# DSC 210: Image Compression Techniques


| Saeji Hong | Zijie Feng | Henry Xu | Tushar Mohan |
|---|---|---|---|
| sahong@ucsd.edu | zifeng@ucsd.edu | jix049@ucsd.edu | tmohan@ucsd.edu |
| A16960847 | A16207094 | A17012394 | A59026337 |


## 1 Introduction

Image compression techniques and tools have been used by almost everybody all the time. They have become such an important part in people's lives of the modern era. In this project, we explore the relationship between numerical linear algebra and image compression using SVD as well as utilizing a classical approach, Huffman coding, and a state-of-the-art approach, MLIC, as reference. More specifically, we perform an experiment using the 3 algorithms on the same dataset to compare and contrast their performances.

### 1.1 History/Background

In the 20th century, computational communication technologies have taken a leap in development. The image compression problem was therefore derived as a response to the increasing need to store and transmit digital images more efficiently.

Many techniques have been developed and advanced in the past several decades. Back in 1948, Claude Shannon published the paper "A Mathematical Theory of Communication"[4], introducing concepts like redundancy and entropy, building the foundation of modern data compression. A few years later, in 1951, David Huffman and a few of his classmates wrote a term paper for their information theory class at MIT[7]. The paper focused on the problem of finding the most efficient binary code. As Huffman was about to give up since the problem was quite hard, he came up with the idea of using a frequency-sorted binary tree, and this method was proved to be more efficient than the methods previously created by Claude Shannon.

Continuing on to the 1970s, Nasir Ahmed introduced the discrete cosine transform (DCT)[1], which became the basis of most modern image and video compression standards. In addition, Run-Length Encoding (RLE)[11] was introduced for fax transmission, and this technique is particularly good at compressing binary images with large uniform areas. Later in the 1980s, Vector Quantization was introduced and it was a lossy compression method[12], but it did save storage space. Not too long later, Lempel-Ziv-Welch (LZW)[5] was also introduced as an adaptive algorithm that offers efficient and lossless compression, and it became very popular in formats like Graphics Interchange Format (GIF).

Starting in the 1990s up to today, a lot of the really important formats we use today appeared, including JPEG[8] (Joint Photographic Experts Group), PNG[10] (Portable Network Graphics), JPEG File Interchange Format[9] (JFIF), WebP[13], and High Efficiency Image Format (HEIC)[6]. Each of these different formats has different advantages and is developed for different purposes for the choice of users. Only with these various image compression methods and formats can we store images locally with efficiency or share images with others over the Internet so easily.

### 1.2 Applications

In the real world, image compression is really important in many contexts. For example, for applications on a mobile device, it is crucial that the app can be loaded quickly and efficiently in terms of power consumption. This means images in the application that need to be loaded cannot be

too large; they need to be compressed to a smaller volume while still maintaining good quality and this is where a good image compression technique shines. We are in an era that contains an explosive amount of data, especially visual data, so it is important to store visual data efficiently in a lot of fields, including streaming and broadcasting, cloud storage, digital photograph, medical imaging, retail and e-commerce, security and law enforcement, and many more.

### 1.3 State-of-the-art

One state-of-the-art approach is called Learned Image Compression (LIC). Deep learning methods now dominate SOTA in image compression, using end-to-end trainable neural networks. Recent advances in LIC have demonstrated significant improvements over classical approaches, with models optimizing the fundamental trade-off between compression rate and reconstruction quality. These neural architectures typically employ variational autoencoders with sophisticated entropy models, enabling both higher compression ratios and better perceptual quality. In our work, we utilize the CompressAI framework, which implements several cutting-edge LIC models. We conduct comprehensive experiments using this framework to evaluate compression performance and present our findings in the Experiments section.

## 2 Problem Formulation

### 2.1 Relation to numerical linear algebra

The relation between image compression and numerical linear algebra is quite clear in the fact that images can be easily thought of as matrices. Images are normally rectangular, which is the same for matrices. Images contain one single pixel at each distinct spot, which is similar to how matrices have a single entry at each position. This way, we can represent all images as matrices and perform processes like LU decomposition, QR decomposition, SVD or any other methods we can deploy on the matrix representation of the image. Among these methods, SVD has been the most popular to reduce the size of the original image matrix, to accomplish the goal of image compression.

