Lab 4 - Conditional VAE for Video Prediction

313551055 柯柏旭

i. Derivate conditional VAE formula

$$L(X, c, q, \theta) = E_{z \sim q(Z|X,c;\phi)} \log p(X|Z, c; \theta) - KL(q(Z|X, c; \phi)||p(Z|c)) \quad (2)$$

ii. Introduction

這份報告探討了在 Conditional Variational Autoencoder, CVAE 應用於 video frame 預測的過程中，如何通過不同的策略來優化模型訓練的效果。報告涵蓋了模型的訓練與測試實作、reparameterization 技巧的實現、teacher forcing strategy 的設定以及 KL divergence annealing 比率的設置。通過對不同訓練設定下的損失曲線和 PSNR 曲線的分析，展示了 KL divergence annealing 和 teacher forcing strategy 對模型性能的影響，並探討了不同策略在提高生成影像品質和模型穩定性方面的效果。

iii. Implementation details

1. How do you write your training/testing protocol

- Training protocal

```
def training_one_step(self, img, label, adapt_TeacherForcing):
    self.train()
    self.optim.zero_grad()


    total_loss = 0

    img = img.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C,
H, W)
    label = label.permute(1, 0, 2, 3, 4) # change tensor into (seq,
B, C, H, W)
    output = img[0]
```

```
    # Process each frame in the sequence separately
    for i in range(1, img.shape[0]):  # Iterate over the sequence
length
        current_img = img[i]
        current_label = label[i]

        # Forward pass
        frame_feature = self.frame_transformation(current_img)
        label_feature = self.label_transformation(current_label)

        # Conduct Posterior prediction in Encoder
        z, mu, logvar = self.Gaussian_Predictor(frame_feature,
label_feature)


        if adapt_TeacherForcing:
            img0_feature = self.frame_transformation(img[i - 1])
        else:
            img0_feature = self.frame_transformation(output)


        # Decoder fusion and generative model
        latent = self.Decoder_Fusion(img0_feature, label_feature, z)
        output = self.Generator(latent)

        # sigmoid
        output = torch.nn.functional.sigmoid(output)

        # Compute losses
        recon_loss = self.mse_criterion(output, current_img)
        kl_loss = kl_criterion(mu, logvar, self.batch_size)
        loss = recon_loss + self.kl_annealing.get_beta() * kl_loss

        total_loss += loss
```

```
    # total_loss /= img.size(0)  # Average the loss over the sequence length

    # Backward pass and optimization
    total_loss.backward()
    self.optimizer_step()

    return total_loss
```

主要流程如下：

首先，模型設為訓練模式，並將梯度歸零。

接著遍歷序列中的每一幀圖像（第一幀除外），對於每一幀圖像：

藉由 `frame encoder` 和 `label encoder` 做特徵提取，接著將兩者的特徵作為 input 給 `Gaussian predictor`，`Gaussian predictor` 會生成 latent variable 的 平均值和平方差。接著為了能夠做參數最佳化，會接著做 **reparameterization trick**。

- 以下為 reparameterization trick 的詳細說明：

首先，我們的目標是從一個高斯分佈$\mathcal{N}(\mu, \sigma^2)$中抽樣 latent variable z，這裡的 μ 和 $\log(\sigma^2)$ 是通過編碼器學習到的。但直接從 $\mathcal{N}(\mu, \sigma^2)$ 抽樣 z 是一個隨機過程，這會使得網絡的梯度難以計算。因此需要 reparameterization，我們可以將隨機變量 z 表示為一個可微分的函數：

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

其中 ϵ 是從標準正態分佈 $\mathcal{N}(0, 1)$ 中抽樣出來的。
按照作業要求，σ 是通過 $\log(\sigma^2)$ 計算出來的，具體公式是：

$$\sigma = \exp\left(\frac{\log(\sigma^2)}{2}\right)$$

上述公式中的 ϵ 負責引入隨機性 (生成式模型需具備多樣性的分佈)，而 μ 和 σ 負責調整抽樣結果，使其符合所學到的分佈。

使用這個技巧後，z 現在是由 μ 和 σ 控制的，而 μ 和 σ 是網絡的輸出，因此 z 的計算過程是可微分的。我們可以對網絡進行反向傳播，優化參數，這解決了直接抽樣無法計算梯度的問題。

說明完 **reparameterization trick**，接著是 **teacher forcing**。教師強制的目的是幫助模型更快地收斂並學會正確的序列預測。為了幫助模型更好地學習，教師強制技術允許在訓練時，根據情況選擇使用前一幀的真實圖像（ground truth）作為當前步驟的輸入，而不是使用模型先前生成的輸出來預測下一幀。

然後最後就是將 features (前一 frame 的相似性)和 latent variable(下一 frame 的隨機性) 結合起來丟給 decoder 融合得到新的輸出 latent。然後再把 latent 作為輸入，傳遞給 Generator，生成最終的圖像輸出。

最後的最後會再套一層 sigmoid，這使得最終的輸出圖像中的每個像素值都位於這個範圍內。這樣可以保證生成的圖像有合理的亮度和對比度 (之後會在 test 的時候按照比例變回 255)。

至於 loss 的計算方式為 mse 的 reconstruction loss 和 kl divergence loss 合再一起計算總損失。其中 kl loss，是通過計算生成的分佈（由 mu 和 logvar 描述的正態分佈）與標準正態分佈之間的 Kullback-Leibler (KL) 散度來獲得的。KL 散度度量了兩個概率分佈之間的差異，這裡的目的是使生成的 latent variable 分佈與標準正態分佈盡可能接近，以便在生成階段可以從標準正態分佈中方便地抽樣。

而 kl loss 的計算又使用了 kl annealing 的策略來調整他在 total loss 中的比例。

- Testing protocal

Valid

```
@torch.no_grad
def val_one_step(self, img, label):
    total_loss = 0
    total_psnr = 0

    img = img.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    label = label.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)

    decoded_frame_list = [img[0]]

    # Process each frame in the sequence separately
    for i in range(1, img.shape[0]):  # Iterate over the sequence length
        current_img = img[i]
        current_label = label[i]

        # Forward pass
        frame_feature = self.frame_transformation(decoded_frame_list[-1])
```

```
        label_feature = self.label_transformation(current_label)

        # Conduct Posterior prediction in Encoder
        z, mu, logvar = self.Gaussian_Predictor(frame_feature,
label_feature)

        # Decoder fusion and generative model
        decoder_feature = self.Decoder_Fusion(frame_feature,
label_feature, torch.randn_like(z))
        output = self.Generator(decoder_feature)

        output = torch.nn.functional.sigmoid(output)

        decoded_frame_list.append(output)

        # Compute losses
        recon_loss = self.mse_criterion(output, current_img)
        kl_loss = kl_criterion(mu, logvar, self.batch_size)
        loss = recon_loss + self.kl_annealing.get_beta() * kl_loss

        total_loss += loss

        # Compute PSNR
        psnr = Generate_PSNR(output, current_img)
        total_psnr += psnr

    # total_loss /= img.size(0)  # Average the loss over the sequence
length
    total_psnr /= img.size(0)  # Average the PSNR over the sequence
length

    generated_frame = stack(decoded_frame_list).permute(1, 0, 2, 3,
4)
    self.make_gif(generated_frame[0],
os.path.join(self.args.save_root, f'pred_seq.gif'))
```

```
    return total_loss, total_psnr
```

