Comp Eng 4TN4

Project 2 – image Compression Pipeline Cameras

Abd Elelah Arafah - arafaha - 400197623

Course Code: CoE 4TN4

Term: Winter 2023

Date: April 12th, 2023

Introduction:

Images are a widely used technology and it is an important part of modern life as it has a wide range of applications, such as: social media and medical imaging. With that being said, we need the use of different image processing techniques to allow us to perform many necessary functions. One of which is image compression which will be explored in this paper.

Overview of Image Compression:

There are two different types of image compression methods: lossy and lossless. Lossless techniques result in a very accurate representation of the input image, while lossy compression techniques produce a degraded version of the input image.

Examples of lossy up-sampling methods:

- Bilinear interpolation
- Bicubic interpolation

Examples of Lossy down-sampling methods:

- Average sampling kernel
- N nearest neighbors

In this report, we explore another method for up-sampling and down-sampling to achieve a higher PSNR value and a higher SSIM value than the first phase of the project.

Proposed Method:

The proposed method for this phase of the project is performing both down-sampling and up-sampling with convolutional neural network models. The decision to utilize CNN models to perform the compression pipeline was done due to the following reasons:

- Model training: The ability for the model to learn, where we're able to train CNN models given a training dataset. This allows us to have a sampling method which is robust and generalized when designed, trained, and adjusted carefully.
- Non-linear predicted targets: During training, we can use non-linear activation functions to learn the different patterns and relationships between input and output pixels, where methods like bilinear interpolation might not be able to capture.
- Feature learning: Due to the design of convolutional neural network models, we have a better extraction of features between the input and output of the model.
- End-to-end optimization: The models can be adjusted to optimize the follow of the data between the inputs and outputs of each CNN model. During training, we are also able to monitor and minimize the Mean Squared Error to find the most optimized model from training.

Up-sampling CNN layers design:
2x-upsampling CNN design:

```
def CNN_design():
  input = Input(shape=(None, None, 1))
  x = Conv2D(32,3, activation=None, padding='same')(input)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(64,3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(128,3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = UpSampling2D(2)(x)

  x = Conv2D(64,3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(32,3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(1,3, activation=None, padding='same')(x)
  x = Activation('tanh')(x)
  x = x*127.5 + 127.5

  model = Model([input], x)
  model.summary()
  return model
```

This CNN contains 6 convolutional layers with the following details:

- Input layer: The input layer has (None, None, 1) as its parameters which allows for sampling images with any sizes for the input in gray scale.
- First Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3, attempting to optimize 320 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Second Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3, attempting to optimize 18496 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Third Convolutional layer: This Conv2D layer has 128 filters with a kernel size of 3x3, attempting to optimize 73856 parameters.
- Up-sampling: This UpSampling2D layer performs the 2x up-sampling (doubles the height and width).
- Fourth Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3.
- Fifth Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3.
- Sixth Convolutional layer: This output Conv2D layer has 1 filter with kernel size of 3x3. This layer has a tanh activation function which outputs a value between -1 to 1. Which is then followed by a line to map values into 0 to 255.

4x-upsampling CNN design:

```
def CNN_design():
  input = Input(shape=(None, None, 1))
  x = Conv2D(32, 3, activation=None, padding='same')(input)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(64, 3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(128, 3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = UpSampling2D(2)(x)

  x = Conv2D(128, 3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(64, 3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = UpSampling2D(2)(x)

  x = Conv2D(64, 3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(32, 3, activation=None, padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = Conv2D(1, 3, activation=None, padding='same')(x)
  x = Activation('tanh')(x)
  x = x*127.5 + 127.5

  model = Model([input], x)
  model.summary()
  return model
```

This CNN contains 8 convolutional layers with the following details:

- Input layer: The input layer has (None, None, 1) as its parameters which allows for sampling images with any sizes for the input in gray scale.
- First Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3, attempting to optimize 320 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Second Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3, attempting to optimize 18496 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Third Convolutional layer: This Conv2D layer has 128 filters with a kernel size of 3x3, attempting to optimize 73856 parameters.
- Up-sampling: This UpSampling2D layer performs the 2x up-sampling (doubles the height and width).
- Fourth Convolutional layer: This Conv2D layer has 128 filters with a kernel size of 3x3.
- Fifth Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3.
- Up-sampling: This UpSampling2D layer performs the 2x up-sampling (doubles the height and width).
- Sixth Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3.
- Seventh Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3.
- Eighth Convolutional layer: This output Conv2D layer has 1 filter with kernel size of 3x3. This layer has a tanh activation function which outputs a value between -1 to 1. Which is then followed by a line to map values into 0 to 255.

