JPEG compression detection and restoration

Caleb Tham

Supervisor: Victor Sanchez

Year of Study: 3rd

University of Warwick

May 2023

Abstract

JPEG compression is commonly used to reduce the storage size of digital images. However, the process introduces complex artifacts that degrade the visual quality of images. This is particularly problematic for images that are compressed multiple times, often referred to as double compressed images. Due to the lossy nature of JPEG compression, we are unable to retrieve the original image. Still, it is possible to enhance the image's visual appeal by estimating its appearance prior to its compression. To tackle this, we propose a system that detects whether an image has undergone JPEG compression and, if necessary, applies an artifact removal process. In particular, we develop a detector that uses a convolutional neural network to differentiate between single and double compressed images, achieving near state-of-the-art accuracy in this task, despite using a simpler architecture compared to existing methods. For the restorer, we use a generative adversarial network and demonstrate that it is equally effective on both single and double compressed images. Furthermore, by incorporating MS-SSIM and the widely used VGG-19 network into our perceptual loss function, as well as employing NoGAN training, we outperform a previous state-of-the-art method for JPEG artifact removal. We deploy our system as a web application.

***Keywords*—** JPEG artifact removal, JPEG compression, double compression, discrete cosine transform, convolutional neural networks, generative adversarial networks, web application

Contents

1 Introduction

The increased production and online sharing of digital images has led to ever-growing storage requirements and bandwidth usage. Therefore, compression has become indispensable, with JPEG compression being the most widely used method. Most digital images are subject to this type of compression, whether that be by the camera capturing the image, sharing the image to social media, or backing it up to the cloud.

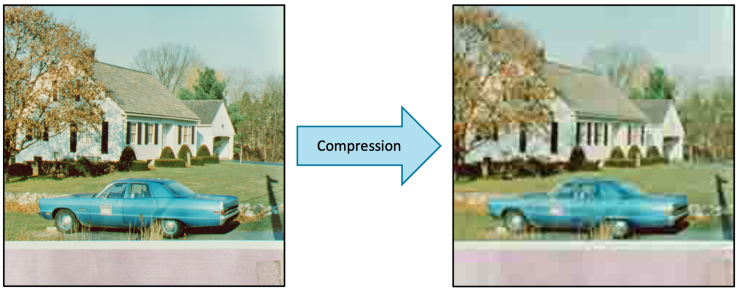


Figure 1: An image before and after JPEG compression. The compressed image shows obvious blocking artifacts. (Zoom in to inspect the blocking artifacts).

When compressing an image using the JPEG standard, some information is lost. During the quantization stage of JPEG compression, 8×8 pixel blocks of the image are transformed into coefficients of cosine waves of varying frequencies using the DCT2 algorithm. An 8×8 quantization table, determined by the quality factor (QF) of the compression, is then used to remove certain frequencies. The process is performed independently for each 8×8 block, without consideration for adjacent blocks. This leads to discontinuities at the block borders, which results in blocking artifacts. Additionally, high frequencies are more likely to be removed, leading to image blurring. This also causes the Gibbs phenomenon, which produces ringing effects along edges in the image [1].

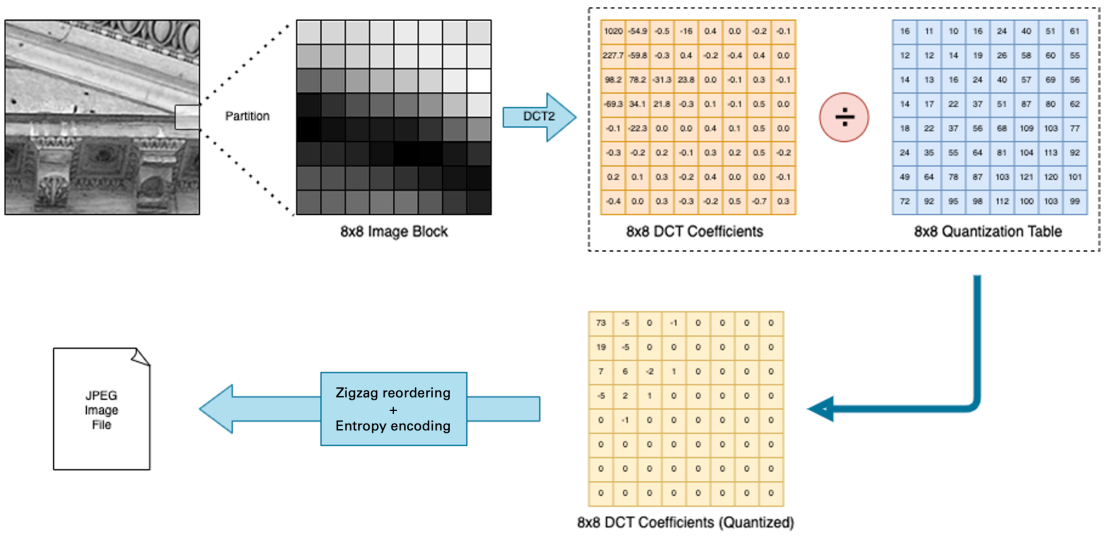


Figure 2: JPEG compression of a grayscale image. Diagram adapted from information in [2].

Due to the lossy nature of JPEG compression, it is impossible to recover the original uncompressed image from the compressed image. This is a problem since the introduced artifacts not only decrease the visual appeal of an image, but also have adverse effects for image processing applications, such as contrast enhancement [3] and edge detection [4]. In the case of contrast enhancement, the artifacts become more obvious to the human eye, and for edge detection, the blocking artifacts could be misinterpreted as edges of an object in the image. To counteract these issues, we develop an algorithm that can take a compressed image as input, eliminate or obscure the compression artifacts, and then output the resulting image.

Exacerbating the problem, however, a high proportion of images on the web have been subject to repeated compression arising from the downloading and reuploading of images on social media. Such images are referred to as "double compressed" images. The "double" term can be generalized for images that have been compressed *at least* twice. As a result, the negative effects of JPEG compression are compounded as the artifacts become more pronounced. We hypothesize that if we can detect whether an image is single or double compressed, we can use this information to inform a JPEG artifact removal process for improved results, as we can then account for the differences between the two. There has already been significant research into the detection of double compressed images in the field of digital forensics, but the focus here is usually to detect image manipulation [5]. Our main concern, on the other hand, is with restoring the visual quality of compressed images.

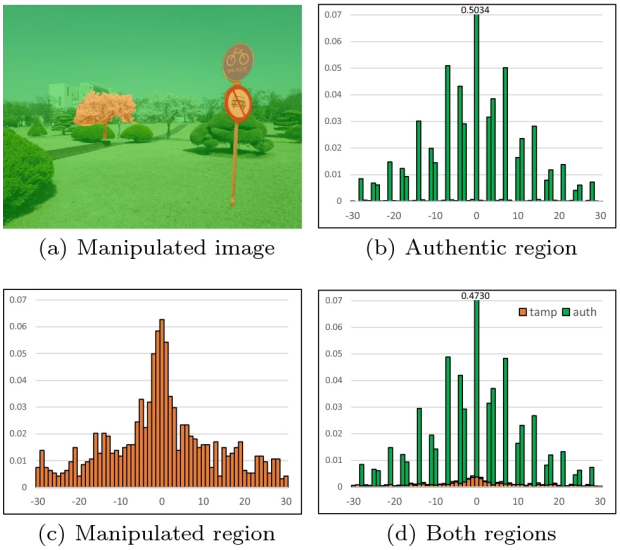
Thus, we propose a system which can reverse the degradation of quality in both single and double compressed images. Specifically, the system is able to detect single and double JPEG compression and use this information to indicate if JPEG artifacts need to be removed. If so, the system executes a JPEG artifact removal process. Motivated by current state-of-the-art methods, we use a convolutional neural network for the detection task and a generative adversarial network for the restoration task. Our detector model is based on an architecture proposed by Park et al. [6], although our model is smaller in terms of the number of parameters and removes quantization tables as an input. Thus, our model is more practical to use. Despite this, we are still able to obtain close to the same accuracy. In addition, our restoration model closely follows an architecture proposed by Zhao et al. [7]. Again, we make the model smaller, but we are able to improve its performance through NoGAN training and use of a novel perceptual loss function that considers information from multiple scales and incorporates the activations of the VGG-19 network, which is pretrained for image classification [8]. Our model also works on coloured images unlike Zhao's model.

The organization of this report is as follows. In Section 2, we look at existing methods for JPEG compression detection and artifact removal. Then, in Section 3, we propose our own system to address the limitations of the existing methods. In Section 4, we cover some technical background to understand our solution. In Section 5, we experiment with our system, and then in Section 6, we implement the final version. We evaluate our solution in Section 7, and discuss the management of our project in Section 8. Finally, we conclude in Section 9.

2 Related work

2.1 Double JPEG compression detection

The detection of double compressed JPEG images is a widely studied area, due to its significant applications in digital forensics – notably in image manipulation detection. Tampering of images often begins with a singly compressed image, which is loaded into an image editing software (decompressed), altered in some way, and then resaved (recompressed). Therefore, detecting double compression can provide evidence of image manipulation. We are able to differentiate between single and double compressed images since double compressed images leave detectable statistical traces, which we later explore. Furthermore, techniques from double compression detection can be used to localize the manipulated regions of an image. For example, in image splicing, only a portion of an image is altered as it is replaced with another image. When such an image is recompressed, the authentic region will show statistical characteristics of a double compressed image, whereas the manipulated region will likely not [9]. Thus, the manipulated area can be isolated. This is illustrated in Figure 3. While for our application we are not concerned with image manipulation, we are able to leverage the research already conducted in this field.



(a) Manipulated image (b) Authentic region

(c) Manipulated region (d) Both regions

Figure 3: This shows the statistical differences, due to double compression, between the Y-channel DCT-coefficients at the frequency (1,1) for manipulated and tampered regions. Taken from a paper on image manipulation by Kwon et al. [10]

In literature, it is widely known that the statistical traces left by JPEG compression are more pronounced in the DCT domain than in the pixel domain. Consequently, most methods to differentiate between double compressed and single compressed images rely on mathematical techniques applied to the DCT coefficients. However, it should be noted that the statistical characteristics of single and double compressed images are only distinguishable if the quality factor of the first compression (QF1) differs from that of the second compression (QF2). Histograms of the

DCT coefficients are a useful tool for revealing these traces, and it is usually sufficient to only consider the luminance channel (in YCbCr space). Double compressed images generally exhibit periodically distributed zeros or peaks in their histograms, which are not as prevalent for single compressed images. Figure 4 provides examples of these histograms. This distinction of single and double compressed histograms is also precisely what is shown in Figure 3.

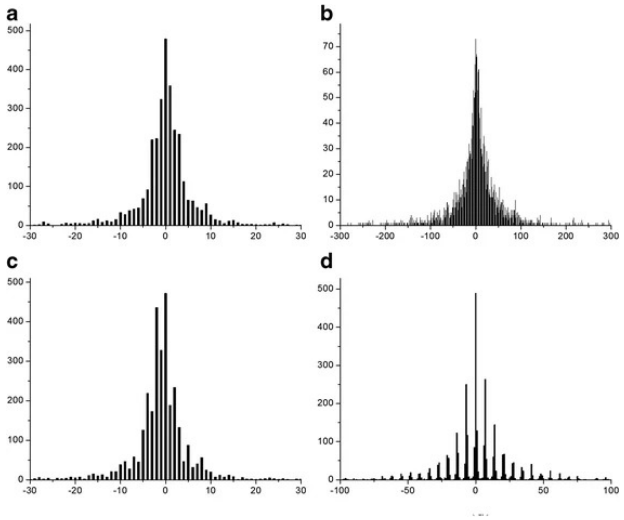


Figure 4: This shows histograms of the Y-channel DCT-coefficients at the position (0,1) for **(a)** QF1 = 60 only, **(b)** QF1=90 only, **(c)** QF1=90 and QF2=60 **(d)** QF1=60 and QF2=90. Taken from a paper on double compression by Wang et al. [9]

An early technique, developed by A. C. Popescu, showed that double compression detection could be achieved by examining the Fourier transform of the histograms of DCT coefficients [11]. However, the algorithm is complicated and laborious. Moreover, the testing was only conducted on a limited dataset of 100 images, and it still produced inaccuracies when QF1 > QF2, resulting in many false positives. Despite these limitations, the core idea showed promise and was later improved by B. Mahdian et al. who used machine learning, specifically with support vector machines, to analyze the FFT of the DCT coefficient histograms to reduce the number of false positives [12].

However, in recent years, deep learning techniques, notably with convolutional neural networks (CNNs), have seen huge developments, attributed to the groundbreaking results AlexNet achieved in the task of image classification [13]. This paper demonstrated how GPUs could be used to accelerate inference and training, which made the complex architecture feasible to use.

Since then, it has been shown that CNNs can also be applied in the double compression detection task to give excellent results that improve upon previous methods through supervised learning [5]. An approach we follow closely is given by Park et al., who also analyzes DCT histograms, but now represents the histograms as convolutional features and uses a CNN to classify whether a histogram corresponds to single or double compression [6]. The histogram features were obtained through a process proposed by Barni et al., who computes the DCT with

a CNN layer to help with network performance [14]. Park et al.'s method was able to achieve state-of-the-art results for various QFs, making it suitable for real-life applications [6]. However, it makes use of quantization tables that are stored in the JPEG file header, limiting its applicability to only JPEG images, which is not ideal. For instance, a compressed image could later be resaved in a lossless format, such as PNG, rendering the method unsuitable.

2.2 JPEG artifact removal

In the context of JPEG artifact removal or JPEG compression restoration, the objective is to effectively eliminate or conceal the artifacts, thereby improving the visual quality of an image and bringing it closer to the original uncompressed version. Current approaches can be categorized into either deblocking-oriented methods or restoration-oriented methods.

Most early research used deblocking-oriented techniques. The primary goal of such methods is to obscure the blocking and ringing artifacts of JPEG compression. It is possible to do this directly in the pixel domain. For example, one can simply apply a 3×3 filter along the block boundaries to blur the artifacts [15]. However, this also results in blurring of true edges. A more sophisticated approach is to apply a non-local means filter to the image blocks. This is a non-linear, edge-preserving, smoothing filter, whose weighted parameters are determined by the similarity of image block neighbourhoods so that the filter can be adaptively applied [16]. Another approach is to consider the DCT domain instead, using a shape-adaptive discrete cosine transform to reconstruct local estimates of the signal [17]. Although these filtering methods are effective in removing blocking and ringing, there is a loss of high frequencies, so they fail to accurately reproduce sharp edges, which leads to blurry textures.

Restoration-oriented methods are more able to address blurring, in addition to blocking and ringing. Similar to double JPEG compression detection, deep learning methods, particularly CNNs, offer superior results due to their non-linear mapping capabilities. In 2015, Dong et al. extended their CNN, originally for super-resolution, specializing it for the artifact removal task. This led to a four-layer CNN architecture (ARCNN) [18]. The model was trained using supervised learning, where the model attempts to predict the original uncompressed image from an input compressed image. ARCNN is able to obtain significantly sharper images compared to traditional deblocking techniques. Building upon this, more general-purpose CNN architectures that use skip connections were developed [19, 20]. These obtained improved results compared to ARCNN.

Furthermore, a recent development in the field of deep learning that has helped with compression restoration is the concept of generative adversarial networks (GANs), which were introduced by Goodfellow et al. [21]. GANs incorporate a discriminator network to enable the generation of more photorealistic results compared to using just a CNN [22]. They have drastically improved many image processing applications, including super-resolution [23], text-to-image generation [24], and notably image-to-image translation, which can be attributed to the development of

Pix2Pix [25]. Pix2Pix utilizes a "U-net" generator architecture, and introduces a "PatchGAN" discriminator [25]. A similar architecture can be applied for artifact removal since the task can be viewed as a translation of a compressed image to an image that appears uncompressed. Mameli et al. do exactly this [26], but they additionally employ a NoGAN training approach, previously used in the popular DeOldify architecture for colourization [27]. They showed that this resulted in a model that could generalize successfully for artifact removal [26]. Numerous other studies have demonstrated successful results using GANs for artifact removal, with many claiming to be the state-of-the-art both perceptually and quantitatively. These methods use varying training methods and perceptual loss functions, which we explain in detail later, but their model architectures often have similarities to Pix2Pix [26, 28, 7]. In this work, we closely follow the architecture proposed by Zhao et al. [7].

We note that this is not a problem isolated in literature. Similar methods have also been deployed in real-world applications. For example, Adobe Photoshop has "neural filters", which includes a "JPEG Artifact Removal" filter [29]. This allows artifact removal from images subject to "low", "medium", and "high" levels of JPEG compression. This is shown in Figure 5. However, it is not apparent what exact method is used.

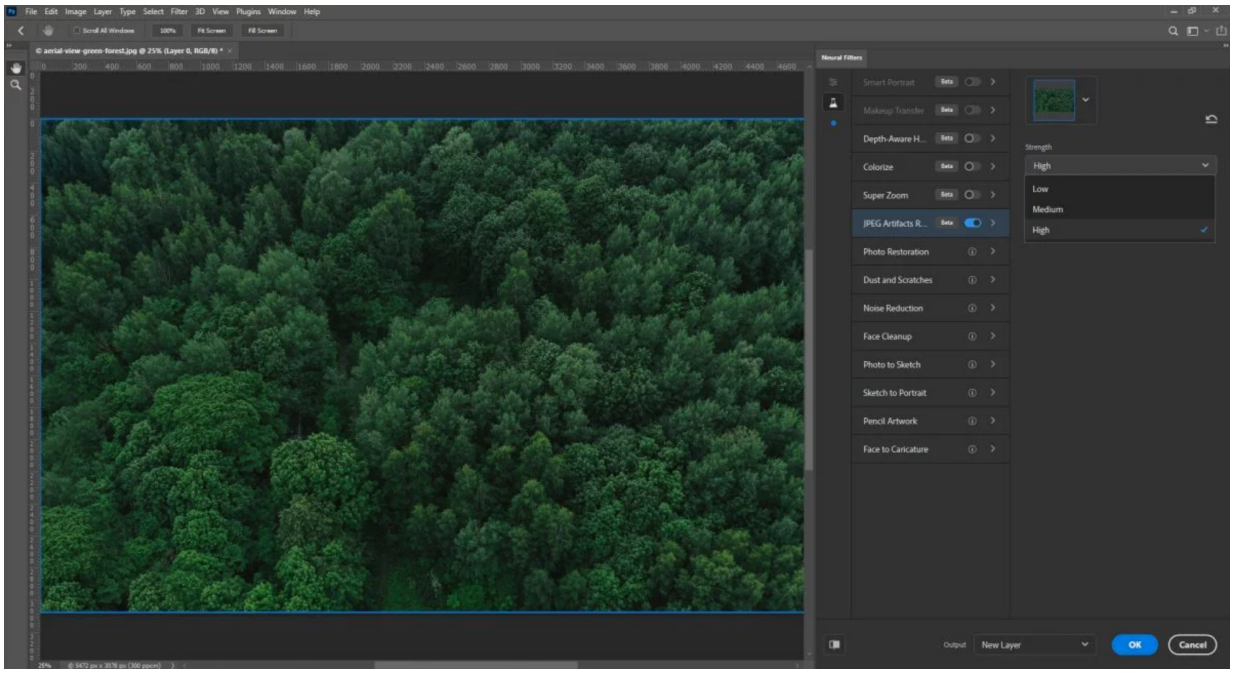


Figure 5: Adobe Photoshop's JPEG artifact removal neural filter. Taken from [30]

3 Objectives

Our primary goal is to create a flexible system for JPEG compression restoration that can be applied in many practical situations. To achieve this, we leverage the existing methods discussed in Section 2 while also addressing some of their limitations.

3.1 Addressing current limitations

One drawback with current JPEG artifact removal methods is that compression is not detected beforehand. As such, it is left to the user's discretion to decide whether an image needs to be restored; it would be unfavourable to apply restoration to an uncompressed image since the process alters the image and thus reduces its fidelity. This motivates the development of a system that can automate the process of deciding whether to restore an image by first detecting if it has been compressed.

Moreover, current JPEG artifact removal methods only consider single compressed images. Therefore, there is a lack of investigation into the effect of double compression on artifact removal, despite the abundance of double compressed images in the real-world. We therefore test a hypothesis that double compressed images require a more powerful restoration model compared to single compressed images in order to handle the more complex JPEG artifacts. Our experimental results demonstrate that our proposed model can effectively handle both cases with comparable performance.

Turning our attention to double compression detection, the method proposed by Park et al. [6] uses quantization tables taken from the JPEG file header, and hence there is an implicit assumption that images are stored in the JPEG format. While there are other methods that do not use quantization tables [9, 14], they still assume that the images are at least singly compressed. We remove this restriction and propose a method for detecting compression that does not assume any prior compression of images and does not require reading the file headers. Consequently, we rely solely on the pixel data of an image, making our system applicable to a wider range of scenarios, including when a compressed image is resaved in a lossless format. Thus, we can classify an image as either uncompressed, single compressed, or double compressed.

We also note that Park's model [6] is very large in terms of the number of parameters. We therefore attempt to minimize the size of our network for compression detection, while maintaining high accuracy. Similarly, we attempt to make our restoration model as small as possible. This means that the models have increased portability as less storage space is required, and they are faster in execution as the computational requirement is reduced.

