Lab5: CVAE For Video Prediction

Wei-Yun Hsu
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University

1. Introduction

In this lab, we will need to use a conditional VAE for video prediction.
There are the following requirements

- Implement a conditional VAE model
- Implement dataloader, teacher forcing, KL annealing, and reparameterization trick
- Plot the training loss and PSNR curves during training
- Make videos or gif images for test result
- Output the prediction at each time step

A. Dataset

The bair robot pushing small dataset contains roughly 44,000 sequences of robot pushing motions, and each sequence include 30 frames. For the each time step, it also contains action and end-effector position.

2. Derivation of CVAE

As we all know, both VAE and CVAE are generative models. It can be seen literally that VAE is a purely generative model, which means that it cannot control what it generates, while CVAE can be generated according to a given label. In this lab, we need to predict the next frame based on the given previous frames using CVAE, which in parallel to encoder output also take the condition as inputs, it is a main difference with VAE.

Start from EM algorithm, we can know that

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

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

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

Next, we introduce an arbitrary distribution $q(z|c)$ on both sides and integer over z

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

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

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

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

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

Then, we introduce a distribution $q(z|x, c; \phi)$ which is modeled by a neural network with parameter ϕ

$$L = \log p(x|c; \theta)$$

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

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

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

$$- \int q(z|c) \log q(z|c)\, dz$$

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

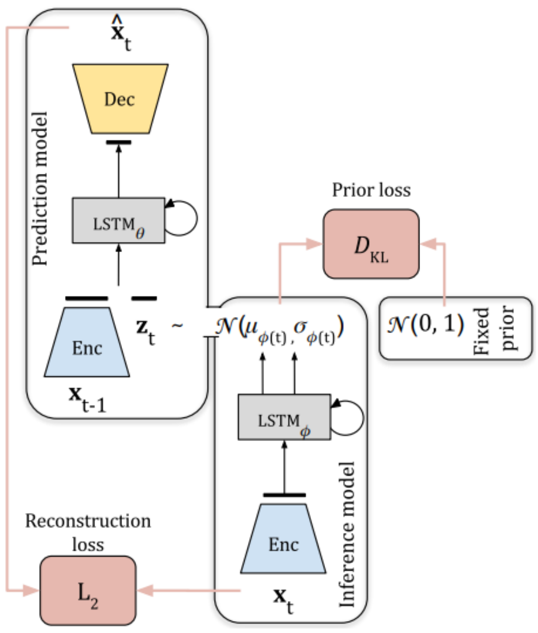
$$- E_{z \sim q(z|x, c; \phi)} \log q(z|x, c; \theta)$$

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

3. Implementation details

A. Describe how you implement your model

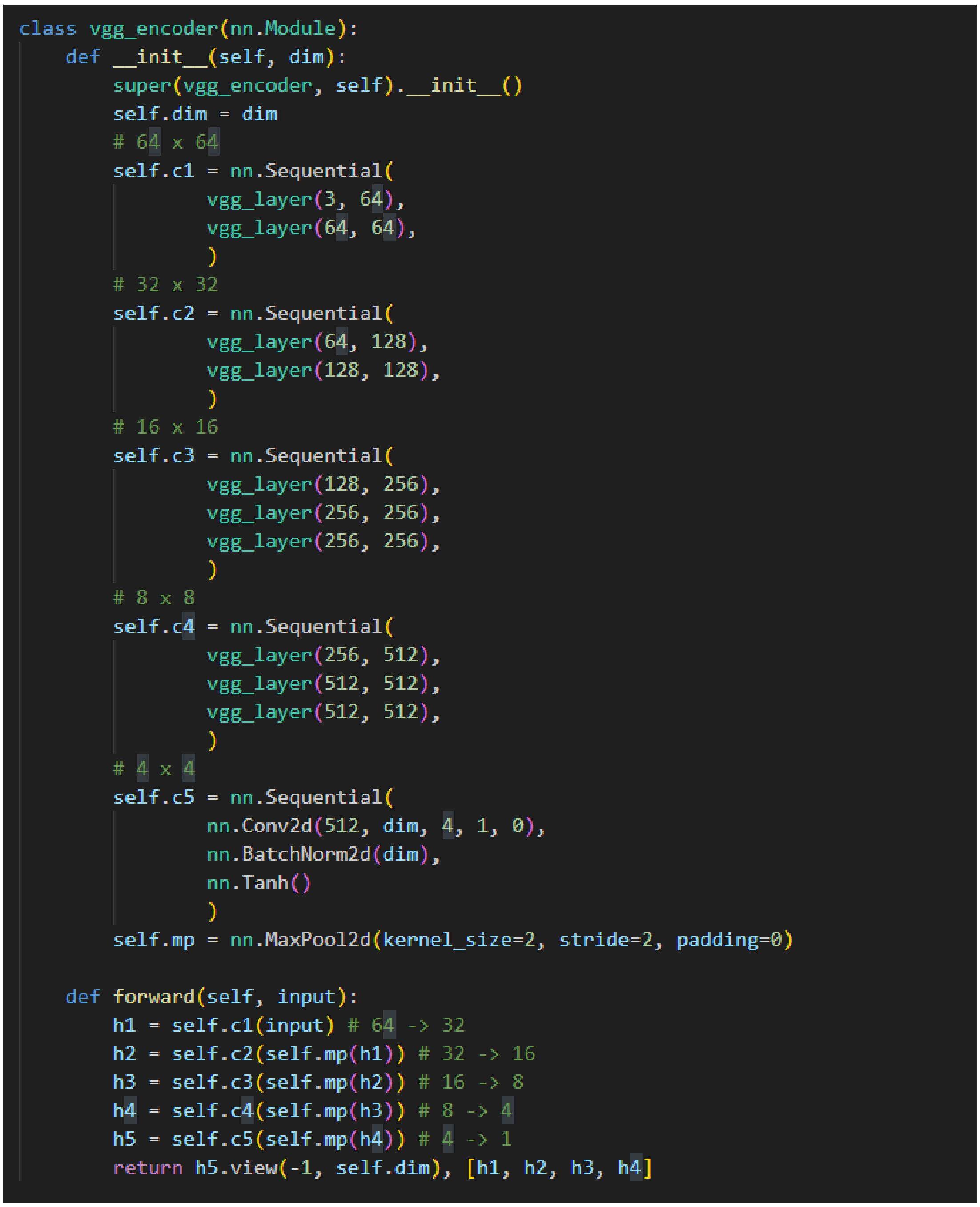
In this part, we use VGGNet, which was provided by TA, as encoder and decoder of the prediction model. We put the frame x_{t-1} into encoder with $t-1$ timestamp of the video, and the decoder was fed by the output of the encoder, the condition (action and position) and a latent variable z_t which is sampled at each time step. Then, we use the generated frame $\hat{x}_t$ and the ground truth frame x_t to calculate reconstruction loss. The overall model architecture is as follows.



A-1. Encoder

We use different output channels for about 10 vgg layers to suppress the input data from 64×64 to 4×4. The vgg layers contain a 2D convolutional layer, a batch normalization layer and a activation layer. Here, the LeakyReLU activation function was applied. The implementation of encoder is as follows.