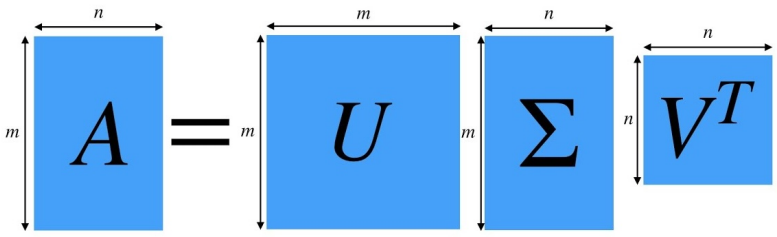
### 2.2 Approach description

In this project, we focus on 2 classical approaches to compress images: SVD and Huffman Coding.

**SVD:**
As previously mentioned, the problem of image compression can be solved by numerical linear algebra, particularly through singular value decomposition (SVD), which is a method that can estimate the image matrix by reducing the rank, decreasing the amount of data needed to be used as representation of the image while also maintaining most visual features.

We can think of an image as a matrix $A \in \mathbb{R}^{m \times n}$, where m is the height of the image and n is the width of the image. Each matrix entry $A_{ij}$ is a pixel in the original image. Using SVD, the original image matrix $A$ can be decomposed as shown in this picture:

$$A = U\Sigma V^T$$




In this decomposition, $\Sigma$ is really important as it contains $\sigma_i$ sorted in decreasing order. Each singular value ($\sigma_i$) represents the importance of the corresponding singular vectors. If we select the first $k$

singular values, this means we have kept the top $k$ most important columns of information, so that our compressed image still looks like the original.

Now, with the selected $k$ such that $k < n$, we can produce a new matrix $A_k$ or new image from:

$$A_k = U_k \Sigma_k V_k^T$$

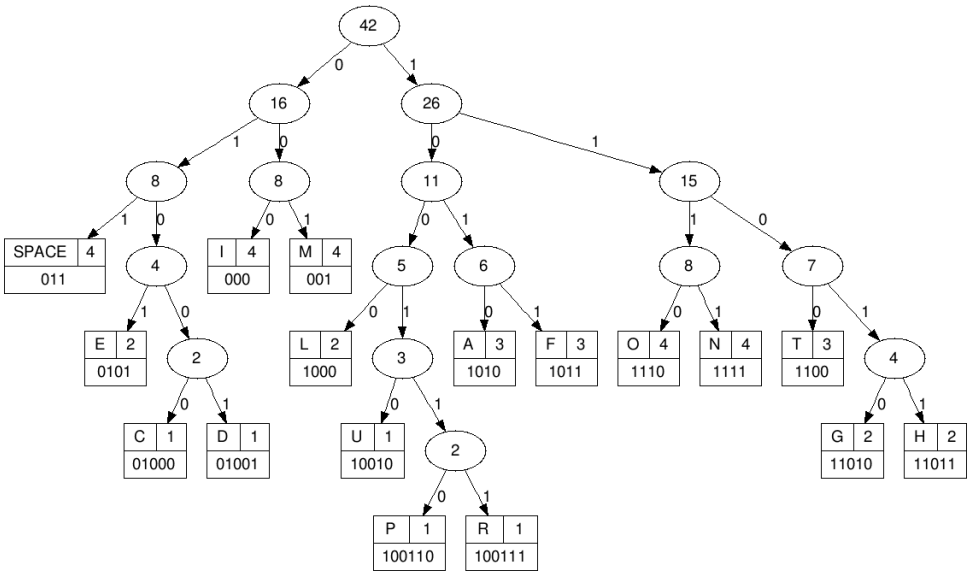
The value $k$ here represents the number of the largest singular values we want to keep from the original image matrix $A$. By selecting a $k$ that's smaller than the original $n$, we create a rank-k approximation $A_k$ that takes up less space.

Therefore, the following steps can be taken to compress and store an image:

1. Compute the SVD of the original image matrix $A$
2. Select $k$ largest singular values for the construction of $A_k$
3. Store $U_k$, $\Sigma_k$, and $V_k$ for a compressed representation
4. To reconstruct the image, multiply $U_k \Sigma_k V_k^T$

**Huffman Coding:**
Huffman coding isn't necessarily related to linear algebra but it is a widely used algorithm for data compression of all kinds, not only limited to image compression. The core idea of Huffman coding is that we can assign shorter binary codes to more frequent symbols and longer codes for less frequent ones. This way, we can save data space since the majority of data will get shorter binary codes.