With this, we propose a system that can input an image and detect whether it has undergone single, double, or no JPEG compression. Depending on the type of compression detected, it can then apply a suitable JPEG artifact removal algorithm. This process is illustrated in Figure 6. Our system could then be extended and deployed in a wide variety of applications – take for instance a web extension that scans all the images on a webpage and restores them only if

necessary. Here, we still save network bandwidth by transmitting compressed images, but for the user, the images appear uncompressed. For our purposes, we do not implement such a web extension, but we instead develop a user-friendly web application, where users can upload images, detect compression, and restore them if necessary. We use a CNN for our JPEG compression detector and a GAN for our JPEG artifact remover as these have previously given state-of-the-art results [14, 6, 26, 31, 7]. The CNN is based on Park et al.'s method [6], although we address the limitations as explained above. The GAN follows Zhao et al.'s approach [7], but additionally we employ NoGAN training [27] and propose a novel perceptual loss function to obtain improved performance. We also consider coloured (RGB) images so that our model is more practical.

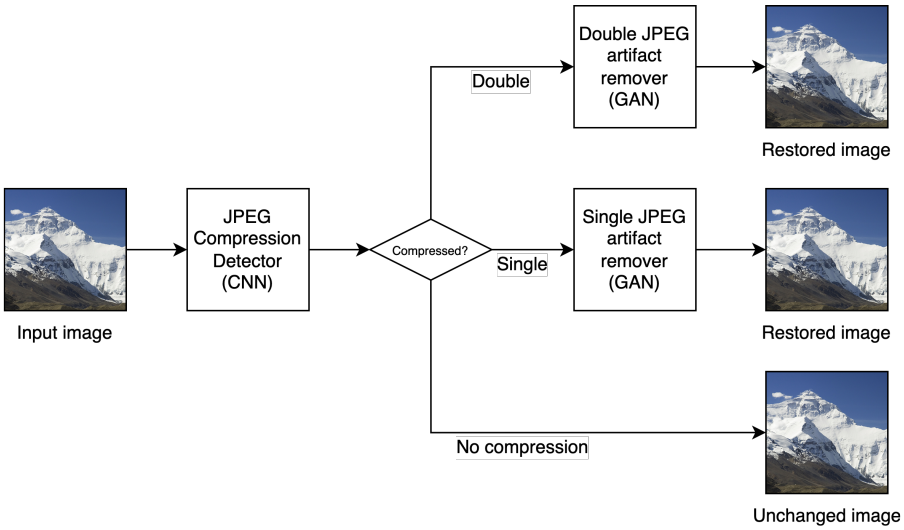


Figure 6: Outline of the proposed system pipeline, where we first detect the type of compression, and then apply the appropriate artifact removal process.

3.2 Aims

We can decompose our goal of developing a practical system for JPEG compression restoration as follows:

Detector We aim for a detector that: (1) is accurate in differentiating between images that have undergone single, double, or no JPEG compression, (2) is small in terms of the number of parameters, and (3) accepts a wide variety of images (e.g., any image format, any image size, and RGB images).

Restorer We aim for a restorer that: (1) is convincing in removing JPEG artifacts, (2) accounts for the differences between single and double compression, (3) is small in terms of the number of parameters, and (4) accepts a wide variety of images (e.g., any image format, any image size, and RGB images).

Final solution Using the detector and restorer models, we aim for a system that: (1) automates the process of detecting JPEG compression before removing any artifacts and (2) takes form of a web application.

3.3 System requirements

For us to successfully achieve our aims, we must adhere to the following system requirements, which were decided upon close to the beginning of the project. We met all of these requirements, excluding the **COULD** requirement, which we demonstrate in Section 7.

MUST The system must be able to:

M1. Input a batch of uncompressed images (e.g., PNG, TIFF) and output single compressed and double compressed versions of the image. These should then be suitably labelled, saved, and added to the dataset for models to be trained on.

M2. For an input (RGB) image, accurately determine if it has undergone single JPEG compression, double JPEG compression, or no JPEG compression, using a (small) CNN model that looks at solely pixel data (i.e., using no file header information).

M3. For an input (RGB) image, convincingly remove any JPEG artifacts, using a (small) GAN model. The new image should be saved in a lossless format.

M4. Allow a user to upload an image, and if it has been subject to any type of compression, remove the JPEG artifacts. The type of compression detected and new image should be displayed to the user, and they should have the ability to download the image in a lossless format.

SHOULD The system should be able to:

S1. Be easy to use for non-technical users by taking form of an application with a simple, user-friendly GUI.

COULD The system could be able to:

C1. Take form of a web extension, which reads all images loaded on a page, detects if they have been compressed, and if so, replaces them with restored versions of the image.

3.4 Restriction of scope

Initially, we were only going to consider one combination of image compression QFs to train the models as storage and performance constraints restricted the amount of data we could process. This is not ideal since images in the real-world are compressed with a variety of QFs and our

models would only be suitable for one. However, due to good progress throughout the project, we were able to consider a wider range of QFs. Still, as resource constraints were present, we compromised on considering QF $\in \{10, 30, 50, 70, 90\}$.

Also, double compression can technically be categorized into two types: aligned and non-aligned. Non-aligned compression refers to when an image may be cropped or resized before the second round of compression and thus the 8×8 compression blocks will not be aligned. Typically, this results in more complex artifacts. However, we limited our scope to only include aligned double compression, so that we could better focus our research given our limited time frame.

4 Background

In order to gain a deeper understanding of the problem, we introduce fundamental concepts and ideas that serve as a necessary foundation for solving it.

4.1 JPEG compression

4.1.1 JPEG compression algorithm

The JPEG compression algorithm can be broken down into five steps: (1) Colour space conversion, (2) Chroma subsampling, (3) Discrete cosine transform (DCT), (4) Quantization, and (5) Entropy coding [32].

In the first step, we convert the image from the RGB colour space to the YCbCr colour space. This separates the luminosity, given by the Y component, from the chrominance of the image, given by the Cb (blue-difference) and Cr (red-difference) components. We perform this conversion since the human eye is more sensitive to changes in luminosity than changes in colour. By separating brightness and colour, we can downsample colour more strongly. This downsampling occurs in the chroma subsampling stage, where we reduce the number of distinct colours represented in the image by discarding some and assigning each pixel to the nearest represented colour.

After subsampling, the image is partitioned into 8×8 blocks of pixels. Each block is transformed from the spatial domain (pixels) to the frequency domain (DCT coefficients) using the two-dimensional discrete cosine transform (DCT2). This gives an 8×8 block of frequency coefficients. Let $p(x, y)$ be the pixel values of the image, and $F(u, v)$ be the DCT coefficients. Then the DCT2 is defined as [33]:

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} p(x,y)\cos\left(\frac{(2x+1)u\pi}{16}\right)\cos\left(\frac{(2y+1)v\pi}{16}\right)$$

where

$$u, v \in \{0, \ldots, 7\}$$

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}}, & u = 0 \\ 1, & u \neq 0 \end{cases}$$

Then in the quantization phase, for each block, the DCT coefficients are "quantized", meaning they are element-wise divided by a quantization matrix. The resulting values are then rounded to the nearest integer. This is summarized in Equation 1. This has the effect of setting some quantized coefficients to be 0, hence removing the frequency represented. Note that the values in the quantization matrix are chosen based on the compression setting, or quality factor (QF). For simplicity, we only consider standard QFs, which specify the quantization matrix using a numbered system from 1 to 100. The lower the QF, the higher the values in the quantization

matrix, and hence the more aggressive the quantization. Generally, the quantization matrix values are designed to more heavily penalize higher frequencies, which humans are less able to perceive.

Let $Q(u, v)$ be the quantization matrix. Then, quantization is defined as:

$$F_Q(u, v) = \text{round}\left(\frac{F(u, v)}{Q(u, v)}\right) \tag{1}$$

After, the quantized coefficients are typically run-length encoded and Huffman encoded. Finally, the Huffman codes and other metadata are written into a compressed JPEG file format. The specifics of the encoding are not essential for us to understand. However, we note that the quantization matrix used in compression is stored in the JPEG file header.

The process for JPEG decompression (to recover the image) is simply the inverse of the algorithm above: entropy decoding, followed by dequantization, and then inverse discrete cosine transformation. Because of the many-to-one mapping of subsampling and quantization, the recovered image is only an approximation of the uncompressed image – some original information is lost.

4.1.2 Double JPEG quantization

Double compression means that the quantization process is repeated. We reconstruct a well-known analysis on how double quantization affects the distribution of the image's DCT coefficients [32]. We then also perform our own analysis into the limitations of using this approach for double compression detection.

Let Q_α and Q_β be the primary quantization matrix and the secondary quantization matrix, respectively. Derived from Equation 1, double quantization is given by:

$$F_{Q_\beta}(u, v) = \text{round}\left(\frac{F_{Q_\alpha}(u, v)Q_\alpha(u, v)}{Q_\beta(u, v)}\right) = \text{round}\left(\frac{\text{round}\left(\frac{F(u,v)}{Q_\alpha(u,v)}\right)Q_\alpha(u, v)}{Q_\beta(u, v)}\right)$$

Now consider the distribution of the DCT coefficients. For an uncompressed image, we can assume that (for "natural" images, including images taken from a camera) this will usually follow a normal distribution, centred around 0 (taking values from [. . . -2, -1, 0, 1, 2 . . .]). To simplify our analysis, assume that all entries in the quantization matrix take the same value, f. Therefore, the element-wise division using the matrix corresponds to a simple scalar division with factor f.

Firstly, we consider the case that a double compressed image is produced with a primary factor $f_1 = 2$ and a secondary factor $f_2 = 3$. Therefore, the single compressed image's reconstructed DCT coefficients will take even values from [. . . -4, -2, 0, 2, 4, 6, 8, 10, . . .]. When quantizing the second time we divide by 3 and round, giving values from [. . . -1, -1, 0, 1, 1, 2, 3, 3, . . .]. We notice how some values (e.g., -1, 1, and 3) are mapped to more frequently than other values (e.g., 0 and 2), with the pattern at which this occurs being at set intervals. This results in periodic peaks in the histogram for the DCT coefficients, which can be seen in Figure 7. In general, similar results are obtained for when $f_1 < f_2$ (which corresponds to when QF1 > QF2 for standard QFs).

Next, we consider the case that a double compressed image is produced with a primary factor $f_1 = 3$ and a secondary factor $f_2 = 2$. Therefore, the single compressed image's reconstructed DCT coefficients will take values from [. . . -6, -3, 0, 3, 6, 9, 12, 15, . . .]. When quantizing the second time we divide by 2 and round, giving values from [. . . -3, -1, 0, 2, 3, 5, 6, 8, . . .]. Notice how some values (e.g., -2, 1, 4, 7) are missing, with the pattern occurring at set intervals. This results in periodic zeros in the histogram for the DCT coefficients, which can be seen in Figure 7. In general, similar results are obtained for when $f_1 > f_2$ (which corresponds to when QF1 < QF2 for standard QFs).

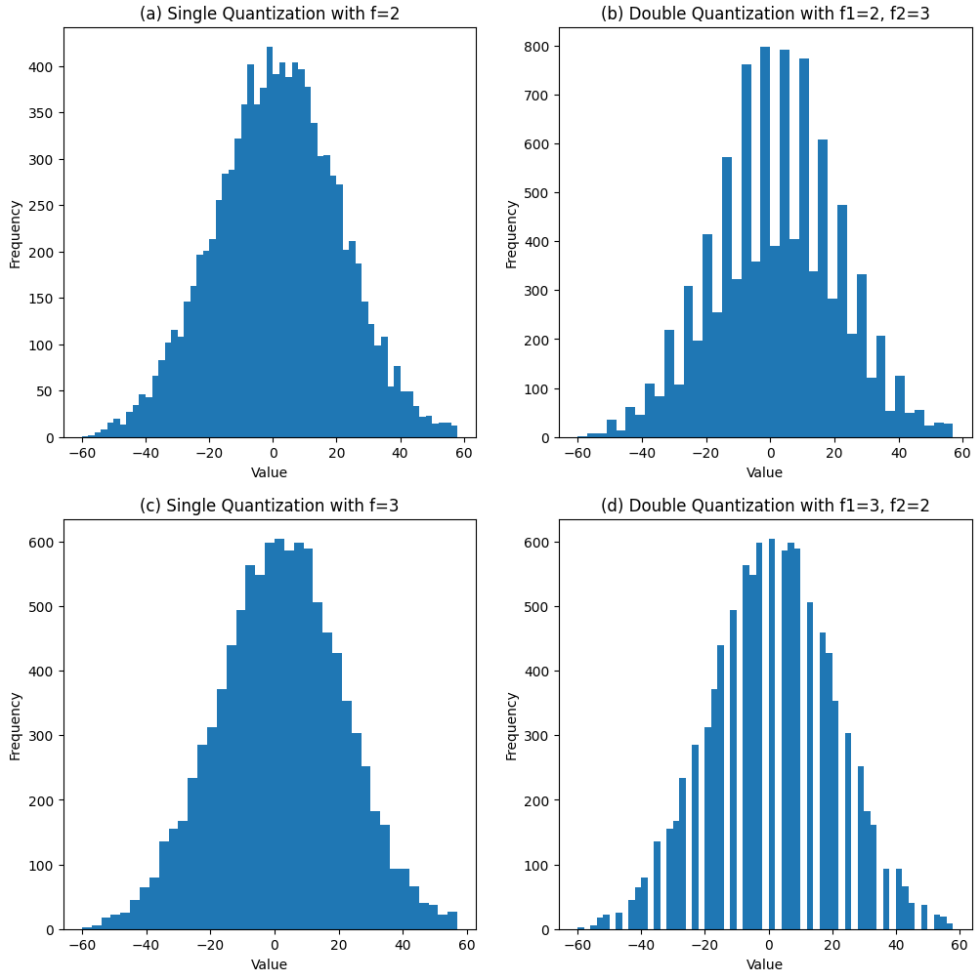


Figure 7: This shows the effect of quantization applied to random values following the normal distribution. Quantized with: **(a)** factor 2 only, **(b)** factor 2 then 3, **(c)** factor 3 only, **(d)** factor 3 then 2. Notice periodic peaks for **(b)** and periodic empty bins for **(d)**.

Hence, it is possible to differentiate between single and double compressed images by considering their DCT histograms. However, we note that our assumption of normally distributed coefficients is not always accurate, and quantization matrix entries almost never take just one value. Consequently, the histogram patterns may not be as obvious as depicted in Figure 7. Therefore, as discussed in Section 2, sophisticated methods are required to detect trends in the histograms.

Furthermore, there are still inherent problems with this approach. We note that when $f_1 << f_2$ (i.e., QF1 >> QF2 for standard QFs), the periodic peaks become much less noticeable. It then becomes difficult to discern whether the peaks are due to the effect of double quantization, or just variations in the image. An even more difficult case is when $f_1 = f_2$ (i.e., QF1 = QF2 for standard QFs). As shown in Figure 8, single and double quantization with $f = 2$ give identical histograms. The difficulty of these cases is well-documented in literature, and given the scope of our project, solving them was beyond our intended focus.

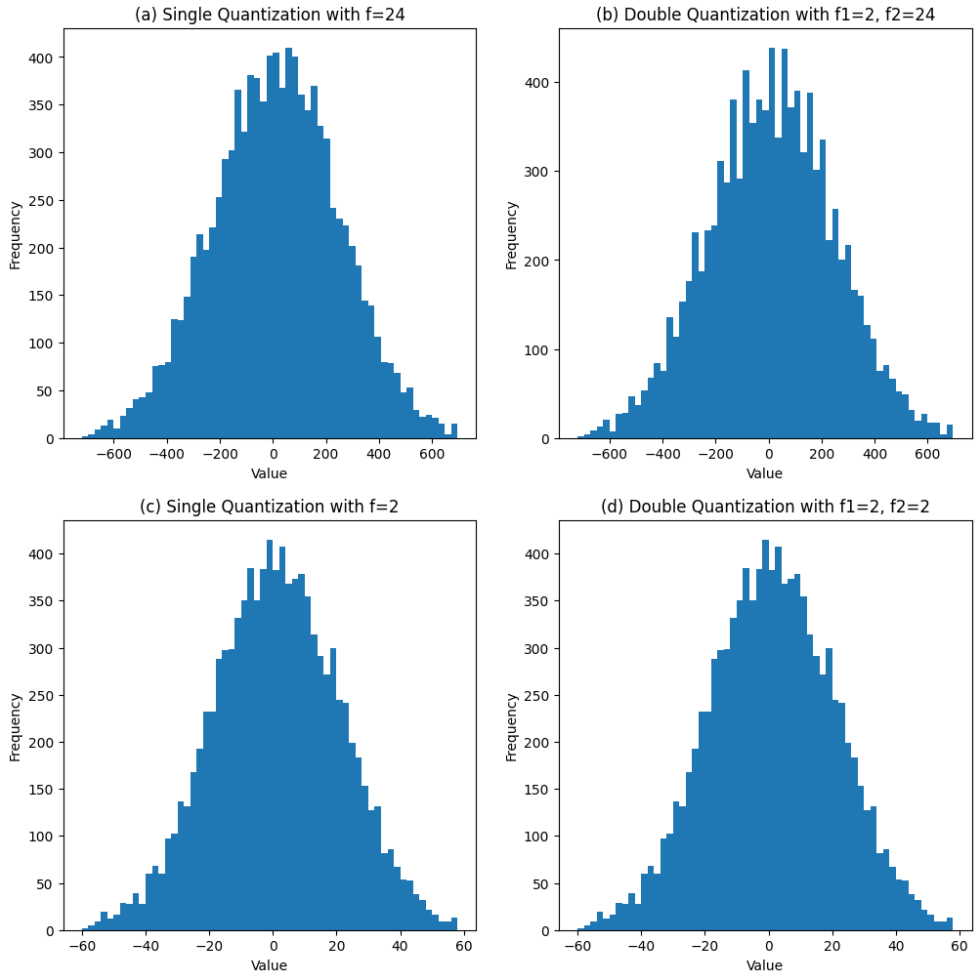


Figure 8: This shows challenging cases of detecting double compression. Quantized with: **(a)** factor 24 only, **(b)** factor 2 then 24, **(c)** factor 2 only, **(d)** factor 2 then 2 again. The histograms are much more difficult to discriminate. In fact, **(c)** and **(d)** are identical.

4.2 Artificial neural networks (ANNs)

For sake of brevity, we assume basic knowledge of neural networks and supervised learning techniques, including multilayer perceptrons and mini-batch gradient descent. We divert the reader to [34] for further reading. Still, we address some specific aspects of neural networks that we use throughout.

4.2.1 Sigmoid activation

The Sigmoid function is a common activation function, defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The main disadvantage of using this is that it suffers from the "vanishing gradient" problem. This is due to the gradient of Sigmoid having a maximum value of 0.25, and tending towards 0 as $x \to \pm\infty$. For deep networks, the near zero gradient is multiplied for each of the layers in backpropagation, making the gradients very small. This can result in the network refusing (or being too slow) to learn further (i.e., update its weights). Therefore, we do not use the Sigmoid function for hidden layers, but we still make use of it in output layers when we wish to predict a probability and thus require the output to be in the interval $(0, 1)$.

4.2.2 Tanh activation

Tanh is a rescaled version of Sigmoid, giving outputs in the interval $(-1, 1)$, as opposed to $(0, 1)$. Therefore, it also suffers from the vanishing gradient problem, so we only make use of it in output layers. It is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

4.2.3 ReLU activation

Rectified Linear Units (ReLUs) can be used as a way of avoiding the vanishing gradient problem as its gradient is 1 for $x > 0$. This results in much faster training, in terms of the number of iterations, and overall better accuracy compared to using binary units (i.e., Sigmoid or tanh) [35]. ReLU can be defined as the following:

$$\mathrm{ReLU}(x) = \max(0, x)$$

One disadvantage is that ReLU suffers from the "dying ReLU" problem, which refers to when ReLU neurons become inactive and only output 0 due to a negative input. This means that the gradient is always 0 and so updates are unable to be made during training, leading to a persistent 0 output. To solve this, we may use Leaky ReLU, which is defined as:

$$\mathrm{LReLU}_{\alpha}(x) = \max(\alpha x, x), \quad \alpha \in (0, 1)$$

Therefore, the output is only 0 for $x = 0$ and the gradient is α for $x < 0$, avoiding the dying ReLU problem. In practice, the dying ReLU problem is not always a hindrance so ReLU and LReLU may perform comparably [34]. We choose ReLU or LReLU depending on which gives better performance based on empirical experiments.

4.2.4 Softmax activation

Softmax activation is used for the output layers in neural networks for multiclass classification. Given a vector, $\mathbf{x}$, and a number of classes, k, it is defined as follows:

$$\mathrm{s}(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^{k} e^{\mathbf{x}_j}}$$

This ensures that the values for each class is between 0 and 1, and that the sum of the values is 1. Therefore, the output values represent a probability distribution. Softmax is useful because it "preserves the rank order of its input values, and is a differentiable generalization of the 'winner-take-all' operation of picking the maximum value" [36, p213].

4.2.5 Cross-entropy loss

We use cross-entropy as a cost function in our neural networks. Cross-entropy is used to measure the difference between two probability distributions, which for us is the probability distribution predicted by the neural network, $\hat{y}$, and the true probability distribution, y. If there are only two classes for the neural network to predict (and thus we are using a single output node), we use binary cross-entropy. If there are n examples in the training batch, then the binary cross-entropy cost function is defined as:

$$C(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$$

When there are multiple class labels, we use categorical cross-entropy. Let k be the number of classes. Then, the categorical cross-entropy cost function is defined as:

$$C(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} y_{i,j} \log \hat{y}_{i,j}$$

We use cross-entropy due to its use of log, which inverses the exp term(s) in the Sigmoid or Softmax functions. Objective functions that do not have a log term, such as mean squared error, can fail to train the network, even when it makes highly incorrect predictions [37]. This is due to large negative values in the exp term causing the gradient to vanish.

