4TN4 Phase 2: Image Compression Pipeline

Eric Nguyen
McMaster University
1280 Main St W, Hamilton, ON L8S 4L8
Nguyee13@McMaster.ca

Abstract

Phase 2 is an improvement on the simple lossy image compression method built in Phase 1. The image compression is still performed by using a color transformation between the RGB and YUV color spaces. However, the image decoding and inverse color transformation is performed by a convolutional neural network instead of using bilinear interpolation or bicubic interpolation methods. The reason why the input image is transformed into the YUV color space is because YUV images can be compressed at a greater factor than RGB images while preserving the same amount of digital information. This behavior is due to the fact that the red, green, and blue channels in the RGB color space are more correlated to one another compared to the luminance (Y), chrominance (U, V) channels in the YUV color space. The luminance (Y) channel also stores more meaningful information than the chrominance channels as it represents the brightness and contrast of the image which is crucial to a person's perception of an image. This fact is taken advantage of by down sampling the U and V channels twice as much as the Y channel during the encoding process. The resulting encoded YUV channel images are then fed into a convolutional neural network that is specifically trained to convert YUV channel images into a 512x512 RGB image. After testing this method on a dataset of images, the resulting peak signal-to-noise ratio and structural similarity index of this method has greatly improved compared to the phase 1 compression pipeline. The significant increase in image quality is due to the flaw found in phase 1's approach to decoding an image. Machine learning models can restore fine details and the edges of encoded images, which is where bilinear and bicubic interpolation techniques perform poorly. Although this method is much more computationally expensive than phase 1's method, the significant improvement of image quality opens the possibilities for great breakthroughs in image processing techniques.

1. Introduction

Fundamental to all forms of technology is the transmission and storage of files. There is a critical need to compress information, particularly in the modern day where data is produced at a faster rate than we can usefully process. Phase 2 is an improvement to Phase 1's implementation of an image compression pipeline by using a convolutional neural network. The flow of the pipeline remains the same, which is to compress an RGB image by first converting the image to a YUV color space. The resulting YUV image is then split into its three channels, where the chrominance (U,V) channels are down sampled twice as much as the luminance (Y) channel. The difference between Phase 1 and Phase 2's implementation lies in the image decoding or up-sampling method. Instead of using bilinear interpolation to decode the down-sampled YUV channels, a convolutional neural network is used instead. This convolutional neural network has been specifically trained and tuned to receive the down sampled YUV channel images and return a 512x512 RGB image. The reasoning behind each step will be explained with source code, mathematical equations, and diagrams.

1.1. Program Structure

The goal is to implement an image compression method without the use of external built-in functions from libraries such as OpenCV. However, NumPy is used for matrix and vector operations such as storing images and transforming color spaces. Python Image Library (PIL) was also used for easier image loading and saving. TensorFlow is also used to construct and train the convolutional neural network on the DIV2K dataset.

The structure of this image compression method is split into six steps and the modules are executed/called in the following order:

Step 1: Load the Original Image

- *Main.py*: This is the main program file that the user executes. This implements the IO pipeline for the user to input their RGB image. This file also calls the necessary functions to perform the remaining image transformations and prints the image comparison metrics.

Step 2: Transform Image from RGB to YUV

- *Colour_conversion.py:* This module contains the *rgb_to_yuv* function which loops through each RGB pixel of the input image and extracts the corresponding YUV channel values.

Step 3: Down sample YUV Image

- *downsample.py:* Implements the down sampling of a YUV image with the *down_sample* function. Down sampling is done using the averaging of nearby pixels. This particular pipeline down samples channel Y by a factor of two and channels U, V by 4. This function returns the down-sampled Y, U, V channels.

Step 4: Pre-process Dataset for Training

- *process_data_phase1.py:* This script creates the necessary directories to access and store the processed dataset. The dataset is split into training and validation. Each image in the dataset is first resized to a uniform shape of 512x512. They are then converted to the YUV color space and down sampled to form the training/validation datasets. Each YUV channel image sets, and RGB label images are stored and saved in corresponding NumPy arrays. These NumPy arrays will be used in the CNN training process.

Step 5: Construct and Train CNN

- *train_CNN_2.py:* This script constructs and compiles the CNN model. The training data NumPy arrays generated in process_data_phase1.py is loaded and used to train the CNN. Once training is completed, the model is saved for future predictions in main.py. This module also contains function model_predict(model, y_img, u_img, v_img), which receives a model and YUV channel images to return the saved model's prediction. A pyplot is also generated to display the training and validation loss during the training process.

Step 6: Load Saved Model and Predict

- *main.py:* The trained model is loaded and calls model_predict() from train_CNN_2 to produce an upscaled RGB image from the down sampled YUV channel images.

Step 7: Compute PSNR

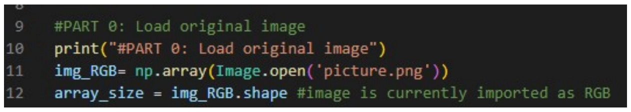
- *Psnr_mse.py:* The function *MSE_PSNR* computes the mean square error and peak signal-to-noise ratio by comparing the pixels of both the original and final decoded image. Both values of MSE and PSNR are returned when the function is called.

Step 8: Compute SSIM

- *SSIM.py:* The function SSIM is called from main.py and returns the structural similarity index of the original RGB image and the model's RGB prediction output.

1.2. Loading the Original Image

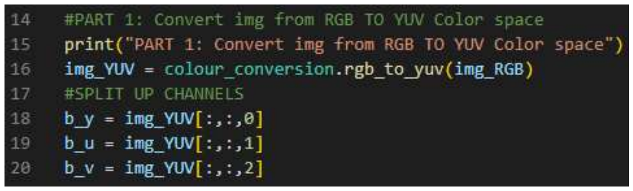
The following code is in main.py, which is how the user inputs their target input RGB image file. The image is opened from the source directory using PIL and is immediately converted into a NumPy array.



```
#PART 0: Load original image
print("#PART 0: Load original image")
img_RGB= np.array(Image.open('picture.png'))
array_size = img_RGB.shape #image is currently imported as RGB
```

1.3. Color Space Transformation

Once the user inputs a valid image, *rgb_to_yuv* is immediately called from main.py.



```
#PART 1: Convert img from RGB TO YUV Color space
print("PART 1: Convert img from RGB TO YUV Color space")
img_YUV = colour_conversion.rgb_to_yuv(img_RGB)
#SPLIT UP CHANNELS
b_y = img_YUV[:,:,0]
b_u = img_YUV[:,:,1]
b_v = img_YUV[:,:,2]
```

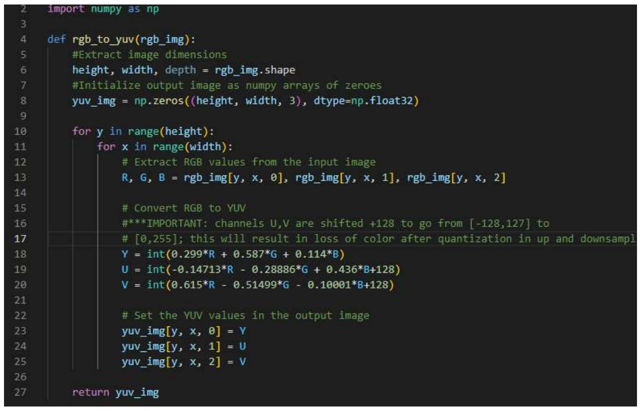
Rgb_to_yuv(rgb_img) creates an empty NumPy array with the same dimensions as the original RGB image. Note the original RGB image dimensions is Height x Width x 3 (3 color channels: R, G, B). Each pixel of the input image is looped over, and the corresponding R, G, B channel value is saved. Each corresponding YUV channel value of the pixel is calculated by multiplying specific coefficients with each RGB channel value. [1] Note that channels U, V are a linear transformation of the R, G, B channels and can potentially result in a negative value. As a result, U and V are shifted by +128 to shift the range from [-128,127] to [0,255] to match the range of an 8-bit image.