To use Huffman coding for image compression, the image needs to be converted into a suitable representation, such as intensity values for grayscale images. We can take grayscale images as an example. For this kind of images, the intensity levels at each pixel will range from 0 to 255. In Huffman coding, the intensity levels can be considered as the symbols so we can assign frequencies to these intensity levels and sort them from most frequent to the least. The most crucial part of Huffman coding is the building of the Huffman tree, which is the process that combines the 2 least frequent symbols into a node and repeat until all symbols have been used and combined into a single binary tree. This will result in a tree that has the least frequent symbols furthest away from the root. In addition, the little "legs" on each node will be marked as 0 on the left and 1 on the right by convention, and this is for the binary encoding of each symbol. Let's look at the tree in this image as an example:



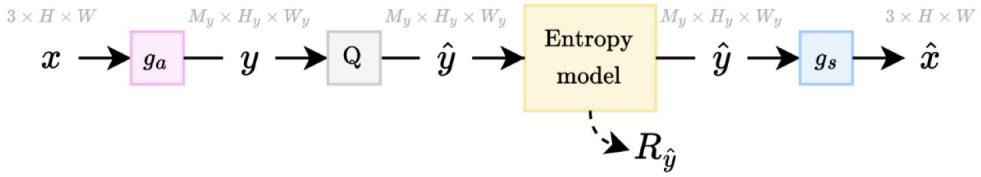

This example tree is built on text data instead of image data but the idea is the same. From this example tree, we can tell that the word "space" is more frequent than the letter "P" since it's closer to

the root. Therefore, it makes sense for "space" (011) to have a shorter binary code than the one of "P" (100110). This way, we have assigned the more frequent symbols shorter binary codes, to save space.

Decoding the binary codes encoded by Huffman coding is going to give us a lossless image so it is very powerful and still popular today.

### 2.3 SOTA approach description

People no longer use Numerical Linear Algebra (NLA) approaches for image compression. Instead, deep learning methods now dominate the state-of-the-art (SOTA) in this field, utilizing end-to-end trainable neural networks. Recent advances in Learned Image Compression (LIC) have demonstrated significant improvements over classical methods, with models that optimize the fundamental trade-off between compression rate and reconstruction quality. These neural architectures often employ variational autoencoders along with sophisticated entropy models, allowing for both higher compression ratios and improved perceptual quality.




The Learned Image Compression structure consists of four main parts

1. Analysis transform $g_a$: transform raw input $x$ into more compact representation $y$. This could be multiple downsampling convolutional layers.
2. Quantization $Q$: transform real number $y_i$ to lower precision for better storage. One example of such $Q$ is uniform quantizer, which is basically rounding.
3. Entropy coding: resulted $\hat{y}$ is losslessly compressed with an entropy coding method. One example is arithmetic coding, which compress data by building an encoding distribution where frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits.
4. Synthesis transform $g_s$: In the end, quantized $\hat{y}$ is feed through a synthesis transform to reconstruct an image $\hat{x}$ In learned Image compression this is typically multiple upsampling convolutional layers with lots of trainable parameters.

The training objective is minimizing the size of the compressed image $R_y$ and the distortion between $x$ and $\hat{x}$, typically measured by MSE[3][2].

### 2.4 Evaluation

We will be looking at the compressed images by each different algorithm, considering image quality vs. compression efficiency. For image quality, we use PSNR (Peak Signal-to-Noise Ratio) as measurement. For compression efficiency, we simply use compression ratio which is $\frac{\text{Uncompressed Size}}{\text{Compressed Size}}$. With these metrics, we can tell the advantages and disadvantages of each algorithm.

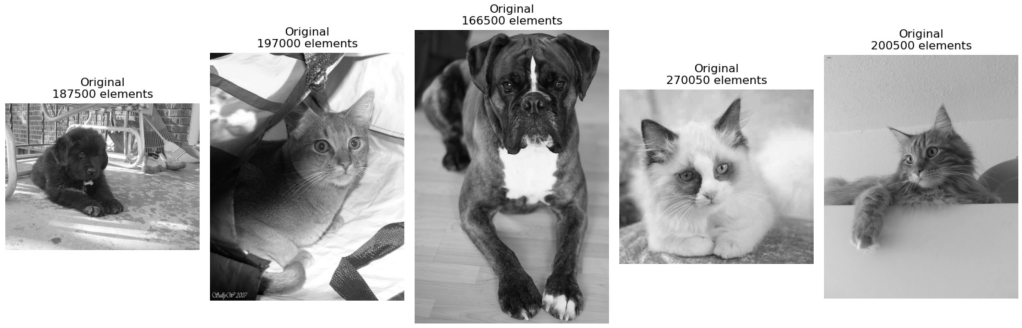
## 3 Experiments

### 3.1 Setup and logistics

Python is the programming language we chose for this project. We utilized many outside libraries to support our project. NumPy is used to provide support for numerical computation and manipulation of arrays/matrices. Matplotlib is used for all the plotting of results. Skimage is used for reading and writing images. Heapq is used for building a heap queue. Pickle is used for serializing and deserializing Python objects. Torch is used to provide deep learning framework. PIL is also used for manipulating image files. Compressai is used for the implementation of LIC.

