Lab 04: Conditional VAE for Video Prediction

112550127 Pin-Kuan Chiang

August 5, 2025

1 Introduction

Video prediction—forecasting future frames from past context—is key for applications in autonomous driving, robotics, and compression. Deterministic models tend to blur under uncertainty, while Variational Autoencoders (VAEs) can capture multi-modal futures but often suffer from KL collapse or incoherent dynamics. Conditional VAEs (CVAEs) mitigate these issues by learning a context-dependent prior [2], and cyclical KL annealing further prevents collapse [3].

In this work, we implement a CVAE for video prediction following Chan et al. [1], introduce a flexible teacher forcing schedule, compare no, monotonic, and cyclical KL annealing, and evaluate performance via PSNR and loss metrics. Our main contributions are:

- A reproducible CVAE training/testing pipeline for frame prediction.
- Reparameterization and epoch-wise teacher forcing with adjustable decay.
- Empirical comparison of KL schedules: none, monotonic, and cyclical.
- Quantitative analysis using per-frame PSNR and loss curves.

2 Implementation Details

```python
def forward(self, curr_frame, prev_frame, curr_label):
    # Encode previous frame and label
    prev_feat = self.frame_transformation(prev_frame).detach()      # [batch, F_dim]
    curr_feat = self.frame_transformation(curr_frame)                # [batch, F_dim]
    label_feat = self.label_transformation(curr_label)               # [batch, L_dim]

    # Posterior prediction
    z, mu, logvar = self.Gaussian_Predictor.forward(curr_feat, label_feat)

    # Decoder fusion
    fusion = self.Decoder_Fusion.forward(prev_feat, label_feat, z)

    # Generate frame
    gen_frame = self.Generator.forward(fusion)                   # [batch, C, H, W]

    return gen_frame, mu, logvar
```

The `forward` method encodes the previous frame and current frame features, as well as the label features, then predicts the latent variable z along with its distribution parameters μ and $\log \sigma^2$. These features are fused in the decoder fusion module, and the fused representation is passed through the generator to produce the new frame.

```python
def training_stage(self):
    for i in range(1, self.args.num_epoch + 1):
        train_loader = self.train_dataloader()
        adapt_TeacherForcing = True if random.random() < self.tfr else False

        epoch_loss = 0

        for batch_idx, (img, label) in enumerate(pbar := tqdm(train_loader, ncols=120)):
            img = img.to(self.args.device)
            label = label.to(self.args.device)
```

```
            loss, beta = self.training_one_step(img, label, adapt_TeacherForcing)
            epoch_loss += loss.item()

            if adapt_TeacherForcing:
                self.tqdm_bar('train [TeacherForcing: ON, {:.1f}], beta: {}'.format(self.tfr,
                    beta), pbar, loss.detach().cpu(), lr=self.scheduler.get_last_lr()[0])
            else:
                self.tqdm_bar('train [TeacherForcing: OFF, {:.1f}], beta: {}'.format(self.tfr,
                     beta), pbar, loss.detach().cpu(), lr=self.scheduler.get_last_lr()[0])

        # wandb log per epoch
        self.wandb.log({
            'Loss/train_epoch': epoch_loss / len(train_loader),
            'TeacherForcingRatio/epoch': self.tfr,
            'Beta/epoch': self.kl_annealing.get_beta(),
        }, step=self.current_epoch)

        if self.current_epoch % self.args.per_save == 0:
            self.save(os.path.join(self.args.save_root, f"epoch={self.current_epoch}.ckpt"))

        self.eval()
        self.current_epoch += 1
        self.scheduler.step()
        self.teacher_forcing_ratio_update()
        self.kl_annealing.update()
```

The `training_stage` method runs the main training loop over epochs. For each epoch, it iterates through the training data loader, randomly decides whether to apply Teacher Forcing based on the current ratio, and calls `training_one_step` to compute and accumulate the batch loss. After each epoch, it logs metrics to Weights and Biases, saves a checkpoint periodically, runs validation, and updates learning rate and scheduling parameters.

```
@torch.no_grad()
def eval(self):
    val_loader = self.val_dataloader()
    val_losses = []
    val_psnrs = []
    for (img, label) in (pbar := tqdm(val_loader, ncols=120, desc=f"Validation Epoch {self.
        current_epoch}")):
        img = img.to(self.args.device)
        label = label.to(self.args.device)
        loss, avg_psnr = self.val_one_step(img, label)
        val_losses.append(loss.item())
        val_psnrs.append(avg_psnr)
        pbar.set_postfix(loss=f"{loss.item():.6f}", lr=f"{self.scheduler.get_last_lr()[0]:.6f}
            ", psnr=f"{avg_psnr:.4f}")
    self.wandb.log({
        'Loss/val_epoch': np.mean(val_losses),
        'PSNR/val_epoch': np.mean(val_psnrs),
        'lr': self.scheduler.get_last_lr()[0],
    }, step=self.current_epoch)
    print(f"(val) Epoch {self.current_epoch} completed")
```

