Deep Learning and Practice Report
Lab5: Conditional VAE for Video Prediction

1 Introduction

Variational **A**uto**E**ncoder (**VAE**) is a generative model widely applied to lots of computer vision tasks. In this lab, we are asked to implement a **c**onditional **VAE** (**CVAE**) for video prediction. The model is trained on the bair robot pushing small dataset which contains roughly 44,000 examples of robot pushing motions. Action and end effector position at each time step are also provided. The model should take previous frame as input and generate its next following frame conditioning on the action and end effector position. Eventually, **PSNR** (**P**eak **S**ignal-to-**N**oise **R**atio) is adopted to evaluate the generated results.

2 Derivation of CVAE

For training CVAE, we introduce a new variable c denoting the condition. The idea is to train a model with parameter $\boldsymbol{\theta}$ to learn the conditional distribution $p(\boldsymbol{X}|c;\boldsymbol{\theta})$.

$$p(\boldsymbol{X}|c;\boldsymbol{\theta}) = \int p(\boldsymbol{X}|\boldsymbol{Z},c;\boldsymbol{\theta})p(\boldsymbol{Z}|c)d\boldsymbol{Z} \tag{1}$$

Same as VAE, integration over $\boldsymbol{Z}$ is intractable in general when $p(\boldsymbol{X}|\boldsymbol{Z},c;\boldsymbol{\theta})$ is modeled by a neural network. To circumvent this difficulty, we recall the derivation from lecture 13, page 24.

$$\log p(\boldsymbol{X}|c;\boldsymbol{\theta}) = \log p(\boldsymbol{X},\boldsymbol{Z}|c;\boldsymbol{\theta}) - \log p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta}) \tag{2}$$

By introducing an arbitrary distribution $q(\boldsymbol{Z}|c)$ on both sides and integrate over $\boldsymbol{Z}$,

$$\begin{aligned}
\log p(\boldsymbol{X}|c;\boldsymbol{\theta}) &= \int q(\boldsymbol{Z}|c)\log p(\boldsymbol{X}|c;\boldsymbol{\theta})d\boldsymbol{Z} \\
&= \int q(\boldsymbol{Z}|c)\log p(\boldsymbol{X},\boldsymbol{Z}|c;\boldsymbol{\theta})d\boldsymbol{Z} - \int q(\boldsymbol{Z}|c)\log p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta})d\boldsymbol{Z} \\
&= \int q(\boldsymbol{Z}|c)\log p(\boldsymbol{X},\boldsymbol{Z}|c;\boldsymbol{\theta})d\boldsymbol{Z} - \int q(\boldsymbol{Z}|c)\log q(\boldsymbol{Z}|c)d\boldsymbol{Z} \\
&\quad + \int q(\boldsymbol{Z}|c)\log q(\boldsymbol{Z}|c)d\boldsymbol{Z} - \int q(\boldsymbol{Z}|c)\log p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta})d\boldsymbol{Z} \\
&= \mathcal{L}(\boldsymbol{X},q,\boldsymbol{\theta}|c) + \mathrm{KL}(q(\boldsymbol{Z}|c) \parallel p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta}))
\end{aligned} \tag{3}$$

where $\mathcal{L}(\boldsymbol{X},q,\boldsymbol{\theta}|c)$ is the **e**vidence **l**ower **bo**und (**ELBO**), and $\mathrm{KL}(q(\boldsymbol{Z}|c) \parallel p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta}))$ denotes the KL divergence between $q(\boldsymbol{Z}|c)$ and $p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta})$. Both terms are list as follows.

$$\mathcal{L}(\boldsymbol{X},q,\boldsymbol{\theta}|c) = \int q(\boldsymbol{Z}|c)\log p(\boldsymbol{X},\boldsymbol{Z}|c;\boldsymbol{\theta})d\boldsymbol{Z} - \int q(\boldsymbol{Z}|c)\log q(\boldsymbol{Z}|c)d\boldsymbol{Z} \tag{4}$$

$$\mathrm{KL}(q(\boldsymbol{Z}|c) \parallel p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta})) = \int q(\boldsymbol{Z}|c)\log\frac{q(\boldsymbol{Z}|c)}{p(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta})}d\boldsymbol{Z} \tag{5}$$

