DLP Lab5 Report

▼ 資科工碩 穆冠蓁

Introduction

The objective of this experiment is to utilize a conditional VAE to predict videos by generating the next ten frames based on the preceding two frames. When we input frame x_{t-1} to the encoder, it will generate a latent code h_{t-1}. We will sample z_t from fixed prior. At last, we take the output from the encoder and z_t with the action and position as the input for the decoder an we expect that the output frame should be next frame $\hat{x}_t$.

- Dataset

  Dataset include more than 400000 sequence of robot pushing motions, and there is 30 frames, action and end effector position in each sequence.

Derivation of CVAE

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

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

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

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

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

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

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

∵ The KL divergence is non-negative, $KL(q\|p) \geq 0$, it follows that $\log p(x|c;\theta) \geq L(x,c,q,\theta)$, with $q(z|c) = p(z|x,c;\theta)$

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

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

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

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

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

Implementation details

1. Describe how you implement your model (encoder, decoder, reparameterization trick, dataloader)

   The sample code provided by TA has covered mostly the VAE, the part we should implement is to add the condition into the traditional VAE, and the reparameterization part.

   - VGG64 encoder

     VGG64 encoder will output latent code and 4 different scale feature map that will be use to decode predicted frame.

     ```
     import torch
     import torch.nn as nn

     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]
```

- LSTM

  LSTM is use as an encoder to tranform the embedding VGG64 has output to a latent vector. And it will be used to decode a predicted frame.

```
import torch
import torch.nn as nn
from torch.autograd import Variable

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):
        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
```

- VGG64 decoder

  We 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
```

- Reparameterization

  Repatameterize trick is to do the exponential on the new variance and generate a new vector called `eps` by normal distribution.Finally, we multiply `eps` and `logvar` together and plus the `mean` to get the hidden vector.

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

- Dataloader

  This part of code are provide by TA. In this lab, we only use 12 images of sequence to train our model and each image should be normalized between 0 and 1.

```
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.seq_len = args.n_past + args.n_future
        self.mode = mode
        if mode == 'train':
            self.ordered = False
        else:
            self.ordered = True

        self.transform = transform
        self.dirs = []

        for dir1 in os.listdir(self.root):

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

        self.seed_is_set = False
        self.idx = 0
        self.cur_dir = self.dirs[0]
        self.d=0

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

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

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

```
        image_seq = []
        for i in range(self.seq_len):
            fname = '{}/{}.png'.format(self.cur_dir, i)
            img = Image.open(fname)
            image_seq.append(self.transform(img))
        image_seq = torch.stack(image_seq)

        return image_seq

    def get_csv(self):
        with open('{}/actions.csv'.format(self.cur_dir), newline='') as csvfile:
            rows = csv.reader(csvfile)
            actions = []
            for i, row in enumerate(rows):
                if i == self.seq_len:
                    break
                action = [float(value) for value in row]
                actions.append(torch.tensor(action))

            actions = torch.stack(actions)

        with open('{}/endeffector_positions.csv'.format(self.cur_dir), newline='') as csvfile:
            rows = csv.reader(csvfile)
            positions = []
            for i, row in enumerate(rows):
                if i == self.seq_len:
                    break
                position = [float(value) for value in row]
                positions.append(torch.tensor(position))
            positions = torch.stack(positions)

        condition = torch.cat((actions, positions), axis=1)

        return condition

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

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

   - Implement

     I design my teacher forcing ration decay as below, I let the ratio decay linearly and it will decay to the lower bound in the last epoch.

     ```
     if epoch >= args.tfr_start_decay_epoch:
       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))
       print(f'Teacher ratio: {args.tfr}')
     ```

   - Main idea

     There is usually two mode to train a RNN:

     - Free running mode:Take the output of last state as input of this state.
     - Teacher forcing mode:Instead of using the output of the previous state as the input for the next state, the ground truth is directly used as the input for the next state.

     To solve the problem of free-running mode, such as, slow convergence and also the output of the previous step will seriously influence the other state, teacher forcing mode has been proposed.

