For a better reading experience, please visit my HackMD

Lab4 Conditional VAE for video prediction

311554046 □□□

1. Introduction

An VAE model for video prediction based on **SVG-FP** model (**Emily Denton & Rob Fergus, 2018**), which the model architecture is as the figure shown below:

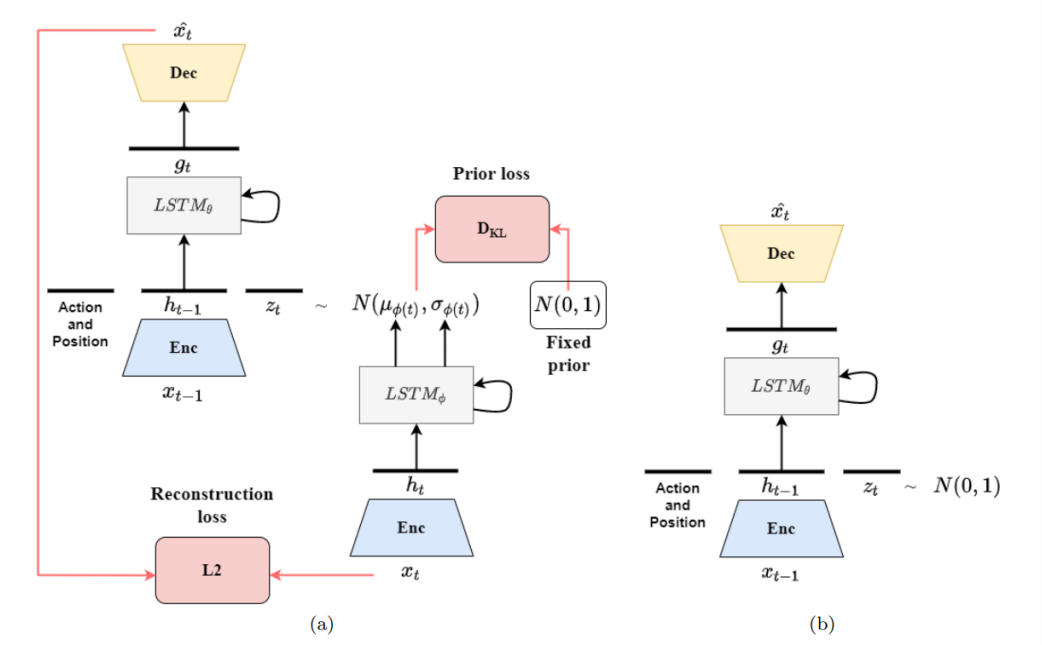


Figure 1: The illustration of overall framework. (a) Training procedure (b) Generation procedure

> This model take past frame x_{t-1} as input for encoder, which will output an latent vector h_{t-1}, and we will sample z_t from the posterior we learned. Eventually, we take h_{t-1} and z_t with action and position (the condition) as the input for decoder, which the output is expected to be the next frame $\hat{x_t}$.

We need to implement this VAE model. Furthermore, the **action and position data** are used as the condition to make this an CVAE model, **Teacher forcing** and **KL annealing schedules** are required in the training function.
Once the model finish training, we test it out by making gif image and prediction at each time step to see if the model prediction is close to the ground-truth image.

> Note that in the testing phase, we sample z_t from fixed prior, which we assumed here is a **standard normal distribution**.

2. Derivation of CVAE

The derivation of the CVAE objective function:
To determine θ, we would intuitively hope to maximize $p(X|c;\theta)$

$$p(X|c;\theta) = \int p(X|Z,c;\theta)p(Z|c)dZ$$

This however becomes difficult since the integration over Z would have no closed-form solution when $p(X|Z,c;\theta)$ is modeled by a neural network.

By the chain rule of probability, we know:

$$p(X,Z|c;\theta) = p(X|c;\theta)p(Z|X,c;\theta)$$

$$\Longrightarrow \log p(X,Z|c;\theta) = \log p(X|c;\theta) + \log p(Z|X,c;\theta)$$

$$\Longrightarrow \log p(X|c;\theta) = \log p(X,Z|c;\theta) - \log p(Z|X,c;\theta)$$

