DLP-Lab5

ILE 310553043 You-Pin, Chen

May 12, 2022

1 Introduction

In this assignment, we are asked to implement a conditional VAE for video prediction. VAE[4] has been applied to many computer vision tasks such as super-resolution, compression. Specifically, your model should be able to do prediction based on past frames. For example, when we input frame x_{t-1} to the encoder, it will generate a latent vector h_{t-1} . Then, we will sample z_t from fixed prior. Eventually, 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 $\hat{x}_t$. Our goal is to over-achieve baseline PSNR score 25 and test different KL annealing mode for experiment. We had also done the learned prior as extra. The code is available in github[link].

2 Derivation of CVAE

A general techique for finding maximum likelihood solutions is in 2.0.1 and for probabilistic models having latent variables Z is in 2.0.2. In traditional VAE, we try to generate $x \in X$ with latent variables Z, the idea is to take a sample from $P(X)$. CVAE is the situation with condition which our goal is to find a sample from $P(X|c)$.

$$\theta^* = \arg\max_{\theta} \ p(X;\theta) \tag{2.0.1}$$

$$p(X;\theta) = \sum_{Z} \ p(X,Z;\theta) \tag{2.0.2}$$

In conditional probability, we know that the fraction of probability B that intersects with A:

$$P(A|B) = \frac{P(A \ \cap \ B)}{P(B)}$$

which in our cases is

$$\because \ P(Z|X) = \frac{P(Z \ \cap \ X)}{P(X)}$$

$$\therefore \ P(Z|X)P(X) = P(Z \ \cap \ X)$$

$$\therefore \ P(Z,X) = P(X)P(Z|X)$$

$$\therefore \ P(X,Z) = P(X)P(Z|X)$$

$$\therefore \ P(X,Z,c) = P(X,c)P(Z|X,c)$$

To see how the Expectation Maximization (EM) algorithm works, the chain rule of probability suggests in 2.0.3.

$$\begin{aligned} &\because\ p(X, Z, c; \theta) = p(X, c; \theta)\ p(Z|X, c; \theta) \\ &\therefore\ \log p(X, Z, c; \theta) = \log p(X, c; \theta) + \log p(Z|X, c; \theta) \\ &\therefore\ \log p(X|c; \theta) = \log p(X, Z|c; \theta) - \log p(Z|X, c; \theta) \end{aligned} \tag{2.0.3}$$

Problem is that we do not know what is $P(X|c)$ distribution look like, so we try to take a sample from Z from arbitrary distribution $q(Z|c)$.

$$\begin{aligned} L = &\ \log p(X|c; \theta) \\ = &\int q(Z|c) \log p(X|c; \theta)\ dZ \\ = &\int q(Z|c) \log p(X, Z|c; \theta)\ dZ - \int q(Z|c) \log p(Z|X, c; \theta)\ dZ \\ = &\int q(Z|c) \log p(X, Z|c; \theta)\ dZ - \int q(Z|c) \log q(Z|c)\ dZ + \\ &\int q(Z|c) \log q(Z|c)\ dZ - \int q(Z) \log p(Z|X, c; \theta)\ dZ \\ = &\ \mathcal{ELBO} + D_{KL}(q(Z|c)\ ||\ p(Z|X, c; \theta)) \end{aligned}$$

$$\begin{aligned} \mathcal{ELBO} = &\int q(Z|c) \log p(X, Z|c; \theta)\ dZ - \int q(Z|c) \log q(Z|c)\ dZ \\ = &\int q(Z|c) \log p(X|Z, c; \theta)\ dZ + \int q(Z|c) \log p(Z|c)\ dZ - \\ &\int q(Z|c) \log q(Z|c)\ dZ \\ = &\ E_{Z\sim q(Z|X,c)} \log p(X|Z, c; \theta) + E_{Z\sim q(Z|X,c)} \log p(Z|c) - \\ &\ E_{Z\sim q(Z|X,c)} \log q(Z|c) \\ = &\ E_{Z\sim q(Z|X,c)} \log p(X|Z, c; \theta) - D_{KL}(q(Z|X, c)\ ||\ p(Z|c)) \end{aligned}$$

Since D_{KL} is non-negative, $D_{KL}(q\ ||\ p) \geq 0$. It follows that $\log p(X; \theta) \geq \mathcal{L}(X, q, \theta)$ which $\mathcal{L}(X, q, \theta)$ is also called Evidence Lower Bound ($\mathcal{ELBO}$).