Valid 的部分和上面的 training 大同小異，其中比較特別的是 z 在 valid (test) 的時候進入 decoder 融合的東西是 features 和從高斯分佈中抽取的一個和 z 長的一樣的 tensor。會這麼做的原因推測是為了引入隨機性，以模擬生成過程的隨機抽樣特性。使模型不要過度依賴訓練中學到的 latent，以避免 overfitting 並提高泛化能力。除此之外，為了更清楚訓練成效，最後會計算每個 step 的 PSNR，並且可視化生成結果。

Test

```
def val_one_step(self, img, label, idx=0):
    img = img.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    label = label.permute(1, 0, 2, 3, 4) # change tensor into (seq, B, C, H, W)
    assert label.shape[0] == 630, "Testing pose seqence should be 630"
    assert img.shape[0] == 1, "Testing video seqence should be 1"

    # decoded_frame_list is used to store the predicted frame seq
    # label_list is used to store the label seq
    # Both list will be used to make gif
    decoded_frame_list = [img[0]]
    label_list = []

    # TODO
    for t in range(1, label.shape[0]):
        current_img = img[0] if t == 1 else decoded_frame_list[-1]
        current_label = label[t]

        # Forward pass
        frame_feature = self.frame_transformation(current_img)
        label_feature = self.label_transformation(current_label)

        # Conduct Posterior prediction in Encoder
        z, mu, logvar = self.Gaussian_Predictor(frame_feature, label_feature)
```

```
        # Decoder fusion and generative model
        decoder_feature = self.Decoder_Fusion(frame_feature,
label_feature, torch.randn_like(z))
        output = self.Generator(decoder_feature)
        output = torch.nn.functional.sigmoid(output)

        decoded_frame_list.append(output)
        label_list.append(current_label)


    # Please do not modify this part, it is used for visulization
    generated_frame = stack(decoded_frame_list).permute(1, 0, 2, 3,
4)
    label_frame = stack(label_list).permute(1, 0, 2, 3, 4)

    assert generated_frame.shape == (1, 630, 3, 32, 64), f"The shape
of output should be (1, 630, 3, 32, 64), but your output shape is
{generated_frame.shape}"

    self.make_gif(generated_frame[0],
os.path.join(self.args.save_root, f'pred_seq{idx}.gif'))

    # Reshape the generated frame to (630, 3 * 64 * 32)
    generated_frame = generated_frame.reshape(630, -1)

    return generated_frame
```

Test 和 valid 幾乎一樣，只有讀入的資料集不一樣而已。

2. How do you implement reparameterization tricks

```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)  # 計算標準差
    eps = torch.randn_like(std)    # 從標準正態分佈中抽樣
    return mu + std * eps          # 計算最終的潛在變量
```

針對 reparameterization 的目的已經在上一小節的 training part 說明過了。接著在這邊說一下實作細節，**logvar** 是模型輸出的 log 變異數。由於變異數在計算中經常被對數化以增加數值穩定性，我們需要先將 log 變異數轉換回變異數，再計算標準差。

torch.exp(0.5 * logvar) 是將 log 變異縮放到 0.5，並計算其指數，以獲得標準差 (std)。這是因為 log 變異數是對數空間中的數值，直接取 exp 可以將其轉換回變異數，取 0.5 的原因是因為標準差是變異數的平方根。

torch.randn_like(std) 是生成與 std 張量形狀相同的隨機噪聲張量 eps，其元素來自均值為 0、標準差為 1 的標準正態分佈。這個隨機噪聲的作用是引入隨機性，使得生成的潛在變量 z 在每次前向傳播時都略有不同，這是 VAE 中進行生成的核心。

而最終的 latent variable z 由 $mu + std * eps$ 計算得出，這樣的計算方式確保了 z 的值會隨機變動，但仍然遵循 mu 和 logvar 定義的正態分佈。

3. How do you set your teacher forcing strategy

```
def teacher_forcing_ratio_update(self):
    if self.current_epoch >= self.tfr_sde:
        self.tfr -= self.tfr_d_step
        self.tfr = max(0, self.tfr)
```

在 training stage 會根據當下的 teacher forcing ratio 去做 random 即 `adapt_TeacherForcing = True if random.random() < self.tfr else False`。如果為 `True`，則會進行下面的 `if` statement。

```
if adapt_TeacherForcing:
    img0_feature = self.frame_transformation(img[i - 1])
else:
    img0_feature = self.frame_transformation(output)
```

`img0_feature = self.frame_transformation(img[i - 1])`：如果應用教師強制 (adapt_TeacherForcing 為 True)，則使用真實的前一幀影像 img[i - 1] 進行特徵轉換（frame_transformation）。

`img0_feature = self.frame_transformation(output)`：如果不應用教師強制 (adapt_TeacherForcing 為 False)，則使用模型生成的上一幀影像 output 進行特徵轉換。