### 3.2 Dataset and programming

Since the topic is image compression, the dataset for this project specifically will be a set of pictures from The Oxford-IIIT Pet Dataset. We selected 5 pictures from the "images.tar.gz" file to showcase algorithm performances here:




### 3.3 Implementation

**SVD:**

```
U, S, VT = np.linalg.svd(image, full_matrices=False)
```

This part is where we do the SVD decomposition and divide the original image matrix into the two orthogonal matrices and a diagonal matrix with the singular values

```
U_k = U[:, :k]
S_k = S[:k, :k]
VT_k = VT[:k, :]
```

Here, we extract the k top singular values to retain in our compressed image. This would mean we retain the first k columns of U, the first k by k diagonal of S, and the first k rows of VT. Multiplying these out:

```
compressed_image = np.dot(U_k, np.dot(S_k, VT_k))
```

We would have the compressed image with the result of the multiplication of the truncated matrix decomposed matrices. This resultant matrix would be the compressed image.

**Huffman Coding:**

```
# Function to build the Huffman tree
def build_huffman_tree(frequency):
    heap = [Node(char, freq) for char, freq in frequency.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]
```

This is the part where the huffman tree/heap is generated and the reduction is applied till there is only 1 node left in the tree.

```
# Function to encode the image using Huffman coding
def huffman_encoding(image):
    flat_image = image.flatten()
    frequency = Counter(flat_image)
    root = build_huffman_tree(frequency)
    codebook = build_codes(root)
    encoded_image = ''.join(codebook[pixel] for pixel in flat_image)
    return encoded_image, root, image.shape
```

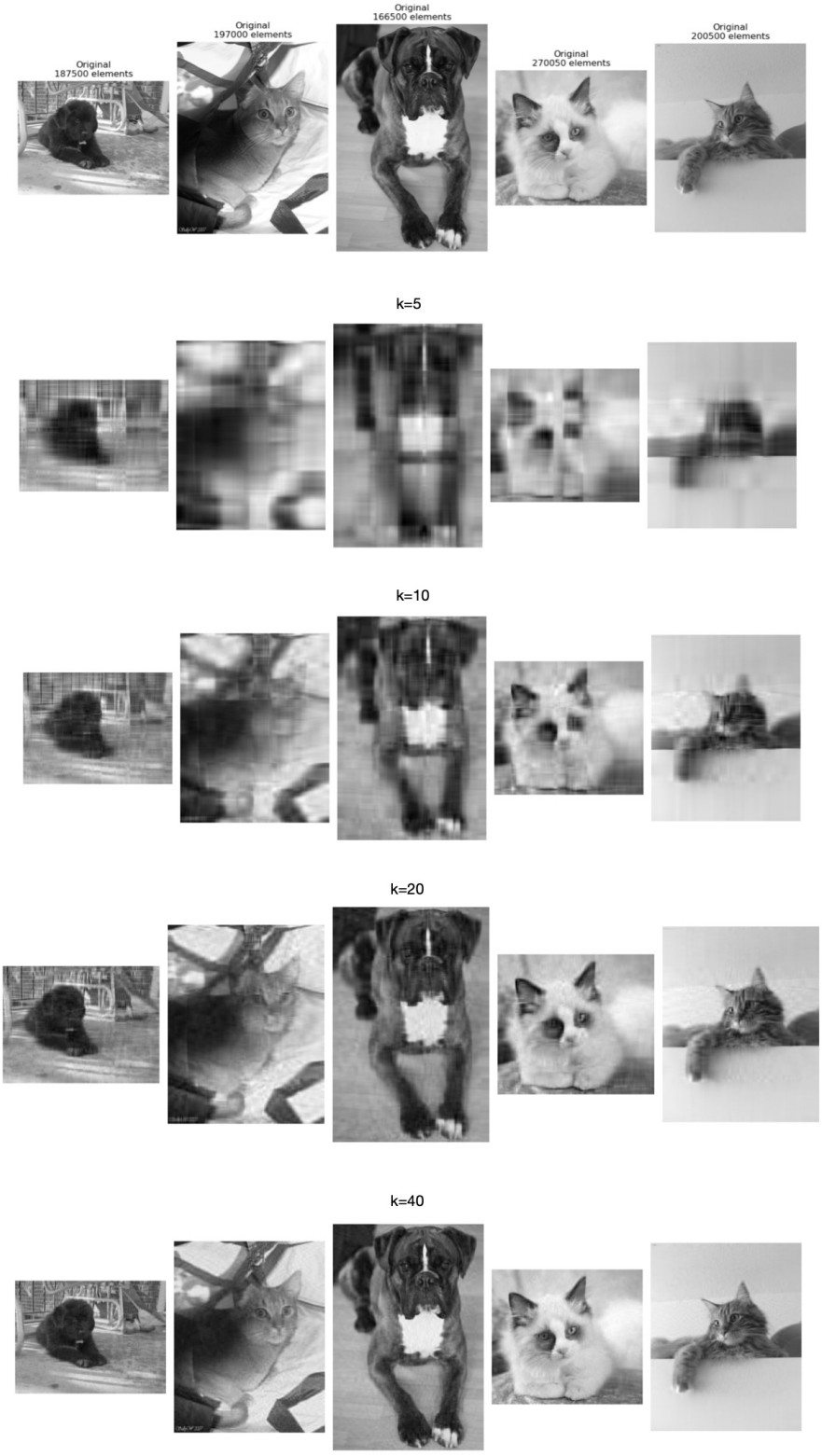
The above picture shows the high-level overview of the huffman coding algorithm. First, we flatten the image and then find the frequency of each distinct pixel value. This frequency map is used to build the huffman heap in bottom-up fashion. With the root of the huffman tree, codes for each distinct pixel is generated using the recursive function which helps encode the image in entirety.

```
# Function to build the Huffman codes
def build_codes(node, prefix="", codebook={}):
    if node is not None:
        if node.char is not None:
            codebook[node.char] = prefix
        build_codes(node.left, prefix + "0", codebook)
        build_codes(node.right, prefix + "1", codebook)
    return codebook
```

