Deep Learning Homework 4 Report

Cheng-Liang Chi

April 15, 2025

Contents

1 Introduction

In this lab, we focus on implementing **Conditional Variational Autoencoder (cVAE)** for the task of **video prediction**. The goal is to generate a sequence of future video frames, conditioned on a sequence of *pose images* (labels) and a *starting frame*. Specifically, given an initial frame x_1 and a set of pose conditions $\{P_2, \ldots, P_k\}$, the model generates frames $\{x_2, \ldots, x_k\}$ that follow the specified motion.

The architecture is based on a **VAE framework**, where a latent variable z is used to capture the stochastic aspects of motion. A posterior network (Gaussian Predictor) generates μ and $\log \sigma^2$ from the current frame and label, from which the latent code z is sampled using the *reparameterization trick*. The **generator** then uses the sampled latent variable, the previous frame, and the current pose label to generate the next frame.

The task is inspired by the ICCV 2019 paper *"Everybody Dance Now"* [1], which uses a GAN-based model for pose-to-frame generation, and the ICML 2018 paper *"Stochastic Video Generation with a Learned Prior"* [2], which utilizes a VAE model augmented with temporal modules like LSTMs. While the GAN-based approach emphasizes image quality, the VAE-based model allows for **diverse and temporally consistent frame generation**, which is suitable for this task.

To improve training and stability, techniques like **KL annealing** (both monotonic and cyclical) and **teacher forcing** are employed. These help in mitigating common issues in VAE training such as KL collapse and exposure bias in autoregressive generation.

In this report, we detail our implementation, training strategy, and analysis of the model's performance on video prediction, measured by *loss curves* and *PSNR per frame*.

2 Implementation Details

2.1 Training and Testing Protocol

During training, each sample consists of a sequence of k frames $\{x_1, x_2, \ldots, x_k\}$ and corresponding pose labels $\{P_1, P_2, \ldots, P_k\}$. The first frame x_1 is used as the starting point to predict the following $k-1$ frames. For each time step t, the Gaussian predictor takes (x_t, P_t) as input and outputs the parameters $(\mu_t, \log \sigma_t^2)$ of a Gaussian distribution. A latent variable z_t is then sampled using the reparameterization trick.

The decoder fuses the latent code z_t, the previous frame x_{t-1} (or the predicted $\hat{x}_{t-1}$), and the current label P_t to generate the current frame $\hat{x}_t$. A combination of mean squared error (MSE) and KL divergence is computed as the loss.

Teacher forcing is applied conditionally based on a decaying teacher forcing ratio (TFR). At inference time, the model uses $z_t \sim \mathcal{N}(0, I)$ instead of using the posterior predictor, and the previously generated frame is recursively fed to the generator.

```python
    def training_one_step(
        self, img, label, adapt_TeacherForcing: bool
    ) -> tuple[torch.Tensor, tuple[float, float, float]]:
        # image shape: [batch_size, video_len, channel, height, width]
        # label shape: [batch_size, video_len, channel, height, width]

        mse_loss = torch.zeros(1).to(self.args.device)
        kl_loss = torch.zeros(1).to(self.args.device)
```

```
        psnr = torch.zeros(1).to(self.args.device)

        prev_frame = img[:, 0, :, :, :].clone()
        for i in range(1, self.args.train_vi_len):
            trans_prev_frame = self.frame_transformation(prev_frame)
            trans_cur_frame = self.frame_transformation(img[:, i, :, :, :])
            trans_cur_label = self.label_transformation(label[:, i, :, :, :])
            z, mu, logvar = self.Gaussian_Predictor(trans_cur_frame, trans_cur_label)
            df_out = self.Decoder_Fusion(trans_prev_frame, trans_cur_label, z)
            pred_frame = self.Generator(df_out)

            kl_loss += kl_criterion(mu, logvar, self.batch_size)
            mse_loss += self.mse_criterion(pred_frame, img[:, i, :, :, :])
            psnr += Generate_PSNR(pred_frame, img[:, i, :, :, :], data_range=1.0)

            prev_frame = img[:, i, :, :, :] if adapt_TeacherForcing else pred_frame

        mse_loss /= self.args.train_vi_len - 1
        kl_loss /= self.args.train_vi_len - 1
        psnr /= self.args.train_vi_len - 1
        loss = mse_loss + self.kl_annealing.get_beta() * kl_loss

        self.optim.zero_grad()
        loss.backward()
        self.optimizer_step()

        return loss, (mse_loss.item(), kl_loss.item(), psnr.item())
```

```
    def training_stage(self):
        cnt = 0
        for _ in range(self.args.num_epoch):
            train_loader = self.train_dataloader()

            mse_all = 0.0
            kl_all = 0.0
            psnr_all = 0.0
            loss_all = 0.0

            for img, label in (pbar := tqdm(train_loader, dynamic_ncols=True)):
                adapt_TeacherForcing: bool = (
                    True if random.random() < self.tfr else False
                )
```

```python
                img = img.to(self.args.device)
                label = label.to(self.args.device)
                loss, (mse, kl, psnr) = self.training_one_step(
                    img, label, adapt_TeacherForcing
                )
```