The `eval` method evaluates the model on the validation set without gradient updates. It collects per-batch validation loss and PSNR, updates the progress bar, and logs aggregated metrics to Weights and Biases for monitoring validation performance.

```
def training_one_step(self, img, label, adapt_TeacherForcing):
    self.train()
    batch_size, seq_len, C, H, W = img.size()
    loss_total = 0

    prev_frame = img[:, 0]  # initial previous frame
    for t in range(1, seq_len):
        curr_frame = img[:, t]
        curr_label = label[:, t]

        # Teacher Forcing: use ground truth previous frame, else use generated
        if adapt_TeacherForcing or t == 1:
            input_prev = prev_frame
        else:
            input_prev = gen_frame.detach()  # use last generated frame
```

```
        # Forward pass: expects current frame and previous frame
        out = self.forward(curr_frame, input_prev, curr_label)
        gen_frame, mu, logvar = out  # adjust if your forward returns differently

        # NaN prevention for outputs
        gen_frame = torch.clamp(gen_frame, 0, 1)
        logvar = torch.clamp(logvar, min=-10.0, max=10.0)

        # Reconstruction loss
        recon_loss = self.mse_criterion(gen_frame, curr_frame) / batch_size
        # KL loss
        kl_loss = kl_criterion(mu, logvar, batch_size)
        # Annealing
        beta = self.kl_annealing.get_beta()
        loss = recon_loss + beta * kl_loss

        loss_total += loss
        prev_frame = curr_frame if adapt_TeacherForcing else gen_frame

    self.optim.zero_grad()
    loss_total.backward()
    self.optimizer_step()
    return loss_total, beta
```

The `training_one_step` method performs a single training pass over a sequence. It iterates frame by frame, decides whether to use the ground-truth previous frame (Teacher Forcing) or the model's own prediction, computes reconstruction and KL-divergence losses with annealing, accumulates the total loss, and finally backpropagates and updates the model parameters.

```
@torch.no_grad()
def val_one_step(self, img, label):
    super().eval()
    batch_size, seq_len, C, H, W = img.size()
    loss_total = 0
    psnr_list = []

    prev_frame = img[:, 0]
    for t in range(1, seq_len):
        curr_frame = img[:, t]
        curr_label = label[:, t]

        input_prev = prev_frame if t == 1 else gen_frame.detach()
        gen_frame, mu, logvar = self.forward(curr_frame, input_prev, curr_label)
        gen_frame = torch.clamp(gen_frame, 0, 1)

        recon_loss = self.mse_criterion(gen_frame, curr_frame) / batch_size
        kl_loss = kl_criterion(mu, logvar, batch_size)
        beta = self.kl_annealing.get_beta()
        loss = recon_loss + beta * kl_loss

        loss_total += loss
        psnr_list.append(Generate_PSNR(gen_frame, curr_frame).item())
        prev_frame = gen_frame

    avg_psnr = np.mean(psnr_list)

    """
        Plot PSNR-per frame to wandb
        ...
    """

    return loss_total, avg_psnr
```

The `val_one_step` method runs validation for one sequence without updating weights. It computes and accumulates per-frame reconstruction and KL losses, calculates PSNR for each frame, logs a PSNR plot to Weights and Biases, and returns the total loss and average PSNR.

2.1 Reparameterization Trick

To enable gradients to flow through a stochastic sampling operation, we adopt the standard reparameterization formulation:

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I),$$

where the encoder network predicts the mean μ and log-variance $\log \sigma^2$. By expressing the random variable z as a deterministic function of μ, σ, and a noise term ϵ, we can backpropagate through the sampling step.

Concretely, given the encoder outputs μ and $\log \sigma^2$, we compute:

$$\sigma = \exp\big(0.5 \log \sigma^2\big), \quad \epsilon \sim \mathcal{N}(0, I), \quad z = \mu + \sigma \odot \epsilon.$$

This formulation ensures that gradients with respect to μ and $\log \sigma^2$ are well-defined, since the randomness is isolated in ϵ, which does not depend on network parameters.

