Quantization Guided JPEG Artifact Correction: Appendices

A Additional Evaluation Details

In this section we elaborate on the evaluation procedure for prior works as well as discuss a number of hyperparameters critical to correct evaluation. In our results section, three of the four prior works did not have native handeling of color channels. To evaluate them on color images, we applied their Y channel network to both Y, Cb, and Cr channels separately as well as R, G, and B channels separately. In all cases, using the Y, Cb, and Cr channels performed the best, so these are the results we report (*e.g.*, we report the scheme that gives prior works the best numbers). Note that we do not modify the published network structure to take a three channel input as was done in IDCN. We do this to remain as faithful to the published methods as possible, and we note that by examining the numbers reported in IDCN, the ranking of the methods does not change. Altering the network structures to take a three channel input does, however, improve their results on color images even if it is a small improvement.

Next, we note important evaluation hyperparameters. We defer to the ARCNN evaluation code for these settings, although they are not objectively correct. SSIM evaluation in particular uses an 8×8 window with uniform weighting in contrast to the default 11×11 gaussian window. Setting this correctly is critical to producing a fair comparison and we have found prior works are not uniform in correctly setting it. ARCNN uses a strict definition of the Y channel giving an output in the range $[16, 240]$, this was intended to match the YCbCr transform used in the JPEG standard, however it is incorrect and stems from the default MATLAB settings. JPEG uses the full-frame Y channel conversion giving outputs in $[0, 255]$. We would like to see this corrected in future works, however it seems unlikely as it changes the comparisons quite a bit. Finally, we note that PSNR-B is an assymetric measure, *e.g.*, the blocking effect factor (BEF) is only computed

on the degraded image, so the order of the arguments is critical. We have seen at least one prior work that passes these arguments in reverse order resulting in nearly perfect PSNR-B (defined as PSNR-B very close to PSNR).

We have made our model and evaluation code as well as pretrained weights avaible at https://gitlab.com/Queuecumber/quantization-guided-ac. The evaluation code is reimplemented in PyTorch using ARCNN MATLAB code as a reference and checked for accuracy. We invite future work to use this framework for correct evaluation.

B Further Analysis

In this section we provide futher analysis of our model. We start by examining the Convolution Filter Manifold layers in more detail, providing visualizations of what they learn in order to better understand their contribution to our result. Next, we examine model interpolation in more detail by showing qualitative comparisons for varying interpolation strengths between the regression and GAN model. We then conduct a study that shows how much space can be saved by storing low quality JPEG images and using our method to restore them. We then examine the frequency domain qualitative results and show that our GAN model is capabile of generating images that have more high frequency content than the regression model alone. We conclude by examining the runtime throughput of our model compared to the other methods we tested against.

B.1 Understanding Convolutional Filter Manifolds

CFM layers are both our largest departure from a vanilla CNN and also quite important to learning quality invariant features, so it is a natural result to try to visualize their operation. In Figure 1, we compute the final 8×8 convolution weight for different quality levels. The quality levels, on the vertical axis, are 10, 50, and 100. The horizontal axis shows three different channels from the weight. What we see makes intuitive sense: the filters in different channels have different patterns, but for the same channel, the pattern is roughly the same as the quality increases. Furthermore, the filter response becomes smaller as the quality increases since the filters have to do less "work" to correct a high quality JPEG.

Next we visualize compression artifacts learned by the weight. To do this we find the image that maximally activates a single channel of the CFM weight. The result of this is shown in Figure 2. Again the horizontal axis shows different channels of the weight and the vertical axis shows quality levels 10, 50, and 100. The result shows clear images of JPEG artifacts. At quality 10, the local blocking artifacts are extremely prominant. By quality 50, the blocking artifacts are suppressed, while structural artifacts remain. The qualtiy 100 images are almost untouched, leaving only the input noise pattern. It makes sense that quality 100 filters are only minmally activated since there is not much correction

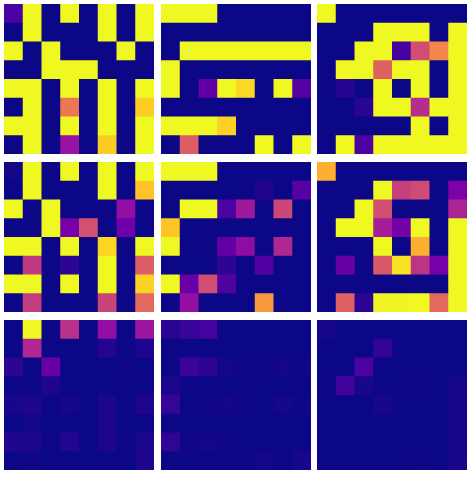

Fig. 1: **CFM Weight Visualization.** Horizontal axis shows different channels of the weight, vertical axis shows quality. Quality levels shown are Top: 10, Middle: 50, Bottom: 100.