2.2 Reparameterization Trick

To allow backpropagation through the stochastic latent variable, the reparameterization trick is used:

$$z = \mu + \epsilon \cdot \sigma \quad \text{where} \quad \epsilon \sim \mathcal{N}(0, 1)$$

The log variance is used during training for numerical stability. Below is the implementation:

```python
class Gaussian_Predictor(nn.Sequential):
    def __init__(self, in_chans=48, out_chans=96):
        super(Gaussian_Predictor, self).__init__(
            ResidualBlock(in_chans, out_chans // 4),
            DepthConvBlock(out_chans // 4, out_chans // 4),
            ResidualBlock(out_chans // 4, out_chans // 2),
            DepthConvBlock(out_chans // 2, out_chans // 2),
            ResidualBlock(out_chans // 2, out_chans),
            nn.LeakyReLU(True),
            nn.Conv2d(out_chans, out_chans * 2, kernel_size=1),
        )

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        # eps = Variable(std.data.new(std.size()).normal_())
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, img, label):  # type: ignore
        feature = torch.cat([img, label], dim=1)
        parm = super().forward(feature)
        mu, logvar = torch.chunk(parm, 2, dim=1)
        z = self.reparameterize(mu, logvar)

        return z, mu, logvar
```

2.3 Teacher Forcing Strategy

Teacher forcing is applied by choosing whether to use the ground truth frame x_{t-1} or the generated frame $\hat{x}_{t-1}$ as the next input, based on a probabilistic threshold set by TFR. The ratio starts high and decays over epochs.

```python
    def teacher_forcing_ratio_update(self):
        if self.tfr > 0.0:
            if self.current_epoch >= self.tfr_sde:
                self.tfr -= self.tfr_d_step
                if self.tfr < 0.0:
                    self.tfr = 0.0
            else:
                self.tfr = self.args.tfr
        else:
            self.tfr = 0.0
        self.tfr = max(self.tfr, 0.0)
        self.tfr = min(self.tfr, 1.0)
        return self.tfr
```

2.4 KL Annealing Strategy

KL annealing is used to gradually introduce the KL divergence loss term by multiplying it with a factor β. We support three modes:

- **Monotonic**: β increases linearly to 1 over the training schedule.
- **Cyclical**: β follows a repeated ramp-up schedule.
- **Without KL**: $\beta = 1$ throughout training.

This strategy helps to prevent KL collapse and stabilize training by allowing the model to learn useful latent codes before regularizing them. Our implementation is based on the example provided by Fu et al. [3], who introduced the cyclical annealing schedule to mitigate the vanishing KL problem in variational models.

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

        match self.anneal_type:
            case "constant":
                self.L = torch.ones(args.num_epoch)
            case "linear":
                self.L = torch.linspace(0, 1, args.num_epoch)
            case "cyclical":
                self.L = self.frange_cycle_linear(
                    args.num_epoch,
                    start=1e-12,
                    stop=1.0,
                    n_cycle=args.kl_anneal_cycle,
                    ratio=args.kl_anneal_ratio,
                )
        self.L = self.L.to(args.device)
```