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

By using this trick, our variational encoder can learn both the mean and variance parameters directly via gradient descent, while still modeling uncertainty in the latent space.

2.2 Teacher Forcing Strategy

Rather than decaying the teacher forcing ratio linearly at every time step, we decide once per epoch whether to apply teacher forcing for all time steps in that epoch. Concretely, at the start of each epoch we sample

$$\text{adapt\_TeacherForcing} \sim \text{Bernoulli}\big(r(t)\big),$$

where $r(t)$ is our current teacher forcing ratio. To gradually reduce reliance on ground-truth frames, after a warm-up period of T_{sde} epochs we multiply the ratio by a fixed decay factor $d < 1$ each epoch, i.e.

$$r(t+1) = \max\big(0,\ r(t) \times d\big) \quad \text{for } t \geq T_{\text{sde}}.$$

```python
def teacher_forcing_ratio_update(self):
    if self.current_epoch >= self.tfr_sde:
        self.tfr = max(0.0, self.tfr * self.tfr_d_step)

# At the start of each epoch:
adapt_TeacherForcing = True if random.random() < self.tfr else False
```

The method `teacher_forcing_ratio_update` applies a multiplicative decay $d =$ `self.tfr_d_step` (e.g. 0.95) once the epoch index `current_epoch` exceeds the warm-up threshold `tfr_sde`. Then, for each epoch we draw a single Bernoulli sample `adapt_TeacherForcing` with probability equal to the current ratio `self.tfr`. If `True`, all decoder steps in that epoch use the ground-truth previous frame; otherwise the model's own predictions are fed back. This approach maintains stable training early on and smoothly transitions to fully autoregressive generation.

2.3 KL Annealing Ratio

We implement a modular KL annealing schedule in the `kl_annealing` class, allowing for no annealing, monotonic increase, or cyclical behavior. The weight $\beta(t)$ is obtained via the `get_beta()` method based on the chosen strategy:

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

    def update(self):
        self.current_epoch += 1
```

```
        return self.current_epoch

    def get_beta(self):
        if(self.kl_anneal_type == 'Cyclical'):
            return self.frange_cycle_linear(
                self.current_epoch,
                start=0.0, stop=1.0,
                n_cycle=self.kl_anneal_cycle,
                ratio=self.kl_anneal_ratio
            )
        elif(self.kl_anneal_type == 'Monotonic'):
            if(self.current_epoch < self.kl_anneal_cycle):
                return self.frange_cycle_linear(
                    self.current_epoch,
                    start=0.0, stop=1.0,
                    n_cycle=self.kl_anneal_cycle,
                    ratio=self.kl_anneal_ratio
                )
            else:
                return 1.0
        elif(self.kl_anneal_type == 'None'):
            return 1.0

