 -> 1
41        return h5.view(-1, self.dim), [h1, h2, h3, h4]

```

The **encoder** is composed of **5 convolution blocks** and **4 max pooling layers**. Most of these convolution blocks comprise of the aforementioned **VGG convolution blocks**.

The input is a mini-batch, which is made up of **12** (batch size) sequences, each with **30** frames. Each frame has **3** channels (RGB) and the size is **64** by **64**. Since, we will process these video sequences **frame-by-frame**, the input shape should be **(12, 3, 64, 64)**, with single frames at some timestamp for these 12 videos.


1. After the first convolution block, the shape of data becomes **(12, 64, 64, 64)**, denoted **h1**.
2. And then a  $2 \times 2$  max pooling with stride 2, the shape becomes **(12, 64, 32, 32)**.
3. After the second convolution block, the shape of data becomes **(12, 128, 32, 32)**, denoted **h2**.

4. And then a  $2 \times 2$  max pooling with stride 2, the shape becomes (12, 128, 16, 16).
5. After the third convolution block, the shape of data becomes (12, 256, 16, 16), denoted  $h_3$ .
6. And then a  $2 \times 2$  max pooling with stride 2, the shape becomes (12, 256, 8, 8).
7. After the fourth convolution block, the shape of data becomes (12, 512, 8, 8), denoted  $h_4$ .
8. And then a  $2 \times 2$  max pooling with stride 2, the shape becomes (12, 512, 4, 4).
9. Finally, the final convolution block which has **128** (default value for the dimension of  $h_t$  and  $h_{t-1}$ )  $4 \times 4$  kernels and no padding makes the shape (12, 128, 1, 1). Before output, we discard the 1-dimension, reshaping it as (12, 128), the final output shape.

Except for the output feature maps (the (12, 128) tensors),  $h_1$  up to  $h_4$  will be returned as well. They will be used for the **decoder** as part of the inputs. This is called the **skip-connection** technique. The reason why we did this is to generate a **stable background** for each frames. If the argument `args.last_frame_skip` is set to **False** (default), the skip connection goes between the last ground truth frame, meaning that the background would be generated from the last ground truth frame given. Otherwise, the background would be connected from the previous frame.

## Decoder

---

 This part is given in the sample code and left unchanged.

Note that the structure of the decoder is somehow symmetrical to that of the encoder!

```

1  class vgg_decoder(nn.Module):
2      def __init__(self, dim):
3          super(vgg_decoder, self).__init__()
4          self.dim = dim
5          # 1 x 1 -> 4 x 4
6          self.upc1 = nn.Sequential(
7              nn.ConvTranspose2d(dim, 512, 4, 1, 0),
8              nn.BatchNorm2d(512),
9              nn.LeakyReLU(0.2, inplace=True)
10             )
11         # 8 x 8
12         self.upc2 = nn.Sequential(
13             vgg_layer(512*2, 512),
14             vgg_layer(512, 512),
15             vgg_layer(512, 256)
16         )
17         # 16 x 16
18         self.upc3 = nn.Sequential(
19             vgg_layer(256*2, 256),
20             vgg_layer(256, 256),
21             vgg_layer(256, 128)
22         )
23         # 32 x 32
24         self.upc4 = nn.Sequential(
25             vgg_layer(128*2, 128),
26             vgg_layer(128, 64)
27         )
28         # 64 x 64
29         self.upc5 = nn.Sequential(
30             vgg_layer(64*2, 64),
31             nn.ConvTranspose2d(64, 3, 3, 1, 1),
32             nn.Sigmoid()
33         )
34         self.up = nn.UpsamplingNearest2d(scale_factor=2)
35
36     def forward(self, input):
37         vec, skip = input
38         d1 = self.upc1(vec.view(-1, self.dim, 1, 1)) # 1 -> 4
39         up1 = self.up(d1) # 4 -> 8
40         d2 = self.upc2(torch.cat([up1, skip[3]], 1)) # 8 x 8
41         up2 = self.up(d2) # 8 -> 16
42         d3 = self.upc3(torch.cat([up2, skip[2]], 1)) # 16 x 16
43         up3 = self.up(d3) # 8 -> 32
44         d4 = self.upc4(torch.cat([up3, skip[1]], 1)) # 32 x 32
45         up4 = self.up(d4) # 32 -> 64
46         output = self.upc5(torch.cat([up4, skip[0]], 1)) # 64 x 64
47         return output

```

The **decoder** is composed of **5 deconvolution blocks** and **4 upsampling layers**. Most of these deconvolution blocks comprise of the aforementioned **VGG convolution blocks**.

The input is a tuple. The first term is tensors having shape of (12, 128), where **12** is the default batch size and **128** is the default dimension for the input of decoder,  $g_t$ . The second term is a list containing 4 tensors  $h_1$  to  $h_4$ , which are the **skip-connections** from the encoder. They will be concatenated with the inputs of different deconvolution layers later on. Before we plunge the data into the first deconvolution block, we will first reshape them as (12, 128, 1, 1), just like 128 feature maps with  $1 \times 1$  size.

1. After the first deconvolution block with **512**  $4 \times 4$  kernels and no padding, the shape of data becomes (12, 512, 4, 4).
2. And then a *nearest neighbor* upsampling layer with the scale factor of 2, the shape becomes (12, 512, 8, 8).
3. The data are then concatenated with  $h_4$  of size (12, 512, 8, 8) at dimension 1, making the new shape (12, 1024, 8, 8).
4. After the second deconvolution block, the shape of data becomes (12, 256, 8, 8).
5. And then a *nearest neighbor* upsampling layer with the scale factor of 2, the shape becomes (12, 256, 16, 16).
6. The data are then concatenated with  $h_3$  of size (12, 256, 16, 16) at dimension 1, making the new shape (12, 512, 16, 16).
7. After the third deconvolution block, the shape of data becomes (12, 128, 16, 16).
8. And then a *nearest neighbor* upsampling layer with the scale factor of 2, the shape becomes (12, 128, 32, 32).
9. The data are then concatenated with  $h_2$  of size (12, 128, 32, 32) at dimension 1, making the new shape (12, 256, 32, 32).
10. After the fourth deconvolution block, the shape of data becomes (12, 64, 32, 32).
11. And then a *nearest neighbor* upsampling layer with the scale factor of 2, the shape becomes (12, 64, 64, 64).
12. The data are then concatenated with  $h_1$  of size (12, 64, 64, 64) at dimension 1, making the new shape (12, 128, 64, 64).
13. Finally, the data go through the final deconvolution block with **3**  $3 \times 3$  kernels with the **same padding**, the size of output data become (12, 3, 64, 64), which is just the **same shape as the input of the encoder!**



The **frame predictor** is a **2-layer LSTM** model. The input size is a  $(12, 199)$ -sized tensor, where **12** is the batch size. As for **199**, it's coming from the 3 different tensors forming the input of frame predictor:

1. the **condition**  $c_t$  of the next timestamp  $t$  with shape  $(12, 7)$ ,
2. the **encoded frame** for current timestamp ( $x_{t-1}$  or  $\hat{x}_{t-1}$  depending on the on or off of teacher forcing) from the encoder with size  $(12, 128)$ ,
3. the **latent vector**  $z_t$  sampled from the normal distribution whose parameters are determined by the **posterior LSTM**. The latent vector contains the information of the encoded frame at the next timestamp  $t$ ,  $x_t$  and has the size  $(12, 64)$ .

These 3 different tensors are then concatenated over dim 1, forming a  $(12, 7 + 128 + 64) = (12, 199)$  tensor as the input of the frame predictor  $LSTM_\theta$ .

The default dimension of the **hidden states** is **256**, and **128** for the output  $g_t$ . Thus, the output  $g_t$  has the shape of  $(12, 128)$ .