4.2.6 L2-regularization

Overfitting is a problem in machine learning where a model learns the training data too well and is unable to generalize to new data, hence giving poor performance on test data. One way we mitigate this is by using L2-regularization. This adds a penalty to our objective function based on the L2-norm of the model weights. This has the effect of discouraging very large weights, which is one cause of overfitting. If $C(y, \hat{y})$ is our objective function, and $\mathbf{w}$ represents the model weights, and $\lambda > 0$ is a hyperparameter, then the new objective function can be written as:

$$C(y, \hat{y})_{\mathrm{L2}} = C(y, \hat{y}) + \lambda \cdot ||\mathbf{w}||^2$$

4.2.7 Adam optimizer

Adam derives its name from "adaptive moment estimation" and is a gradient-based stochastic optimization algorithm that maintains per-parameter learning rates to help the network in training [38]. It combines aspects from Stochastic Gradient Descent (SGD) with Momentum and Root Mean Square Propagation (RMSProp), both of which involve calculating exponential moving averages.

Let $J(\theta)$ be the objective function based on the network parameters, θ. Then, $\nabla_\theta J(\theta)$ represents the gradient for $J(\theta)$ with respect to θ. Let $\alpha > 0$ be the learning rate, and let $\beta_1, \beta_2 > 0$ be hyperparameters to control the decay rate of the exponential moving averages for momentum and RMSProp, respectively (usually 0.9 and 0.999, respectively). Also let θ_t be the parameters at iteration t. Then, to update the parameters at iteration t, we use the following equations:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_\theta J(\theta_{t-1}) \tag{2}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla_\theta J(\theta_{t-1}))^2 \tag{3}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{5}$$

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{6}$$

We can see Adam's similarities to SGD with momentum and RMSProp. SGD with momentum uses similar to Equation 2 to calculate an exponential moving average of the gradients (the mean) to update its parameters, using the mean to centre the gradient. Therefore, previous values of gradients help guide the next step's direction. RMSProp uses similar to Equation 3 to calculate an exponential moving average of squared gradients (the uncentred variance) to normalize the gradient itself, using the uncentred variance to scale the updates. This balances the step size, decreasing the step for large gradients to avoid exploding, and increasing the step for small gradients to avoid vanishing. Since Adam uses Equation 2 and Equation 3, it achieves the benefits of both momentum and RMSProp.

As a result, Adam is well-suited for large-scale problems, in terms of the data and the number of parameters involved. In practice, it results in faster and better overall convergence compared to other stochastic optimization methods [38]. We utilize Adam for all our neural network training.

4.2.8 Batch normalization

Batch normalization is a technique that standardizes the inputs to a layer for each mini-batch. It applies a transformation that maintains the mean output (close) to β and the output standard

deviation (close) to γ, where β and γ are learned parameters. Let x be the values over a mini-batch, $\mathcal{B}$, and let ϵ be an arbitrarily small value. Also let $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$ be the mean of $\mathcal{B}$, and let $\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ be its variance. Then, the batch-normalized values are defined as follows:

$$\hat{x}_i = \gamma \cdot \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta$$

It has been shown that re-centering and re-scaling in this way stabilizes training, reduces the sensitivity of hyperparameters, and results in much faster network convergence [39].

4.3 Convolutional neural networks (CNNs)

In deep learning, a convolutional neural network (CNN) is an artificial neural network specialized for processing data with a grid-like topology [34]. For our purposes, we consider two-dimensional or three-dimensional input data (i.e., grayscale images or images with multiple channels, such as RGB images). CNNs are able to input images and learn various aspects/objects about the images. A key benefit of CNNs over traditional feed-forward neural networks is that there is less preprocessing required with CNNs as they can independently learn characteristics that previously hand-engineered features would try to encapsulate. They are able to do this by using "convolutional" layers. Also, "pooling" layers are usually present, with "max-pooling" being the most widely used variant.

4.3.1 Convolutional layers

CNNs use convolutional layers to extract high-level features of an input image with a "kernel" or "filter", which is a multidimensional array of parameters that are learned in training. In the convolution operation, a filter is traversed across the image with a certain stride value, and at each step, it calculates a weighted sum based on the sub-image values and the filter weights. The output of the filter operation is referred to as a "feature map". A single convolutional layer may have multiple filters, resulting in multiple feature maps. If there are f filters, then the output of the convolutional layer will be an image with f channels, with each channel representing a feature map.

Let the input to the convolutional layer be an image, I, which has multiple channels. I is therefore a three-dimensional array, and so a single filter will be a three-dimensional array. We can represent multiple filters as a single four-dimensional array, K. Now, $I_{i,j,k}$ gives the value of the image within the i^{th} channel at the j^{th} row and k^{th} column. $K_{i,j,k,l}$ gives the weight of the i^{th} filter for the j^{th} channel of the input, with an offset of k rows and l columns. If we have a stride value of s, the convolution function, c, can then be defined as [34]:

$$c(K, I, s)_{i,j,k} = \sum_{l,m,n} I_{l,m+sj,n+sk} K_{i,l,m,n}$$

where $c(K, I, s)_{i,j,k}$ denotes the value of the output within the i^{th} feature map in its j^{th} row and k^{th} column. Note that we assume indices are 0-based. In practice, we may also pad the input I with zeros to ensure that the height and width of the output feature maps are the same as the input image. Figure 9 demonstrates a two-dimensional convolution, where there is no padding, resulting in the image size to decrease.

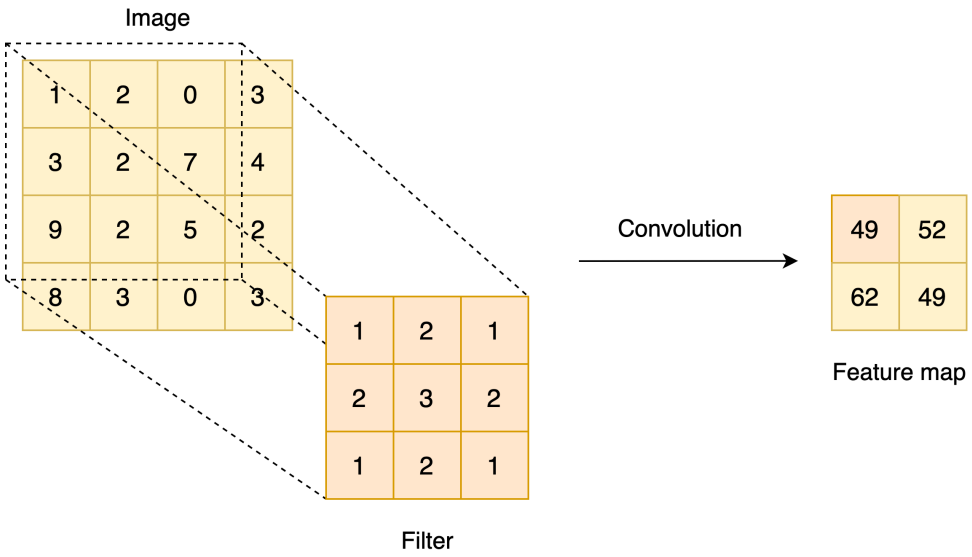


Figure 9: Convolution of a 4×4 grayscale image using a 3×3 filter, with a stride of 1 and no padding, resulting in a 2×2 feature map.

The use of convolutional layers allows the network to capture the spatial and temporal relations of an image, with each layer extracting some features. Stacking convolutional layers allows the network to learn more complex high-level features by extracting more information from the previous feature maps. Shallower layers may pick up on low-level features, such as edges, while deeper layers may pick up on high-level features, such as objects in the image. This makes convolution well-suited for processing JPEG artifacts for our restorer model as artifacts are complex and spatially dependent. For our detector, we perform convolution on the DCT histograms since the periodic zeros or peaks are also spatially dependent. Here, two-dimensional convolutional layers are used to additionally capture the relationship between adjacent histograms [14].

4.3.2 Max-pooling layers

In addition to convolutional layers, CNNs also use pooling layers, typically after the activation of convolutional layers, to reduce feature dimensions and computational resources. Max-pooling is the most widely-used approach, in which a two-dimensional pooling window of a specified size moves across an image and only returns the maximum value (for each channel) in the window. Pooling with a window size of $M \times N$ reduces the total size of a feature map by a factor of MN.

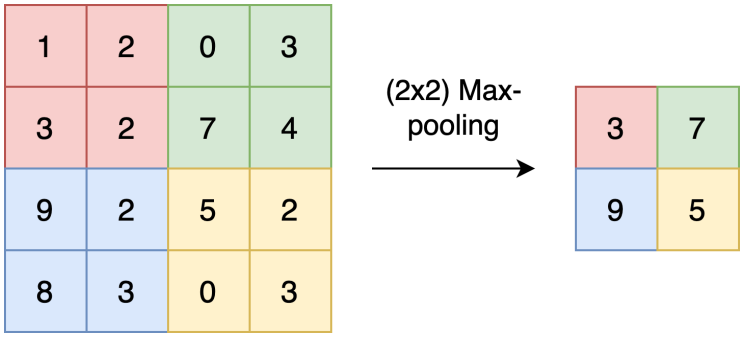


Figure 10: Max-pooling of a 4×4 grayscale image using a 2×2 window, resulting in a 2×2 image.

Max-pooling has the effect of discarding unnecessary details and noise, abstracting the features of a feature map and reducing the computational cost of the network. This permits the extraction of even more complex features while making the network more resilient to noise.

4.4 Generative adversarial networks (GANs)

A generative adversarial network (GAN) is composed of two networks: a generator and discriminator. For us, these will both be CNNs since our task involves images. During training, the generator and discriminator are in a competitive game, where the discriminator attempts to distinguish between "real" images from a dataset and "fake" images produced by the generator, which are made to look like they are from the dataset. The generator uses the gradient of the discriminator's loss function to update its own weights in backpropagation. This means that the generator will become better at producing fake images that can fool the discriminator while the discriminator will improve in distinguishing between real and fake images. Theoretically, on convergence, the generator should produce images that are indistinguishable to ones in the dataset, and so the discriminator must give a 50-50 guess [34].

For our task of image restoration, we train a GAN using supervised learning, with our dataset consisting of compressed and uncompressed versions of images. In training, when the generator generates an image from an input compressed image, it uses the corresponding uncompressed image (the ground truth), in addition to the discriminator's gradients, to learn and update its weights. To do this, we require a conditional GAN, rather than a traditional GAN.

4.4.1 Traditional GAN loss

In the paper that originally introduced GANs [21], the discriminator model, D, and generator model, G, play a two-player minimax game where G tries to minimize a value function, $V(D, G)$, while D tries to maximize it. Let y be data from the dataset and let z be a random noise vector. Note that the shape of y is the same as $G(z)$, and $D(x)$ outputs a probability representing how likely x is from the dataset. The game objective is as follows:

$$\min_G \max_D V(D, G) = \mathbb{E}_y[\log(D(y))] + \mathbb{E}_z[\log(1 - D(G(z)))]$$

To achieve this using backpropagation, the loss functions of the discriminator and the generator can be respectively defined as:

$$L_D = -\log(D(y)) - \log(1 - D(G(z)))$$

$$L_G = \log(1 - D(G(z)))$$

4.4.2 Conditional GAN (cGAN) loss

As we can see, in a traditional GAN, the generator only takes in random noise to generate an output. This of course is not helpful for our restoration task since we need the generator to output an image that is dependent upon an input compressed image. To address this, we can use a conditional GAN (cGAN), where both the discriminator and generator now observe the input image [40].

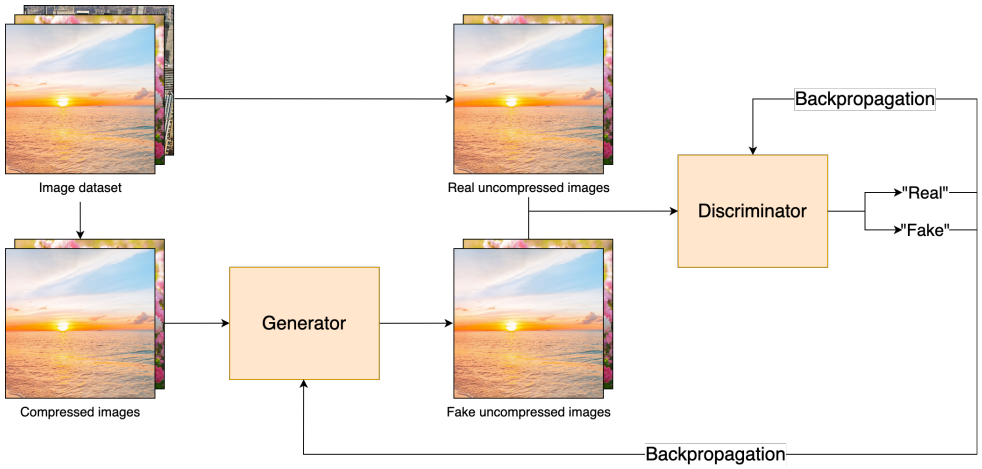


Figure 11: Relationship of the generator and discriminator of a conditional GAN in the setting of generating fake uncompressed images (i.e., restored images) from compressed images.

The conditional GAN objective is similar to the traditional GAN, but D and G now take an extra input x that they use to condition their output on. Thus, the objective becomes:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x,y}[\log(D(x, y))] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z)))]$$

which gives the discriminator and generator loss functions respectively as:

$$L_D = -\log(D(x, y)) - \log(1 - D(x, G(x, z)))$$

$$L_G = \log(1 - D(x, G(x, z))) + L_p(G(x, z), y)$$

There is an extra term, L_p, in the generator's loss function, which corresponds to a "perceptual" loss function. Now, instead of simply trying to fool the discriminator, the generator also attempts to minimize the distance from its generated image, $G(x, z)$, and a given ground truth image from the dataset, y. L_p is used to measure this distance.

An additional benefit of cGANs is that they are less prone to the "mode collapse" problem, where the generator simply learns to produce a limited set of outputs that it knows can fool the discriminator. This is due to the conditioning inputs encouraging the network to produce diverse outputs [40].

4.4.3 Perceptual loss

For our task, we require the perceptual loss function to measure the difference between a generated image and the corresponding ground truth image. There are many candidates for this, and they can be categorized as either pixel-based or perception-based.

Pixel-based approaches simply compare pixels element-wise using vector norms. Mean absolute error (MAE), or L1-loss, uses the L1-norm. Assume that $y_{i,j}$ and $G(x)_{i,j}$ each represent a vector of the images' c channel values at row i and column j. Then, MAE can be defined as:

$$\text{MAE}(x, y) = \frac{1}{mnc} \sum_{i=1}^{m} \sum_{j=1}^{n} ||y_{i,j} - G(x)_{i,j}||_1$$

Another pixel-based loss function is mean squared error (MSE), or L2-loss, which uses the L2-norm as follows:

$$\text{MSE}(x, y) = \frac{1}{mnc} \sum_{i=1}^{m} \sum_{j=1}^{n} ||y_{i,j} - G(x)_{i,j}||_2^2$$

However, pixel-wise measures do not effectively capture the *perceived* differences between images. They are unable to consider the overall structure and visual patterns present in the images. For example, if we were to translate all pixels of an image by (1,1), the MSE and MAE between the original and shifted images may be very high even though the semantic information of the two images are the same. To address this issue, we can use perception-based measures, which are more invariant to changes in the pixel space.

One such approach is to use a loss function based on the multiscale structural similarity index measure (MS-SSIM). MS-SSIM is an extension of structural similarity index measure (SSIM), which is a widely used metric for image quality assessment as it accounts for structural information, including luminance, contrast, and textures. SSIM uses the following formulas to measure similarity in luminosity, contrast, and structure between x and y, respectively [41]:

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

where C_1, C_2, C_3 are small constants (whose derivations we omit), μ_x is the mean of x, σ_x is the standard deviation of x, and σ_{xy} is the standard deviation of x and y together.

To calculate MS-SSIM, we iteratively apply a low-pass filter and downsample the filtered image by a factor of 2. The original image is indexed as 1, and the most downsampled image is indexed as M (obtained after $M-1$ iterations of scaling). In each scaling iteration, j, we use the SSIM formulas (shown above) to calculate similarity in contrast and structure, denoted as $c_j(x,y)$ and $s_j(x,y)$, respectively. We only calculate the luminance similarity at the M^{th} scale, denoted as $l_M(x,y)$. The MS-SSIM is then defined as [41]:

$$\text{MS-SSIM}(x,y) = l_M(x,y)^{\alpha_M} \cdot \prod_{j=1}^{M} [c_j(x,y)^{\beta_j} \cdot s_j(x,y)^{\gamma_j}]$$

where the β_js, γ_js, and α_M are constants used to adjust the importance of the different components. Note that the above shows the calculation for one image channel. For RGB images, we take a weighted average for each channel.

By utilizing multiple scales, MS-SSIM provides more flexibility as it accounts for variations in viewing conditions, which means it can give a better approximation for perceived image quality [41]. However, since MS-SSIM measures the similarity rather than the differences of images, we must negate it when using it as a loss function. Therefore, we can define the loss function as:

$$L_{\text{MS-SSIM}}(x,y) = 1 - \text{MS-SSIM}(y, G(x))$$

Another perception-based loss function that we use is based on the high-level features extracted from the VGG-19 network, which is a CNN pretrained for image classification [8] on the ImageNet dataset [42]. Specifically, when comparing two images, we input each of the images to the network and obtain the activations of the 16^{th} convolutional layer of the VGG-19 network. We denote $\mathcal{V}(x)$ as the output of this operation for the image x. Then, $\mathcal{V}(x)_{i,j}$ represents a vector of each of the feature maps' values in the i^{th} row and j^{th} column. To obtain the loss, we use the squared L2 distance of the outputs. Thus, the loss can be described as:

$$L_{\text{VGG}}(x,y) = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} ||\mathcal{V}(y)_{i,j} - \mathcal{V}(G(x))_{i,j}||_2^2$$

A VGG-based perceptual loss was first introduced in a GAN for style transfer and super-resolution [43]. It was found that using the deep features of a pretrained CNN for the loss function resulted in more visually pleasing results than using a pixel-based loss. It was suggested that this is due to the perceptual loss function transferring semantic knowledge from the VGG-19 network to the generator network, and thus the generator network does not need to learn relevant image semantics from scratch. This idea is also helpful for our restoration task.

4.4.4 GAN loss function remarks

It is possible to have a generator network without the discriminator network, training it with any one of the perceptual loss functions discussed. This corresponds to a stand-alone generative CNN model whose loss function does not include the cGAN loss. However, as discussed in

Section 2, this results in less visually pleasing results. Generators relying solely on pixel-wise loss functions, such as MSE, struggle to handle the uncertainty inherent in images. This is due to MSE encouraging the network to find pixel-wise averages of plausible solutions, rather than images in the "natural image manifold", which is the set of plausible photorealistic solutions [23]. This results in images that appear overly smooth since high frequency information such as textures are not produced [44]. In contrast, with a GAN, the discriminator is able to detect the lack of high frequency details as an indication that an image is fake. Therefore, the GAN loss guides the generator to avoid this in training, and so it learns to produce images that exist in the natural image manifold. The effect of this is illustrated in Figure 12.

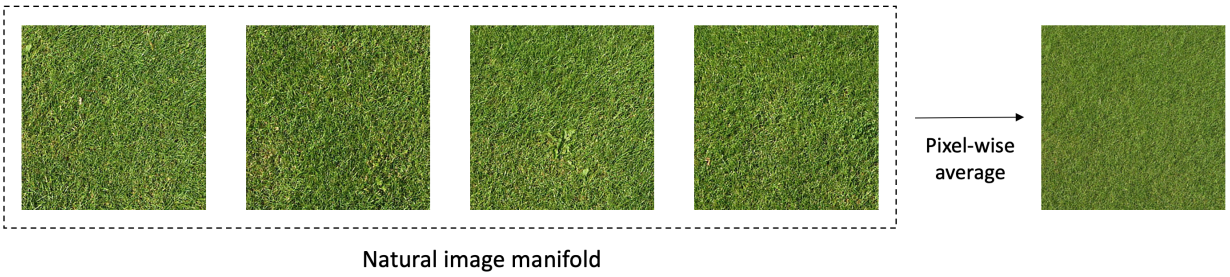


Figure 12: This shows possible images of grass in a natural image manifold, and we take the pixel-wise mean of these images. Notice the inherent variation in the natural images, which is lost in the mean. The mean image also looks smoother. A GAN is more likely generate an image in the natural image manifold than a generative CNN without a discriminator.

Regarding the perceptual loss in a GAN, existing approaches for JPEG artifact removal generally use one of the pixel-based loss functions [26, 28, 31, 7]. Some approaches additionally have a perception-based loss [26, 28, 31]. However, there are mixed opinions on which combination is the best. For example, Galteri et al. use MSE and SSIM for its loss function [28]. However, Zhao et al. do a comparison on different combination of loss functions and found that MAE loss outperforms MSE loss, which they claim is due to MSE being more prone to becoming stuck in a local minimum [45]. They also show that a SSIM-based loss resulted in noise around edges in the image, which was not the case for MS-SSIM. They therefore suggest a mix of MAE and MS-SSIM for the loss function. However, they omit comparison of a VGG-based loss. In other research, Abu-Srhan et al. do include VGG-based losses in their comparisons, and while they show that a MAE and VGG-based loss could obtain better results than a MAE and SSIM-based loss, this was in a different image-to-image translation task [40]. In our image restoration task, Luo et al. are able to use a MAE and VGG-based loss to obtain good results [31], but they do not compare it to other loss functions. Therefore, it is not clear which loss function would result in the best results. So, to help us choose the most effective one, we do our own comparison of different loss functions, which can be seen in Section 5.