4. How do you set your kl annealing ratio

```
class kl_annealing():
```

```
    def __init__(self, args, current_epoch=0):
        self.current_epoch = current_epoch
        self.beta_schedule =
self._compute_beta_schedule(args.num_epoch, args.kl_anneal_cycle,
args.kl_anneal_ratio)
        # print(self.beta_schedule)

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

    def get_beta(self):
        return self.beta_schedule[self.current_epoch]

    def _compute_beta_schedule(self, num_epochs, n_cycle=1, ratio=1,
start=0.0, stop=1.0):
        beta = np.ones(num_epochs)
        period = num_epochs / n_cycle
        step = (stop - start) / (period * ratio)

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

主要的重點在於 `_compute_beta_schedule` 這個 function。

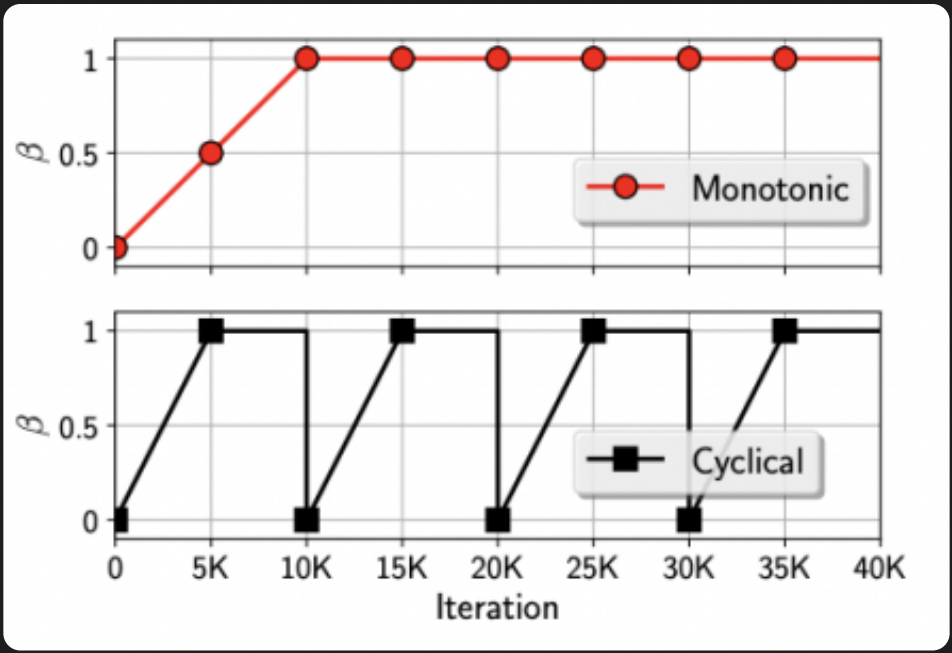
首先，`_compute_beta_schedule` 方法會初始化一個長度為 num_epochs 的 beta 陣列，所有元素初始值均為 1，這個陣列將會儲存每個 epoch 對應的 beta 值。

接著，方法會計算每個週期的長度 period 和在每個週期內 beta 值每一步的增量 step。period 是總的 epoch 數除以週期數 n_cycle，決定了每個週期會持續多少個 epoch。