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{1}$$

$$U = -0.14713 * R - 0.28886 * G + 0.436 * B + 128 \tag{2}$$

$$V = 0.612 * R - 0.51499 * G - 0.10001 * B + 128 \tag{3}$$

Once each YUV value is computed, it is stored in the corresponding indices in the originally empty NumPy array yuv_img. Once the nested for loops are completed, yuv_img is returned.

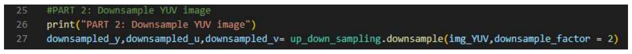


```python
import numpy as np

def rgb_to_yuv(rgb_img):
    #Extract image dimensions
    height, width, depth = rgb_img.shape
    #Initialize output image as numpy arrays of zeroes
    yuv_img = np.zeros((height, width, 3), dtype=np.float32)

    for y in range(height):
        for x in range(width):
            # Extract RGB values from the input image
            R, G, B = rgb_img[y, x, 0], rgb_img[y, x, 1], rgb_img[y, x, 2]

            # Convert RGB to YUV
            #***IMPORTANT: channels U,V are shifted +128 to go from [-128,127] to
            # [0,255]; this will result in loss of color after quantization in up and downsampl
            Y = int(0.299*R + 0.587*G + 0.114*B)
            U = int(-0.14713*R - 0.28886*G + 0.436*B+128)
            V = int(0.615*R - 0.51499*G - 0.10001*B+128)

            # Set the YUV values in the output image
            yuv_img[y, x, 0] = Y
            yuv_img[y, x, 1] = U
            yuv_img[y, x, 2] = V

    return yuv_img
```

1.4. Down Sampling YUV Image

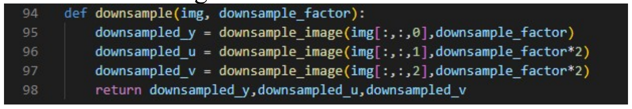
Down sampling the YUV image is simply resizing each channel by a specific factor. In this case, the chrominance channels U, V are down sampled twice as much as the luminance channel Y. The reason for this design choice is because the human visual system extracts much more meaningful information from the Y channel compared to the chrominance channels. This means that the UV channels can be compressed much more without sacrificing image quality. This is the sole reason why the image was converted from RGB to YUV.

The function downsample is called from main.py with the YUV image and a downsample factor of 2 as inputs. Three down sampled YUV channels are returned from this function call.



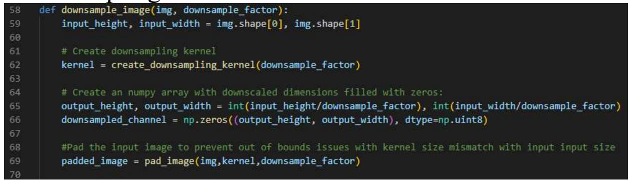
```python
#PART 2: Downsample YUV image
print("PART 2: Downsample YUV image")
downsampled_y,downsampled_u,downsampled_v= up_down_sampling.downsample(img_YUV,downsample_factor = 2)
```

Downsample is a wrapper function that calls downsample_image to downs sample all 3 YUV channels. Note how channels U and V are down sampled with a factor twice as large as Y.

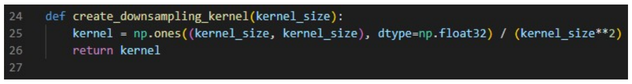


```python
def downsample(img, downsample_factor):
    downsampled_y = downsample_image(img[:,:,0],downsample_factor)
    downsampled_u = downsample_image(img[:,:,1],downsample_factor*2)
    downsampled_v = downsample_image(img[:,:,2],downsample_factor*2)
    return downsampled_y,downsampled_u,downsampled_v
```

The function downsample_image will call two functions, create_downsampling_kernel and pad_image. Create_downsampling_kernel creates a numpy array with the shape of (downsample_factor X downsample_factor). Pad_image will pad the input image with zeroes to ensure no out of bounds issues during the convolution downsampling.

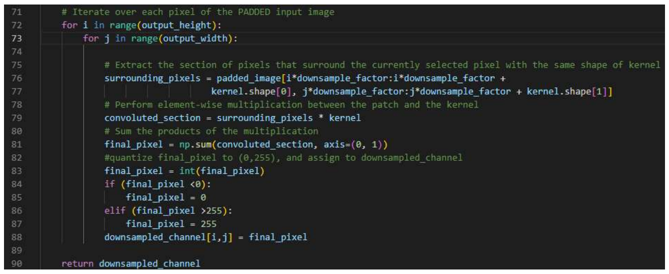


```python
def downsample_image(img, downsample_factor):
    input_height, input_width = img.shape[0], img.shape[1]

    # Create downsampling kernel
    kernel = create_downsampling_kernel(downsample_factor)

    # Create an numpy array with downscaled dimensions filled with zeros:
    output_height, output_width = int(input_height/downsample_factor), int(input_width/downsample_factor)
    downsampled_channel = np.zeros((output_height, output_width), dtype=np.uint8)

    #Pad the input image to prevent out of bounds issues with kernel size mismatch with input input size
    padded_image = pad_image(img,kernel,downsample_factor)

```



```python
def create_downsampling_kernel(kernel_size):
    kernel = np.ones((kernel_size, kernel_size), dtype=np.float32) / (kernel_size**2)
    return kernel

```

A kernel is created to perform nearest neighbor averaging, which is averaging the dot product of a section of the input image and the kernel. For example, a 2x2 pixel section of the original image would be averaged to become the value of a single pixel in the down sampled image.



```python
    # Iterate over each pixel of the PADDED input image
    for i in range(output_height):
        for j in range(output_width):

            # Extract the section of pixels that surround the currently selected pixel with the same shape of kernel
            surrounding_pixels = padded_image[i*downsample_factor:i*downsample_factor +
                                     kernel.shape[0], j*downsample_factor:j*downsample_factor + kernel.shape[1]]
            # Perform element-wise multiplication between the patch and the kernel
            convoluted_section = surrounding_pixels * kernel
            # Sum the products of the multiplication
            final_pixel = np.sum(convoluted_section, axis=(0, 1))
            #quantize final_pixel to (0,255), and assign to downsampled_channel
            final_pixel = int(final_pixel)
            if (final_pixel <0):
                final_pixel = 0
            elif (final_pixel >255):
                final_pixel = 255
            downsampled_channel[i,j] = final_pixel

    return downsampled_channel
```

In this code, an image is down sampled by looping through each pixel of the image. In this nested for loop, the kernel is mapped onto the current pixel and nearby pixels to compute an average value of that 2x2 section. The average of the 2x2 section is then quantized to an integer value between 0 and 255 and is copied to the output array downsampled_channel that represents the encoded image. On the subsequent iterations, the kernel is shifted, and a new average is computed until the entire original input image has been down sampled.

