Image Compression Pipeline

Sheshank Suresh
McMaster University
1280 Main St. West, Hamilton, ON, Canada
suress9@mcmaster.ca

Abstract

This report will further expand on the image compression pipeline with developing the methods from Phase 1 further. The first phase was a basic down sampling and up sampling model whereas, the second phase integrate machine learning to provide a better compression pipeline. The previous phased outlined the steps taken to achieve lossy image compression through additional methods such as bilinear up sampling. Phase 2 implements a convolutional neural network (CNN) to complete a more accurate process of up sampling while evaluating the peak signal-to-noise ratio, structural similarity index and normalized absolute error. The implementation was completed using Python with the use of the TensorFlow library. Near the end, it will conclude key takeaways and future improvements that can further the study of lossy image compression.

1. Introduction

The world has been overruled with technology in every corner which requires a lot of technical details to be considered. Image compression is especially useful in various industries such as cloud transfer applications, conserve bandwidth usage for larger groups and more. As a result, the pipeline has been consistently evolving to manage resources effectively and efficiently. However, with various strategies, images can experience content loss specifically when up sampling is implemented.

1.1. Problem Definition

The problem addressed in this project is the limitations behind basic strategies of down and up sampling methods such as bilinear or bicubic up sampling. Through these methods, there tends to be a lot of losses in image quality especially around the edges and details. Thus, a more advanced method such as a convolutional neural network is used to up sample the YUV channels of the image. The model should assist in improving the accuracy of the final image by preserving more of the edges and details. A set of 5 images were used for the experiment.

1.2. Existing Methods

Existing methods such as bilinear and bicubic up sampling are useful in developing a basic understanding behind image compression pipelines but lack accuracy as mentioned before.

Bilinear up sampling is an interpolation method used to estimate a missing pixel while decompressing an image by averaging the four closest pixels. Bicubic up sampling is slightly more accurate as it uses a weighted average with a larger set of pixels resulting in smoother interpolation.

However, these conventional methods are not effective due to lossy compression, edge and detail losses.

2. Implementation

2.1. Model Design - CNN

The first step of the design is to define the input layer of the CNN model with a determined shape. The ReLu activation function is used with the same padding to ensure the output feature map has the same dimensions as the input feature map for 32, 64 and 128 filters. An up-sampling layer is added with a factor of 2 to recover fine details of the image by scaling the dimensions by 2. Next, a few more convolutional layers are added with the ReLu activation function with 3x3 64 and 32 filters. The last filter does not have an activation function to output the up sampled image. The hyperbolic tangent activation function is used to map the output in range of [-1,1]. The next line is used to scale the output layers to the pixel range, [0, 255].

```python
def get_model():

    input = Input(shape=(32,32,1))
    x = Conv2D(32, 3, activation='relu', padding='same')(input)
    x = Conv2D(64, 3, activation='relu', padding='same')(x)
    x = Conv2D(128, 3, activation='relu', padding='same')(x)
    x = UpSampling2D(2)(x)
    x = Conv2D(64, 3, activation='relu', padding='same')(x)
    x = Conv2D(32, 3, activation='relu', padding='same')(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
```

Figure 1. Model Design in Python

2.2. Data Collection for Training

This function collects the image dataset that will be used for training. The images are converted grayscale using the Y channel and resize them 128x128 pixels. The images are down sampled to 64x64 size images, and the values are appended to NumPy arrays.

```python
def downsample(img, factor):
    rows, cols = img.shape[0], img.shape[1]
    downsampled_img = np.zeros((rows, cols, 3), 'uint8')

    for i in range(rows):
        for j in range(cols):
            if i%factor==0 and j%factor == 0:
                downsampled_img[(i//factor),(j//factor),0] = img[i, j, 0] #b
                downsampled_img[(i//factor),(j//factor),1] = img[i, j, 1] #g
                downsampled_img[(i//factor),(j//factor),2] = img[i, j, 2] #r
    return downsampled_img
```