step 是 beta 在每個週期內從 start 到 stop 的變化量，ratio 則控制了在每個週期內 beta 上升部分的比例。

接下來，通過迴圈來遍歷每個週期，在每個週期內，beta 的值會從 start 開始逐步增加到 stop。這個迴圈中的 v 是當前 beta 的值，i 是在週期內的位置索引。當 v 增加超過 stop 或者當前 epoch 超過總 epoch 數時，該週期的計算就會結束。

最後，這個方法會返回計算好的 beta 陣列，包含了每個 epoch 對應的 beta 值。



對於作業要求的 `_compute_beta_schedule` 主要是為 **Cyclical** 所設計的，至於 **Monotonic**，即是把該 function 的 `n_cycle` 設為 1 ，便可以實作。

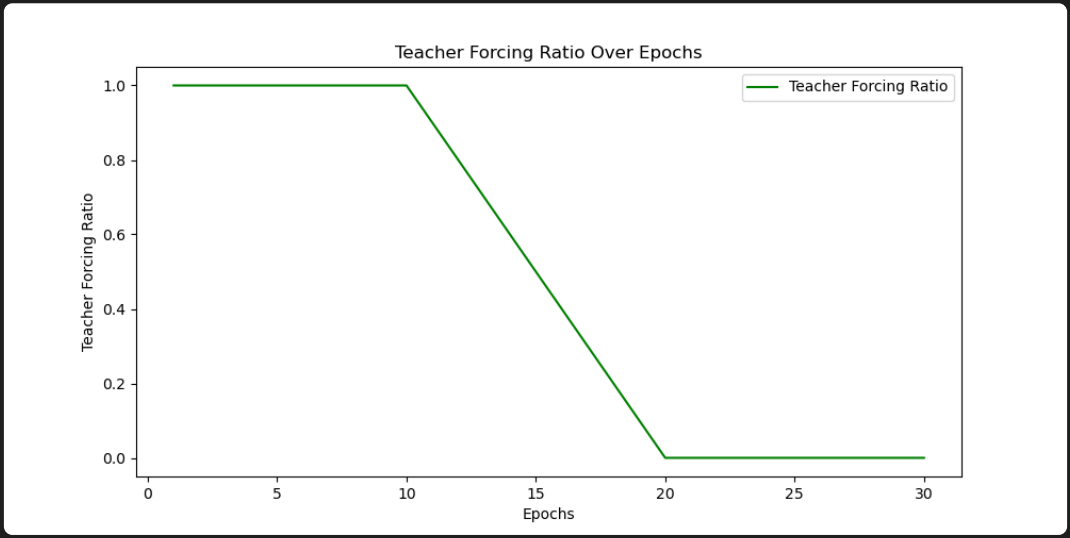
iv. Analysis & Discussion

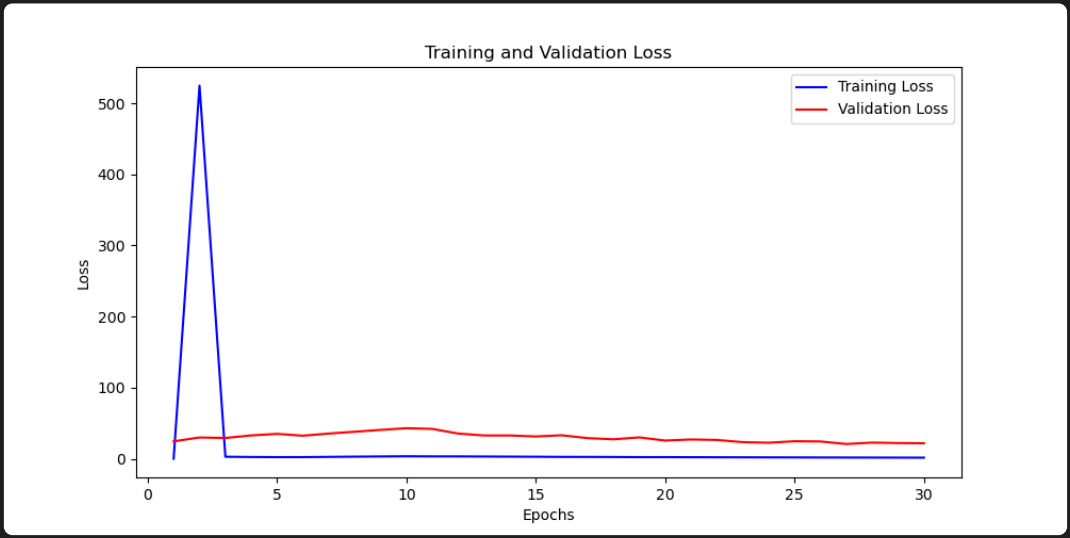
1. Plot Teacher forcing ratio

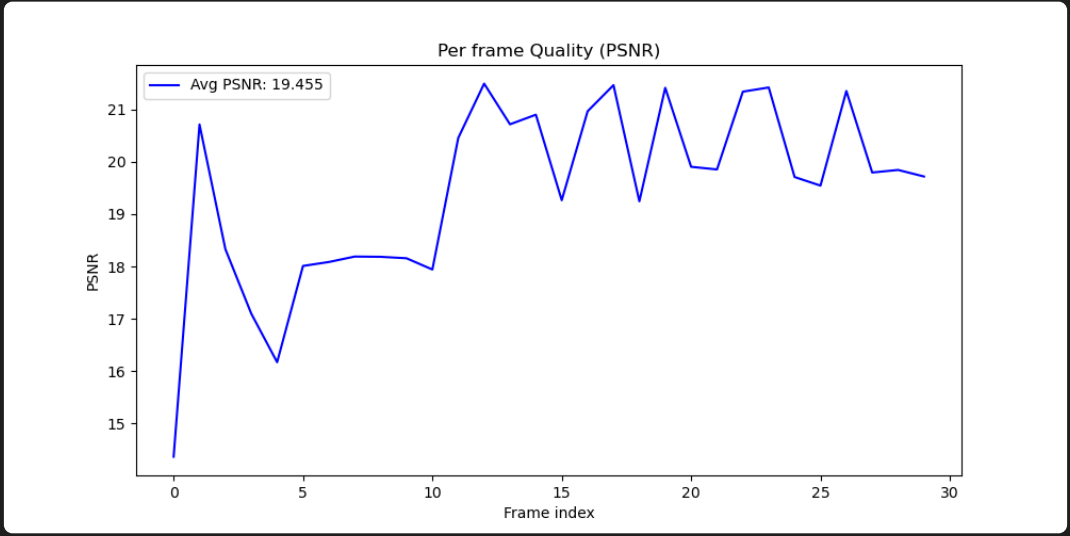
a. Analysis & compare with the loss curve

固定設定: 30 epochs、Monotonic KL annealing

- With teacher forcing ratio (tfr=1, tfr_sde=10, tfr_d_step=0.1)

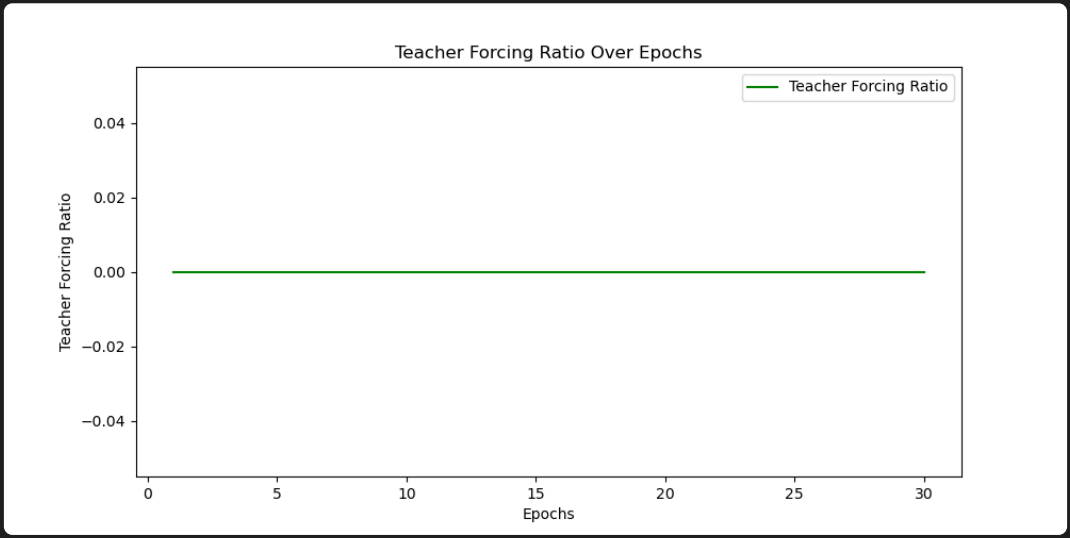


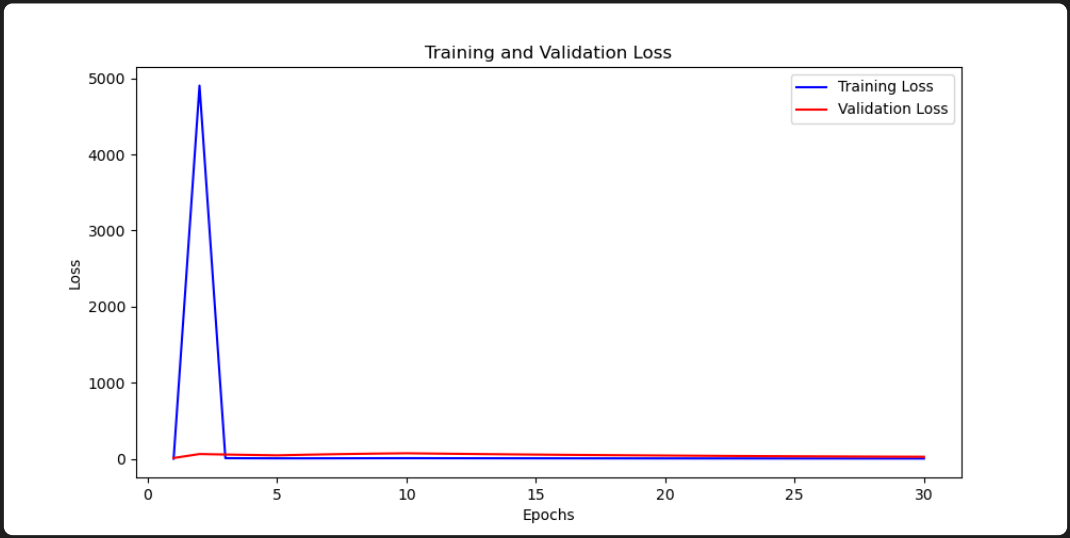


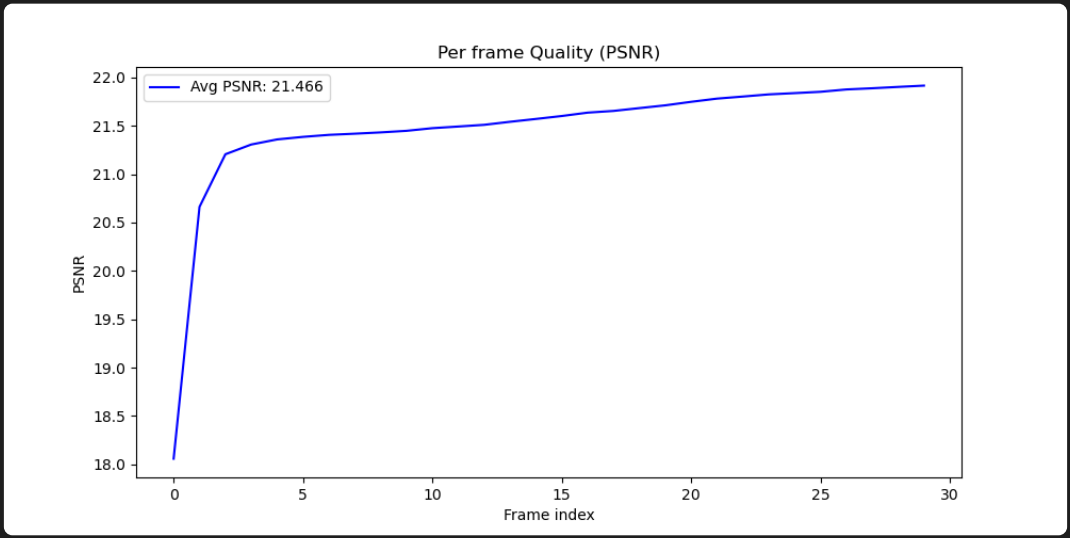


p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

- Without teacher forcing ratio







p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

結果分析與比較