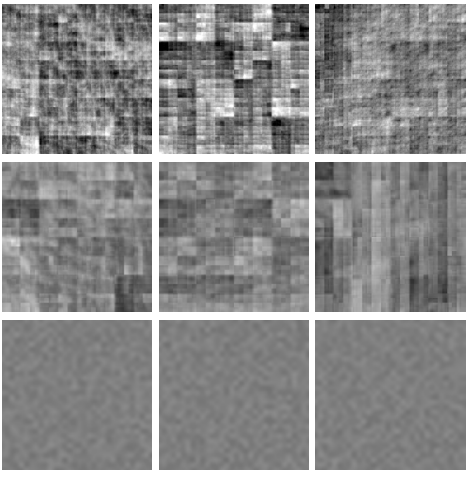

Fig. 2: **Images Which Maximally Activate CFM Weights.** Horizontal axis shows different channels from the weight, vertical axis shows quality. Quality levels shown are Top: 10, Middle: 50, Bottom: 100.

to do on a quality 100 JPEG. Note that we only show Y channel response for this figure and that Figures 1 and 2 use the same channels from the same layer.

Finally we examine the manifold structure of the CFM. We claim in Section 3.1 (and the name implies) that the CFM learns a smooth manifold of filters through quantization space. If this is true, then a quality 25 quantization matrix should generate a weight halfway inbetween a qualty 20 and a quality 30 one. To show that this happens, we generate weights for all 101 quanitzation matrices (0 to 100 inclusive) and then compute t-SNE embeddings to reduce the dimensionality to 2. We plot 3 channels from the weight embeddings with the quality level that was used to generate the weight given as the color of the point. This plot is shown in Figure 3. What see is a smooth line through the space starting from dark (low quality) to bright (high quality) showing that the CFM has not only separated the different quality levels but has ordered them as well. Futhermore we see that the low quality filters are separated in space, indicating that they are quite different (and perform different functions), a property that is important for effective neural networks. As the quality increases and the problem becomes easier, the filters tend to converge on a single point where they are all doing very little to correct the image.

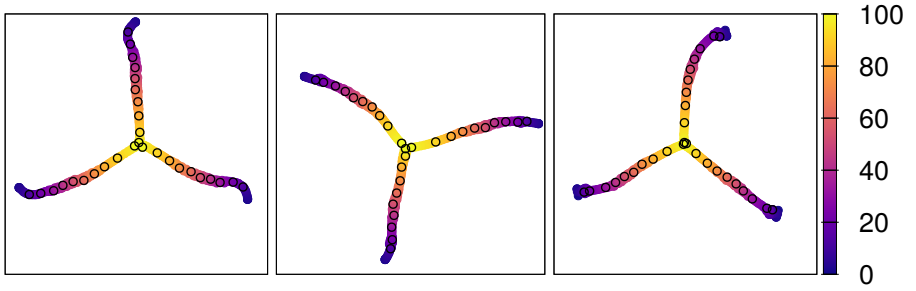

Fig. 3: **Embeddings for Different CFM Layers.** 3 channels are taken from each embedding, color shows JPEG quality setting that produced the input quantization matrix. Circled points indicate quantization matrices that were seen during training.

B.2 Model Interpolation

Here we show more model interpolation results. Model interpolation creates a new model by linearly interpolating the GAN and regresion model parameters as follows

$$\Theta_I = (1 - \alpha)\Theta_R + \alpha\Theta_G \tag{1}$$

where Θ_I are the interpolated parameters, Θ_R are the regression model parameters and Θ_G are the GAN model parameters with $\alpha \in [0, 1]$ being the interpolation parameter. The new model blends the result of the GAN and regression results. We observe that using the GAN model alone can introduce artifacts (see Figure 4), blending the models in this way helps surpress those artifacts. Note that in this scheme, $\alpha = 0$ gives the regression model and $\alpha = 1$ gives the GAN model. Model interpolation has been shown to produce cleaner results than image interpolation, and has the added benefit of not needing to run two models to produce a result. In Figure 4 we show the model interpolation results for $\alpha \in \{0.0, 0.7, 0.9, 1.0\}$ for several images from the Live-1 dataset. This figure also serves as additional qualitative results for our method. These results were generated from quality 10 JPEGs.

B.3 Equivalent Quality

One major motivation for JPEG artifact correction is that space or bandwidth can be saved by transmitting a small low quality JPEG and algorithmically correcting it before display. We explore how effective our model is at this by computing the equivalent quality JPEG file for a restored image. Our argument is that a system can get the storage space savings of the lower quality JPEG and the visual fidelity of a higher quality JPEG by using our model.

