1. Introduction

In this lab, I implement the conditional VAE with LSTM by pytorch on python. And implement some training skill like KL annealing for VAE and teacher forcing for LSTM. Then apply this model on BAIR dataset to predict future images and adopt PSNR to evaluate the perceptual quality. Finally I get PSNR = 23.5 in the test set.

2. Derivation of CVAE

$$\log P(X \mid c;\theta) = \log p(x, z \mid c;\theta) - \log p(z \mid x, c;\theta)$$

introduce an arbitrary distribution $q(z|c)$ on both side and integrate over Z

$$\int q(z|c) \log P(x \mid c;\theta)\, dZ$$

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

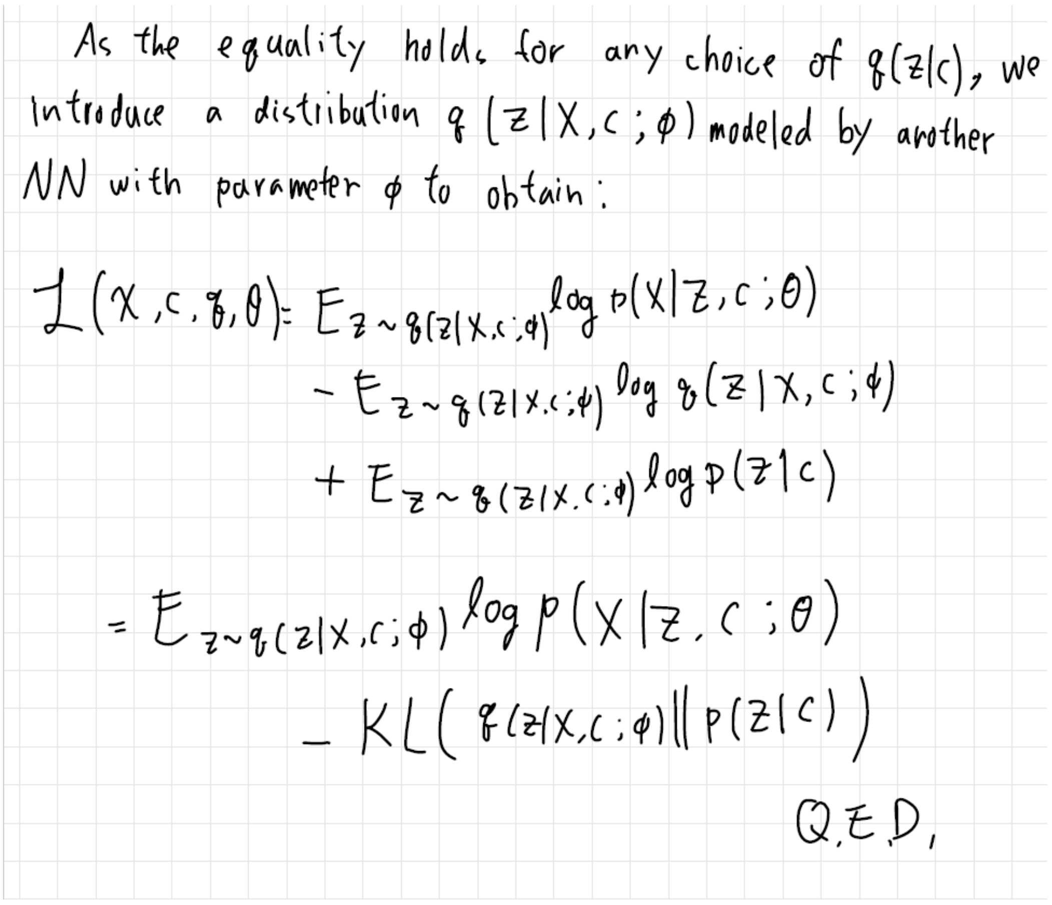
$$= \int q(z|c) \log p(x, z \mid c;\theta)\, dz - \int q(z|c) \log q(z|c)\, dZ + \int q(z|c) \log q(z|c)\, dz - \int q(z|c) \log p(Z \mid x, c;\theta)\, dZ$$

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

$$= \mathcal{L}(X, c, q, \theta) + KL\left(q(z|c) \,\|\, p(z \mid x, c;\theta)\right)$$

A rearrangement gives:

$$\mathcal{L}(x, c, q, \theta) = \int q(z) \log p(x \mid c;\theta)\, dz - KL\left(q(z|c) \,\|\, p(z \mid x, c;\theta)\right)$$



As the equality holds for any choice of $q(z|c)$, we introduce a distribution $q(z|X,c;\phi)$ modeled by another NN with parameter ϕ to obtain:

$$\begin{aligned}\mathcal{L}(X,c,q,\theta) = \; & E_{z\sim q(z|X,c;\phi)} \log p(X|z,c;\theta) \\ & - E_{z\sim q(z|X,c;\phi)} \log q(z|X,c;\phi) \\ & + E_{z\sim q(z|X,c;\phi)} \log p(z|c) \\ = \; & E_{z\sim q(z|X,c;\phi)} \log p(X|z,c;\theta) \\ & - KL(q(z|X,c;\phi) \| p(z|c))\end{aligned}$$

Q.E.D.

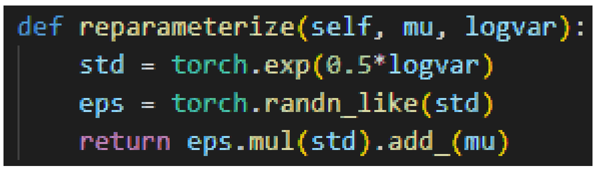
3. Implementation details

- **Describe how you implement your model**

Encoder & Decoder:

No change, just directly use the vgg net in models folder.