3 Implementation Details

3.1 Dataloader

For dataloader, We have two function: *get_seq* and *get_csv*. *get_seq* loads the image sequence which contains 30 frames and we reshaped all frames to $(1, 3, 64, 64)$. Then, we get condition by *get_csv* function. The number of condition is seven which includes four actions and three positions.

```
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_dir = args.data_root
        self.frame_num = 30
        if mode == 'train':
            self.data_dir = f'{self.root_dir}/train'
            self.ordered = False
        else:
            self.data_dir = f'{self.root_dir}/{mode}'
            self.ordered = True
        self.dirs = []
        for d1 in os.listdir(self.data_dir):
            for d2 in os.listdir(f'{self.data_dir}/{d1}'):
                self.dirs.append(f'{self.data_dir}/{d1}/{d2}')
        self.seq_len = args.n_past + args.n_future
        self.seed_is_set = False
        self.d = 0
        self.cur_dirs = ''
        self.transform = transform

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

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

    def get_seq(self):
        if self.ordered:
            self.cur_dirs = self.dirs[self.d]
            if self.d == len(self.dirs) - 1:
                self.d = 0
            else:
                self.d += 1
        else:
            self.cur_dirs = self.dirs[np.random.randint(len(self.dirs))]
        image_seq = []
        for i in range(self.frame_num):
            fname = f'{self.cur_dirs}/{i}.png'
            im = self.transform(Image.open(fname)).reshape((1, 3, 64, 64))
            image_seq.append(im)
        image_seq = torch.Tensor(np.concatenate(image_seq, axis=0))
        return image_seq

    def get_csv(self):
        cond_seq = []
        actions = [row for row in
            csv.reader(open(os.path.join(f'{self.cur_dirs}/actions.csv'), newline=''))]
        positions = [row for row in
            csv.reader(open(os.path.join(f'{self.cur_dirs}/endeffector_positions.csv'),
            newline=''))]
```

```
        for i in range(self.frame_num):
            concat = actions[i]
            concat.extend(positions[i])
            cond_seq.append(concat)
        cond_seq = torch.Tensor(np.array(cond_seq, dtype=float))
        return cond_seq

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

3.2 Model

Fig.1 illustrates the model structure in this assignment. We use VGG Net[5] as encoder and decoder while there is a LSTM[3] between them. We put x_{t-1} into encoder which $t-1$ means timestamp of video stream. The output h_{t-1} concatenate conditions which we got in dataloader and z_t. z_t is a variable from latent distribution. Then, frames are input to the LSTMs via a feedforward convolutional network, shared across all three parts of the model. A convolutional frame decoder maps the output of the frame predictor's recurrent network back to pixel space. KL cost annealing is implemented to minimize the cost between normal distribution and the distribution of timestamp t. We use teacher forcing mechanism to train the model by decay the ratio linearly.

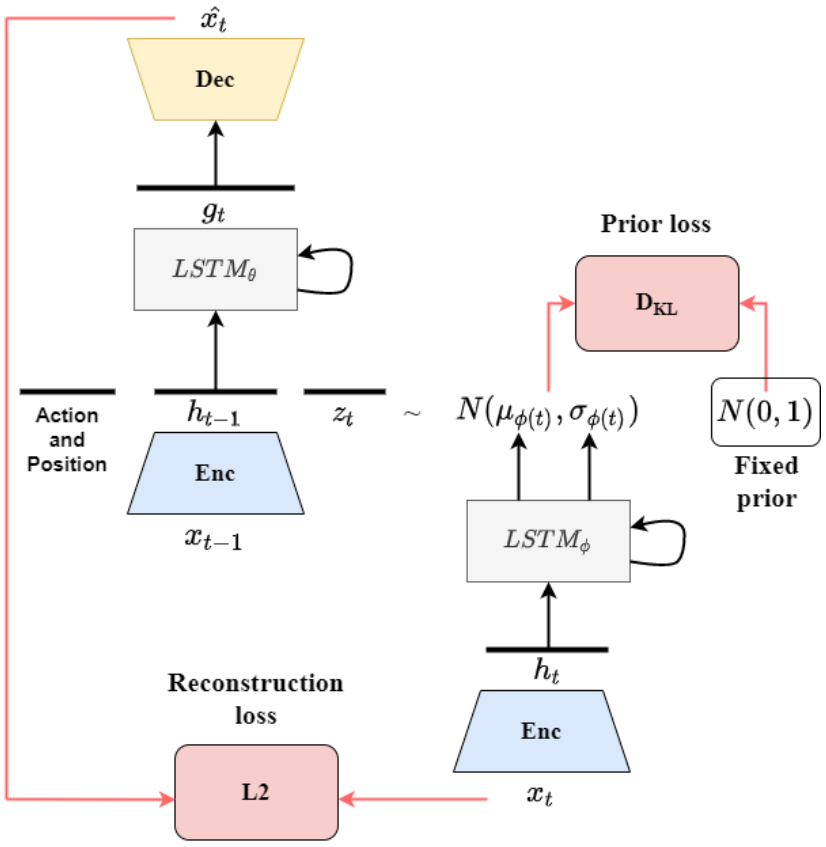


Figure 1: Model architecture.

3.2.1 Encoder and Decoder Implementation