4.4.5 Pix2Pix architecture: U-net and PatchGAN

U-net is a CNN architecture originally introduced for the task of biomedical image segmentation [46]. However, its use in GANs (as the generator) was popularized by Pix2Pix, a network for image translation [25]. Previously, problems in this area used a CNN-based encoder-decoder network [47], where the input image is progressively downsampled using convolutional layers, until a bottleneck layer, and then is upsampled using transposed convolutional layers. The convolutional layers are able to represent the image with deep features, which are then used by the transposed convolutional layers to construct a new image. This type of architecture also allows inputs of varying sizes due to use of only convolutional parts. However, a problem with this is that all the information about an image must pass through every network layer, including the bottleneck. This could lead to a loss of low-level information extracted by the early downsampling layers. This is disadvantageous since the low-level information represents the fine-grained details of an image, which may be important for the construction of the generated image. To address this problem, the U-net architecture still uses an encoder-decoder design, but it adds skip connections to bypass the bottleneck. The convolutional and transposed convolutional layers all have a stride of 2, which allows them to downsample/upsample by a factor of 2. Thus, the encoder and decoder are symmetric in terms of the number of layers (and number of channels in each layer). The skip connections are between each layer i and layer $n - i$, where layer n is the last layer. When a downsample layer is connected to an upsample layer via a skip connection, their outputs are concatenated. This allows the low-level information to be recovered in the output [25].

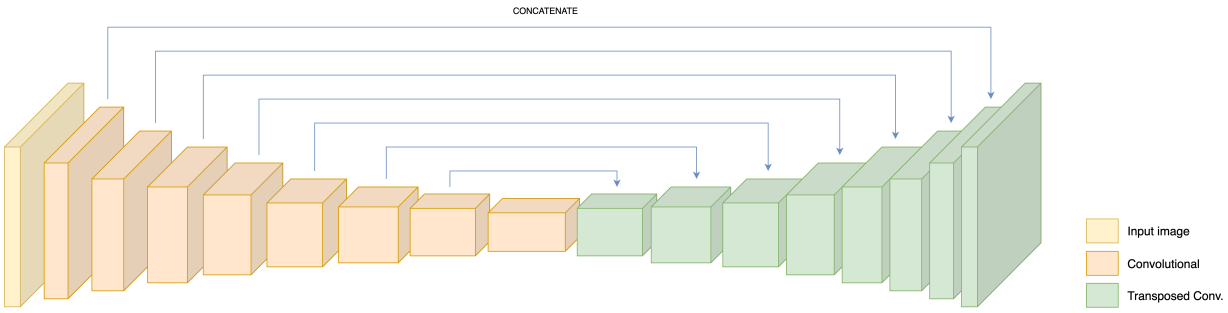


Figure 13: Pix2Pix's [25] U-net generator. If a convolutional layer is connected to a transposed convolutional layer by a skip connection, this means that the output of the convolutional layer is concatenated to the output of the transposed convolutional layer. (The width of the blocks in this diagram is determined by the number of channels in the output, and hence skips occur between blocks of the same width.).

In addition, Pix2Pix also uses a Markovian discriminator, named "PatchGAN" [25]. The idea is to restrict the GAN discriminator to only model the high frequency structure of images since the pixel-based perceptual loss can effectively capture low frequencies. Therefore, PatchGAN only penalizes structure in patches, predicting the probability that an $N \times N$ patch is real. The discriminator is run convolutionally across the image, averaging the output of each patch for the

final output of D. It is shown that N can be much smaller than the full size of an image while still providing the benefits of a GAN loss, which are explained in Section 4.4.4. Therefore, PatchGAN has fewer parameters, is less computationally expensive, and can be applied to images of varying sizes [25]. It is suggested that a 70×70 PatchGAN gives the best trade-off between performance and efficacy.

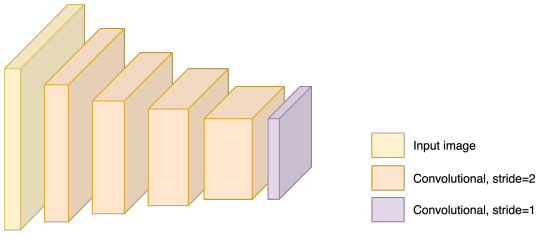


Figure 14: Pix2Pix's [25] 70×70 PatchGAN discriminator. The first four layers are the same as the first four downsampling layers in the generator architecture. The last layer is a convolutional layer that has a 1-channel output, which is Sigmoid activated. Therefore, each entry in the final output gives the probability that the corresponding patch is from a real image.

Many existing methods for JPEG artifact removal [26, 28, 7] use ideas from the Pix2Pix architecture to obtain good results, and we do the same.

4.4.6 Resize-convolution layers

The U-net architecture makes use of transposed convolutional layers in its decoder [46]. This can result in some images generated by the neural network to exhibit checkerboard patterns [48]. This is due to the transposed convolution unevenly mapping the input feature maps to the output, which is illustrated in Figure 15.

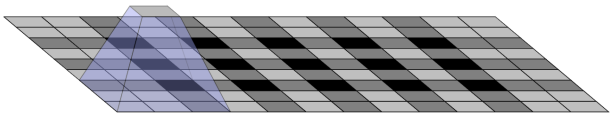


Figure 15: Uneven mapping caused by transposed convolution. Here, the stride is 2 with a kernel of size 3×3. Taken from [48].

The checkerboard patterns produced by transposed convolutional layers are dependent upon the size and stride of the kernel. A stride that is a factor of the kernel size usually helps mitigate the production of the patterns as the mapping is more even, but it is still not completely resolved. For example, Pix2Pix uses a kernel size of 4×4 [25], which is a multiple of its stride size of 2. Another approach is to use "resize-convolution" layers, which resize the image (usually using nearest-neighbour interpolation) before applying a convolutional layer (which has a stride of 1), in place of the transposed convolutional layers [48]. We can see how this avoids the checkerboard patterns in Figure 16.

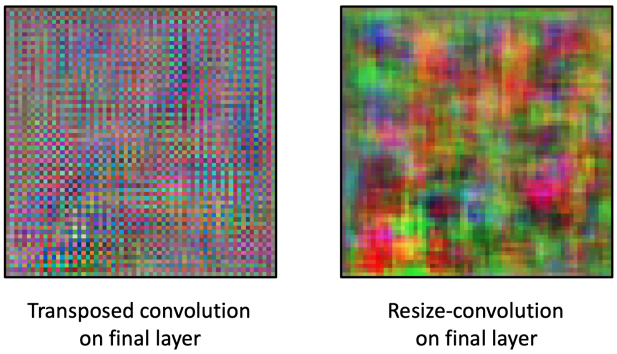


Figure 16: Left image shows the initial randomized output of a network with a transposed convolutional layer (stride=2, size=4×4) as the final layer. Right image shows the initial randomized output of a network with resize-convolution layers.

To the best of our knowledge, no network for the task of artifact removal uses resize-convolution. Thus, we test its effectiveness in Section 5.

5 Experimentation

The design of our final solution is based on many findings from our experimentation. In fact, the majority of time spent in this project was in the experimentation phase. We tested various aspects for our models, including different network architectures, loss functions, and training methods. We also conducted some preliminary tests to guide our experimentation focus, determining hyperparameters and feasibility of different methods. In this report, we omit these preliminary tests in favour of more interesting experiments. We outline our research methodology in ensuring fair tests in Section 8. Notably, we conduct repeated experiments for reliable results.

5.1 Dataset

For the training of our models, we use the RAISE dataset [49]. This consists of 8156 high-resolution RAW (uncompressed) images taken by the Nikon D40, D90, and D7000 cameras. Therefore, it consists of natural images. However, given our limited resources, we found that we could only feasibly handle the storage and processing of the reduced 1000 image dataset. This still gives the models many training examples to learn from since each image is partitioned into 256×256 sized images. Also, as we stored many preprocessed versions of the images, the storage requirement was very large, peaking at around 142.7GB[1] during our experimentation.

To gather the dataset, we wrote a script to download the uncompressed images, and then compressed them into single and double compressed versions. These were appropriately labelled and then stored to the cloud in Google Drive. Initially, we were only going to focus our research on one quality factor combination for single and double compression, so we used QF1=50 and QF2=90 to begin with. Therefore, many of our detector tests focus on this. However, due to good progress of the project, we decided to expand our experimentation to test various quality factors, which included $\text{QF} \in \{10, 30, 50, 70, 90\}$.

5.2 JPEG compression detection

For the task of JPEG compression detection, our main objective is to attain high accuracy while being mindful of the practicality and versatility of the model, meaning we wish for it to be small and to be applicable in a wide range of real-world scenarios. To achieve this, we train a CNN as used by state-of-the-art methods [14, 6]. The main ideas and advantages of CNNs are explained in Section 4.

In our experiments for the detector, 256×256 image blocks were taken from each of the 1000 images, resulting in a dataset of 639,192 examples. For the majority of experiments, we reserved 20% of the dataset for testing and a further 20% for validation, leaving 60% of the remaining data for training. However, for our later experiments, which involved testing five different QF

[1] In our code submission, some preprocessed versions of the images may be omitted since they are not used to train the final models, and so the size of the dataset may be less.

combinations (as opposed to one), each image contributed to 5× more data than before. Due to storage and performance constraints (notably with RAM), we were only able to use 20% of the images for training/validation in these tests. Still, this gives roughly the same number of training examples as before, and we also still use 20% of the data for testing (which gives even more testing examples than before).

5.2.1 Type of input

Our initial experiments test the effects of different types of input for the CNN. As discussed in Section 4, we can differentiate between single and double compression using histograms of DCT coefficients. However, since the state-of-the-art methods obtain the histograms using convolutional layers (in a predefined and handcrafted way, as proposed by Barni et al. [14]), we test whether we can input just the DCT coefficients to the network, so that it can learn how to obtain the histograms (or perhaps another useful representation) in training. We also test whether the same applies for using just the pixels as input. Note that we only use the Y-component of the YCbCr representation of the image (i.e., the grayscale image) to detect compression, which is consistent with all existing methods. This is because inclusion of all three channels does not result in any better performance, and so processing them is a waste of computing resources.

To test the different types of input, we use a preliminary CNN model, which closely follows Park et al.'s solution [6]. We simply swap out the first layer to be compatible with the different sizes of the different input types. Note that Park et al.'s solution [6] also has the compression quantization tables, taken from the JPEG file header, as input to the network. We do not require images to be stored in a JPEG format since this limits the applicability of the detector in real-world scenarios, as we mention in Section 3. Therefore, we omit this input for our solution. (Although, we still compare how quantization tables affect the accuracy of a model in our evaluation, in Section 7).

Model Input		Class test accuracy (%)			
Input type	Input size	None	Single	Double	**Overall**
DCT histogram features	64×120	98.7	93.8	99.8	97.4
DCT coefficients	256×256	98.4	85.1	95.5	93.0
Pixels	256×256	99.9	66.2	5.3	57.2

Table 1: Comparison of the different input types for the CNN, showing each model's input and test accuracy in predicting each class label. The model architectures are based on Park et al.'s solution [6] and trained on QF1=50 and QF2=90.

As we can see from Table 1, we confirm the consensus that detection of double compression is much easier to solve in the DCT domain than in the pixel domain. Working with just DCT coefficients could still obtain a relatively high overall accuracy of 93.0%. This shows that the model learned to extract some useful representation from the DCT coefficients to differentiate between the different types of compression. In contrast, the pixel input could only obtain a poor overall accuracy of 57.2%. Not only was the model completely unable to differentiate between single and double compressed images, but it even struggled to detect any kind of compression, often defaulting to a prediction of no compression.

Overall, the best accuracy was attained by the model with DCT coefficient histogram features as the input, with a 97.4% overall accuracy. This justifies the extra preprocessing step in obtaining the histograms rather than simply using DCT coefficients or pixels. Thus, we decide to use this for our proposed detector model.

5.2.2 Model architecture

One drawback of the preliminary models based on Park et al.'s solution [6] is that they are very large. In fact, they each require around 4GB to store. Therefore, we conduct an ablation study to determine whether this size is necessary. We do this by reducing the dimensions of the network layers and removing some layers. We consider three models: C_{Large} (which is most similar to the Park et al. model [6]), C_{Small}, and C_{Tiny} (which are both smaller versions of C_{Large}). Note that we now only consider the DCT histograms as input.

Model information				Class test accuracy (%)			
Model	Params.	Size (MB)	Inference time (ms)	None	Single	Double	**Overall**
C_{Large}	359,258,243	3,990	0.165	98.7	99.8	93.8	97.4
C_{Small}	4,841,603	58.8	0.132	98.9	99.9	92.7	97.1
C_{Tiny}	1,209,411	14.9	0.129	98.7	99.9	92.5	97.0

Table 2: Comparison of the different CNN architectures, showing each model's information and test accuracy in predicting each class label. The models are trained on QF1=50 and QF2=90.

As we can see in Table 2, a much smaller model is still able to achieve a very similar (albeit slightly worse) accuracy to the largest model. C_{Small} is 72× smaller than C_{Large} but only suffers a 0.3% drop in accuracy. Better yet, C_{Tiny} is nearly 4× smaller than C_{Small} but only is 0.1% less accurate. An additional benefit to C_{Tiny} is that its inference time is around 20% faster than C_{Large}. We view C_{Tiny}'s very slight loss in accuracy as a favourable trade-off for a massive storage size reduction and a slight improvement in speed. Thus, we use C_{Tiny}'s architecture for our proposed detector model.

5.2.3 Various input sizes

Because the network architecture uses dense layers, the input image size is fixed at 256×256. This would severely limit the applicability of our model in real-world scenarios. However, there is a simple solution for each case where the input image is smaller or larger.

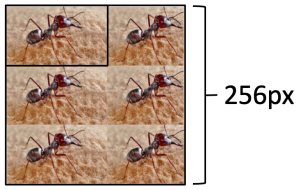


Figure 17: Padding an image by duplication to be 256×256.

Figure 18: Partitioning an image into 256×256 squares. This example shows $N = 3$. For this particular image, four 256×256 squares can be obtained, from which we randomly choose 3.

For images with a height or width of <256px, we simply pad the image by duplicating itself, which is demonstrated in Figure 17. We do this instead of just padding by zeros since image patches of zeros do not give a clear indication of whether an image is compressed. Therefore, we use the original image to retain the same compression information in the padded image. However, we must be careful to ensure that the 8×8 compression blocks align as with a normal 256×256 image. Therefore, we crop the original image so that its width and height are both multiples of 8 before padding. Note that this means that the minimum image width and height we permit is 8. The impact of this on the applicability of our system is negligible since it is highly improbable we will encounter an image smaller than this in the real-world context of our task.

For images with a height and width of >256px, we partition the image into 256×256 squares and at random choose up to N of these squares. We then input each of the squares to the CNN model and then take the modal result. Again, we want to align the squares such that the 8×8 compression blocks align as with a normal 256×256 image. We also do not want to choose two partitions that overlap, since then we repeatedly perform inference on the overlapped part of the image, which is wasted computation. Therefore, we start partitioning from the top-left corner and then separate each partition by 256px. This is illustrated in Figure 18.

Having multiple squares to derive results from has the benefit of increasing the confidence of the final result, but it also increases the time taken to derive it. We therefore conduct experiments to see what value of N gives the best trade-off. Our results show that this was $N = 21$. As we can see from Figure 19, the inference time remained relatively constant up until $N = 21$, due to the operations being effectively parallelized. Conveniently, this was also the point at which the final accuracy started to plateau, giving a maximum accuracy of 99.7%, which we can see from Figure 20. Notice how this is much more accurate than considering just one partition of an image. Images in the real-world are likely to have a height and width of >256px, and so our model is actually more accurate for these real-world images.

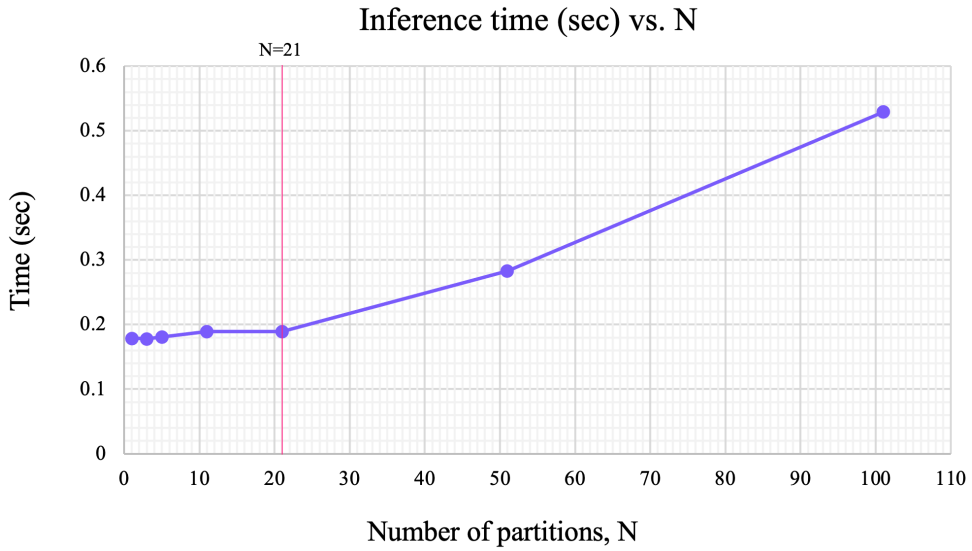


Figure 19: Time taken to obtain final prediction vs. the number of 256×256 squares, N. Using C_{Tiny} as before.

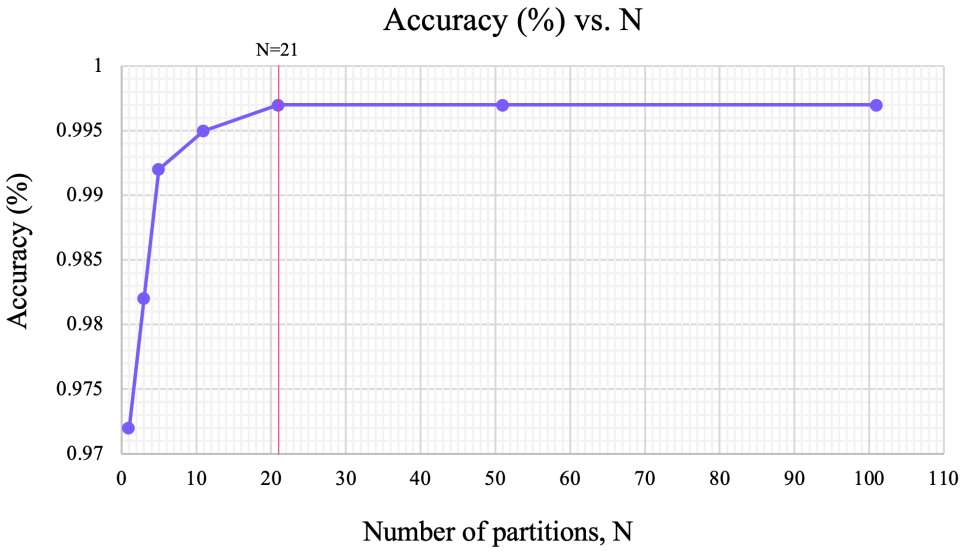


Figure 20: Test accuracy of the final prediction vs. the number of 256×256 squares, N. Using C_{Tiny} as before.

5.2.4 Various quality factors

In our previous tests, we only considered images that are single compressed with QF=50 and double compressed with QF1=50 and QF2=90. This was initially our sole focus, but due to good progress, we are able to train the detector with a wider variety of quality factors to test the model's generalizability. A generalized model is much more versatile than our previous one-QF models as it is more applicable for real-life scenarios. To do this, we use our favoured one-QF model from before (C_{Tiny}), but now train it with: uncompressed images, single compressed images with QF $\in \{10, 30, 50, 70, 90\}$, and double compressed images with QF1=50 and QF2 $\in \{10, 30, 50, 70, 90\}$. Note that this model is still not completely generalized since we only consider QF1=50 for double compressed images. We could not include other QFs for QF1 since this would result in storing more training examples, which we did not have the resources to do. We fix QF1=50, so then we cover cases when QF1 < QF2, QF1 = QF2, and QF1 > QF2.

	Class test accuracy (%)			
QF	None	Single	Double	**Overall**
10	98.5	35.0	67.8	67.1
30	98.5	82.0	95.3	91.9
50	98.5	63.8	56.8	73.0
70	98.5	97.3	99.7	98.5
90	98.5	98.5	99.8	98.9
Overall	98.5	75.3	83.9	**85.9**

Table 3: Test accuracy of the model trained on single compression QF $\in \{10, 30, 50, 70, 90\}$ and double compression QF1=50, QF2 $\in \{10, 30, 50, 70, 90\}$. Results are broken down into the test accuracy for each class label for each (final) QF applied.

As we can see from Table 3, we obtain a smaller overall accuracy of 85.9% compared to when we only considered one QF combination for single and double compression, where C_{Tiny} obtained 97.0% (Table 2), but this is to be expected. As explained in Section 4, double compression is very difficult to detect when QF1 = QF2 or QF1 >> QF2. Table 3 shows that this is the case for our model, since when QF2=10 (i.e., QF1 >> QF2), it is unable to accurately differentiate between single and double compressed images, resulting in only 67.1% overall accuracy for this QF. Likewise, when QF2=50 (i.e., QF1 = QF2), it only obtains 73.0% overall accuracy. Still, with the other cases, it is able to effectively generalize the differences of single and double compression, attaining overall accuracies of at least 91.9%. In fact, for the case QF2=90, which we previously considered in isolation for our one-QF (C_{Tiny}) model, the generalized model obtains 98.9% overall accuracy, improving upon the one-QF model (97.0%, from Table 2). Perhaps the generalized model is able to learn more about the characteristics of single and double compressed images by considering a range of QFs, and thus becomes more accurate for specific QFs. Moreover, the

ability of the model to detect any kind of compression is only slightly less than the one-QF model, attaining 98.5% accuracy for uncompressed images (compared to 98.7%, from Table 2).