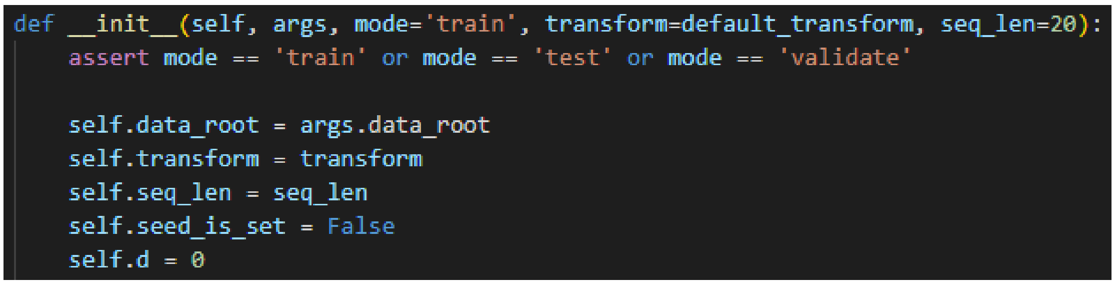
Reparameterization trick:



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

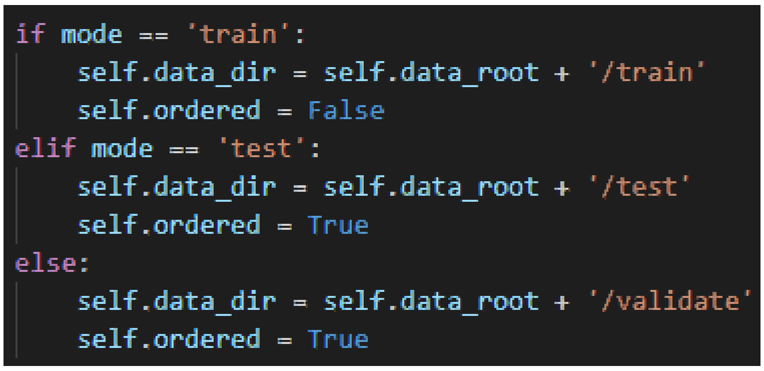
Following the tutorial in pytorch document. Calculate the standard derivation = e^(0.5*logvar) first. Then sample in N(0,1) and multiplied by the standard derivation and add mean value.

Dataloader:



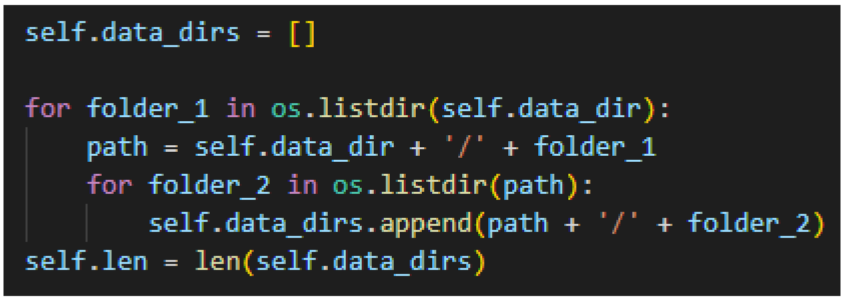
```python
def __init__(self, args, mode='train', transform=default_transform, seq_len=20):
    assert mode == 'train' or mode == 'test' or mode == 'validate'

    self.data_root = args.data_root
    self.transform = transform
    self.seq_len = seq_len
    self.seed_is_set = False
    self.d = 0
```

In the constructor, I add one more parameter call seq_len, indicate how long data should the __getitem__ return. And save necessary data in the class.



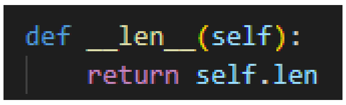
```python
if mode == 'train':
    self.data_dir = self.data_root + '/train'
    self.ordered = False
elif mode == 'test':
    self.data_dir = self.data_root + '/test'
    self.ordered = True
else:
    self.data_dir = self.data_root + '/validate'
    self.ordered = True
```

Calculate the data directory according to its mode. And for training set, I would make it to pick random item when calling __getitem__.



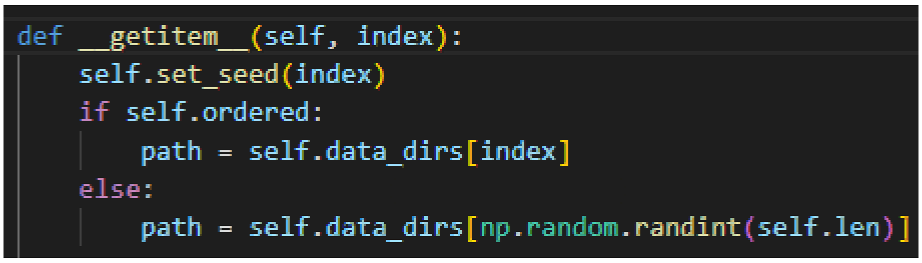
```python
self.data_dirs = []

for folder_1 in os.listdir(self.data_dir):
    path = self.data_dir + '/' + folder_1
    for folder_2 in os.listdir(path):
        self.data_dirs.append(path + '/' + folder_2)
self.len = len(self.data_dirs)
```

Then, I use a list to save all the image sequences' folders.



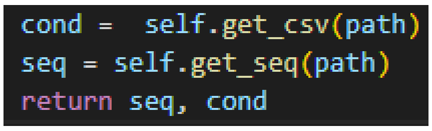
```python
def __len__(self):
    return self.len
```

Return the length calculated in the constructor.



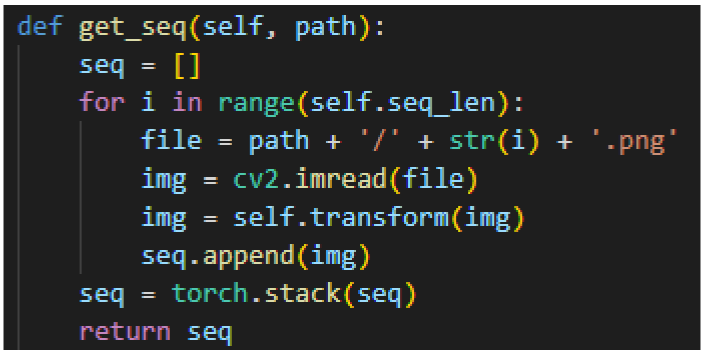
```python
def __getitem__(self, index):
    self.set_seed(index)
    if self.ordered:
        path = self.data_dirs[index]
    else:
        path = self.data_dirs[np.random.randint(self.len)]
```

In the __getitem__ function. I would use a random function to get an index if it need to randomly pick an element (training set). Otherwise, I would directly use input parameter index as its searching index.