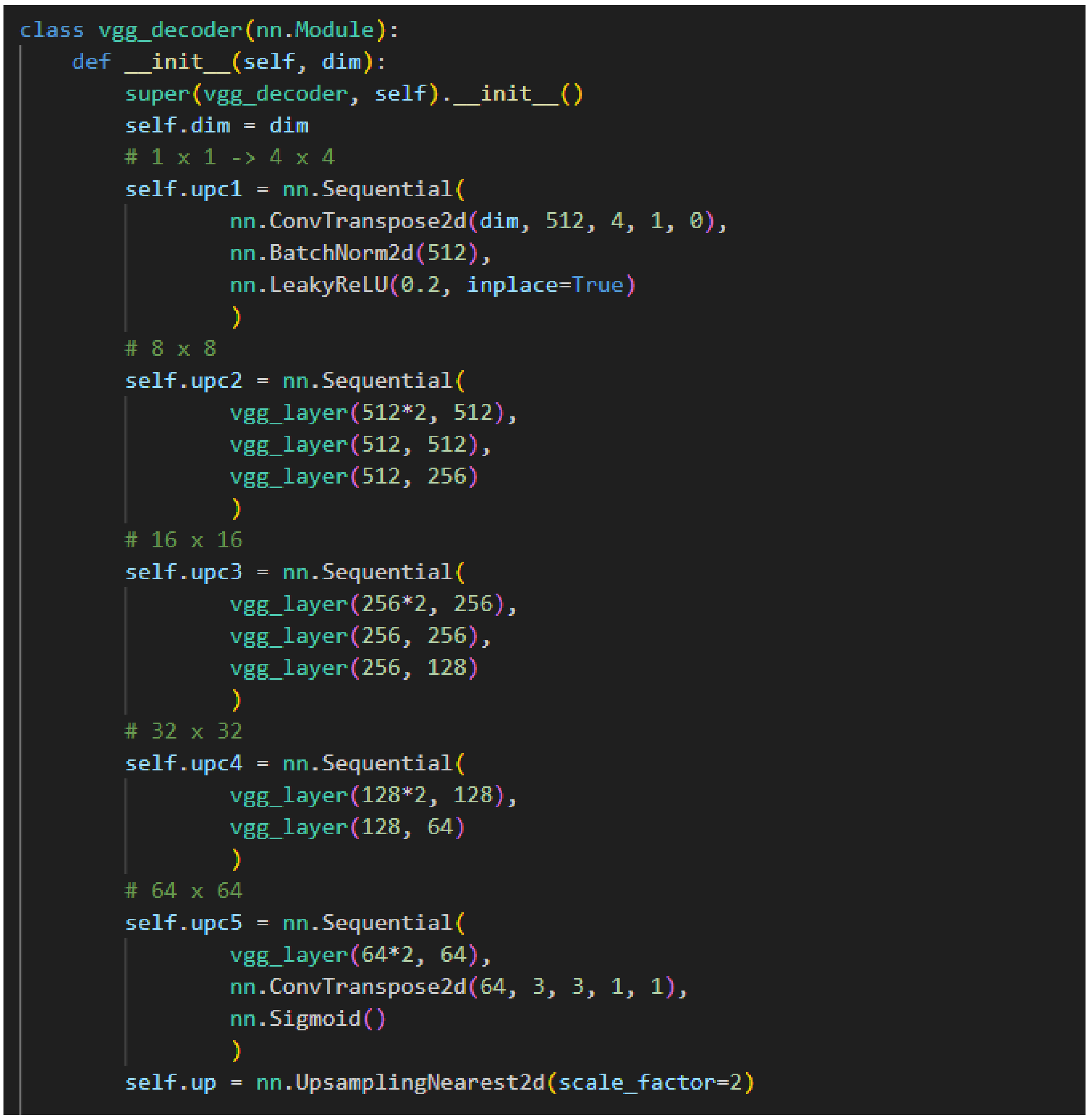
```
class vgg_layer(nn.Module):
    def __init__(self, nin, nout):
        super(vgg_layer, self).__init__()
        self.main = nn.Sequential(
                nn.Conv2d(nin, nout, 3, 1, 1),
                nn.BatchNorm2d(nout),
                nn.LeakyReLU(0.2, inplace=True)
                )

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



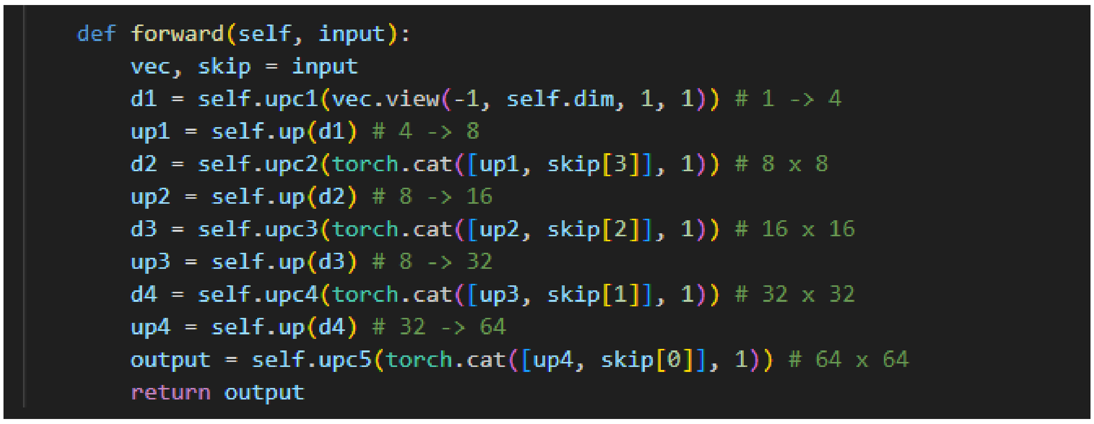
```python
class vgg_encoder(nn.Module):
    def __init__(self, dim):
        super(vgg_encoder, self).__init__()
        self.dim = dim
        # 64 x 64
        self.c1 = nn.Sequential(
                vgg_layer(3, 64),
                vgg_layer(64, 64),
                )
        # 32 x 32
        self.c2 = 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]
```

A-2. Decoder

In the decoder, it's a little bit like the encoder does in reverse, it reconstructs the original data, the dimension is from 1×1 to 64×64 using the different layer size of vgg laygers. The implementation of decoder is shown below.