```

1 class lstm(nn.Module):
2     def __init__(self, input_size, output_size, hidden_size, n_layers,
3         batch_size, device):
4         super(lstm, self).__init__()
5         self.device = device
6         self.input_size = input_size
7         self.output_size = output_size
8         self.hidden_size = hidden_size
9         self.batch_size = batch_size
10        self.n_layers = n_layers
11        self.embed = nn.Linear(input_size, hidden_size)
12        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size)
13            for i in range(self.n_layers)])
14        self.output = nn.Sequential(
15            nn.Linear(hidden_size, output_size),
16            nn.BatchNorm1d(output_size),
17            nn.Tanh())
18        self.hidden = self.init_hidden()
19
20    def init_hidden(self):
21        hidden = []
22        for _ in range(self.n_layers):
23            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size)
24                .to(self.device)),
25                Variable(torch.zeros(self.batch_size, self.hidden_size)
26                    .to(self.device))))
27        return hidden
28
29    def forward(self, input):
30        embedded = self.embed(input)
31        h_in = embedded
32        for i in range(self.n_layers):
33            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
34            h_in = self.hidden[i][0]
35
36        return self.output(h_in)

```

The hidden states of both the LSTM layers are first initialized with **0s** by `self.init_hidden()`.

The input shape of this frame predictor is (12, 199), but since the dimension of hidden states is **256**, the data will first pass through a **linear** layer as a role of *embedding layer*.

After that, the (12, 256) embeddings will be fed into the **2-layer LSTM**.

Finally, the hidden tensors will go through a linear layer to reshape into the size of (12, 64), the output dimension. After a batch normalization layer and a hyperbolic tangent activation layer, we then get the final output tensor  $g_t$ .

## Reparameterization Trick

The reason why we should implement the so-called *reparameterization trick* is due to the feasibility of **end-to-end training**. Instead of directly generating latent variables  $\vec{z}$  as what tradition autoencoders do, variational

autoencoders generate the **probabilistic distribution** of  $\vec{z}$ , and then sample a  $\vec{z}$  from it. (By generating the distribution, I mean generating the *parameters' value*.) Also, the loss function of VAE also considers this generated distribution of  $\vec{z}$  into account, in the form of *KL divergence*.

However, if we use this kind of **sampling** method, **we cannot take derivatives**, and therefore unable to conduct backpropagation. Since our prior is assumed to be standard Gaussian distribution and that our posterior distribution of  $\vec{z}$  is Normal distribution as well, we can successfully realize end-to-end training in virtue of *reparameterization* as so:

Set an arbitrary variable  $\vec{\epsilon} \sim N(\vec{0}, I)$  and the distribution of  $z$  being  $N(\vec{\mu}_t(\vec{x}_t), \Sigma_t(\vec{x}_t))$ , where  $\vec{x}_t$  is the current ground truth frame. Then by the **affine transformation** property of normal distribution:

$$\vec{z}_t = \vec{\mu}_t(\vec{x}_t) + (\Sigma_t(\vec{x}_t))^{\frac{1}{2}} \vec{\epsilon} \sim N(\vec{\mu}_t(\vec{x}_t), \Sigma_t(\vec{x}_t))$$

We can get  $\vec{z}_t$  from the  $N(\vec{\mu}_t(\vec{x}_t), \Sigma_t(\vec{x}_t))$  as what sampling does without any worries regarding unavailability of BP.

Please note that the parameters' value of the distribution of  $\vec{z}$  generated is the **mean** and the **log variance** of Gaussian distribution instead of variance directly. This is because that the variance is constrained to be **non-negative**. So if we were to try to learn the variance, we would have to constrain somehow the output of the neural network to be non-negative, which is impractical. Taking exponential of log variance can still guarantee us a non-negative variance.

$$e^{\log \sigma^2} = \sigma^2 \in \{\mathbb{R}^+, 0\}$$

The following is my code implementation

```

1 class gaussian_lstm(nn.Module):
2     def __init__(self, input_size, output_size, hidden_size, .....):
3         .....
4
5     def init_hidden(self):
6         .....
7
8     def reparameterize(self, mu, logvar):
9         logvar = logvar.mul(0.5).exp_()
10        eps = Variable(logvar.data.new(logvar.size()).normal_())
11        return eps.mul(logvar).add_(mu)
12
13    def forward(self, input):
14        .....

```

The reparameterization trick is implemented in the class `gaussian_lstm`, which is the network structure for **posterior distribution of  $\vec{z}_t$** . First, we multiply the log variance by  $\frac{1}{2}$  and take exponential to get the **standard deviation  $\Sigma^{\frac{1}{2}}$** .

$$e^{\frac{1}{2}(\log \Sigma)} = e^{\log(\Sigma^{\frac{1}{2}})} = \Sigma^{\frac{1}{2}}$$

And then, generate a variable that follows multivariate standard normal distribution.

$$\vec{\epsilon} \sim N(\vec{0}, I)$$

Finally, get  $\vec{z}_t$  as

$$\vec{z}_t = (\Sigma^{\frac{1}{2}})\vec{\epsilon} + \vec{\mu}$$

## Posterior LSTM <sub>$\phi$</sub>

The **posterior** is a **single layer LSTM** model. The input size is a (12, 128)-sized tensor, where **12** is the batch size and **128** is the dimension of the encoding embedding  $h_t$  of the next frame  $x_t$  to be predicted.

The default dimension of the **hidden states** is **256**, and **64** for the output latent tensor  $z_t$ . Thus, the output  $z_t$  has the shape of (12, 64).