A rearrangement of equation 3 gives,

$$\mathcal{L}(\boldsymbol{X}, q, \boldsymbol{\theta}|c) = \log p(\boldsymbol{X}|c;\boldsymbol{\theta}) - \mathrm{KL}(q(\boldsymbol{Z}|c) \parallel p(\boldsymbol{Z}|\boldsymbol{X}, c;\boldsymbol{\theta})) \tag{6}$$

Since the equality holds for any choice of $q(\boldsymbol{Z}|c)$, we can also introduce a distribution $q(\boldsymbol{Z}|\boldsymbol{X}, c, \boldsymbol{\theta'})$ modeled by another neural network (the encoder) with parameter θ'.

$$\begin{aligned}\mathcal{L}(\boldsymbol{X}, q, \boldsymbol{\theta}|c) &= \log p(\boldsymbol{X}|c;\boldsymbol{\theta}) - \mathrm{KL}(q(\boldsymbol{Z}|\boldsymbol{X}, c;\boldsymbol{\theta'}) \parallel p(\boldsymbol{Z}|\boldsymbol{X}, c;\boldsymbol{\theta'})) \\ &= \mathbb{E}_{\boldsymbol{Z}\sim q(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta'})}[\log p(\boldsymbol{X}|\boldsymbol{Z}, c;\boldsymbol{\theta}) + \log p(\boldsymbol{Z}|c) - \log q(\boldsymbol{Z}|\boldsymbol{X}, c;\boldsymbol{\theta'})] \\ &= \mathbb{E}_{\boldsymbol{Z}\sim q(\boldsymbol{Z}|\boldsymbol{X},c;\boldsymbol{\theta'})}[\log p(\boldsymbol{X}|\boldsymbol{Z}, c;\boldsymbol{\theta})] - \mathrm{KL}(q(\boldsymbol{Z}|\boldsymbol{X}, c;\boldsymbol{\theta'}) \parallel p(\boldsymbol{Z}|c))\end{aligned} \tag{7}$$

Hence, the evidence lower bound for conditional VAE is derived as shown above. We can see that all distributions just condition on a variable c. The first term is the likelihood of reconstructing the input image X and the second one is the regularization term. Specifically, we attempt to maximize the evidence lower bound. This is equivalent to minimizing the reconstruction error and the KL divergence between encoded distribution $q(\boldsymbol{Z}|\boldsymbol{X}, c;\boldsymbol{\theta'})$ and prior distribution $p(\boldsymbol{Z}|c))$.

3 Implementation Details

3.1 Describe how you implement your model

3.1.1 DataLoader

```python
class bair_robot_pushing_dataset(dataset):
    def __init__(self, args, mode="train", transform=default_transform):
        super(bair_robot_pushing_dataset, self).__init__(args, mode, transform)

        self.args = args
        self.mode = mode
        self.transforms = transform

        ## Get all data paths
        self.data_dirs = []
        self.mode_data_root = "{}/{}".format(self.args.data_root, self.mode)
        for data_dir in os.listdir(self.mode_data_root):
            if data_dir[0] == ".":
                continue
            idx_dirs = os.listdir("{}/{}".format(self.mode_data_root, data_dir))
            for idx_dir in idx_dirs:
                if idx_dir[0] == ".":
                    continue
                self.data_dirs.append("{}/{}/{}".format(self.mode_data_root, data_dir, idx_dir))

        self.seed_is_set = True ## Whether the random seed is already set or not

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

    def get_seq(self, index):
        """Get the sequence of frames"""
        seqs = []
        frame_files = [file for file in os.listdir(self.data_dirs[index]) if ".png" in file and file[0] != "."]
        frame_files.sort(key=lambda x: int(x.split(".")[0]))
        for frame_file in frame_files:
            img_arr = Image.open("{}/{}".format(self.data_dirs[index], frame_file))
            img_tensor = self.transforms(img_arr)
            seqs.append(img_tensor)

        ## Transform to tensor of shape (30, 3, 64, 64)
        seqs = torch.stack(seqs)
        return seqs
```