5.3 JPEG artifact removal

In the task of JPEG artifact removal, our main aim is to develop a model that produces visually pleasing images while keeping the model small to maximize resource efficiency. We also wish to consider the differences between single and double compressed images. To achieve this, we utilize a GAN, which is also used by current state-of-the-art methods [26, 31, 7].

To evaluate our model's performance, we need a way to measure the "visual appeal" of the generated images. We can do this qualitatively, by visually examining the image, or we can do it quantitatively, by comparing the similarity of the generated image and the corresponding uncompressed image (the ground truth). A widely used metric to measure the similarity of compressed images with its uncompressed version is Peak Signal-to-Noise Ratio (PSNR), which is defined as $\text{PSNR}(x, y) = 10 \cdot \log_{10}(\frac{\text{MAX}(y)^2}{\text{MSE(x,y)}})$, where $\text{MAX}(y)$ gives the maximum possible pixel value of the ground truth image, y. Note that PSNR is measured in decibels (dB). Using PSNR to compare the generated image to the uncompressed image gives us an objective measurement as to how effective the model is at removing the JPEG artifacts.

As opposed to our detector, which uses grayscale images, our restorer works on RGB images. This requires more storage space due to the increased colour information. To manage this, we limited our dataset to only one 256×256 partition from each image. We briefly tested training our models with more partitions, but we did not observe any significant improvements in results. Moreover, due to the increased RAM requirements, using more partitions required the use of Google Colab's Premium GPUs, which are more expensive per hour. Thus, we opted not to continue using the Premium GPUs and instead used standard GPUs, so that we could conduct more experiments, which we felt would be more beneficial than training on additional data.

Throughout our experimentation, we used an 80-10-10 split for training, validation, and testing data, respectively. For qualitative testing, we used QF=10 to compress the images since the differences are more obvious and thus the results are more easily evaluated. We mostly used QF=30 for quantitative testing, but we also conducted rudimentary tests with other QFs to ensure the generalizability of the results. Where trends were identical, the results may be omitted for sake of brevity and clarity.

5.3.1 Perceptual loss

It is not clear in literature which perceptual loss function is best for our task. Thus, we compare different combinations of MSE, MAE, MS-SSIM-based loss, and VGG-19-based loss, which are explained in Section 4. To do this, we use a preliminary GAN model, whose architecture closely follows Zhao et al.'s solution [7], although our models consider coloured images as opposed to grayscale images.

Perceptual loss	PSNR (dB)
MSE	30.7
MAE	31.9
MAE, MS-SSIM	32.0
MAE, VGG-19	31.7
MAE, MS-SSIM, VGG-19	32.3

Table 4: Average PSNR score of the generated images compared to their corresponding uncompressed images for models trained with different loss functions. Note that these preliminary models are based on Zhao et al.'s solution [7] and trained on single compressed images with QF=30.

As we can see from Table 4, using MSE as the loss resulted in poor performance compared to MAE, giving a 1.2dB lower PSNR score. Therefore, we only considered losses that included MAE in our remaining tests. The results show that a combination of MAE, MS-SSIM-based loss, and VGG-19-based loss gave the best PSNR of 32.3dB. Therefore, we use this as our loss function for our proposed model, and to the best of our knowledge, this combination has not been used before. Our proposed perceptual loss function can be described as:

$$L(x, y) = \alpha \cdot \text{MAE}(x, y) + \beta \cdot L_{\text{MS-SSIM}}(x, y) + \gamma \cdot L_{\text{VGG}}(x, y)$$

where α, β, and γ are hyperparameters to weigh the importance of each term.

5.3.2 Patch-based training

An approach suggested by Mameli et al. [26] is to train the model initially using 64×64 patches instead of full images as a way to focus on high frequency details. The model is then later trained on full images. However, as shown by Table 5, this did not result in any improvements for our model, and in fact did slightly worse.

Image patch	PSNR (dB)
256×256 only	32.3
64×64, 256×256	32.2

Table 5: Average PSNR score of the generated images compared to their corresponding uncompressed images for models trained with different image patches. Note that models are trained on single compressed images with QF=30.

We suggest the lack of improvement is due to our use of a PatchGAN discriminator, which is explained in Section 4. Our version of PatchGAN already only considers 70×70 patches of the images and thus there is no need to restrict the input image size. Doing so only complicates the training process, and so we opt to not use this method.

5.3.3 NoGAN training

Another training method suggested by Mameli et al. [26] is "NoGAN" training. This method was first introduced for the task of image colourization [27], but also proved useful for JPEG artifact removal [26]. NoGAN training is where the generator and discriminator are initially trained separately, meaning the generator does not use the discriminator's gradients in backpropagation. Then, towards the end of the training, the generator and discriminator learn simultaneously, as in a normal GAN. To determine the effectiveness of this method, we compare a traditionally trained GAN with a NoGAN-trained GAN (postGAN). We also include the results of the NoGAN-trained GAN before training with the discriminator (preGAN), which is essentially a stand-alone generative CNN.

Training method	PSNR (dB)
GAN	32.3
NoGAN (preGAN)	32.2
NoGAN (postGAN)	33.0

Table 6: Average PSNR score of the generated images compared to their corresponding uncompressed images for models trained with different methods. GAN is a model trained the normal way, preGAN is a model using noGAN training *before* incorporating the discriminator, and postGAN is a model using noGAN training *after* incorporating the discriminator. Note that models are trained on single compressed images with QF=30.

From Table 6, we can see that, interestingly, the traditionally trained GAN only performs marginally better than preGAN. However, postGAN was able to achieve a noticeable improvement, gaining 0.7dB in PSNR compared to the normal GAN. Hence, we adopt the NoGAN training method for our proposed model.

Additionally, by comparing the generated images of preGAN and postGAN, we are able to observe the effect of training with and without a discriminator, showing how the GAN loss helps the network to learn to produce photorealistic images. A comparison is shown in Figure 21. As we can see, both preGAN and postGAN are effective in removing the blocking artifacts of the compressed images. However, preGAN's output looks slightly too smooth compared to the ground truth images. PostGAN is able to rectify this by introducing some artificial noise to the images that seems to emulate the noise in the ground truth images. This shows how the GAN loss is guiding the model to better reconstruct high frequency details as the discriminator picks up on "smoothness" as an indicator that an image is fake. This is explained in more detail in Section 4.

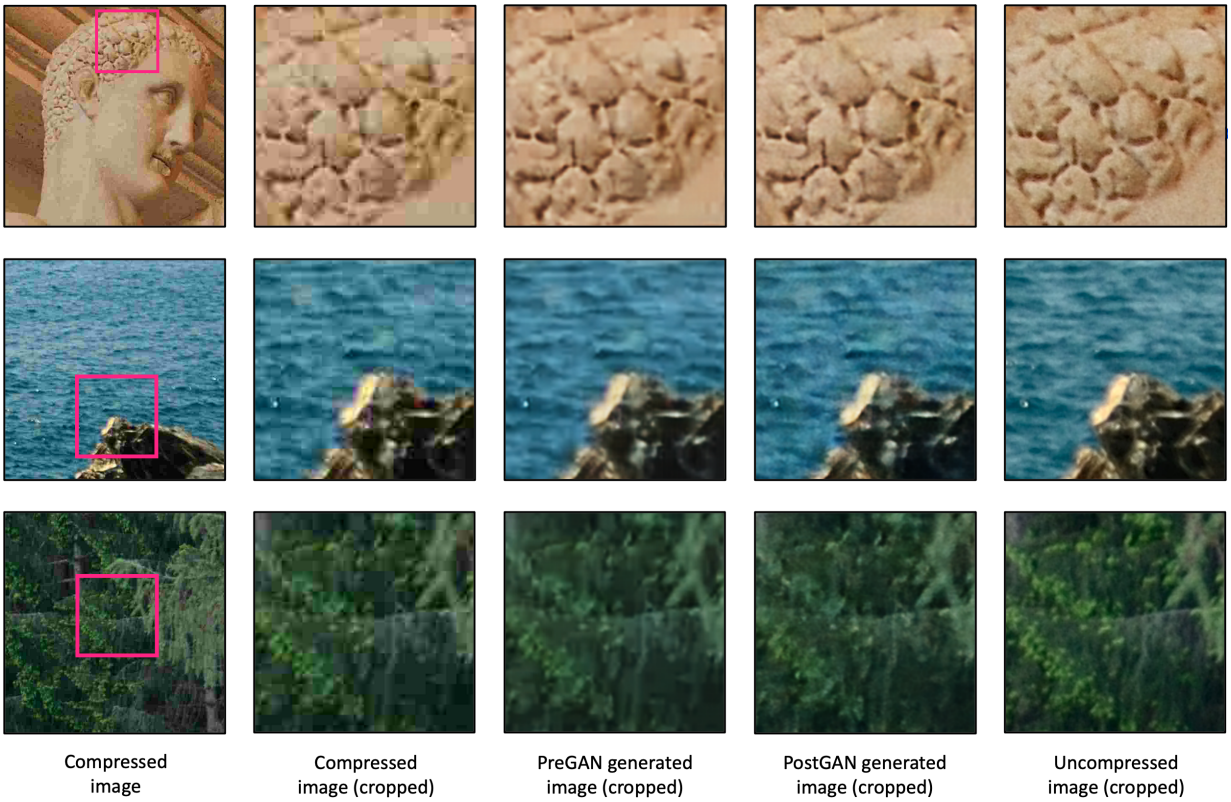


Figure 21: Qualitative results to compare the generated images of preGAN and postGAN. Models are trained with single compressed images with QF=10. (Zoom in to inspect the differences).

5.3.4 Resize-convolution layers

As explained in Section 4, transposed convolutional layers, which are used by our preliminary model, may generate images with checkerboard artifacts. One solution is to use resize-convolution layers instead [48]. As far as we are aware, no current methods for artifact removal use resize-convolution, so we test its effectiveness compared to the more conventional transposed convolution.

Decoder layers	PSNR (dB)
Transposed convolution	33.0
Resize-convolution	32.1

Table 7: Average PSNR score of the generated images compared to their corresponding uncompressed images for models trained with either transposed convolutional layers or resize-convolutional layers. Note that models are trained on single compressed images with QF=30.

As we can see from our quantitative results in Table 7, resize-convolution did not give a better PSNR. We can also see from our qualitative results in Figure 22 that checkerboard patterns are equally unnoticeable in both. This shows that the GAN's own ability in learning how to avoid checkerboard artifacts is sufficient to avoid this phenomena in our task. This could possibly be due

to the discriminator recognizing the checkerboard pattern as an indication of a fake image. The resize-convolution layers also seem not to permit the additional artificial noise in areas of solid colour. This means that there are still some smoothness in the generated images, reducing their photorealism. Since resize-convolution did not improve upon either qualitative or quantitative results, we opt not to use it for our final model.

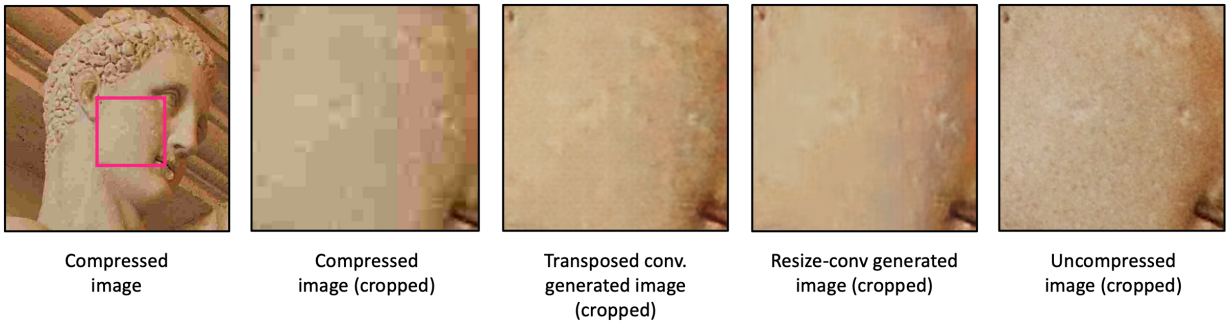


Figure 22: Qualitative results to compare the generated images of models using either transposed convolution layers or resize-convolution. Models are trained with single compressed images with QF=10. (Zoom in to inspect the differences).

5.3.5 Generator architecture

Similar to our detector model, we want to keep the architecture of the network as small as possible while retaining good results. Since only the generator network is used for the final system, we only consider different architectures for the generator, using the same PatchGAN discriminator to train each model. To alter the size of the generator, we simply vary the number of downsampling and upsampling layers. We consider four models: $G_{Pix2Pix}$ (which uses the same architecture as Pix2Pix [25]), G_{Zhao} (which uses the same architecture as Zhao et al. [7])[2], G_{Small}, and G_{Tiny} (both of which are smaller versions of G_{Zhao}).

Generator information				Test
Model	Parameters	Storage size (MB)	Generation time (ms)	PSNR (dB)
$G_{Pix2Pix}$	54,425,859	220	5.15	32.5
G_{Zhao}	6,173,635	25.5	3.71	33.0
G_{Small}	1,451,971	6.4	2.65	32.9
G_{Tiny}	270,787	1.5	1.60	31.6

Table 8: Comparison of the different generator architectures, showing each model's information and PSNR score. The models are trained on QF=30.

Interestingly, we can see from Table 8 that the largest model, $G_{Pix2Pix}$, did not obtain the

[2]Note that $G_{Pix2Pix}$ and G_{Zhao} are still overall different models to the original Zhao et al. model [7] and Pix2Pix model [25], respectively, since we use our novel perceptual loss function and the NoGAN training method.

greatest PSNR score. This was obtained by G_{Zhao}, which achieved 33.0dB, even though it is nearly 10× smaller. However, we could obtain a very similar (albeit slightly lower) score of 32.9dB using G_{Small}, which is more than 4× smaller than G_{Zhao}. Due to G_{Small}'s smaller size, it also generates images slightly faster. We consider these benefits to be favourable trade-offs for the practicality of our model, and so we choose G_{Small} for our final generator model. We decided not to use a smaller model since we observe that doing this significantly impacts the effectiveness of the restoration as G_{Tiny} could only obtain a PSNR of 31.6dB.

5.3.6 Single vs. double compression

Our previous experiments only show results for models trained on single compressed images. Thus, we now consider double compressed images, and we test our hypothesis that double compressed images require a more powerful restoration model compared to single compressed images. We in fact show that this is not the case since the same model as before performs equivalently well for double compressed images. To show this, we trained three separate models, with one trained on just single compressed images, one on just double compressed images, and one on both single and double compressed images. We then test each of the models on these different subsets of data.

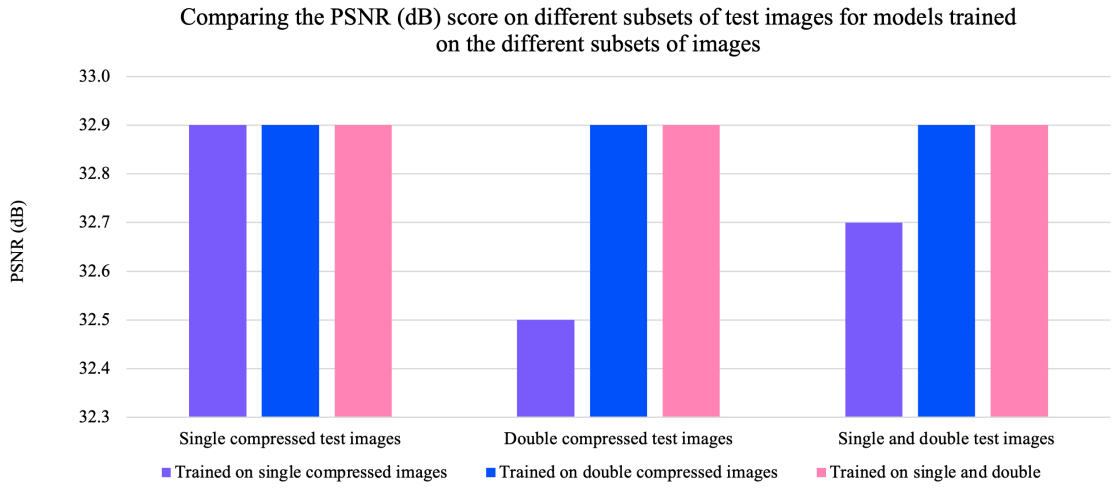


Figure 23: Bar chart showing the PSNR score of three models trained on different subsets of the dataset, tested on each dataset subset. Images are single compressed with QF=30 and double compressed with QF1=50 and QF2=30.

From Figure 23, we can see that the model trained solely on single compressed images did not perform well for double compressed images. However, the model trained solely on double compressed images was able to generalize and performed similarly on single compressed images compared to the model trained solely on single compressed images. Importantly, the model trained on both single and double compressed images was able to equal the best performance of the other models. So, rather than developing two different models for single and double

compression, it is clearly more logical to train one model for both. Therefore, to handle single and double compressed images, our final solution is a model trained on both.

5.3.7 Various quality factors

The previous models use the same effective[3] QF to train a single model. Therefore, we now test whether a single model can handle compressed images of various QFs. We do this by comparing a generalized model (trained with QF=30, QF=50, and QF=70), with a model trained on only QF=30, only QF=50, or only QF=70, depending on the subset of data being tested. This is shown in Figure 24.

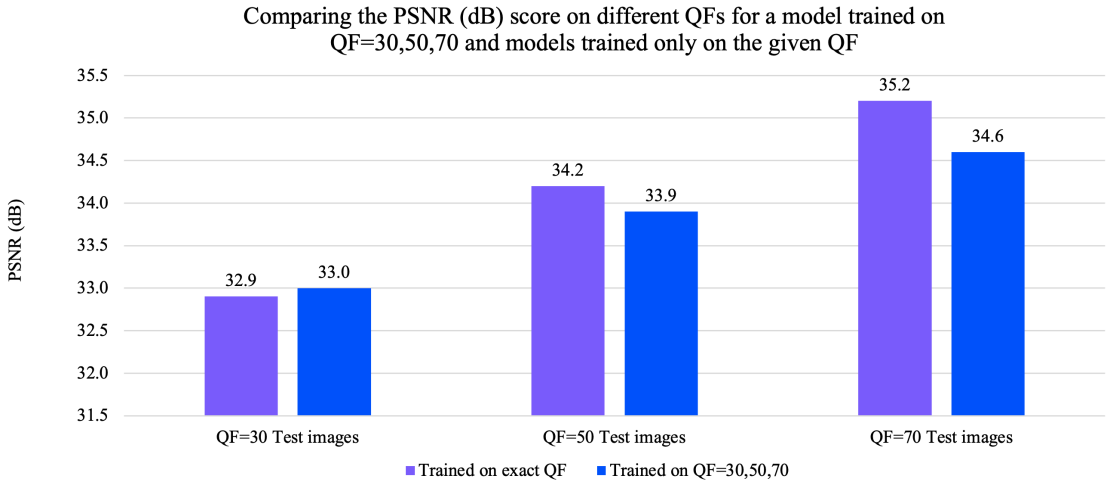


Figure 24: Bar chart showing the PSNR score of a model trained with QF= 30, 50, 70 compared to models trained on a specific QF.

Generally, images compressed with a particular QF are more effectively restored with a model trained on that specific QF than with a model trained on multiple QFs. The only case where this is not true is QF=30, where the general model performs slightly better, but the difference is very small. This has exposed a notable limitation in our approach, since, to effectively account for multiple QFs, we must use different models. However, we have no way to detect the compression QF beforehand in order to invoke the appropriate model. Due to our initial focus of scope and lack of additional time, we were unable to experiment with solutions for this. We continue discussing our model's limitations in Section 7.

[3]By "same effective" QF, we mean that the QF used for single compression is the same as the lowest QF used in double compression.

6 Implementation

Incorporating our domain knowledge (discussed in Section 4) and experimentation results (discussed in Section 5), we have valuable insights into the most effective approaches for our detector and restorer models. Drawing upon this, we propose our final solution. Furthermore, we discuss the implementation of our website, which allows users to apply the models on their own images.

6.1 Detector

Our detector is a CNN, which takes as input an image's DCT coefficient histogram features, obtained through Barni et al.'s method [14]. As determined by our experimentation in Section 5, our detector model uses C_{Tiny}'s architecture, which is shown in detail in Figure 25. This architecture uses ReLU activation throughout, except in the final output layer, which uses a Softmax activation. There is also batch normalization between each of the layers. To train the model, we use a categorical cross-entropy loss with L2-regularization and an Adam optimizer. The benefits of our network configurations are explained in Section 4 and demonstrated in Section 5. We train the detector model using images that are compressed with a variety of QFs (single QF $\in \{10, 30, 50, 70, 90\}$ and double QF $\in \{(50, 10), (50, 30), (50, 50), (50, 70), (50, 90)\}$), and thus the model can generalize for many QFs.

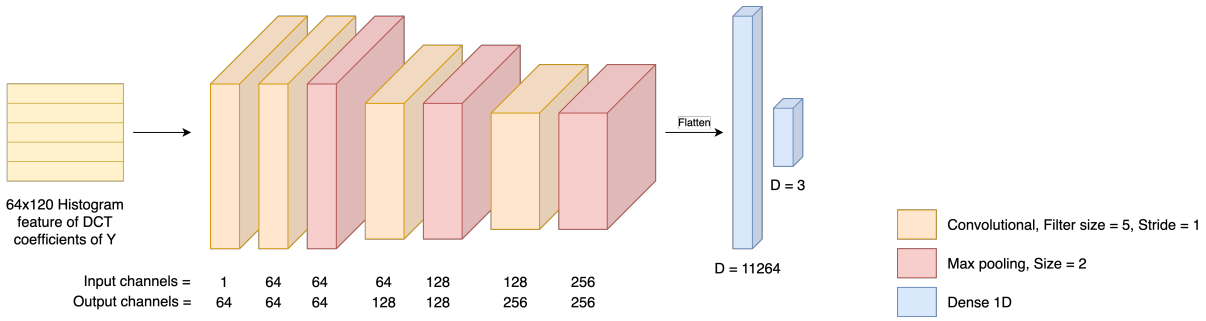


Figure 25: Proposed C_{Tiny} architecture.

