Summer 2022 Deep Learning

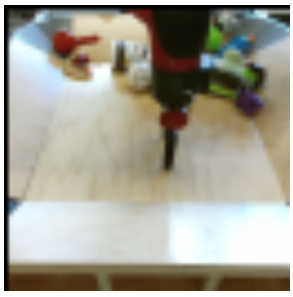
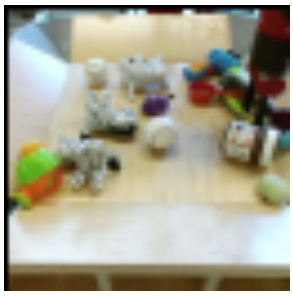
Report of Lab #4

Student name: 劉子齊 Jonathan
Student ID: 311605004

Part 1: Introduction

In lab4, our goal is to implement a Conditional Variational Autoencoder, a.k.a. CVAE, for video prediction. To evaluate the predicted video, we will make use of the Peak signal-to-noise ratio to measure the perceptual quality. Besides, we need to adopt teacher forcing and kl cost annealing during the training process. As each epoch came to an end, we will output a gif image with each frames predicted.

For the dataset, we got 168 sets of datasets for the training data, each dataset contained over 44, 000 sequences of datasets, and each gif had 30 frames, which was saved as png files. Also, we got 2 csv files for each gif, which recorded the action and the final position of the end effector. Below is how a frame of the dataset looked like.

Part 2: Derivation of CVAE

Conditional Variational Autoencoder is an extension of Variational Autoencoder. In the VAE, we have no control on the data generation process. Hence, by inputting observations modulate the prior on Gaussian latent variables that generate the output, we could solve the problem to some extent. The following figure below is the derivation of CVAE.

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

$$\Rightarrow \int \underbrace{q(x|c)}_{\text{any distribution of probability}} \cdot \log P(X|C)\cdot dx$$

$$= \int g(x|c)\cdot \log P(X,z|C)\cdot dz - \int g(x|c)\cdot \log P(z|X,C)\cdot dz$$

$$= \underbrace{\int q(z|c)\cdot \log P(X,z|C)\,dz - \int g(z|c)\cdot \log q(z|c)\cdot dz}_{\mathcal{L}} + \underbrace{\int q(z|c)\cdot \log g(z|c)\cdot dz - \int q(x|c)\cdot \log P(z|X,C)\cdot dz}_{KL}$$

$$= \mathcal{L}(X,q,\theta) + KL\left(q(z,C)\,\|\,P(z|X,C)\right)$$

$$\Rightarrow \int q(z|C)\cdot \log P(X|C)\cdot dz$$

$$= \log P(X|C)\cdot \int g(z|c)\cdot dz = \log P(X|C) \Rightarrow \log P(X|C) \geq \mathcal{L}(X,q,\theta)$$

$$\mathcal{L} = \int q(z|c)\cdot \log P(X,z|C)\cdot dz - \int g(z|c)\cdot \log q(z|c)\cdot dz$$

$$= \int q(z|c)\cdot \log P(X|z,C)\cdot dz + \int q(z|c)\cdot \log P(z|c)\cdot dz - \int q(z|c)\cdot \log q(z|c)\cdot dz$$

By changing $q(z|c)$ into $q(z|X,c;\theta')$

$$\Rightarrow \mathcal{L} = E_{z\sim q(z|X,C;\theta')}\cdot \log P(X|z,C;\theta) + E_{z\sim q(z|X,C;\theta')}\cdot P(z|C;\theta) - E_{z\sim q(z|X,C;\theta')}\cdot q(z|X,C;\theta')$$

$$= E_{z\sim q(z|X,C;\theta')}\cdot \log P(X|z,C;\theta) - KL\left(q(z|X,C;\theta')\,\|\,P(z,C;\theta)\right)$$

