

Conditional VAE formula Derivation

Derivation of CVAE:

$$\log P(X|C; \theta) = \log P(X, z|C; \theta) - \log P(z|X, C; \theta)$$

Multiply an arbitrary distribution $q(z|c)$ on both sides and integrate over z

$$\begin{aligned} & \int q(z|c) \cdot \log P(X|C; \theta) dz \\ &= \int q(z|c) \log P(X, z|C; \theta) dz - \int q(z|c) \log P(z|X, C; \theta) dz \\ &= \int q(z|c) \log P(X, z|C; \theta) dz - \int q(z|c) \log q(z|c) dz \\ & \quad + \int q(z|c) \log q(z|c) dz - \int q(z|c) \log P(z|X, C; \theta) dz \\ &= \mathcal{L}(X, C, q, \theta) + KL(q(z|c) || \log P(z|X, C; \theta)) \end{aligned}$$

$$\text{Where } \mathcal{L}(X, C, q, \theta) = \int q(z|c) \log P(X, z|C; \theta) dz - \int q(z|c) \log q(z|c) dz$$

$$\text{and } KL(q(z|c) || \log P(z|X, C; \theta)) = \int q(z|c) \log \frac{q(z|c)}{P(z|X, C; \theta)} dz$$

Since $KL(q(z|c) || \log P(z|X, C; \theta)) \geq 0$

$\rightarrow \log P(X|C; \theta) \geq \mathcal{L}(X, C, q, \theta)$ equals when $q(z|c) = P(z|X, C; \theta)$

Therefore $\mathcal{L}(X, C, q, \theta)$ is a lower bound of $P(X|C; \theta)$

$$\begin{aligned} \mathcal{L}(X, C, q, \theta) &= \int q(z|c) \log P(X, z|C; \theta) dz - \int q(z|c) \log q(z|c) dz \\ &= \int q(z|c) \log P(X|z, C; \theta) dz + \int q(z|c) \log P(z|C; \theta) dz - \int q(z|c) \log q(z|c) dz \\ &= E_{z \sim q(z|X, C; \theta)} \log P(X|z, C; \theta) \\ & \quad + E_{z \sim q(z|X, C; \theta)} \log P(z|C; \theta) - E_{z \sim q(z|X, C; \theta)} \log q(z|X, C; \theta) \\ &= \underline{E_{z \sim q(z|X, C; \theta)} \log P(X|z, C; \theta) + KL(q(z|X, C; \theta) || P(z|C; \theta))} \end{aligned}$$

Introduction

This report presents the implementation and analysis of a Conditional Variational Autoencoder (VAE) for video prediction tasks. It details the training and testing protocols, including the setup of datasets, models, and optimization strategies such as KL annealing and teacher forcing. The report also provides an in-depth discussion of different KL annealing strategies and their impact on model performance, as well as the effects of teacher forcing on training stability. Finally, the analysis includes a comparison of PSNR growth over time, demonstrating the model's improvement in predicting video frames.

Implementation Details

Training / Test Protocol

Training

Prepare for training: setup dataset; setup model and set the model to training mode; setup optimizer and setup multistep learning rate; setup the MSE loss; instantiate the kl annealing object and the teacher forcing strategy object.

```
def train(args):
    # set random seed
    set_random_seed(args.seed)

    # dataset
    train_loader = get_data_loader(
        root=args.dataset_root,
        frame_H=args.frame_H,
        frame_W=args.frame_W,
        mode="train",
        video_len=args.train_vi_len,
        batch_size=args.batch_size,
        num_workers=args.num_workers,
        partial=args.fast_partial if args.fast_train else args.partial,
        shuffle=False,
        drop_last=True,
    )
    val_loader = get_data_loader(
        root=args.dataset_root,
        frame_H=args.frame_H,
        frame_W=args.frame_W,
        mode="val",
        video_len=args.train_vi_len,
        batch_size=args.batch_size,
        num_workers=args.num_workers,
        partial=1.0,
        shuffle=False,
        drop_last=True,
    )

    # model
    os.makedirs(args.model_root, exist_ok=True)
    model = VAE_Model(args).to(args.device)

    # optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(
        optimizer, milestones=[2, 5], gamma=0.1
    )

    # load latest checkpoint if available
    current_epoch = (
        load_latest_checkpoint(model, optimizer, scheduler, args.model_root)
    )

    # loss
    mse_criterion = torch.nn.MSELoss()

    # KL annealing
    kl_anneal = kl_annealing(args, current_epoch)

    # Teacher Forcing Ratio
    tf = teacher_forcing(args, current_epoch)

    # training
```

