Universiteit
Leiden
The Netherlands

Opleiding Informatica

Image Compression

with Neural Networks

Orson R. L. Peters

Supervisors:
Dr. Wojtek Kowalczyk & Dr. Lu Cao

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl` 12/08/2019

Abstract

In this bachelor thesis we describe methods for compressing computer images with traditional neural networks. Two entirely separate methods are discussed, one lossy and the other lossless. The lossy compression uses a neural network to approximate an image after which the network weights are stored as the compressed image. The lossless method improves upon the FLIF image format, replacing a simple prediction with a neural network. Neither approach immediately leads to state-of-the art performance, but could be used as a basis for future research.

Contents

Chapter 1

Introduction

Image compression is a technique with which we can reduce the amount of data required to store an image in a file. This is possible as images interesting to humans are not random - they contain various patterns. These patterns can be very high-level such as subject matter (e.g. 'human faces') or very low-level patterns such as gradients or lines. An image compressor can take advantage of these patterns.

Image compression is an important backbone of our digital society. On the HTTP archive [HTT19] we find that on average roughly half of the data by a web page from 2016 through 2018 consists of image data. And this is already compressed with current mainstream algorithms such as PNG or JPEG. If this data were sent in a raw uncompressed pixel grid form it would be orders of magnitude bigger.

Similarly image compression is vital for data archival. In a modern world where archival happens digitally the limit of storage is not real estate but the price of storage hardware. A better compression ratio directly leads to cost savings or an increase in archival capacity.

Neural networks are a hot topic in machine learning, rapidly finding applications left and right, especially in computer vision. This thesis is no exception, and in it we will explore two methods of applying neural networks to the problem of image compression. While there are many powerful methods all falling under the umbrella of neural networks, in this thesis we will be restricting our scope to only *multilayer perceptrons* (see Section 3.1) for simplicity.

There are two possible approaches to image compression: *lossy* and *lossless*. Lossless methods are objective methods that retain full information of the original image, and merely compress it. After decompressing, the resulting image must be pixel-by-pixel entirely identical to the original. These methods are most important for archival. Not only because any errors in the data are unwanted, but as you re-encode images over the years these errors would compound until the result is unrecognizable if you were to use lossy methods.

Lossy methods are subjective methods that do not have to exactly reproduce the original image. Instead they aim to encode the image in such a way that when decoded it is as 'similar' as possible to the original image. This invariably introduces some error, but the advantage is often a vastly improved compression rate which can go up to several orders of magnitude. With a carefully tweaked algorithm the error can be imperceivable to humans while maintaining significant file size savings. While image quality is inherently subjective, there exist objective metrics that aim to encapsulate perceptual image quality. See Section 4.1 for more details.

Our lossless method builds on the work of FLIF [SW16]. In the core workings of FLIF's algorithm it encodes the differences of pixel values from a weak predictor. We replace this weak predictor with a multilayer perceptron for marginal improvements.

Our lossy compression method was inspired by CocoNet [BI18], which transforms pixel coordinates into colors to approximate a picture. Our method goes beyond that, and chooses to interpret the learned network weights as a compressed image file, an approach that to our knowledge has not been seen before.

In Chapter 2 we discuss related work. In Chapter 3 we explain the methods we use, firstly multilayer perceptrons (Section 3.1) followed by methods for lossy compression (Section 3.2) and lossless compression (Section 3.3). In Chapter 4 we discuss the metric we use for perceptual image quality as well as our experiments. Finally, in Chapter 5 we discuss our results and in Chapter 6 our conclusions and avenues for future research.

Chapter 2

Related work

While technically our methods compete with any other image compression method, the field of image compression is old and large, and citing all possible methods is an exercise in futility. Therefore we limit ourselves to specifically to the algorithms we chose to compare ourselves with (PNG, JPEG and FLIF) and methods specifically using neural networks to compress images. FLIF [SW16] is discussed in detail in Section 3.3.1.

PNG [ISO04] is a ubiquitous lossless image format that is very general and thus has various implementations with varying quality of compression. All share a common core which uses the DEFLATE algorithm on rows of pixels with a simple filter applied per row that allows it to encode the differences of adjacent pixels instead of absolute values if this is beneficial.