```

1 class gaussian_lstm(nn.Module):
2     def __init__(self, input_size, output_size, hidden_size, n_layers,
3         batch_size, device):
4         super(gaussian_lstm, self).__init__()
5         self.device = device
6         self.input_size = input_size
7         self.output_size = output_size
8         self.hidden_size = hidden_size
9         self.n_layers = n_layers
10        self.batch_size = batch_size
11        self.embed = nn.Linear(input_size, hidden_size)
12        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size)
13            for _ in range(self.n_layers)])
14        self.mu_net = nn.Linear(hidden_size, output_size)
15        self.logvar_net = nn.Linear(hidden_size, output_size)
16        self.hidden = self.init_hidden()
17
18    def init_hidden(self):
19        hidden = []
20        for _ in range(self.n_layers):
21            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size)
22                .to(self.device)),
23                Variable(torch.zeros(self.batch_size, self.hidden_size)
24                    .to(self.device))))
25        return hidden
26
27    def reparameterize(self, mu, logvar):
28        logvar = logvar.mul(0.5).exp_()
29        eps = Variable(logvar.data.new(logvar.size()).normal_())
30        return eps.mul(logvar).add_(mu)
31
32    def forward(self, input):
33        embedded = self.embed(input)
34        h_in = embedded
35        for i in range(self.n_layers):
36            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
37            h_in = self.hidden[i][0]
38        mu = self.mu_net(h_in)
39        logvar = self.logvar_net(h_in)
40        z = self.reparameterize(mu, logvar)
41        return z, mu, logvar

```

The hidden states of the LSTM layer is first initialized with **0s** by `self.init_hidden()` .

The input shape of this frame predictor is (12, 128), but since the dimension of hidden states is **256**, the data will first pass through a **linear** layer as a role of *embedding layer*.

After that, the (12, 256) embeddings will be fed into the **single layer LSTM**.

Then, the hidden tensors will be passed to 2 different neural networks, one for predicting the **mean** of the **posterior Gaussian distribution**, and the other for the **log variance** of the **posterior Gaussian distribution**.

These 2 networks are simply **linear** layers, with input size (12, 256) and output size (12, 64).

Note that as mentioned in the previous

**Reparameterization trick** part, we use *log* variance instead of pure variance directly. This is due to the non-

negative property of the variance. Since we don't want to constraint the model additionally (which might cause some trouble), we use the log variance which can be any real number.

Originally, the latent vector  $z_t$  needs to be **sampled**, but we utilize the aforementioned **reparameterization trick** to help get  $z_t$  with the capability of **end-to-end training** preserved.

The implementation details of reparameterization have already been elaborated above, so I will just skip them here.

Finally, the latent vector  $z_t$  along with the **mean**  $\mu_\phi(t)$  and the **log-variance**  $(\log \sigma^2)_\phi(t)$  will be output. All 3 of them have the shape of (12, 64). The latent vector  $z_t$  will be concatenated with the last encoded frame  $h_{t-1}$  and the conditions  $c_t$  as the input of **frame predictor**. As for  $\mu_\phi(t)$  and  $(\log \sigma^2)_\phi(t)$ ,

They will be used to compute the **KL divergence** between the posterior normal distribution and the prior which is just a **standard Normal distribution** in this case (*fixed prior*). KL divergence also contributes to the loss.

## KL Cost Annealing

---

In this implementation, we also added a variable weight  $\beta$  to the KL divergence term in the loss function as:

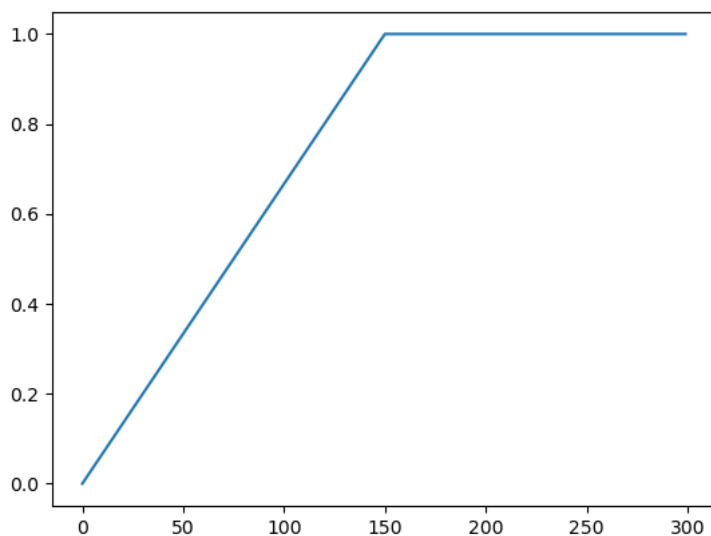
$$Loss = MSE + \beta \times KLD$$

to leverage the tradeoff between minimizing frame prediction error and distribution fitting error. The mean squared error (MSE) loss comes from the reconstruction, whereas the KL divergence (KLD) loss comes from the difference between the prior and the learned posterior.

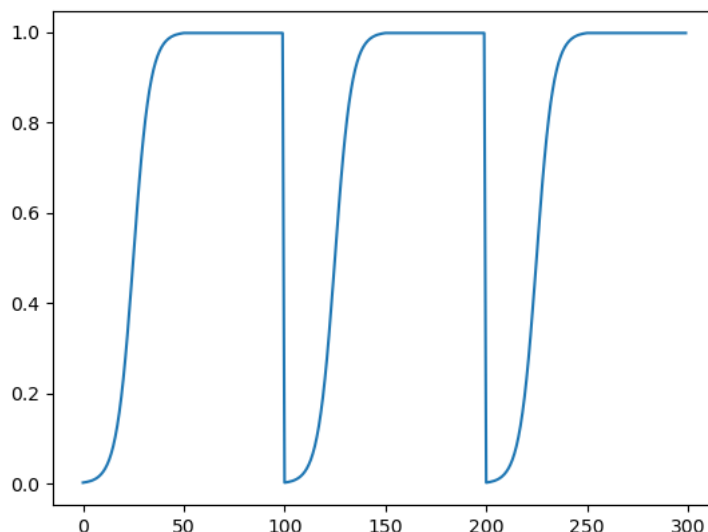
```
1  beta = kl_anneal.get_beta()
2  loss = mse + kld * beta
3  loss.backward()
```

The KL annealing technique is used to prevent **posterior collapse problem**, where the posterior simply degenerates to standard Gaussian as the prior to achieve minimum KL divergence (0), causing the KL divergence to vanish. Here, we applied 2 different annealing patterns: **monotonic** and **cyclical**. Both starts from 0 at the initial, so that the encoder can spend some time learning to encode the information of  $\vec{x}_t$  to  $\vec{z}_t$ . Later, as the training proceeds, the  $\beta$  increases to 1, constraining the posterior close to standard normal.

The monotonic one looks like so:



And the cyclical one looks like so: we use **sigmoid** function for the curvy part.



Following is my code implementation of KL annealing class