To show this we use the Live-1 dataset. For qualities in [10, 50] in steps of 10, we compute the average increase in JPEG quality incurred by our model. We do this by compressing the input image at higher and higher qualities until

Fig. 4: Model interpolation results 1/2

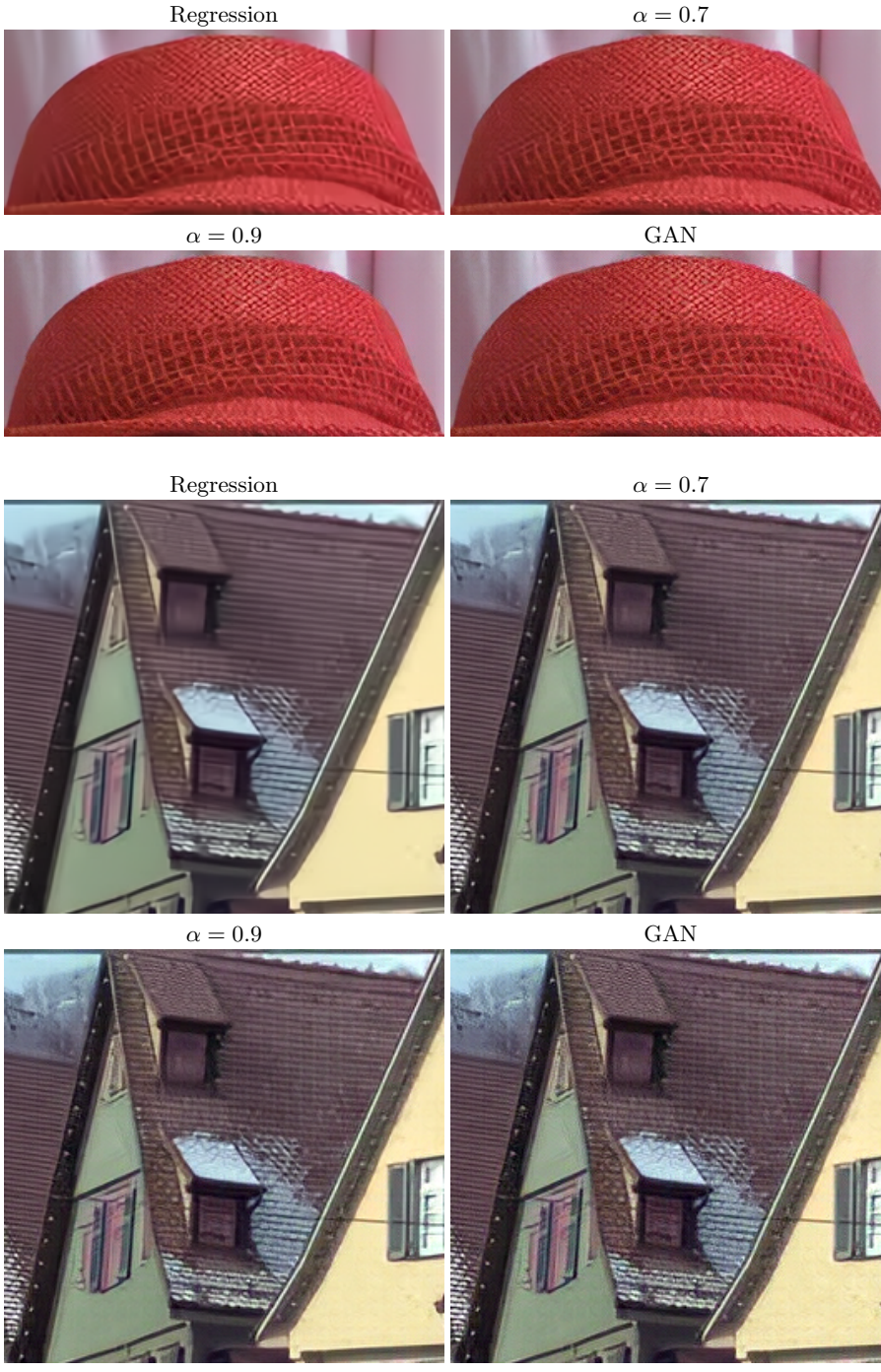

Fig. 4: Model interpolation results 2/2

we find the first quality with SSIM greater than or equal to our restoration's SSIM. We then save the low quality JPEG and the equivalent quality JPEG and measure the size difference in kilobytes. We average the quality increase and space savings over the entire dataset, to show the amount of space saved by using our method over using the higher quality JPEG directly. This result is shown in Figure 5. We also show qualitative examples for several images in Figure 6. Note that because the SSIM measure is not perfect, often our model outputs images that look better than the equivalent quality JPEG.