In each epoch, a sequence of images and labels are loaded from the data loader.

After loading the data, `train_step()` is called.

```
progress_bar = tqdm(train_loader, ncols=120)
for img, label in progress_bar:
    img = img.to(args.device)
    label = label.to(args.device)
    loss = train_step(
        model,
        img,
        label,
        tf.get_tfr(),
        optimizer,
        mse_criterion,
        kl_anneal_beta,
    )
```

At the end of the current epoch, validation is conducted with `validate()`. `validate()` do inference on the validation dataset, and return the average PSNR of the model. After validation, `kl_anneal` and `tf` are updated.

The validation logic is almost the same as `test()`, which will be discussed in the following.

```
# validate
avg_psnr = validate(model, val_loader, args.device)
psnr_list.append(avg_psnr)

kl_anneal.update()
tf.update()
```

In `train_step()`:

- Exchange the first and second dimension of `images` and `labels` (the frame number axis and the batch number axis).
- Determine if teacher forcing is adapted in this video prediction session.
- For each frame in the video loaded from `train()`, the following things are executed sequentially:
 - The previous frame (could be the last prediction or a real image) $X_{(t-1)}$, real image R_t and pose P_t are encoded with encoders respectively.
 - Predict the next frame X_t with the previous prediction $X_{(t-1)}$ and pose P_t .
 - Predict mean and variance with the Gaussian predictor.
 - Sample noise from the normal distribution with parameters mean and variance predicted previously.
 - Update `last_pred` with the predicted frame X_t .
 - The MSE loss and the KL loss are calculated respectively. The total loss of this frame is calculated by summing the MSE loss and the weighted KL loss.
- At the end of `train_step()`, the gradient is calculated, and the model weight is updated accordingly.

```
B, T, C, H, W = images.shape
images = images.view(T, B, C, H, W)
labels = labels.view(T, B, C, H, W)
last_pred = None
total_loss = 0.0

# t: frame_t to generate
adapt_teacher_forcing = random.random() < tfr
for t in range(1, T):
    img = images[t - 1] if adapt_teacher_forcing or last_pred is None else last_pred

    # encode
    img_features = model.frame_transformation(img)
    real_frame_features = model.frame_transformation(images[t])
    label_features = model.label_transformation(labels[t])

    # gaussian predictor
    z, mu, logvar = model.Gaussian_Predictor(real_frame_features, label_features)

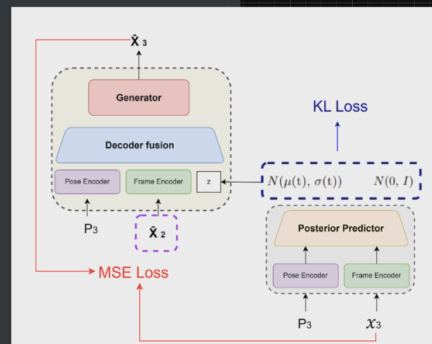
    # decode
    output = model.Decoder_Fusion(img_features, label_features, z)

    # generate
    prediction = model.Generator(output)
    last_pred = prediction

    # loss
    mse_loss = mse_criterion(prediction, images[t])
    kl_loss = kl_criterion(mu, logvar, B)
    loss = mse_loss + kl_anneal_beta * kl_loss
    total_loss += loss

optimizer.zero_grad()
total_loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1)
optimizer.step()

return total_loss
```



Test

Similar to the training part, the model is loaded, data loader is instantiated, and the model is set to evaluation mode.