JPEG [Wal91] is a similarly ubiquitous lossy image format. The algorithm transforms the pixels into a color space that separates brightness and color. Then each 8×8 block of pixels is transformed to a frequency domain using a Discrete Cosine Transform [Str99] (DCT). These frequencies are then quantized, with higher frequencies receiving less precision, with the idea that human perception is less sensitive to high frequency changes in an image and more focused on large-scale changes to color or brightness. Finally these quantized blocks are compressed further using a lossless algorithm and stored.

In [TVJ$^+$16] Google research uses a full neural network architecture to beat JPEG, as a first. They separate the problem in two neural networks, an encoder that generates codes from an image and a decoder that turns codes back into an image. They iteratively improve the quality of an image by encoding residuals of the previous iteration's encoded image subtracted from the original, facilitated by the network's internal memory due to recurrent nodes.

A different approach is used in [JTL$^+$17]. Two convolutional neural networks are used: the first downsamples the image and a traditional lossy method is used to compress the downsampled image. The decompressed image is then upscaled again and the second network is used on it to predict a residual which corrects some of the compression artifacts.

Chapter 3

Methodology

3.1 Multilayer perceptrons

The multilayer perceptron (MLP, also known as multilayer feedforward network) is one of the oldest and simplest of neural network architectures. It consists of one layer of input nodes, one or more hidden layers of neurons and one layer of output neurons. It maps the inputs to the outputs in such a way that we can automatically learn the mapping based on example input/output pairs.

Each neuron n_i^l in layer l is connected to all nodes in the previous layer, with a distinct weight $w_{i,k}^l$ for each connection k. The value of a neuron can be computed by the weighted sum of its connections plus a bias value b_i^l as an added constant.

However, this would simply form a linear relationship between the input nodes and output nodes. There would be no additional benefit of introducing hidden layers at all (compositions of linear maps stay linear). Therefore, each neuron applies a non-linear activation f to its weighted sum:

$$n_i^l = f\left(b_i^l + \sum_k w_{i,k}^l n_k^{l-1}\right)$$

The last layer (output layer) may or may not have an activation function applied depending on the desired output. Then it is a well-known result [Hor91] that with a continuous, bounded and non-constant activation function such a network can arbitrarily approximate continuous mappings over compact input sets, *assuming you find the right weights and biases*. This is where two very powerful tools come into play: automatic differentiation and backpropagation. They allow us to automatically approximate good weights/biases using gradient descent based on some set of example inputs X with known correct values $\boldsymbol{y}$.

Automatic differentiation allows us to exactly and quickly compute the derivative of an arbitrarily complex function with respect to any of its variables, with one condition: the function consists strictly of the composition of simple functions for which we know the derivative. Assuming we choose f as a simple to differentiate function, then our entire network qualifies.

At its core automatic differentiation (AD) is the repeated application of the chain rule $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$ where y, z are dependent variables on x. Since our complex function consists of only the composition of functions, we can write it as a series of steps, each step applying a function on temporaries we've previously computed. E.g. for $y = f(g(x))$:

$$
\begin{aligned}
t_0 &= x \\
t_1 &= g(t_0) \\
t_2 &= f(t_1) \\
y &= t_2
\end{aligned}
$$

There are two main modes of operation in AD. In forward mode AD we use the chain rule to recursively compute $\frac{dt_i}{dx}$, and in backwards mode AD we recursively compute $\frac{dy}{dt_i}$. Backwards mode is more efficient if you wish to know the derivative of each internal expression in the function with respect of the output. Backpropagation wishes to exactly know that, so we'll stick with backwards mode AD. We can repeatedly use the following identity:

$$\frac{dy}{dt_i} = \frac{dy}{dt_{i+1}} \cdot \frac{dt_{i+1}}{dt_i}$$

With as base case $\frac{dy}{dy} = 1$. To apply this, we first do a forward pass, and keep the value of each temporary in memory. Then we can go backwards and fill out $t'_i = \frac{dy}{dt_i}$ for each i. As an example, let $f(x) = x^2$ and $g(x) = \sin(x)$ and $x = \frac{1}{3}\pi$:

$$
\begin{aligned}
t_0 &= x & &= \frac{1}{3}\pi & t'_2 &= \frac{dy}{dy} = 1 \\
t_1 &= \sin(t_0) & &= \frac{1}{2}\sqrt{3} & t'_1 &= t'_2 \cdot \frac{dt_2}{dt_1} = 1 \cdot 2t_1 = \sqrt{3} \\
t_2 &= (t_1)^2 & &= \frac{3}{4} & t'_0 &= t'_1 \cdot \frac{dt_1}{dt_0} = \sqrt{3} \cdot \cos(t_0) = \frac{1}{2}\sqrt{3} \\
y &= t_2 & &= \frac{3}{4} & \frac{dy}{dx} &= t'_0 = \frac{1}{2}\sqrt{3}
\end{aligned}
$$

Then, backpropagation can use this information to learn our weights and biases (and any other *parameters* for more advanced neural networks). If we let T be our network, and $\hat{y}_i = T(x_i)$ be the output(s) of our network applied to x_i. Then if we define an error metric $E(\hat{\boldsymbol{y}}, \boldsymbol{y})$ again as a composition of simple differentiable functions, we can apply automatic differentiation to the entire construction, allowing us to efficiently and accurately compute

$$\frac{d}{dw^l_{i,k}} E(\hat{\boldsymbol{y}}, \boldsymbol{y}) \quad \text{and} \quad \frac{d}{db^l_i} E(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

and similar for each other parameter in the neural network. In plain words, we can compute exactly how steep the gradient of each parameter is with respect to the error, and in what direction it points. There are various optimizers that can be considered backpropagation, but they all perform some form of gradient descent, using the above gradients to nudge parameters against the gradient such that the total error is reduced.

After the parameters have been updated the whole process repeats again, with a forwards step followed by a backpropagation step, iteratively improving parameters. In each iteration the parameters are updated in proportion with a *learning rate*. It is common for optimizers to not use a fixed learning rate, but slowly decrease it over the iterations to balance early exploration v.s. late exploitation. A solution may get stuck in a local optimum, although various optimizers employ techniques to deal with those as well.

3.2 Lossy compression

3.2.1 Interpreting images as $[0,1]^2 \to [0,1]^3$ functions

A traditional computer image is formed by *pixels* arranged in a $w \times h$ grid, each pixel being a color in an RGB (red, green, blue) color space. However this is a very limited interpretation of an image. Instead, let's consider that there exists a *true* image, which has no concept of pixels and instead has a color defined at every point of its domain. A grid of pixels is then only a collection of (evenly spaced) point samples of the true image.

If we normalize the domain (ignoring aspect ratio and stretching the image) and color space to $[0,1]$, we can view an image as a $[0,1]^2 \to [0,1]^3$ function:

$$f(x,y) = (r,g,b)$$

If function f returns the (r,g,b) color at coordinates x,y, we can say that f **is** the image itself. Now except for digital vector art or other mathematical descriptions of images, we never truly know f, we only know f at certain points (the samples). However this is exactly what neural networks are good at: given samples to learn an underlying function.

And that's exactly what we do: we learn the image. We consider a computer image not as a grid of pixels, instead we consider each pixel as a $(x,y) \to (r,g,b)$ training sample where x,y are the coordinates of the pixel and r,g,b are the color components of the pixel, both normalized to $[0,1]$. Then we train a neural network T on these samples, and after it (hopefully) learns from these samples, we have $T \approx f$. At this point **the trained neural network represents the image**, with a quality depending on how well T approximates f.

Now that we have T we can use it as our image format. If we save T to a file and send it to someone they can load T, and turn it back into a grid of pixels by calculating the coordinates (x,y) of each pixel and evaluating T with those coordinates to retrieve the color of that pixel.