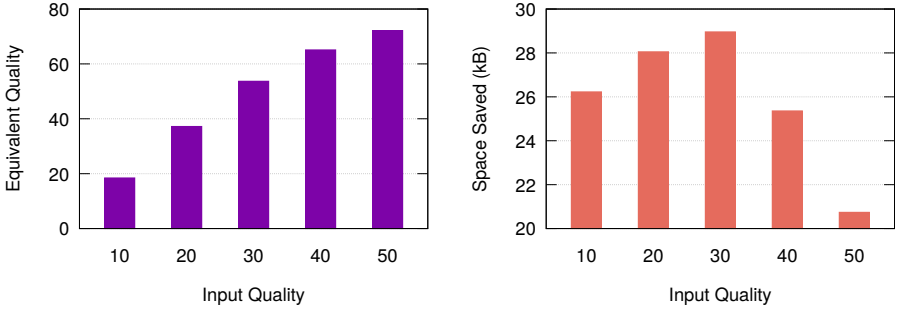

Fig. 5: Equivalent quality and space savings for Live-1 dataset.

B.4 Frequency Domain Analysis

In this section we show results in the DCT frequency domain. A well known phenomenon of JPEG compression is the removal of high frequency information. To check how well our model restores this information, we take the Y channel from several images and show the colormapped DCT of the original image, the JPEG at quality 10, the image as restored by our regression model, and the image restored by our GAN model. Next, for each image, we plot the probability that each of the 15 spatial frequencies in a DCT block are set (*e.g.*, has a magnitude greater than 0). This is shown in Figure 7. While our regression model is able to fill in high frequencies, our GAN model nearly matches the original images in terms of frequency saturation. Additionally since our network operates in the DCT domain, these outputs serve as an interesting qualitative result.

B.5 Runtime analysis

We show the runtime inference performance of our network compared to the other networks we ran against. We measure FPS on our NVIDIA Pascal GPU for 100 720p (1280×720) frames and plot frames per second vs SSIM increase for quality 10 Live-1 images in Figure 8. We do not include ARCNN in this figure as the authors do not provide GPU accelerated inference code. For grayscale only

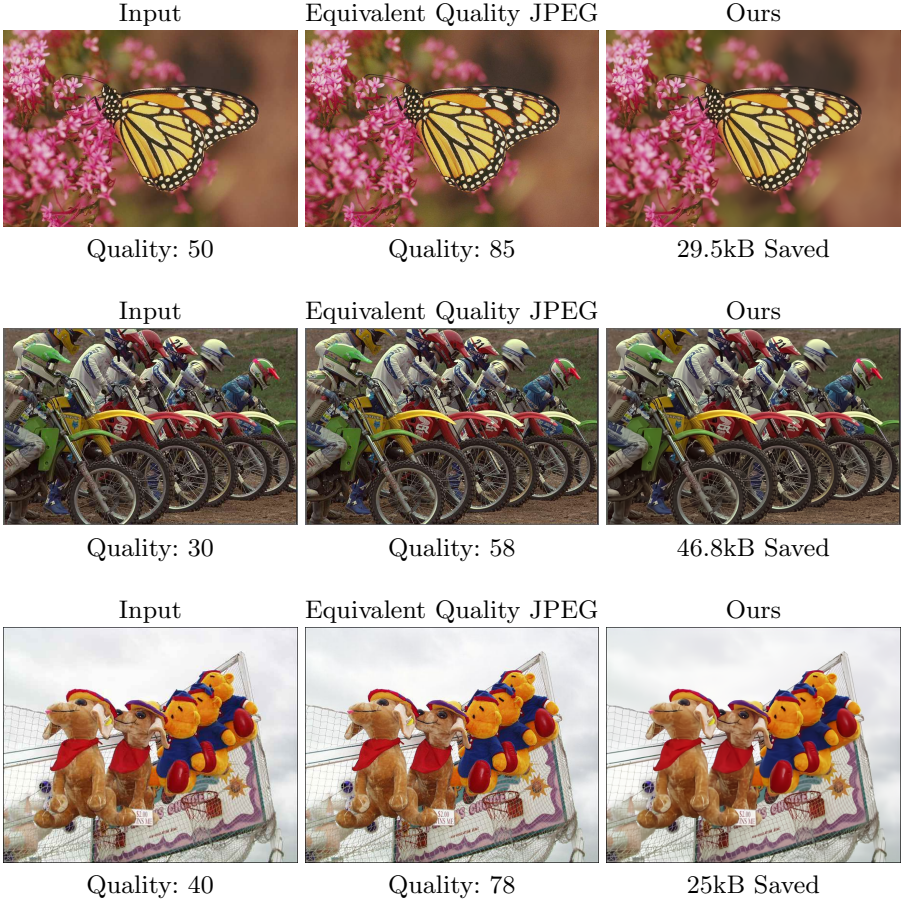

Fig. 6: Equivalent quality visualizations. For each image we show the input JPEG, the JPEG with equivalent SSIM to our model output, and our model output.