雖然從 loss curve (第二張) 看不太出來明顯的差異，但仔細觀察可以發現，沒有使用 teacher forcing 的訓練模式，validation loss 會比較收斂，也不會跳來跳去。我覺得看 validation dataset 的 PSNR curve (第三張) 會更明顯。推測原因可能有以下三種:

(1) tfr 設的不當，但我有試過從 0.8, 0.6, 0.4 開始，PSNR curve 還是長得差不多。

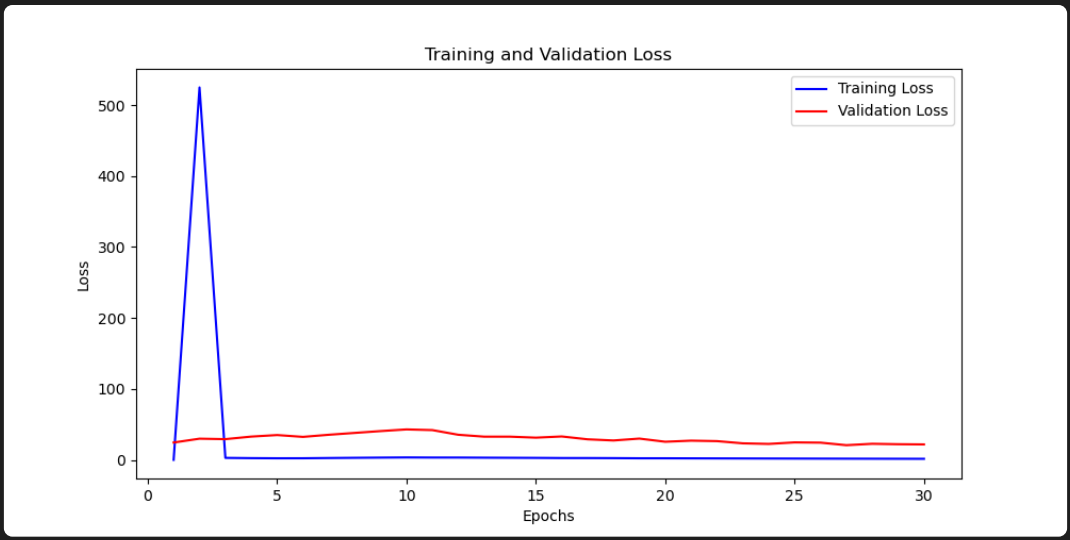
(2) 有些模型在完全自回歸的訓練方式下（即沒有教師強制）可能更容易穩定下來，因為**模型會逐步適應生成自身預測的輸出**。在有教師強制的情況下，模型每一步都被提供真實的輸入，這可能導致在實際應用中面對連續生成的挑戰時表現不佳。

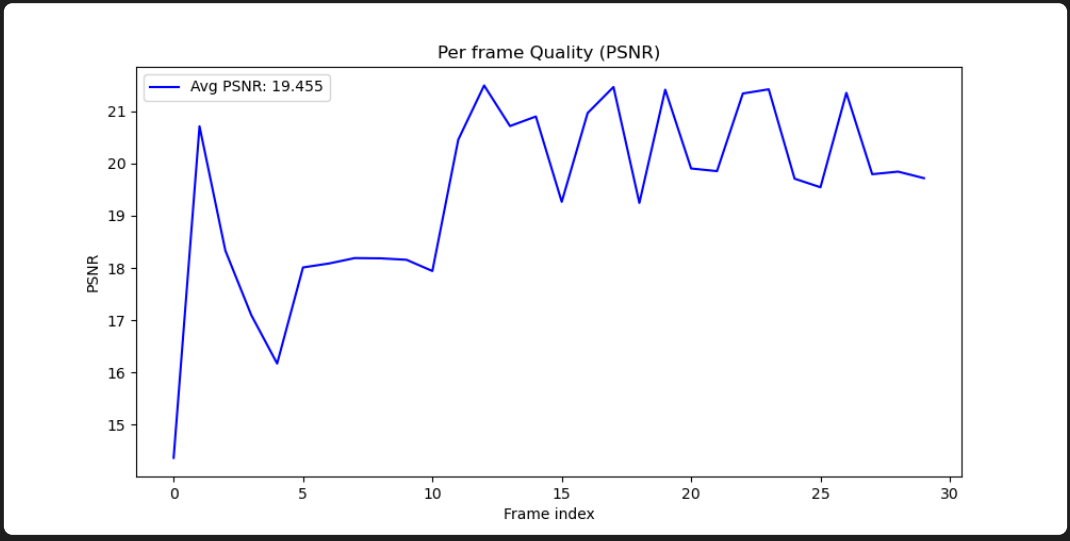
(3) 在使用教師強制時，模型可能容易**過度擬合於訓練數據**，因為它始終能夠依賴真實的輸入而不是生成的輸出。在訓練中去掉教師強制，模型可能會更專注於學習如何在面對不完美的輸入（即之前生成的輸出）時進行預測，從而具有更好的泛化能力。

2. Plot the loss curve while training with different settings. Analyze the difference between them

固定設定: 30 epochs、teacher forcing (tfr=1, tfr_sde=10, tfr_d_step=0.1)

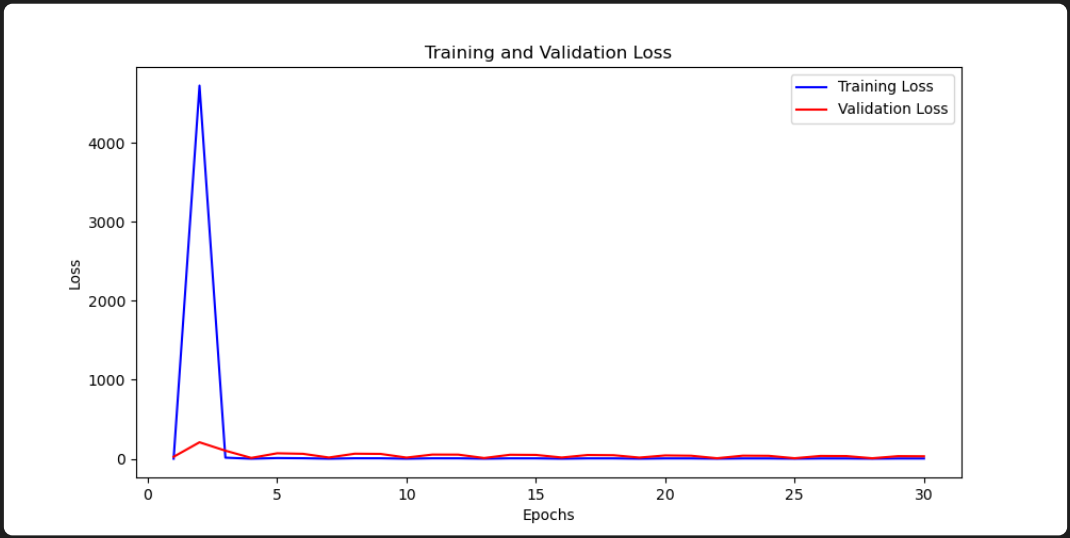
a. With KL annealing (Monotonic)

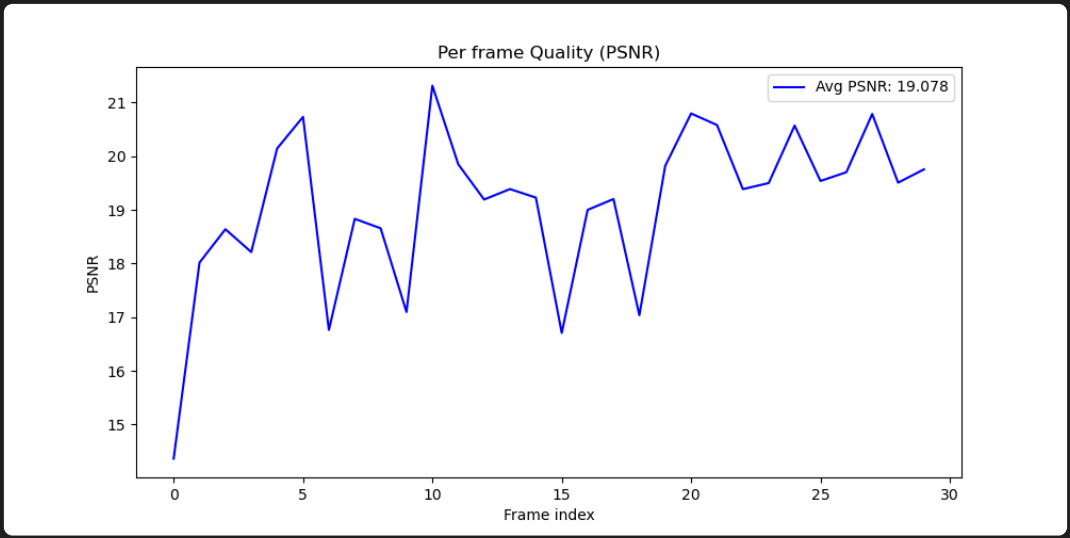




p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

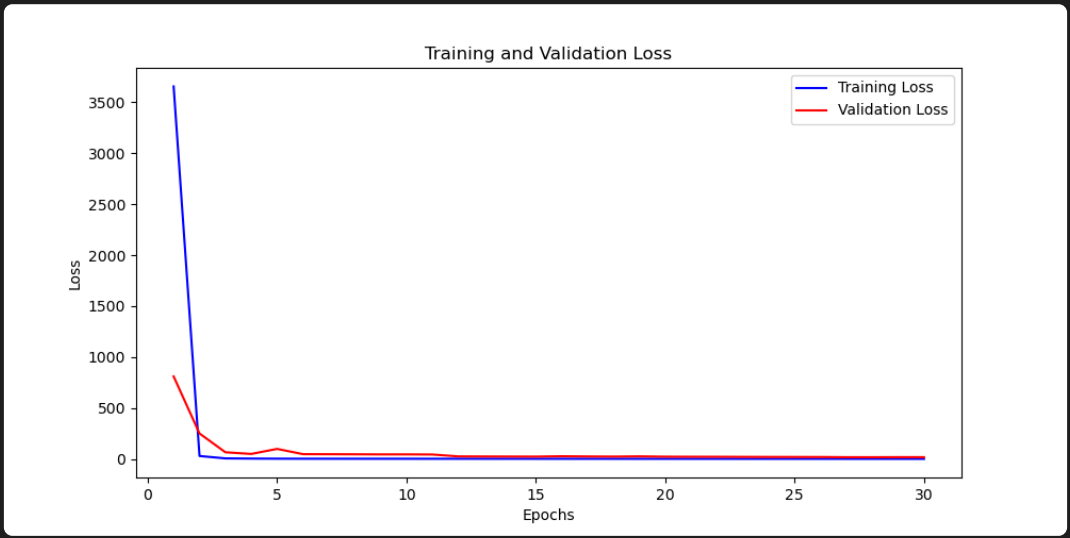
b. With KL annealing (Cyclical) (n_cycle = 10, 一個 cycle 涵蓋三個 epochs)

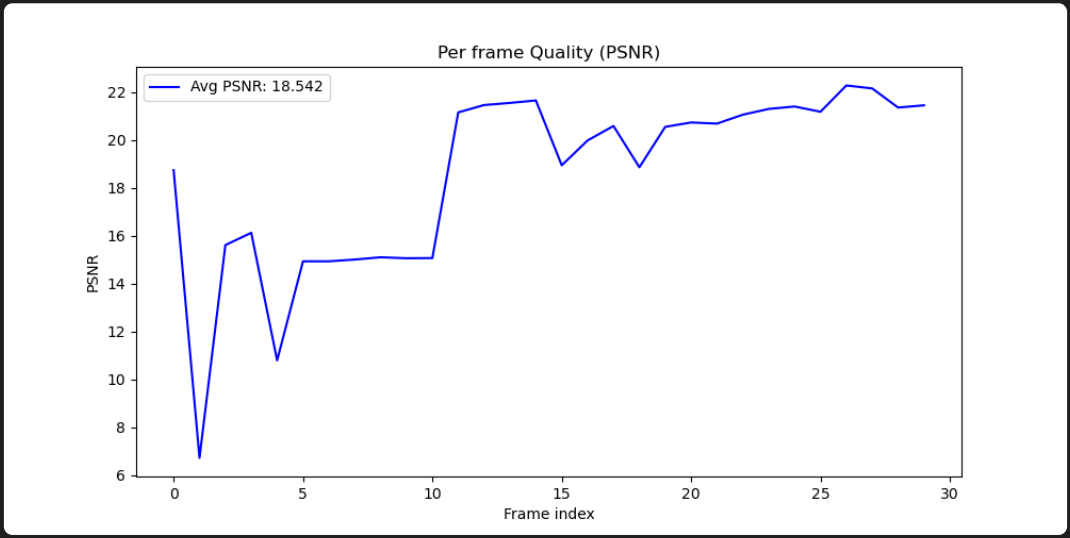