6.2 Restorer

Our restorer is a GAN, which takes as input an RGB compressed image to generate a restored version of the image. Through our experimentation in Section 5, we discovered that we are able to use the same model to handle both single and double compression. The generator uses G_{Small}'s architecture, which is shown in detail in Figure 26. This architecture uses leaky ReLU activation for downsampling layers and the last upsampling layer, with ReLU activation for remaining upsampling layers. The output uses tanh activation. There is batch normalization between each of the layers, except for the first downsampling layer. We also utilize a 70×70 PatchGAN discriminator [25], which is explained in Section 4, but we do not bundle it in our final solution as it is only used to train the generator.

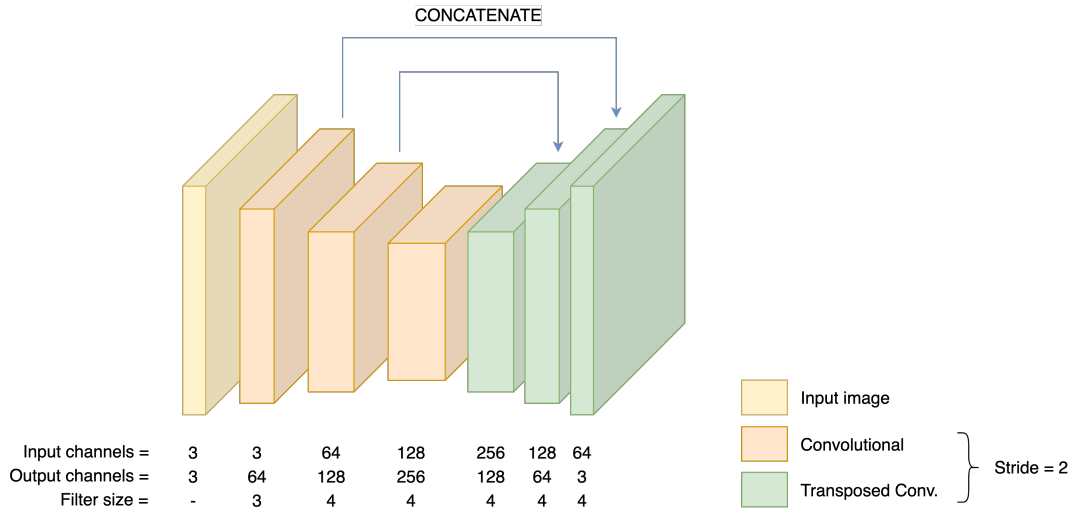


Figure 26: Proposed G_{Small} architecture.

To train the GAN, we employ NoGAN training and use a novel perceptual loss function. Equation 7 shows the final combined loss function for the generator, which incorporates cGAN loss, MAE, MS-SSIM, and intermediate activation of the VGG-19 network. We minimize the generator and discriminator losses using Adam. The benefits of our network configurations are explained in Section 4 and demonstrated in Section 5. A limitation of our approach is that images compressed with a particular QF are restored more effectively with a model trained for that specific QF. However, in our final solution, we only invoke a model trained on QF=10 since we have no method for detecting an image's compression QF (and this is the QF used for qualitative testing). For experimentation purposes, we also include alternative models.

$$
\begin{aligned}
L_G = & \\
& \log(1 - D(x, G(x, z))) \\
& + \alpha \cdot \frac{1}{mnc} \sum_{i=1}^{m} \sum_{j=1}^{n} ||y_{i,j} - G(x)_{i,j}||_1 \\
& + \beta \cdot (1 - \text{MS-SSIM}(y, G(x))) \\
& + \gamma \cdot \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} ||\mathcal{V}(y)_{i,j} - \mathcal{V}(G(x))_{i,j}||_2^2
\end{aligned}
\tag{7}
$$

6.3 Final solution

Our final solution combines the detector and restorer models so that only compressed images are subject to restoration. This is done simply with a decision step between the detector and restorer, where the restorer is only invoked if any type of compression has been detected. Also, since the detector only takes DCT coefficient histogram features as input, we must also have a preprocessing step for input images. This includes padding/partitioning the image as described in Section 5.2.3, and then obtaining the DCT histograms from the (padded/partitioned) image(s)

using Barni et al.'s method [14]. There is less preprocessing required for the restorer model since we input the image in its original RGB representation. However, we still need to pad the image to make the height and width multiples of 16, so that the dimensions are compatible with the downsampling layers. The generated image is then cropped to its original size.

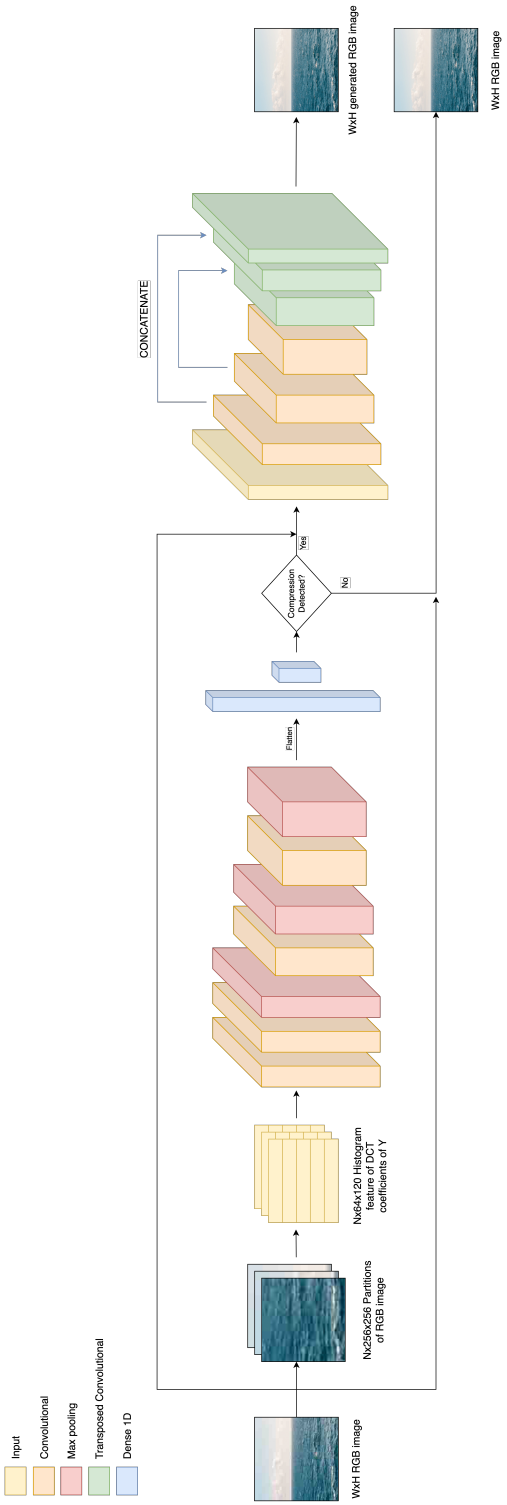


Figure 27: Proposed final solution. See Figure 25 for more details on the detector, and Figure 26 for more details on the restorer.

6.4 Web application

To facilitate user access to our final solution, we develop a user-friendly web application, where users can upload an image to detect whether it has been JPEG compressed and, if so, apply a JPEG artifact removal process. The organization of the website, which can be broken down into frontend and backend components, is summarized in Figure 28.

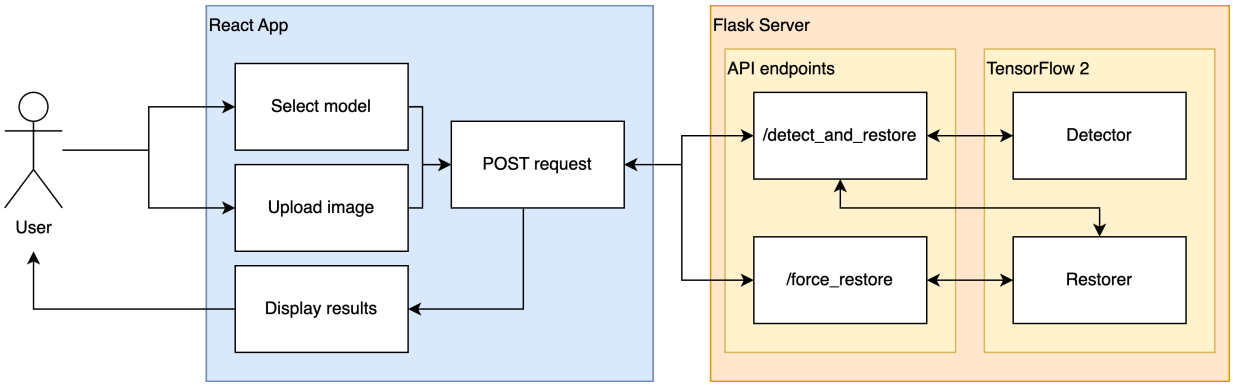


Figure 28: Overview of the website component interactions. The frontend is a React app and the backend is a Flask server.

We build the frontend with React.js to create a single-page application (SPA). SPAs are web applications that dynamically update the content of a single web page in response to a user event or when data changes. The content changes without requiring the browser to reload the entire page as different components of the page are reloaded separately. This reduces the interaction latency, which improves the user experience since the website is as fluid as a native application. The design of the SPA is shown in Figure 29.

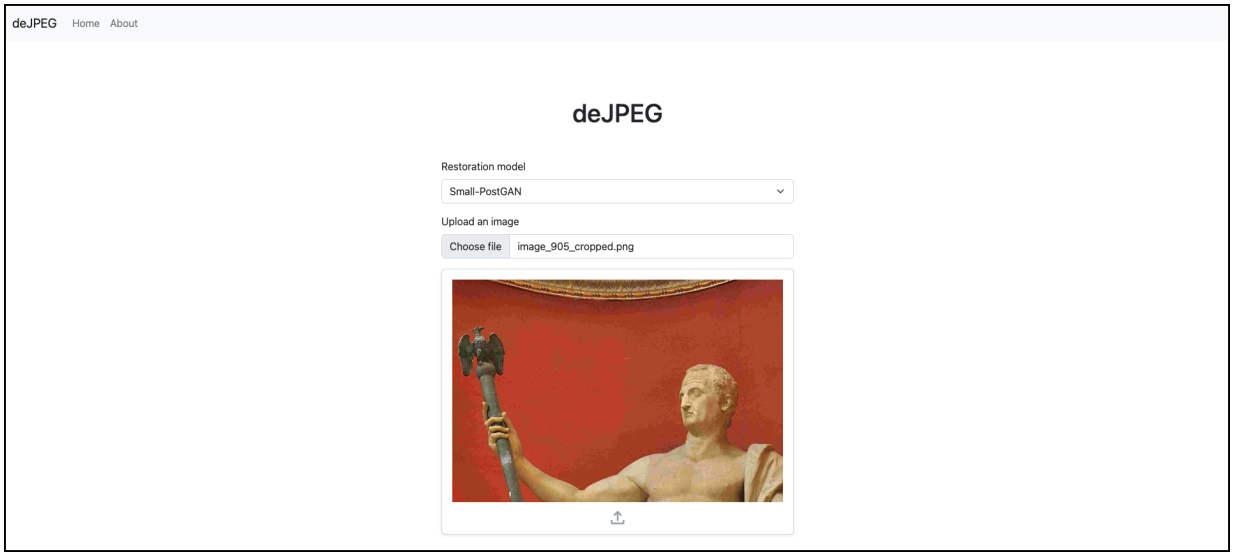


Figure 29: Uploading an image for detection and restoration.

To interact with the detector and restorer models, the React app makes HTTP requests to REST API endpoints, which are hosted with a Flask server. We use Flask since then we can natively run the models, which are written in Python. We only implement two API endpoints. One endpoint is used to first detect if an image is compressed and, if so, returns the restored version of the image. For experimentation purposes, it also allows users to specify different restorer models. Its definition is shown in Figure 30. The other endpoint returns the restored version of the image without checking for compression beforehand since we allow users to "force restoration". Its definition is similar to Figure 30, but it does not return any compression information. Both endpoints only allow POST requests so that the image file can be included in the request payload.

```yaml
/detect_and_restore:
  post:
    summary: Restore an image using a specified model if detection is detected
    requestBody:
      required: true
      content:
        multipart/form-data:
          schema:
            type: object
            properties:
              model:
                description: The name of the machine learning model to use for restoration
                type: string
              img:
                description: The image file to restore
                type: string
                format: binary
    responses:
      '200':
        description: Successful detection and restoration
        content:
          application/json:
            schema:
              type: object
              properties:
                compressionAllPredictions:
                  description: All compression predictions for the image
                  type: array
                  items:
                    $ref: '#/definitions/allPredictions'
                compressionPredictions:
                  description: Average (final) compression predictions for the image
                  type: array
                  items:
                    $ref: '#/definitions/predictions'
                compressionLevel:
                  description: Average (final) compression level prediction for the image
                  type: number
                imagePath:
                  description: URL of the restored image. Empty if no compression detected.
                  type: string
                detectionTime:
                  description: Time elapsed for image detection
                  type: number
                restorationTime:
                  description: Time elapsed for image restoration
                  type: number
```

Figure 30: "/detect_and_restore" API endpoint definition.

When the React app receives the response from the server, which includes a URL to the restored image, the restored image is downloaded and shown to the user. We use a custom React component [50] that allows the user to compare the restored image to the original uploaded image using a slider. This is shown in Figure 31. There are also buttons that allow users to download the image or view the image in full-screen. In addition, users can expand to view more information on the model results. Here, they can view the elapsed time and the probability predicted for each class label for every 256×256 partition of the image, which is shown in Figure 32.

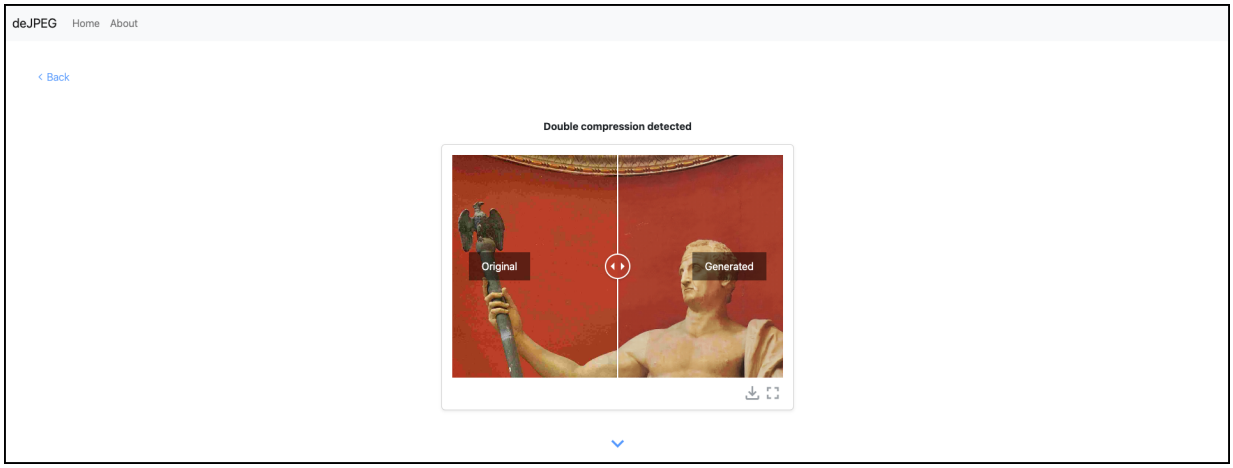


Figure 31: Viewing the restored image of a double compressed image. The slider can be moved to view more/less of the original or generated image.

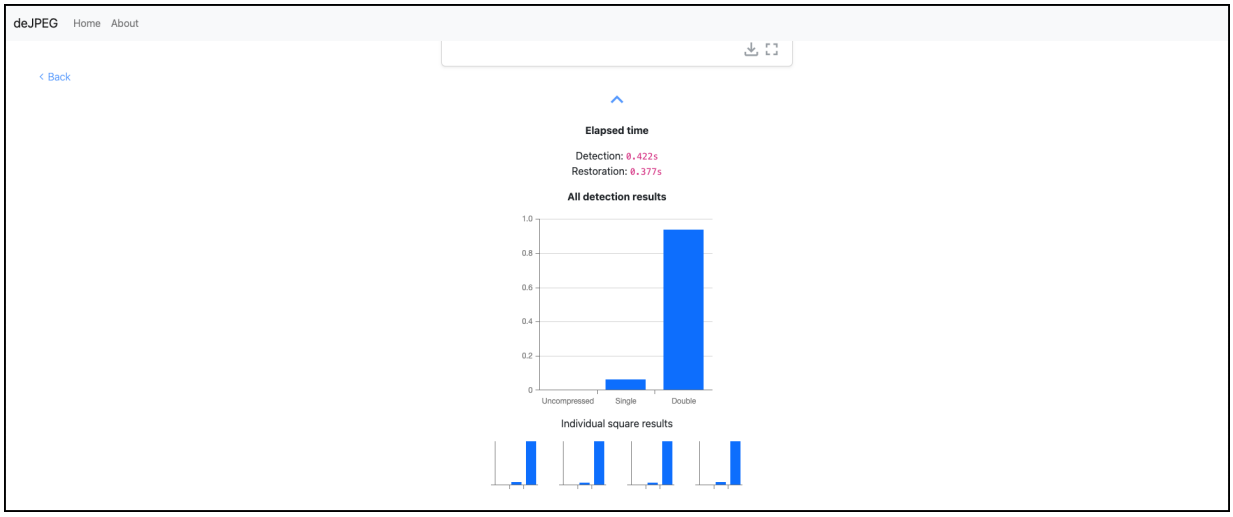


Figure 32: Viewing detailed results. The large bar chart shows the final (average) detector predictions, and the smaller bar charts represent the individual predictions for each image partition.

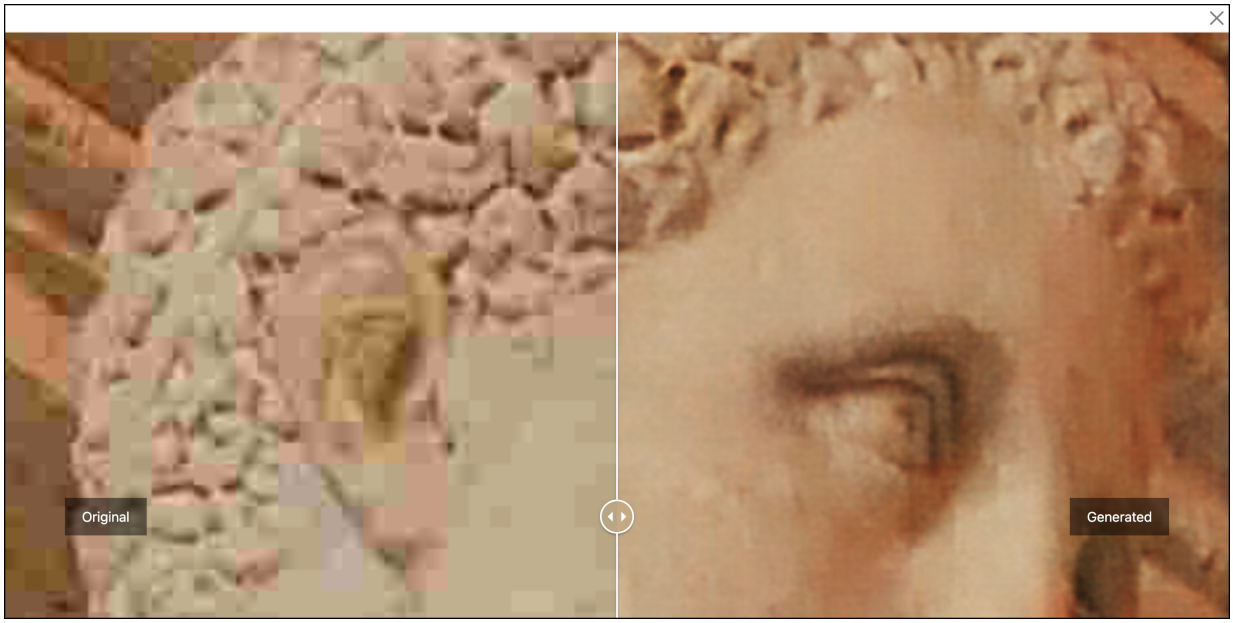


Figure 33: Viewing a restored image (of a statue head) in full-screen.