Figure 2. Data Collection in Python

2.3. Model Training

Using the pre-defined CNN model, the dataset is trained with the Adam optimizer with a learning rate of 0.0001 and a mean squared error loss function. The best model is saved using TensorBoard callbacks and maintain training logs relevant to the loss values. Finally, the model is trained using the training and validation data with a batch size of 4 and 100 epochs.

```python
model = get_model()
curr_path = os.getcwd()
curr_path = curr_path + '\DIV2K_train_HR\*.png'
x, y = get_data(curr_path)
print(x.shape,y.shape)

X_train, X_val, y_train, y_val = train_test_split(x, y, test_size=0.2, random_state=42)

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

save_model_callback = tf.keras.callbacks.ModelCheckpoint(
    'model/model_u_channel.h5',
    monitor = 'val_loss',
    verbose = 1,
    save_best_only = True,
    mode = 'min',
    save_freq = 'epoch'
)

tbCallBack = tf.keras.callbacks.TensorBoard(log_dir='./Graph', histogram_freq=0, write_graph=True, write_images=True)
batch_size = 4
epochs = 100
model.fit(X_train, y_train, validation_data=(X_val, y_val), batch_size=batch_size, epochs=epochs, validation_split=0.1, callbacks=[tbCallBack, save_mode
```

Figure 3. Up sampling Python Code

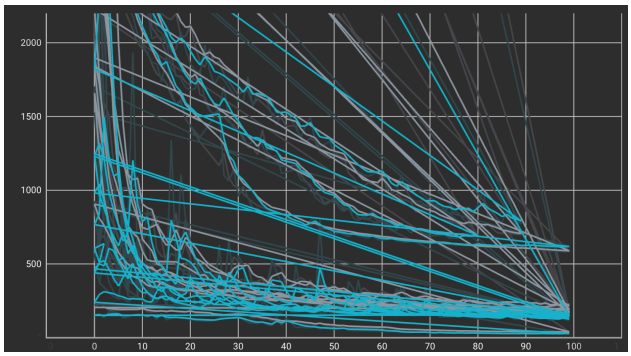

Figure 4. Loss Value Graph for 100 Epochs

2.4. Modifications to Phase 1

The Phase 1 code for up sampling and down sampling was modified to use the CNN model instead of mathematical expressions. The down sampling followed the same logic as the previous phase. The image is compressed to a 64x64 image to handle the model. The input image is first converted from the RGB to YCrBr color space. The model is used to predict the Y up sampled channel values using the imported model. A 3D array is created using the U and V channels containing the up sampled values which is then concatenated with the Y up sampled values. A conversion matrix is used to convert the image back to the RGB color space with the standard pixel range of [0, 255] and converted to 8-bit integer data type. The image is then down sampled to ideally return to the original image.

```python
def upsample(rgb_downsampled_image):

    model = load_model("model.h5", compile=False)

    YCrCb_img = cv2.cvtColor(rgb_downsampled_image, cv2.COLOR_RGB2YCrCb)

    y_channel = YCrCb_img[:,:,0]
    u_channel = YCrCb_img[:,:,1]
    v_channel = YCrCb_img[:,:,2]

    y_interpolate = cv2.resize(y_channel, (256,256), interpolation=cv2.INTER_AREA)
    y_input = np.expand_dims(y_interpolate, axis=0)

    y_upsampled = model.predict(y_input)
    u_upsampled = cv2.resize(u_channel, (512, 512), interpolation=cv2.INTER_AREA)
    v_upsampled = cv2.resize(v_channel, (512, 512), interpolation=cv2.INTER_AREA)
    uv_upsampled = np.stack((v_upsampled, u_upsampled), axis=-1)
    yuv_upsampled = np.concatenate((y_upsampled[0], uv_upsampled), axis=2)

    conversion_matrix = np.array([[1.164, 0.000, 1.596],
                                  [1.164, -0.392, -0.813],
                                  [1.164, 2.017, 0.000]])
    yuv_upsampled = yuv_upsampled.astype(np.float32)
    yuv_upsampled[:, :, 1:] -= 128
    rgb_upsampled = np.dot(yuv_upsampled, conversion_matrix.T)
    rgb_upsampled = np.clip(rgb_upsampled, 0, 255)
    rgb_upsampled = rgb_upsampled.astype(np.uint8)

    cv2.imwrite('Images/upsampled_dress.png', rgb_upsampled)
    cv2.imshow("Upsampled Image", rgb_upsampled)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return rgb_upsampled
```