If we choose a very complex network for T with many weights, we will likely very closely approximate f. But our original goal isn't to learn f, it is to compress our original image! The main idea is to intentionally limit the number of neurons and connections in T to a small amount such that it selectively learns the most important patterns in f while foregoing small details. Since the number of neurons and connections in T will be small, it also should not take a lot of storage space to store T. In other words, we have lossy compression.

This is a fairly non-traditional use of neural networks and can be confusing, so let's reiterate. For **each** image we wish to compress we train a neural network to approximate that image. Our trained neural network then

represents the compressed image, and can be saved and/or transported. To decompress we load the neural network and evaluate it for each pixel location to deduce the color of that pixel.

$$\begin{array}{rlcl} \text{Compression:} & \text{Image as pixel grid} & \xrightarrow{\text{decompose}} (x,y,r,g,b)^{w\times h} \xrightarrow{\text{train}} & \text{network } T \\ \text{Decompression:} & (x,y)^{w\times h} & \xrightarrow{\text{evaluate with network } T} (x,y,r,g,b)^{w\times h} \xrightarrow{\text{compose}} & \text{Image as pixel grid} \end{array}$$

Note that this approach of viewing images as functions isn't new within the context of neural networks. CocoNet [BI18] also views an image as a $[0,1]^k \to [0,1]^3$ function and uses neural networks to learn this mapping. The focus in their paper is on the applications of this technique regarding up/downscaling and inpainting, although they do mention the uses for compression. The uses mentioned are different than the ones we explore here though, as the related work they cite is all based on pre- or post-processing of other lossy compression algorithms using neural networks, instead of using the actual trained network itself as the compressed format. To our knowledge that approach is novel.

As a final note, due to the activation function we use (tanh) the neural network prefers if all input and output is in $[-1,1]$ rather than $[0,1]$ so we normalize the input and output domain as such.

3.2.2 Storing neural networks

In order for us to be able to use the neural network as an image file we must be able to store the neural network in some sort of file format. We could simply use our neural network library of choice to export our model, but the resulting file would be relatively large. It would store each weight with 32-bit (or even 64-bit) floating point, contain all the network metadata, connections, etc.

There are pre-built solutions for network compression and quantization such as TensorFlow [ABC$^+$16] Lite, however those (and most existing literature, e.g. Hubara and Courbariaux et. al. [HCS$^+$17]) focus on reducing the precision and size of the network to increase prediction runtime performance or energy consumption. While we certainly would enjoy the benefits of those in this project as well, our main focus is to reduce the size of the stored neural network, so their solutions are not directly applicable. However they do show that neural networks are resilient to small errors introduced by **quantization**.

We quantize all weights and biases in our neural network to 8 bits per parameter. To minimize the loss of precision from quantization we use *equal frequency binning*, with the idea that less-frequently occurring values have a relatively larger error than common values, minimizing quantization error across all values.

In Section 4.1.1 we set up an experiment to find an approximate distribution of weights/biases so we can apply equal frequency binning. The result of this is a single table of $2^8 = 256$ values used for all images. Each parameter we save is rounded to its closest value in this table and the index saved instead. To load a weight back up we load the value at the index specified.

We did not train any models to specifically cope with quantization, except we put a constraint using Keras'

[C⁺15] `max_norm` such that no bias or weight had an absolute value greater than the maximal value in our table to prevent the model from acquiring weights that could not be reasonably quantized.

With the parameters saved in a file, only one more thing is necessary, the topology of the network. How many layers does it have, how many neurons in each layer? Instead of encoding this, we assume that both the encoder and decoder would have a couple of known good configurations, and an index into this list would be saved instead.

3.2.3 Auxiliary input encoding

While in principle with enough neurons a neural network can learn to approximate any function, having appropriate features is a very important aspect of machine learning, even with neural networks. Already in the CocoNet [BI18] paper they noticed an improvement in performance if they primed the neural network by synthesizing additional features beyond simply the (x, y) coordinates. They added polar coordinates and the (x, y) coordinates with a different origin as extra features and found increased performance of the network.