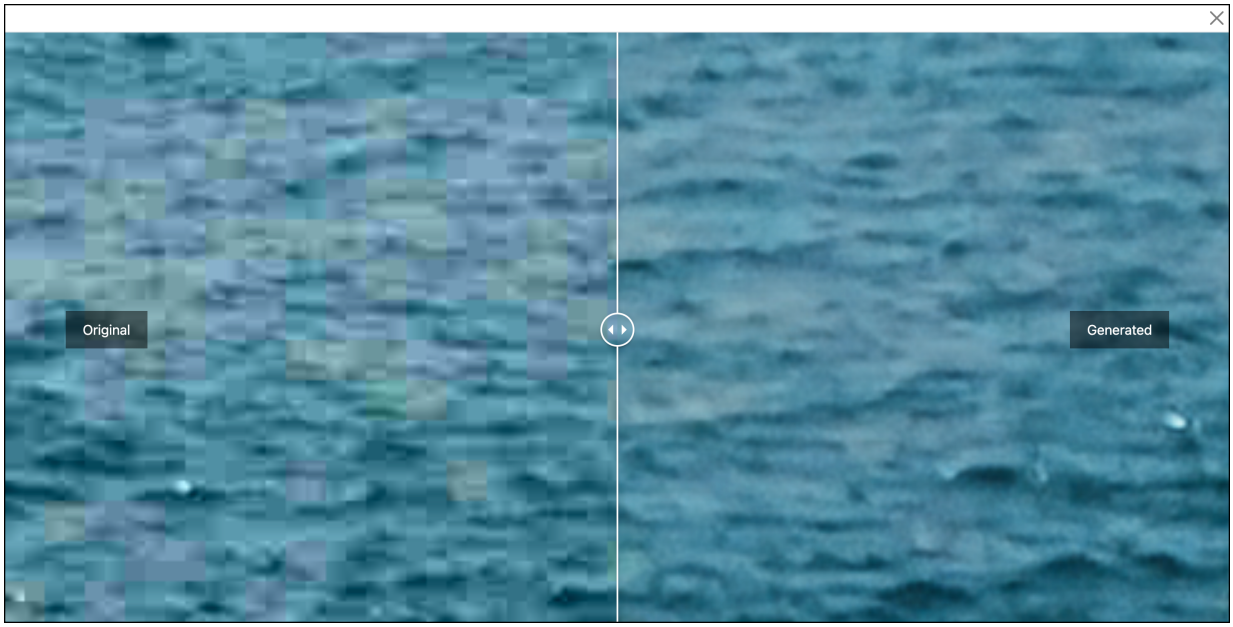


Figure 34: Viewing a restored image (of the ocean) in full-screen.

7 Evaluation

Reflecting upon our system requirements, as outlined in Section 3, it is clear to see we have fulfilled objectives **M1** and **M4**, as we constructed a dataset to train the models, and we implemented a website where users can use the models. We also believe we have satisfied **S1** as we made the user experience of the website as smooth as possible, such as by implementing a SPA. We opt not to do any user testing to formally verify this since the application is not complex enough to necessitate this (and this was not our primary focus). It is also clear that we have partially fulfilled objectives **M2** and **M3** since we have implemented a detector and restorer that take as input information based solely on pixel data. Thus, it remains to be seen that we have fully satisfied **M2** and **M3**, where we require the detector to be "accurate", and the restorer to be "convincing". We show that we have achieved this by comparing our models to some recent existing solutions, which, upon the time they were published, claimed to be state-of-the-art. We did not complete our **COULD** objective, **C1**, of implementing a web extension, as we felt there was more value in spending our time further improving the models rather than implementing another use case for them.

When comparing our models to existing solutions, we attempt to recreate the existing models as true to the original as possible according to the available documentation, our understanding, and the resources available to us. However, there may be some modifications that need to be made, and these are explicitly stated. Additionally, we train all models using our own dataset, rather than their original datasets, to enable comparability of the results. We further outline our research methodology in ensuring fair tests in Section 8.

7.1 JPEG compression detection (M2)

We based our proposed detector model on Park et al.'s solution for double JPEG compression detection [6], although we modified their model so that it can additionally detect no JPEG compression, and we removed the extra quantization table input. We also reduced the size of the network to be more portable. While these changes seem as though they would impair the performance of the model, we show that its accuracy is in fact not drastically affected. We do this by comparing our proposed model to Park et al.'s original model, as outlined in their paper[4] [6]. We train both models using images compressed with a variety of QFs (single QF $\in \{10, 30, 50, 70, 90\}$ and double QF $\in \{(50, 10), (50, 30), (50, 50), (50, 70), (50, 90)\}$) to test their general accuracy. This is shown in Table 9.

The results show that our model is only slightly less accurate than Park's model, despite our model being much simpler, as we achieve an overall accuracy that is 0.3% below Park's overall accuracy. It seems that the main advantage of Park's model is its detection of uncompressed images, where it obtains 100% accuracy. This is because the model can simply memorize the

[4]The model is slightly different to Park et al.'s model since we train it to additionally detect no JPEG compression to fit our specific task. This involves training with uncompressed images, which do not have quantization tables. To make these images compatible with the network, we simply use a matrix of 1s as the compression quantization table.

quantization table corresponding to uncompressed images. Another case where Park's model is more accurate is QF2 = 30 (i.e., QF1 > QF2), as it achieves 93.3% accuracy compared to 91.9% given by our model. However, there is no significant improvement for the remaining cases, with the overall accuracy of our model and Park's model being within 1% for each remaining case. Also, while our model is better at predicting double compressed images, and Park's is better for single compressed images, the difference in accuracy is not noteworthy as this information only provides insight into which prediction the models default to when they are uncertain between single and double compression.

		Class test accuracy (%)			
Model	QF	None	Single	Double	**Overall**
Ours	10	98.5	35.0	67.8	67.1
	30	98.5	82.0	95.3	91.9
	50	98.5	63.8	56.8	73.0
	70	98.5	97.3	99.7	98.5
	90	98.5	98.5	99.8	98.9
	Overall	98.5	75.3	83.9	**85.9**
Park's [6]	10	100	49.7	52.5	67.4
	30	100	92.0	88.0	93.3
	50	100	69.3	47.5	72.2
	70	100	98.1	98.5	98.9
	90	100	99.4	97.4	98.9
	Overall	100	81.7	76.8	**86.2**

Table 9: Comparison of the test accuracy of our model and Park et al.'s model [6]. They are trained on single compression QF $\in \{10, 30, 50, 70, 90\}$ and double compression QF1=50, QF2 $\in \{10, 30, 50, 70, 90\}$. Results are broken down into test accuracy of each class label for each QF.

We believe that these results show that the large size and quantization table requirement of Park's model are not necessary for an accurate model, and we favour our model's improved portability and applicability for a wider range of scenarios.

7.2 JPEG artifact removal (M3)

We based our restorer model's architecture on Zhao et al.'s solution for JPEG artifact removal [7], but we made the model smaller. We also introduced NoGAN training and a novel perceptual loss function, resulting in a model with improved performance, despite its smaller architecture.

We can see this in our quantitative testing, which is shown in Figure 35, where we compare our proposed model with Zhao et al.'s original model[5] [7]. Both models are trained using three distinct subsets of our dataset to cover the cases when QF1 < QF2, QF1 = QF2, and QF1 > QF2. In every case, our model is able to achieve a PSNR score that is greater by around 2dB, showing the effectiveness of NoGAN training and our perceptual loss function.

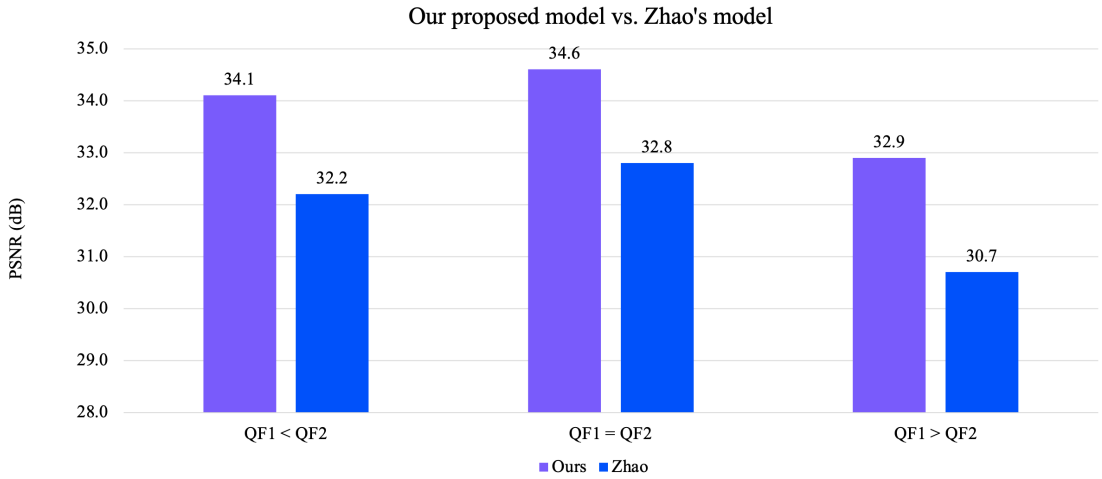


Figure 35: Comparison of the PSNR (on the generated images and the corresponding uncompressed images) of our model and Zhao et al.'s model. The graph shows the models trained on three different subsets of data. QF1 < QF2 corresponds to single QF=50 and double QF1=50, QF2=70. QF1 = QF2 corresponds to single QF=50 and double QF1=50, QF2=50. QF1 > QF2 corresponds to single QF=30 and double QF1=50, QF2=30.

In addition, we conduct some qualitative testing to verify that the improved PSNR score translates to more visually pleasing images. From Figure 36, we can see that both models can convincingly remove the JPEG blocking artifacts. The resulting images also appear photorealistic, with the models introducing artificial image noise that emulates the noise of natural images. However, Zhao's model seems to be excessive in its added noise, resulting in grainy images compared to the original uncompressed ones. On the other hand, our model's added noise is not as strong as the uncompressed images, but we prefer the appearance of this. Zhao's model also tends to over-saturate the colour of the images. Perhaps this is because it was originally developed with grayscale images, unlike our model, which is more colour accurate.

[5]The model we use to compare is slightly different as it considers RGB images rather than grayscale images as in the original paper [7].

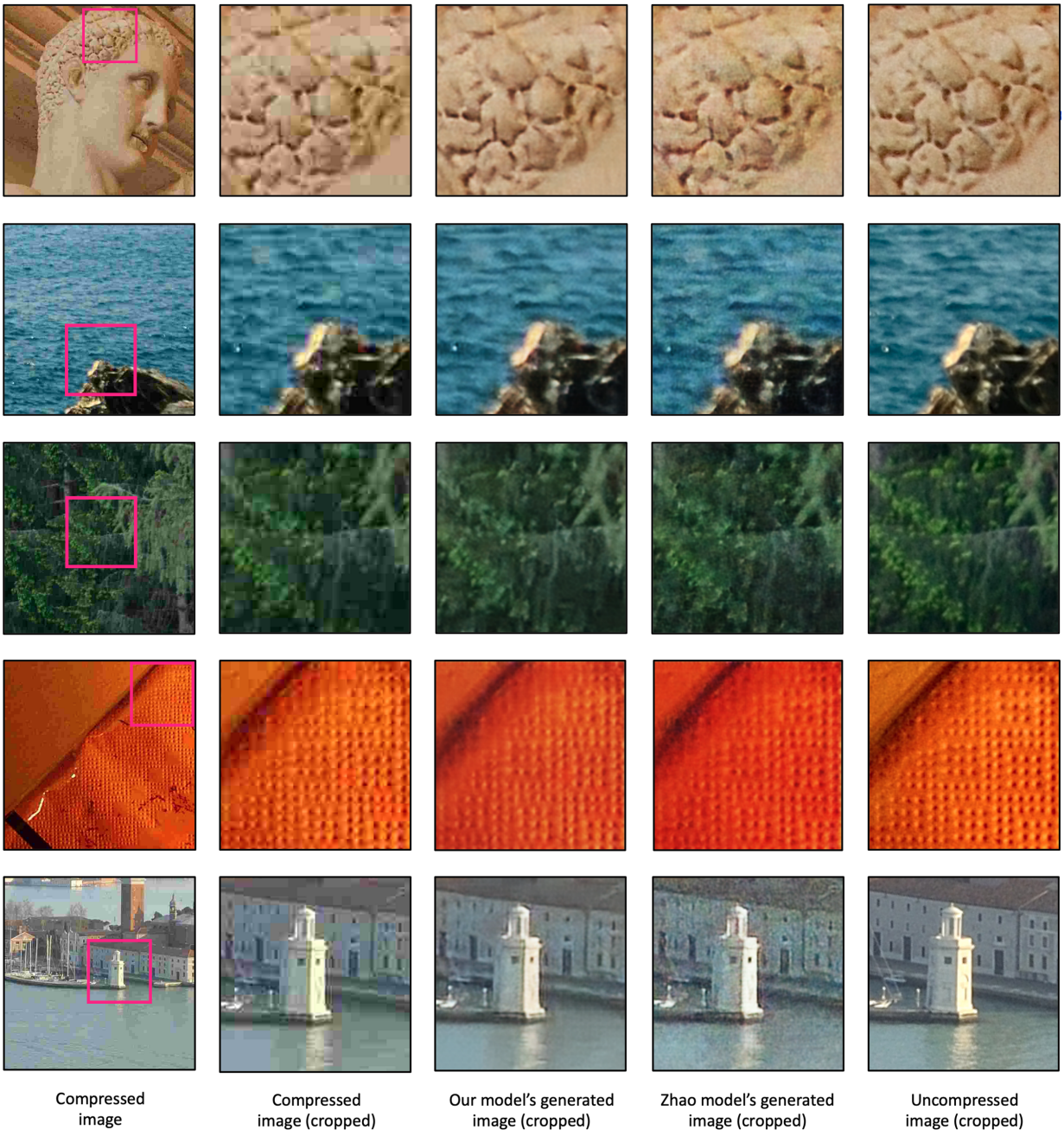


Figure 36: Comparison of the generated images of our model and Zhao et al.'s model. Both models were trained on images compressed with single compression QF=10 and double compression QF1=50, QF2=10. Note this only shows the case QF1 > QF2. (Zoom in to inspect the differences).

7.3 Limitations

Despite the impressive performance of our models, they come with their drawbacks. Many of these are due to our use of a limited dataset of only 1000 images and only five different QFs. For many instances of training, we do not even use the whole dataset due to performance and memory restrictions. It is not clear whether increasing the number of training examples would have significantly improved test results, but the limited variety of the data certainly restricts the applicability of our models for a wider range of images, which goes against our goal to make the models more practical. Specifically, the efficacy of our models drops significantly when considering images compressed with QFs outside of our dataset or images that were not taken by a camera (i.e. computer graphical images). This is due to these images not being represented by our dataset, and so the models are unable to learn to deal with them effectively.

Figure 37 shows an example of an image compressed with QF=50 being restored with a restorer trained with images of QF=10. As we can see, the restored image loses a lot of detail as the restorer has learned to be aggressive in its artifact removal due to training with a low QF. This results in an image that is less appealing than the input compressed image, demonstrating how the restorer is not effective for images with QFs dissimilar to the training QF.

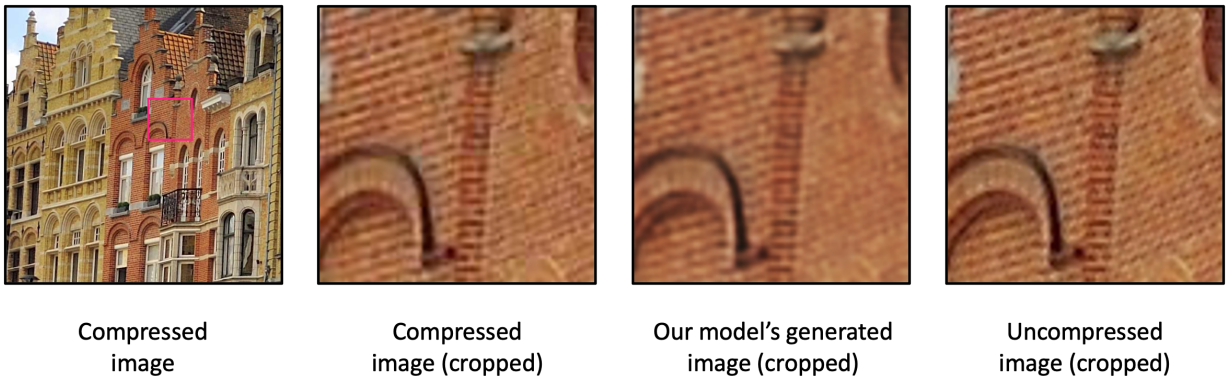


Figure 37: Generated image of an image compressed with QF=50 using a restorer trained with QF=10. (Zoom in to inspect the differences).

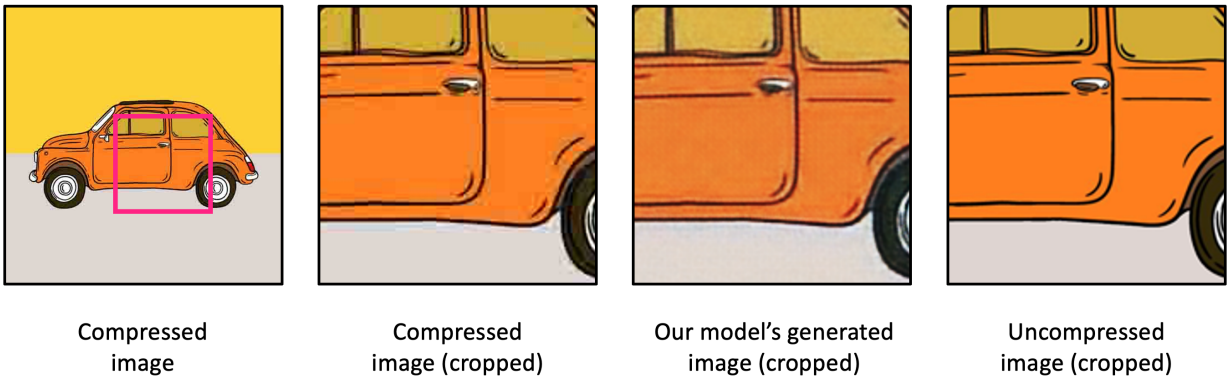


Figure 38: Generated image of a computer graphical image by our model. Model was trained on images compressed with QF=10. (Zoom in to inspect the differences).

We can also see an example of a poorly restored graphical image in Figure 38, where the restorer is still able to remove some JPEG blocking artifacts, but the resulting image is much less appealing. This is due to the restorer introducing artificial noise. While this makes natural images look more realistic, it is not desirable for graphical images as we require the image to be smooth in texture.

Furthermore, unlike the detector, the restorer does not generalize well when trained with a broad range of QFs, which we demonstrated in Section 5. This means that for different QF ranges, we require different models that are trained on specific subsets of the dataset. For our final solution, we are only able to include a single restorer trained for one QF (we opt for QF=10) since we have no method to identify an image's QF to invoke the appropriate restorer model.

An issue specific to the detector is that accuracy is poor for when QF1 >> QF2 or when QF1 = QF2. We discuss the reason for this in Section 4. However, we can observe that this is also an issue even for the more complex Park et al. model, shown by our results in Table 9. It is well-known that these cases are difficult to solve and there does not seem to be a clear solution, especially if we are to only consider information derived from pixels, as we have. As such, it is beyond our intended scope to develop a solution for this.

7.4 Future work

To address the limitations of our solution, we suggest potential approaches to mitigate the issues. We were unable to test the effectiveness of the following approaches due to resource and time constraints on our project. However, we believe they are fruitful ways to improve our solution.

An obvious improvement would be to use a larger dataset, using the full 8156 image RAISE dataset instead of the limited 1000 image dataset [49]. This could help the models achieve improved test results in general. However, to make the models more applicable to real-world images, it would also be advantageous to have a more diverse image dataset, including computer graphic images as well as images compressed with a wider variety of QFs. Then, for the detector, we could better test its generalizability, since, as shown in Section 5, it seems to improve for specific QFs when we introduce other QFs in training.

As for the restorer, we already know that it is unable to generalize for a small number of QFs, so it is unlikely to be able to generalize for even more QFs. Therefore, it may be beneficial to have multiple restorers, each trained for a specific range of QFs. We then require a method to estimate the QF of an input compressed image, so that the most suitable restorer can be invoked. There is already research into methods for estimating an image's compression QF, such as by exploiting the variance distribution on 8×8 blocks of an image in the spatial domain [51]. Thus, we can resolve our issue regarding different QFs of images. A similar idea could be implemented to handle graphical images. We know that a fundamental difference between natural images and graphical images is that natural images have image noise. It is our assumption that training a single restorer for both types of images would not result in a restorer that adds noise exclusively

for natural images and not for graphical images. Therefore, we would need to train distinct models for each case, and so we require a method to detect whether an image is a natural or graphical image beforehand. We believe this could be effectively achieved using a CNN as the two types of images have differing semantic properties, which convolutional layers would be well-equipped to discern.

8 Project management

To achieve our goals, we had to carefully consider the organization and execution of our project. This involved an informed choice of appropriate methodologies and tools to facilitate the project. Since our project encompassed both research and development, we had to effectively account for both in our plan.

8.1 Scope

The requirements for the project were decomposed into work packages and organized hierarchically to constitute our Work Breakdown Structure (WBS) [52]. This is shown in Figure 39. The WBS defines the total scope of the project, breaking down the deliverables into individual tasks. This enabled clear understanding of the work required, facilitating the construction of a project schedule. By decomposing the project into smaller and more manageable tasks, the WBS made it easier to manage and track project progress, improving overall project efficiency.