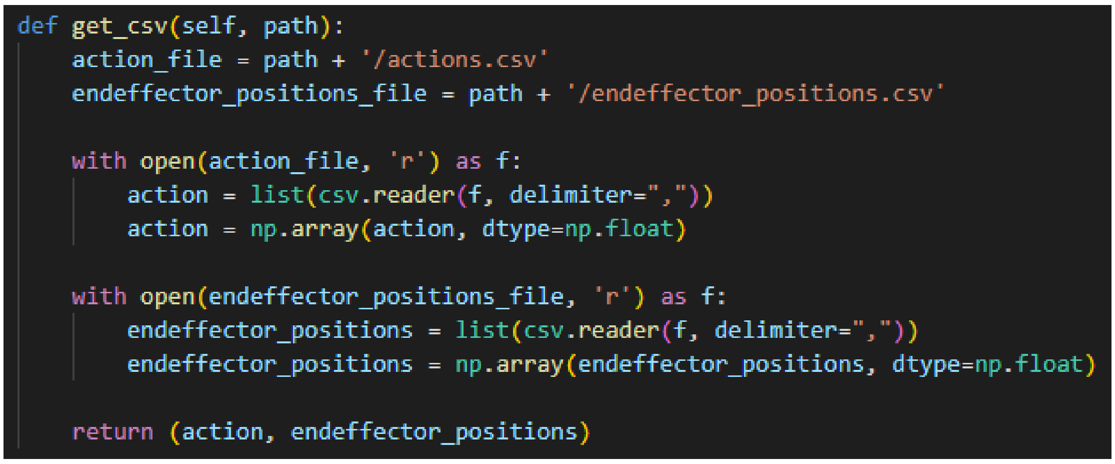
```python
cond =  self.get_csv(path)
seq = self.get_seq(path)
return seq, cond
```

Then, I would use the path to get its image sequence and condition.



```python
def get_seq(self, path):
    seq = []
    for i in range(self.seq_len):
        file = path + '/' + str(i) + '.png'
        img = cv2.imread(file)
        img = self.transform(img)
        seq.append(img)
    seq = torch.stack(seq)
    return seq
```

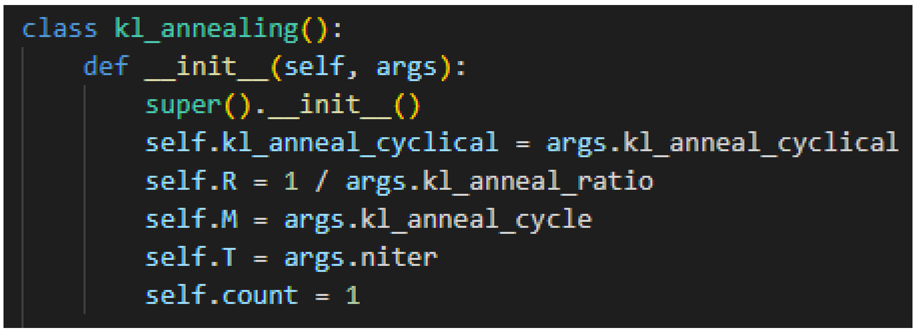
In the get_seq function, I would use a list to save all the image in that sequence. For each image, I would use opencv to read the file and use the transform to convert it to proper format. After all, I would use torch.stack to convert the list to a Tensor.



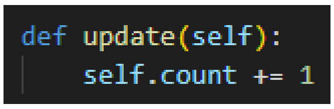
```python
def get_csv(self, path):
    action_file = path + '/actions.csv'
    endeffector_positions_file = path + '/endeffector_positions.csv'

    with open(action_file, 'r') as f:
        action = list(csv.reader(f, delimiter=","))
        action = np.array(action, dtype=np.float)

    with open(endeffector_positions_file, 'r') as f:
        endeffector_positions = list(csv.reader(f, delimiter=","))
        endeffector_positions = np.array(endeffector_positions, dtype=np.float)

    return (action, endeffector_positions)
```

In the get_csv function, I will go to the path to find corresponding csv files. Then, use csv reader to load them and convert them to numpy array.

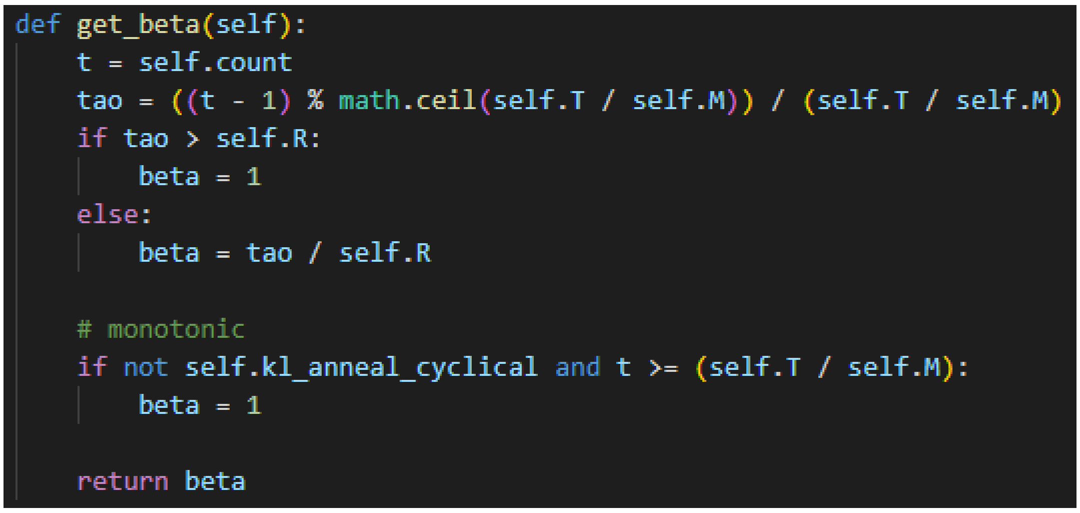
KL annealing:


```python
class kl_annealing():
    def __init__(self, args):
        super().__init__()
        self.kl_anneal_cyclical = args.kl_anneal_cyclical
        self.R = 1 / args.kl_anneal_ratio
        self.M = args.kl_anneal_cycle
        self.T = args.niter
        self.count = 1
```