p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

c. Without KL annealing





p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

比較與分析

KL Annealing 的作用：KL Annealing 通常用來逐漸增加 KL divergence 的權重，這樣模型在訓練初期不會過度依賴於正則項(KL)，而更專注於重建誤差 (MSE loss)。在後期逐步增加 KL divergence 的權重，以迫使模型生成更具代表性的 latent variable 分佈。

- **Monotonic** 策略似乎能夠更穩定地訓練模型，使得Loss曲線更平滑，PSNR波動較小，顯示出模型生成影像的品質相對穩定。
- **Cyclical** 策略則會在不同的週期內對模型的學習進行調整，這種策略可能會導致 Loss 和 PSNR 出現更大幅度的波動，但在某些情況下，這種變化可能對探索不同的 latent variable 分佈有幫助。
- **沒有使用KL Annealing** 的情況下，模型容易過早收斂，並且模型可能會過擬合，導致生成的影像品質不如使用 KL Annealing 的情況(AVG PSNR 只有 18)。

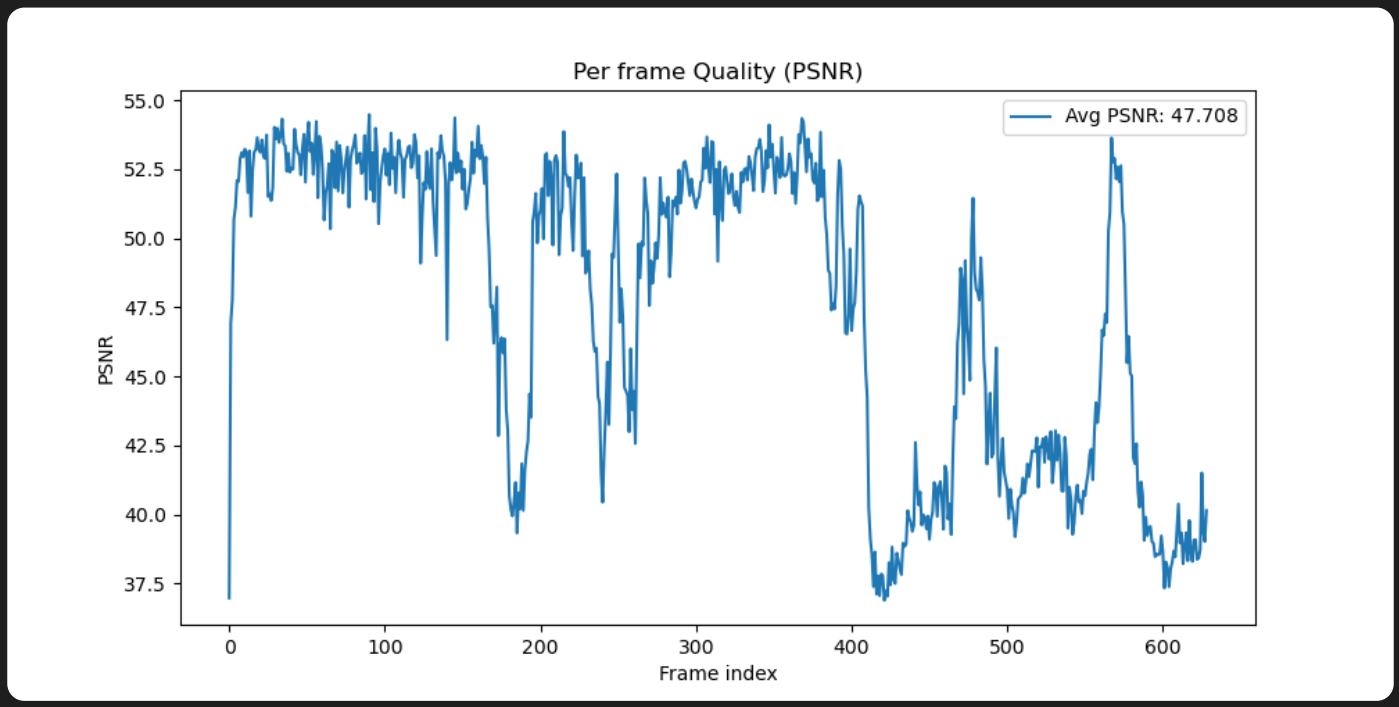
這些觀察指出，KL Annealing 是一種有效的技術，可以幫助模型在訓練過程中更好地平衡 MSE loss 和正則化項(KL divergence)，從而提升生成影像的品質和模型的穩定性。在具體應用中，選擇合適的KL Annealing策略（如Monotonic或Cyclical）需要根據數據和任務的特性來調整。

3. Plot the PSNR-per frame diagram in validation dataset

針對這個 section，我改良了 `Tester.py` 的程式，寫了一個 `val_PSNR.py` 的程式。參數的輸入和 `Tester.py` 的設定一樣，輸出則是會根據 validation dataset 產出 PSNR-per frame diagram。

