Lab 5: CVAE

310551083 許嘉倫

1 Introduction

In this lab, we need to implement a conditional VAE for video prediction. Our 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 code h_{t-1}. Then, we will sample z_t from fixed prior. Finally, we take the output from the encoder and z_t with the action and position (the condition) as the input for the decoder and we expect that the output frame should be next frame $\hat{x_t}$. We can see the training and generation procedure in figure1.

1.1 Dataset

We use bair robot pushing small dataset to train CVAE. The dataset contains roughly 44,000 sequences of robot pushing motions, and each sequence include 30 frames, action and end effector position. In this lab, we only use 2 past frames to predict 10 future frames.

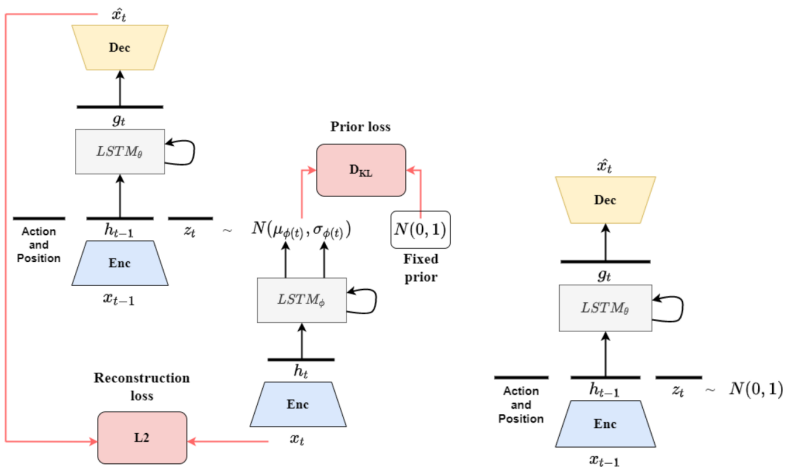


Figure 1: (a) Training procedure (b) Generation procedure

2 Derivation of CVAE

Figure 2 shows the derivation of CVAE.

3 Implementation details

3.1 vgg64 encoder

The implementation of vgg64 encoder is as follow. This encoder will output latent code and 4 different scale feature map that will be use to decode predicted frame.