We next introduce an arbitrary distribution $q(Z)$ on both side and integrate over Z:

$$\int q(Z)\log p(X|c;\theta)dZ = \int q(Z)\log p(X,Z|c;\theta)dZ - \int q(Z)\log p(Z|X,c;\theta)dZ$$

$$= \int q(Z)\log p(X,Z|c;\theta)dZ - \int q(Z)\log q(Z)dZ + \int q(Z)\log q(Z)dZ - \int q(Z)\log p(Z|X,c;\theta)dZ$$

$$= L(X,c,q,\theta) + KL(q(Z)||(Z|X,c;\theta))$$

where

$$L(X,c,q,\theta) = \int q(Z)\log p(X,Z|c;\theta)dZ - \int q(Z)\log q(Z)dZ$$

$$KL(q(Z)||(Z|X,c;\theta)) = \int q(Z)\log q(Z)dZ - \int q(Z)\log p(Z|X,c;\theta)dZ$$

$$\therefore \log p(X|c;\theta) = L(X,c,q,\theta) + KL(q(Z)||p(Z|X,c;\theta))$$

$$\Longrightarrow L(X,c,q,\theta) = \log p(X|c;\theta) - KL(q(Z)||p(Z|X,c;\theta))$$

Now, instead of directly maximizing the intractable $p(X|c;\theta)$, we attempt to maximize $ L (X,c,q,\theta)$

$$\because L(X,c,q,\theta) = \int q(Z)\log p(X,Z|c;\theta)dZ - \int q(Z)\log q(Z)dZ$$

$$= E_{Z\sim q(Z)}[\log p(X,Z|c;\theta)] - E_{Z\sim q(Z)}[\log q(z)]$$

$$= E_{Z\sim q(Z)}[\log p(X|Z,c;\theta)] + E_{Z\sim q(Z)}[\log p(Z|c)] - E_{Z\sim q(Z)}[\log q(z)]$$

$$= E_{Z\sim q(Z)}[\log p(X|Z,c;\theta)] - KL(q(Z)||p(Z|c))$$

$$\therefore L(X,c,q,\theta) = E_{Z\sim q(Z)}[\log p(X|Z,c;\theta)] - KL(q(Z)||p(Z|c))$$

Because the equality holds for any choice of $q(Z)$, we introduce a distribution $q(Z|X;\phi)$ modeled by another neural network with parameter ϕ.

To maximize

$$L(X,c,q,\theta,\phi) = \log p(X|c;\theta) - KL(q(Z|X;\phi)||p(Z|X,c;\theta))$$

which amounts to maximizing

$$E_{Z\sim q(Z|X;\phi)}[\log p(X|Z,c;\theta)] - KL(q(Z|X;\phi)||p(Z|c))$$

3. Implementation details

1. Model implementation

a. Encoder

The frame encoder uses the same architecture as **VGG16** (**Simonyan & Zisserman, 2015**) up until the fourth pooling layer and the final convulutional layer contain a **conv4-128** with no padding, besides, the activation function is replaced with *Tanh*, as the figure shown below.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

```
# Final layer
self.c5 = nn.Sequential(
        nn.Conv2d(512, dim, 4, 1, 0),
        nn.BatchNorm2d(dim),
        nn.Tanh()
        )
# Maxpool
self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

The input *x* shape here is **(*b*, 3, 64, 64)**; as for the output, the latent vector *h* shape is **(*b*, 128, 1, 1)**.

> *b* represents the batch size.

b. Decoder

The decoder is basically a mirrored version of the encoder with pooling layer replaced with **spatial up-sampling** and a **sigmoid** output layer, as the figure shown below.

```
self.c5 = nn.Sequential(
        nn.Conv2d(512, dim, 4, 1, 0),
        nn.BatchNorm2d(dim),
        nn.Tanh()
        )
self.mp = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

The input g shape here is **(b, 128, 1, 1)** , which is as same as the output shape of the encoder; the output $\hat{x}$ shape is **(b, 3, 64, 64)**, which is as same as the input shape of the encoder.

c. Reparameterization trick

Since the LSTM in the inference model is expected to output the mean and variance of a Gaussian distribution, where z is sampled from. This however is *non-differientiable*, that is, the model could not be trained **end-to-end**.
In order to solve this problem, we need to implement the so-called *"reparameterization trick"*, which simply drawn a ϵ from standard normal distribution, and the operation is as follow:

$$\epsilon \sim N(0,1)$$

$$z = \epsilon * \sigma + \mu$$

so that

$$z \sim N(\mu, \sigma)$$

where μ and σ are the output of the LSTM in the inference model.

```
def reparameterize(self, mu, logvar):
    logvar = logvar.mul(0.5).exp_()
    eps = logvar.detach().new(logvar.size()).normal_()
    return eps.mul(logvar).add_(mu)
```

d. Dataloader

The dataloader loads a batch of the image sequence data (.png) and the condition data (.csv) every time.

