Lab5 - Conditional VAE For Video Prediction

311581005 智能碩一 吳佳豪

1 INTRODUCTION

本次作業主要使用 Condition VAE (CVAE)訓練在 bair robot pushing small dataset 來進行影片預測，實作 CVAE model 包括 train functions、teacher forcing、KL annealing and reparameterization trick，並且顯示 training loss and PSNR 結果和預測的序列照片與影片。實驗結果發現當使用 teacher forcing 和 monotonic 與 cyclical annealing 的 KL annealing ，PSNR 分數皆可達到 **25 以上**，但如果都使用 teacher forcing 而沒有 Scheduled Sampling 時最高卻只能到 24，另外也有發現 optimizer 使用 Adam 相較 SGD 有較好的表現。

2 DERIVATION OF CVAE

首先

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

再來，在兩邊都加上任意分布$q(Z \mid c)$並在 Z上積分。

$$\begin{aligned}\int q(Z \mid c)\log p(X \mid c;\theta)dZ &= \int q(Z \mid c)\log p(X,Z \mid c;\theta)dZ - \int q(Z \mid c)\log p(Z \mid X,c;\theta)dZ \\ &= \int q(Z \mid c)\log p(X,Z \mid c;\theta)dZ - \int q(Z \mid c)\log q(Z \mid c)dZ \\ &\quad + \int q(Z \mid c)\log q(Z \mid c)dZ - \int q(Z)\log p(Z \mid X,c;\theta)dZ \\ &= \mathcal{L}(X,c,q,\theta) + KL\big(q(Z \mid c)|p(Z \mid X,c;\theta)\big)\end{aligned}$$

接著，引入由神經網路參數 θ 生成的分布$q(Z \mid X,c;\phi)$。

$$\begin{aligned}\mathcal{L}(X,c,q,\theta) &= \int q(Z \mid c)\log p(X,Z \mid c;\theta)dZ - \int q(Z \mid c)\log q(Z \mid c)dZ \\ &= \int q(Z \mid c)\log p(X \mid Z,c;\theta)dZ + \int q(Z \mid c)\log p(Z \mid c)dZ - \int q(Z \mid c)\log q(Z \mid c)dZ \\ &= E_{Z\sim q(Z|X,c;\phi)}\log p(X \mid Z,c;\theta) + E_{Z\sim q(Z|X,c;\phi)}\log p(Z \mid c) \\ &\quad - E_{Z\sim q(Z|X,c;\phi)}\log q(Z \mid X,c;\theta) \\ &= E_{Z\sim q(Z|X,c;\phi)}\log p(X \mid Z,c;\theta) - K\,L(q(Z \mid X,c;\theta)||\,p(Z \mid c))\end{aligned}$$

3 IMPLEMENTATION DETAILS

3.1 DESCRIBE HOW YOU IMPLEMENT YOUR MODEL

3.1.1 Encoder

主要是抽取輸入圖片的特徵值。模型則是使用提供的 VGG 架構的 encoder 模型，會將輸入 64 * 64 的圖片進行 down-sampling 與抽取特徵。

```
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):
        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]
```

Figure 1. Implement Encoder

3.1.2 Decoder

主要將 vector 轉換為原始圖片。模型則是使用提供的 VGG 架構的 decoder 模型，會將 vector 進行 up-sampling 盡可能生成 64 * 64 的原始輸入圖片。

```
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
```

Figure 2. Implement Decoder

3.1.3 Lstm

主要核心為透過閘門控制的概念，分別有決定哪些資訊要忘掉的 Forget Gate、決定哪些資訊要被記錄下來與更新主要單元的 Input Gate 和決定多少資訊要輸出的 Output Gate，來改善 RNN 在長期記憶的問題，使得 Lstm 比一般 RNN 有更長的記憶力。