```
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, nc=3):
        super(vgg_encoder, self).__init__()
        self.dim = dim
        # 64 x 64
        self.c1 = nn.Sequential(
                vgg_layer(nc, 64),
                vgg_layer(64, 64),
                )
        # 32 x 32
        self.c2 = nn.Sequential(
                vgg_layer(64, 128),
                vgg_layer(128, 128),
                )
        # 16 x 16
        self.c3 = nn.Sequential(
                vgg_layer(128, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 256),
                )
        # 8 x 8
        self.c4 = nn.Sequential(
                vgg_layer(256, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 512),
                )
        # 4 x 4
        self.c5 = nn.Sequential(
                nn.Conv2d(512, dim, 4, 1, 0),
                nn.BatchNorm2d(dim),
                nn.Tanh()
                )
        self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

    def forward(self, input):
        h1 = self.c1(input) # 64 -> 32
        h2 = self.c2(self.mp(h1)) # 32 -> 16
        h3 = self.c3(self.mp(h2)) # 16 -> 8
        h4 = self.c4(self.mp(h3)) # 8 -> 4
        h5 = self.c5(self.mp(h4)) # 4 -> 1
        return h5.view(-1, self.dim), [h1, h2, h3, h4]


class vgg_decoder(nn.Module):
    def __init__(self, dim, nc=3):
```

```
        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, nc, 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
```

3.2.2 LSTM Implementation

```
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)
```

3.3 Reparameterization Trick

Reparameterization Trick is a trick in order to backpropagate through a random node. Intuitively, if we were training a neural network to act as a VAE, then eventually we would need to perform backpropagation across a node in the network that was stochastic. But we cannot perform backpropagation across a stochastic node, because we cannot take the derivative of a random variable. So instead, we used something called the reparameterization trick, to restructure the problem by adding a constant value ϵ.

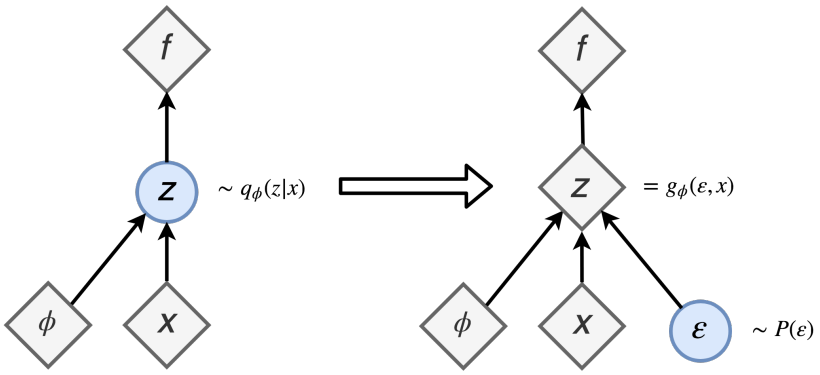


Figure 2: Illustration of the reparameterization trick.

```
def reparameterize(self, mu, logvar):
    try:
        logvar = logvar.mul(0.5).exp_()
        eps = Variable(logvar.data.new(logvar.size()).normal_())
        return eps.mul(logvar).add_(mu)
    except:
        raise NotImplementedError
```

3.4 KL Cost Annealing

For KL cost annealing, we tried two strategies that is Monotonic and Cyclical[2]. Fig.3 illustrates the different between them. The objective of VAE is learning an informative latent feature $z \sim q(z|x)$ to represent the observation x, while regularized towards the latent prior $p(z)$. However, when trained with a constant schedule $\beta = 1$, the KL term will lead to the KL vanishing issue. Cyclical schedule is introduced to dynamically change the value of β during training.

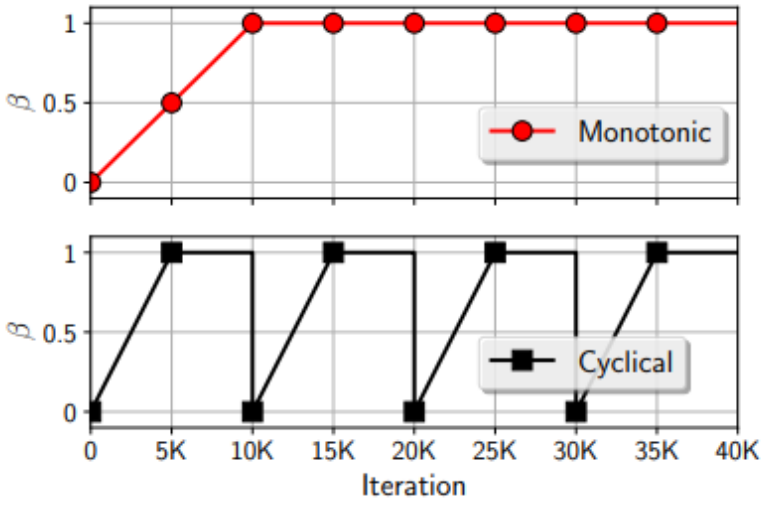


Figure 3: Two different strategies of KL cost annealing.