```python
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)
```



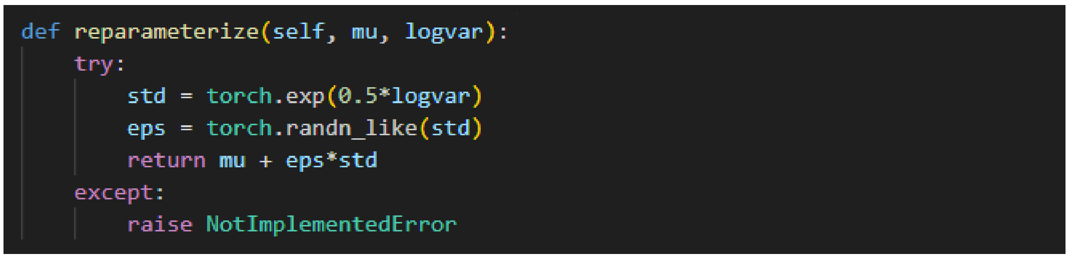
```python
    def forward(self, input):
        vec, skip = input
        d1 = self.upc1(vec.view(-1, self.dim, 1, 1)) # 1 -> 4
        up1 = self.up(d1) # 4 -> 8
        d2 = self.upc2(torch.cat([up1, skip[3]], 1)) # 8 x 8
        up2 = self.up(d2) # 8 -> 16
        d3 = self.upc3(torch.cat([up2, skip[2]], 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
```

A-3. Reparameterization trick

The VAE predicts the parameters of a distribution which then is used to generate encoded embeddings. One of VAE’s problem is this process of sampling from a distribution that is parameterized by our model is not differentiable. We can solve this problem by applying the reparameterization trick to our embedding function, which means that we need to treat random sampling as a noise term. The reparameterization trick rewrites the representation of z as a random variable into the following expression:

$$z \in N(\mu, \sigma^2) \rightarrow z = \sigma \odot \epsilon, \text{ where } z \in N(0, 1)$$

The implementation is as follows.



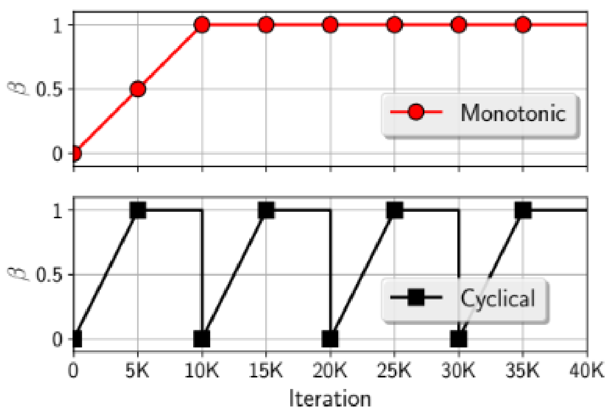
```python
def reparameterize(self, mu, logvar):
    try:
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std
    except:
        raise NotImplementedError
```

A-4. KL annealing

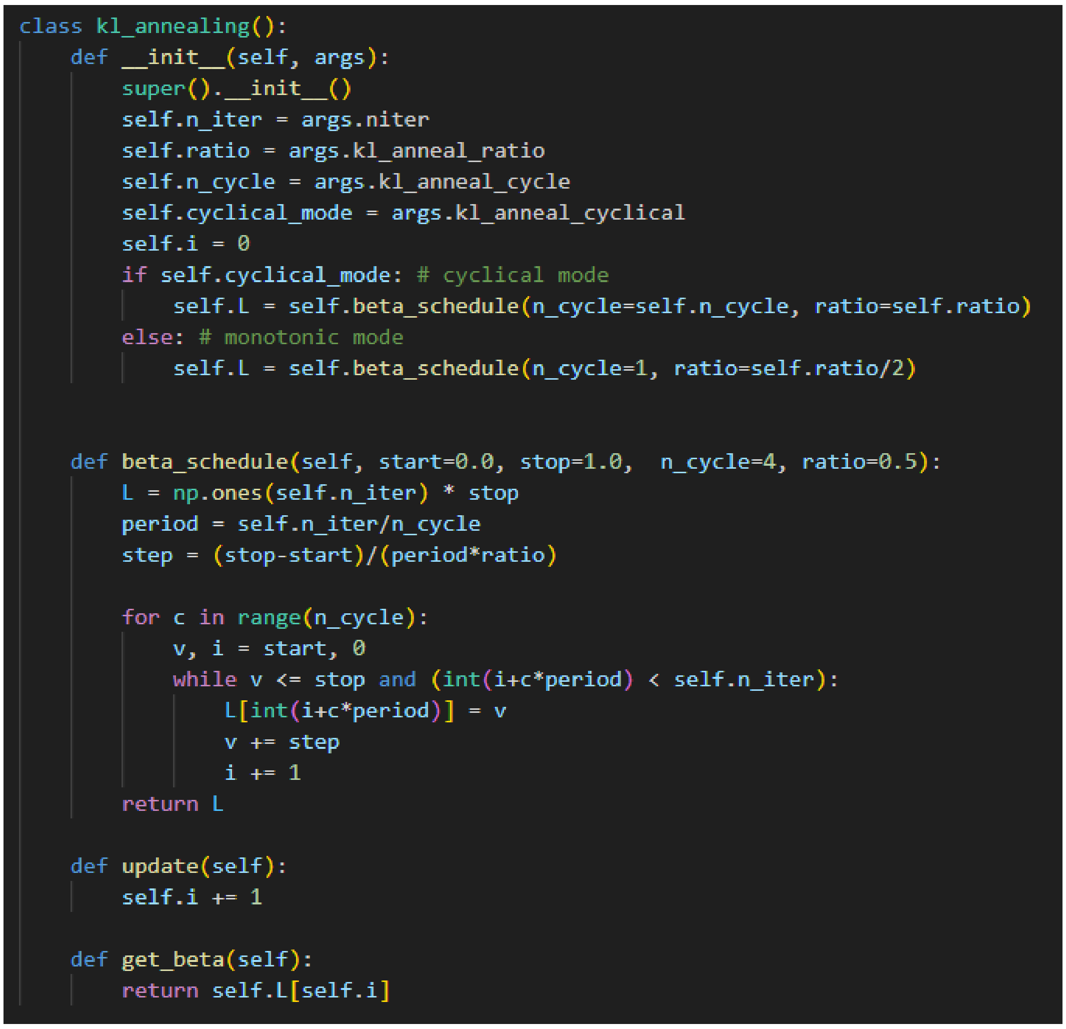
The objective function of VAE has two components, reconstruction loss and KL loss. The former indicates how well VAE can reconstruct the input data from the latent variables, while the latter measures how similar two data distributions are with each other. We usually calculate the KL divergence between the latent distribution and standard normal distribution. As we can see in the below function, β controls how much KL-divergence weighs in the total loss.

$$L(x, \hat{x}) + \beta \sum_j KL(q_j(z|x)||p(z))$$

But there is a KL vanishing problem, which means that the KL term will close to zero during training. One way to alleviate this problem is to apply the annealing schedules for the KL term, this trick has two modes: monotonic and cyclical.



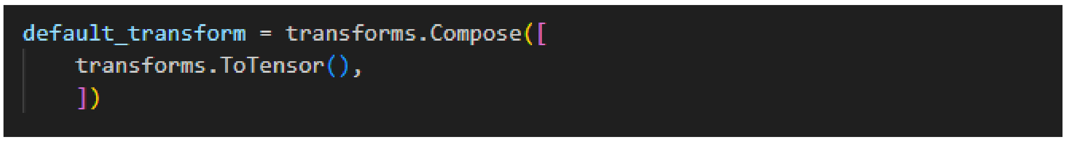
The implementation of KL annealing is as follows.



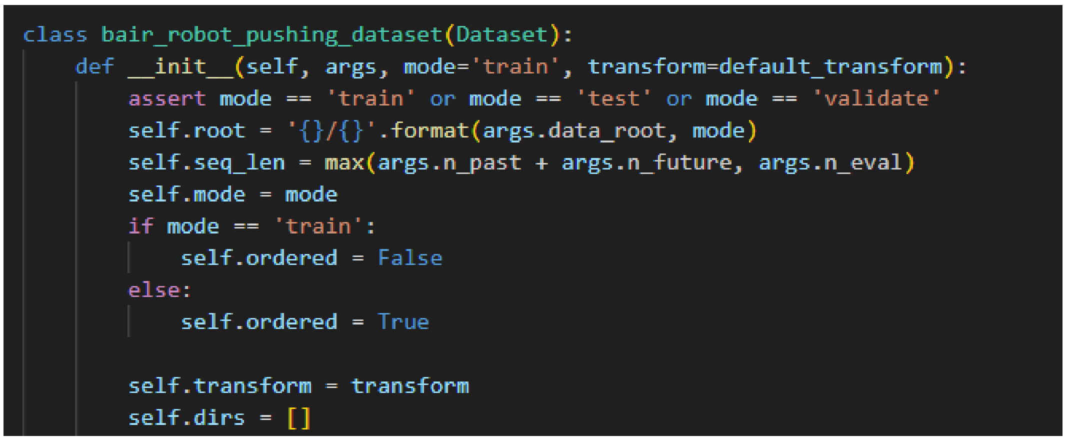
```python
class kl_annealing():
    def __init__(self, args):
        super().__init__()
        self.n_iter = args.niter
        self.ratio = args.kl_anneal_ratio
        self.n_cycle = args.kl_anneal_cycle
        self.cyclical_mode = args.kl_anneal_cyclical
        self.i = 0
        if self.cyclical_mode: # cyclical mode
            self.L = self.beta_schedule(n_cycle=self.n_cycle, ratio=self.ratio)
        else: # monotonic mode
            self.L = self.beta_schedule(n_cycle=1, ratio=self.ratio/2)