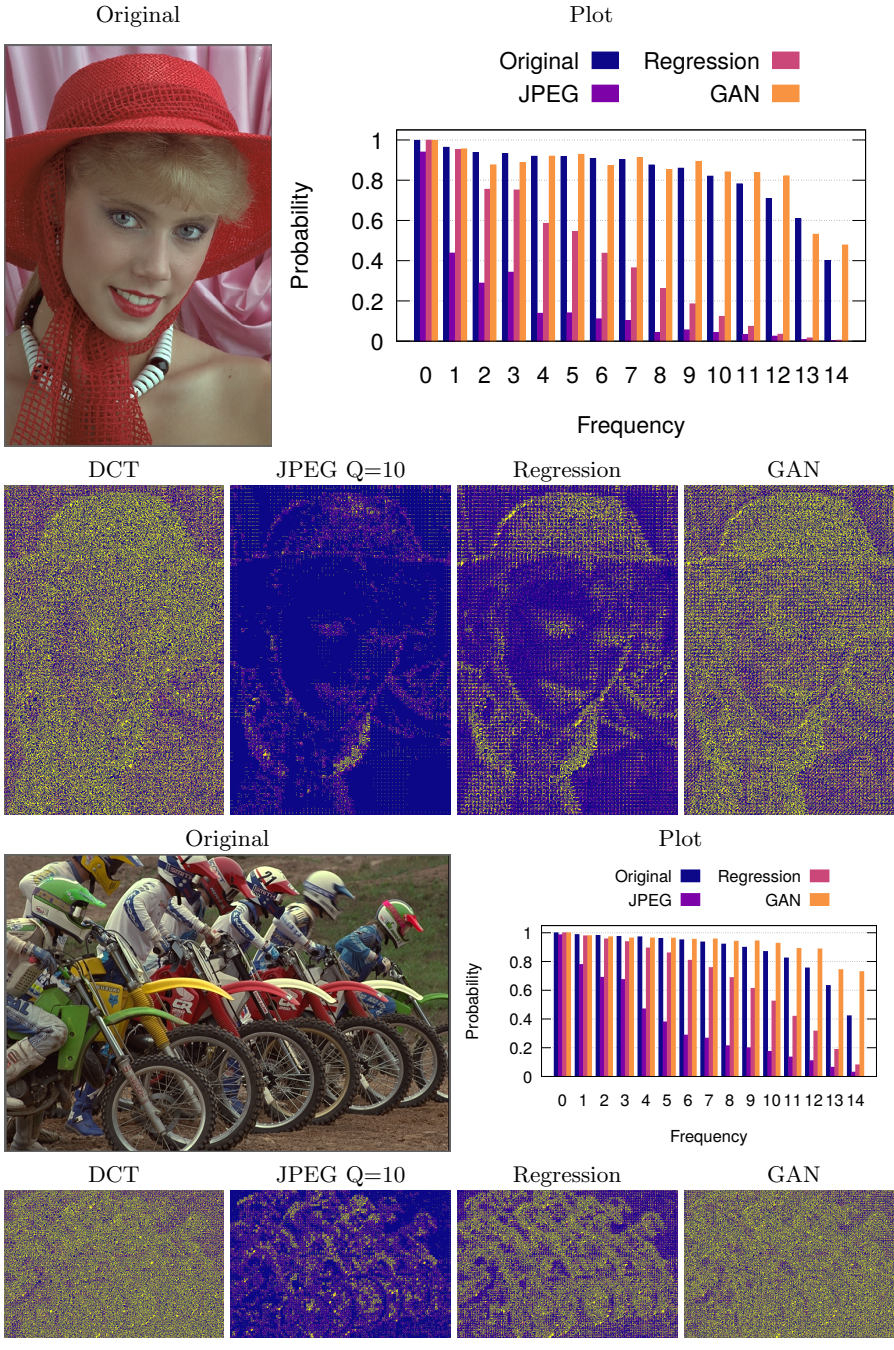

Fig. 7: Frequency domain results 1/2.