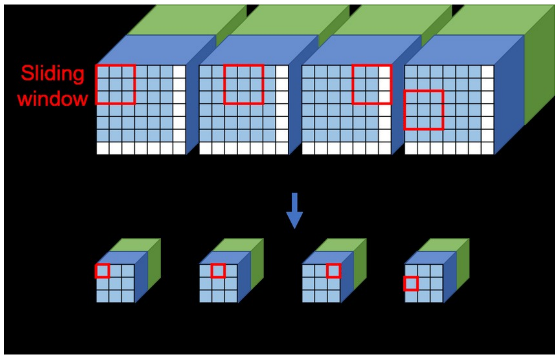
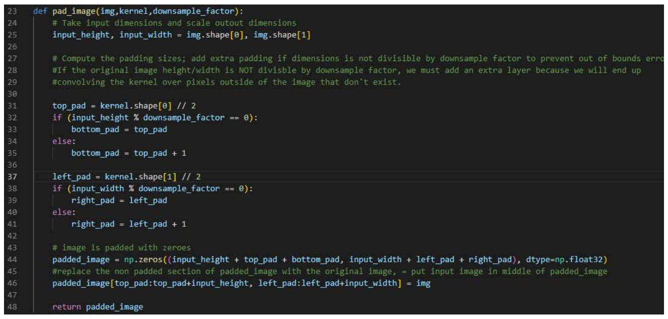


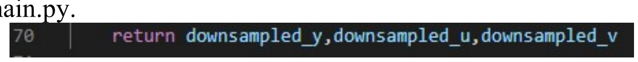
Figure 1: The kernel acts as a "sliding window" that is used to compute the average of the section of pixels.

Recall that the wrapper function downsample() also called pad_image(). This was done to ensure that the image being down sampled has dimensions divisible by the down sample factor. If the input image were to not be padded, there may be an out of bounds error where the kernel may be overlapping with pixels “outside” of the input image that do not exist.



```python
def pad_image(img,kernel,downsample_factor):
    # Take input dimensions and scale outout dimensions
    input_height, input_width = img.shape[0], img.shape[1]

    # Compute the padding sizes; add extra padding if dimensions is not divisible by downsample factor to prevent out of bounds erro
    #If the original image height/width is NOT divisble by downsample factor, we must add an extra layer because we will end up
    #convolving the kernel over pixels outside of the image that don't exist.

    top_pad = kernel.shape[0] // 2
    if (input_height % downsample_factor == 0):
        bottom_pad = top_pad
    else:
        bottom_pad = top_pad + 1

    left_pad = kernel.shape[1] // 2
    if (input_width % downsample_factor == 0):
        right_pad = left_pad
    else:
        right_pad = left_pad + 1

    # image is padded with zeroes
    padded_image = np.zeros((input_height + top_pad + bottom_pad, input_width + left_pad + right_pad), dtype=np.float32)
    #replace the non padded section of padded_image with the original image, = put input image in middle of padded_image
    padded_image[top_pad:top_pad+input_height, left_pad:left_pad+input_width] = img

    return padded_image
```

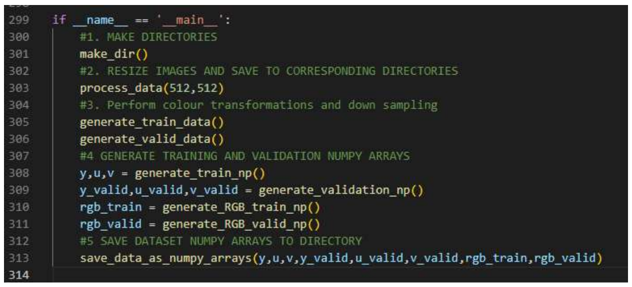
Once these three down sampling steps are completed for each Y, U, V channel, the three images are returned to main.py.



```python
    return downsampled_y,downsampled_u,downsampled_v
```

1.5. Data Preprocessing

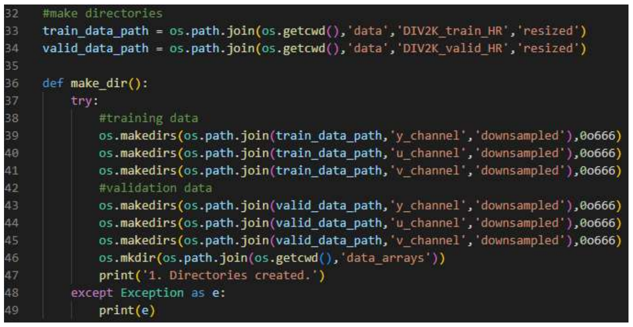
The machine learning model is only as good as the data it is trained on, which makes it crucial to preprocess the dataset. There are five main steps performed in this script:



```python
if __name__ == '__main__':
    #1. MAKE DIRECTORIES
    make_dir()
    #2. RESIZE IMAGES AND SAVE TO CORRESPONDING DIRECTORIES
    process_data(512,512)
    #3. Perform colour transformations and down sampling
    generate_train_data()
    generate_valid_data()
    #4 GENERATE TRAINING AND VALIDATION NUMPY ARRAYS
    y,u,v = generate_train_np()
    y_valid,u_valid,v_valid = generate_validation_np()
    rgb_train = generate_RGB_train_np()
    rgb_valid = generate_RGB_valid_np()
    #5 SAVE DATASET NUMPY ARRAYS TO DIRECTORY
    save_data_as_numpy_arrays(y,u,v,y_valid,u_valid,v_valid,rgb_train,rgb_valid)
```

1.5.1 Directory Creation

When the images are preprocessed for the model training, they will be saved to either a training or validation dataset. Each image will also be converted to the YUV color space and each down-sampled channel will be saved as a separate image.



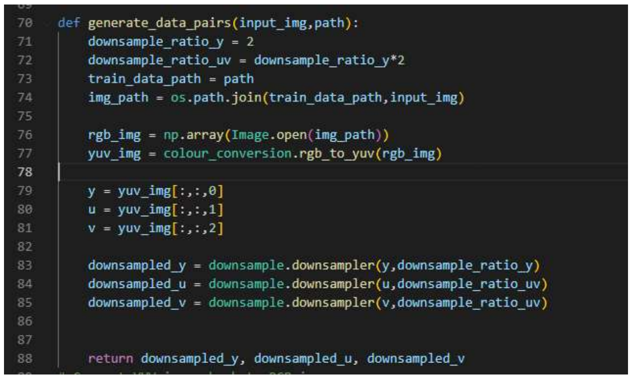
```python
#make directories
train_data_path = os.path.join(os.getcwd(),'data','DIV2K_train_HR','resized')
valid_data_path = os.path.join(os.getcwd(),'data','DIV2K_valid_HR','resized')

def make_dir():
    try:
        #training data
        os.makedirs(os.path.join(train_data_path,'y_channel','downsampled'),0o666)
        os.makedirs(os.path.join(train_data_path,'u_channel','downsampled'),0o666)
        os.makedirs(os.path.join(train_data_path,'v_channel','downsampled'),0o666)
        #validation data
        os.makedirs(os.path.join(valid_data_path,'y_channel','downsampled'),0o666)
        os.makedirs(os.path.join(valid_data_path,'u_channel','downsampled'),0o666)
        os.makedirs(os.path.join(valid_data_path,'v_channel','downsampled'),0o666)
        os.mkdir(os.path.join(os.getcwd(),'data_arrays'))
        print('1. Directories created.')
    except Exception as e:
        print(e)
```

1.5.2 Resize Source Images

The DIV2K dataset contains 900 images of varying dimensions which will all be resized to 512x512 images. The reason why these images are resized is because the model being trained in this implementation requires a dataset of uniform shape.