In addition to this we wish to test three additional ideas. The first two are very similar in nature, and are known as thermometer encoding [YC99]. The idea is to split the input encoding of x up into n inputs (the **granularity**), where each input is responsible for a portion of the domain of x. The kth input has a minimal value when x is below its portion, linearly interpolates between its minimum value and maximum value when x is inside it, and maintains its maximum value when x is greater than it. This looks very similar to a thermometer when visualized:

We chose n evenly spaced portions for the domains of x, y with the minimum and maximum values of each input set to -1 and 1 respectively. This gives the following formula for the kth input (for x, analogous for y), assuming $x \in [0, 1]$ and $k \in \{0, 1, \ldots, n-1\}$:

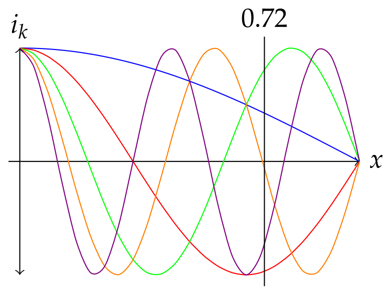
$$x_k = \operatorname{clip}(2(xn - k) - 1, -1, 1)$$

where

$$\operatorname{clip}(x, a, b) = \begin{cases} a & x < a \\ x & a \leq x \leq b \\ b & x > b \end{cases}$$

This however creates discontinuities in the derivative of the network, which can lead to visual artifacts as we'll see later. For that reason we also explore *soft* thermometer encoding, which has a very similar formula but uses the tanh function instead of the clip function to limit the range of each input to $[-1, 1]$:

$$x_k = \tanh(2(xn - k) - 1)$$



$[0.426, -0.969, 0.809, -0.063, -0.729]$

Figure 3.1: An example auxiliary encoding of $x = 0.72$ with granularity $n = 5$ using the DCT method.

For a similar reason we do not use regular polar coordinates, but ones that go from 0 to π in the positive quadrants and then back down from π to 0 in the negative quadrants to prevent a discontinuous jump from 2π to 0. At least this is the case conceptually, these have been normalized to go from -1 to 1 like everything else. Unfortunately this still creates a point discontinuity at the origin.

Finally we use an input coding technique that is inspired by JPEG [Wal91]. A fundamental theorem of signal processing is that all signals can be expressed as an (infinite) sum of sinusoidal functions, found by the Fourier transform. Interestingly, there are also discrete versions of this theorem one of which is known as the Discrete Cosine Transform [Str99] (DCT). It transforms n evenly spaced real samples into a series of n coefficients of cosine functions with different periods such that their sum reconstructs the original signal.

JPEG uses the DCT to reduce the amount of information needed to encode the image while accepting some loss. The main idea is that human perception is focused on low-frequency signals and does not notice noise in the high frequency domain. Roughly speaking, JPEG stores the image in the frequency domain, keeping only values up to a certain frequency while leaving out anything higher. With that in mind the idea is to provide a neural network with n additional inputs that form the first n cosine functions of a DCT:

$$x_k = \cos((k + \frac{1}{2})\pi x)$$

3.3 Lossless compression

3.3.1 FLIF

FLIF [SW16] is a recent new image format that provides state-of-the art lossless compression rates. Our lossless compression method modifies the internals of FLIF to create a variant that improves compression rates further. We will not go through the exact mechanics of how FLIF works, but provide a quick overview.

Firstly, FLIF encodes one pixel at a time in a serial manner. To be more precise, it encodes one pixel *component* at a time. Each component lives in a *plane*. Most image formats in use work with the RGB color space, but

1	8	6	8	4	8	6	8	2	8	6	8	4	8
7	8	7	8	7	**8**	**7**	**8**	**7**	**8**	7	8	7	8
5	8	6	8	5	**8**	**6**	?	**5**		6		5	
7		7		7		**7**		**7**		7		7	
3		6		4		6		3		6		4	

Figure 3.2: A partial decoding of a 14×10 picture, illustrating the order in which pixels get decoded. Highlighted in bold is a small context example of pixels that could be used to predict the unknown pixel indicated by the question mark. Figure from the FLIF author's slides from their talk at ICIP 2016.