   - Benefits/ Drawbacks

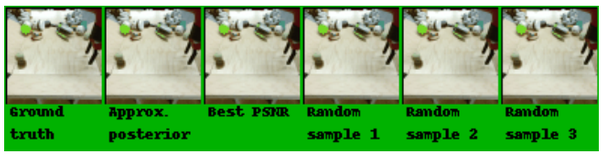
     - Benefits:
       - It is possible to correct the model’s prediction in order to prevent the amplification of errors during the sequence generation process.
       - It can greatly accelerate the convergence speed of the model, making the training process faster and more stable.
     - Drawbacks

- Exposure bias:This means there is inconsistency between the decoding behavior during training and inference, resulting in predictions inferred from different distributions during training and inference.Also this kind of model may have poor cross domain capability.
- It also lead to a problem called Overcorrect.

Results the discussion

- Show your result of video prediction
  - Make videos or gif images for test result(select one sequence)

    This is the gif of fixed_prior using the monotonic mode in KL annealing schedule.



  - Output the prediction at each time step

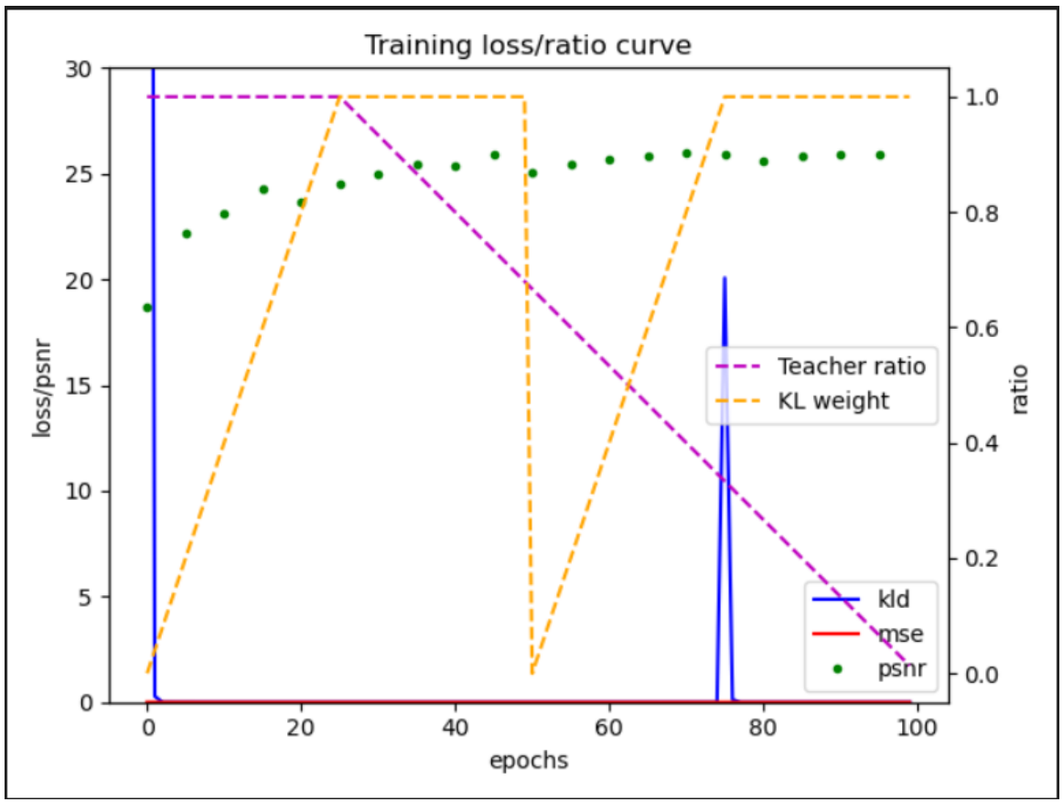
    1. Ground truth

    2. Predict

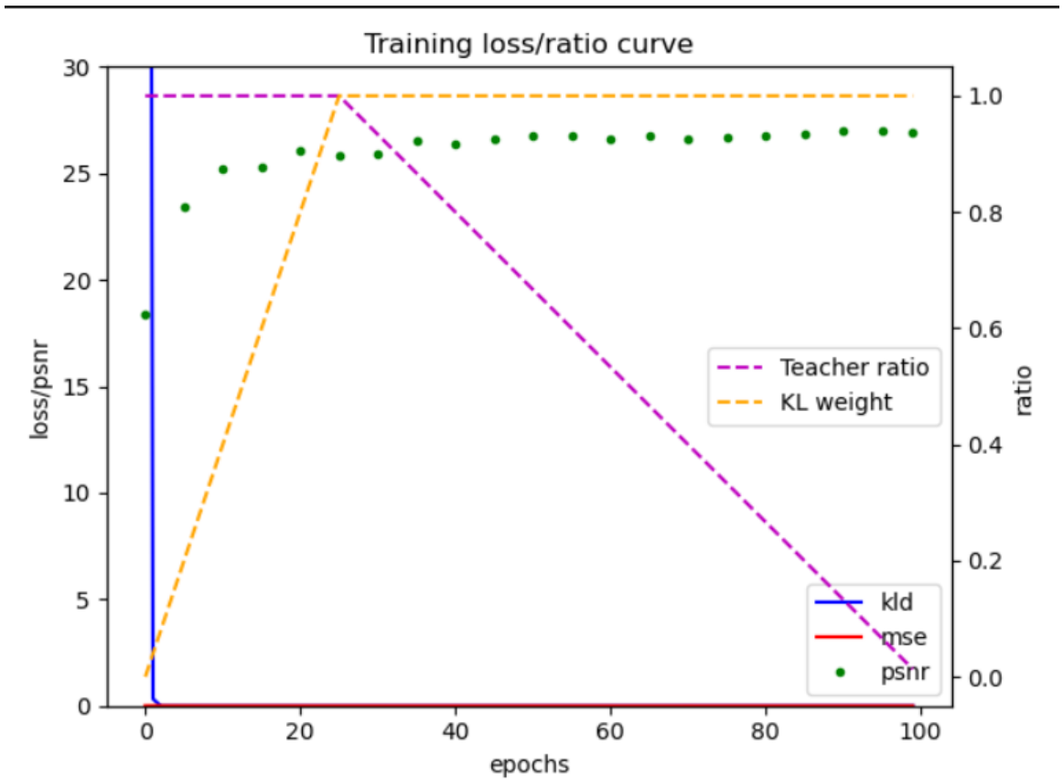
- Plot the KL loss and PSNR curves during training

	Cyclical	Monotonic
PSNR	25.46791	26.37529

    1. Cyclical



2. Monotonic



- Discuss the results according to your setting of teaching forcing ratio, KL weight, and learning rate. Note that this part mainly focuses on your discussion, if you simply just paste your results, you will get a low score.
  - Settings:
    - `batch_size`:24
    - `epoch`:300

- `tfr`:1.0
- `kl_anneal_ratio`:2
- `lr`:0.002

- Teaching forcing ratio

  Setting a higher teacher forcing rate at the beginning allows the model to avoid learning from errors as much as possible. Then gradually decreasing the rate enables more comprehensive learning. If only teacher forcing is used without transitioning to the original prediction method, there will not be significant progress in the later stages. Conversely, if only non-teacher forcing is used from the beginning, the initial results may not be as ideal.

- KL weight

  Using the Cyclic method for KL divergence can yield better PSNR results compared to the Monotonic method.