```

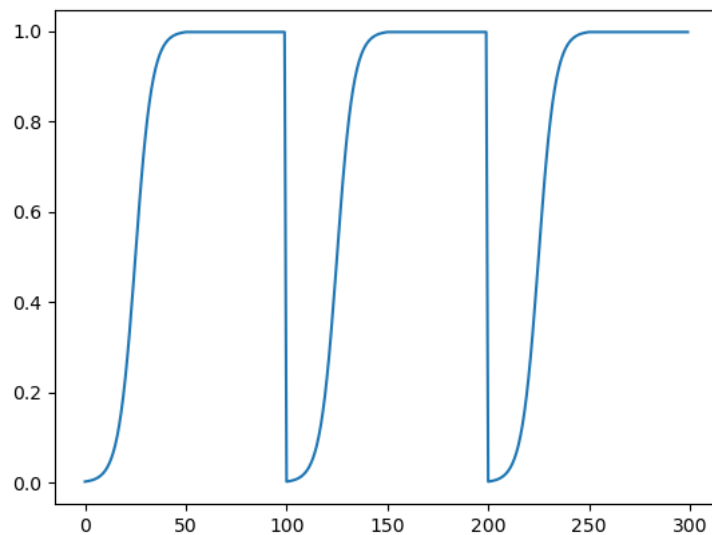
1 .....
2 parser.add_argument('--kl_anneal_cyclical', default=True, help='use cyclical mode'
3 parser.add_argument('--kl_anneal_ratio', type=float, default=0.5,
4 help='The decay ratio of kl annealing')
5 parser.add_argument('--kl_anneal_cycle', type=int, default=3,
6 help='The number of cycle for kl annealing during training (if use cyclical mode)'
7 .....
8 class kl_annealing:
9     def __init__(self, args):
10         self.epochs = args["niter"]
11         self.epoch_size = args["epoch_size"]
12         self.mode = args["kl_anneal_cyclical"]
13         self.ratio = args["kl_anneal_ratio"]
14         self.cycles = args["kl_anneal_cycle"] if self.mode else 1
15         self.period = self.epochs / self.cycles
16         self.step = 1.0 / (self.period * self.ratio) # within [0, 1]
17         self.v = 0.0
18         self.i = 0
19         self.t = 0
20
21     def update(self):
22         self.v += self.step
23         self.i += 1
24         if self.i == self.period:
25             self.v = 0.0
26             self.i = 0
27         if self.v > 1.0:
28             self.v = 1.0
29
30     def get_beta(self):
31         beta = 1.0 / (1.0 + np.exp(-(self.v * 12.0 - 6.0))) if self.mode
32         else self.v
33         self.t += 1
34         if self.t == self.epoch_size:
35             self.t = 0
36             self.update()
37         return beta

```

There are 3 arguments controlling the KL annealing patterns:

1. `kl_anneal_cyclical` : a boolean argument, indicating the program is using the **cyclical** annealing pattern or not. If false, use the **monotonic** pattern.
2. `kl_anneal_ratio` : the decay ratio of KL annealing. This argument controls how fast the  $\beta$  will grow from 0 to 1. Default value is **0.5**, meaning that in each cycle, the increase of  $\beta$  takes **half** the cycle, and then  $\beta = 1$  for the rest.
3. `kl_anneal_cycle` : totally how many annealing cycles to use. If **monotonic** pattern is chosen, this argument will forced to be **1**. In the example figure of cyclical pattern above as well as in my code implementation, the argument was set to be **3**.





`self.period` is the length of each cycle. For example in the cyclical figure above, the value is  $300 \div 3 = 100$ .

`self.step` is the increasing step and `self.v` is the current  $\beta$  value in **monotonic** mode. In cyclical mode, the increasing step as well as the current  $\beta$  value is fully controlled by the sigmoid function. `self.t` is a counter, since we don't need to update  $\beta$  value in different iterations within the same training epoch. We will wait until the next epoch, and then call `self.update()` to update the  $\beta$  value. Finally, `self.i` is another counter for controlling the  $\beta$  value in each cycle period.

As for teacher forcing, we will leave it to the next section.

## Training function

---

```

1 def train(x, cond, modules, optimizer, kl_anneal, args):
2     # x.shape = (30, 12, 3, 64, 64), cond.shape = (30, 12, 7)
3     modules['frame_predictor'].zero_grad()
4     modules['posterior'].zero_grad()
5     modules['encoder'].zero_grad()
6     modules['decoder'].zero_grad()
7
8     # initialize the hidden state.
9     modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
10    modules['posterior'].hidden = modules['posterior'].init_hidden()
11    mse = 0
12    kld = 0
13    use_teacher_forcing = True if random.random() < args.tfr else False
14    cond_seq = [cond[i] for i in range(args.n_past + args.n_future)]
15    x_pred = None
16
17    for i in range(1, args.n_past + args.n_future):
18        if use_teacher_forcing:
19            h_target = modules['encoder'](x[i])[0]
20            if args.last_frame_skip or i < args.n_past:
21                h_pred, skip = modules['encoder'](x[i-1])
22            else:
23                h_pred = modules['encoder'](x[i-1])[0]
24            z_t, mu, logvar = modules['posterior'](h_target)
25            h = modules['frame_predictor'](torch.cat([cond_seq[i], h_pred, z_t], 1)
26            x_pred = modules['decoder']([h, skip])
27        else:
28            h_target = modules['encoder'](x[i])[0]
29            if x_pred is None:
30                x_pred = x[0]
31            if args.last_frame_skip or i < args.n_past:
32                h_pred, skip = modules['encoder'](x_pred)
33            else:
34                h_pred = modules['encoder'](x_pred)[0]
35            z_t, mu, logvar = modules['posterior'](h_target)
36            h = modules['frame_predictor'](torch.cat([cond_seq[i], h_pred, z_t], 1)
37            x_pred = modules['decoder']([h, skip])
38
39        mse += mse_criterion(x_pred, x[i])
40        kld += kl_criterion(mu, logvar, args)
41
42    beta = kl_anneal.get_beta()
43    loss = mse + kld * beta
44    loss.backward()
45
46    optimizer.step()
47
48    return loss.detach().cpu().numpy() / (args.n_past + args.n_future),
49    mse.detach().cpu().numpy() / (args.n_past + args.n_future),
50    kld.detach().cpu().numpy() / (args.n_future + args.n_past), beta

```

The original shapes of a mini-batch sequences and conditions are (12, 30, 3, 64, 64) and (12, 30, 7), respectively, where **12** is the batch size and **30** is the sequence length. For the convenience with regard to extract the sequence ***frame-by-frame***, we will transpose the first 2 dimensions before feeding the data into this function. Thus, the sequences `x` have the shape (30, 12, 3, 64, 64) whereas the conditions `cond` have the shape (30, 12, 7).

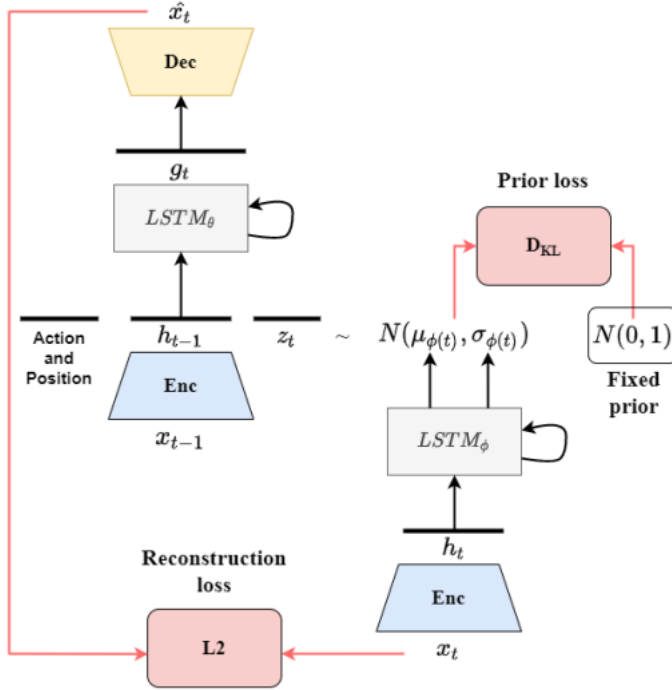
There are 4 modules in the experimental setting, **encoder**, **decoder**, **frame predictor** and **posterior**. We will clear their gradients first before each training iteration. Also,

since **frame predictor** and **posterior** are **LSTM**-based, we need to initialize their hidden states by calling