After the preparation, for each video, `test_step()` is called to generate a sequence of predictions. Finally, the csv file `submission.csv` is generated with `save_submission()`.

```
def test(args):
    # Load model
    model = VAE_Model(args).to(args.device)
    if not args.model_path:
        raise ValueError("Please specify the model path")
    model.load_state_dict(torch.load(args.model_path, weights_only=True))

    # Load data
    test_loader = get_dataloader(
        root=args.dataset_root,
        frame_H=args.frame_H,
        frame_W=args.frame_W,
        mode="test",
        video_len=args.val_vi_len,
        batch_size=1,
        num_workers=args.num_workers,
        partial=1.0,
        shuffle=False,
        drop_last=True,
    )

    model.eval()
    pred_seq_list = []
    with torch.no_grad():
        for idx, (img, label) in enumerate(tqdm(test_loader, ncols=80)):
            img = img.to(args.device)
            label = label.to(args.device)
            pred_seq = test_step(model, img, label, args.device, idx, args.pred_root)
            pred_seq_list.append(pred_seq)

    save_submission(pred_seq_list, args.pred_root)
```

For each `test_step()` invocation, a sequence of frames are generated.

For each frame:

- The previous frame $X_{(t-1)}$ and pose P_t are encoded with encoders respectively.
- Sample noise from the standard normal distribution.
- Predict the next frame X_t with the previous prediction $X_{(t-1)}$ and pose P_t .
- Update `last_pred` with the predicted frame X_t .

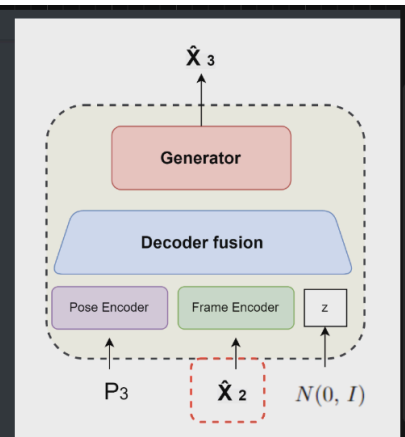
`test_step()` return a list of predictions eventually.

```
def test_step(model, images, labels, device, idx, pred_root):
    last_pred = images[0].to(device)
    # t: frame t to predict
    for t in range(1, T):
        img_features = model.frame_transformation(last_pred)
        label_features = model.label_transformation(labels[t])

        z = torch.randn(B, 12, H, W).to(device)
        output = model.Decoder_Fusion(img_features, label_features, z)
        output = model.Generator(output)
        last_pred = output

        decoded_frame_list.append(output.cpu())

    # Please do not modify this part, it is used for visualization
    generated_frame = torch.stack(decoded_frame_list).permute(1, 0, 2, 3, 4)
```



Reparameterization Tricks

The log variance is predicted in the last model output.

The standard deviation is derived from the following calculation:

$$\begin{aligned} \log var &= \log(var) \\ &= \log(std^2) \\ &= 2\log(std) \\ \Rightarrow std &= \exp(\frac{1}{2}\log(var)) \end{aligned}$$

The second line in the following snippet is the realization of the above calculation.

Next, the random noise `eps` is sampled from the standard normal distribution.

Finally, the noise `z` is calculated.