```python

    def update(self):
        self.step += 1

    def get_beta(self):
        return self.L[self.step]

    def frange_cycle_linear(self, n_iter, start=0.0, stop=1.0, n_cycle=4, ratio=0.5):
        L = torch.ones(n_iter) * stop
        period = n_iter / n_cycle
        step = (stop - start) / (period * ratio)  # linear schedule

        for c in range(n_cycle):
            v, i = start, 0
            while v <= stop and (int(i + c * period) < n_iter):
                L[int(i + c * period)] = v
                v += step
                i += 1
        return L
```

3 Analysis & Discussion

3.1 Teacher Forcing Ratio

I experimented with different initial teacher forcing ratios (`tfr`) and decay strategies. The ratio controls the probability of using the ground truth frame x_{t-1} instead of the generated frame $\hat{x}_{t-1}$ during training. The configurations I tested include:

- **TF1:** `tfr = 1.0`, `tfr_sde = 0`, `tfr_d_step = 0.1`
- **TF2:** `tfr = 0.5`, `tfr_sde = 10`, `tfr_d_step = 0.05`
- **TF3:** `tfr = 0.0` (no teacher forcing)

These experiments were conducted with default parameters, including a batch size of 4 and a learning rate of 0.0001. The first configuration used a high initial TFR of 1.0, which decayed to 0.0 over 10 epochs. The second configuration started with a TFR of 0.5 and start decaying after 10 epochs, reaching 0.0 after 20 epochs. The third configuration used no teacher forcing, meaning the model always used its own predictions as input.