1.5.3 Generate Training and Validation Datasets

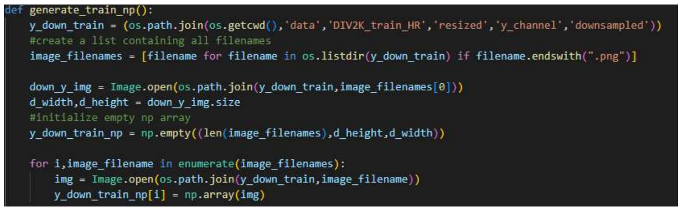
The resized images are now transformed to the YUV color space and down-sampled. This function converts an input image to the YUV color space, and down-samples each individual channel. The color conversion and down-sampling are performed by calling functions from colour_conversion.py and downsample.py. This function is called for all images in both the resized training and validation datasets.



```python
def generate_data_pairs(input_img,path):
    downsample_ratio_y = 2
    downsample_ratio_uv = downsample_ratio_y*2
    train_data_path = path
    img_path = os.path.join(train_data_path,input_img)

    rgb_img = np.array(Image.open(img_path))
    yuv_img = colour_conversion.rgb_to_yuv(rgb_img)

    y = yuv_img[:,:,0]
    u = yuv_img[:,:,1]
    v = yuv_img[:,:,2]

    downsampled_y = downsample.downsampler(y,downsample_ratio_y)
    downsampled_u = downsample.downsampler(u,downsample_ratio_uv)
    downsampled_v = downsample.downsampler(v,downsample_ratio_uv)


    return downsampled_y, downsampled_u, downsampled_v
```

1.5.4 Convert Training and Validation Images to NumPy

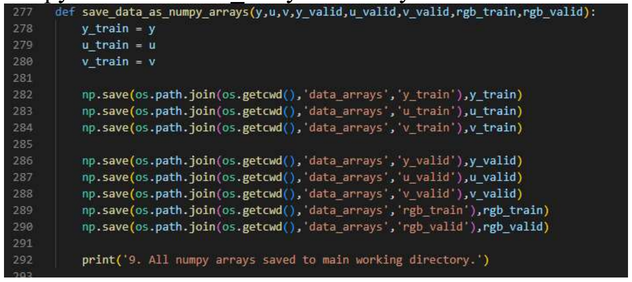
TensorFlow models cannot be trained on images directly. These images must be converted to NumPy arrays to train the model. The generate_train_np() function iterates through each down-sampled Y, U, V image set in the training and validation data directories. This particular section of code displays the initialization of the NumPy array 'y_down_train_np'. Each Y channel image of the training dataset will be converted to a NumPy array representation and is appended to this array. This is repeated for all Y, U, V color channels and the resized RGB images in both training and validation datasets.



```python
def generate_train_np():
    y_down_train = (os.path.join(os.getcwd(),'data','DIV2K_train_HR','resized','y_channel','downsampled'))
    #create a list containing all filenames
    image_filenames = [filename for filename in os.listdir(y_down_train) if filename.endswith(".png")]

    down_y_img = Image.open(os.path.join(y_down_train,image_filenames[0]))
    d_width,d_height = down_y_img.size
    #initialize empty np array
    y_down_train_np = np.empty((len(image_filenames),d_height,d_width))

    for i,image_filename in enumerate(image_filenames):
        img = Image.open(os.path.join(y_down_train,image_filename))
        y_down_train_np[i] = np.array(img)
```

1.5.5 Save NumPy Arrays to Directory

Once the entire dataset has been converted and saved to their respective NumPy array, these arrays will be saved as a .npy file in the data_arrays directory.



```python
def save_data_as_numpy_arrays(y,u,v,y_valid,u_valid,v_valid,rgb_train,rgb_valid):
    y_train = y
    u_train = u
    v_train = v

    np.save(os.path.join(os.getcwd(),'data_arrays','y_train'),y_train)
    np.save(os.path.join(os.getcwd(),'data_arrays','u_train'),u_train)
    np.save(os.path.join(os.getcwd(),'data_arrays','v_train'),v_train)

    np.save(os.path.join(os.getcwd(),'data_arrays','y_valid'),y_valid)
    np.save(os.path.join(os.getcwd(),'data_arrays','u_valid'),u_valid)
    np.save(os.path.join(os.getcwd(),'data_arrays','v_valid'),v_valid)
    np.save(os.path.join(os.getcwd(),'data_arrays','rgb_train'),rgb_train)
    np.save(os.path.join(os.getcwd(),'data_arrays','rgb_valid'),rgb_valid)

    print('9. All numpy arrays saved to main working directory.')
```

1.6. Construct and Train CNN

A convolutional neural network is a type of deep neural network commonly used for image classification and recognition. A convolutional neural network consists of a series of layers that perform different operations on the input data. The purpose of these connected layers is for the neural network to learn increasingly complex representations of the input data. Ultimately, the goal of a neural network is to eventually learn the relationships between the input training data and the ground truths to make accurate predictions on new data. In this case, the CNN is being trained to determine the relationship between the down-sampled Y, U, V channel images and the original RGB image.

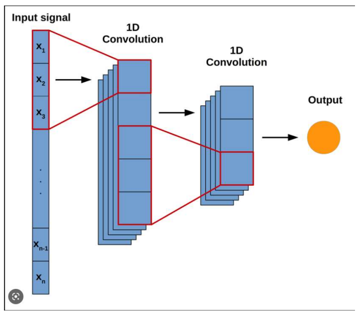
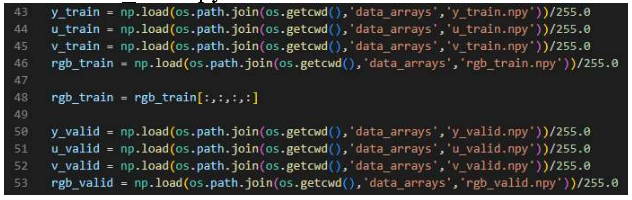


Figure 2 A simple 1D CNN model

1.6.1 Data Loading

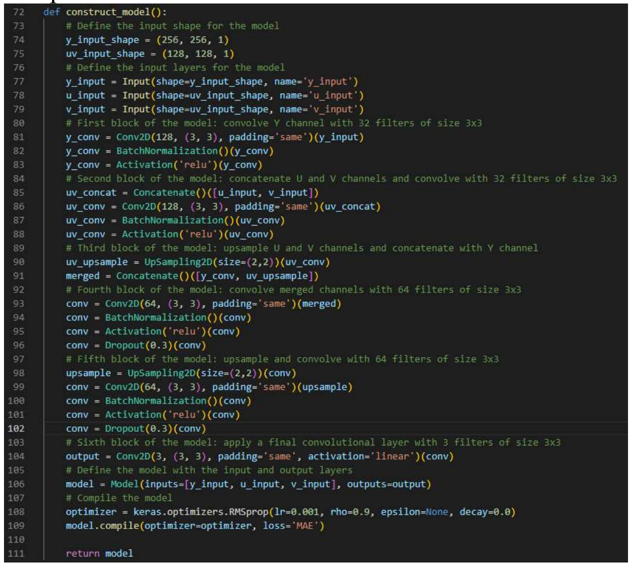
Now that the original dataset has been preprocessed and saved as NumPy arrays, it is now possible to train a CNN. First, the data arrays are loaded and divided by 255.0 to normalize the dataset. Normalization is crucial for reducing complexity which ultimately increases the possibility of the model to converge. The following code is found in train_CNN.py.