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

Teacher Forcing Strategy

- `tfr_sde`: The epoch that teacher forcing ratio starts to decay.
- `tfr_d_step`: Size of decay step.
- `update()`: Called at the end of the training loop. Recalculate the new teacher forcing ratio and update it accordingly.
- `calculate_tfr()`: Check if the current number of epoch has exceeded `tfr_sde`. If so, start decaying the ratio according to `tfr_d_step`.
- `get_tfr()`: Return the ratio.
- `adapt_teacher_forcing()`: Return if teacher forcing should be adopted in this epoch.

```
class teacher_forcing:
    def __init__(self, args, current_epoch=0):
        self.tfr_init = args.tfr_init
        self.tfr_min = args.tfr_min
        self.tfr_sde = args.tfr_sde
        self.tfr_d_step = args.tfr_d_step
        self.current_epoch = current_epoch
        self.tfr = self.calculate_tfr()

    def update(self):
        self.current_epoch += 1
        self.tfr = self.calculate_tfr()

    def calculate_tfr(self):
        if self.current_epoch <= self.tfr_sde:
            tfr = self.tfr_init
        else:
            tfr = max(
                self.tfr_min,
                self.tfr_init - self.tfr_d_step * (self.current_epoch - self.tfr_sde),
            )
        return tfr

    def get_tfr(self):
        return self.tfr

    def adapt_teacher_forcing(self):
        return random.random() < self.tfr
```

KL Annealing Strategy

- `kl_anneal_cycle`: Cycle length of kl annealing.
- `beta`: Weight of KL loss.
- `update()`: Called at the end of the training loop. Recalculate the new `beta` and update it accordingly.
- `calculate_beta()`: If `kl_anneal_type` is `Monotonic`, `beta` grow linearly until it reaches 1.0; If `kl_anneal_type` is `Cyclical`, in each cycle with length `kl_anneal_cycle`, `beta` grow linearly in the first half, and the value hold in the second half; Else, `beta` is 1.0.
- `get_beta()`: Return `beta`.

```
class kl_annealing:
    def __init__(self, args, current_epoch=0):
        self.kl_anneal_type = args.kl_anneal_type
        self.kl_anneal_cycle = args.kl_anneal_cycle
        self.kl_anneal_ratio = args.kl_anneal_ratio
        self.current_epoch = current_epoch
        self.beta = self.calculate_beta()

    def calculate_beta(self):
        if self.kl_anneal_type == "Cyclical":
            cycle = self.current_epoch % self.kl_anneal_cycle / self.kl_anneal_cycle
            return min(1.0, 2 * cycle)
        elif self.kl_anneal_type == "Monotonic":
            return min(1.0, self.current_epoch / self.kl_anneal_cycle)
        else:
            return 1.0

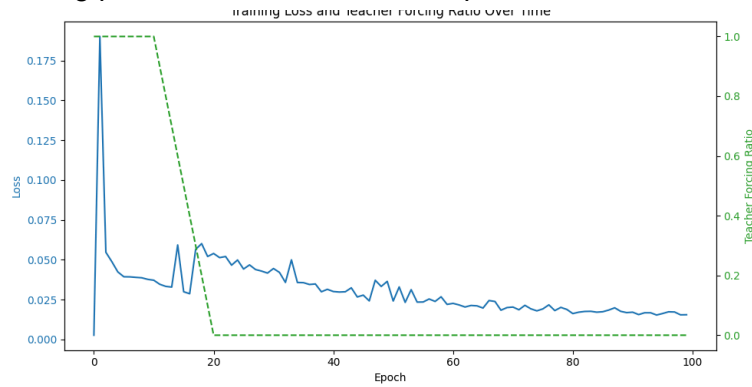
    def update(self):
        self.current_epoch += 1
        self.beta = self.calculate_beta()

    def get_beta(self):
        return self.beta
```