Figure 1 shows the decay of the TFR and its relationship to the training loss. I observed that high initial TFR led to faster convergence early on, but sometimes caused the model to overfit to teacher inputs. In contrast, training with `tfr = 0.0` converged more slowly initially, but achieved better consistency in frame generation.

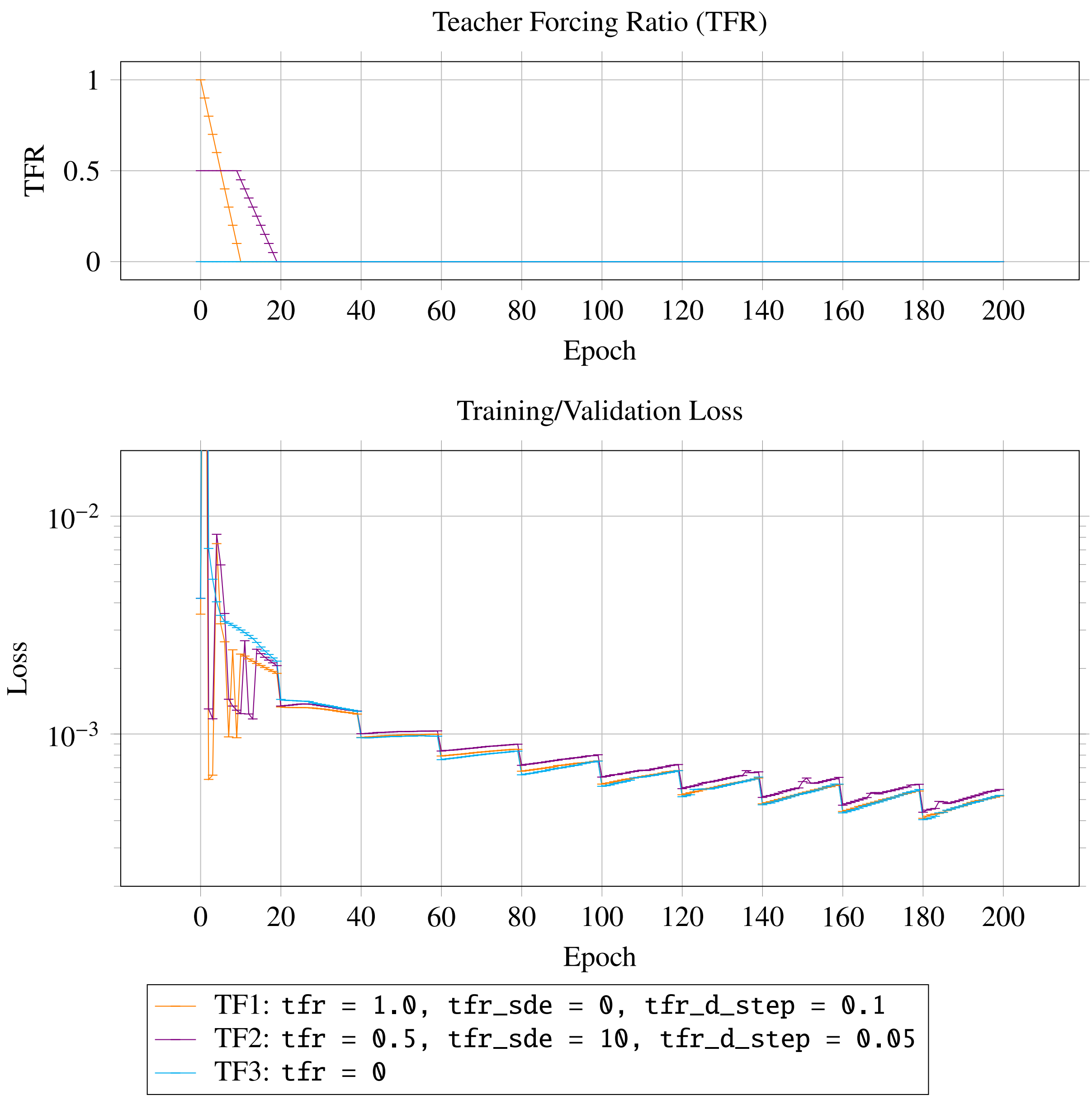

Figure 1: Teacher Forcing Ratio and Corresponding Loss

3.2 Loss Curve Under Different KL Annealing Strategies

I compared three KL annealing strategies:

- **KL1: Cyclical**: Repeated ramp-up and reset schedule
- **KL2: Monotonic**: Linearly increasing β
- **KL3: Constant**: No annealing; $\beta = 1$ throughout
- **KL4: Cyclical R0.5**: Cyclical with `kl_anneal_ratio` = 0.5

These experiments were conducted with a batch size of 4 and a learning rate of 0.0001. The parameters for the cyclical schedule were `kl_anneal_ratio` = 1 and `kl_anneal_cycle` = 10. And the parameters for cyclical R0.5 schedule were `kl_anneal_ratio` = 0.5 and `kl_anneal_cycle` = 10.

Figure 2 shows the training loss curves. As expected, the `monotonic` and `cyclical` schedules provided more stable training in the early epochs. Without KL annealing, the KL term dominated early, leading to unstable gradients and degraded frame quality. Among the three, the `cyclical` method—based on Fu et al. [3]—produced the best overall convergence and avoided KL collapse.

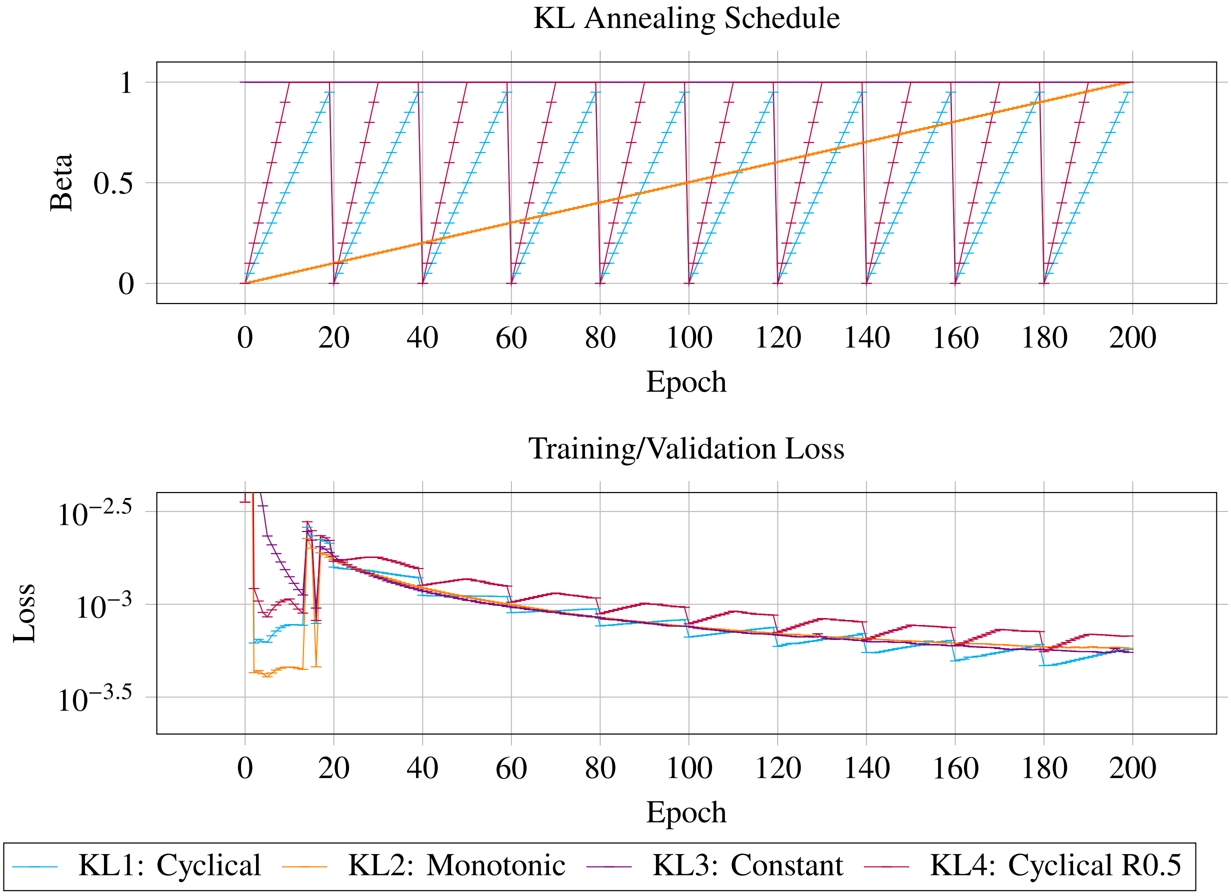

Figure 2: Training Loss with Different KL Annealing Strategies

3.3 PSNR-per-Frame in Validation Set