在 Gussian lstm 中加入雜訊項來訓練 decoder，並且增加 update_device() function 來更新訓練或是預測時會用到運算 device 資源，避免更換裝置時發生無法訓練的問題。

```python
class gaussian_lstm(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, n_layers, batch_size, device):
        super(gaussian_lstm, self).__init__()
        self.device = device
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.batch_size = batch_size
        self.embed = nn.Linear(input_size, hidden_size)
        self.lstm = nn.ModuleList([nn.LSTMCell(hidden_size, hidden_size) for i in range(self.n_layers)])
        self.mu_net = nn.Linear(hidden_size, output_size)
        self.logvar_net = nn.Linear(hidden_size, output_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        hidden = []
        for _ in range(self.n_layers):
            hidden.append((Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device)),
                           Variable(torch.zeros(self.batch_size, self.hidden_size).to(self.device))))
        return hidden

    def update_device(self,device):
        self.device = device
        hidden = []
        for _, (x,y) in enumerate(self.hidden):
            hidden.append((x.to(device),y.to(device)))
        self.hidden = hidden

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

    def forward(self, input):
        embedded = self.embed(input)
        h_in = embedded
        for i in range(self.n_layers):
            self.hidden[i] = self.lstm[i](h_in, self.hidden[i])
            h_in = self.hidden[i][0]
        mu = self.mu_net(h_in)
        logvar = self.logvar_net(h_in)
        z = self.reparameterize(mu, logvar)
        return z, mu, logvar
```

Figure 3. Implement Lstm

3.1.4 KL cost annealing

VAE 模型主要通過優化損失函數 ELBO 來實現神經網路的參數優化，及最大化重建項並最小化 KL divergence 項，但 VAE 在重建項較強大，即 decoder 強大所以就會忽視由 Encoder 產生的後驗分布$q(Z \mid X;\phi)$而只從$N(0,I)$中採樣就可生成，且 KL 項約束很強$L_{KL}=0$，即後驗分布$q(Z \mid X;\phi)$退化成$p(z)$一樣的高斯分布，則會發生 KL-Vanishing，無法為 Decoder 提供資訊。

$$\mathcal{L}(\mathrm{X},\mathrm{q},\theta) = E_{Z\sim q(Z|X;\phi)} \log p(X|Z;\theta) - KL(q(Z|X;\phi)||p(Z))$$

where $q(Z|X;\phi)$ is considered as encoder and $p(X|Z;\theta)$ as decoder.

Figure 4. ELBO

為了解決 KL-Vanishing 的問題，因此改在 KL 項多乘上一個權重係數 w，並一開始設定為 w = 0，使得一開始訓練模型會先忽視 KL，選擇先降低重建項得 errors，並在之後慢慢增加 w，使模型慢慢開始重視降低 KL 項。而這次作業權重增加的方法有 Monotonic 與 Cyclical 兩種方法，前面一種是隨著 step 的增加 w 由 0 到 1 慢慢遞增，後面一種方法是 w 由 0 到 1 循環。