Figure 1: First part of my dataloader.

```
    def get_csv(self, index):
        """Get the actions and positions as conditions"""
        conds = []
        files = [
            "{}/actions.csv".format(self.data_dirs[index]),
            "{}/endeffector_positions.csv".format(self.data_dirs[index])
        ]
        for csv_file in files:
            with open(csv_file) as f_csv:
                rows = csv.reader(f_csv)
                cond_arr = np.array(list(rows)).astype(float)
                cond_tensor = self.transforms(cond_arr)
            conds.append(cond_tensor)

        ## Concatenate 2 conditions, transform to tensor of shape (30, 7)
        conds = torch.cat(conds, dim=2)[0]
        conds = conds.float()
        return conds

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

Figure 2: Second part of my dataloader.

Figure 1 and 2 show the implementation of my dataloader. The dataset object first retrieves the paths of all sequences in the function `__init__`. The function `get_seq` is used to read in a sequence of frames and transform them into the shape of (number of frames, 3, 64, 64). Similarly, `get_csv` reads in the conditions (actions & end effector positions) of a corresponding sequence and transform them into the shape of (number of frames, 7). When iterating through each batch, `__getitem__` is automatically called and it returns a batch of sequences and conditions.

3.1.2 Reparameterization Trick

The reparameterization trick is computed as follows,

$$\boldsymbol{Z} = \mu(\boldsymbol{X}) + \Sigma(\boldsymbol{X}) * \epsilon \tag{8}$$

where $\mu(\boldsymbol{X})$ and $\Sigma(\boldsymbol{X})$ denotes the generated mean and standard deviation, and $\epsilon \sim \mathcal{N}(0, \boldsymbol{I})$.

```
def reparameterize(self, mu, logvar):
    #raise NotImplementedError
    std = torch.exp(logvar / 2) ## log(variance) = log(std^2)
    eps = torch.randn_like(std) ## N(0, I) with same shape as std
    return mu + eps * std
```

Figure 3: Code of reparameterization trick.

Figure 3 shows my implementation of reparameterization. It is worthy to note that the we let the model learn to generate log variance instead of variance. By definition, the variance has to be a positive real number, which limits the range of model's prediction. By applying log transform, we map the range of the model from $[0, 1]$ to $[-\infty, \log(1)]$, which is more stable.

3.1.3 Training Procedure

Figure 4 shows the code for training a batch of sequences. I first encode the first 12 frames by the VGG encoder. Next, the model generates the frame of each time step in a for loop. Specifically, in each iteration, the model first obtain the encoded representation of last frame h_{t-1}. Next, the encoded

```
## Encoder all frames first
h_seq = [self.modules["encoder"](x[i]) for i in range(self.args.n_past + self.args.n_future)]

## Iterate through first 12 frames
for i in range(1, self.args.n_past + self.args.n_future):
    ## Get encoded information
    h_t, _ = h_seq[i]

    if self.args.last_frame_skip or (i < self.args.n_past):
        h_in, skip = h_seq[i - 1]
    else:
        if use_teacher_forcing:
            h_in, _    = h_seq[i - 1]
        else:
            h_in, _    = self.modules["encoder"](x_pred)

    ## Obtain latent vector z
    z_t, mu, logvar = self.modules["posterior"](h_t)

    ## Decode the image
    lstm_in = torch.cat([h_in, z_t, cond[i - 1]], dim=1)
    g_t     = self.modules["frame_predictor"](lstm_in)
    x_pred  = self.modules["decoder"]([g_t, skip])