The main difference between the 2x and 4x up-sampling networks is that the 4x sampling CNN has an extra UpSampling2x to up sample the input image into 4x. It also has a few intermediate layers before the second UpSampling2x layer to better extract features.

Down-sampling CNN layers design:

2x-downsampling CNN design:

```
def CNN_design():
    input = Input(shape=(None, None, 1))
    x = Conv2D(32, 3, activation=None, padding='same')(input)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(64, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(128, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = MaxPooling2D(pool_size=(2, 2))(x)

    x = Conv2D(64, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(32, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(1, 3, activation=None, padding='same')(x)
    x = Activation('tanh')(x)
    x = x*127.5 + 127.5

    model = Model([input], x)
    model.summary()
    return model
```

This CNN contains 6 convolutional layers with the following details:

- Input layer: The input layer has (None, None, 1) as its parameters which allows for sampling images with any sizes for the input in gray scale.
- First Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3, attempting to optimize 320 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Second Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3, attempting to optimize 18496 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Third Convolutional layer: This Conv2D layer has 128 filters with a kernel size of 3x3, attempting to optimize 73856 parameters.
- Down-sampling: This MaxPooling2D layer performs the 2x down-sampling (halves the height and width) using a 2x2 pooling kernel.
- Fourth Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3.
- Fifth Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3.
- Sixth Convolutional layer: This output Conv2D layer has 1 filter with kernel size of 3x3. This layer has a tanh activation function which outputs a value between -1 to 1. Which is then followed by a line to map values into 0 to 255.

4x-downsampling CNN design:

```
def CNN_design():
    input = Input(shape=(None, None, 1))
    x = Conv2D(32, 3, activation=None, padding='same')(input)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(64, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(128, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)

    x = Conv2D(64, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(32, 3, activation=None, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(1, 3, activation=None, padding='same')(x)
    x = Activation('tanh')(x)
    x = x*127.5 + 127.5

    model = Model([input], x)
    model.summary()
    return model
```

This CNN contains 6 convolutional layers with the following details:

- Input layer: The input layer has (None, None, 1) as its parameters which allows for sampling images with any sizes for the input in gray scale.
- First Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3, attempting to optimize 320 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Second Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3, attempting to optimize 18496 parameters. This layer is followed by a batch normalization layer stabilize the results and it uses ReLU as the activation function.
- Third Convolutional layer: This Conv2D layer has 128 filters with a kernel size of 3x3, attempting to optimize 73856 parameters.
- Down-sampling: This MaxPooling2D layer performs the 2x down-sampling (halves the height and width) using a 2x2 pooling kernel.
- Down-sampling: This MaxPooling2D layer performs the 2x down-sampling (halves the height and width) using a 2x2 pooling kernel.
- Fourth Convolutional layer: This Conv2D layer has 64 filters with a kernel size of 3x3.
- Fifth Convolutional layer: This Conv2D layer has 32 filters with a kernel size of 3x3.
- Sixth Convolutional layer: This output Conv2D layer has 1 filter with kernel size of 3x3. This layer has a tanh activation function which outputs a value between -1 to 1. Which is then followed by a line to map values into 0 to 255.

The only difference between the 2x and 4x down-sampling CNN design is that we have 2 MaxPooling2D layers after each other to 4x down-sample the image.

For the up-sampling and Down-sampling CNN models, the 2x models are used to down and up sample the Y-channel of the images, while the 4x models are used to up and down sample the U-channel and V-channel. This is because most of the information of the images is retained in the Y-channel, therefore, we only down-sample the image by 2x to maintain as much of the information as possible, while still compressing the image reasonably well.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
mse_loss = 'mse'
train_model.compile(loss=mse_loss, optimizer=optimizer)