The above picture outlines how huffman codes are built recursively.

### 3.4 Results

**SVD Results:**



**Huffman Coding Results:**


### 3.5 SOTA Implementation

```python
def compress_image_with_pretrained(img, quality=1):
    """
    Compress an image using a pre-trained CompressAI model
    Handles both RGB and grayscale numpy arrays
    """
    # Convert numpy array to PIL Image if necessary
    if isinstance(img, np.ndarray):
        # Normalize to 0-255 range if in float
        if img.dtype == np.float64 or img.dtype == np.float32:
            img = (img * 255).astype(np.uint8)

        if len(img.shape) == 2:  # Grayscale
            img = Image.fromarray(img, mode='L')
        elif len(img.shape) == 3:  # RGB
            img = Image.fromarray(img, mode='RGB')

    # Convert grayscale to RGB for model
    if img.mode == 'L':
        img = img.convert('RGB')

    # Load model
    model = models['bmshj2018-factorized'](quality=quality, pretrained=True)
    model.eval()
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model = model.to(device)
```

This is the part we convert image to gray scale and load the “bmshj2018-factorized” model from compressAI.

```
# Transform and pad
x = transforms.ToTensor()(img).unsqueeze(0).to(device)
_, _, h, w = x.shape
pad_h = (64 - (h % 64)) % 64
pad_w = (64 - (w % 64)) % 64

if pad_h > 0 or pad_w > 0:
    x = torch.nn.functional.pad(x, (0, pad_w, 0, pad_h))

# Compress and decompress
with torch.no_grad():
    compressed = model.compress(x)
    decompressed = model.decompress(compressed['strings'], compressed['shape'])

# Post-process
x_hat = decompressed['x_hat']
if pad_h > 0 or pad_w > 0:
    x_hat = x_hat[:, :, :h, :w]

# Convert back to PIL Image and then to grayscale if input was grayscale
out_img = transforms.ToPILImage()(x_hat.squeeze().cpu().clamp_(0, 1))

# always convert back to gray
out_img = out_img.convert('L')

# Calculate stats - handle both PIL Image and numpy array inputs
if isinstance(img, np.ndarray):
    original_size = img.size * (1 if len(img.shape) == 2 else 3)
else:  # PIL Image
    original_size = img.size[0] * img.size[1] * (1 if img.mode == 'L' else 3)

compressed_size = sum(len(s[0]) for s in compressed['strings'])
```

This is where we further preprocess the image then apply the model to compress and decompress the image.