    def frange_cycle_linear(self, n_iter, start=0.0, stop=1.0, n_cycle=1, ratio=1):
        x = (n_iter % n_cycle) / n_cycle
        if x <= ratio:
            return start + (stop - start) * (x / ratio)
        else:
            return stop
```

At each epoch, `update()` increments the internal epoch counter. The `get_beta()` method then returns:

- **None**: $\beta = 1.0$ always.
- **Monotonic**: during the first `kl_anneal_ratio` $\times$ `kl_anneal_cycle` epochs, β increases linearly from 0 to 1, then remains at 1.0 thereafter.
- **Cyclical**: within each cycle of length `kl_anneal_cycle` epochs, β increases linearly from 0 to 1 over the first `kl_anneal_ratio` $\times$ `kl_anneal_cycle` epochs, then stays at 1.0 for the remainder of the cycle.

The helper `frange_cycle_linear` computes a value in $[0, 1]$ by taking the current epoch modulo the cycle length and scaling it linearly up to `ratio`, after which it stays at 1.0 for the remainder of the cycle.

In the training loop, the returned β is applied as

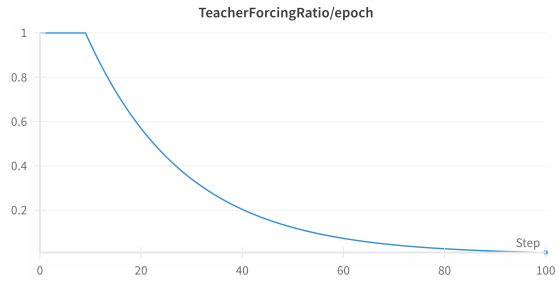
$$\text{loss} = \text{recon\_loss} + \beta \times \text{kl\_loss},$$

so that the KL term is gradually introduced according to the selected schedule.

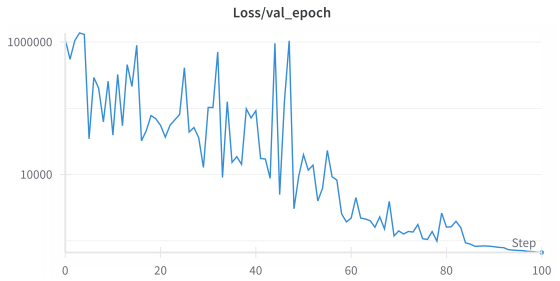
3 Analysis & Discussion

All experiments used the same configuration: 100 epochs; Adam optimizer; initial learning rate 1e-4 decayed to 1e-6 via a cosine LR scheduler; batch size of 10 (the largest feasible); KL weight ratio initialized at 0.5 with a cycle length of 10 epochs; teacher forcing warm-up start epoch 10 (`tfr_sde`=10) and initial ratio 0.95.

3.1 Teacher Forcing Ratio Curve



(a) Teacher forcing ratio per epoch

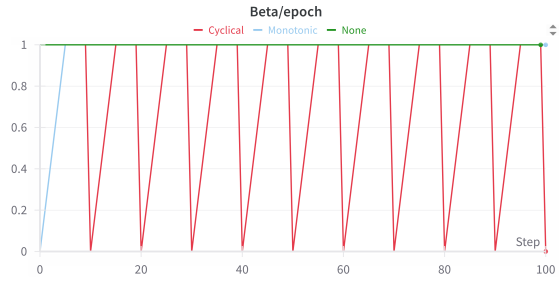


(b) Training loss

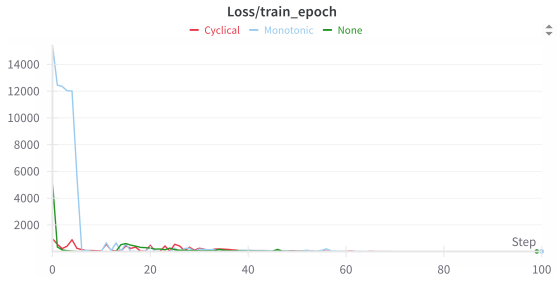
Figure 1: Teacher forcing ratio and training loss under monotonic decay ($d = 0.95$).

Early in training, the model remains heavily guided by teacher forcing and cannot yet learn to generate full video sequences on its own, resulting in very high loss values. As epochs progress and the model gradually masters autonomous frame prediction, the training loss declines steadily. However, since the teacher forcing ratio never reaches zero, occasional forced ground-truth inputs still occur and cause sudden spikes in the loss curve.

3.2 Validation Loss and PSNR



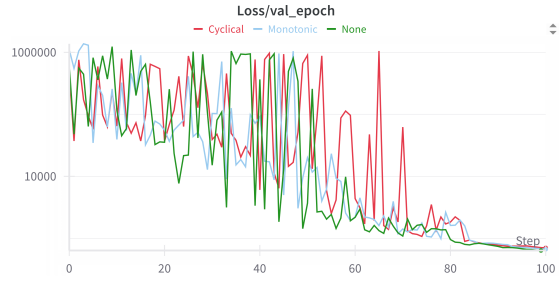
(a) KL weight $\beta(t)$ schedules



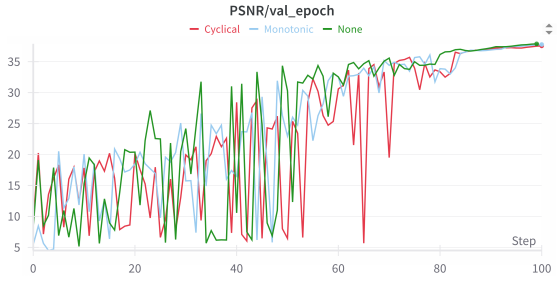
(b) Total training loss over epochs

Figure 2: KL weight schedules and corresponding training loss trajectories.

The KL weight schedules plot shows cyclical annealing oscillating between 0 and 1 every cycle, monotonic rising to 1 and staying there, and no annealing fixed at 1 throughout. The training loss trajectories are very similar across all strategies; although monotonic seems to converge slightly more slowly overall, its first 10 epochs match cyclical exactly, indicating these small differences may stem from experimental randomness rather than inherent strategy advantages.