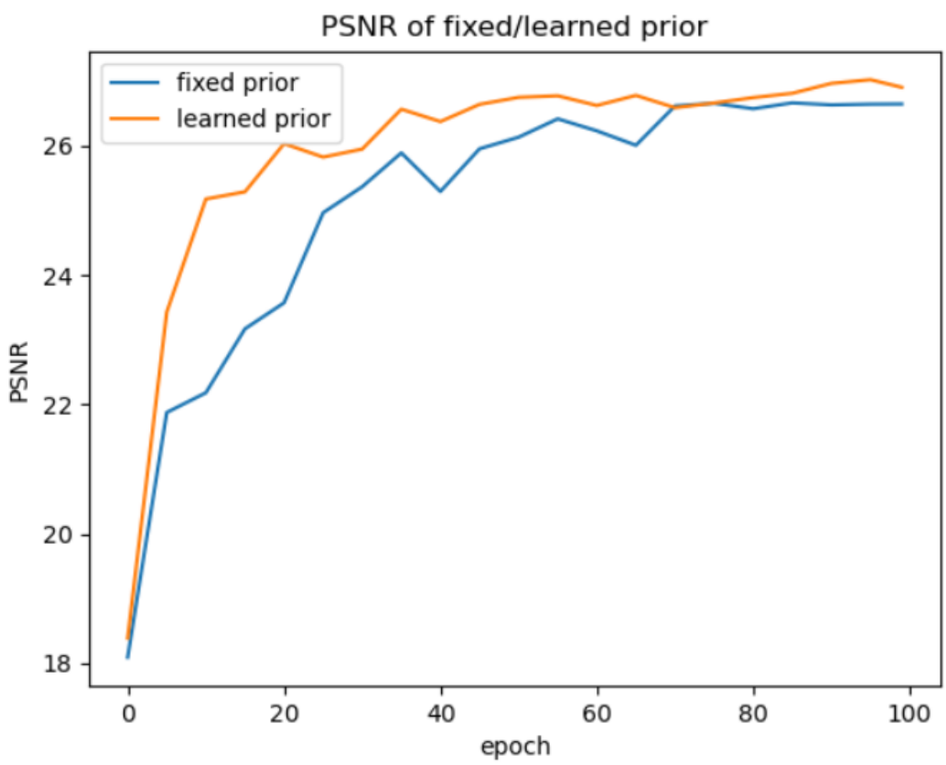
- Learning rate

  The learning rate has always been a crucial factor in training because if the learning rate is too low, the rate of change in the loss function becomes slower, leading to a higher risk of overfitting.

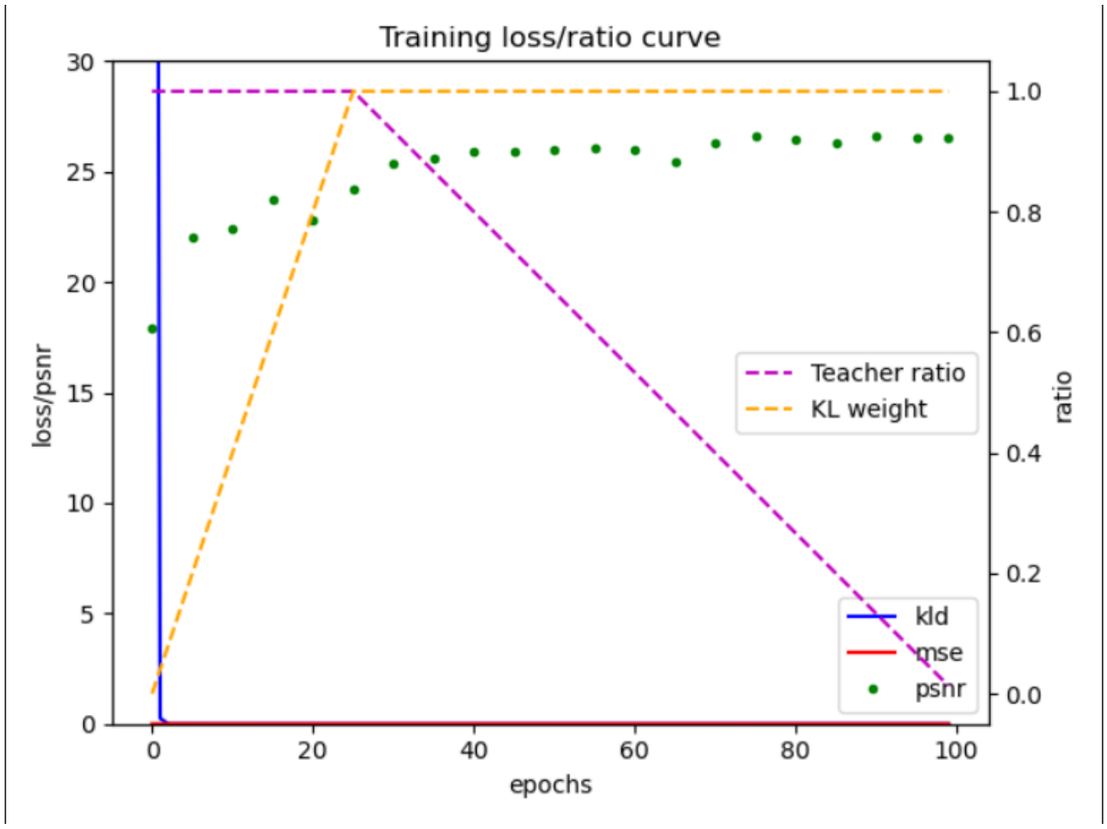
Extra

	Fixed prior	Learned prior
PSNR	26.37529	25.95334

1. PSNR of learned prior and fixed prior



2. PSNR of learned prior


Training loss/ratio curve
loss/psnr
ratio
epochs
Teacher ratio
KL weight
kld
mse
psnr
0
5
10
15
20
25
30
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100