Part 3: Implementation Details

Part 3-A: Model Architecture

Part 3-A-1 Encoder

In this lab, we need to make prediction through CVAE. Different to VAE, CVAE's encoder generates latent vector z by fed x and decoder generate target output y by fed z. Here I took the VGGnet, which was recommended by the TA, as the model applied. Below is the implementation of the encoder, as an encoder, I lower the size of the input data from 64*64 to 4*4 by the "vgg_layer" with different layer sizes, which contained a Conv2d layer, a BatchNorm2d layer, and a LeakyReLU layer. The implementation of the VGG Layer is also shown below.

```
class vgg_layer(nn.Module):
    def __init__(self, nin, nout):
        super(vgg_layer, self).__init__()
        self.main = nn.Sequential(
                nn.Conv2d(nin, nout, 3, 1, 1),
                nn.BatchNorm2d(nout),
                nn.LeakyReLU(0.2, inplace=True)
                )

    def forward(self, input):
        return self.main(input)
```

```
class vgg_encoder(nn.Module):
    def __init__(self, dim):
        super(vgg_encoder, self).__init__()
        self.dim = dim
        # 64 x 64
        self.c1 = nn.Sequential(
                vgg_layer(3, 64),
                vgg_layer(64, 64),
                )
        # 32 x 32
        self.c2 = nn.Sequential(
                vgg_layer(64, 128),
                vgg_layer(128, 128),
                )
        # 16 x 16
        self.c3 = nn.Sequential(
                vgg_layer(128, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 256),
                )
        # 8 x 8
        self.c4 = nn.Sequential(
                vgg_layer(256, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 512),
                )
```

```
        # 4 x 4
        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)

    def forward(self, input):
        input = input.to("cuda:0")
        h1 = self.c1(input) # 64 -> 32
        h2 = self.c2(self.mp(h1)) # 32 -> 16
        h3 = self.c3(self.mp(h2)) # 16 -> 8
        h4 = self.c4(self.mp(h3)) # 8 -> 4
        h5 = self.c5(self.mp(h4)) # 4 -> 1
        return h5.view(-1, self.dim), [h1, h2, h3, h4]
```

Part 3-A-2 Decoder

Like the encoder, the decoder was also based on the VGGnet. Different to VAE, CVAE's decoder decode latent vector z and condition c then get output. Below is the implementation of the decoder. In the decoder, I return the size from 1*1 all the way back to 64*64 through different layer sizes of "vgg_layer".

```
class vgg_decoder(nn.Module):
    def __init__(self, dim):
        super(vgg_decoder, self).__init__()
        self.dim = dim
        # 1 x 1 -> 4 x 4
        self.upc1 = nn.Sequential(
                nn.ConvTranspose2d(dim, 512, 4, 1, 0),
                nn.BatchNorm2d(512),
                nn.LeakyReLU(0.2, inplace=True)
                )
        # 8 x 8
        self.upc2 = nn.Sequential(
                vgg_layer(512*2, 512),
                vgg_layer(512, 512),
                vgg_layer(512, 256)
                )
        # 16 x 16
        self.upc3 = nn.Sequential(
                vgg_layer(256*2, 256),
                vgg_layer(256, 256),
                vgg_layer(256, 128)
                )
        # 32 x 32
        self.upc4 = nn.Sequential(
                vgg_layer(128*2, 128),
                vgg_layer(128, 64)
                )
        # 64 x 64
        self.upc5 = nn.Sequential(
                vgg_layer(64*2, 64),
                nn.ConvTranspose2d(64, 3, 3, 1, 1),
                nn.Sigmoid()
                )
        self.up = nn.UpsamplingNearest2d(scale_factor=2)
```

```
def forward(self, input):
    vec, skip = input
    d1 = self.upc1(vec.view(-1, self.dim, 1, 1)) # 1 -> 4
    up1 = self.up(d1) # 4 -> 8
    d2 = self.upc2(torch.cat([up1, skip[3]], 1)) # 8 x 8
    up2 = self.up(d2) # 8 -> 16
    d3 = self.upc3(torch.cat([up2, skip[2]], 1)) # 16 x 16
    up3 = self.up(d3) # 8 -> 32
    d4 = self.upc4(torch.cat([up3, skip[1]], 1)) # 32 x 32
    up4 = self.up(d4) # 32 -> 64
    output = self.upc5(torch.cat([up4, skip[0]], 1)) # 64 x 64
    return output
```

Part 3-A-3 Reparameterization Trick

In the CVAE, our data flow will pass through encoder and decoder, this helps a lot to the model during the training process. However, we couldn't calculate the gradient for the parameters of the encoder through the distribution. In other word, we couldn't train the encoder through the loss obtained from the decoder. Hence, we will need to adopt the reparameterization trick. By passing "mu" and "logvar" into the function below, we would obtain a z, which is the reparametrized z. Also, in the figure below was how I implemented the reparameterization trick.