    def beta_schedule(self, start=0.0, stop=1.0,  n_cycle=4, ratio=0.5):
        L = np.ones(self.n_iter) * stop
        period = self.n_iter/n_cycle
        step = (stop-start)/(period*ratio)

        for c in range(n_cycle):
            v, i = start, 0
            while v <= stop and (int(i+c*period) < self.n_iter):
                L[int(i+c*period)] = v
                v += step
                i += 1
        return L

    def update(self):
        self.i += 1

    def get_beta(self):
        return self.L[self.i]
```

A-5. Dataloader

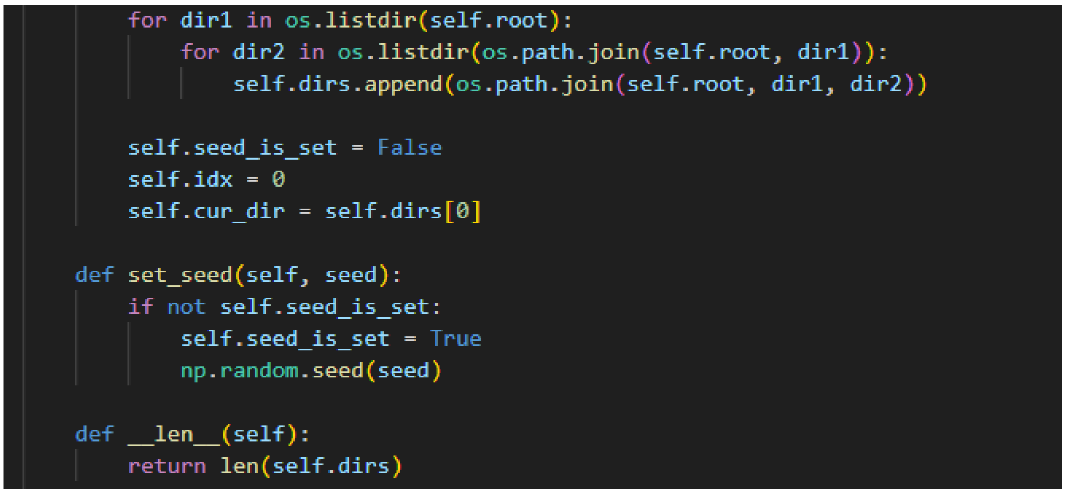
In the dataloader, in addition to the basic functions, get_seq() is also defined. It is used to get the image sequence and reshape the frames to (1, 3, 64, 64). The implementation od dataloader is as follows.



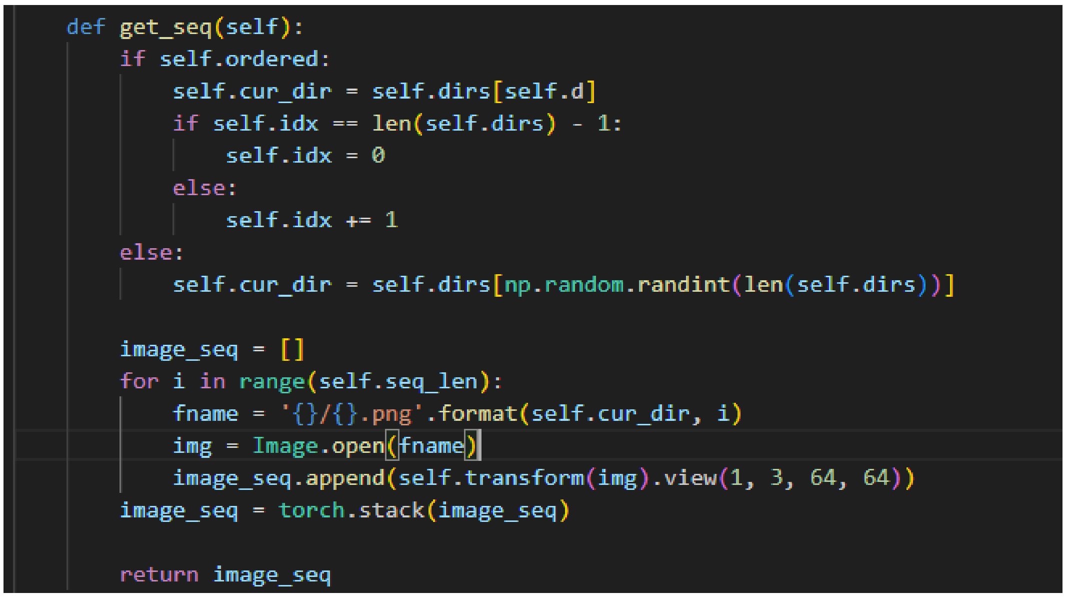
```python
default_transform = transforms.Compose([
    transforms.ToTensor(),
    ])
```



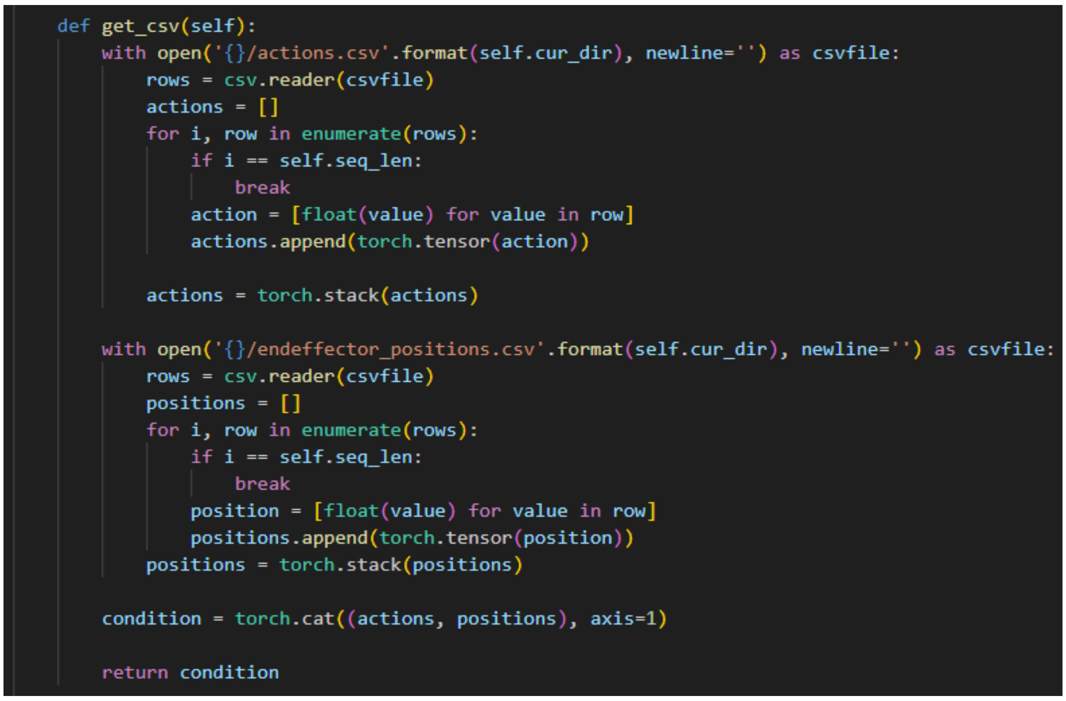
```python
class bair_robot_pushing_dataset(Dataset):
    def __init__(self, args, mode='train', transform=default_transform):
        assert mode == 'train' or mode == 'test' or mode == 'validate'
        self.root = '{}/{}'.format(args.data_root, mode)
        self.seq_len = max(args.n_past + args.n_future, args.n_eval)
        self.mode = mode
        if mode == 'train':
            self.ordered = False
        else:
            self.ordered = True