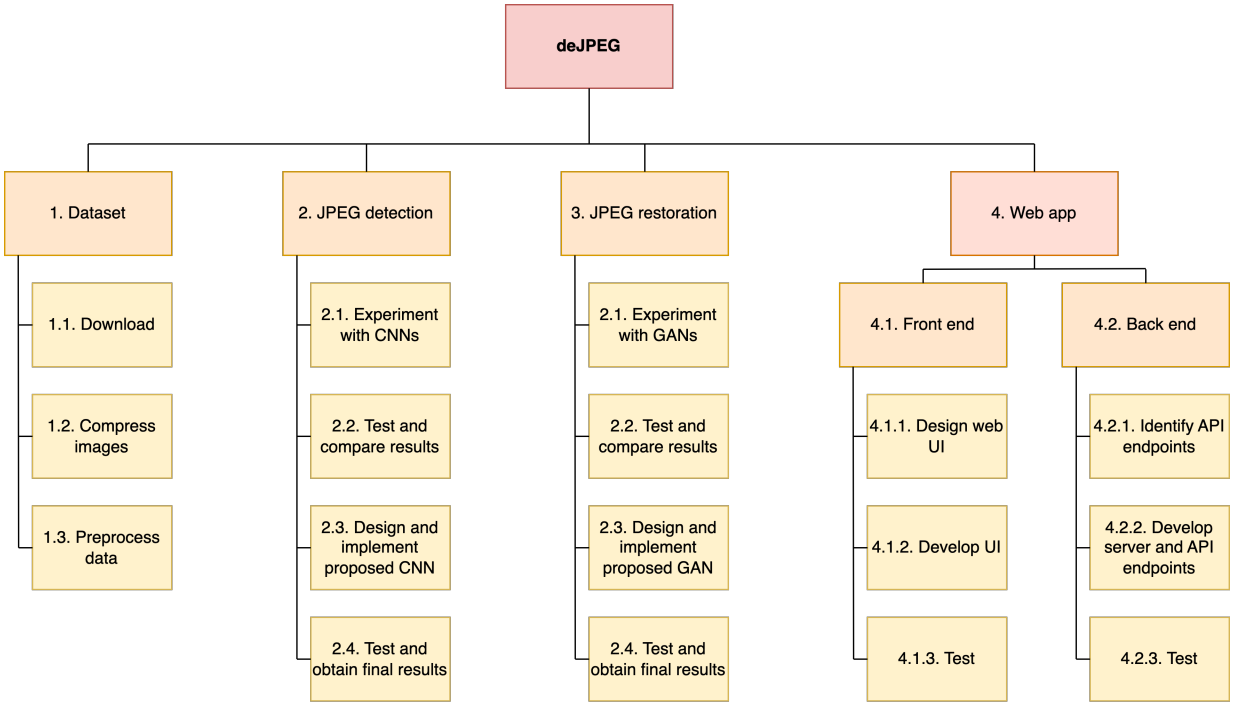


Figure 39: Our WBS.

8.2 Schedule

Our timetable was organized as a Gantt chart, which allowed us to clearly visualize the project timeline and track progress. Because of our Agile approach, the timetable was flexible and changed throughout project execution. We have included the final version of the Gantt chart in Appendix A. We believe that the use of the Gantt chart and its flexibility played a key role in our success.

Overall, progress throughout the project was good, and we closely followed the initial schedule. However, we experienced some delays in week 13, where development of the GAN for the JPEG restoration process proved more challenging than anticipated. Despite our best efforts, it took two additional weeks to complete the GAN experimentation and design. Fortunately, we had identified the potential risk of delays early on and built slack into the schedule as a countermeasure, so the impact of the delay was mitigated. Also, one of the primary obstacles with the GAN design was that there was a significant amount of time required for training, which meant that there was a lot of idle time. To address this issue, we brought web frontend development forward so that it could run in parallel with GAN design, during the downtime. This proved to be an effective solution, maximizing productivity and ensuring progress. As a result, we were able to complete the project within the given deadline.

8.3 Methodology

8.3.1 Research methodology

Our research involved many empirical experiments and so a high priority was placed on collecting valid results. Therefore, we followed the scientific method, which included implementing fair tests to obtain reliable results. To ensure fairness in our tests, we used the same subsets of the dataset for training and testing when comparing different models. This involved setting a seed so that the same random training and test examples were selected for each experiment. This helped eliminate any potential bias and ensured consistency for comparable results. Also, given that training of neural networks involves randomness, we repeated the tests at least three times and took the mean values to help avoid anomalies. It was also important to test the models on unseen examples to ensure that we evaluate the models' ability to generalize rather than memorize. To quantify the effectiveness of our models, we employed various metrics. These provided an objective way of measuring the performance, allowing us to draw reliable conclusions. In addition, qualitative testing was used to evaluate the visual appeal of images, which has some inherent subjectivity.

8.3.2 Development methodology

For development, we made use of agile principles since new ideas from research would influence the design and implementation of the system. Therefore, we had to remain flexible as prototypes and iterations of the system needed to be implemented and evaluated easily, suiting the agile approach. To enable this, checklists were maintained every week to ensure progress towards meeting the requirements. The success of this project can be largely attributed to our agile approach. As outlined in Section 8.2, the flexibility permitted by this methodology allowed us to make necessary timetable changes that would have not been possible with a more rigid waterfall approach, leading to project success. Moreover, our agile approach allowed us to extend our scope to test a variety of QFs, which produced more interesting results.

8.4 Tools

When experimenting with the different machine learning models, we predominantly used Python, specifically the TensorFlow 2 library. This simplified the development of the models, with much of the mathematics already implemented and abstracted as an API. The pretrained VGG-19 network is also included in the library, which was helpful as we did not need to rewrite and train the network ourselves. Furthermore, the library enabled accelerated computation using either CUDA for NVIDIA GPUs or Google's TPUs, which are well-optimized for modern machine learning workflows.

To access such hardware acceleration, we used Google Colab's cloud-based Jupyter notebook environment, which allowed us to run our code on virtual machines hosted by Google. The runtime environments were flexible, and we could choose between a GPU environment or a TPU environment. We more often made use of the GPU runtime since we purchased a subscription to Google Colab Pro, and so we could use the Premium GPUs. These were more powerful and had larger RAM, enabling us to reduce training time while also increasing the size of the training dataset – use of mulitple QFs for training would not have been possible without this. Google Colab also allowed us to offload our large dataset to Google Drive as Colab provided easy access to this.

In addition, to monitor the training of the machine learning models, we used metrics recorded from the TensorFlow stdout outputs, Matplotlib-drawn graphs, or Google's Tensorboard. These metrics were critical in tracking progress and fine-tuning the hyperparameters as they allowed us to identify unstable training patterns that would otherwise lead to poor convergence and, hence, subpar results.

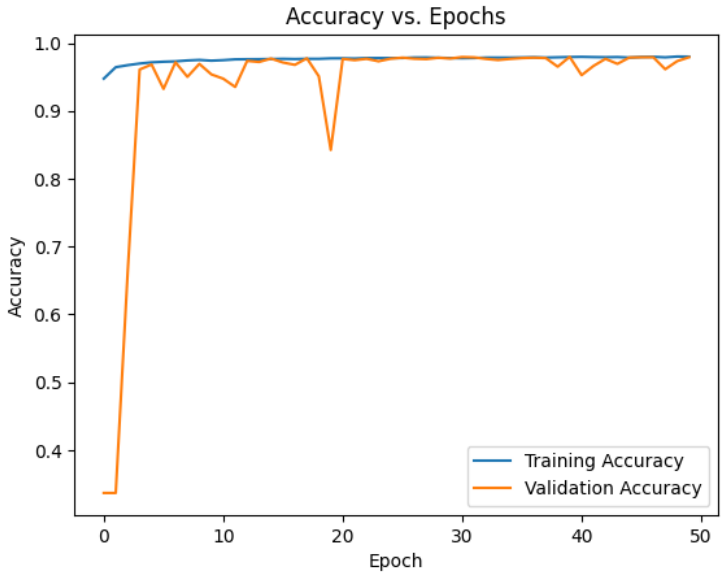


Figure 40: An example of a Matplotlib graph showing the training and validation accuracy for each epoch in CNN training.

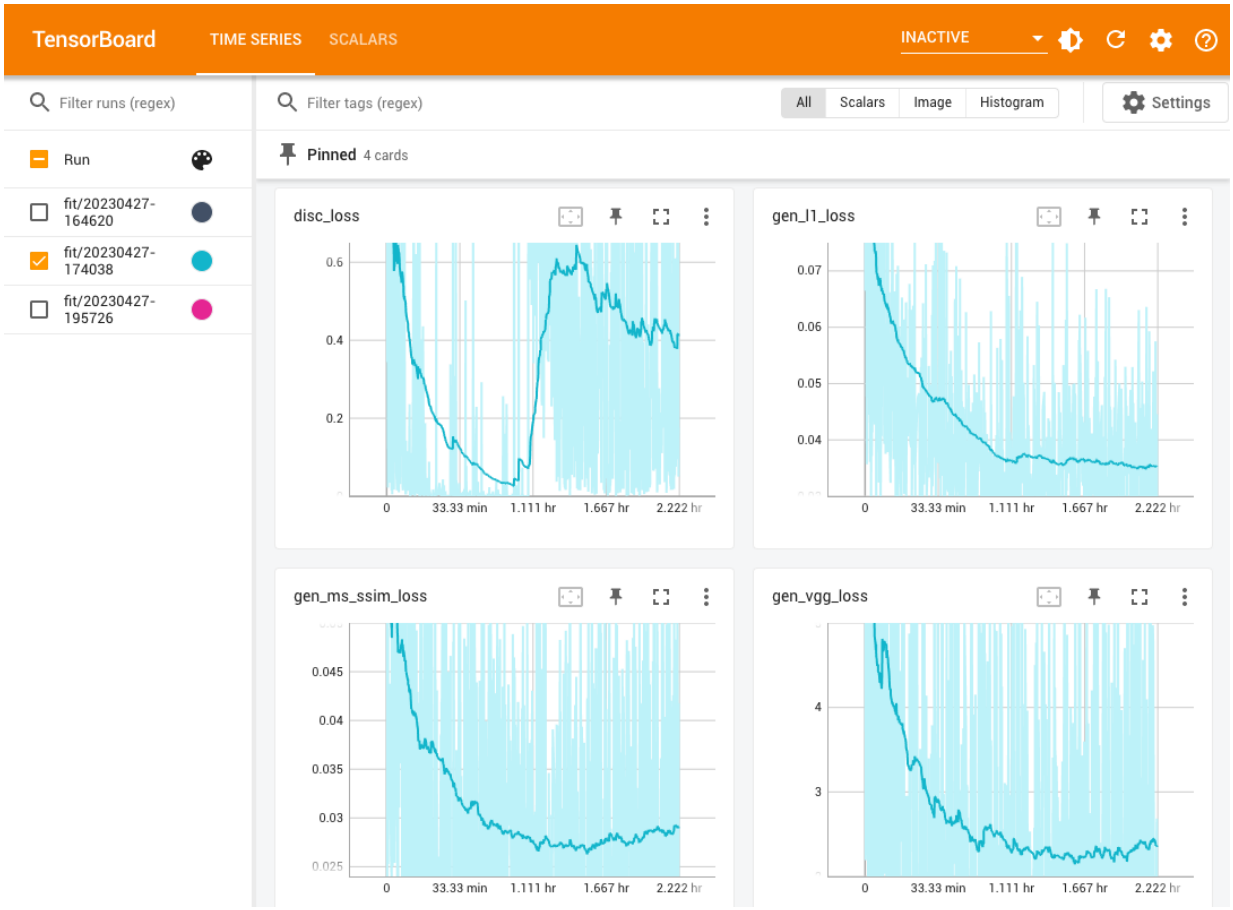


Figure 41: An example Tensorboard window during GAN training. Note that the sudden increase in discriminator loss is due to commencement of simultaneous discriminator and generator training (following NoGAN training, explained in Section 5). Also note that the high volatility of the losses is due to each training step using one image (i.e., batch size of 1), which was recommended by the Pix2Pix paper for image generation tasks [25]. Therefore, we use a rolling average to obtain a smoother line that more effectively captures the trend.

To implement the web application, we used Flask for the backend and React.js for the frontend, as discussed in Section 6. To manage version control for the website, we utilized Git repositories and uploaded them to GitHub. This enabled productive organization of the code as new features could be isolated in their own branch during prototyping. However, we found Git to not be as useful for machine learning experimentation, which involved frequent and ad hoc modifications to hyperparameters and other settings. Rather than creating new branches for each combination of parameters, we found it more effective to externally record the experiment's hyperparameters and other settings, along with its results, to avoid overcomplicating the codebase. Still, periodic backups were made when we believed changes to the machine learning code to be significant.

9 Conclusion

In summary, we are able to improve the degraded visual quality of JPEG compressed images with a system that can input an image, detect whether the image is JPEG (single or double) compressed, and if so, remove the JPEG artifacts. We have deployed the system as a user-friendly web application, developed using React.js for the frontend and Flask for the backend.

To detect the type of compression for an image, we obtain the DCT coefficient histogram features of an image with a method proposed by Barni et al. [14]. Then, to classify the histogram features, we implement a CNN, which closely follows Park et al.'s solution for double JPEG compression detection [6]. However, we simplify their model, reducing the number of parameters in the network, as well as removing the quantization table input. Consequently, our model is highly portable as it is nearly 300× smaller, and it can be applied to a wider variety of real-life scenarios as it does not rely on input images being stored in the JPEG format. We further enhance the practicality of our model by developing padding and partitioning methods that enable the model to accept images of varying sizes. Moreover, despite the reduced complexity of our model, we are still able to achieve close to the same overall accuracy as Park's model for images compressed with various compression QFs.

For the restoration of compressed JPEG images, we implement a GAN based on Zhao et al.'s solution for JPEG artifact removal [7]. Therefore, it utilizes a U-net generator [46] and a PatchGAN discriminator [25]. The U-net generator's use of only convolutional layers allows images of varying sizes to be input, and the PatchGAN discriminator guides the generator to accurately reconstruct high-frequency details in its output. As a result, the model is practical, and its generated images are more photorealistic than the generated images of a stand-alone generative CNN. To further improve the practicality of Zhao's model, we reduce the size of the network and modify it to accept coloured images. Despite these changes making the task more difficult, we are able to surpass the effectiveness of Zhao's model. We achieve this by employing NoGAN training, where the discriminator and generator networks are only trained simultaneously towards the end of training. We also introduce a novel perceptual loss function, which considers structural properties in multiple scales (using MS-SSIM) and incorporates semantic information from deep convolutional features (using the VGG-19 network). Furthermore, we are able to show that the same restoration model can be used to account for the differences between single and double compression.

Therefore, we have achieved all of our main objectives and thus our project is a success. However, there is still exciting work to be completed to improve our system. It would be beneficial to retrain the models with a larger dataset as we have only used 1000 images. In addition, to extend the applicability of our system, we could train multiple restoration models to account for different ranges of QFs, as well as for the differences between natural and graphical images. To use these models in practice, we would also need a detection system to determine the compression QF of an image, as well as whether an image is natural or graphical.

10 Legal, social, ethical and professional issues

The RAISE dataset [49] is available to use for the purpose of research. There are no other legal, social, ethical, or professional issues to consider with this project, and it does not involve any activities which require ethical consent.

11 Acknowledgements

I would like to express my sincere gratitude to my supervisor, Victor Sanchez, for his invaluable mentorship and support throughout the project. Victor's expertise and advice have been crucial in guiding my research and ensuring its success.

References

[1] R. C. Gonzalez and R. E. Woods, *Digital image processing (2nd Edition)*. Prentice Hall, 2002.

[2] E. Charbon, P. Ienne, P. Brisk, and T. Kluter, "Speculative dma for architecturally visible storage in instruction set extensions," *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis*, pp. 243–248, 2008.

[3] Y. Li, F. Guo, R. T. Tan, and M. S. Brown, "A contrast enhancement framework with jpeg artifacts suppression," *European conference on computer vision*, pp. 174–188, 2014.

[4] P. Dollár and C. L. Zitnick, "Structured forests for fast edge detection," *Proceedings of the IEEE international conference on computer vision*, pp. 1841–1848, 2013.

[5] R. Thakur and R. Rohilla, "Recent advances in digital image manipulation detection techniques: A brief review," *Forensic science international*, vol. 312, p. 110311, 2020.

[6] J. Park, D. Cho, W. Ahn, and H.-K. Lee, "Double jpeg detection in mixed jpeg quality factors using deep convolutional neural network," *Proceedings of the European conference on computer vision (ECCV)*, pp. 636–652, 2018.

[7] Z. Zhao, Q. Sun, H. Yang, H. Qiao, Z. Wang, and D. O. Wu, "Compression artifacts reduction by improved generative adversarial networks," *EURASIP Journal on Image and Video Processing*, no. 62, 2019.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[9] Q. Wang and R. Zhang, "Double jpeg compression forensics based on a convolutional neural network," *EURASIP Journal on Information Security*, vol. 2016, no. 23, 2016.

[10] M.-J. Kwon, S.-H. Nam, I.-J. Yu, H.-K. Lee, and C. Kim, "Learning jpeg compression artifacts for image manipulation detection and localization," *International Journal of Computer Vision*, vol. 130, no. 8, pp. 1875–1895, 2022.

[11] A. C. Popescu, "Statistical tools for digital image forensics," Ph.D. dissertation, Dartmouth College, 2005.

[12] B. Mahdian and S. Saic, "Detecting double compressed jpeg images," *IET Seminar Digest*, pp. 1–6, 2009.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[14] M. Barni, L. Bondi, N. Bonettini, P. Bestagini, A. Costanzo, M. Maggini, B. Tondi, and S. Tubaro, "Aligned and non-aligned double jpeg detection using convolutional neural networks," *Journal of Visual Communication and Image Representation*, vol. 49, pp. 153–163, 2017.

[15] H. C. Reeve III and J. S. Lim, "Reduction of blocking effects in image coding," *Optical Engineering*, vol. 23, no. 1, pp. 34–37, 1984.

[16] C. Wang, J. Zhou, and S. Liu, "Adaptive non-local means filter for image deblocking," *Signal Processing: Image Communication*, vol. 28, no. 5, pp. 522–530, 2013.

[17] A. Foi, V. Katkovnik, and K. Egiazarian, "Pointwise shape-adaptive dct for high-quality denoising and deblocking of grayscale and color images," *IEEE transactions on image processing*, vol. 16, no. 5, pp. 1395–1411, 2007.

[18] C. Dong, Y. Deng, C. C. Loy, and X. Tang, "Compression artifacts reduction by a deep convolutional network," *Proceedings of the IEEE international conference on computer vision*, pp. 576–584, 2015.

[19] P. Svoboda, M. Hradis, D. Barina, and P. Zemcik, "Compression artifacts removal using convolutional neural networks," *arXiv preprint arXiv:1605.00366*, 2016.

[20] L. Cavigelli, P. Hager, and L. Benini, "Cas-cnn: A deep convolutional neural network for image compression artifact suppression," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 752–759.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 2672–2680, 2014.

[22] ——, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[23] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4681–4690.

[24] T. Xu, P. Zhang, Q. Huang, H. Zhang, Z. Gan, X. Huang, and X. He, "Attngan: Fine-grained text to image generation with attentional generative adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1316–1324.

[25] P. Isola, J.-Y. Zhu, T. Zhou, and A. Efros, "Image-to-image translation with conditional adversarial networks," 07 2017, pp. 5967–5976.

[26] F. Mameli, M. Bertini, L. Galteri, and A. Del Bimbo, "A nogan approach for image and video restoration and compression artifact removal," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 9326–9332.

[27] J. Antic, J. Howard, and U. Manor, "Decrappification, deoldification, and super resolution," https://www.fast.ai/posts/2019-05-03-decrappify.html, 2019.

[28] L. Galteri, L. Seidenari, M. Bertini, and A. Del Bimbo, "Deep generative adversarial compression artifact removal," 2017.

[29] J. Ramirez, "Make smarter edits with neural filters." https://www.adobe.com/uk/products/photoshop/neural-filter.html, Accessed 2022.

[30] T. Hieber, "What is jpeg artifact removal? — adobe photoshop tutorial," https://taylorhieber.co/what-is-the-jpeg-artifact-removal-neural-filter-adobe-photoshop/, Accessed: April 2023.

[31] Y. Luo, H. Zi, Q. Zhang, and X. Kang, "Anti-forensics of jpeg compression using generative adversarial networks," in *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, pp. 952–956.

[32] H. Farid, "Lecture notes on digital image forensics," https://farid.berkeley.edu/downloads/tutorials/digitalimageforensics.pdf.

[33] N. T. University, "Jpeg still picture compression," https://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/jpeg/.

[34] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[35] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," *ICML*, pp. 807–814, 2010.

[36] J. Bridle, "Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters," *Advances in Neural Information Processing Systems*, vol. 2, 1989.

[37] J. S. Bridle, "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *Neurocomputing: Algorithms, architectures and applications*. Springer, 1990, pp. 227–236.

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, 2015.

[39] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37, pp. 448–456, 2015.

[40] A. Abu-Srhan, M. A. Abushariah, and O. S. Al-Kadi, "The effect of loss function on conditional generative adversarial networks," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 9, pp. 6977–6988, 2022.

[41] Z. Wang, E. P. Simoncelli, and A. C. Bovik, "Multiscale structural similarity for image quality assessment," in *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2. Ieee, 2003, pp. 1398–1402.

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[43] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*. Springer, 2016, pp. 694–711.

[44] A. Dosovitskiy and T. Brox, "Generating images with perceptual similarity metrics based on deep networks," *Advances in neural information processing systems*, vol. 29, 2016.

[45] H. Zhao, O. Gallo, I. Frosio, and J. Kautz, "Loss functions for image restoration with neural networks," *IEEE Transactions on computational imaging*, vol. 3, no. 1, pp. 47–57, 2016.

[46] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer, 2015, pp. 234–241.

[47] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[48] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, vol. 1, no. 10, p. e3, 2016.

[49] D.-T. Dang-Nguyen, C. Pasquini, V. Conotter, and G. Boato, "Raise - a raw images dataset for digital image forensics," *Proceedings of the 6th ACM multimedia systems conference*, pp. 219–224, 2015.

[50] R. ?, "React compare slider," https://react-compare-slider.vercel.app/, Accessed: April 2023.

[51] F. Retraint and C. Zitzmann, "Quality factor estimation of jpeg images using a statistical model," *Digital Signal Processing*, vol. 103, p. 102759, 2020.

[52] *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 6th ed. Project Management Institute, 2017.

A Gantt chart