$$z = mu + rand\_like(\exp\left(\frac{logvar}{2}\right))$$

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

Part 3-A-4 KL Loss & KL Weight Annealing

Since general loss functions mainly focus on the accuracy of the output, they can hardly tell that whether vector z followed the rule of Gaussian Normal Distribution. Hence, we would need to apply KL Loss here. The following is the derivation and the code implementation of KL Loss.

$$KL(N(u,\sigma^2)\|N(0,1))$$
$$= \int \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-u)^2}{2\sigma^2}} \left(\log \frac{e^{-\frac{(x-u)^2}{2\sigma^2}}/\sqrt{2\pi\sigma^2}}{e^{-\frac{x^2}{2}}/\sqrt{2\pi}} \right) \cdot dx$$
$$= \int \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-u)^2}{2\sigma^2}} \cdot \log\left(\frac{1}{\sqrt{\sigma^2}} \cdot \exp\frac{1}{2}\left(x^2 - \frac{(x-u)^2}{\sigma^2}\right) \right) \cdot dx$$
$$= \int \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-u)^2}{2\sigma^2}} \cdot \frac{1}{2} \cdot \left(-\log\sigma^2 + x^2 - \frac{(x-u)^2}{\sigma^2} \right) \cdot dx$$
$$= \frac{1}{2}\left(-\log\sigma^2 + u^2 + \sigma^2 - 1\right) \#$$

```
def kl_criterion(mu, logvar, args):
  # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
  KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
  KLD /= args.batch_size
  return KLD
```

As the training process proceeded, the KL loss would start to get smaller and smaller. If we kept using the same loss function, which " β " is 1, as the following equation.

$$Loss = MSE_{Loss} + \beta * KL_{Loss}$$

Hence, to resolve the kl vanishing problem, we need to apply kl weight annealing to control the kl weight, which is the β in the equation above. There are two ways of kl weight annealing applied here, which are "monotonic" and "cyclical", the following figure was how I implemented them.

```python
# ============================================================
# KL Annealing

class kl_annealing():
    def __init__(self, args):
        super().__init__()
        self.beta = args.beta
        # raise NotImplementedError

    def update(self):
        raise NotImplementedError

    def get_beta(self, mode, epochs):
        if mode == "monotonic":
            if epochs > 100:
                beta = 1
            else:
                beta = 0.01 * (epochs % 100)
        else: #"cyclical"
            if epochs % 100 > 50:
                beta = 1
            else:
                beta = 0.02 * (epochs % 100)
        return beta
        # raise NotImplementedError

kl_anneal = kl_annealing(args)

# ============================================================
```

```python
beta = kl_anneal.get_beta("cyclical", epoch)

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

Part 3–B: Teacher Forcing

There are basically two training modes in deep neural network, which are free-running mode and teacher-forcing mode. The free-running mode is the common way to train a network, which we take the output of the previous state as the input of the next state. And Teacher forcing is a strategy for training recurrent

neural networks that uses ground truth as input, instead of model output from a prior time step as an input.

If without Teacher Forcing, there might be some probability that the hidden states of the model will be updated by a sequence of wrong predictions, which would cause the errors to accumulate, and it is difficult for the model to predict something right in that situation.

However, teacher forcing might also bring some problems. Since in the inference, there was usually no ground truth available, as an alternative, we must take the previous prediction of the model its own back as the ground truth for the next prediction. With this difference between the training and prediction process, I thought it was obvious that this might bring poor model performance and instability, which is also known as “Exposure Bias”.

The following figure is how I implemented teacher forcing in my model.

- Iterate the teacher forcing ratio as the epoch increases.