**For the experiments:**

```
def calculate_psnr(original, compressed):
    # def decode_binary_to_image(binary_string, shape):
    #     # Split binary string into 8-bit chunks
    #     byte_array = bytes(int(binary_string[i:i+8], 2) for i in range(0, len(binary_s

    #     # Convert byte array into a NumPy array and reshape to the original image shap
    #     decompressed_image = np.frombuffer(byte_array, dtype=np.uint8).reshape(shape)
    #     return decompressed_image

    # compressed = decode_binary_to_image(compressed)
    assert(original.dtype == np.float64 or original.dtype == np.float32)
    assert(compressed.dtype == np.float64 or compressed.dtype == np.float32)

    mse = np.mean((original - compressed) ** 2)
    if mse == 0:
        return float('inf')  # No error means infinite PSNR
    max_pixel = 1
    psnr = 10 * np.log10((max_pixel ** 2) / mse)
    return psnr
```

The figure above shows how we implemented the formula of PSNR, which is a measure of reconstruction quality.

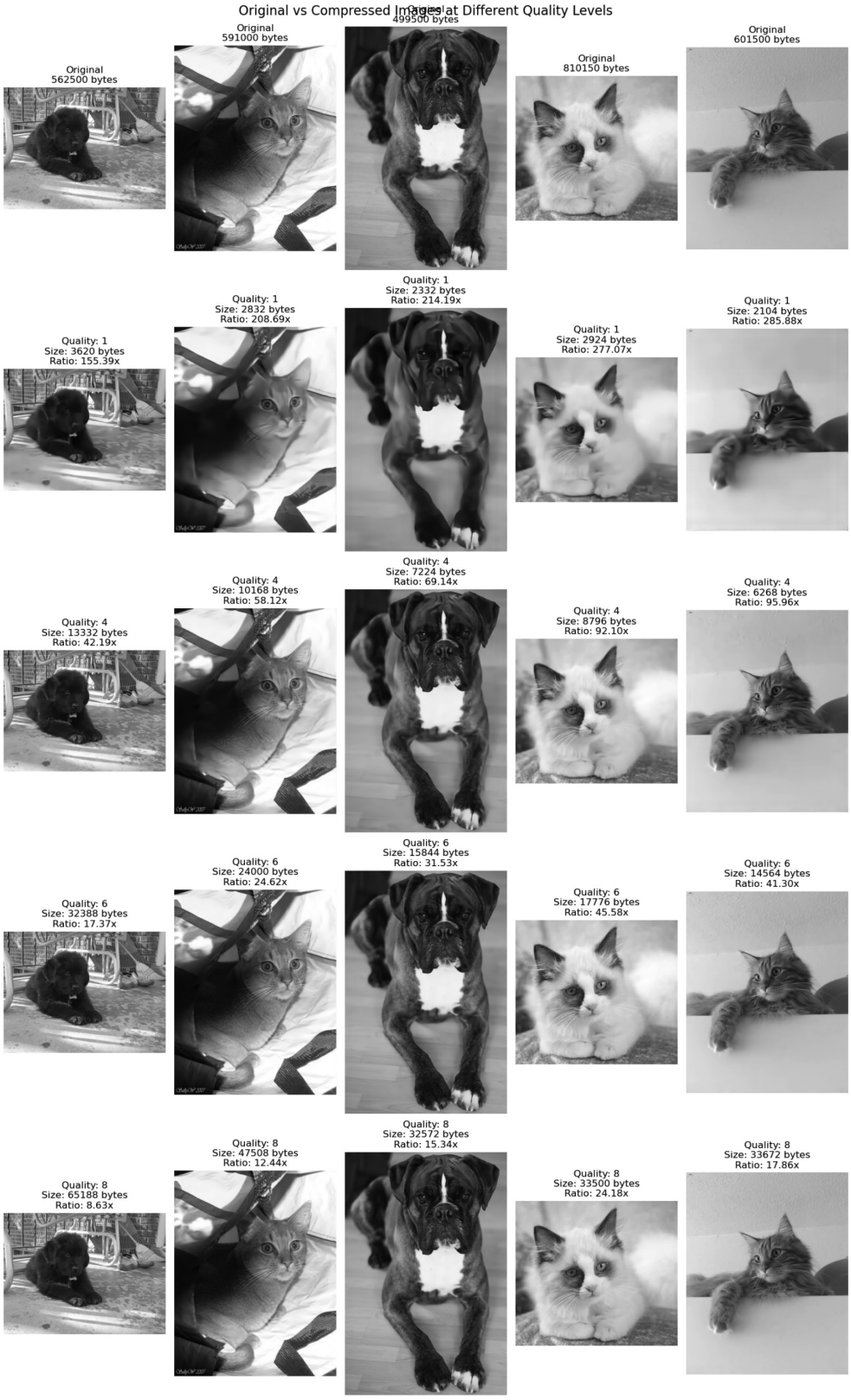
```
# SVD
# various quality
for k in [5, 10, 20, 40]:
    U_k, S_k, VT_k, compressed_image = svd_compress(image, k)
    compressed_bytes = sum(get_size_in_bytes(arr) for arr in [U_k, S_k, VT_k])
    psnr=calculate_psnr(image, compressed_image)
    svd_results[str(k)].append((psnr,compressed_bytes*8/num_pixel))

# Huffman
encoded_image, tree, shape = huffman_encoding(image)

# Calculate sizes
encoded_size = len(encoded_image) // 8  # Convert bits to bytes
tree_size = get_tree_size_bytes(tree)
shape_size = get_shape_size_bytes(shape)
total_compressed_size = encoded_size + tree_size + shape_size

# Decode image
decoded_image = huffman_decoding(encoded_image, tree, shape)
# huffman outputs integer image
if (decoded_image.dtype == np.uint8 or decoded_image.dtype == np.int8) :
    decoded_image = decoded_image.astype(np.float32) / 255.0

psnr=calculate_psnr(image, decoded_image)
huffman_results.append((psnr,total_compressed_size*8/num_pixel))

#LIC
results = compare_quality_levels(image)
for k in range(4): #quality 1, 4, 6, 8
    compressed_bytes = results[k]["compressed_size_bytes"]
    decoded_image = np.array(results[k]["out_img"])
    if (decoded_image.dtype == np.uint8 or decoded_image.dtype == np.int8) :
        decoded_image = decoded_image.astype(np.float32) / 255.0
    psnr = calculate_psnr(image, decoded_image)
    lic_results[str(k)].append((psnr,compressed_bytes*8/num_pixel))
print(f"Finished image: {i}\n") # report progress
```