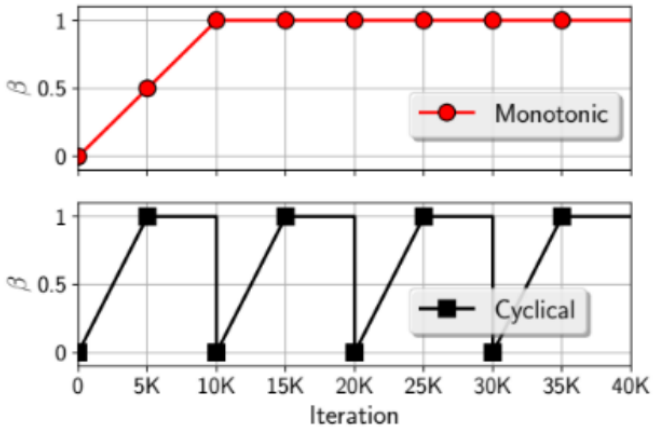


Figure 5. The design of KL annealing schedule

```python
class kl_annealing():
    def __init__(self, args):
        super().__init__()
        self.cyclical = args.kl_anneal_cyclical
        self.ratio = args.kl_anneal_ratio
        self.cycle = args.kl_anneal_cycle if self.cyclical else 1
        self.period = args.niter / self.cycle
        self.step = 1 / (self.period * self.ratio)
        self.v = 0
        self.i = 0

    def update(self):
        self.v += self.step
        self.i += 1
        if self.v >= 1:
            if self.i >= self.period:
                self.v = 0
                self.i = 0
            else :
                self.v = 1

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

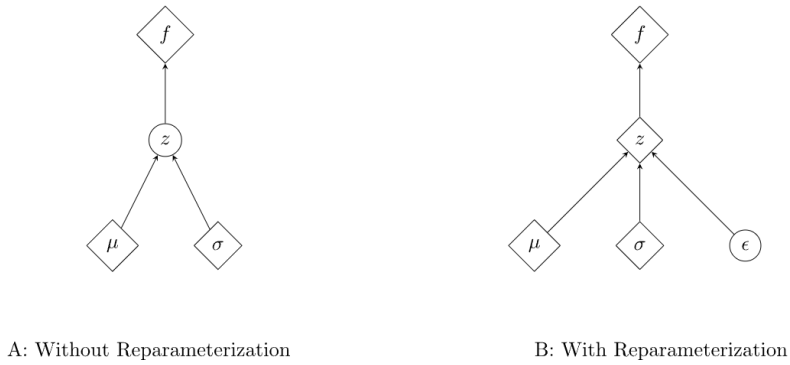
Figure 6. Implement KL annealing

3.1.5 Reparameterization trick

因為原本 VAE 是從模型參數化的分布中抽樣，但過程中無法微分因此不能使用梯度的方法來更新參數，所以需要將模型的預測與隨機抽樣的元素分開。首先是

先將隨機採樣改為用雜訊項 ε ，並且將雜訊項設定為高斯分布，使得可通過其 mean 和 standard deviation 進行參數化。

上面等於將原本從$N(\mu, \sigma^2)$中採樣一個 Z，變成從$N(0, I)$中採樣一個 ε ，使得 $Z = \mu + \varepsilon \times \sigma$。從前面公式可以發現已經將隨機的元素與學習的參數化分離了，因此可以使用梯度方法來更新模型參數。



A: Without Reparameterization　　B: With Reparameterization

Figure 7. Reparameterization concept

```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = Variable(std.data.new(std.size()).normal_())
    return eps.mul(std).add_(mu)
```

Figure 8. Implement reparameterization

3.2 DESCRIBE THE TEACHER FORCING

常見序列生成任務的訓練方式有分為兩種，一種是 free-running mode，另一種是 teacher-forcing mode。前一個主要是將上一個 state 輸出的結果給下一個 state 作為輸入，但可以簡單想到如果前一個 state 的輸出與實際輸出差距很大，且又將錯誤的資料輸入給下一個 state 做判斷，就會使後面的訓練都發生錯誤。因此後面一種主要是將訓練資料的標準答案當作每一個 state 的輸入，而不是用模型上一個 state 的輸出當作輸入，因此可以避免發生預測錯誤後導致後面的預測結果都受到嚴重的影響並使訓練過程更快收斂。

但缺點有以下幾點 ：

1. 發生 Exposure Bias：因為訓練期間都是使用 Ground truth 進行下個 state 預測，所以模型都沒有從中學習到自己的預測錯誤，當預測期間時就會導致 decoder 無法預測出下一個。

2. Overcorrect 問題：可能會發生當在預測中間階段時，突然將原本預測的值變更成 ground truth 做為下一個 state 的輸入，而破壞了原本接下去預測的邏輯或語意正確性，導致後面訓練錯誤。

3. 限制多樣性問題：因為每次生成都會受到 ground truth 的影響，所以會使得生成都受到約束，而無法有不同的表現方式。

為了解決 Exposure Bias 問題，因此使用 Scheduled Sampling 的方式解決：

主要解決方法為在模型訓練過程每個 step 都有 p 的機率使用 teacher-forcing，而有 1 - p 的機率使用上一個 state 的輸出作為下一個 state 的輸入，可以達到前期快速收斂與有效提升修正自我錯誤的能力。以下為實作的 code 與可比對標題 4.3 中有無使用 teacher-forcing mode 實驗的差別。