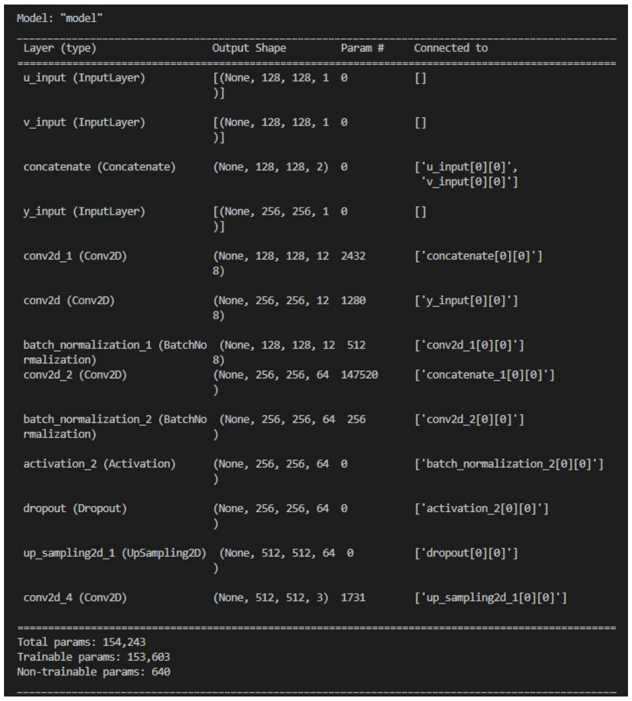
```python
y_train = np.load(os.path.join(os.getcwd(),'data_arrays','y_train.npy'))/255.0
u_train = np.load(os.path.join(os.getcwd(),'data_arrays','u_train.npy'))/255.0
v_train = np.load(os.path.join(os.getcwd(),'data_arrays','v_train.npy'))/255.0
rgb_train = np.load(os.path.join(os.getcwd(),'data_arrays','rgb_train.npy'))/255.0

rgb_train = rgb_train[:,:,:,:]

y_valid = np.load(os.path.join(os.getcwd(),'data_arrays','y_valid.npy'))/255.0
u_valid = np.load(os.path.join(os.getcwd(),'data_arrays','u_valid.npy'))/255.0
v_valid = np.load(os.path.join(os.getcwd(),'data_arrays','v_valid.npy'))/255.0
rgb_valid = np.load(os.path.join(os.getcwd(),'data_arrays','rgb_valid.npy'))/255.0
```

1.6.2 CNN Model Design

The convolutional neural network is constructed and compiled in the code below:



```python
72   def construct_model():
73       # Define the input shape for the model
74       y_input_shape = (256, 256, 1)
75       uv_input_shape = (128, 128, 1)
76       # Define the input layers for the model
77       y_input = Input(shape=y_input_shape, name='y_input')
78       u_input = Input(shape=uv_input_shape, name='u_input')
79       v_input = Input(shape=uv_input_shape, name='v_input')
80       # First block of the model: convolve Y channel with 32 filters of size 3x3
81       y_conv = Conv2D(128, (3, 3), padding='same')(y_input)
82       y_conv = BatchNormalization()(y_conv)
83       y_conv = Activation('relu')(y_conv)
84       # Second block of the model: concatenate U and V channels and convolve with 32 filters of size 3x3
85       uv_concat = Concatenate()([u_input, v_input])
86       uv_conv = Conv2D(128, (3, 3), padding='same')(uv_concat)
87       uv_conv = BatchNormalization()(uv_conv)
88       uv_conv = Activation('relu')(uv_conv)
89       # Third block of the model: upsample U and V channels and concatenate with Y channel
90       uv_upsample = UpSampling2D(size=(2,2))(uv_conv)
91       merged = Concatenate()([y_conv, uv_upsample])
92       # Fourth block of the model: convolve merged channels with 64 filters of size 3x3
93       conv = Conv2D(64, (3, 3), padding='same')(merged)
94       conv = BatchNormalization()(conv)
95       conv = Activation('relu')(conv)
96       conv = Dropout(0.3)(conv)
97       # Fifth block of the model: upsample and convolve with 64 filters of size 3x3
98       upsample = UpSampling2D(size=(2,2))(conv)
99       conv = Conv2D(64, (3, 3), padding='same')(upsample)
100      conv = BatchNormalization()(conv)
101      conv = Activation('relu')(conv)
102      conv = Dropout(0.3)(conv)
103      # Sixth block of the model: apply a final convolutional layer with 3 filters of size 3x3
104      output = Conv2D(3, (3, 3), padding='same', activation='linear')(conv)
105      # Define the model with the input and output layers
106      model = Model(inputs=[y_input, u_input, v_input], outputs=output)
107      # Compile the model
108      optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
109      model.compile(optimizer=optimizer, loss='MAE')
110
111      return model
```



```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 u_input (InputLayer)           [(None, 128, 128, 1  0           []
                                )]

 v_input (InputLayer)           [(None, 128, 128, 1  0           []
                                )]

 concatenate (Concatenate)      (None, 128, 128, 2)  0           ['u_input[0][0]',
                                                                  'v_input[0][0]']

 y_input (InputLayer)           [(None, 256, 256, 1  0           []
                                )]

 conv2d_1 (Conv2D)              (None, 128, 128, 12  2432        ['concatenate[0][0]']
                                8)

 conv2d (Conv2D)                (None, 256, 256, 12  1280        ['y_input[0][0]']
                                8)

 batch_normalization_1 (BatchNo  (None, 128, 128, 12  512        ['conv2d_1[0][0]']
 rmalization)                   8)
 conv2d_2 (Conv2D)              (None, 256, 256, 64  147520      ['concatenate_1[0][0]']
                                )

 batch_normalization_2 (BatchNo  (None, 256, 256, 64  256        ['conv2d_2[0][0]']
 rmalization)                   )

 activation_2 (Activation)      (None, 256, 256, 64  0           ['batch_normalization_2[0][0]']
                                )

 dropout (Dropout)              (None, 256, 256, 64  0           ['activation_2[0][0]']
                                )

 up_sampling2d_1 (UpSampling2D)  (None, 512, 512, 64  0          ['dropout[0][0]']
                                )

 conv2d_4 (Conv2D)              (None, 512, 512, 3)  1731        ['up_sampling2d_1[0][0]']

==================================================================================================
Total params: 154,243
Trainable params: 153,603
Non-trainable params: 640
__________________________________________________________________________________________________
```

Figure 3 Model Summary displaying each layer.

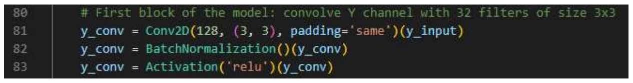
This particular model requires three inputs: a Y channel array of shape (256,256,1), U channel and V channel arrays of shape (128,128,1). These input specifications are created on lines 77-79.

The remaining section of code consists of the model architecture, model optimizer, and model compilation.