        self.transform = transform
        self.dirs = []
```



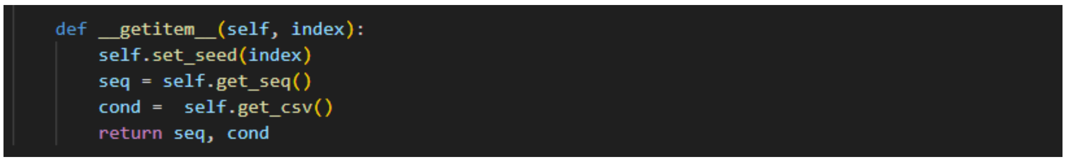
```python
        for dir1 in os.listdir(self.root):
            for dir2 in os.listdir(os.path.join(self.root, dir1)):
                self.dirs.append(os.path.join(self.root, dir1, dir2))

        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)
```



```python
    def get_seq(self):
        if self.ordered:
            self.cur_dir = self.dirs[self.d]
            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).view(1, 3, 64, 64))
        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()
    cond =  self.get_csv()
    return seq, cond
```

B. Describe the teacher forcing

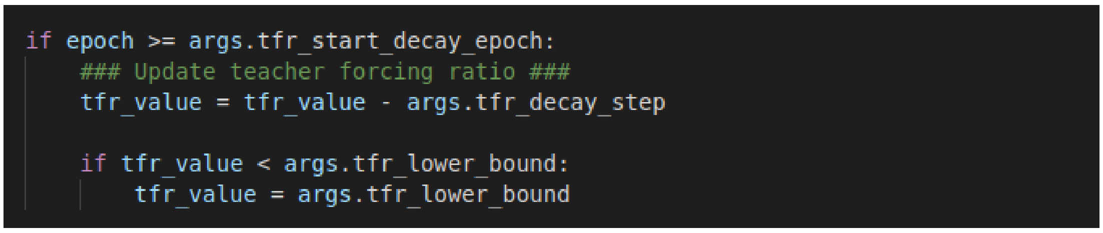
When training a model using teacher forcing, it uses the ground truth as the input of the next layer, instead of the output of this layer. If we use the output of the current layer as the input of the next layer, there are some possibility of updating model by the wrong predictions (garbage in, garbage out).

However, there are still some problems with using teacher forcing. Teacher-forcing relies too much on the ground truth data. During the training process, the model will have a better effect, but because it cannot get the support of the ground truth during the test, if the currently generated sequence is very different during the training process. In other words, the cross-domain capability of this model will be worse, that is, if the test data set and the training data set come from different domains, the performance of the model will deteriorate.

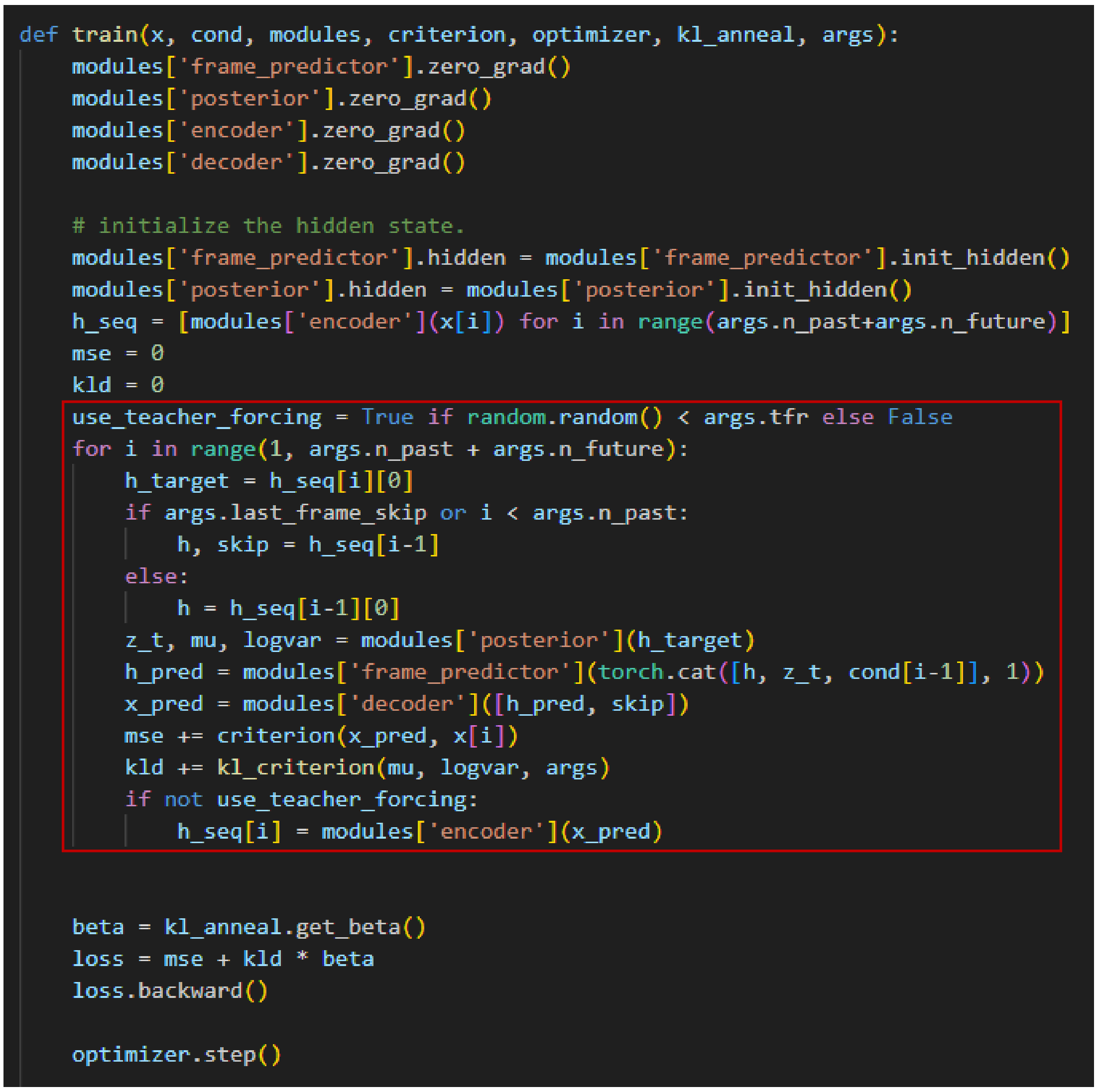
To avoid this problem, we decay the teacher forcing ratio during the training phase. There are two strategy we used to implement it.

- Decay ratio from 20 epoch, and stop decaying when the ratio reduces to 0.001. (decay step=0.01666)
- Decay ratio from 30 epoch, and stop decaying when the ratio reduces to 0.001. (decay step=0.004)

The implementation of teacher forcing is as follows.