```python
if epoch >= args.tfr_start_decay_epoch:
    args.tfr -= args.tfr_decay_step
    if args.tfr < args.tfr_lower_bound:
        args.tfr = args.tfr_lower_bound
```

Figure 9. Implement teacher forcing decay in main function

```python
seq = [modules['encoder'](x[:,i].to(device)) for i in range(args.n_past + args.n_future)]
for i in range(1, args.n_past + args.n_future):
    target = seq[i][0]
    if args.last_frame_skip or i < args.n_past:
        h, skip = seq[i - 1]
    else:
        if random.random() < args.tfr:
            h = seq[i- 1][0]
        else :
            h = modules['encoder'](x_pred.to(device))[0]
    z_t,mu,logvar = modules['posterior'](target)
    h_pred = modules['frame_predictor'](torch.cat([h,z_t,cond[:,i].to(device)],1))
    x_pred = modules['decoder']([h_pred,skip])
    mse += mse_criterion(x_pred,x[:,i].to(device))
    kld += kl_criterion(mu,logvar,args)
```

Figure 10. Implement Scheduled Sampling in train function

4 RESULTS AND DISCUSSION

超參數設定：

Table 1. Model hyperparameter settings

Hyperparameter	
niter	300
Epoch_size	600
Batch_size	18
Learning rate	0.002
n_past	2
N_future	10
Random seed	1

4.1 SHOW YOUR RESULTS OF VIDEO PREDICTION

4.1.1 Make videos or gif images for test result

影片網址: Predict_result_video

4.1.2 Output the prediction at each time step

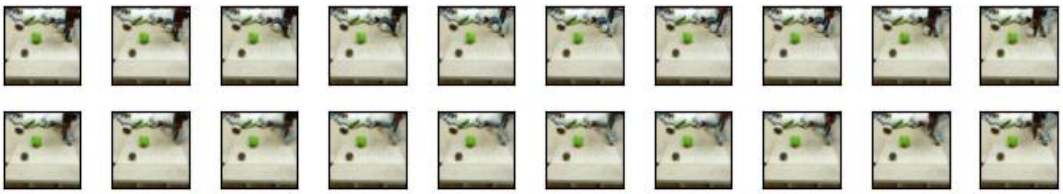


Figure 11. Above is ground truth and below is prediction

4.2 PLOT THE KL LOSS AND PSNR CURVES DURING TRAINING

下方實驗結果皆依照 Table 1.參數進行設定，Optimizer 使用 Adam。

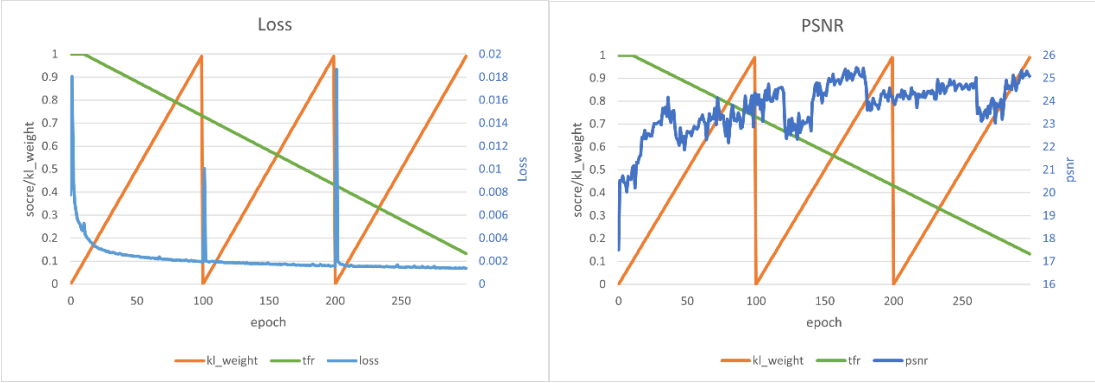


Figure 12. KL loss and RSNR curves with Adam

4.3 DISCUSS THE RESULTS ACCORDING TO YOUR SETTING OF TEACHER FORCING RATIO, KL WEIGHT, AND LEARNING RATE.

以下實驗結果都依照 Table 1.參數所設定。

首先比較使用 teacher forcing mode 時有無調整 ratio 的差別。從 Figure 13. 可以發現如果 ratio 如果都設定為 1 時，代表都使用 ground truth 做為下一個 state 輸入時，所以會發生 Exposure Bias 的情況(標題 3.2 有詳細解釋)，所以預測時最高只有到達 PSNR 24。而從 Figure 12.可以發現如果使用 Scheduled Sampling 的方法並參數設定為從 epoch 為 10 時開始每個 epoch 下降 0.003 的 ratio 時 PSNR 可到 25 以上。

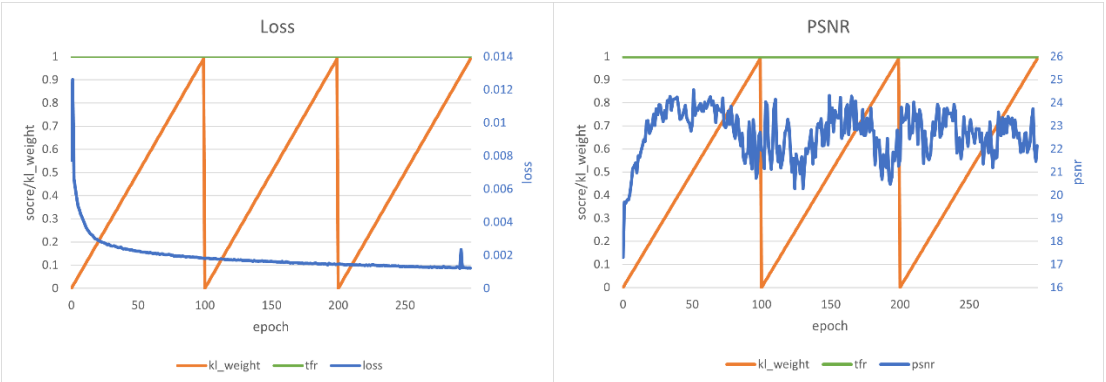