```
class kl_annealing():
    def __init__(self, args):
        super().__init__()
        self.n_iter = args.niter
        self.n_cycle = args.kl_anneal_cycle
        self.cyclical_mode = args.kl_anneal_cyclical
        self.i = 0
        if self.cyclical_mode:
            self.L = self.frange_cycle_linear(n_cycle=self.n_cycle)
        else:
            self.L = self.frange_cycle_linear(n_cycle=1, ratio=0.25)

    def frange_cycle_linear(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]
```

3.5 Setup

The hyperparameters setup for baseline are as follows:

– *niters*	100	– *learning rate*	$2e-3$
– *epoch*	600	– *optimizer*	*Adam*
– *batch size*	24	– *loss function*	$MSE + KL$

4 Results

4.1 The highest testing accuracy

The setup for our best result in fixed prior is using cyclical strategy. Teacher forcing ratio start by 1.0 and decay linearly from first epoch to the end. We achieved PSNR score 26.97 in inference while the baseline is 25. We also implemented learned prior approach which PSNR score got 28.44. Since the document can not shows GIF file, we uploaded the file in [here].

Figure 4: An illustration of prediction at each time step.

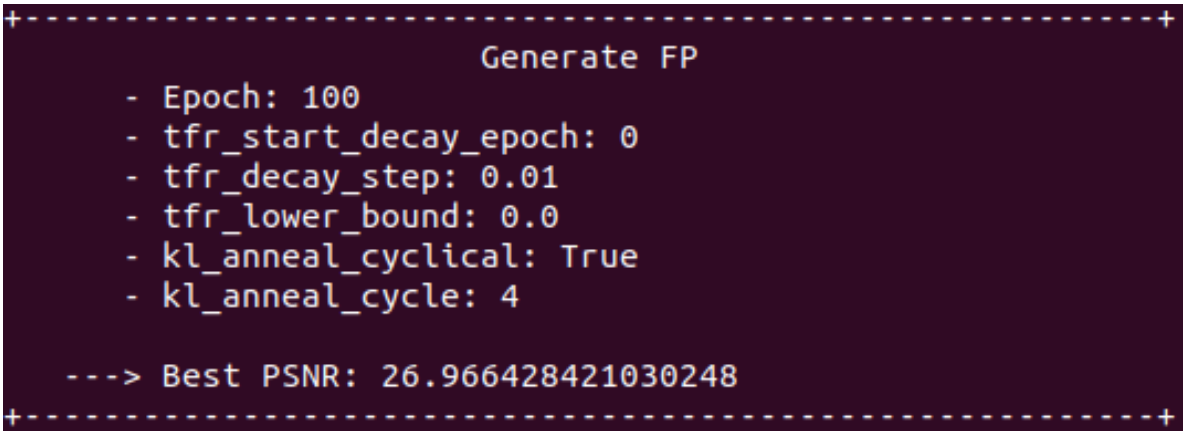
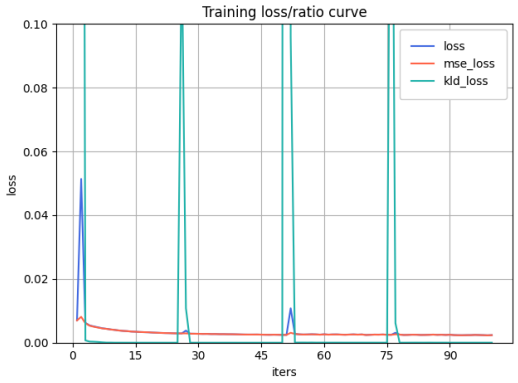


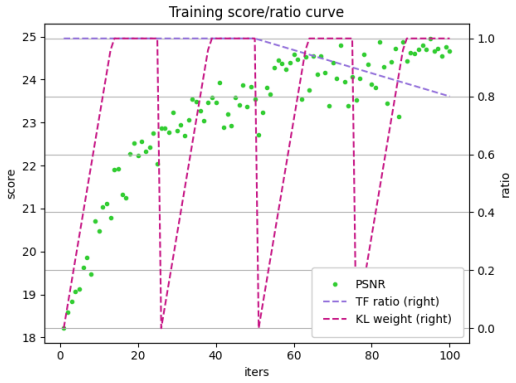
Figure 5: Screenshot for our best result in fixed prior.

4.2 Comparison figures

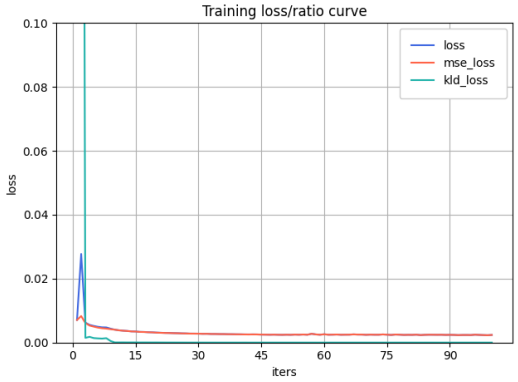