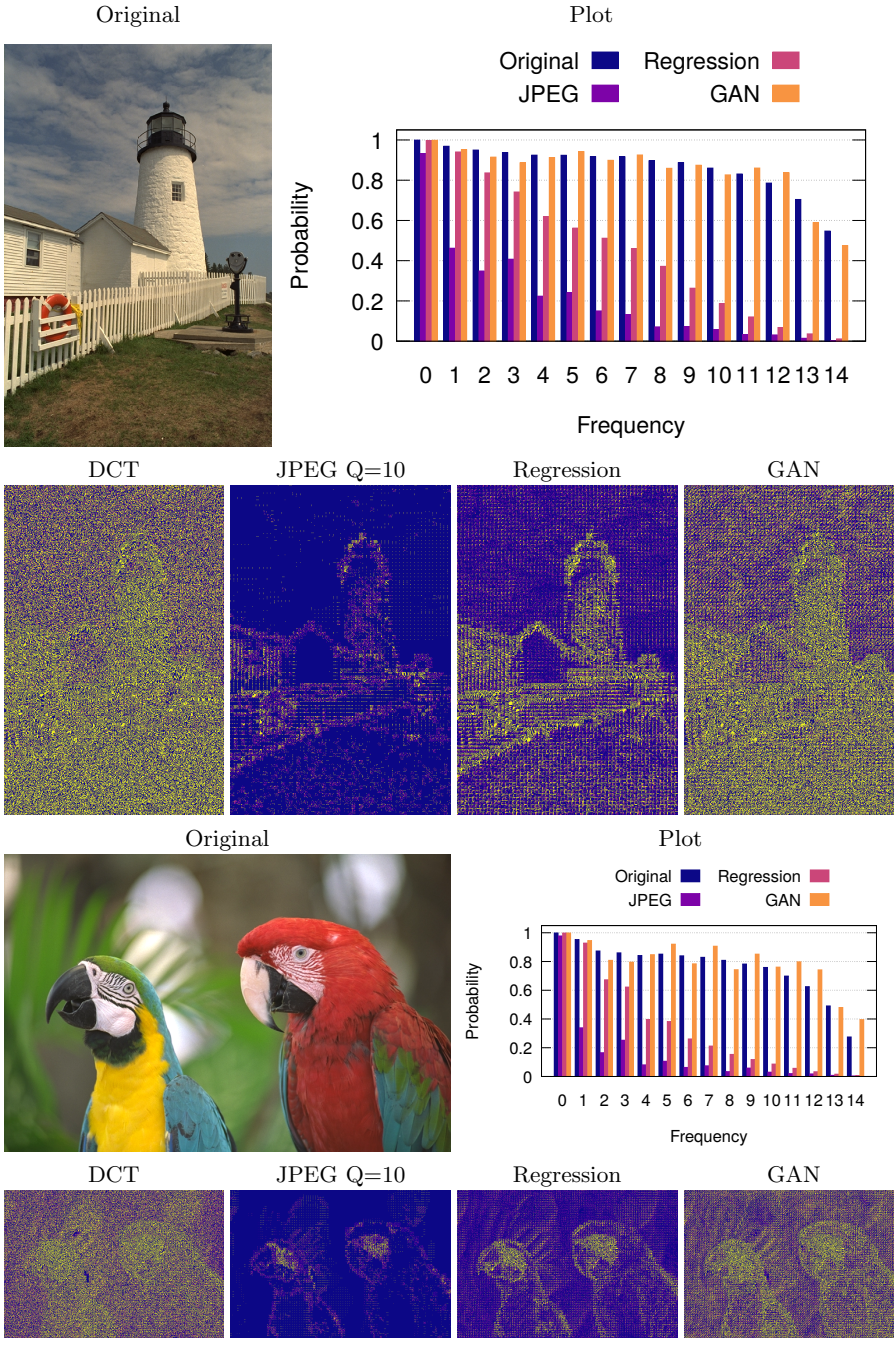

Fig. 7: Frequency domain results 2/2.

models we only use single channel test images (we not not run the model three times as would be required to produce an RGB output).

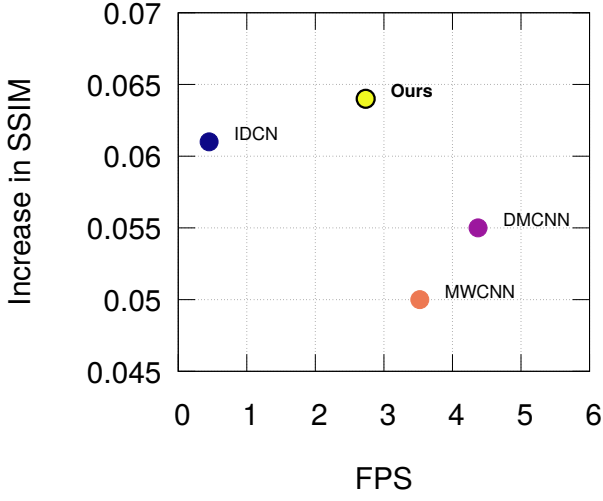

Fig. 8: Increase in SSIM vs FPS. Our result is highlighted.