Figure 13. Teacher forcing ratio always set 1

再來比較使用 KL annealing 使用 Monotonic(Figure 14.)與 Cyclical (Figure 12.)的差別。可能是因為 Cyclical 當到周期時都會使 KL weight 歸零，所

以會導致 loss 瞬間上升，並且會使 PSNR 較為抖動，但以上兩種方法都可以使 PSNR 達到 25 以上的分數。

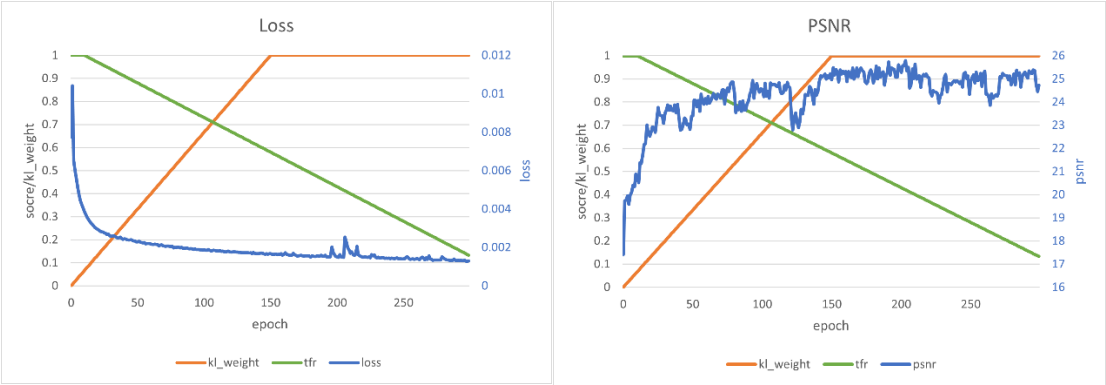


Figure 14. KL annealing use Monotonic method

learning rate 皆設定為 0.002，然後增加使用 ReduceLROnPlateau 方法的 scheduler 來看是否有差別，參數為監控 PSNR 數值如果 10 個 epochs 沒有上升則 learning rate 下降 0.5 倍。雖然 Adam 已經有自適應，但是如果到一定的程度下降還是會有所限制，因此想說增加 scheduler 看是否有比較好的表現，發現有使用 scheduler(Figure 15.)的最佳值 PSNR 相較沒有使用 scheduler(Figure 12.)來說增加 0.1 的分數，而且 loss 收斂的速度較快與低於 0.002 的表現。

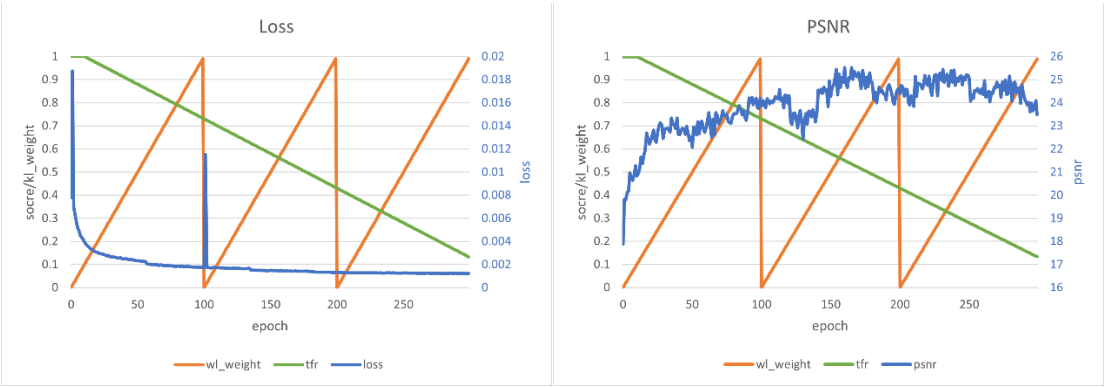


Figure 15. KL loss and RSNR curves with Adam + scheduler

最後則是比較 optimizer 使用 Adam 與 SGD + momentum + scheduler 的差別。如下圖(Figure 16.)發現 SGD 就算是動態調整 learning rate 的大小，還是可能因為下降速度沒有辦法與 Adam 相比且更加彈性的調整，所以收斂速度相較 Adam(Figure 12.)都明顯的較慢，因此訓練到 160 個 epoch 提前結束訓練。

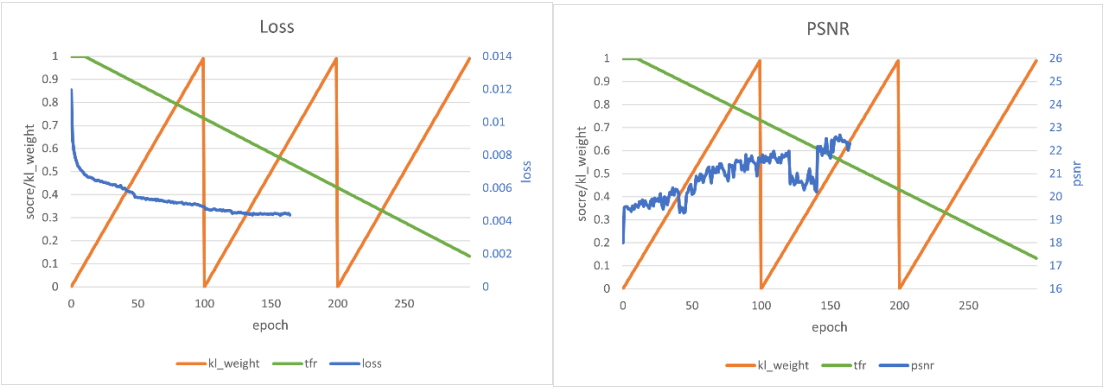


Figure 16. KL loss and RSNR curves with SGD + momentum + scheduler