Figure 5. Up Sample Function

2.5. PSNR Calculation

The peak signal-to-noise ratio is calculated using the compressed and decompressed image measured in decibels to evaluate the quality of the output. PSNR values around 30dB are generally determined to be high quality images while values that are lower than 20dB are low quality images. PSNR is calculated using the following expression:

$$PSNR = 10log(\frac{255}{MSE}) \quad (7)$$

where Mean Squared Error (MSE) is expressed by:

$$MSE = \frac{1}{W \times H \times 3} \times \sum_{i,j,c} (\hat{I}_{I,J,C} - I_{I,J,C})^2 \quad (8)$$

```python
def calculate_psnr(compressed_img, decompressed_img):
    mse = np.mean((compressed_img - decompressed_img) ** 2)
    if mse == 0:
        return ("PSNR Error - MSE = 0")
    else:
        psnr = 10 * np.log10(255 ** 2 / mse)
        return psnr
```

Figure 6. PSNR Calculation Function

2.6. SSIM Calculation

The Structural Similarity Index (SSIM) is used to measure the similarity between two images by comparing the luminance, contrast, and structural information. Generally, images with SSIM values closer to 1 are high quality images after decompression.

```python
def calculate_ssim(input_img, compressed_img):

    img_avg = np.mean(input_img)
    compressed_img_avg = np.mean(compressed_img)

    img_var = np.var(input_img)
    compressed_img_var = np.var(compressed_img)

    covar = np.cov(input_img.flatten(), compressed_img.flatten())[0][1]

    c1 = (0.01 * 255) ** 2
    c2 = (0.03 * 255) ** 2
    numerator = (2 * img_avg * compressed_img_avg + c1) * (2 * covar + c2)
    denominator = (img_avg ** 2 + compressed_img_avg ** 2 + c1) * (img_var + compressed_img_var + c2)
    ssim = numerator / denominator
    return ssim
```

Figure 7. SSIM Calculation Function

3. Experimentation Results

A set of 5 images were used to test the results of the training model and experiment.

3.1. Dress Image

Figure 8. Original Image

Figure 9. Compressed Image

Figure 10. Up Sampled Image

```
SSIM: 0.9189383402706416
PSNR: 27.924654018573385
NAE: 0.7687589109333512
```

Figure 11. PSNR, SSIM and NAE Values

3.2. Lighthouse

Figure 12. Original Image

Figure 13. Compressed Image

Figure 14. Up Sampled Image

```
SSIM: 0.9548312126606041
PSNR: 27.975901848163627
NAE: 0.8535880892884498
```

Figure 15. PSNR, SSIM and NAE Values

3.3. Nature

Figure 16. Original Image

Figure 17. Compressed Image

Figure 18. Up Sampled Image

```
SSIM: 0.9672848905849812
PSNR: 28.372133296233937
NAE: 0.83483571569904
```

Figure 19. PSNR, SSIM and NAE Values

3.4. Peppers

Figure 20. Original Image

Figure 21. Compressed Image

Figure 22. Up Sampled Image

```
SSIM: 0.9672848905849812
PSNR: 28.372133296233937
NAE: 0.83483571569904
```

Figure 23. PSNR, SSIM and NAE Values

3.5. Among Us

Figure 24. Original Image

Figure 25. Compressed Image

Figure 26. Up Sampled Image

```
SSIM: 0.9893005732559261
PSNR: 31.243804225254628
NAE: 0.7085631975161484
```

Figure 27. PSNR, SSIM and NAE Values

4. Conclusion

Overall, the metrics are generally in the range of the accepted values which further proves the machine learning model is more effective than conventional methods such as bilinear and bicubic sampling.