In the constructor, I save necessary data from args. I use the same notation in KL annealing paper. Also, I use a variable "count" to save the current iteration number.


```python
def update(self):
    self.count += 1
```

In update function, just update "count" by 1.

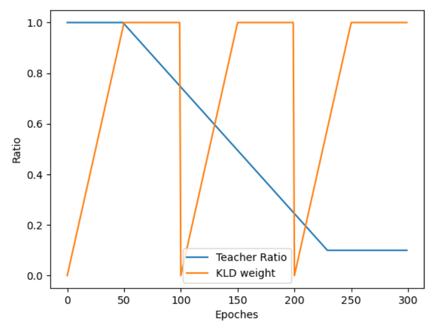
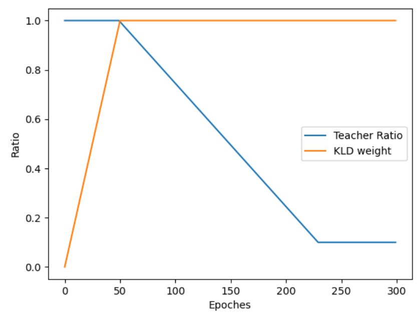

```python
def get_beta(self):
    t = self.count
    tao = ((t - 1) % math.ceil(self.T / self.M)) / (self.T / self.M)
    if tao > self.R:
        beta = 1
    else:
        beta = tao / self.R

    # monotonic
    if not self.kl_anneal_cyclical and t >= (self.T / self.M):
        beta = 1

    return beta
```

In get_beta function, I follow the paper's formula to calculate τ first. Then check whether it is larger than the kl annealing ratio. As for the monotonic, I would check whether it is in the first cycle, if not, set beta to 1.

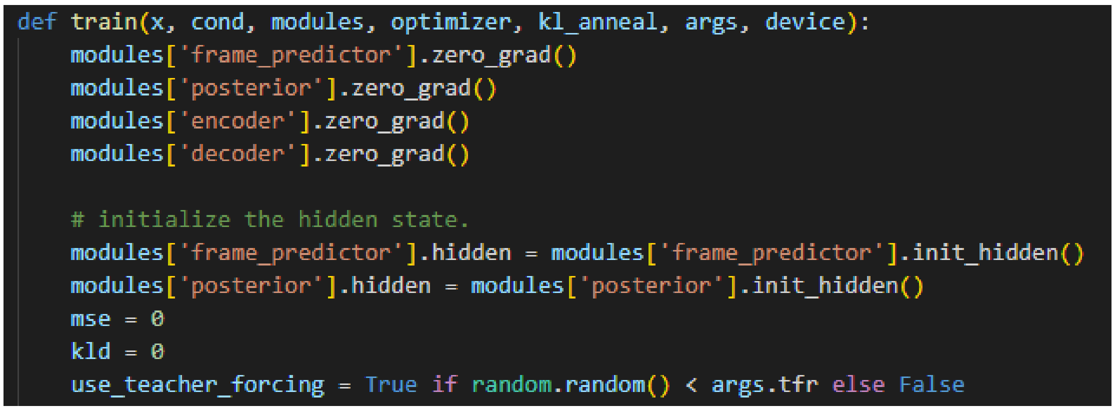
$$\beta_t = \begin{cases} f(\tau), & \tau \leq R \\ 1, & \tau > R \end{cases} \quad \text{with} \qquad (6)$$

$$\tau = \frac{\mathrm{mod}(t-1, \lceil T/M \rceil)}{T/M}, \qquad (7)$$

The following image is how my implementation work in 300 iteration. The orange line is the betas.

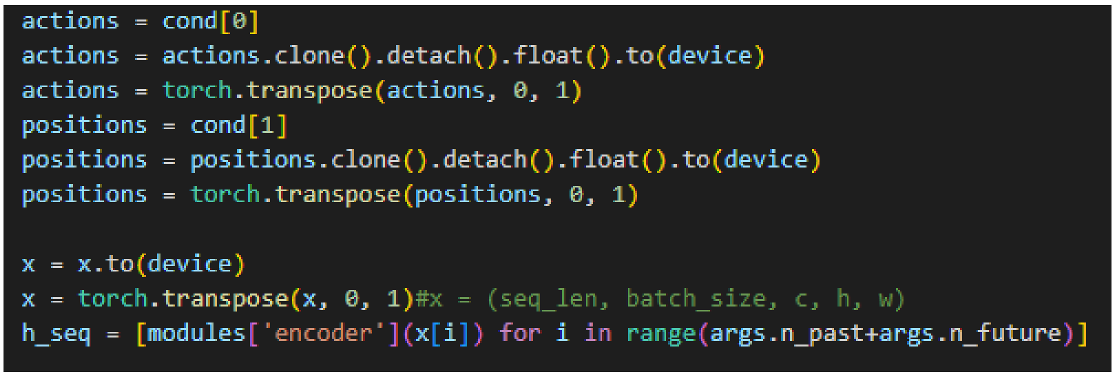
	
Cyclical with ratio 2, cycle3	Monotonic with ratio 2, cycle3

Train function:



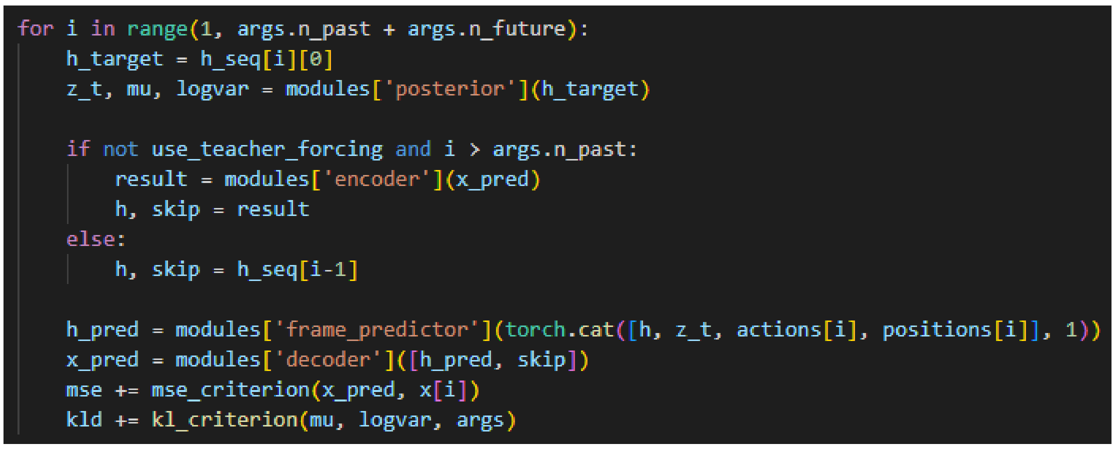
```
def train(x, cond, modules, optimizer, kl_anneal, args, device):
    modules['frame_predictor'].zero_grad()
    modules['posterior'].zero_grad()
    modules['encoder'].zero_grad()
    modules['decoder'].zero_grad()

    # initialize the hidden state.
    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
    modules['posterior'].hidden = modules['posterior'].init_hidden()
    mse = 0
    kld = 0
    use_teacher_forcing = True if random.random() < args.tfr else False
```

Set network's gradient to 0 and reset LSTMs' hidden unit.

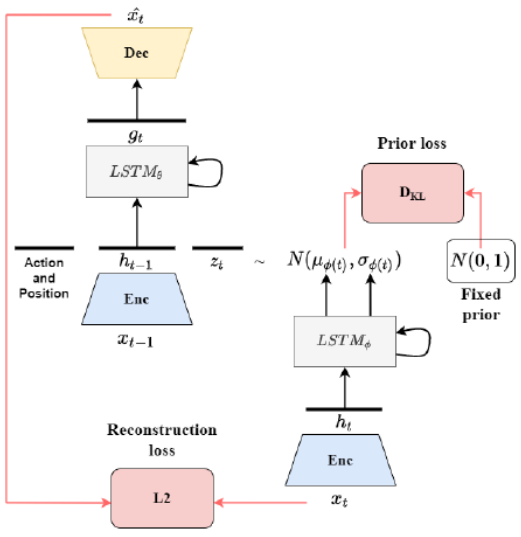


```
actions = cond[0]
actions = actions.clone().detach().float().to(device)
actions = torch.transpose(actions, 0, 1)
positions = cond[1]
positions = positions.clone().detach().float().to(device)
positions = torch.transpose(positions, 0, 1)

x = x.to(device)
x = torch.transpose(x, 0, 1)#x = (seq_len, batch_size, c, h, w)
h_seq = [modules['encoder'](x[i]) for i in range(args.n_past+args.n_future)]
```

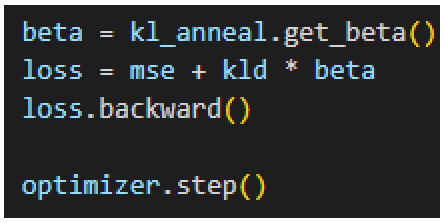
Then, I would move all the input data to GPU and transpose the first and second dimension. From (batch size, seq len, …) to (seq len, batch size, …). Then, I would put all the image into encoder to get their latent vectors.



```python
for i in range(1, args.n_past + args.n_future):
    h_target = h_seq[i][0]
    z_t, mu, logvar = modules['posterior'](h_target)

    if not use_teacher_forcing and i > args.n_past:
        result = modules['encoder'](x_pred)
        h, skip = result
    else:
        h, skip = h_seq[i-1]

    h_pred = modules['frame_predictor'](torch.cat([h, z_t, actions[i], positions[i]], 1))
    x_pred = modules['decoder']([h_pred, skip])
    mse += mse_criterion(x_pred, x[i])
    kld += kl_criterion(mu, logvar, args)
```



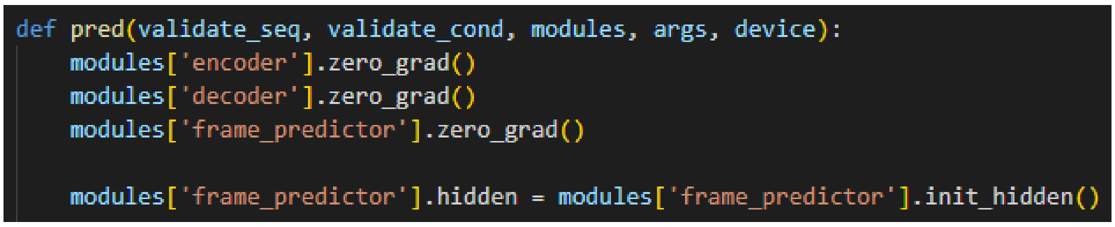
In the loop of time sequence, I first put current frame's latent vector ht into "LSTM posterior" to get the sample zt and model's mu, logvar. Then, check whether it is not using teacher forcing and it is predicting future frames. If yes, I would put last frame generated by the model into the encoder to get the input "h" of frame predictor, otherwise, use the last input frame's latent vector as "h". After that, concatenating the latent vector "h", sample vector "z", and the condition vector action and position. Putting them into frame predictor and put its output vector to the decoder to get the predicted frame. Finally, put the predicted frame and ground truth into MSE to calculate its reconstruction error and posterior's "mu" and "logvar" into KLD to calculate its prior loss.