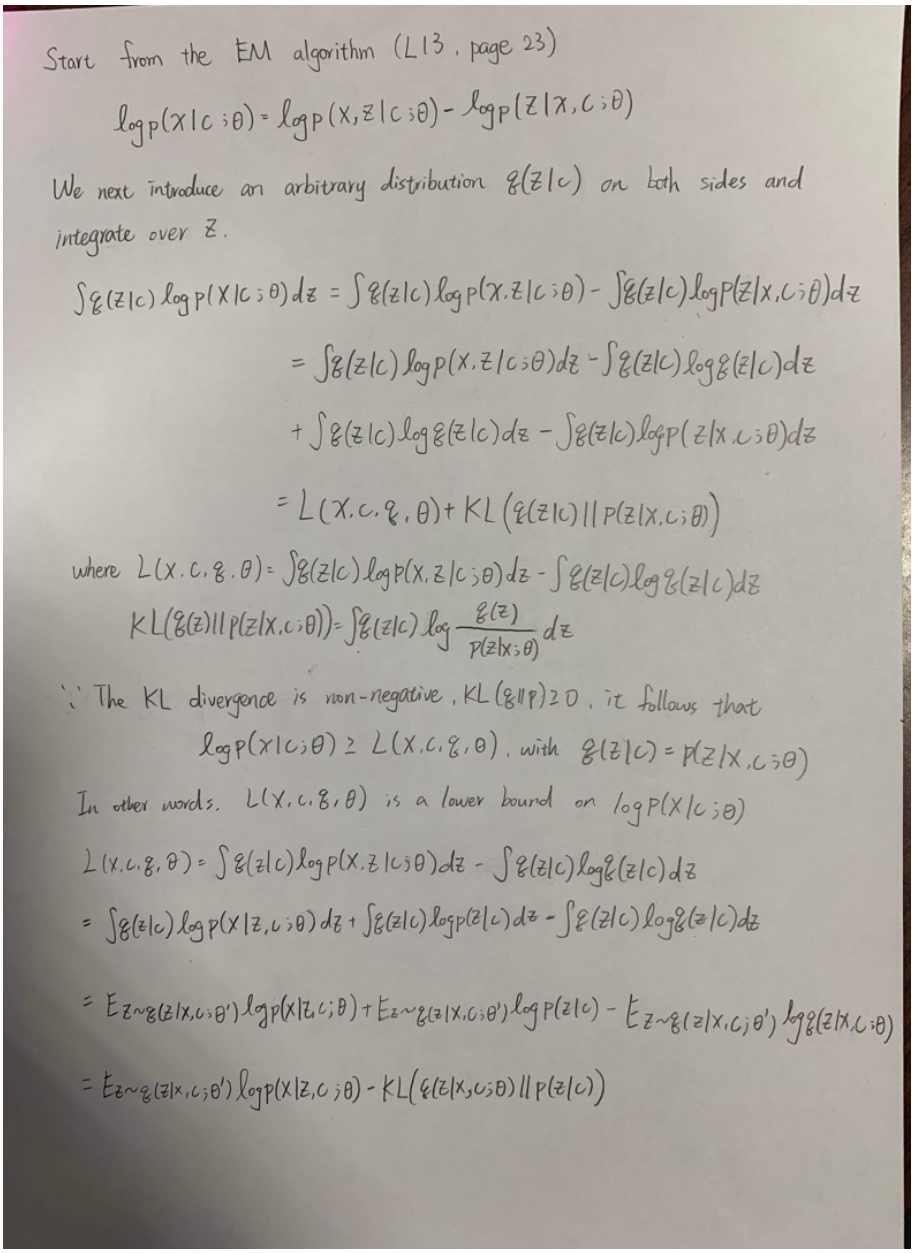


Figure 2: Derivation of Conditional VAE

```
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]
```

3.2 LSTM

The module lstm is used to embed output of vgg64 encoder to a latent vector. And this latent code will be used to decode predicted frame.

The module gaussian lstm is used to learn gaussian distribution. And we will use posterior to reconstruct predicted frame.

```
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)

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):
        # TODO
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    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
```

3.3 vgg64 decoder

The implementation of decoder is as follow. Use the output of LSTM and 4 different scale feature map of vgg64 encoder to predict next time step frame.

```
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
```

3.4 reparameterization trick

To train the model end-to-end, we adopt the reparameterization trick. The output of reparameterization trick should be log variance instead of variance directly.

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

3.5 dataloader

The implementation of dataloader is as follow. In this lab, we only use 12 images of sequence to train our model and each image should be normalized between 0 and 1 . In addition, we need to concatenate action and end effector position as conditional data.

```python
class bair_robot_pushing_dataset(Dataset):
    def __init__(self, args, mode='train', transform=default_transform):
        # TODO
        assert mode == 'train' or mode == 'test' or mode == 'validate'
        self.root_dir = args.data_root
        self.data_dir = os.path.join(self.root_dir, mode)
        if mode == 'train':
            self.ordered = False
        else:
            self.ordered = True

        self.dirs = []
        for d1 in os.listdir(self.data_dir):
            for d2 in os.listdir(os.path.join(self.data_dir, d1)):
                self.dirs.append(os.path.join(self.data_dir, d1, d2))

        self.seq_len = args.n_past + args.n_future
        self.seed_is_set = False
        self.d = 0
        self.transform = transform
        self.cur_dir = ''

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

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

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

        image_seq = []
        for i in range(self.seq_len):
            fname = os.path.join(self.cur_dir, f'{i}.png')
            im = self.transform(Image.open(fname)).view(1, 3, 64, 64)
            image_seq.append(im)
        image_seq = torch.cat(image_seq, dim=0)
        return image_seq
```

```
    def get_csv(self):
        d = self.cur_dir
        csv_seq = []
        actions = list(csv.reader(open(os.path.join(d, 'actions.csv'),
                    newline='')))
        endeffector = list(csv.reader(open(os.path.join(d,
                    'endeffector_positions.csv'), newline='')))
        for i in range(self.seq_len):
            row_list = actions[i]
            row_list.extend(endeffector[i])
            csv_seq.append(row_list)
        csv_seq = torch.tensor(np.array(csv_seq).astype(np.float),
                               dtype=torch.float)
        return csv_seq

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

3.6 Describe the teacher forcing

The implementation of teacher forcing ratio decay is as follow. I let teacher forcing ratio decay linearly, so when last epoch, it will decay to teacher forcing lower bound.

```
if epoch >= args.tfr_start_decay_epoch:
    ### Update teacher forcing ratio ###
    slope = (1.0 - args.tfr_lower_bound) / (args.niter - args.tfr_start_decay_epoch)
    tfr = 1.0 - (epoch - args.tfr_start_decay_epoch) * slope
    args.tfr = min(1, max(args.tfr_lower_bound, tfr))
```

When training the model, using teacher forcing will feed ground truth frame into encoder to get latent code, and then predict next time step frame. If not using teacher forcing, we will feed predicted frame into encoder, and then predict next time step frame, repeat this way until the end of training.
The benefits of teacher forcing is let model learn correct input to predict next frame. But the drawbacks is that our work is to use predicted frame to predict next time step frame, the input data may has some bias. Therefore, the next time predicted frame may also has some bias.
To avoid this situation, we use teacher forcing ratio decay strategy. In the beginning, let model learn ground truth input to predict next time step frame. After some epochs, teacher forcing ratio decline let model learn bias input to predict next time step frame.