C Qualitative Results

In this section we show qualitative results on Quality 10 and 20 images for our regression network. These results are in Figure 9.

D JPEG Compression Algorithm

Since the JPEG algorithm is core to the operation of our method, we describe it here in detail. Where the JPEG standard is ambiguous or lacking in guidance, we defer to the Independent JPEG Group's libjpeg software.

Compression JPEG compression starts with an input image in RGB color space (for grayscale images the procedure is the same using only the Y channel equations) where each pixel uses the 8-bit unsigned integer represenation (*e.g.*, the pixel value is an integer in [0, 255]). The image is then converted to the YCbCr color space using the full 8-bit represenation (pixel values again in [0, 255], this is in contrast to the more common ITU-R BT.601 standard YCbCr

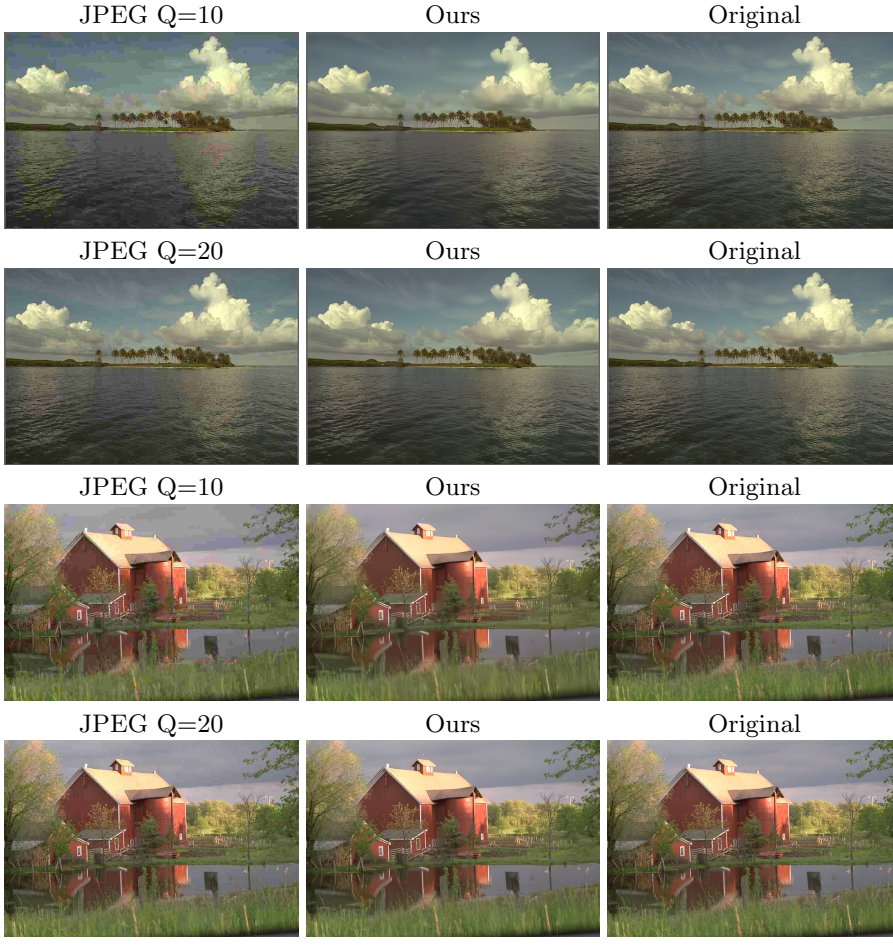

Fig. 9: Qualitative results 1/2. Live-1 images.

Fig. 9: Qualitative results 2/2. ICB images.

color conversion) using the equations:

$$\begin{aligned} Y &= 2.99R + 0.587B + 0.114G \\ Cb &= 128 - 0.168736R - 0.331264B + 0.5G \\ Cr &= 128 + 0.5R - 0.418688B - 0.081312G \end{aligned} \tag{2}$$

Since the DCT will be taken on non-overlapping 8×8 blocks, the image is then padded in both dimensions to a multiple of 8. Note that if the color channels will be chroma subsampled, as is usually the case, then the image must be padded to the scale factor of the smallest channel times 8 or the subsampled channel will not be an even number of blocks. In most cases, chroma subsampling will be by half, so the image must be padded to a multiple of 16, this size is referred to as the minimum coded unit (MCU), or macroblock size. The padding is always done by repeating the last pixel value on the right and bottom edges. The chroma channels can now be subsampled.