```
def plot_psnr_curve(self, psnr_list):
    plt.figure(figsize=(10, 5))
    plt.plot(psnr_list, label=f'Avg PSNR: {np.mean(psnr_list):.3f}')
    plt.title('Per frame Quality (PSNR)')
```

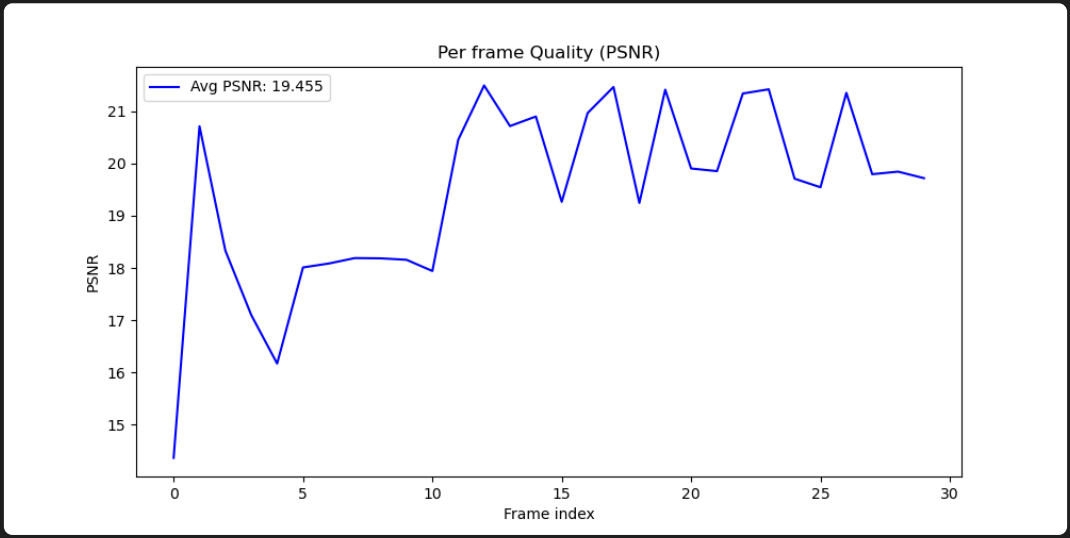
```
    plt.xlabel('Frame index')
    plt.ylabel('PSNR')
    plt.legend()
    plt.savefig(os.path.join(self.args.save_root, 'psnr_curve.png'))
    plt.close()