This section shows the figures in our experiments. We tried Monotonic/Cyclical with two teacher forcing strategies which is decay from first epoch and decay after 50 epochs respectively. $TF0$ is the first try in our experiments which means teacher forcing ratio decay after 50 epochs. After the end of training, TF ratio will stop at 0.8. $TF1$, instead, will decay from the start of the training and stop at 0 in the end while both ratios start from 1.0. In KL part, KL weight increase from first epoch in both strategies. β will get maximum value after 25 epochs in Monotonic and maintain till the end. However, Cyclical has 4 cycles with 2 ratios which means KL weight will restart from 0 in every 25 epochs. Since the first kld loss is extremely high, the loss curve can not compare clearly, we limited the $y-axis$ in loss curve in range $(0.0, 1.0)$. The details shows in below.



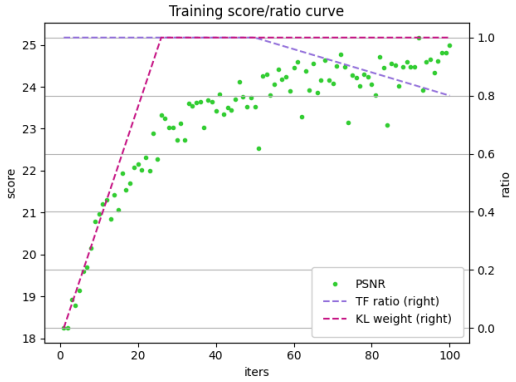
(a) Cyclical/TF0-Loss curve.



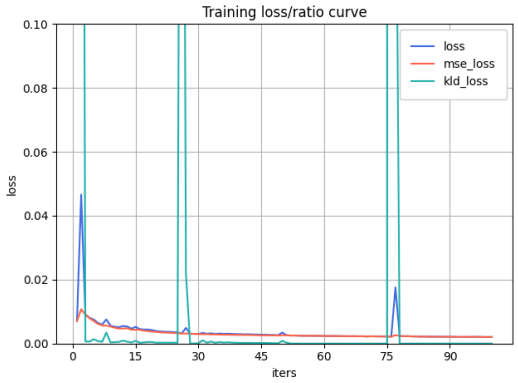
(b) Cyclical/TF0-Score/Ratio curve



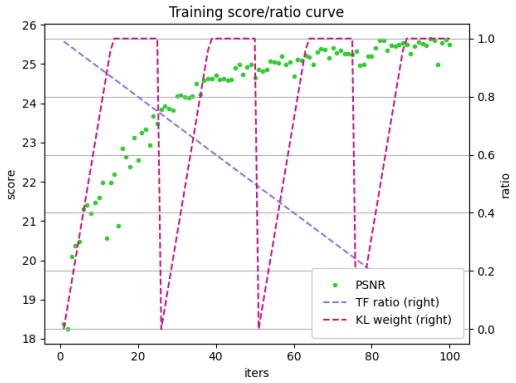
(c) Monotonic/TF0-Loss curve.



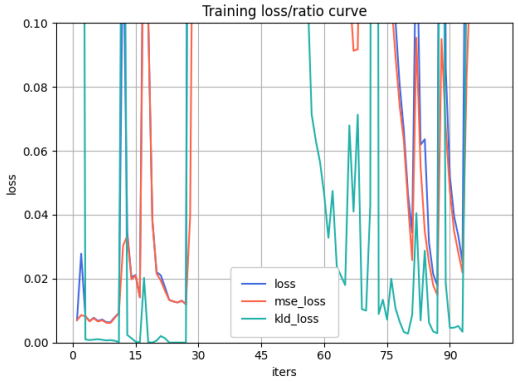
(d) Monotonic/TF0-Score/Ratio curve



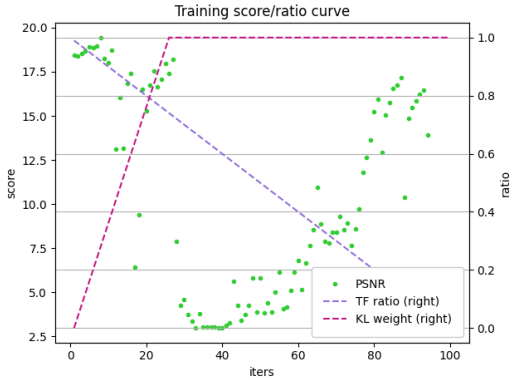
(e) Cyclical/TF1-Loss curve.



(f) Cyclical/TF1-Score/Ratio curve



(g) Monotonic/TF1-Loss curve.



(h) Monotonic/TF1-Score/Ratio curve

Figure 6: Comparison of different strategies in teacher forcing and KL annealing.