FLIF does a lossless conversion from that to the YCoCg [MS03] color space, which consist of three components Y, Co and Cg corresponding to brightness, chrominance orange and chrominance green. Doing this conversion reduces the correlation between the channels leading to better compression.

The order in which FLIF encodes pixels is not a simple top-to-bottom left-to-right order. Instead, it encodes the image in *progressive* layers, called *zoomlevels*. The first layer contains a single pixel: the top left pixel. Then each layer after that either the number rows doubles or the number of columns doubles. In Figure 3.2 we show an example where it also becomes clear what benefit this method has: when predicting a pixel's value we can use context on all sides of the pixel rather than only to the top and left. More on this in Section 3.3.2.

After we have a prediction of our current pixel based on previously decoded pixel values FLIF encodes $x = e - p$, the difference between the predicted value and the actual value. The way this is done is what the authors claim is the main novel portion of FLIF: Meta-Adaptive Near-zero Integer Arithmetic Coding (MANIAC). If the predictions of pixel values are good, we'd expect their differences to be near zero or zero most of the time.

MANIAC is based on arithmetic binary coding, which we won't explore further here. We'll view it as a black box that you can repeatedly feed a data bit to encode plus an expected probability that bit is 1 and it will output a (near) optimal compressed bitstream with maximal entropy (each bit has a 50% chance of being 0 or 1). As long as the probabilities FLIF provides for each bit are close to the true probabilities for each bit to be 1, compression will be effective.

When encoding an integer $x = e - p$, FLIF first outputs a bit indicating whether x is 0. If it is not, then it outputs a bit indicating whether x is positive or negative, followed by the number of bits are needed to encode the value of $|x|$, $n = \lceil \log_2 |x| \rceil$. n is encoded in unary, by outputting n 1 bits followed by a 0 bit. Finally $|x|$ itself is encoded, omitting its leading 1 bit (as its implied).

For each of those bits outputted, FLIF uses tree learning based on pixel context features (using features similar but not identical to the predictors) to dynamically learn the probabilities for each bit to be 1 as the image is encoded or decoded. Both encoder and decoder dynamically construct the same decision tree in the same manner as the image is decoded/encoded, so it does not have to be encoded itself into the image file.

3.3.2 Predicting based on context

In Figure 3.2 we can see that during decoding we can have a 'context' of previously decoded pixels around the current pixel we're trying to predict. We can use this information to build a model to predict it. In default FLIF this is done by an ensemble of three weak predictors:

```
avg = (top + bottom) >> 1;
topleftgradient = left + top - topleft;
median = median3(avg, topleftgradient, left + bottom - bottomleft);

if (predictor == 0) guess = avg;
else if (predictor == 1) guess = median;
else guess = median3(top,bottom,left);
```

Here `top`, `bottom`, etc, are the values surrounding the current pixel.

The FLIF encoder actually uses each predictor to predict a full plane for a zoomlevel before deciding which predictor to use. The predictor with the lowest expected total bit cost for predicting the pixels in this plane ($\sum_i \lceil \log_2 |x_i - p_i| \rceil$) is chosen.

The core of our lossless method is adding a new predictor to FLIF that uses a neural network to generate a prediction based on the context. Unlike our lossy compression method, this uses a single neural network trained on various images that is considered part of the file format. The size of the neural network is thus not included in the file size of the image, as it's assumed the user has received a copy of the neural network when they installed the image compression software.

For our experiments we used a 5×5 grid centered on the pixel to be predicted in the current plane and zoomlevel as our context. However not all pixels in this 5×5 grid might be available. Initially we also had a secondary boolean feature that indicates whether this pixel exists for each pixel, but this solution ended up performing worse than simply assigning zero to these inputs.

In addition to this, we have a feature indicating whether this layer is a horizontal or vertical expanding layer and one feature each indicating the zoomlevel and plane. All features are normalized to a $[-1, 1]$ range, and the output labels are normalized to a $[0, 1]$ range.