For the image, we transforms it into a **tensor**, which would turn it into a **floating point (0~1)**. By concatenating all the image in every time step, we get a image sequence with the shape of **(*t*, 3, 64, 64)**.

> *t* represents the total time step for a image sequence.

```
default_transform = transforms.Compose([
    transforms.ToTensor(),
    ])
```

The condition data is composed of action (dimensionality 4) and position data (dimentionality 3). By concatenating them in every time step, we get a condition with the shape of **(*t*, 7)**.
Since the encoder input shape is **(*b*, 3, 64, 64)**, we need to **transpose** the image sequence between dimension 0 (time step) and dimension 1 (batch), so that the shape would be **(*t*, *b*, 3, 64, 64)**. The condition data should also do the same thing, because we then directly concatenate it with the output *h* from the encoder and *z* from inference model, which is soon to be sended into the LSTM in prediction model.

```
seq, cond = seq.transpose_(0, 1).to(device), cond.transpose_(0, 1).to(device)
```

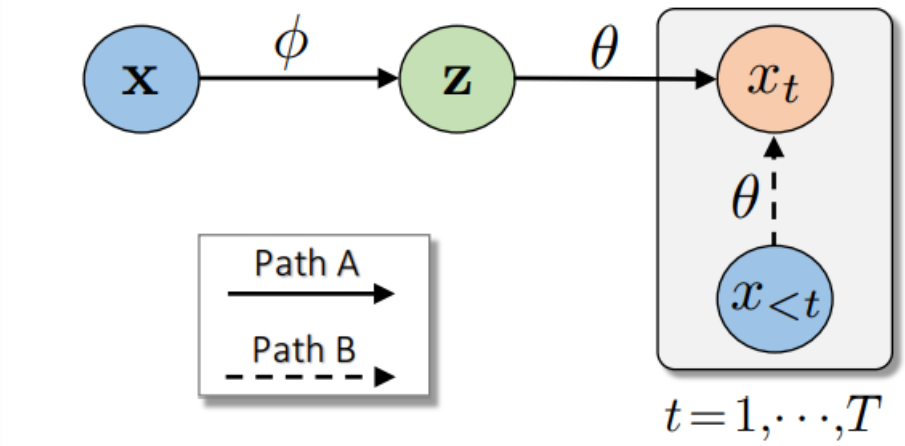
e. Condition

As mentioned above, I simply **concatenates** the condition data with *h* and *z*, as the input for the LSTM in prediction model.

```
h_pred = modules['frame_predictor'](torch.cat([cond[i], h, z_t], 1))
```

f. KL annealing schedules

Due to the sequential nature of text and video, an auto regressive decoder is typically employed in the VAE. This is often implemented with a RNN(LSTM). This introduces one notorious issue when a VAE is trained using traditional methods: the decoder ignores the latent variable, yielding what is termed the ***KL vanishing problem***.
Fu, Li & Liu et al. (2019) hypothesize that the problem is related to the low quality of *z* at the beginning phase of decoder training. A lower quality *z* introduces more difficulties in reconstructing *x* via Path A. As a result, the model is forced to learn an easier solution to decoding: generating *x* via Path B only, as the figure shown below:



(b) VAE with an auto-regressive decoder

It is natural to extend the negative of the objective function in VAE by intoducing a hyperparameter β to control the strength of regularization:

$$-E_{Z\sim q(Z|X;\phi)}[\log p(X|Z,c;\theta)] + \beta KL(q(Z|X;\phi)||p(Z|c))$$

three different schedules for β have been commonly used for VAE.

- Constant schedule
  The standard approach is to keep $\beta = 1$ fixed during training procedure, as it correspondes to optimizing the true VAE objective.
- Monotonic annealing schedules
  $\beta = 0$ is set at the beginning of training, and gradually increases β until $\beta = 1$ is reached.
- Cyclical annealing schedule
  Split the training process into M cycles, each starting with $\beta = 0$ and ending with $\beta = 1$.

```
class kl_annealing():
    def __init__(self, args):
        if not args.kl_anneal_cyclical:
            args.kl_anneal_cycle = 1
        period = args.niter / args.kl_anneal_cycle
        step = 1. / (period * args.kl_anneal_ratio)
        self.L = np.ones(args.niter)
        self.idx = 0

        for c in range(args.kl_anneal_cycle):
            v, i = 0.0, 0
            while v <= 1.0 and (int(i+c*period) < args.niter):
                self.L[int(i+c*period)] = v
                v += step
                i += 1

    def update(self):
        if self.idx < len(self.L)-1:
            self.idx += 1

    def get_beta(self):
        beta = self.L[self.idx]
        return beta
```

This class would construct an instance with variable which is a list contains the kl annealing weight in every epoch, and update function updates the index variable, which the get_beta() function uses as the index to access the beta in the list.