def compute_psnr(self, img1, img2, data_range=1.):
    mse = nn.functional.mse_loss(img1, img2)
    psnr = 20 * torch.log10(torch.tensor(data_range)) - 10 *
torch.log10(mse)
    return psnr.item()
```

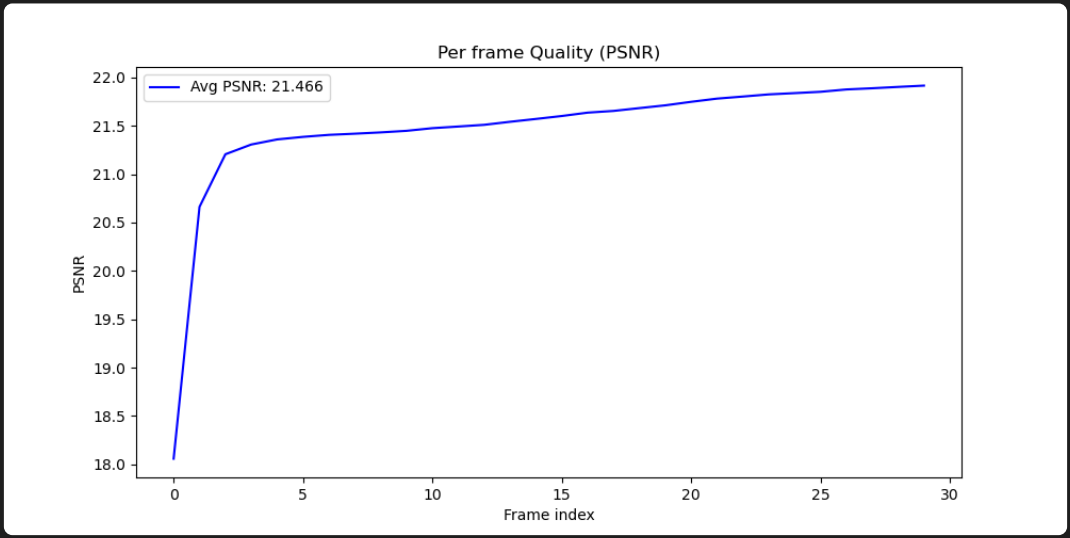


上圖為 kaggle 平台上測試最佳的模型，至於他的訓練策略會在下一個 section 提到。

4. Other training strategy analysis (Bonus)

先前有提到沒有 teacher forcing 的話，模型的訓練會更加穩定，如下圖(w & w/o):



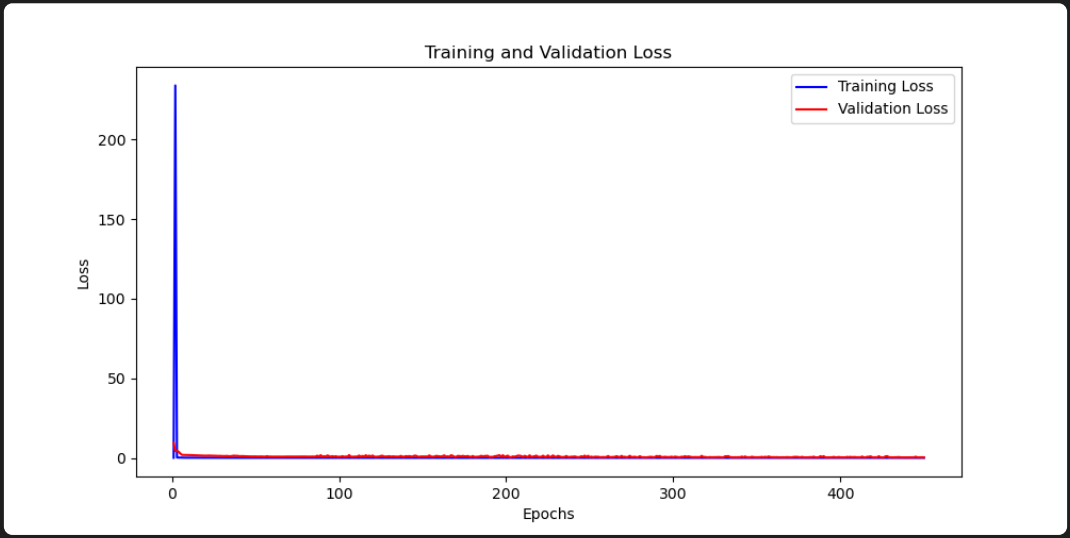


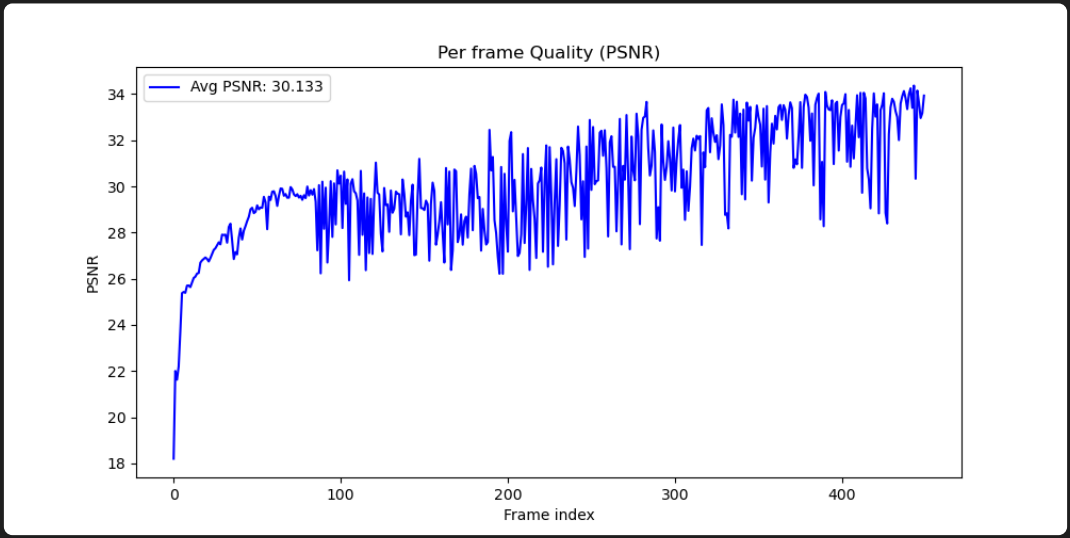
p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

會造成這樣，推測原因是**依賴性**：過早或過度使用 teacher forcing 可能會導致模型過於依賴真實數據，進而在測試時當無法依賴真實數據時表現變差。這種現象好像也被稱為「暴露偏差」（Exposure Bias），因為模型在訓練時未能學會處理自身生成的數據作為下一步的輸入。

但是使用 teacher forcing 也有它的好處在，它可以幫助模型有更好的學習。

因此，我修改了程式，讓 teacher forcing strategy 在訓練的中期 (約 100 epochs) 才出現，使得模型在訓練早期更多依賴自身的輸出來生成下一步的輸入，這使得模型能夠更好地應對自身生成數據的偏差，減少暴露偏差的風險。而以下為我的訓練成果:





p.s. (PSNR curve 的橫軸為 epochs，非 frame index)

感謝助教批改。