1.6.2.1 Model Layer Breakdown

Each layer of the neural network has a convolution operation performed. Convolution layers apply a set of filters to the input data to extract features. This is very similar to the use of kernels in the down-sampling process in section 1.4. The stacking of convolutional layers allows the neural network to learn increasingly complex features by combining the simple features learned in previous layers. Each convolution layer contains weights which are used to recognize patterns in the data. During training, the model will take the input data and produce predictions based on its current set of weights. The error between the prediction and true output is calculated using a loss function. These weights will be adjusted to minimize the loss function. If the model converges, it will be able to map out the relationship between the YUV channel image size and color values to an RGB image size and color values.

1.6.2.1.1 Layer 1: Y channel Convolution



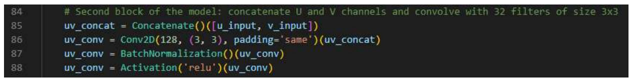
```
    # First block of the model: convolve Y channel with 32 filters of size 3x3
    y_conv = Conv2D(128, (3, 3), padding='same')(y_input)
    y_conv = BatchNormalization()(y_conv)
    y_conv = Activation('relu')(y_conv)
```

This first layer handles the Y channel input. It initially applies a 2D convolution the y_input with 128 filters and a kernel size of (3,3), and will return an output of the same shape of y_input.

It will then apply a batch normalization on the convolution layer's output to reduce covariate shift. It is essentially re-centering and re-scaling the output by maintaining the batch's mean output close to 0 and output standard deviation close to 1. This will allow faster and more stable training.

The rectified linear unit activation function 'ReLU' is applied to the batch normalization layer's output. ReLU converts all negative values to 0. The purpose of this activation function is to allow neural networks to learn and generalize better. This was used because it is computationally efficient and introduced nonlinearity to the model. It is crucial to introduce nonlinearity as the model would simply be a linear regression model if ReLU was not used. Nonlinear models perform better at recognizing any complex nonlinear patterns that may exist in the images.

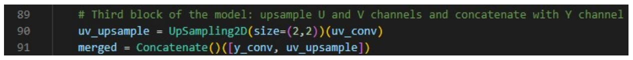
1.6.2.2 Layer 2: U, V channel concatenation and convolution



```
    # Second block of the model: concatenate U and V channels and convolve with 32 filters of size 3x3
    uv_concat = Concatenate()([u_input, v_input])
    uv_conv = Conv2D(128, (3, 3), padding='same')(uv_concat)
    uv_conv = BatchNormalization()(uv_conv)
    uv_conv = Activation('relu')(uv_conv)
```

The U and V input layers are concatenated and the exact same operations performed in Layer 1 are performed on the U, V input layers.

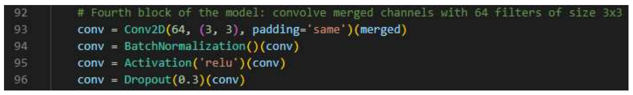
1.6.2.3 Layer 3: UV Up-sampling and Y channel Concatenation



```
    # Third block of the model: upsample U and V channels and concatenate with Y channel
    uv_upsample = UpSampling2D(size=(2,2))(uv_conv)
    merged = Concatenate()([y_conv, uv_upsample])
```

This layer up-samples the U and V channels to match the shape of the Y channel. The UV layer and Y input layer are then concatenated once they are the same size. This concatenation is crucial as convolutional layers operate on the image data as a whole, and not just on individual pixels. This will allow the neural network to learn the color relationships between the YUV and RGB color spacing which is crucial for the inverse color transformation operation.

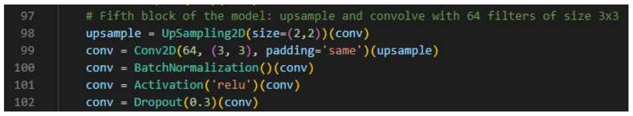
1.6.2.4 Layer 4: Convolution



```
    # Fourth block of the model: convolve merged channels with 64 filters of size 3x3
    conv = Conv2D(64, (3, 3), padding='same')(merged)
    conv = BatchNormalization()(conv)
    conv = Activation('relu')(conv)
    conv = Dropout(0.3)(conv)
```

The concatenated Y, U, V layers are convolved with 64 filters and a kernel size of (3,3). Dropout is a regularization technique that randomly sets values in the output to zero during each training epoch. Dropout was used to prevent the model from overfitting to the training dataset. This ultimately forces the model to learn more robust and generalizable features instead of neurons becoming too specialized to particular features. In this case, a dropout with a rate of 0.3 sets 30% of the values to zero.

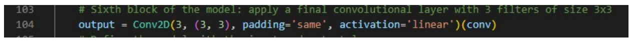
1.6.2.5 Layer 5: Up-sample and Convolution



```
    # Fifth block of the model: upsample and convolve with 64 filters of size 3x3
    upsample = UpSampling2D(size=(2,2))(conv)
    conv = Conv2D(64, (3, 3), padding='same')(upsample)
    conv = BatchNormalization()(conv)
    conv = Activation('relu')(conv)
    conv = Dropout(0.3)(conv)
```

The concatenated Y, U, V layers are now up sampled to match the desired image size of 512x512. Convolution, batch normalization, ReLU activation, and dropout are also performed.

1.6.2.6 Layer 6: Final Convolution



```
    # Sixth block of the model: apply a final convolutional layer with 3 filters of size 3x3
    output = Conv2D(3, (3, 3), padding='same', activation='linear')(conv)
```

The final convolution layer is performed with an activation function named 'linear'. This function does not change the input values and allows the model to output any real number.

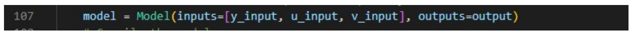
$$f(x) = x \tag{4}$$

There are other activation functions such as sigmoid or previously mentioned ReLU. For example, the sigmoid function produces an output with a probability value between 0 and 1. This is particularly used in binary classification applications.

$$sigmoid(x) = \frac{1}{1 + \exp(-x)} \quad (4)$$

1.6.2.7 Optimizer and Loss Function

The model is now created with the Keras API. An optimizer and the loss function are also specified.



```
model = Model(inputs=[y_input, u_input, v_input], outputs=output)
```

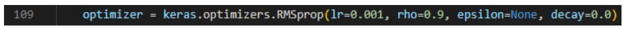
The loss function measures how accurate the model's predictions are given the input data.

An optimizer in machine learning is an algorithm that adjusts the weights and biases to minimize the loss function during training. There are many different types of optimizers, but they all control how fast the model will converge, the model's ability to generalize to new data, and each performs differently to outliers in training data.

In this case, a Ro RMSprop (Root Mean Square) optimizer with a learning rate of 0.001, a decay rate of 0.0, momentum parameter of 0.9, and epsilon constant of None. The RMSprop optimizer is a variant of the stochastic gradient descent algorithm. This optimizer uses a moving average of the squared gradients to normalize the gradient. The goal of this optimizer is to prevent oscillations in the optimization process and to speed up convergence time.