2. Teacher forcing

In the early phase of the training process, The model capability of prediction is low, if one unit output an bad result, it would definitely affect the learning of the following units.
Teacher forcing is a method for efficiently training RNN models, which uses the ground-truth as input, instead of model output from a prior time step as input. By using teacher forcing, the model would converge in the earlier iteration, moreover, the training is more stable than using free-running.

Since this method highly depends on the ground-truth label, it performs

better during the training process. But in the testing phase, without the support of the ground-truth, terrible results may probably be seen if the train data and test data are very different from each other, that is, the cross-domain capability of model would be weak. In order to solve this probelem, one way is to use so-called *curriculum learing*. Instead of using teacher forcing during the entire training process, we use *teacher forcing ratio*, which is the probability of using teacher forcing. Teacher forcing ratio will change over time, in the beginning of the training process, teacher forcing ratio is set close to 1, and gradually reduces the use of groud-truth by decreasing teacher forcing ratio.

```
args.tfr_decay_step = 1. / (args.niter - args.tfr_start_decay_epoch - 1)
if epoch >= args.tfr_start_decay_epoch:
  ### Update teacher forcing ratio ###
  args.tfr -= args.tfr_decay_step
  if args.tfr < args.tfr_lower_bound:
      args.tfr = args.tfr_lower_bound
```

4. Results and discussion

1. Results of video prediction

> The monotonic schedule result here I used is the best result of many experiments. Actually, most of the time in my experiment, monotonic schedule results in a very terrible training of the model.

a. Make videos or gif images for test result

- Cyclical schedule
  fp cyclical video
- Monotonic schedule
  fp monotonic video
- Cyclical schedule (learned prior)
  lp cyclical video

> The ground-truth is on the left hand side and the model prediction is on the right hand side.

b. Output the prediction at each time step

- Ground truth

- Cyclical schedule

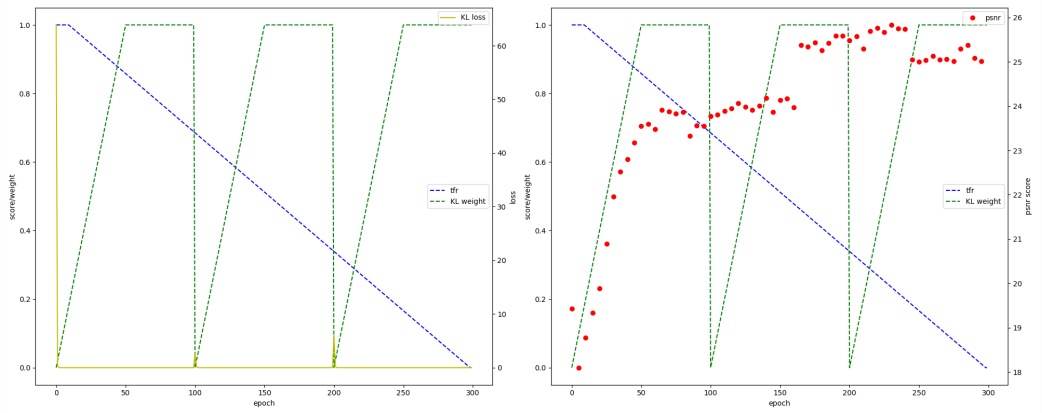
- Monotonic schedule

- Cyclical schedule (learned prior)

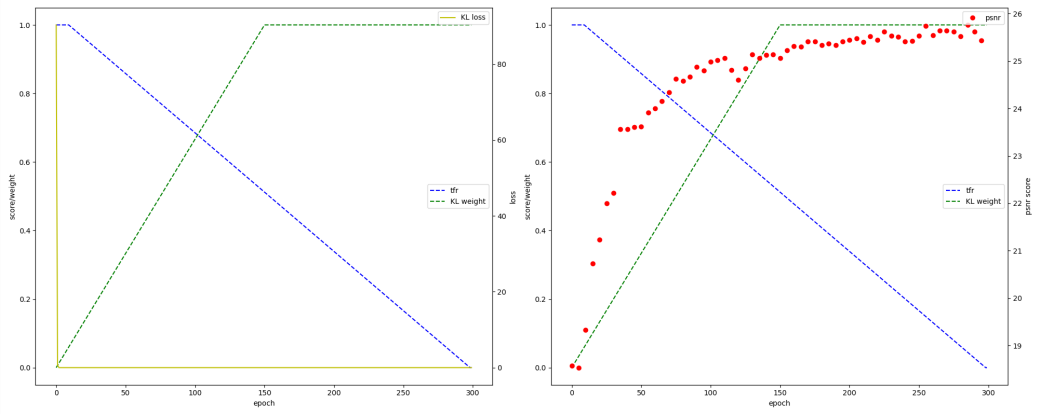
> Note that we use past two frames to predict the frames in the future, so the first two frames in ground truth and prediction are exactly the same.