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

optimizer.step()
```

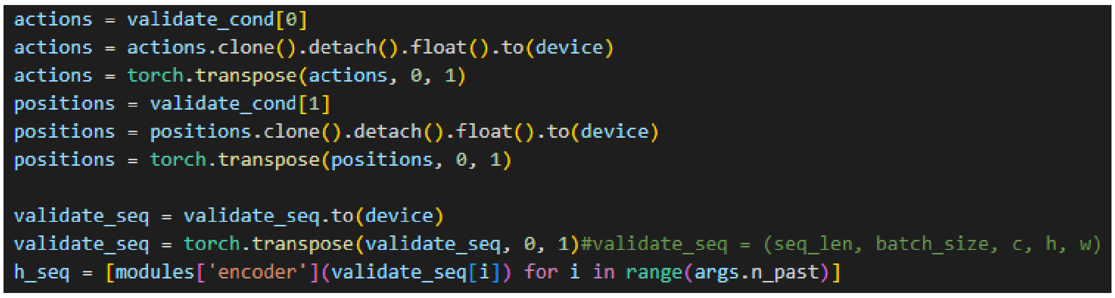
After the loop is finished, get the beta in KL annealing to calculate total loss. And do the back propagation on the model.

Pred function:



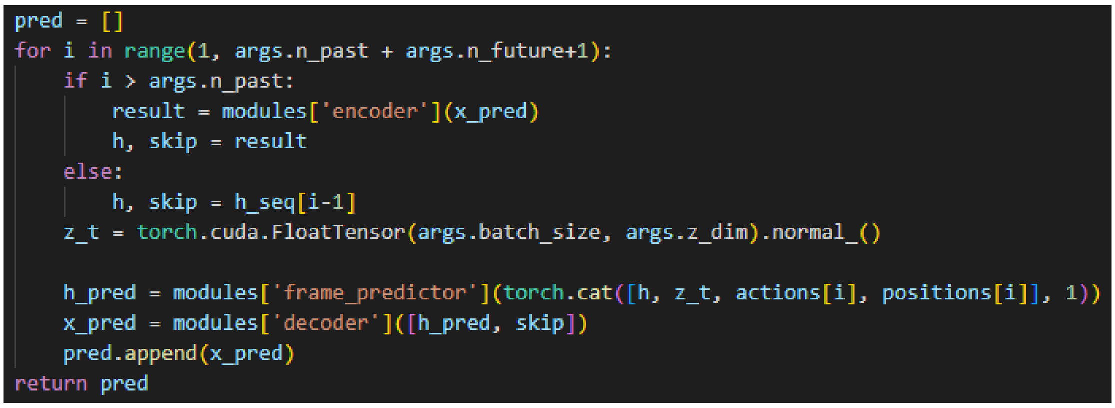
```python
def pred(validate_seq, validate_cond, modules, args, device):
    modules['encoder'].zero_grad()
    modules['decoder'].zero_grad()
    modules['frame_predictor'].zero_grad()

    modules['frame_predictor'].hidden = modules['frame_predictor'].init_hidden()
```

Like train function, set networks' gradient to 0 and reset LSTM's hidden unit.

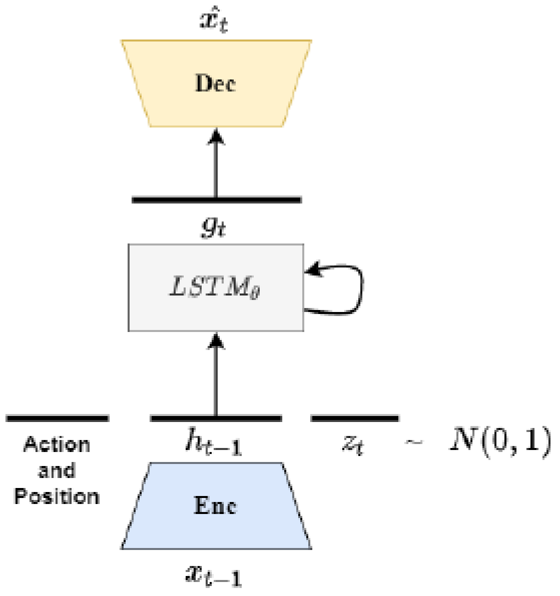


```python
actions = validate_cond[0]
actions = actions.clone().detach().float().to(device)
actions = torch.transpose(actions, 0, 1)
positions = validate_cond[1]
positions = positions.clone().detach().float().to(device)
positions = torch.transpose(positions, 0, 1)

validate_seq = validate_seq.to(device)
validate_seq = torch.transpose(validate_seq, 0, 1)#validate_seq = (seq_len, batch_size, c, h, w)
h_seq = [modules['encoder'](validate_seq[i]) for i in range(args.n_past)]
```

Like train function, move input data to GPU and transpose 1st and 2nd dimension. For the latent vector sequence, I only put the frames in the past to the encoder.



```python
pred = []
for i in range(1, args.n_past + args.n_future+1):
    if i > args.n_past:
        result = modules['encoder'](x_pred)
        h, skip = result
    else:
        h, skip = h_seq[i-1]
    z_t = torch.cuda.FloatTensor(args.batch_size, args.z_dim).normal_()