callbk_save_model = tf.keras.callbacks.ModelCheckpoint(
    'drive/MyDrive/training/model/2x_downsample_model.h5',
    monitor = 'val_loss',
    verbose=1,
    save_best_only=True,
    mode='min',
    save_freq='epoch'
)
error_callbk = tf.keras.callbacks.TensorBoard(log_dir='./drive/MyDrive/training/graph/2x_downsample/', histogram_freq=0, write_graph=True, write_i
callbk_earlyStop = EarlyStopping(monitor='val_loss', patience=5, verbose=1, mode='min', restore_best_weights=True)
```

When training the images, we keep track of the MSE error to minimize it, we see that as we increase the number of epochs, the error of the model forms a decaying exponential function as per the Mean square root error chosen to train the model:

In addition, from the code snippet above, we see that the models are defined with an early stoppage condition callback to avoid over training the model. The early stoppage is configured so that if \the training error does not improve in 5 epochs then it stops training, and it refers back to the model with the lowest error and saves it.

Also, to test the model, a custom SSIM function was implemented to be used as a metric to measure the accuracy of the function. The code of the SSIM function is defined as seen below:

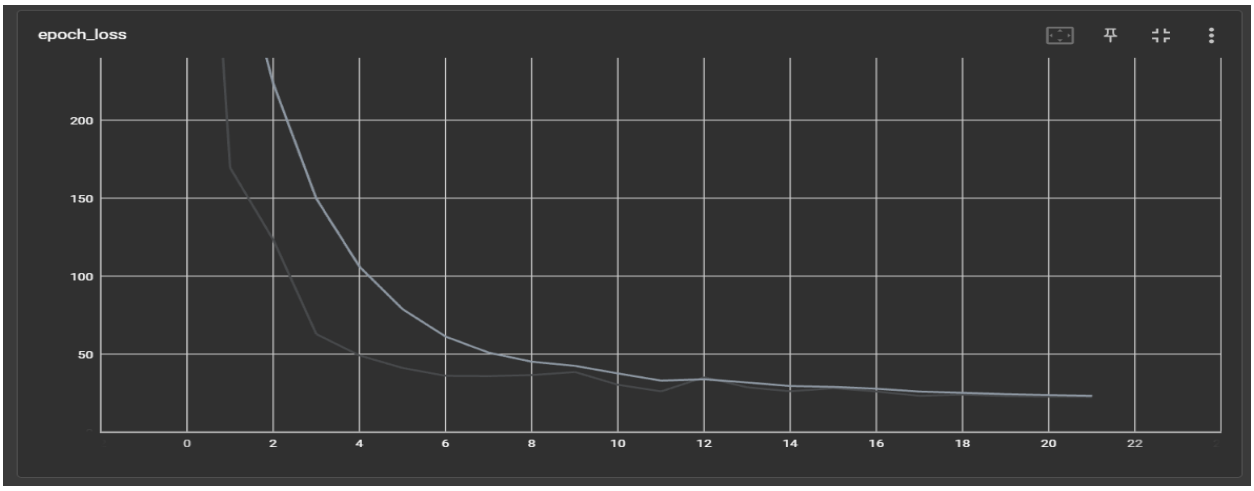
```
def create_gaussian_kernel(kernel_size, sigma_value=1.5):
    # Generate Gaussian kernel for the given kernel size and sigma value
    kx = cv2.getGaussianKernel(kernel_size, sigma_value)
    ky = cv2.getGaussianKernel(kernel_size, sigma_value)

    # Create a 2D Gaussian kernel using the outer product of the 1D Gaussian kernels
    two_d_kernel = np.outer(kx, ky)

    return two_d_kernel

def compute_ssim_metric(img_a, img_b, window_size=11, const_1=0.01, const_2=0.03, sigma=1.5, dynamic_range=1):
    if img_a.shape != img_b.shape:
        raise ValueError("Input images must have the same dimensions.")

    # Constants for SSIM calculation
    Constant_1 = (const_1 * dynamic_range) ** 2
    Constant_2 = (const_2 * dynamic_range) ** 2
    gaussian_kernel = create_gaussian_kernel(window_size, sigma)

    # Compute the mean of each image using the Gaussian kernel
    mean_a = cv2.filter2D(img_a, -1, gaussian_kernel)
    mean_b = cv2.filter2D(img_b, -1, gaussian_kernel)

    mean_a_sq = mean_a ** 2
    mean_b_sq = mean_b ** 2
    mean_a_mean_b = mean_a * mean_b

    # Compute the variance of each image and the covariance between them
    var_a = cv2.filter2D(img_a * img_a, -1, gaussian_kernel) - mean_a_sq
    var_b = cv2.filter2D(img_b * img_b, -1, gaussian_kernel) - mean_b_sq
    cov_ab = cv2.filter2D(img_a * img_b, -1, gaussian_kernel) - mean_a_mean_b