In order to appropriately learn our task we want our optimization metric to match (as closely as possible) our evaluation metric. While we obviously can not possibly include the entire image compression algorithm in our metric, in this case we can optimize exactly for what FLIF uses to determine the best predictor: the binary logarithm of the difference between the predicted and actual value.

Now the issue with this is that this loss metric can be negative, and infinitely negative when $x_i = x_p$, so what gives? The crucial missing piece of information is that FLIF always works with integer samples x_i, x_p rather than real values. The prediction range never exceeds 2^9 samples, so a prediction error of less than 2^{-9} is effectively a perfect prediction after rounding to an integer sample. With that in mind we can construct our loss function, the mean b-bit log loss, using our continuous output domain of $[0, 1]$:

$$\text{MLL}_b = b + \frac{1}{n}\sum_{i=1}^{n}\log_2\left(\max(2^{-b}, |y_i - \hat{y}_i|)\right)$$

It expresses on average how many bits it would take to encode the difference between a prediction and its actual value, assuming that the output domain consists of 2^b discrete samples.

3.3.3 Deterministic cross-platform neural network evaluation

In lossless (de)compression the accuracy of the employed method must be perfect, down to the last bit. Any form of loss will eventually compound as files get re-encoded to new formats (for space but also compatibility reasons) over the years, defeating the purpose of lossless methods. For archiving purposes it is critical that lossless methods really preserve all data.

This may seem simple at first, however it is not trivial to make numerical methods 100% bit for bit reproducible across platforms and architectures. In particular floating point operations are essentially off the table. Not only do standard implementations vary for more involved operations such as trigonometry functions, even basic operations such as a summation are so hard to make consistent a numerical method achieving such a feat is a publishable effort [DN13].

This is an issue, as virtually all neural network libraries and methods use floating point arithmetic throughout. To solve this we implemented a custom neural network evaluator in C++ using Q10.22 [Obe07] fixed-point arithmetic. This means we have 10 bits of integer data and 22 fractional bits. This exactly fits in a signed 32-bit integer.

To convert a floating point number into Q10.22 you simply multiply it by 2^{22} and truncate to a 32-bit signed integer. To convert back you simply convert back to floating point and divide by 2^{22}.

Once a number is in Q10.22 fixed-point format we can approximate real number arithmetic using integer operations. Addition and subtraction directly map to their integer counterparts. To see that this works, let us generalize for a fractional bit size m, here we have $m = 2^{22}$:

$$\frac{\lfloor mx \rfloor + \lfloor my \rfloor}{m} \approx x + y$$

with a maximum error e:

$$e = \left| x + y - \frac{\lfloor mx \rfloor + \lfloor my \rfloor}{m} \right|$$

$$me = |mx - \lfloor mx \rfloor + my - \lfloor my \rfloor|$$

Since $f(x) = x - \lfloor x \rfloor$ has a maximum value of 1, we can conclude:

$$
\begin{aligned}
me &\leq |1 + 1| \\
e &\leq \frac{2}{m}
\end{aligned}
$$

Thus in our case each addition can introduce an absolute error of at most 2^{-21}, which is more than acceptable precision. The proof for subtraction is analogous.

Multiplication is a bit more complicated, but not overly so. If we were to multiply x and y in Q10.22 form directly, the end result would approximate $\lfloor mx \rfloor \cdot \lfloor my \rfloor \approx m^2xy$. But since our numbers should have the form mx we need to correct this extra m factor by doing an integer division. Since our m is a power of two, this can be done very quickly using a bit shift. To prevent overflow of the intermediate product we do require a $32 \times 32 \to 64$ bit multiplication operator. We have:

$$
\frac{1}{m} \left\lfloor \frac{\lfloor mx \rfloor \cdot \lfloor my \rfloor}{m} \right\rfloor \approx x \cdot y
$$

with a maximum error e and WLOG assuming $x, y > 0$:

$$
\begin{aligned}
me &= mxy - \left\lfloor \frac{\lfloor mx \rfloor \cdot \lfloor my \rfloor}{m} \right\rfloor \\
me &\leq 1 + mxy - \frac{\lfloor mx \rfloor \cdot \lfloor my \rfloor}{m} \\
me &\leq 1 + \frac{1}{m} \left(mx \cdot my - \lfloor mx \rfloor \cdot \lfloor my \rfloor \right) \\
me &\leq 1 + \frac{1}{m} \left(mx \cdot my - (mx - 1) \cdot (my - 1) \right) \\
me &\leq x + y + 1 - \frac{1}{m} \\
e &\leq \frac{x + y + 1}{m}
\end{aligned}
$$