    h_pred = modules['frame_predictor'](torch.cat([h, z_t, actions[i], positions[i]], 1))
    x_pred = modules['decoder']([h_pred, skip])
    pred.append(x_pred)
return pred
```



In generating future image, if the time step is still in the past, I would use last image's latent vector as the input h of frame predictor, otherwise use last generated frame's latent vector. Then, sample from N(0, 1) to get "z" and concatenating the latent vector "h", sample vector "z", and the condition vector action and position. Putting them into frame predictor and put its output vector to the decoder to get the predicted frame.

● Describe the teacher forcing

Main idea and benefits:

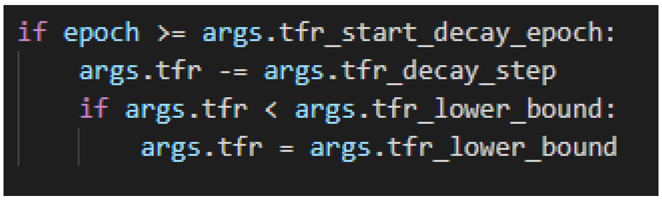
Teacher forcing is a training skill for RNNs that would use previous time step's output as current time step's input directly or indirectly. There are 2 training strategy, free running and teacher forcing. The free running strategy would directly use previous time step's output as current time step's input, the teacher forcing strategy use ground truth as current time step's input. In free running, if the RNN model output a bad output in some time steps, then the following output sequence may be vary from the ground truth. Then this output sequence may be not so meaningful for training. If using teacher forcing, use ground truth as input, then the following output may output a better sequence since the unexpected output wouldn't disturb the following result. Hence, RNNs can converge faster than free running during training.

Drawbacks:

If always use teacher forcing during training, in test/validate phase, the RNN may work badly since there are no ground truth can help in test/validate phase. If the RNN predict an expected result in a certain time step, the following output may not be so good.

Implementation:

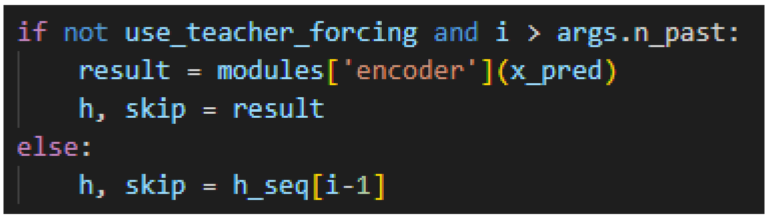
Teacher forcing decay:



```
if epoch >= args.tfr_start_decay_epoch:
    args.tfr -= args.tfr_decay_step
    if args.tfr < args.tfr_lower_bound:
        args.tfr = args.tfr_lower_bound
```

If the epoch is greater than “tfr_start_decay_epoch”, decay the teacher foring rate by “tfr_decay_step”, and make it no less than “tfr_lower_bound”

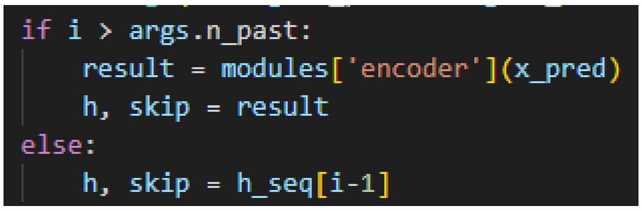
In train function:



```
if not use_teacher_forcing and i > args.n_past:
    result = modules['encoder'](x_pred)
    h, skip = result
else:
    h, skip = h_seq[i-1]
```

If use teacher forcing, LSTM’s input would be from ground truth’s encoder output when predicting future frames, otherwise, from the last predicted frame’s latent vector.

In pred function:



```
if i > args.n_past:
    result = modules['encoder'](x_pred)
    h, skip = result
else:
    h, skip = h_seq[i-1]
```

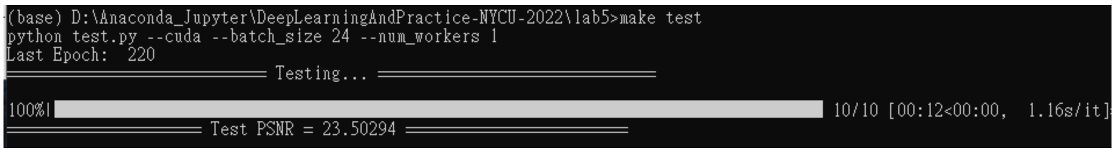
Teacher forcing is not used, LSTM’s input would always be from the last predicted frame’s latent vector when predicting future frames.

4. Results and discussion

- **Show your results of video prediction**

gif result is appended in the zip file

Best PSNR on test set:



```
(base) D:\Anaconda_Jupyter\DeepLearningAndPractice-NYCU-2022\lab5>make test
python test.py --cuda --batch_size 24 --num_workers 1
Last Epoch:  220
═══════════════════════ Testing... ═════════════════════════