    # Compute the SSIM map and the mean SSIM value
    ssim_map = ((2 * mean_a_mean_b + Constant_1) * (2 * cov_ab + Constant_2)) / ((mean_a_sq + mean_b_sq + Constant_1) * (var_a + var_b + Constant_2))
    ssim_mean = np.mean(ssim_map)

    return ssim_mean
```

The function above calculates the Structural Similarity Index Measure (SSIM) between 2 images, which gives a quantitative value for the luminance, contrast, and structure. It first compares the dimensions of

the input images, next it computes the mean, variance and covariance of the input images using the gaussian kernel. The SSIM map is calculated using these stats along with the predefined constants. Then the function returns the SSIM value.

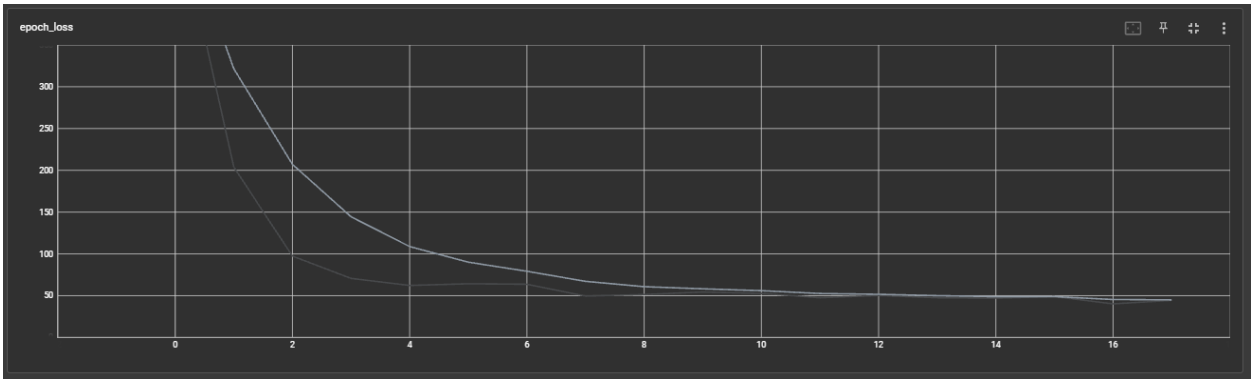
Experimental Results:

The MSE graph for the 2x up-sampling CNN:



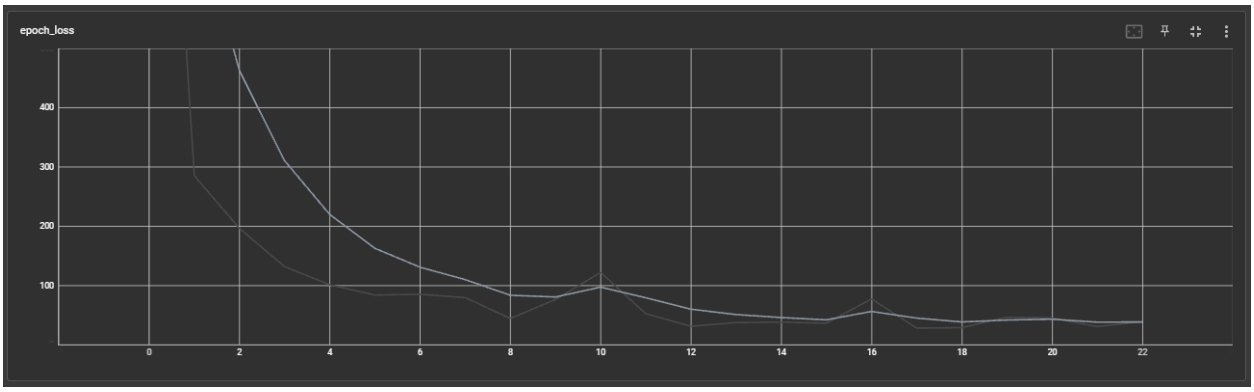