```
self.init_hidden() .
```

Here, we use **schedule sampling** on **teacher forcing**. In short, whether we apply teacher forcing or not depends on a random probability. As the teacher forcing ratio gets smaller and smaller along the training, it is unlikely to activate teacher forcing in later epochs.

Here, the procedures of predicting each frame look like so



where  $LSTM_\theta$  denotes the **frame predictor** and  $LSTM_\phi$  denotes the **posterior** modules here.

1. **With the teacher forcing mode**, in timestamp  $t - 1$ , we will take the ground truth frame  $x_{t-1}$  as the input of the **encoder**, to get the output  $h_{t-1}$ , denoted `h_pred` in my code above. As for the information of the next frame  $x_t$  to be predicted, ground truth  $x_t$  will also be passed through the **encoder** to get  $h_t$  (denoted `h_target`), and then  $h_t$  will go through the **posterior** modules to get 2 parameters value of  $z_t$ 's distribution (assumed to be Gaussian), namely the **mean**  $\mu_{\phi(t)}$  and **log variance**  $\log \sigma_{\phi(t)}^2$ . After the **reparameterization**,  $z_t$  is generated from the distribution.

In the next step, we will take the encoded current ground truth frame  $h_{t-1}$ , the  $z_t$  which somehow contains the information of the next ground truth frame  $x_t$ , as well as

the condition  $c$  which consists of **action** and the **end-effector position** of the bair robot at time  $t$ . Concatenate them together, and feed them into the **frame predictor** LSTM. The resulted output  $g_t$ , denoted `h` in my code, is then fed to the **decoder** as the input. Aside from  $g_t$ , the decoder also takes another thing called `skip` which is the **skip connection** taken from the encoder. This helps to generate more stable background from the last ground truth frame.

Finally, the output of decoder `x_pred` is the predicted frame at time stamp  $t$ ,  $\hat{x}_t$ . The loss is composed of 2 terms: the reconstruction error between  $x_t$  and  $\hat{x}_t$ , and the fitting error between the posterior distribution and the prior distribution. The trade-off is controlled via KL annealing coefficient  $\beta$ , denoted `beta` in my code above.

2. If **without** using teacher forcing, then the **previous predicted frame**  $\hat{x}_{t-1}$  instead of the ground truth frame  $x_{t-1}$  will be used for the input of the encoder. The rest is the same. Note that initially we don't have any predicted frame at the start, so the ground truth frame will be used.

```
1 | if x_pred is None:
2 |     x_pred = x[0]
```

## Teacher Forcing

---

The idea of teacher forcing, is to guide the training into the correct path. Originally, the decoder decodes the image encoded from the **previous predicted output**, which is code **free-running**. However, if any error appear in predicted frames, it will propagate and accumulate, making the training fail horribly in later epochs. This is very likely to happen due to low capability of the model at the start.

**Teacher forcing** technique suggests that we use **ground truth** instead of the prediction decoded from the previous timestamp as the input of the encoder. The benefit for

teacher forcing is that it can guide the models into the correct path timely via introducing the ground truth frame, and therefore **speed up the convergence**.

Even with such a benefit, teacher forcing still comes with a drawback, however. One shortcome is that during the **inference** session, we don't have the ground truth frames, and thus cannot acquire the aid of "teachers". The model then becomes fragile, especially on those cross-domain testing datasets. One way to solve this is **curriculum learning**, with the corresponding sampling technique called **scheduled sampling**.

In **curriculum learning**, every time we predict a sequence, we use a **probability**  $p$  to control whether we're applying the teacher forcing or not. If this probability  $p$  would change during the whole training process, then the process is called **scheduled sampling**.  $p$  is sometimes called **teacher forcing ratio**. At the beginning of training, the model is still weak and cannot make proper predictions, hence  $p$  is set to be large, so that we can help it by teacher forcing more. As the training going on, the model becomes better and better. So, we will gradually reduce the interference from teacher forcing, and let the model make predictions itself instead.

```
1 def train(x, cond, modules, optimizer, kl_anneal, args):  
2     .....  
3     use_teacher_forcing = True if random.random() < args.tfr else False  
4     .....
```

In my code, I introduced the **curriculum** learning teacher forcing in my training function as shown above. The teacher forcing is turned on only when the random number is smaller than the current teacher forcing ratio in `args.tfr`.

This teacher forcing ratio will get smaller towards 0 during the training. This is the implementation of **scheduled sampling** I just mentioned.