    ## Calculate loss values
    mse = mse + mse_criterion(x[i], x_pred)
    kld = kld + kl_criterion(mu, logvar, self.args)
```

Figure 4: Code of training procedure.

representation of current frame h_t is passed through an LSTM cell to obtain the latent vector z_t. Then, another LSTM cell takes h_{t-1}, z_t, and the condition at last time step c_{t-1} as input, and the resulting output is then fed to the VGG decoder to obtain prediction for current step $\hat{x}_t$.

3.1.4 Testing procedure

```
## Iterate through 12 frames
for frame_idx in range(1, args.n_past + args.n_future):
    ## Encode the image at step (t-1)
    if args.last_frame_skip or frame_idx < args.n_past:
        h_in, skip = modules["encoder"](x_in)
    else:
        h_in, _    = modules["encoder"](x_in)

    ## Obtain the latent vector z at step (t)
    if frame_idx < args.n_past:
        h_t, _    = modules["encoder"](validate_seq[frame_idx])
        #z_t, _, _ = modules["posterior"](h_t)
        _, z_t, _ = modules["posterior"](h_t) ## Take the mean
    else:
        z_t = torch.FloatTensor(args.batch_size, args.z_dim).normal_().to(device)

    ## Decode the image based on h_in & z_t
    if frame_idx < args.n_past:
        modules["frame_predictor"](torch.cat([h_in, z_t, cond[frame_idx - 1]], dim=1))
        x_in = validate_seq[frame_idx]
    else:
        g_t  = modules["frame_predictor"](torch.cat([h_in, z_t, cond[frame_idx - 1]], dim=1))
        x_in = modules["decoder"]([g_t, skip])
```

Figure 5: Code of testing procedure.

Figure 5 shows the code of my main testing loop. In each iteration, the model first encode the frame at last time step to get h_{t-1}. Next, the latent code z_t is generated from the encoder and Gaussian LSTM if we are processing the first 2 frames. Otherwise, it is sampled from $\mathcal{N}(0, \boldsymbol{I})$. In other words, teacher forcing ratio is set to 0 during whole testing procedure. Finally, the frame predictor (another LSTM) takes h_{t-1}, z_t and the conditions from last time step c_{t-1} as input, and it generates a vector g_t. The decoder then outputs the prediction of current step $\hat{x}_t$ based on g_t.

3.2 Describe the teacher forcing strategy

Teacher forcing is a training strategy commonly used in auto-regressive generative models. When training models on sequential data such as time series or sentences, an intuitive way is to take the output of step $t-1$ as the input of step t. By adopting teacher forcing, we instead feed the ground-truth of step $t-1$ to the model at step t. With this approach, models often converge in less training steps. It somehow mitigates the problem of error propagation since the model may repeatedly take wrong output from previous step as the input of next step without teacher forcing, thus resulting to low convergence speed. However, the model may lose some generalizability since it relies on labeled data too much. That is, poor performance could probably occur on unseen data if this approach is fully applied during training.