```
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_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
    x = x.permute(1, 0, 2, 3 ,4)
    cond = cond.permute(1, 0, 2)
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    for i in range(1, args.n_past + args.n_future):
        h_target = h_seq[i][0]
        if args.last_frame_skip or i < args.n_past:
            h, skip = h_seq[i-1]
        else:
            h = h_seq[i-1][0]

        z_t, mu, logvar = modules['posterior'](h_target)
        h_pred = modules['frame_predictor'](torch.cat([cond[i-1], h, z_t], 1))
        x_pred = modules['decoder']([h_pred, skip])
        mse += mse_criterion(x_pred, x[i])
        kld += kl_criterion(mu, logvar, args)
        if not use_teacher_forcing :
            h_seq[i] = modules['encoder'](x_pred)
    beta = kl_anneal.get_beta()
    loss = mse + kld * beta
    loss.backward()
    optimizer.step()
```

4 Results and discussion

4.1 Show your results of video prediction

Figure 3 is the screenshot of gif, we can see the gif images for test in the submitted zip file. Figure 4 shows the ground truth and prediction at each time step. The first 2 frames of prediction sequence are ground truth, the following 10 frames are prediction given past 2 frames.

Figure 3: Screenshot of gif

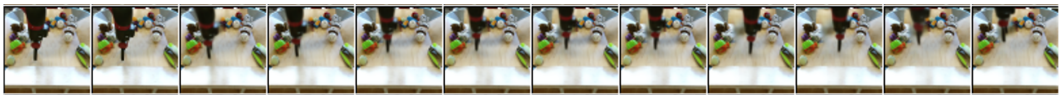
4.2 Plot the KL loss and PSNR curves during training

Figure 5 is the training curve of monotonic and cyclical.

4.3 Discuss the results according to your setting

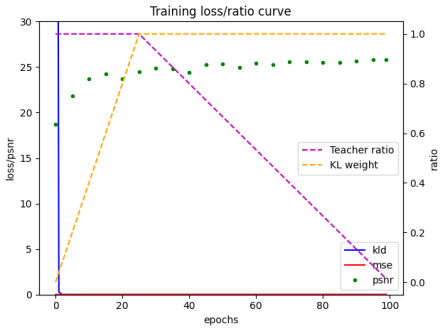
The follow hyperparameters are same in monotonic and cyclical. The difference between monotonic and cyclical is KL weight as figure 5 shows.

(a) ground truth

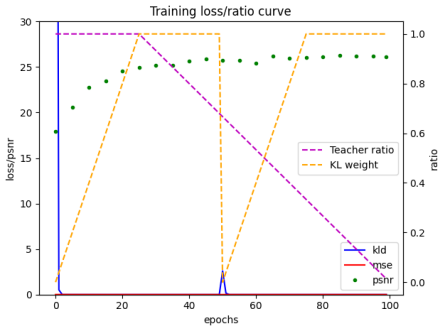


(b) prediction

Figure 4: ground truth and prediction at each time step



(a) monotonic



(b) Cyclical: 2 cycle

Figure 5: Training curve of monotonic and cyclical

- epoch: 100
- beginning tfr: 1.0
- tfr lower bound: 0.0
- tfr start decay epoch
- learning rate: 0.002

4.3.1 KL annealing monotonic and cyclical

In the beginning of training, we hope that the model can focus on mean square loss. Another reason is that KL loss is too higher (roughly 150) than MSE loss (roughly 0.02), the loss may be dominated by KL loss. We can see KL annealing cyclical as finetuning the model cycle-by-cycle. As figure 5 shows, the KL loss suddenly increase to roughly 2.6 in epoch 50. But the value of KL loss is lower than KL loss of epoch 1. And through this way, we can get better PSNR than monotonic as following table shows.

	monotonic	cyclical
PSNR	25.61	**26.25**

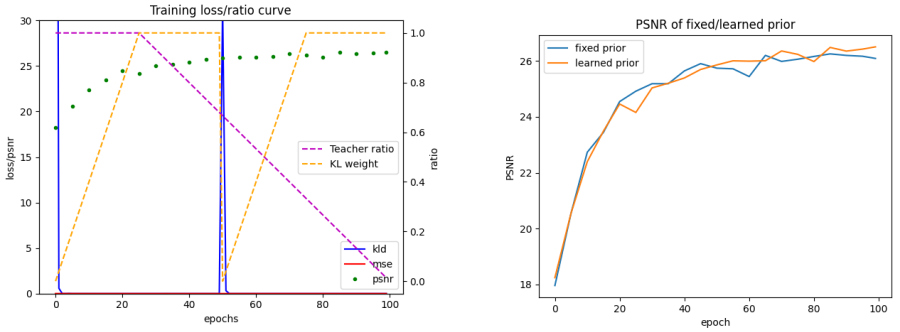
4.3.2 Teacher forcing ratio

As section 3.5 said, our work is using first two ground truth frame to predict 10 future frames. Therefore, we need to use our predicted frame to predict next time step frame. But when training the model, we can't let teacher forcing as 0 at the beginning. Because predicted input data may have bias or may be too blurred, it will influence our model to reconstruct correct frame. To avoid this situation, I set teacher forcing ratio as 1 at first 25 epochs, decay to 0 linearly. This way can let model learn correct reconstruction given ground truth input data at first stage. Then learning reconstruction given bias input data.

5 Extra: implement learned prior

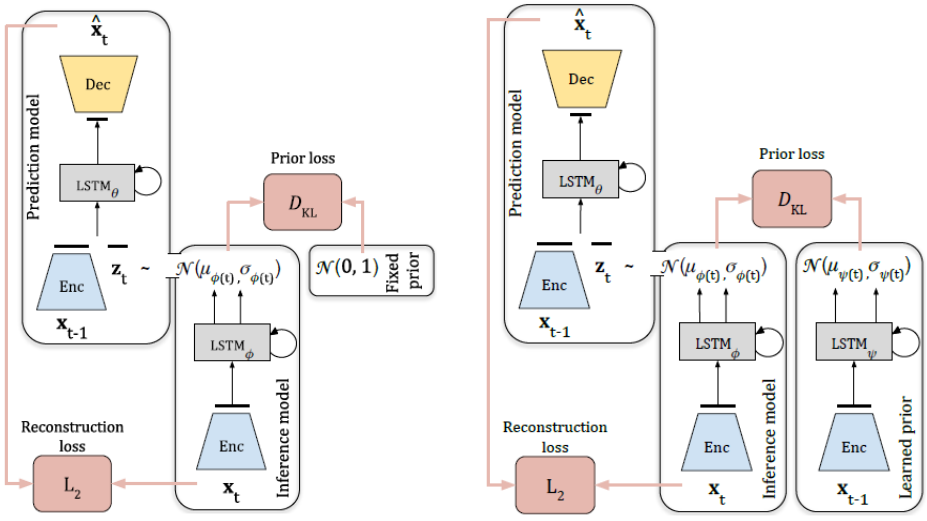
The following table shows the PSNR of fixed and learned prior. As we can see, PSNR of learned prior is higher than fixed prior roughly 0.27. Figure 6 shows the training curve of learned prior and PSNR of learned prior and fixed prior. The difference of learned prior and fixed prior is that replace original normal distribution with learnable distribution just like posterior, we can see the difference in figure 7

	fixed prior	learned prior
PSNR	26.25	**26.52**



(a) Training curve of learned prior (b) PSNR of learned prior and fixed prior

Figure 6: Training curve of learned prior and PSNR of learned prior and fixed prior



(a) fixed prior model (b) learned prior model

Figure 7: Difference of fixed and learned prior model

The implementation of learned prior training is as follow.