Next the channels are centered around zero by subtracing 128 from each pixel, yielding pixel values in [-128, 127]. Then the 2D Discrete type 2 DCT is take on each non-overlapping 8×8 block as follows:

$$D_{i,j} = \frac{1}{4} C(i) C(j) \sum_{x=0}^{7} \sum_{y=0}^{7} P_{x,y} \cos\left[\frac{(2x+1)i\pi}{16}\right] \cos\left[\frac{(2y+1)j\pi}{16}\right] \tag{3}$$

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

Where $D_{i,j}$ gives the coefficient for frequency i, j, and $P_{x,y}$ gives the pixel value for image plane P at position pixel position x, y. Note that $C(u)$ is a scale factor that ensures the basis is orthonormal.

The DCT coefficients can now be quantized. This follows the same procedure for the Y and color channels but with different quanitzation tables. We encourage readers to refer to the libjpeg software for details on how the quantization tables are computed given the scalar quality factor, an integer in [0, 100] (this is not a standardized process). Given the quantization tables Q_Y and Q_C, the quanized coeffcients of each block are computed as:

$$\begin{aligned} Y'_{i,j} &= \text{truncate}\left[\frac{Y_{i,j}}{Q_{Y_{i,j}}}\right] \\ Cb'_{i,j} &= \text{truncate}\left[\frac{Cb_{i,j}}{Q_{C_{i,j}}}\right] \\ Cr'_{i,j} &= \text{truncate}\left[\frac{Cr_{i,j}}{Q_{C_{i,j}}}\right] \end{aligned} \tag{4}$$

The quantized coefficients for each block are then vectorized (flattened) using a zig-zag ordering (see Figure 10) that is designed to place high frequencies further towards the end of the vectors. Given that high frequencies have lower magnitude and are more heavily quanitized, this usually creates a run of zeros at

the end of each vector. The vectors are then compressed using run-length encoding on this final run of zeros (information prior to the final run is not run-length encoded.). The run-length encoded vectors are then entropy coded using either huffman coding or arithmetic coding and then written to the JPEG file along with associated metadata (EXIF tags), quantization tables, and huffman coding tables.

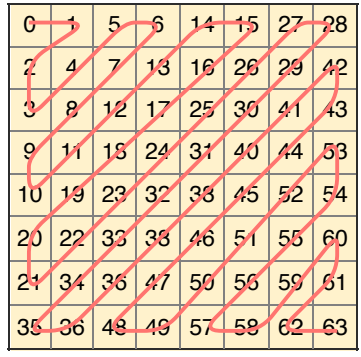

Fig. 10: **Zigzag Ordering**

Decompression The decompression algorithm largely follows the reverse procedure of the compression algorithm. After reading the raw array data, huffman tables, and quantization tables, the entropy coding, run-length coding, and zigzag ordering is reversed. We reiterate here that the JPEG file does not store a scalar quality from which the decompressor is expected to derive a quanitzation table, the decompressor reads the quanitzation table from the JPEG file and uses it directly, allowing any software to correctly decode JPEG files that were not written by it.

Next, the 8×8 blocks are scaled using the quantization table:

$$\begin{aligned} Y_{i,j} &= Y'_{i,j} Q_{Y_{i,j}} \\ Cb_{i,j} &= Cb'_{i,j} Q_{C_{i,j}} \\ Cr_{i,j} &= Cr'_{i,j} Q_{C_{i,j}} \end{aligned} \tag{5}$$

There are a few things to note here. First, if dividing by the quantization table entry during compression (Equation 5) resulted in a fractional part (the result was not an integer), that fractional part was lost during truncation and the scaling here will recover an integer near to the true coefficient (how close it gets depends on the magnitude quantization table entry). Next, if the division in Equation 5 resulted in a number in [0, 1), then that coeffient would be truncated to zero and is lost forever (it remains zero after this scaling process). This is the *only* source of loss in JPEG compression, however it allows for the result to fit

into integers instead of floating point numbers, and it creates larger runs of zeros which leads to significantly larger compression ratios.

Next, the DCT process for each block is reversed using the 2D Discrete type 3 DCT:

$$P_{x,y} = \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i)C(j)D_{i,j}\cos\left[\frac{(2x+1)i\pi}{16}\right]\cos\left[\frac{(2y+1)j\pi}{16}\right] \tag{6}$$

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

and the blocks are arranged in their correct spatial positions. The pixel values are uncentered (adding 128 to each pixel value), and the color channels are interpolated to their original size. Finally, the image is converted from YCbCr color space to RGB color space:

$$\begin{aligned} R &= Y + 1.402(Cr - 128) \\ G &= Y - 0.344136(Cb - 128) - 0.714136(Cr - 128) \\ B &= Y + 1.772(Cb - 128) \end{aligned} \tag{7}$$

and cropped to remove any block padding that was added during compression. The image is now ready for display.