```python
if epoch >= args.tfr_start_decay_epoch:
    ### Update teacher forcing ratio ###
    tfr_value = tfr_value - args.tfr_decay_step

    if tfr_value < args.tfr_lower_bound:
        tfr_value = args.tfr_lower_bound
```



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

    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
    mse = 0
    kld = 0
    use_teacher_forcing = True if random.random() < args.tfr else False
    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([h, z_t, cond[i-1]], 1))
        x_pred = modules['decoder']([h_pred, skip])
        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()
```

C. Experiment setup

The hyper-parameters for experiments as shown below.

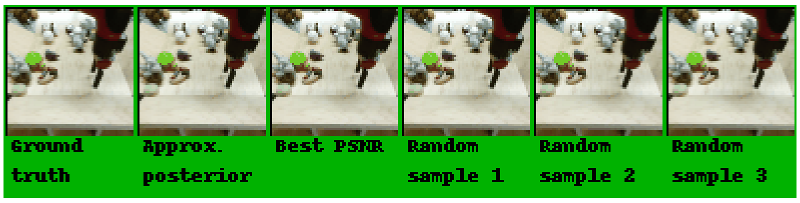
- niter: 100
- batch size: 12
- epoch: 600
- learning rate: 0.002
- optimizer: Adam

4. Results and discussion

A. Show your results of video prediction

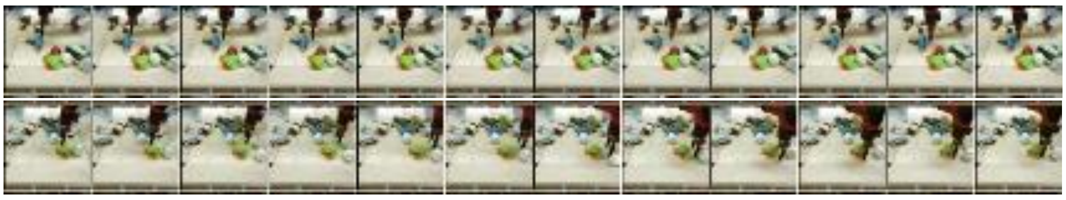
The following sections are the GIF file and the prediction at each time step of one of the best result in fixed prior we achieved, which got the average PSNR score of 26.759.

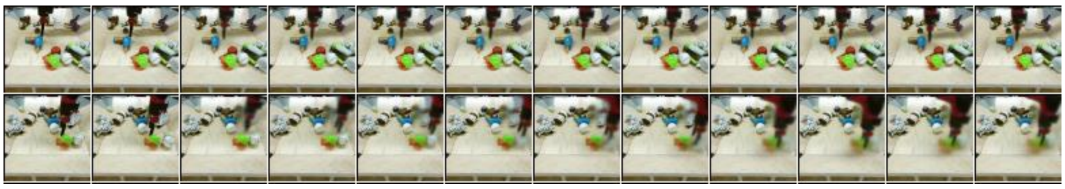
A-1. Make videos or gif images for test result



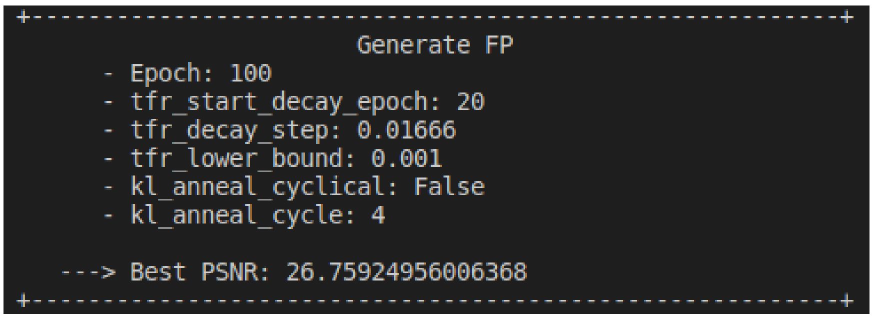
https://drive.google.com/file/d/1X5QQX6YauVxOc7V_h-mPp-cIEwrAC99o/view?usp=sharing

A-2. Output the prediction at each time step


(a) ground truth


(b) prediction

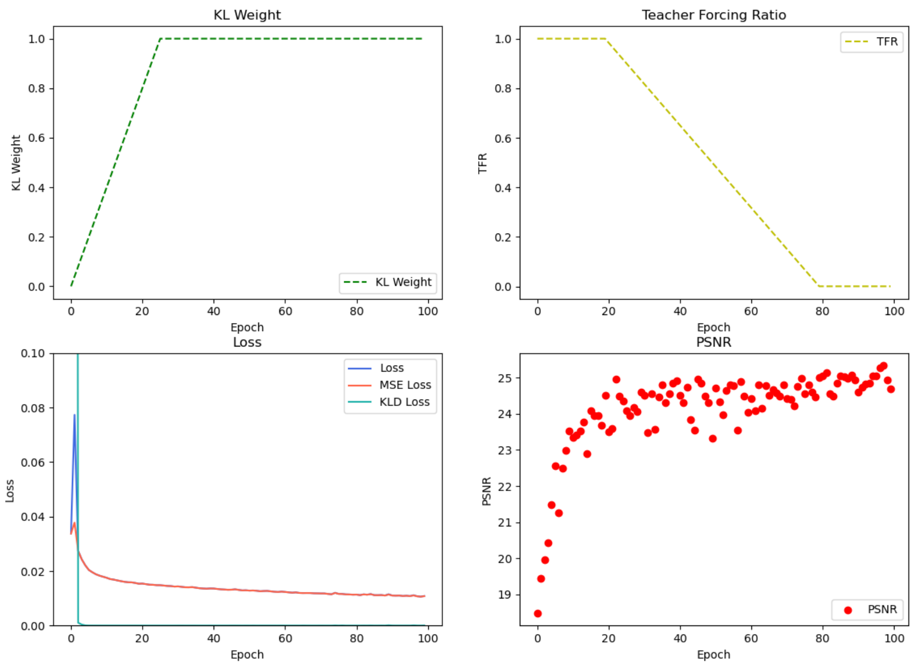
A-3. Screenshot for the best result in fixed prior



```
+----------------------------------------------------------+
                        Generate FP
      - Epoch: 100
      - tfr_start_decay_epoch: 20
      - tfr_decay_step: 0.01666
      - tfr_lower_bound: 0.001
      - kl_anneal_cyclical: False
      - kl_anneal_cycle: 4

   ---> Best PSNR: 26.75924956006368
+----------------------------------------------------------+
```

B. Plot the KL loss and PSNR curves during training

The curve of the best training result is shown below.