```

1  args.tfr_decay_step = 1.0 / (args.niter - args.tfr_start_decay_epoch)
2  for epoch in range(start_epoch, niter):
3      .....
4      for i in range(args.epoch_size):
5          loss, mse, kld, beta = train(seq, cond, modules, optimizer, kl_anneal, arg
6          .....
7          if epoch >= args.tfr_start_decay_epoch:
8              args.tfr -= args.tfr_decay_step
9              if args.tfr < args.tfr_lower_bound:
10                 args.tfr = args.tfr_lower_bound
11         .....

```

The `args.tfr_decay_step` indicates the step size in each epoch when the teacher forcing ratio start to *linearly* decay from 1 to 0. `args.tfr_start_decay_epoch` literally means the epoch where the teacher forcing ratio starts to decay. The lower bound of teacher forcing ratio, `args.tfr_lower_bound` is set to be **0**. If one wants to turn off teacher forcing manually, just set `args.tfr = 0` at the start.

## 4. Results and Discussion

### a. The GIF Image for One Test Sequence



Since GIF image is unlikely to function properly in pdf file, please click this link to redirect to my HackMD page for this lab.

**<https://hackmd.io/@witty27818a/B1N5bq5aq>**

(<https://hackmd.io/@witty27818a/B1N5bq5aq>).

Identical experimental settings:

1. **Teacher forcing ratio** starts to decay at around epoch **10**, from 1 to 0.
2. **Learning rate** is left as default: **0.002**

**Left:** Ground Truth Sequence, **Right:** Predicted Sequence

★ ***KL annealing mode = monotonic*** (default)



★ ***KL annealing mode = cyclical***



Both KL annealing modes generate quite good results. Although there are some blurs around the bair robots, the moving trajectories of predictions are basically the same as those of ground truth sequences.

I also noted a little problem here: in some ground truth sequence, the robot will push some objects and thus make them move away from their original places. However, in the video predictions, these objects do not move. When the robot touches them, they become **blurry** instead of being pushed away.

I think this might be contributed from the **last\_frame\_skip** argument set to **False** in default settings. The **skip connection** that helps generate good and static background connects every predicted frames with the **background** extracted from the **last given ground truth frame** by default, which might explain this phenomenon. However, due to the time-consuming training and approaching deadline, I didn't have time for validating this hypothesis. I would like to try out in the future.

## b. The Prediction at Each Time Step for One Test Sequence

**Up:** Ground Truth Sequence, **Down:** Predictied Sequence

★ **KL annealing mode = monotonic** (default)



★ **KL annealing mode = cyclical**



## The Loss and PSNR Curves During Training

Note: The **validate** PSNR scores used in the training and these figures here were computed over **all 28 predicted frames**, thus might be a bit lower than those in testing session. In **demo** (testing session), the PSNR scores are averaged only on **the first 10 predicted frames**.

## The Average Test PSNR

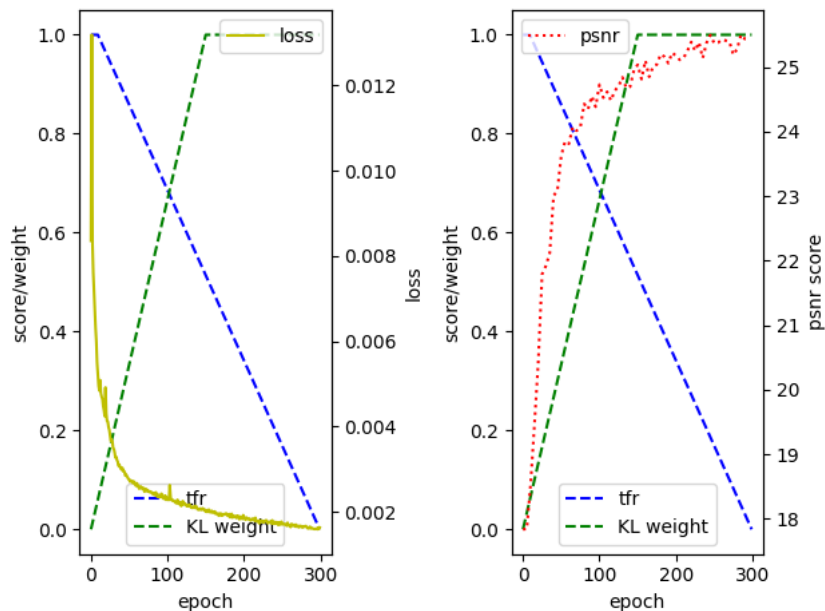
### 1. monotonic annealing: 26.78

```
(base) C:\Users\GL
26.78474476826316
```

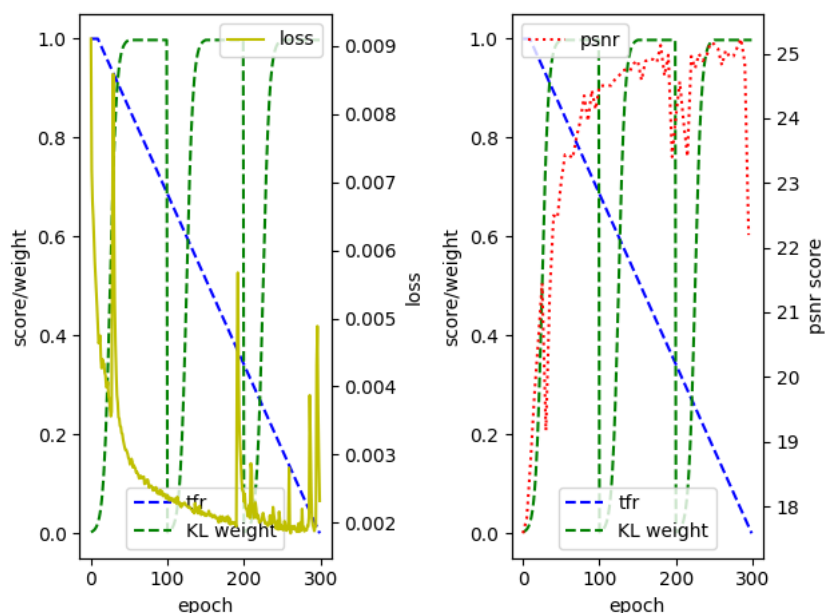
### 2. cyclical annealing: 26.34