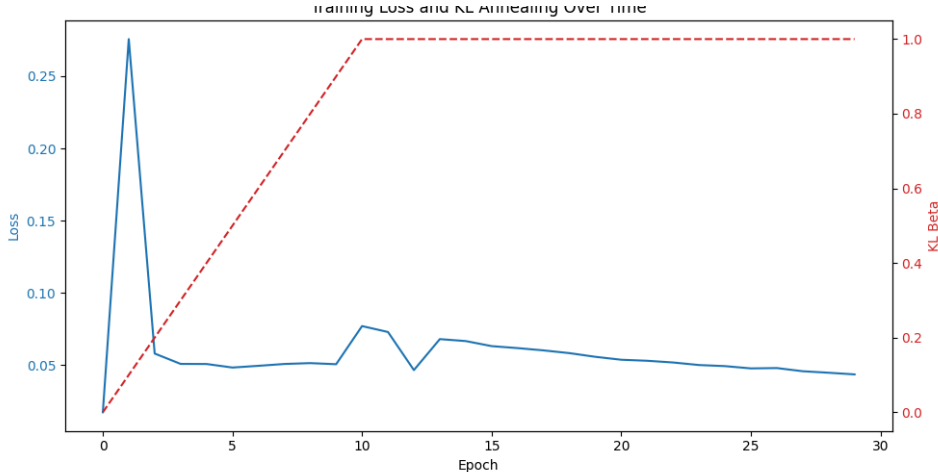
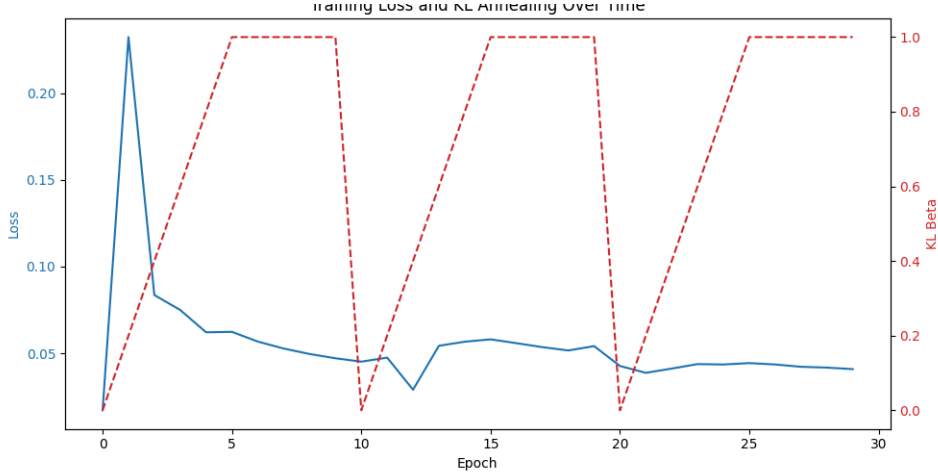
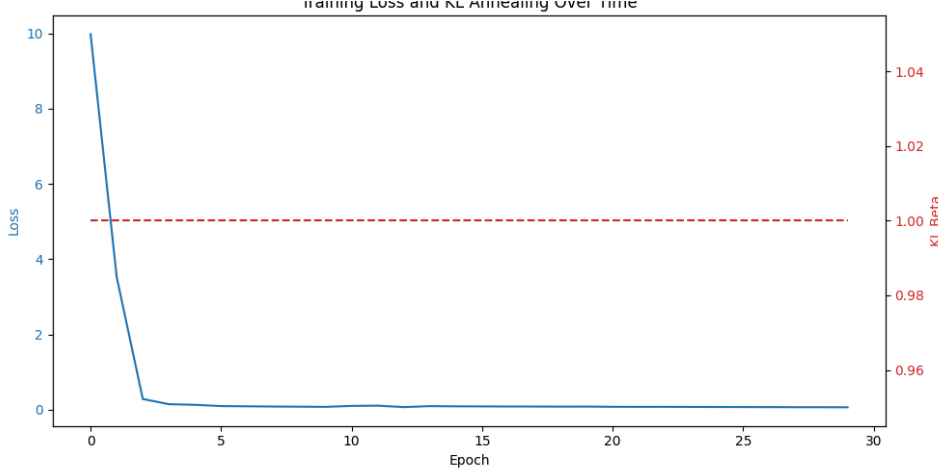
Analysis & Discussion

Teacher Forcing Ratio

As the teacher forcing ratio decreases, the training loss begins to rise and fluctuate, indicating that the model is encountering a more challenging training process. This suggests that predicting the entire video independently is significantly more difficult for the model than making predictions with real frames provided.