```
# ==========

if epoch >= args.tfr_start_decay_epoch:
    tfr_value = tfr_value - args.tfr_decay_step

    if tfr_value < args.tfr_lower_bound:
        tfr_value = args.tfr_lower_bound

# ==========
```

- Determine whether to adopt teacher forcing through the tfr.

```
choices_of_tf = [True, False]
weight_of_tf_true = round(100*tfr_value, 0)
weight_of_tf_false = 100 - weight_of_tf_true
use_teacher_forcing = random.choices(choices_of_tf, weights=[weight_of_tf_true, weight_of_tf_false])
# use_teacher_forcing = False
```

- Different input to the frame predicter according to the decision made above

```
for i in range(1, args.n_past + args.n_future):
    h_target = h_seq[i][0]

    if args.last_frame_skip or i < args.n_past:
        h, skip = h_seq[i-1]
    else:
        h = h_seq[i-1][0]

    z_t, mu, logvar = posterior(h_target)
    # h_pred = frame_predictor(torch.cat([h, z_t, cond[i-1]], 1))
```

```
if use_teacher_forcing[0] == False:
    h, _ = encoder(x_pred_seq[i-1])

# if use_teacher_forcing:
#     h_pred = frame_predictor(torch.cat([h_target, z_t, cond[i-1]], 1))
# else:
h_pred = frame_predictor(torch.cat([h, z_t, cond[i-1]], 1))

x_pred = decoder([h_pred, skip])

x_pred_seq.append(x_pred)