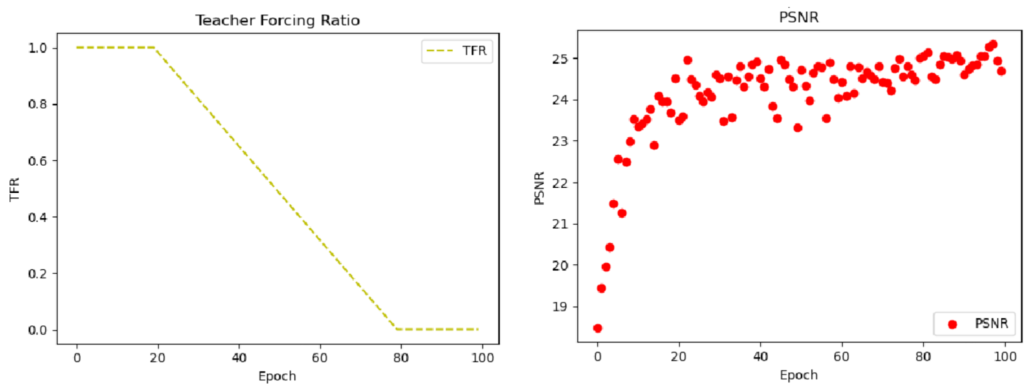


C. Discussion

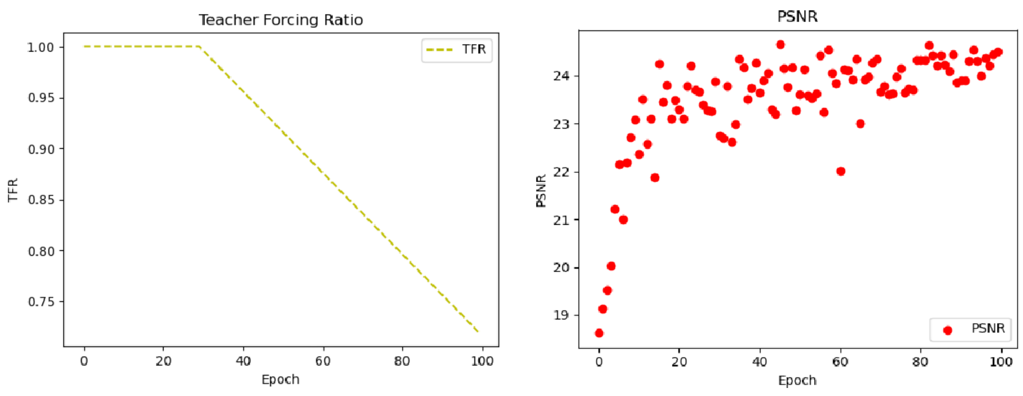
C-1. Setting of teacher forcing ratio

In this issue, we want to know how important teacher forcing ratio is during the training phase. As we mentioned in the section 3-B, if we train the model without teacher forcing strategy (free-running mode), it may be cause "garbage in, garbage out" situation, which means that the hidden layers of the model will be updated by wrong prediction that provided by the previous layer. Finally, model will converge slowly and be instable, we cannot get the expected results.

In the sample code, we can see that there are four parameters related to teacher forcing, which are tfr, tfr_start_decay_epoch, tfr_decay_step and tfr_lower_bound. The teacher forcing ratio is the probability that the model is fed the ground-truth output at each time step during training, as opposed to using its own predicted output as input for the next time step. Originally, I thought that starting to decay when more earlier epoch would make the PSNR score reach the bottleneck, just like the effect of learning rate. Here, we designed two linear decay strategies to accept or reject our idea, the first one is decaying ratio from 20 epochs and decaying step is set to 0.01666, the other one is decaying ratio from 30 epochs and decaying step is set to 0.004. After experiments, we found that the former strategy can achieve the better average PSNR score. The reason to the poor performance caused by decay latter with larger decay step is our task is a complex task, it may benefit from a higher initial teacher forcing ratio and a slower decay rate. For example, the KL annealing schedule is set to monotonic:



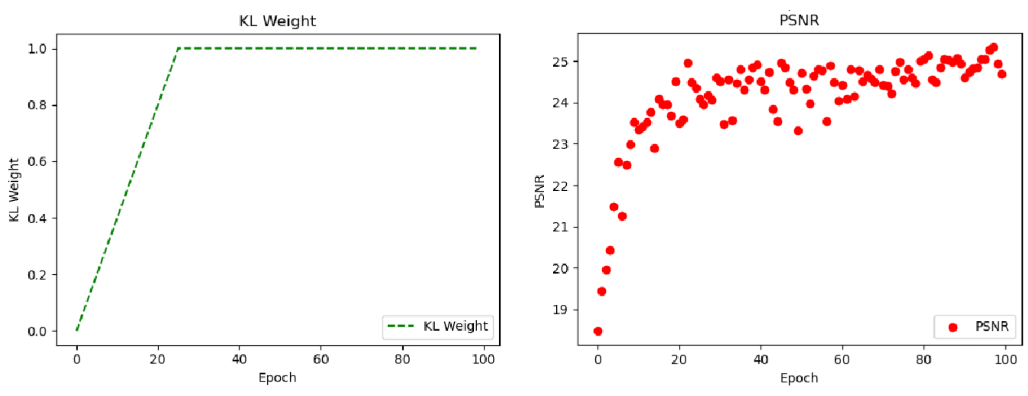
(a) tfr_start_decay_epoch = 20, tfr_decay_step = 0.01666



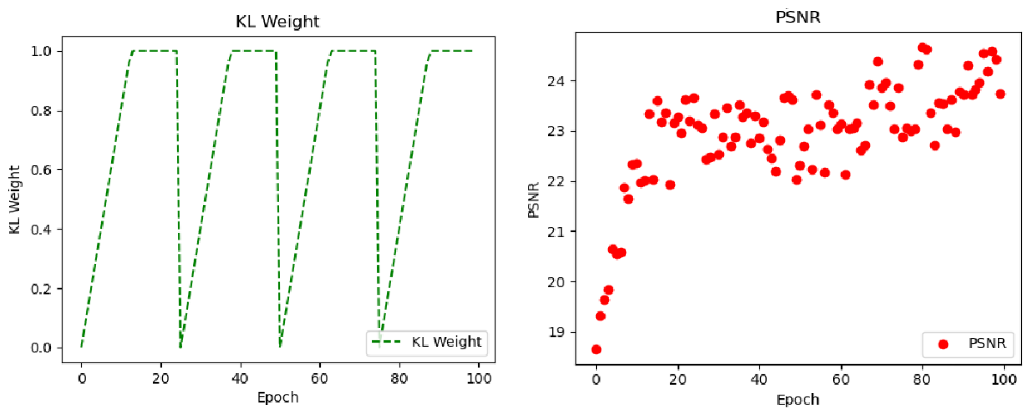
(b) tfr_start_decay_epoch = 30, tfr_decay_step = 0.004