```
(base) C:\Users\GL75\OneDrive\桌面\深度学习\
26.343610771007153
```

★ *KL annealing mode = monotonic* (default)



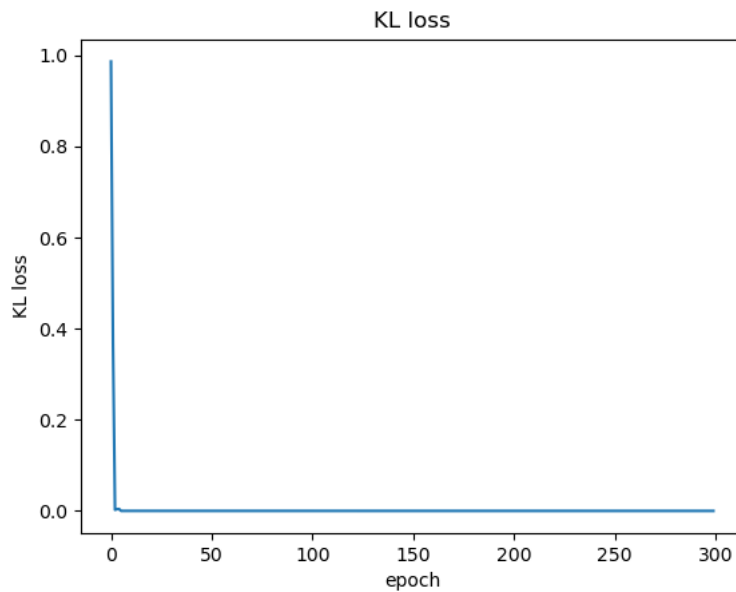
★ *KL annealing mode = cyclical*



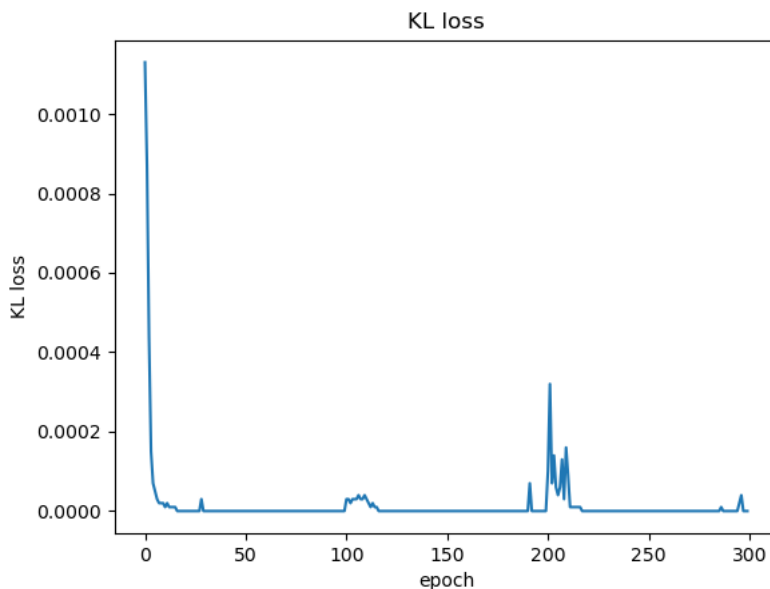
The loss curves above are total losses. For the KL Losses:



★ ***KL annealing mode = monotonic*** (default)



★ ***KL annealing mode = cyclical***



We use **3** cycles for the cyclical annealing schedule, meaning that every  $\frac{300}{3} = 100$  epochs form a period. At the start of each period, the KL weight  $\beta$  will be reinitialized to **0**. Indeed, you can see that besides epoch **0** (the start), at around epoch **100** and **200**, there are some **spikes** appeared in the KL loss curve.

## Discussion

---

For different ***KL annealing modes***

From the total loss curves and the PSNR curves of these 2 modes above, one can see that with **monotonic** annealing, the growth of PSNR scores and the decay of total losses are **much more stable**. Although at the end, both of them reach pretty high PSNR scores. The best *validate* PSNR score is around **25.23** for **cyclical annealing**

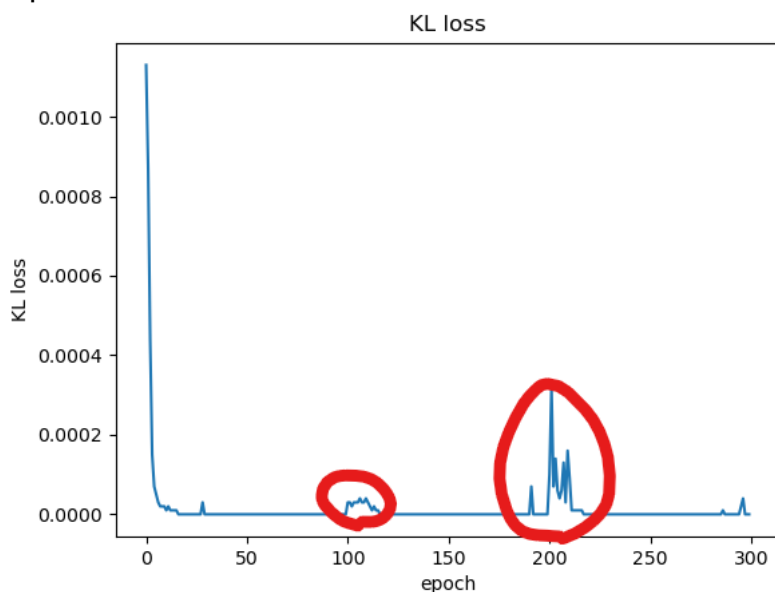
```
[epoch: 285] loss: 0.00209 | mse loss: 0.00209 | kld loss: 0.00000  
===== validate psnr = 25.22903 =====
```

and around **25.50** for **monotonic annealing**.