The optimizer updates the weight in this fashion:

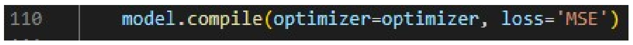
$$weight_{new} = weight_{old} - \frac{learning\ rate}{\sqrt{Root\ Mean\ Square(gradient)}} * gradient \quad (5)$$



```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

The loss function used in this model is the Mean Square Error (MSE) loss function. MSE is a very common loss function used in regression problems. In this case, the MSE loss function would calculate the squared difference between the predicted RGB image and the true RGB input image. The loss function measures the average of the squares of errors meaning that it gives more weight to larger errors.

$$MSE = \frac{1}{N} \sum_{i=1} (I_{i,} - \hat{I}_{i,})^2 \quad (6)$$



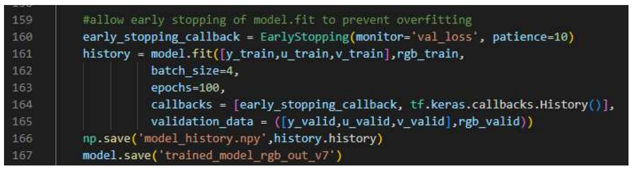
```
model.compile(optimizer=optimizer, loss='MSE')
```

1.6.3 Model Training

Now that the model has now been constructed and compiled, it can now be trained with the training and validation data. The model is fed with the Y, U, V and RGB channel images as training and validation data in model.fit(). The batch size represents the number of training images that are passed through each pass of the model. Batch size is typically chosen as a power of two as most hardware components are optimized for power of two sizes. Epochs represents the number of times the entire dataset will be processed.

There is an early_stopping_callback which allows the model to stop training if the model's validation loss does not improve after 10 consecutive epochs. This was set to reduce overfitting.

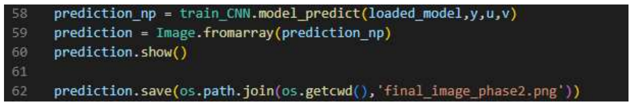
The trained model and it's history are now saved to the main directory.



```
#allow early stopping of model.fit to prevent overfitting
early_stopping_callback = EarlyStopping(monitor='val_loss', patience=10)
history = model.fit([y_train,u_train,v_train],rgb_train,
            batch_size=4,
            epochs=100,
            callbacks = [early_stopping_callback, tf.keras.callbacks.History()],
            validation_data = ([y_valid,u_valid,v_valid],rgb_valid))
np.save('model_history.npy',history.history)
model.save('trained_model_rgb_out_v7')
```

1.7. Load Saved Model and Predict

Now that the model has been trained, it can now be loaded and fed new input data to form a prediction. The down-sampled YUV channels from the test image and the saved model are now fed into model_predict(). The output prediction is a NumPy array and is displayed using PIL.Image.fromarray and .show(). The predicted image is also saved to the main directory.



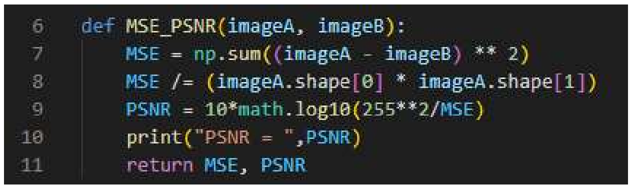
```
prediction_np = train_CNN.model_predict(loaded_model,y,u,v)
prediction = Image.fromarray(prediction_np)
prediction.show()

prediction.save(os.path.join(os.getcwd(),'final_image_phase2.png'))
```

1.8. Calculate Peak Signal-to-Noise Ratio

Peak signal-to-noise ratio represents the measurement of quality between the original and compressed image. The PSNR is greater as quality of the reconstructed image increases. Mean squared prediction error represents the cumulative error between the compressed and original error. PSNR represents the peak error. MSE and PSNR can be calculated using the following formulae:

$$MSE = \frac{1}{Width * Height * 3} \sum_{i,j,c} (I_{i,j,c} - \hat{I}_{i,j,c})^2 \quad (7)$$

$$PSNR = 10 * \log_{10} \frac{255^2}{MSE} \quad (8)$$



```python
def MSE_PSNR(imageA, imageB):
    MSE = np.sum((imageA - imageB) ** 2)
    MSE /= (imageA.shape[0] * imageA.shape[1])
    PSNR = 10*math.log10(255**2/MSE)
    print("PSNR = ",PSNR)
    return MSE, PSNR
```

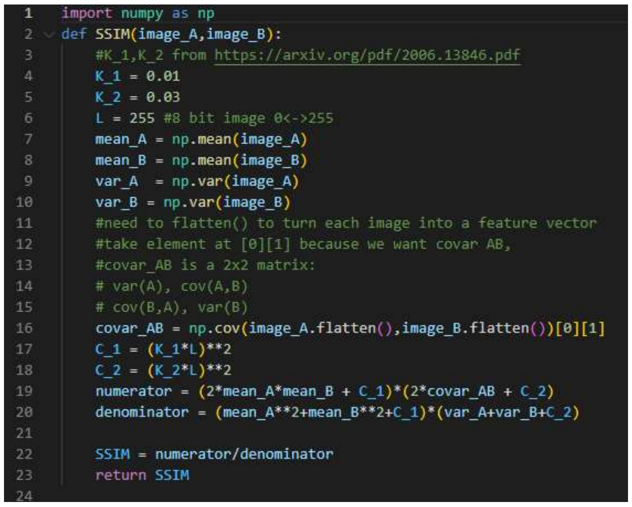
1.9. Calculate Structural Similarity Index

The Structural Similarity Index (SSIM) is a metric used to compare two images that mimic the human visual system. It is different from PSNR as it places greater emphasis on structural properties in images such as edges and textures. It also places less emphasis on changes in color and brightness.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{9}$$

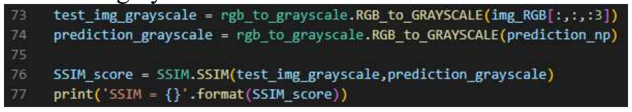
$where:$
$\mu_x\ is\ the\ mean\ of\ x$
$\mu_y\ is\ the\ mean\ of\ y$
$\sigma_x^2\ is\ the\ variance\ of\ x$
$\sigma_y^2\ is\ the\ variance\ of\ y$
$\sigma_{xy}\ is\ the\ covariance\ of\ x\ and\ y$
$c_1 = (k_1L)^2$
$c_2 = (k_2L)^2$

The parameters k1 and k2 are constants that adjust the relative impact of the luminance and contrast terms in equation 9. K1 and K2 are set as 0.01 and 0.03 by default. (Nilsson & Akenine-Möller, 2020)



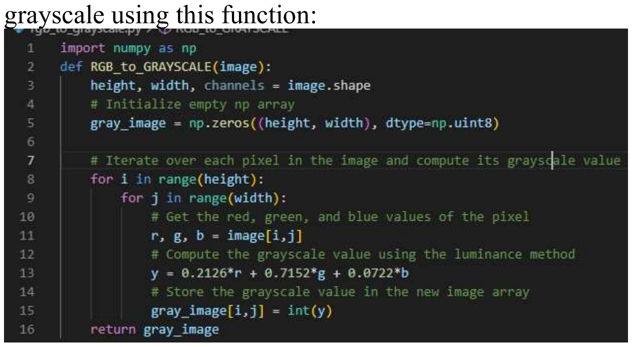
```python
import numpy as np
def SSIM(image_A,image_B):
    #K_1,K_2 from https://arxiv.org/pdf/2006.13846.pdf
    K_1 = 0.01
    K_2 = 0.03
    L = 255 #8 bit image 0<->255
    mean_A = np.mean(image_A)
    mean_B = np.mean(image_B)
    var_A  = np.var(image_A)
    var_B = np.var(image_B)
    #need to flatten() to turn each image into a feature vector
    #take element at [0][1] because we want covar AB,
    #covar_AB is a 2x2 matrix:
    # var(A), cov(A,B)
    # cov(B,A), var(B)
    covar_AB = np.cov(image_A.flatten(),image_B.flatten())[0][1]
    C_1 = (K_1*L)**2
    C_2 = (K_2*L)**2
    numerator = (2*mean_A*mean_B + C_1)*(2*covar_AB + C_2)
    denominator = (mean_A**2+mean_B**2+C_1)*(var_A+var_B+C_2)

    SSIM = numerator/denominator
    return SSIM

```

Please note that the SSIM input images must be converted to grayscale.