4 Results and Discussion

4.1 Show your results of video prediction

(a) Monotonic KL Annealing

(b) Cyclical KL Annealing

Figure 6: Model prediction at each time step.

GIF files of my test results are placed in the folder "result", and the model prediction at each time step is shown in figure 6. The upper one is predicted by model trained with monotonic KL annealing, and the lower one is by model trained with cyclical KL annealing.

4.2 Plot the losses, average PSNR and ratios

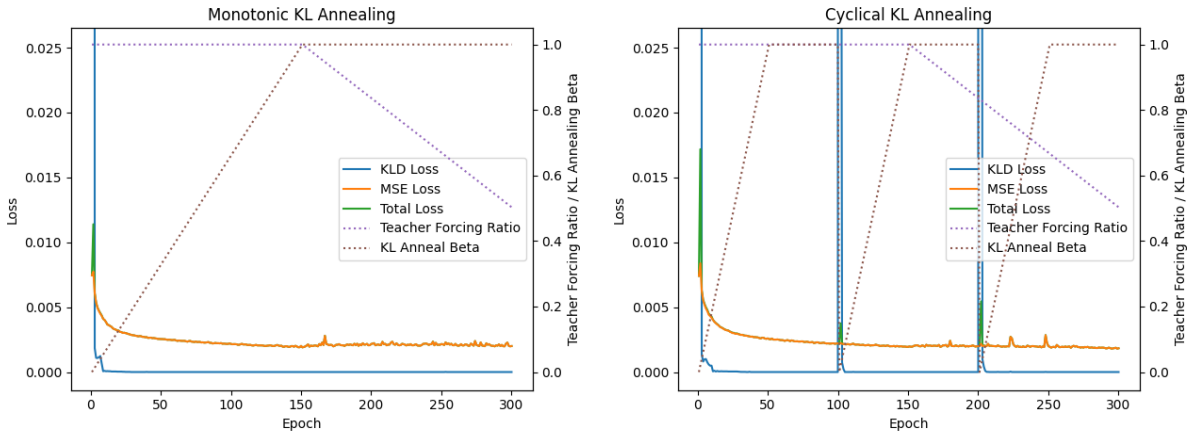

Figure 7: Loss & ratio curves.

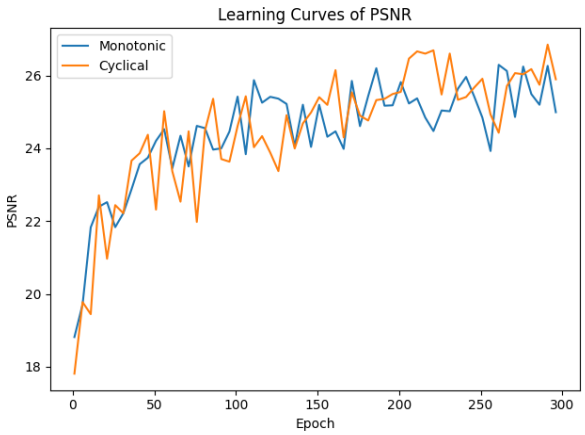

Figure 8: PSNR curves.

Figure 7 and 8 demonstrates the learning curves of losses, teacher forcing ratio, KL annealing β and average PSNR. I plot the PSNR curves separately since its scale with other values don't match well.

4.3 Discuss the results according to your settings

KL Annealing	TFR Decay Epoch	Validation PSNR	Test PSNR
Monotonic	150 / 300	25.6865	25.2948
Cyclical	150 / 300	**26.1893**	25.7681

Table 1: PSNR with different KL annealing strategies.

Table 1 demonstrates the PSNR of models trained with different KL annealing strategies. Scores on both validation set and test set are reported. With cyclical KL annealing, the model achieves higher scores on both sets. Actually, both models have comparable results and they have reached over 25 on PSNR.
On the other hand, two training strategies behave differently on the variation of loss values. We first recall why KL annealing is applied to this task. The motivation is that training VAEs in an auto-regressive manner often results in the KL vanishing problem. A possible consequence is that the input $\boldsymbol{X}$ becomes independent to the latent vector $\boldsymbol{Z}$, making the decoder no longer relies on $\boldsymbol{Z}$ to make predictions. We can also conjecture that the encoded distribution cannot represent the true data distribution accurately. Therefore, setting the weight of KL term to 0 at first aims to encourage the model to encode more information of $\boldsymbol{X}$ into the latent variable $\boldsymbol{Z}$. The KL weight is then increased as the training step grows. From figure 7 we can see that the KL divergence is extremely large at first few epochs, it then reduces dramatically to almost 0. As the MSE loss keeps decreasing, the PSNR score starts to get higher. The main difference of using cyclical KL annealing is that after setting the KL weight back to zero, the KL divergence suddenly boosts up again. Similar phenomenon occurs in the beginning of every cycle.
Another observation is that dropping the teacher forcing ratio in the beginning hurts the model's performance. I guess that the model hasn't gained enough ability to reconstruct the image. In this way, the model keeps on using low quality prediction from previous time step as the input of current time step, thus making the training process less stable.