100%|████████████████████████████████████████████████████| 10/10 [00:12<00:00,  1.16s/it]
════════════════════ Test PSNR = 23.50294 ════════════════════
```

- **Plot the KL loss and PSNR curves during training & Discuss**

Experiment setting:

Batch size: 24

Epoch size: 400

Iteration: 300

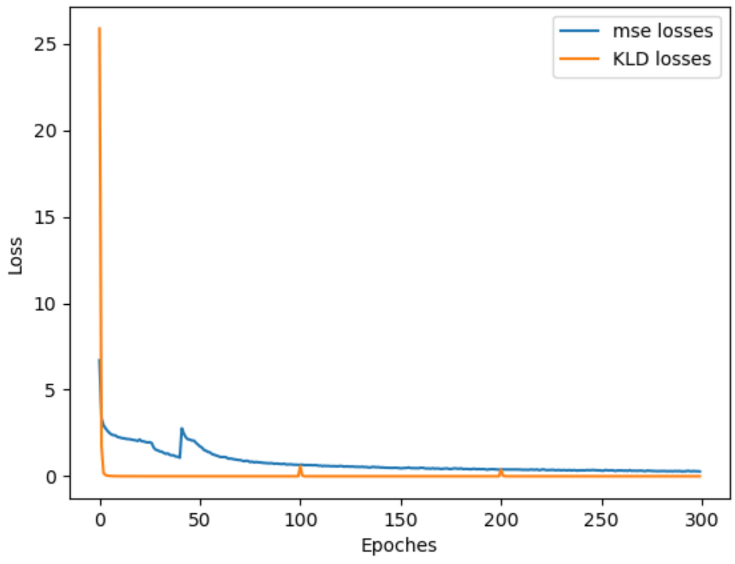
Optimizer: adam

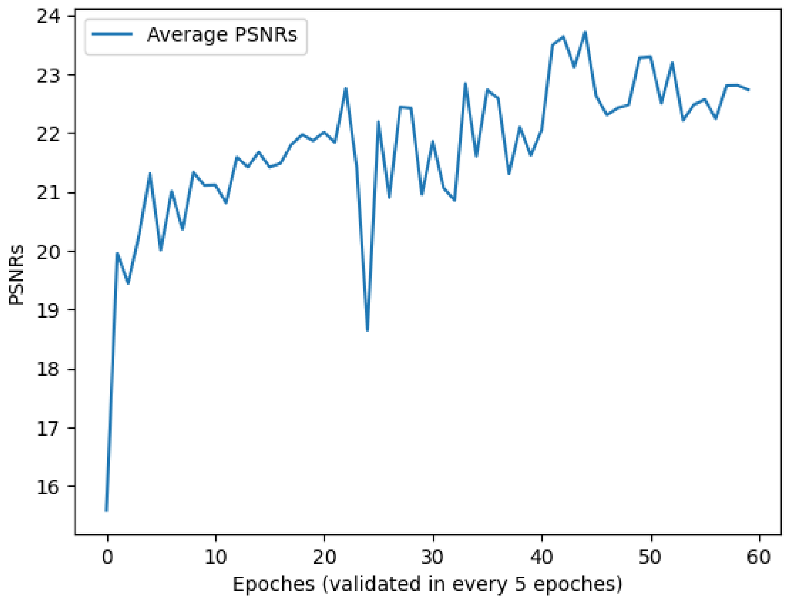
tfr_start_decay_epoch: 100

tfr_decay_step: 0.003

tfr_lower_bound: 0.1

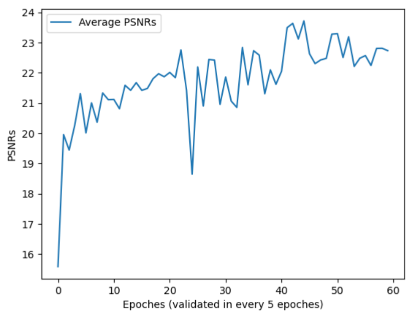
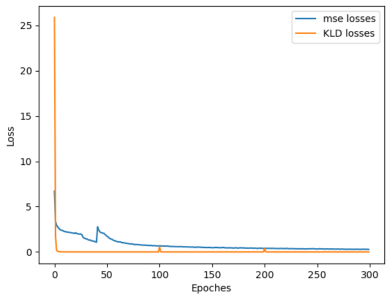
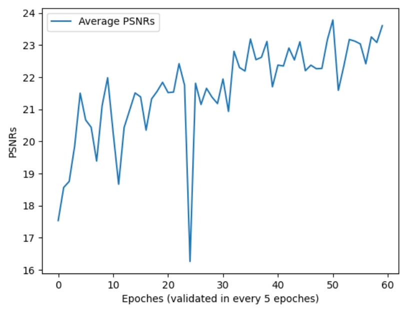
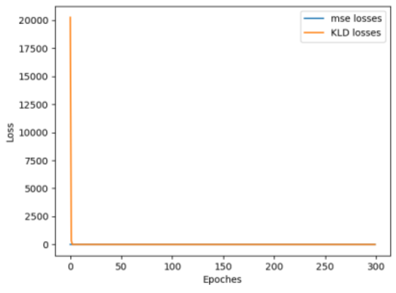
use KL annealing cyclical



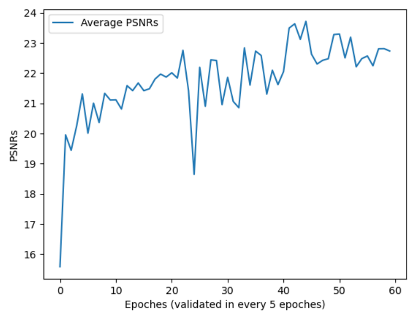
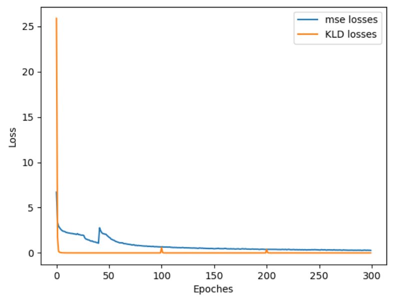
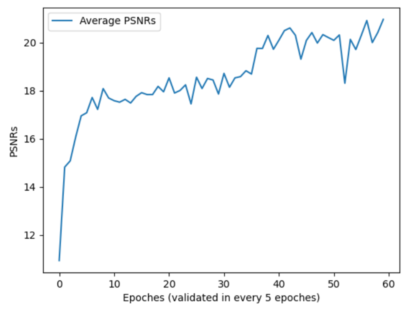
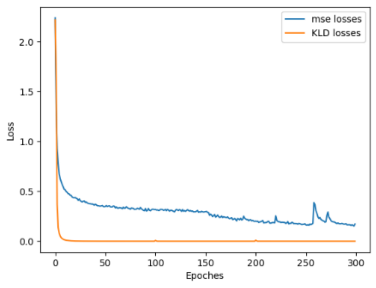


From the loss figure, we can see that both mse loss and KLD loss are very high. When the epochs is going, both loss are decreasing. We can find that the KLD loss would increase at 100th and 200th epoch. That is because we use KL annealing cyclical. The weight beta was set to 0 at 100th and 200th epoch.

Cyclical V.S. Monotonic

KL cyclical	KL monotonic
 	 

These 2 kinds of KL weight scheduler, in fact, I can’t see too much difference on their performance (PSNR and loss). One bigger difference is that at about 120 to 150 epochs, the cyclical seems to have a little higher PSNR than the monotonic one.

Using teacher forcing	Not using teacher forcing
 	 

From the figures, we can see that using teacher forcing’s model would have a much higher converge speed than the one that not use teacher forcing. The model not using teacher forcing has much lower PSNR than the one using teacher forcing