The figure above shows how we apply all three methods on the same image and collect results.

```
# extract results
lic_psnr = [t[0] for sublist in lic_results.values() for t in sublist]
lic_bpp = [t[1] for sublist in lic_results.values() for t in sublist]

svd_psnr = [t[0] for sublist in svd_results.values() for t in sublist]
svd_bpp = [t[1] for sublist in svd_results.values() for t in sublist]

huffman_psnr = [t[0] for t in huffman_results]
huffman_bpp = [t[1] for t in huffman_results]

# Create the scatter plot
plt.figure(figsize=(10, 6))

plt.scatter(lic_bpp, lic_psnr, c='blue', label='LIC', alpha=0.7)
plt.scatter(svd_bpp, svd_psnr, c='red', label='SVD', alpha=0.7)
plt.scatter(huffman_bpp, huffman_psnr, c='green', label='Huffman', alpha=0.7)

# Customize the plot
plt.xlabel('Bits per Pixel (BPP)')
plt.ylabel('PSNR (dB)')
plt.title('Compression Methods Comparison')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# Show the plot
plt.show()
```

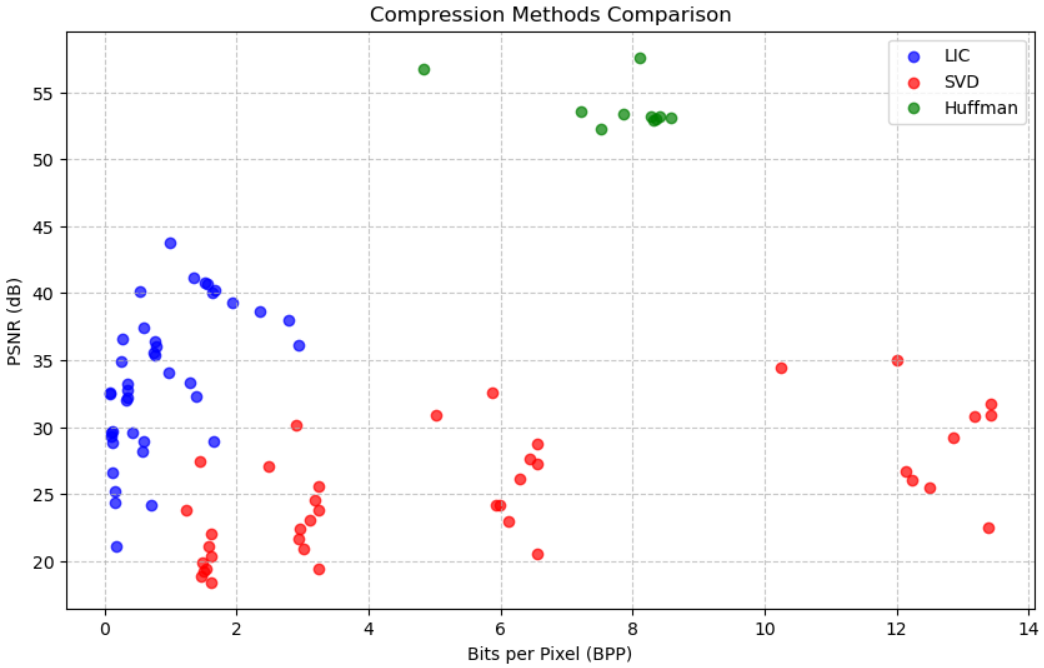
The figure above shows how we produced the combined scatter plot for our experiment.