Different KL Annealing Strategy

KL Annealing Type	Diagram																														
Monotonic	 <p>The plot titled "Training Loss and KL Annealing Over Time" shows training loss (blue solid line) and KL Beta (red dashed line) over 30 epochs. The left y-axis for Loss ranges from 0.05 to 0.25, and the right y-axis for KL Beta ranges from 0.0 to 1.0. The loss starts at ~0.02, peaks at ~0.27 at epoch 1, then drops to ~0.05 by epoch 2 and remains relatively stable with minor fluctuations until epoch 30. The KL Beta starts at 0.0 and increases linearly to 1.0 by epoch 10, where it plateaus.</p> <table><tr><th>Epoch</th><th>Loss</th><th>KL Beta</th></tr><tr><td>0</td><td>0.02</td><td>0.0</td></tr><tr><td>1</td><td>0.27</td><td>0.05</td></tr><tr><td>2</td><td>0.05</td><td>0.1</td></tr><tr><td>5</td><td>0.05</td><td>0.35</td></tr><tr><td>10</td><td>0.07</td><td>1.0</td></tr><tr><td>15</td><td>0.06</td><td>1.0</td></tr><tr><td>20</td><td>0.05</td><td>1.0</td></tr><tr><td>25</td><td>0.05</td><td>1.0</td></tr><tr><td>30</td><td>0.04</td><td>1.0</td></tr></table>	Epoch	Loss	KL Beta	0	0.02	0.0	1	0.27	0.05	2	0.05	0.1	5	0.05	0.35	10	0.07	1.0	15	0.06	1.0	20	0.05	1.0	25	0.05	1.0	30	0.04	1.0
Epoch	Loss	KL Beta																													
0	0.02	0.0																													
1	0.27	0.05																													
2	0.05	0.1																													
5	0.05	0.35																													
10	0.07	1.0																													
15	0.06	1.0																													
20	0.05	1.0																													
25	0.05	1.0																													
30	0.04	1.0																													
Cyclical	 <p>The plot titled "Training Loss and KL Annealing Over Time" shows training loss (blue solid line) and KL Beta (red dashed line) over 30 epochs. The left y-axis for Loss ranges from 0.05 to 0.20, and the right y-axis for KL Beta ranges from 0.0 to 1.0. The loss starts at ~0.02, peaks at ~0.23 at epoch 1, then drops to ~0.08 by epoch 2. It then fluctuates between 0.04 and 0.06 as the KL Beta cycles between 0.0 and 1.0 in a sawtooth pattern every 5 epochs. The loss generally decreases as the KL Beta cycles.</p> <table><tr><th>Epoch</th><th>Loss</th><th>KL Beta</th></tr><tr><td>0</td><td>0.02</td><td>0.0</td></tr><tr><td>1</td><td>0.23</td><td>0.05</td></tr><tr><td>2</td><td>0.08</td><td>0.1</td></tr><tr><td>5</td><td>0.06</td><td>1.0</td></tr><tr><td>10</td><td>0.04</td><td>0.0</td></tr><tr><td>15</td><td>0.05</td><td>1.0</td></tr><tr><td>20</td><td>0.04</td><td>0.0</td></tr><tr><td>25</td><td>0.04</td><td>1.0</td></tr><tr><td>30</td><td>0.04</td><td>0.0</td></tr></table>	Epoch	Loss	KL Beta	0	0.02	0.0	1	0.23	0.05	2	0.08	0.1	5	0.06	1.0	10	0.04	0.0	15	0.05	1.0	20	0.04	0.0	25	0.04	1.0	30	0.04	0.0
Epoch	Loss	KL Beta																													
0	0.02	0.0																													
1	0.23	0.05																													
2	0.08	0.1																													
5	0.06	1.0																													
10	0.04	0.0																													
15	0.05	1.0																													
20	0.04	0.0																													
25	0.04	1.0																													
30	0.04	0.0																													
None	 <p>The plot titled "Training Loss and KL Annealing Over Time" shows training loss (blue solid line) and KL Beta (red dashed line) over 30 epochs. The left y-axis for Loss ranges from 0 to 10, and the right y-axis for KL Beta ranges from 0.96 to 1.04. The loss starts at 10 and drops sharply to near 0 by epoch 2, remaining stable thereafter. The KL Beta is a constant horizontal dashed line at 1.0.</p> <table><tr><th>Epoch</th><th>Loss</th><th>KL Beta</th></tr><tr><td>0</td><td>10</td><td>1.0</td></tr><tr><td>1</td><td>3.5</td><td>1.0</td></tr><tr><td>2</td><td>0.2</td><td>1.0</td></tr><tr><td>5</td><td>0.1</td><td>1.0</td></tr><tr><td>10</td><td>0.1</td><td>1.0</td></tr><tr><td>15</td><td>0.1</td><td>1.0</td></tr><tr><td>20</td><td>0.1</td><td>1.0</td></tr><tr><td>25</td><td>0.1</td><td>1.0</td></tr><tr><td>30</td><td>0.1</td><td>1.0</td></tr></table>	Epoch	Loss	KL Beta	0	10	1.0	1	3.5	1.0	2	0.2	1.0	5	0.1	1.0	10	0.1	1.0	15	0.1	1.0	20	0.1	1.0	25	0.1	1.0	30	0.1	1.0
Epoch	Loss	KL Beta																													
0	10	1.0																													
1	3.5	1.0																													
2	0.2	1.0																													
5	0.1	1.0																													
10	0.1	1.0																													
15	0.1	1.0																													
20	0.1	1.0																													
25	0.1	1.0																													
30	0.1	1.0																													