```
5/5 [==============================] - 0s 9ms/step
 - val_ssim: 1.8632
5/5 [==============================] - 0s 9ms/step
 - val_psnr: 25.6513
160/160 [==============================] - 2s 10ms/step - loss: 281.0102 - val_loss: 223.0141 - val_ssim: 1.8632 - val_psnr: 25.6513
Epoch 10: early stopping
1/1 [==============================] - 0s 483ms/step
downsampled: (256, 384)
```

The MSE graph for the 4x up-sampling CNN:



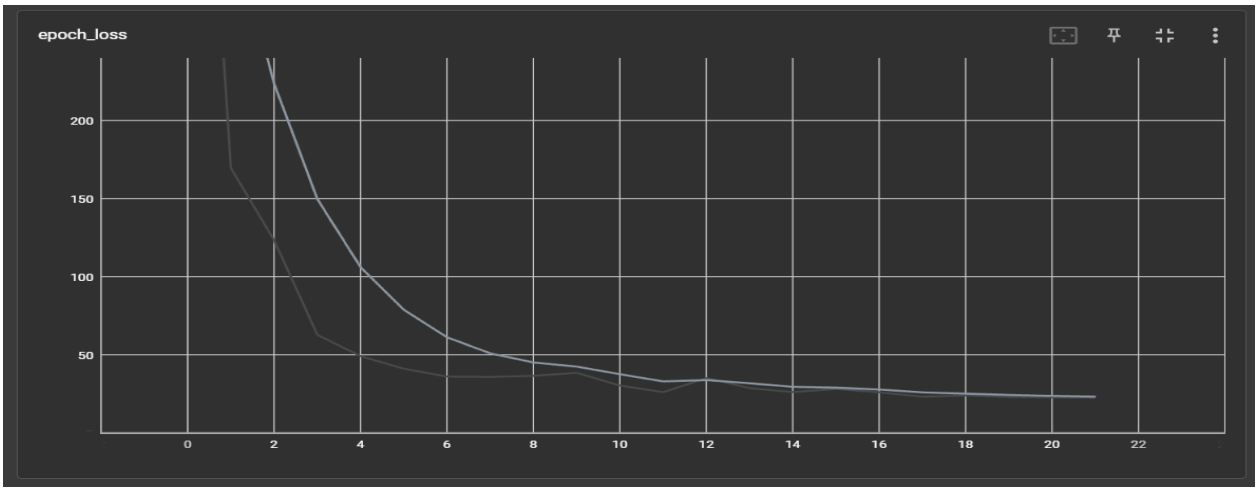

```
Epoch 17: val_loss did not improve from 24.12920
160/160 [==============================] - 3s 20ms/step - loss: 39.8569 - val_loss: 25.4501
Epoch 18/100
160/160 [==============================] - ETA: 0s - loss: 44.0401
Epoch 18: val_loss did not improve from 24.12920
Restoring model weights from the end of the best epoch: 13.
160/160 [==============================] - 3s 20ms/step - loss: 44.0401 - val_loss: 27.6842
Epoch 18: early stopping
1/1 [==============================] - 1s 1s/step
```

The MSE graph for the 2x down-sampling CNN:




```
Epoch 21/100
156/160 [============================>.] - ETA: 0s - loss: 125.3016
Epoch 21: val_loss did not improve from 27.76598
160/160 [==============================] - 2s 12ms/step - loss: 125.1307 - val_loss: 44.9094
Epoch 22/100
160/160 [==============================] - ETA: 0s - loss: 117.2289
Epoch 22: val_loss did not improve from 27.76598
160/160 [==============================] - 2s 12ms/step - loss: 117.2289 - val_loss: 30.3219
Epoch 23/100
156/160 [============================>.] - ETA: 0s - loss: 113.4013
Epoch 23: val_loss did not improve from 27.76598
Restoring model weights from the end of the best epoch: 18.
160/160 [==============================] - 2s 12ms/step - loss: 114.3543 - val_loss: 38.6272
Epoch 23: early stopping
1/1 [==============================] - 1s 717ms/step
```

The MSE graph for the 4x down-sampling CNN:




```
Epoch 21/100
160/160 [==============================] - ETA: 0s - loss: 22.9592
Epoch 21: val_loss did not improve from 6.14691
160/160 [==============================] - 23s 142ms/step - loss: 22.9592 - val_loss: 7.3679
Epoch 22/100
160/160 [==============================] - ETA: 0s - loss: 22.7285
Epoch 22: val_loss did not improve from 6.14691
Restoring model weights from the end of the best epoch: 17.
160/160 [==============================] - 23s 142ms/step - loss: 22.7285 - val_loss: 6.1602
Epoch 22: early stopping
<keras.callbacks.History at 0x7fd9602bce80>
```

As mentioned before while training the images, we keep track of the MSE error to minimize it, we see that as we increase the number of epochs, the error of the model forms a decaying exponential function as per the Mean square root error chosen to train the model. We notice that that all models take less than 22 epochs to find the optimized model.

Results using sample images:

Input image 1:

Input image 2:

Input image 3:

Input image 4:

Input image 5:

Input image 6:

Input image 7:

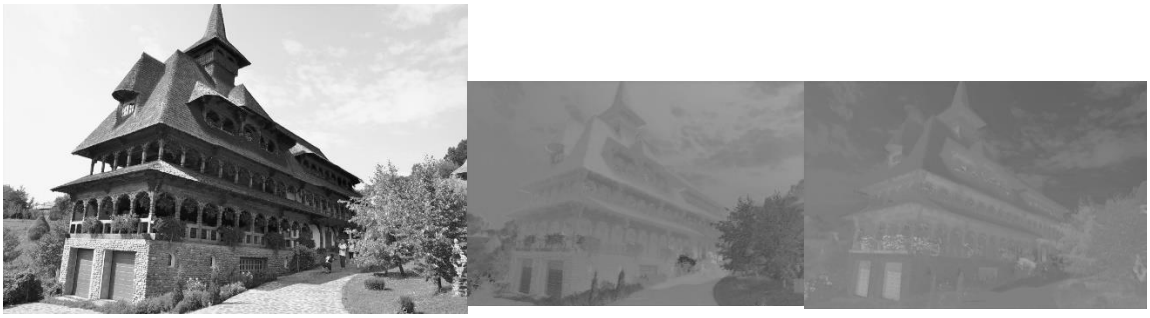
Down-sampled YUV channels of input images:

YUV #1:

YUV #2:

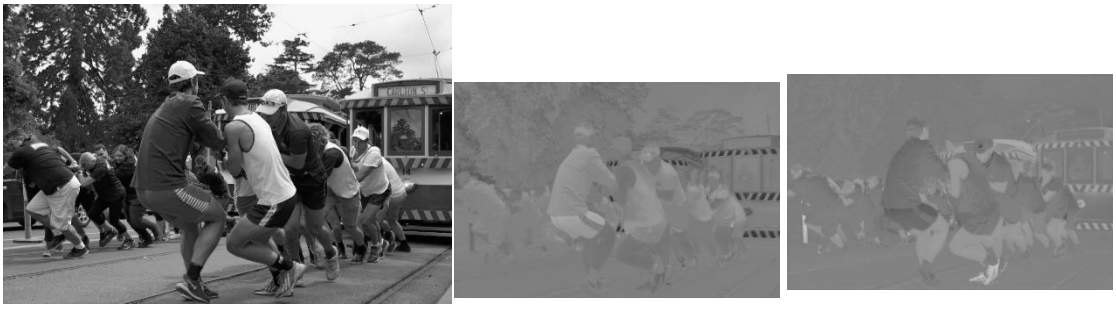
YUV #3:

YUV #4:



YUV #5:

YUV #6:



YUV #7:

Reconstructed images:

Reconstructed #1:

```
 14%|          | 177 [00:34
Metrics for image 1:-

PSNR:-  31.763492194663492
SSIM:-  0.90875757
```

Reconstructed #2:

```
Metrics for image 2:-

PSNR:-  29.88306965507203
SSIM:-  0.8398596
```

Reconstructed #3:

```
Metrics for image 3:-

PSNR:-  29.033977786552736
SSIM:-  0.8420107
```

Reconstructed #4:

```
Metrics for image 4:-

PSNR:-  30.21363865839552
SSIM:-  0.86173576
```

Reconstructed #5:


```
Metrics for image 5:-

PSNR:-  30.35622706959312
SSIM:-  0.86896646
```

Reconstructed #6:


```
Metrics for image 6:-

PSNR:-  30.343856327401934
SSIM:-  0.88862544
```

Reconstructed #7:

```
Metrics for image 7:-

PSNR:-  29.26177500361974
SSIM:-  0.7568526
```

The average metric for 7 random sample images from the DIV2K database:

```
Averaging SSIM value: 0.8524011628968375
Averaging PSNR value: 30.122290956471225
```

In conclusion, comparing these results to the previous stage of the project, we notice that we achieve a slightly higher PSNR value with around 2-3 dB in compared to the first phase. We notice also that the combination of the down-sampling and up-sampling using CNN models produces a fairly accurate result with an average 0.852 value.