mse += mse_criterion(x_pred, x[i].to(device))
kld += kl_criterion(mu, logvar, args)
# raise NotImplementedError
```

Part 4: Experimental results

Part 4-A: Results of Video Prediction

Below are the gif file and the predictions at each time step of one of the best results I obtained, which got an average PSNR score of 25.31547. Since Microsoft Word was not letting me to insert gif files in my report, there's a google drive link below the gif figure below which links to the corresponding gif file.

<p.s. I would compare the results of monotonic annealing and cyclical annealing >

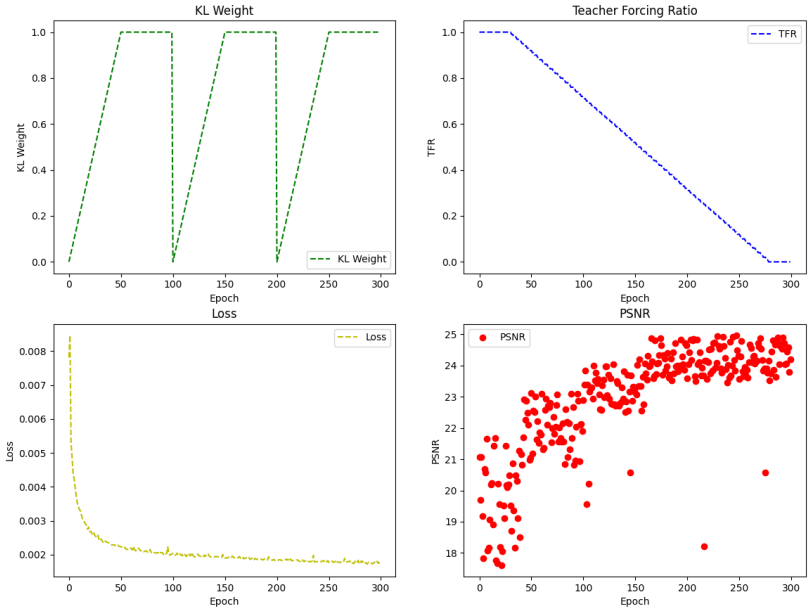
- GIF Image of test result

https://drive.google.com/file/d/1IT_Sil7vn-8NTKQ8JDFPM2A9GKsp5pf8/view?usp=sharing

- Prediction at each time step

Part 4-B: Plot of Loss, PSNR, TFR

Below is the comparison of one of the best results I mentioned above.



Part 4-C: Discussion

Part 4-C-1: Effects of Teacher Forcing Ratio

As I mentioned above, without Teacher Forcing, there might be some probability that the hidden states of the model will be updated by a sequence of wrong predictions, which would cause the errors to accumulate. In other word, teacher forcing kind of provided a guidance to the model. In my settings, teacher forcing ratio determines the probability whether the iteration would adopt teacher forcing or not.

There are 3 parameters related to teacher forcing in this lab, which are tfr_start_decay_epoch, tfr_decay_step, and tfr_lower_bound. Through several experiments, I found if we start to decay the teacher forcing ratio too early, we will still encounter the problem mentioned above. If we start decaying the teacher forcing ratio too late, our model will still get poor performance, and might crash at the second we started to lower the ratio.

In my point of view, the reason to the poor performance caused by decaying the ratio too early is because the model hasn't got enough guidance through teacher forcing. And the reason why the model would crash, I think it might be caused by overfitting, which the model learned too much from the ground truth,

and kind of became learning for learning' s sake. And this made the model kind of overwhelmed when there' s no ground truth as the input.

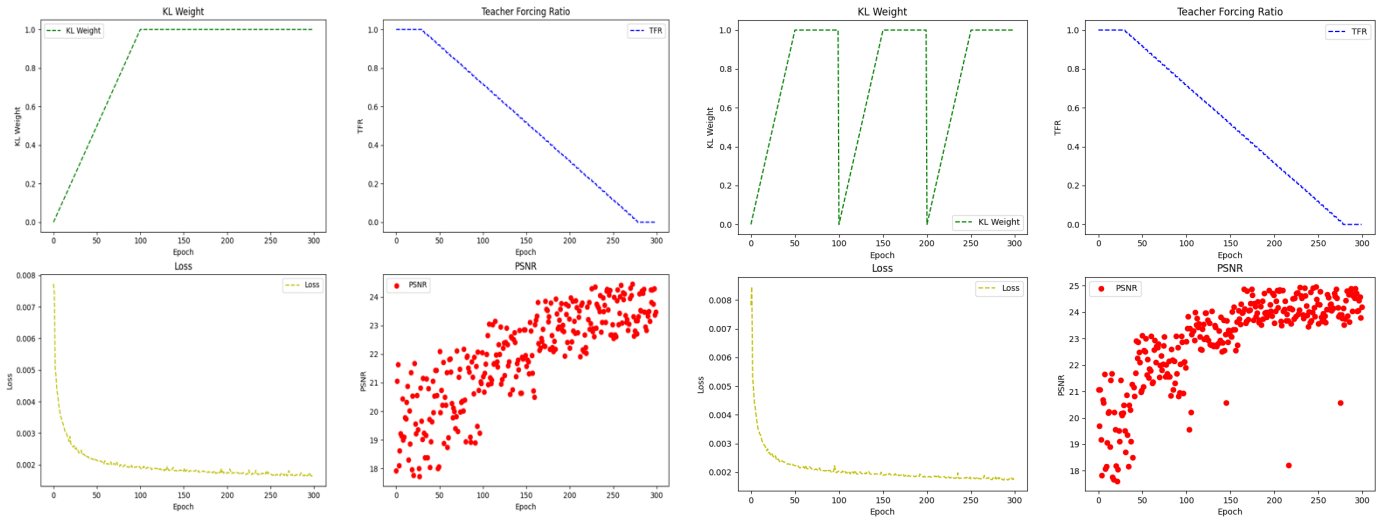
After several experiments, I found that the model performs the best when the tfr_start_decay_epoch was set to 30, tfr_decay_step set to 0.004, and tfr_lower_bound set to 0.001.

Part 4-C-2: Effects of KL Weight

As I mentioned above, I adopted two ways of kl weight annealing in this lab, which are "monotonic" and "cyclical". Below are the figures of their results.

First, I would like to talk about the monotonic annealing, which is a more traditional way to conduct kl annealing. At the very beginning, sine the β is still small, we could make the model to focus more on the input sequence but not the kl loss. When the β increases, as the model already git some know-how through the input sequence, we start to emphasize the shape of data distribution through bringing the weight of kl loss up in the whole loss.

Then let us focus on the cyclical annealing. Through the figure below, we can easily tell that cyclical annealing is kind of like the monotonic anneal which was repeated several times. We can also find the cyclical annealing got better results than the monotonic annealing, the accuracy also converged better than the monotonic annealing, too. After some research, I found it might be caused by the better organized latent space with trivial additional cost in computation of the repeating structure of cyclical annealing.



Part 4-C-3: Effects of Learning Rate

At the very beginning of the experiments, I was struggled by the poor results. I tried many classic solutions trying to improve the model, but they all didn't work too well, the model's PSNR always starts to decay as it reached bigger epoch. Then I came to my mind that in the past experiments in this lab, I set the learning rate as a constant, but not dynamically.

In experiments with big iteration numbers, it is better for us to lower the learning rate as the model reaches higher epoch number. Since as the longer the model learned, the more precise the move or the modification of the model should be. Without lowering the learning rate, the model might keep making big moves even at the sweet spots and might eventually cause crashing of the model.