Parameters

These 3 models are trained under the following parameters:

- batch size: 4
- learning rate: 0.001
- number of epochs: 30
- dataset partial: 0.4
- teacher forcing ratio start decay epoch: 8
- teacher forcing ratio decay step: 0.1

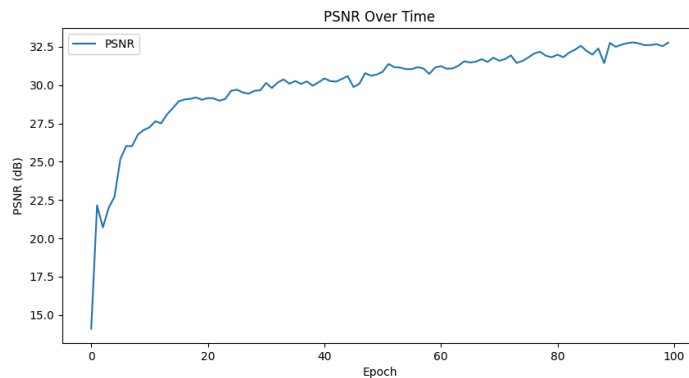
Difference Analysis

The loss value observed with the cyclical KL annealing strategy fluctuates the most among the 3 strategies. This is because the cyclical nature of the KL annealing introduces periodic changes in the weight of the KL divergence term, causing the model to alternate between phases of strong and weak regularization.

The loss value observed with the monotonic KL annealing strategy is smoother compared to other strategies. This is because the KL divergence is gradually increased consistently, allowing the model to adjust more steadily over time.

PSNR-per-Frame Diagram

It can be observed that the PSNR (Peak Signal-to-Noise Ratio) grows over time, which indicates that the model is increasingly improving its performance on the validation dataset. This trend is a positive sign, showing that the model is learning and refining its understanding of predicting the frames.



Execution Instructions

Arguments

Modify arguments in `config/{mode}.yaml`.
`mode` should be `train` or `test`.

Training / Test commands

To train: `python3 Trainer.py`

To do model inference: `python3 Tester.py`