(a) Validation loss over epochs

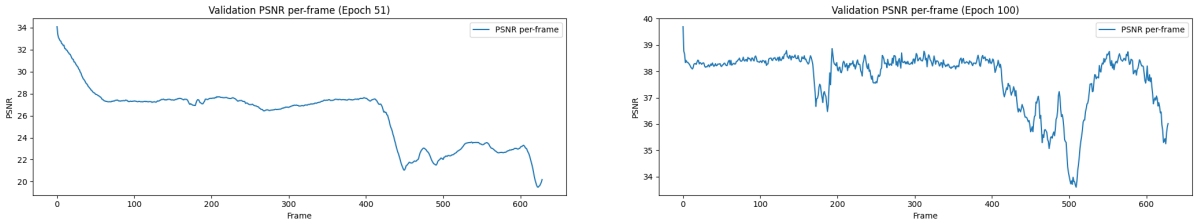


(b) Validation PSNR over epochs

Figure 3: Validation loss and PSNR under no annealing, monotonic, and cyclical KL schedules.

Monotonic and cyclical schedules both bring validation loss down more steadily than no annealing, though cyclical convergence is noticeably slower and oscillates due to its periodic ramping of the KL term. In the end all three strategies reach similar loss levels, but the large fluctuations in the cyclical curve suggest less stable gradient control. The "None" strategy shows very high variance early on, likely because the model never focuses exclusively on reconstruction loss before regularization. Based on these observations, monotonic annealing was chosen for the final submission.

3.3 Discussion of PSNR Trends



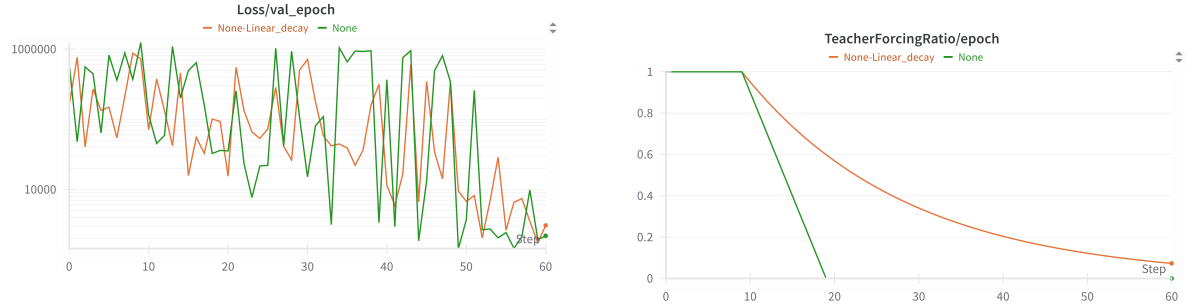
(a) Monotonic at epoch 50

(b) Monotonic at epoch 100

Figure 4: Per-frame PSNR on the validation set at epochs 50 and 100.

We recorded the PSNR for each predicted frame under the monotonic annealing schedule. During the early phase (around epoch 50), the model still relies heavily on teacher forcing and struggles to maintain PSNR, causing a rapid drop after the first few frames. By epoch 100, the generator has learned to produce more balanced predictions, resulting in a relatively stable PSNR curve with reduced variance across frames.

3.4 Other Training Strategy Analysis



(a) Validation loss under exponential and linear TF decay

(b) Teacher forcing ratio schedules

Figure 5: Exponential vs. linear decay of the teacher forcing ratio.

Two decay strategies for the teacher forcing ratio were compared: exponential and linear. With linear decay, the validation loss curve shows noticeably smaller fluctuations throughout training compared to exponential decay. Despite this improved stability, both schedules converge to similar final validation performance.

Additionally, I experimented with sampling the teacher forcing decision independently at each time step rather than once per epoch. This per-step randomization led to severely degraded training stability—validation loss diverged rapidly and gradients exploded. It appears the model cannot handle such rapid switching between ground-truth and generated inputs at the frame level.

References

[1] C. Chan *et al.*, "Everybody Dance Now," *ICCV*, 2019.

[2] E. Denton and L. Fergus, "Stochastic Video Generation with a Learned Prior," *ICML*, 2018.

[3] H. Fu *et al.*, "Cyclical Annealing Schedule: A Simple Approach to Mitigating KL Vanishing," *NAACL*, 2019.