To evaluate the quality of the generated frames, I computed the PSNR between each predicted frame and its ground truth counterpart on the validation set. Figure 3 shows the per-frame PSNR for the different KL annealing strategies.

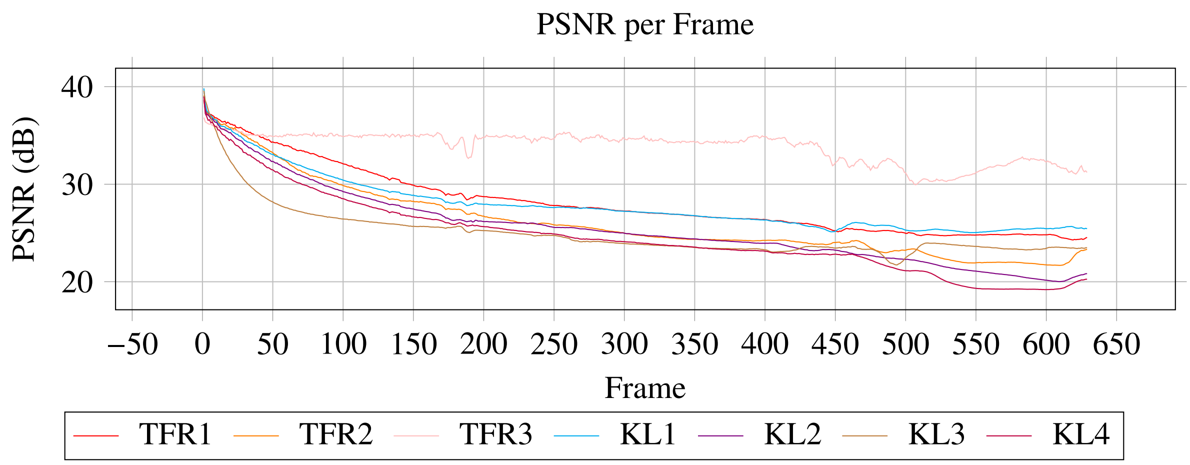

Figure 3: PSNR per Frame in Validation Set. (a.) TFR1 is the configuration with `tfr = 1.0, tfr_sde = 0, tfr_d_step = 0.1`, it has 28.10 dB average PSNR. (b.) TFR2 is the configuration with `tfr = 0.5, tfr_sde = 10, tfr_d_step = 0.05`, it has 26.15 dB average PSNR. (c.) TFR3 is the configuration with `tfr = 0.0`, it has 33.83 dB average PSNR. (d.) KL1 is the configuration with cyclical KL schedule, it has 27.85 dB average PSNR. (e.) KL2 is the configuration with linear KL schedule, it has 25.48 dB average PSNR. (f.) KL3 is the configuration with constant KL schedule, it has 24.95 dB average PSNR. (g.) KL4 is the configuration with cyclical KL schedule with `kl_anneal_ratio = 0.5`, it has 24.65 dB average PSNR.

The results showed that PSNR typically remained stable in the early frames but dropped significantly after around frame 400. The model trained with a monotonic KL schedule achieved the highest average PSNR in early frames, while the cyclical schedule produced more stable long-term quality.

3.4 Other Training Strategy Analysis (Bonus)

In addition to the main training configurations, I explored several auxiliary strategies:

- **Data Augmentation:** I applied RandomResizeCrop and RandomHorizontalFlip to both images and labels among all the images in the dataset. This helped to improve the model's robustness and generalization. The implementation are shown bellow:

```
    def __getitem__(self, index):
        path = self.img_folder[index]
        imgs = []
        labels = []
        for i in range(self.video_len):
            label_list = self.img_folder[(index * self.video_len) +
            ↪  i].split("/")
            label_list[-2] = self.prefix + "_label"

            img_name = self.img_folder[(index * self.video_len) + i]
            label_name = "/".join(label_list)
            img, label = imgloader(img_name), imgloader(label_name)
            img = v2.functional.to_image(img)
```

```
            label = v2.functional.to_image(label)

            imgs.append(self.to_tensor(img))
            labels.append(self.to_tensor(label))

        transformed = self.transform(*imgs, *labels)
        imgs = transformed[: self.video_len]
        labels = transformed[self.video_len :]
        return stack(imgs), stack(labels)
```

```
    def train_dataloader(self):
        transform = v2.Compose(
            [
                v2.RandomResizedCrop((self.args.frame_H, self.args.frame_W)),
                # v2.Resize((self.args.frame_H, self.args.frame_W)),
                v2.RandomHorizontalFlip(p=0.5),
            ]
        )

        dataset = Dataset_Dance(
            root=self.args.DR,
            transform=transform,
            mode="train",
            video_len=self.train_vi_len,
            partial=args.fast_partial if self.args.fast_train else args.partial,
        )
        if self.current_epoch > self.args.fast_train_epoch:
            self.args.fast_train = False

        train_loader = DataLoader(
            dataset,
            batch_size=self.batch_size,
            num_workers=self.args.num_workers,
            drop_last=True,
            shuffle=False,
        )
        return train_loader
```

- **Optimizers:** I compared `Adam` and `AdamW`. The latter improved regularization and resulted in lower validation loss. The implementation is shown below:
- **Schedulers:** I tested `MultiStepLR` and `CosineAnnealing`.

```python
        match self.args.optim:
            case "Adam":
                self.optim = optim.Adam(self.parameters(), lr=self.args.lr)
                self.scheduler = optim.lr_scheduler.MultiStepLR(
                    self.optim, milestones=[2, 5], gamma=0.1
                )
            case "AdamW":
                self.optim = optim.AdamW(
                    self.parameters(),
                    lr=self.args.lr,
                    betas=(0.9, 0.999),
                    eps=1e-8,
                    weight_decay=0.01,
                )
                self.scheduler = optim.lr_scheduler.CosineAnnealingLR(
                    self.optim,
                    T_max=self.args.num_epoch,
                    eta_min=0,
                )
            case _:
                raise ValueError(f"Unknown optimizer: {self.args.optim}")
```

Overall, the best results were achieved using the `AdamW` optimizer, cyclical KL annealing, and moderate teacher forcing rate (about 0.5). Since the experiments were too numerous to cover in detail, I focused on the most significant findings. The data augmentation and the choice of optimizer and scheduler played a crucial role in improving the model's performance. The cyclical KL annealing strategy helped to stabilize training and prevent KL collapse, while the moderate teacher forcing rate allowed for a balance between using ground truth and generated frames during training. The combination of these strategies led to improved convergence and frame generation quality.

References

[1] C. Chan, S. Ginosar, T. Zhou, and A. A. Efros, "Everybody dance now," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[2] E. Denton and R. Fergus, "Stochastic video generation with a learned prior," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[3] H. Fu, C. Li, X. Liu, J. Gao, A. Celikyilmaz, and L. Carin, "Cyclical annealing schedule: A simple approach to mitigating KL vanishing," in *NAACL*, 2019.

[4] Hank891008, *Deep-learning*, GitHub repository, 2023. [Online]. Available: `https://github.com/hank891008/Deep-Learning`.

[5] yeeecheng, *Nycu_deeplearing2024*, GitHub repository, 2024. [Online]. Available: `https://github.com/yeeecheng/NYCU_DeepLearing2024`.