5 Discussion

5.1 KL Annealing

In this experiment, we tried to find out which KL weight strategies are more better. We observed that both of them are comparable in Fig.7. Since we changed the teacher forcing ratio strategy in Fig.8, the PSNR score in Monotonic extremely decreasing to around 3 or 4 while Cyclical scheme improved the accuracy. This situation demonstrates that Monotonic scheme is not stable. Monotonic scheme got maximum value in $25th$ epoch and with or without teacher forcing will lead this plan to different result. Instead, Cyclical scheme maintain better performance no matter what is the ratio of teacher forcing.

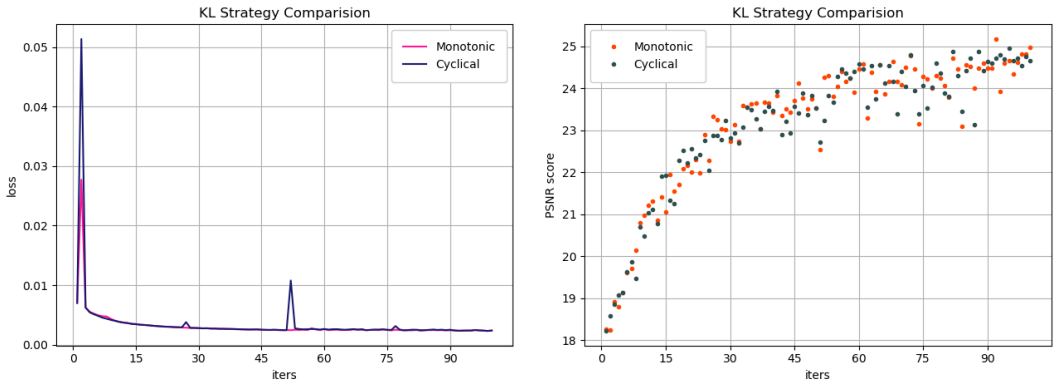


Figure 7: Results Comparison of different KL weight strategies with $TF0$.

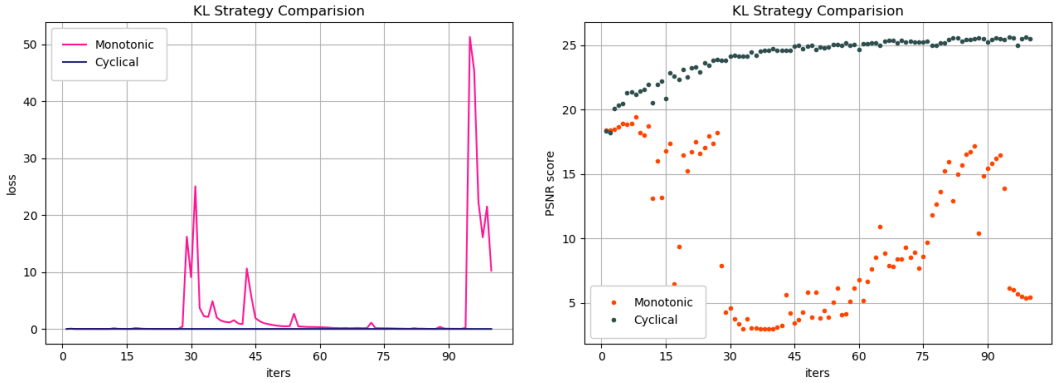


Figure 8: Results Comparison of different KL weight strategies with $TF1$.

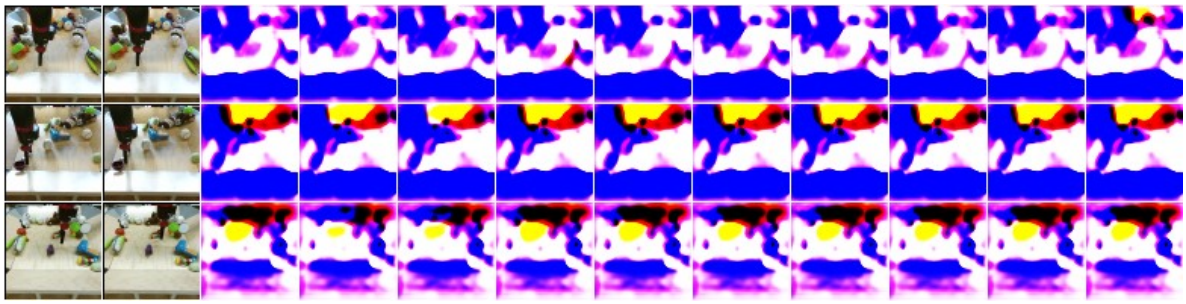


Figure 9: Some failures in Monotonic scheme with low TF ratio.

5.2 Teacher Forcing Ratio

In last section, we observed how crucial teacher forcing ratio is to whole training process, specially in Monotonic mode. Fig.10 illustrates different ratio strategies in Cyclical mode, as you can see, PSNR score improved extremely with decay from first epoch scheme comparing with decay after half of total epochs. We think the reason for this situation is overfitting problem. Because the