```
[epoch: 285] loss: 0.00161 | mse loss: 0.00161 | kld loss: 0.00000  
===== validate psnr = 25.50182 =====
```

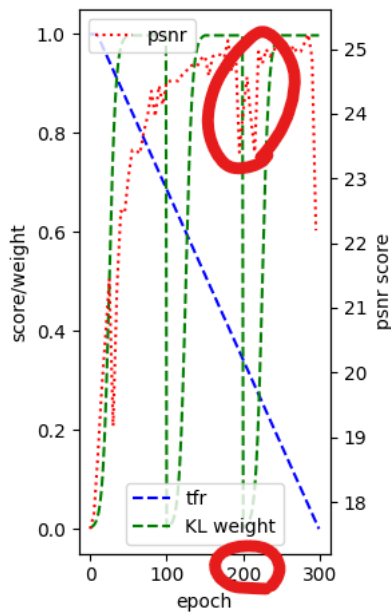
Models with both of the annealing methods reached the **peak** PSNR scores at around **285** out of total 300 epochs. Surprisingly, the model using **monotonic annealing** has slightly better performance.

My hypothesis to this result is that cyclical has many chances (in our default setting, **3** chances for cycle number = 3) to *readjust* itself. So, at the beginning of each cycle, since the KL weight  $\beta$  drops to 0, the posterior distribution of latent  $z_t$  is not forced to follow the standard Normal anymore, and can slightly adjust itself. However,  $\beta = 0$  or small  $\beta$  also **nearly remove the contribution of KL divergence to the total loss**, causing the KL loss to **go up** a bit, as one can see from the KL loss curve figure of cyclical annealing model above (at around epoch **100** and **200**).



Especially at around epoch **200**, if the KL divergence goes **too large** that the backward propagation cannot suppresses it *immediately*, then it might cause validate

PSNR to **plummet**, as one can see this is exactly what happen in the *PSNR curve figure of cyclical annealing model* above, at around epoch 200.

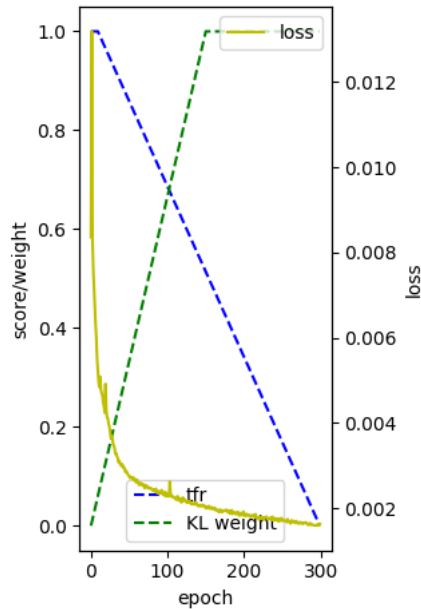


It does **take some epochs to recover** to the original level of high PSNR scores, which wasted some training epochs. On the other hand, **monotonic annealing** has a stable loss curve, and thus **does not need to waste any extra epoch to save** the deteriorated PSNR scores back to original level. Instead, the model using monotonic annealing can make a good use of these saved epochs to **even further raise the PSNR**. I think this is the reason why both annealing methods lead to similar high PSNR scores, but monotonic one is *slightly* better than the other one.

### For the **timing of teacher forcing ratio decay**

From the figures above, one can see that my teacher forcing ratio **linearly decays** from 1 to 0, starting from **epoch 10**, no matter in which KL annealing modes. (See

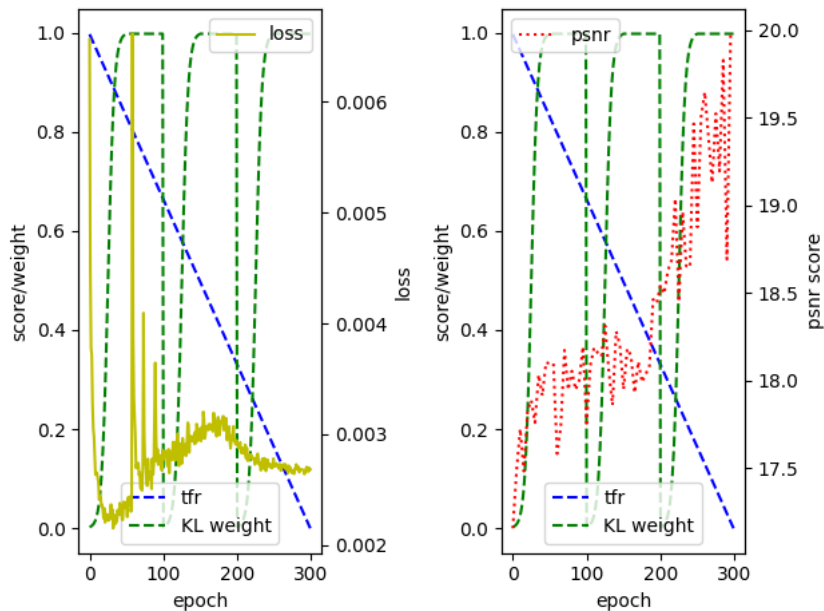
the blue curve below)



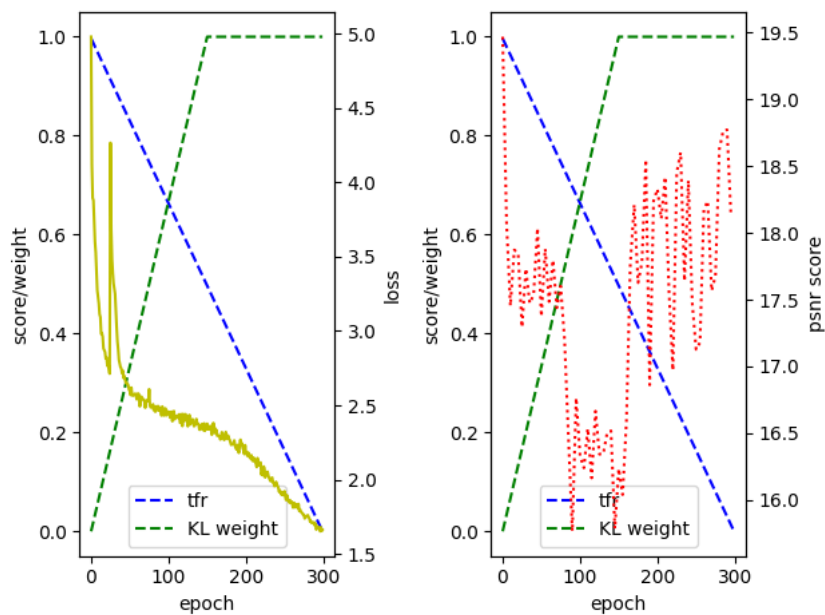
This setting leads to a nice result in both annealing schedules. Originally, I did not change the default setting that the epoch from which teacher forcing ratio starts to decay `args.tfr_start_decay_epoch` being **0**.

Nevertheless, the result turn out to be a mess. Please refer to the figures below:

### ***Cyclical Mode, tfr decaying from epoch 0***



## ***Monotonic Mode, tfr decaying from epoch 0***



In cyclical mode, the total loss starts to increase but not decreasing in the first 180 epochs. Although in the later half of training, the loss did decrease and the PSNR did increase suddenly, it failed to reach high PSNR scores within 300 epochs. In the end, the best PSNR stops at around **20**.

As for monotonic mode, despite the good-looking loss curve which has a decreasing trend, the PSNR scores basically fixed and oscillated at around **18**, simply worse, not to mention that the best PSNR **19.47** happened at the beginning, generated by the model without any training! I reckon the reason why a little **delay** imposed on teacher forcing ratio decay improves significantly (in my successful trials, **10 epochs**), is that this gives the model **more time to be guided by the teachers** (encoded embeddings of the ground truth frames), and thus **more time to learn** a better reconstruction ability. If the teacher forcing ratio decays too early, the models might be obliged to **rely on themselves** from an **early epoch**. If the models are still very **weak** at that time, they might use the wrong results generated, which causes error propagation problems, leading to failure in growth along the proper way! To make a metaphor, if a juvenile is not properly disciplined by his/her parents/teachers, he/she is likely to become a delinquent later on.

## For different *learning rate*

As for the learning rate, I did not have enough time for completing the training before deadlines. So, all my models have a fixed learning rate:  $2 \times 10^{-4}$ . However, I assume that ***smaller*** learning rate might cause the training processes **fail to converge** before 300 epochs, and thus lead to mild PSNR scores. In contrast, ***larger*** learning rate might fail to converge as well, but in another way: the loss might be **oscillated** between 2 different values, but not continue decreasing. The learning rate being too large hinges the gradients descent into the local minimum of the loss surface, since the step size is too large.