C-2. Setting of KL weights

In this part, we want to find out which one is the better schedule between monotonic and cyclical. From the below comparison figures, we can observe that the monotonic mode can get the better scores using different TFR strategies. There is a little bit tricky because the monotonic schedule may not allow the model to explore the space of latent variables as much as a cyclical schedule, cyclical schedule may lead to better performance than monotonic schedule. The only possible reason that comes to mind is tuning the parameters incorrectly. For example, the teacher forcing is set to the first strategy mentioned in the previous section:



(a) monotonic schedule



(b) cyclical schedule

Overall, the choice of KL annealing schedule depends on the specific task and model being used, and may require some experimentation to find the optimal setting.

The overall comparison of teacher forcing and KL annealing is as follows.

	TFR strategy	KL schedule	average PSNR score
Fixed prior	Decay from 20 epochs and decay step is 0.01666	Monotonic	26.759
Fixed prior	Decay from 20 epochs and decay step is 0.01666	Cyclical	26.547
Fixed prior	Decay from 30 epochs and decay step is 0.004	Monotonic	25.795
Fixed prior	Decay from 30 epochs and decay step is 0.004	Cyclical	24.613

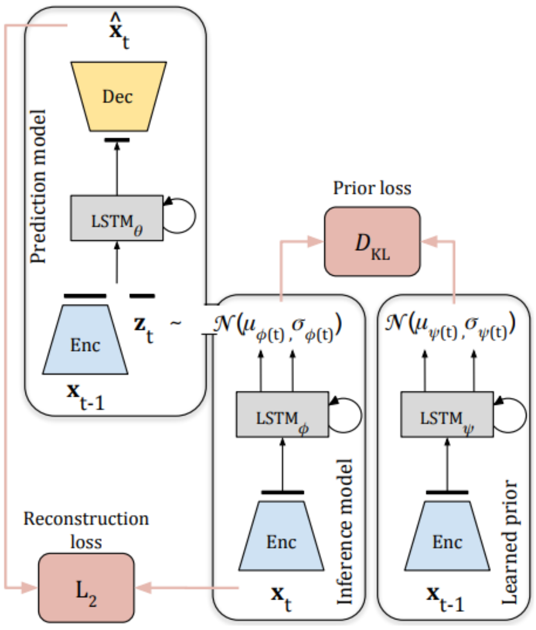
C-3. Setting of learning rate

Before doing this assignment, we planned to use the learning rate decay strategy to train our model, because without using learning rate decay, the optimizer can overshoot the optimal solution and cause the loss function to diverge, the model will fail to learn. But from the above results, it seems that good results can be achieved without learning rate decay. Therefore, we only used the default setting (0.002) as our learning rate, I think if we have more time to doing this lab, we will try to start with a relatively high learning rate and gradually reduce it as training progresses. This allows the optimizer to take large steps in the beginning when the parameters are far from the optimal values, but reduces the step size as the parameters get closer to the optimal values to help the optimizer converge to a good solution.

5. Extra

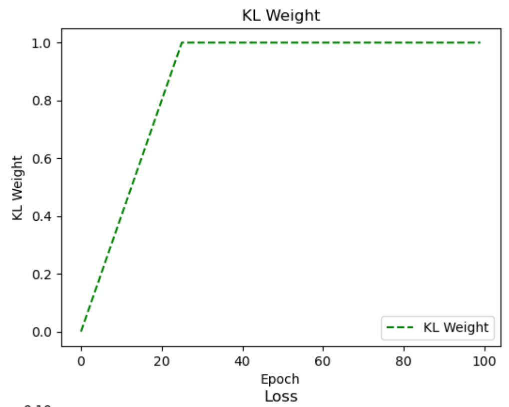
A. Implement learned prior

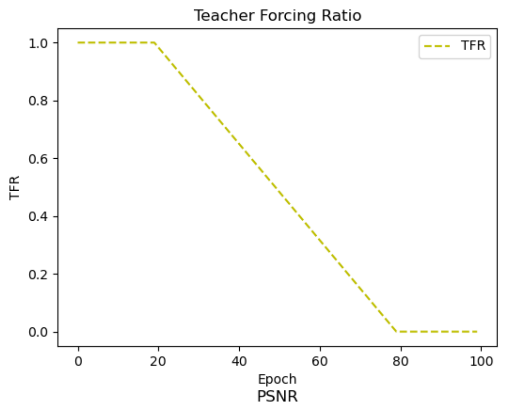
In the fixed prior distribution, the simplest choice for $p(z_t)$ is a fixed Gaussian $N(0, \mathrm{I})$, as is typically used in VAE. But a drawback is that samples at each time step will be drawn randomly, thus ignore temporal dependencies between the frames. The other method is learned prior distribution, it is a more complex method to learn a prior that varies across time, being a function of all past frames up to but not including the frame being predicted $p_\psi(z_t|x_{1:t-1})$. The overall model architecture is as follows.

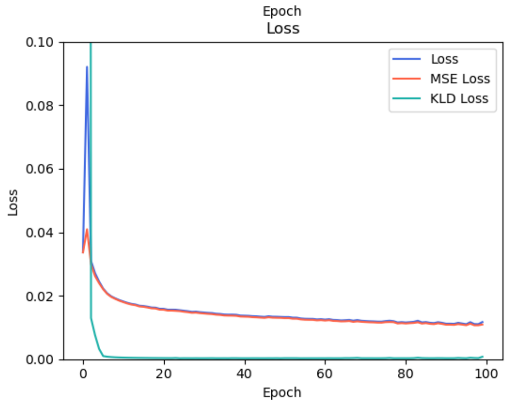


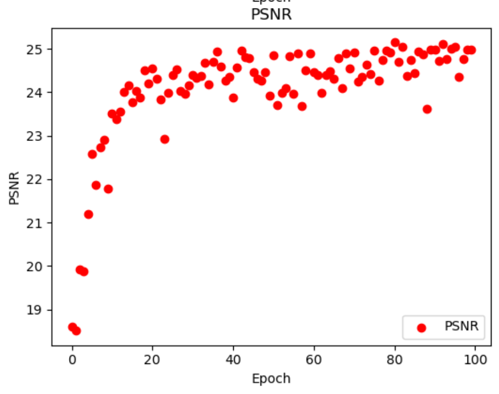
Therefore, I expected learned prior to generate better results than fixed prior. As can be seen from the following results, it is true.

	TFR strategy	KL schedule	average PSNR score
fixed prior	Decay from 20 epochs and decay step is 0.01666	Monotonic	26.759
learned prior	Decay from 20 epochs and decay step is 0.01666	Monotonic	27.424


KL Weight
KL Weight
Epoch
KL Weight


Teacher Forcing Ratio
TFR
Epoch
TFR


Loss
Loss
Epoch
Loss
MSE Loss
KLD Loss


PSNR
PSNR
Epoch
PSNR