### 3.6 SOTA Results



### 3.7 Compare and contrast

You can visually examine the results shown above, for each algorithm. It might be hard to tell which one of the algorithms performed the best. Here is a graph that can better quantify the differences between each algorithm:




This graph was generated by testing 10 images instead of 5, for better interpretation. In this graph, Huffman Coding does indeed lead in terms of image quality (PSNR), but it really struggles to bring down the file size any further as Huffman Coding's compression ratio largely depends on the complexity of the image itself. Looking at the red dots, SVD has 4 distinct lines of data points corresponding to different values of k. BPP for SVD data points has increased a lot without improving PSNR too much. Compared to SVD, LIC seems to be a lot more efficient. By changing the quality parameter in LIC to higher values, it raised the value of PSNR much higher than SVD did while BPP increased very little.

## 4 Conclusion

After a thorough exploration of these 3 different methods: SVD, Huffman Coding, and LIC, we can tell that they all have their own benefits and disadvantages. SVD, as we discussed, seems to be outperformed by LIC in every way but it is actually a faster compression method than LIC. Huffman coding struggles to compress images to really small files but it reserves all the details in the images. LIC is really efficient but it could be computationally expensive depending on the type and amount of images you want to compress. Overall, through the study of this project, we have learned enough about these 3 compression methods to know which one of them would fit our interest the best in specific situations, meaning we can make better decisions when considering the balance between image quality and compression ratio.

## 5 Acknowledgement

We are grateful to Prof. Lily Weng and the TAs of FA24 DSC 210 (Somanshu Singla & Yashowardhan Shinde) for their guidance and insights throughout this project.

## References


[1] N. Ahmed, T. Natarajan, and K. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[2] J. Bégaint, F. Racapé, S. Feltman, and A. Pushparaja. Compressai: a pytorch library and evaluation platform for end-to-end compression research. *arXiv preprint arXiv:2011.03029*, 2020.

[3] Mateen Ulhaq. Learned image compression: Introduction. `https://yodaembedding.github.io/post/learned-image-compression/#fn:25`, 2024. [Online; accessed 10-December-2024].

[4] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[5] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[6] Wikipedia contributors. High efficiency image file format — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=High_Efficiency_Image_File_Format&oldid=1259869875`, 2024. [Online; accessed 30-November-2024].

[7] Wikipedia contributors. Huffman coding — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Huffman_coding&oldid=1259970231`, 2024. [Online; accessed 30-November-2024].

[8] Wikipedia contributors. Jpeg — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=JPEG&oldid=1258902951`, 2024. [Online; accessed 30-November-2024].

[9] Wikipedia contributors. Jpeg file interchange format — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=JPEG_File_Interchange_Format&oldid=1256846574`, 2024. [Online; accessed 30-November-2024].

[10] Wikipedia contributors. Png — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=PNG&oldid=1258243989`, 2024. [Online; accessed 30-November-2024].

[11] Wikipedia contributors. Run-length encoding — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Run-length_encoding&oldid=1254292425`, 2024. [Online; accessed 30-November-2024].

[12] Wikipedia contributors. Vector quantization — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Vector_quantization&oldid=1202727154`, 2024. [Online; accessed 30-November-2024].

[13] Wikipedia contributors. Webp — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=WebP&oldid=1259667110`, 2024. [Online; accessed 30-November-2024].