2. Plot the KL loss and PSNR curves during training

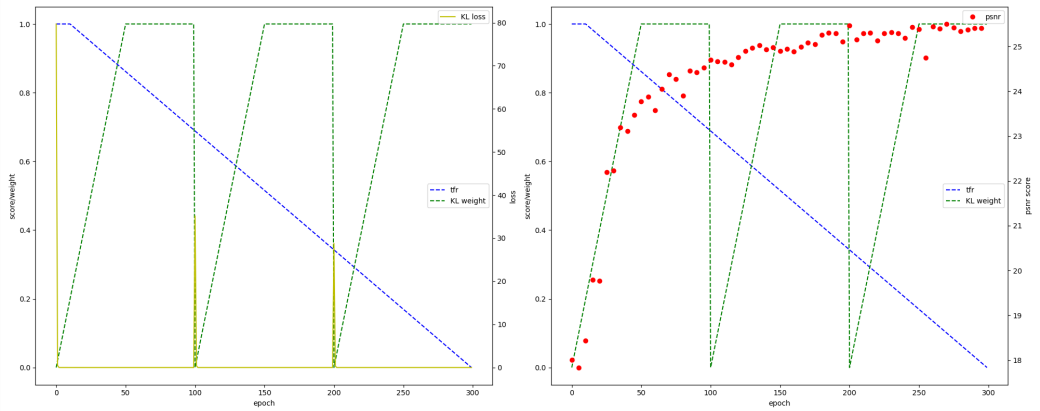
a. Cyclical schedule



b. Monotonic schedule



c. Cyclical schedule (learned prior)



3. Discuss the results according to your setting of teacher forcing ratio, KL weight, and learning rate.

- Teacher forcing ratio

  > In my setting, teacher forcing ratio starts at 1.0, after 11 epochs, it starts to decrease linearly to 0.

  As the curve shown above, in the early stage of training process, it's very helpful for the frame predictor to learn more from the ground truth and converge faster. Further, the decreasing of teacher forcing ratio forces the frame predictor to try to correct the wrong prediction itself, avoiding *"overcorrect"* problem.
  With this setting, the training process becomes more stable

- KL weight

  > In my setting, the number of cycle in cyclical schedule is set to

> be 3. And the KL annealing ratio is set to be 0.5.

In comparison with cyclical schedule, monotonic schedule seems to be more unstable during the training process, it happens that I try several times to train the monotonic one, but most of the results are extremely bad, the validate PSNR drops to a very low value (below 10), loss value is larger than the early phase of training and the generated images is as follow:
monotonic fail video
It seems like the model has totally broken, according to Fu, Li & Liu et al. (2019), this may result from underestimating of prior regularization and the posterior $q(Z|X;\phi)$ collapses into a point estimate, causing sub-optimal decoder learning.
Cyclical schedule here contains two consecutive stages within a cycle:

- Annealing

β is annealed from 0 to 1, z is forced to learn the global representation of x. By gradually increasing β towards 1, $q(Z|X;\phi)$ is regularized to transit from a point estimate to a distribution estimate, spreading out to match the prior.

- Fixing
  We fix $\beta = 1$ for the rest of training steps within one cycle. This drives the model to optimize the full VAE objective until converge.

As the curve plot shown above, KL loss in the cyclical schedule goes up when the KL weight decrease to 0, and immediately drops to lower value, leaving two spikes on the kl loss curve.

- Learning rate

> In my setting, learning rate is set to be 0.002 and using Adam optimizer with momentum term = 0.9.

The learning rate affects the time used for model to converge. If the value of learning rate is too large, it would lead to the model cannot converge; if it's too low, it would lead to very slow convergence.
In my experiments, it seems like the default (0.002) is a good choice for the initial value of learning rate, with Adam optimizer, the loss converge very fast, it takes about 150 epochs to approach the loss

value about 0.002, with the value of validate PSNR is 25.