```python
test_img_grayscale = rgb_to_grayscale.RGB_to_GRAYSCALE(img_RGB[:,:,:3])
prediction_grayscale = rgb_to_grayscale.RGB_to_GRAYSCALE(prediction_np)

SSIM_score = SSIM.SSIM(test_img_grayscale,prediction_grayscale)
print('SSIM = {}'.format(SSIM_score))
```

The predicted and test images are converted to grayscale using this function:



```python
import numpy as np
def RGB_to_GRAYSCALE(image):
    height, width, channels = image.shape
    # Initialize empty np array
    gray_image = np.zeros((height, width), dtype=np.uint8)

    # Iterate over each pixel in the image and compute its grayscale value
    for i in range(height):
        for j in range(width):
            # Get the red, green, and blue values of the pixel
            r, g, b = image[i,j]
            # Compute the grayscale value using the luminance method
            y = 0.2126*r + 0.7152*g + 0.0722*b
            # Store the grayscale value in the new image array
            gray_image[i,j] = int(y)
    return gray_image
```

2. Experimental Results

2.1. Training and Validation Loss

The initial CNN model constructed contained fewer convolution layers with less filters, did not use the Dropout technique, and the Adam optimizer. It's performance was comparably worse during training and when testing on sample images.

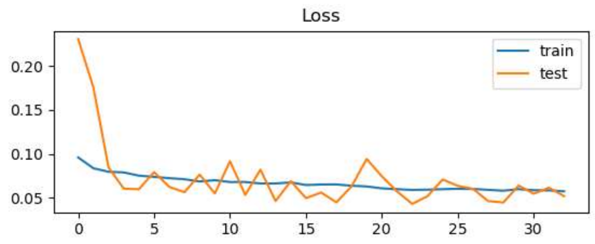


Figure 4 Initial CNN Model Training Performance

The training loss, PSNR, and SSIM metrics all improved after including more convolution layers, using the Dropout technique, and the RMSprop optimizer in the CNN model. It is clear that the RMSprop optimizer greatly reduced the oscillation in the validation loss.

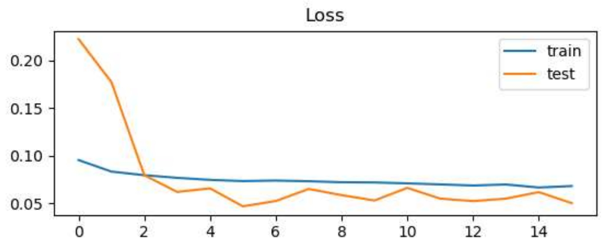


Figure 5 Final CNN Model Training Performance

2.2. Image Quality Comparison

This image compression pipeline was tested on 5 images. The images are fairly accurate visually; however, the loss of digital information is clear when the image is zoomed in. This loss of information can be improved by using another down sampling algorithm. The loss of information is reflected in the PSNR values. However, the SSIM values are all very close to the maximum value of 1 implying that the semantic information is retained.

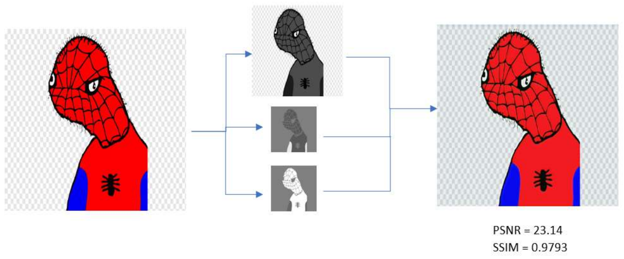


Figure 6 Test Image 1

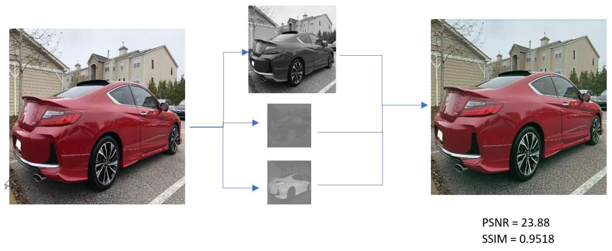


Figure 7 Test Image 2

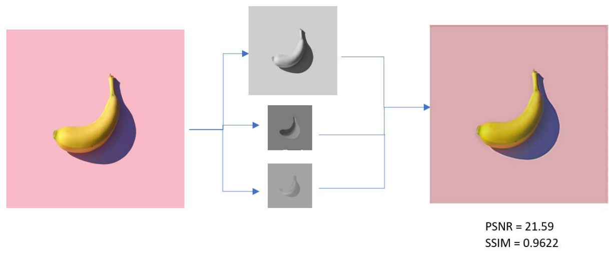


Figure 8 Test Image 3

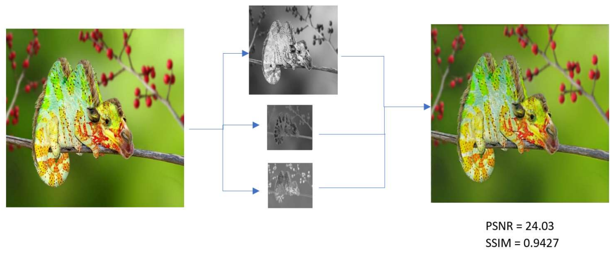


Figure 9 Test Image 4

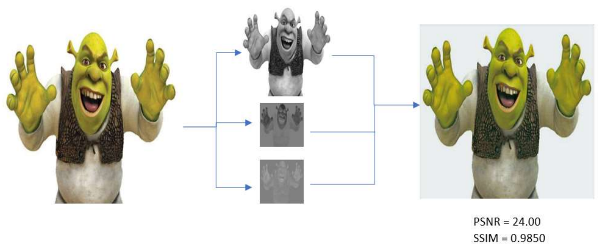


Figure 10 Test Image 5

2.3. File Size Comparison

The compressed file size of the test image dress_512x512 is 518 KB, and the original file size is 1028 KB. The resulting compression rate is approximately 50.38%.

$$Compression\ Ratio = \frac{(size\ before\ compression)}{size\ after\ compression} \quad (9)$$
$$= \frac{518}{1028}$$
$$\cong 50.38\%$$

3. Conclusion

The goal of this image compression pipeline has demonstrated that images can be significantly compressed while preserving most of the semantic information. This pipeline demonstrated that the use of different color spaces such as the YUV color space can greatly improve image compression rates. The implementation of the convolutional neural network also made a significant increase in image quality especially around the edges of the image compared to the bilinear interpolation method used in phase 1. This use of the convolutional neural network demonstrated how machine learning will be used to make significant advances in image processing.

References

[1] ITU-R. (2011, March 8). Recommendation ITU-R BT.2020-2. International Telecommunication Union. ,from https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf

[2] Li, Z.-N., Drew, M. S., & Liu, J. (2021). *Fundamentals of Multimedia* (3rd ed.). Springer.

[3] Nilsson, J., & Akenine-Möller, T. (2020, June 29). *Understanding SSIM*. arXiv.org. Retrieved April 15, 2023, from https://arxiv.org/abs/2006.13846