```
def kl_criterion_lp(mu1, logvar1, mu2, logvar2, args):
    # KL( N(mu_1, sigma2_1) || N(mu_2, sigma2_2)) =
    #   log( sqrt(
    #
    sigma1 = logvar1.mul(0.5).exp()
```

```
    sigma2 = logvar2.mul(0.5).exp()
    kld = torch.log(sigma2/sigma1) + (torch.exp(logvar1) + (mu1 -
    mu2)**2)/(2*torch.exp(logvar2)) - 1/2
    return kld.sum() / args.batch_size


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_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
    x = x.permute(1, 0, 2, 3 ,4)
    cond = cond.permute(1, 0, 2)
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    for i in range(1, args.n_past + args.n_future):
        h_target = h_seq[i][0]
        if args.last_frame_skip or i < args.n_past:
            h, skip = h_seq[i-1]
        else:
            h = h_seq[i-1][0]

        z_t, mu, logvar = modules['posterior'](h_target)
        _, mu_p, logvar_p = modules['prior'](h)
        h_pred = modules['frame_predictor'](torch.cat([cond[i-1], h, z_t], 1))
        x_pred = modules['decoder']([h_pred, skip])
        mse += mse_criterion(x_pred, x[i])
        kld += kl_criterion_lp(mu, logvar, mu_p, logvar_p, args)
        if not use_teacher_forcing :
            h_seq[i] = modules['encoder'](x_pred)

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

    optimizer.step()
```