propose of teacher forcing is that the model to use ground truth as input to predict next frame. However, if the training data always know the answer in each frame, the domain will overfit to designated video sequence. This problem also appear in [1]. And for Monotonic mode, since the KL weight will stop at $25th$ epoch, we decayed the ratio of giving the answer, the model did not learn anything. So in Fig.14 you can see that after $25th$ epoch, the PSNR score drop to $3 \sim 5$ for a while. And then PSNR score start to increase, it seems like the model had learned something after 60 epochs.

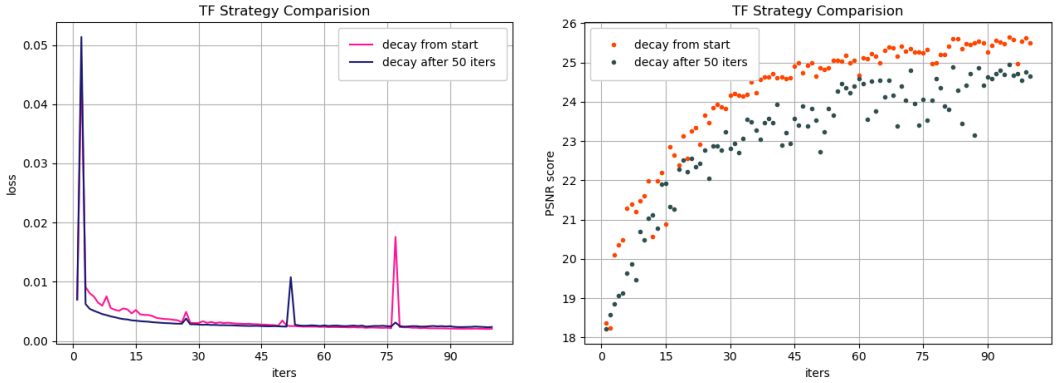


Figure 10: Results Comparison of different TF ratio strategies with Cyclical mode.

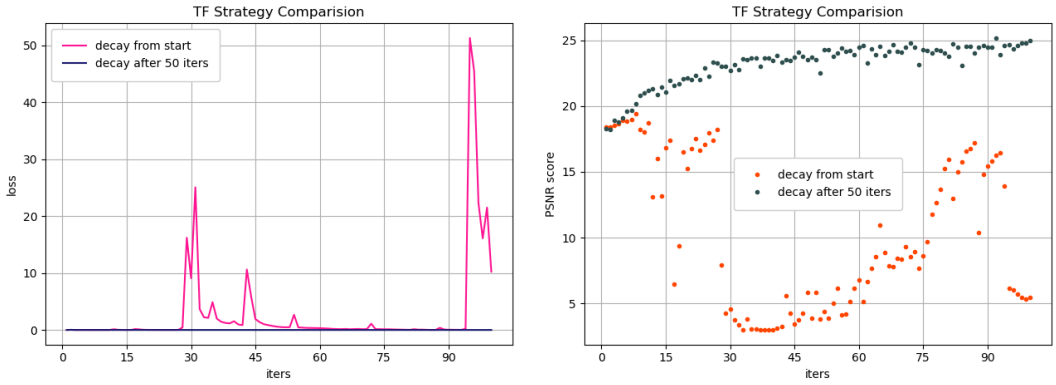


Figure 11: Results Comparison of different TF ratio strategies with Monotonic mode.

	TF Strategy	KL Strategy	Avg. PSNR score
Fixed Prior	Decay after 50 epochs	Monotonic	**26.87**
Fixed Prior	Decay after 50 epochs	Cyclical	**26.93**
Fixed Prior	Decay from start	Monotonic	23.04
Fixed Prior	Decay from start	Cyclical	**26.97**

Table 1: Testing with best weights in each experiment for 256 samples.

6 Extra

6.1 Learned Prior

Since in Sec.2, we had mentioned that we tried to take a sample from Z from arbitrary distribution. The simplest choice is a fixed Gaussian $N(0, 1)$, as is typically used in variational autoencoder. This approach we called $fixedprior$ which is also the main content in this assignment. A more sophisticated approach is to learn a prior that varies across time, being a function of all past frames up to but not including the frame being predicted. Specifically, at time t a prior network observes frames $t-1$ and output the parameters of a conditional Gaussian distribution $N(\mu(x1:t-1), \varphi(x1:t-1))$.

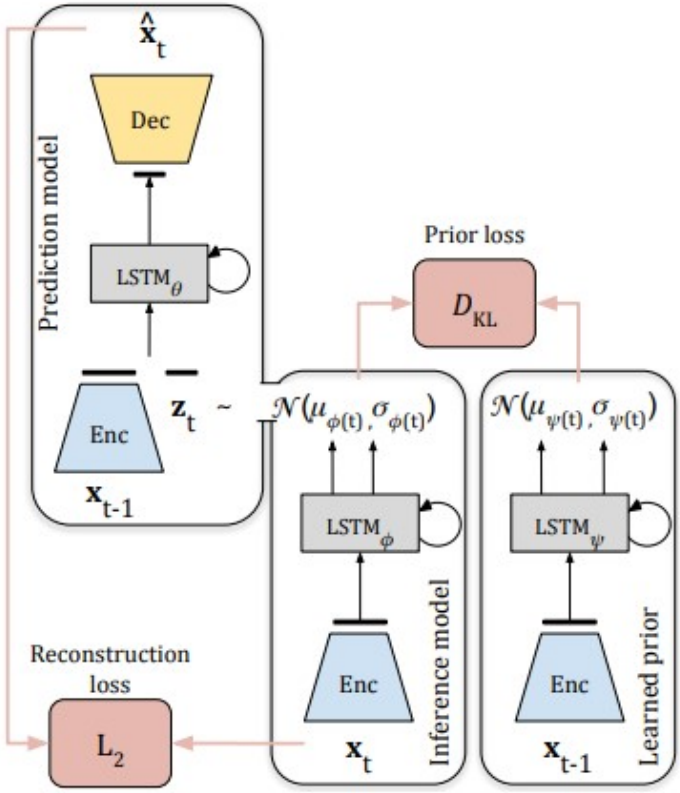


Figure 12: Training with learned prior ($SVG - LP$) in [1]

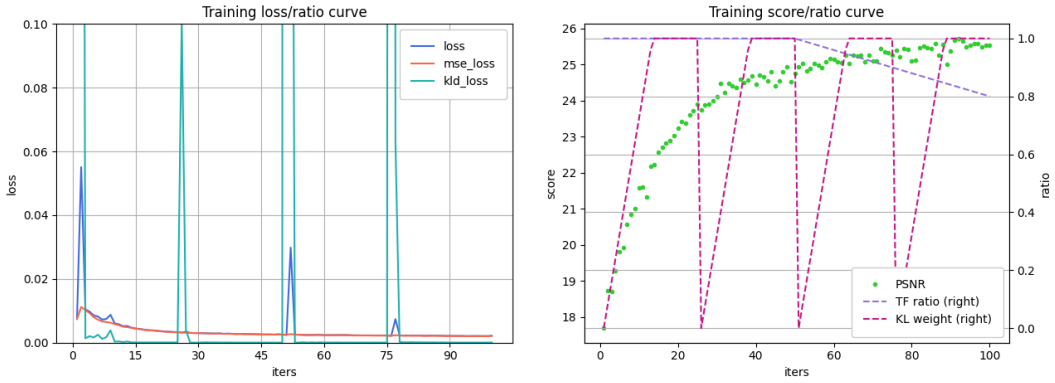


Figure 13: Results of learned prior with $TF0$ and Cyclical mode.

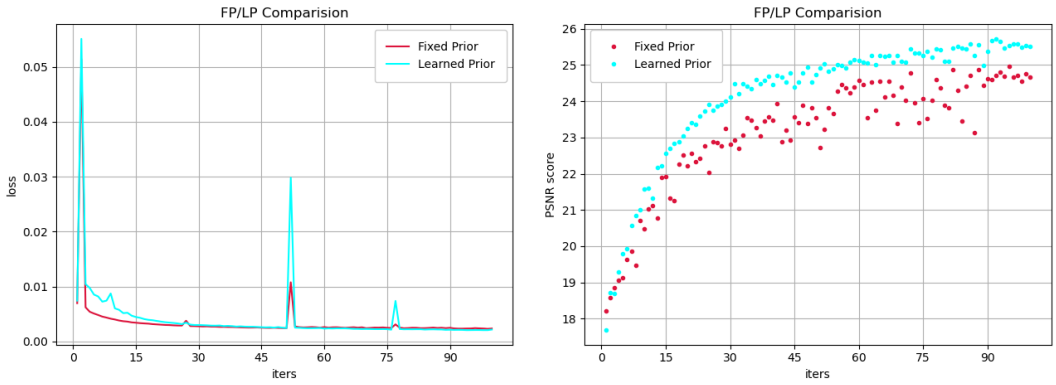


Figure 14: Training curve comparison with fixed prior and learned prior.

	TF Strategy	KL Strategy	Avg. PSNR score
Fixed Prior	Decay after 50 epochs	Cyclical	26.93
Learned Prior	Decay after 50 epochs	Cyclical	**28.44**

Table 2: Testing with best weights in each experiment for 256 samples.

```
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
```

References

[1] Emily Denton and Rob Fergus. Stochastic video generation with a learned prior. In *International conference on machine learning*, pages 1174–1183. PMLR, 2018.

[2] Hao Fu, Chunyuan Li, Xiaodong Liu, Jianfeng Gao, Asli Celikyilmaz, and Lawrence Carin. Cyclical annealing schedule: A simple approach to mitigating kl vanishing. *arXiv preprint arXiv:1903.10145*, 2019.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.