Which means for reasonable weights and values (an absolute value of less than 10) we can expect in our networks the absolute error to be no worse than 5×10^{-6}. Of course errors propagate and accumulate, so a more detailed analysis would be preferable, but we have found that the network's performance in practice is not significantly impacted by performing inference using fixed-point arithmetic.

There is still the issue of overflow which we've been ignoring in the entire above analysis, but we simply assume with 10 bits of integer data (giving a domain of approximately $[-512, 512]$) that an overflow will never occur during the evaluation of our neural networks. If this assumption is violated and an overflow does occur nothing goes terribly wrong though - we just output a poor prediction for that pixel.

Finally, while doing other common mathematical operations in fixed point is all possible (e.g. trigonometry, logarithms, square roots, etc), the algorithms are slow and involved. For this reason, and to reduce implementation effort (as every operation must be programmed from scratch in C++), we have chosen to keep the features used in the neural network simple. Our activation function of choice is therefore the leaky ReLU [XWCL15] with $\alpha = 0.125$ which can be implemented in fixed-point arithmetic as such:

```
int32_t fp_lrelu(int32_t x) {
    if (x > 0) return x;
    return x / 8; // Compiles down to arithmetic shift.
}
```

The reason this ends up being so simple is that zero remains zero in fixed-point arithmetic and that multiplying or dividing by a constant also remains that way (since $\lfloor mx \rfloor \cdot c \cdot \frac{1}{m} \approx cx$).

Chapter 4

Experimental setup

We have trained (and where relevant, evaluated) all neural networks using the Keras library version 2.2.4 with the Tensorflow backend. If some hyperparameter for training was not mentioned we have left this settting on the default of Keras.

Early on in the project we chose the optimizer ADADELTA [Zei12] and a minibatch size of 2048 with some simple experiments and we found that these choices were adequate for all network shapes for lossy compression, thus we kept these options constant.

For our lossless compression method ADADELTA often diverged resulting in no training at all, probably due to issues with our unorthodox loss function. The simpler ADAGRAD [DHS11] did not have this issue. Additionally we found a smaller batch size of 128 compared to our previous 2048 to be critical for sustained learning, as the bigger batch sizes started hitting a noise floor way earlier in training.

4.1 Lossy compression

With lossy compression we wish to minimize the amount of data needed to store an image as much as possible, while accepting a loss of image quality. This means that unlike in lossless compression, our optimization domain is two-dimensional, and no single optimal solution exists. Instead, there is a spectrum of solutions forming a Pareto front of solutions that can all be considered optimal depending on how you value the quality/space trade-off.

To limit the scope and to simplify our optimization we focus our final evaluation and optimization on images that are 4 KiB in size, aiming to achieve the highest quality possible at that size. However a big question remains: what is 'image quality'?

Image quality is not an inherent property of an image, even if we sometimes describe it that way. When we say that an image is 'low quality' we are implicitly referring to a reference image which would have no artifacts or distortions at all, even if that reference image may not exist (such as a camera directly converting an image into a lossy format). However because we are evaluating a compression method we always have an unambiguous and available reference image: the original image before compression, thus we can use *full reference methods*.

Fundamentally image quality is a subjective metric and is in the eye of the beholder. Thus for a true evaluation we must survey our target audience for each pair of images to determine which is preferred. Needless to say this is impractical.

Instead we use an automated metric to evaluate image quality. We chose the structural similarity index [WBSS04] (SSIM) as it is a respected and commonly used metric, with the original paper having over 20,000 citations as of 2019. We chose an implementation by Kornel Lesiński [Les19].

It uses the SSIM algorithm at multiple weighted resolutions in the CIE L*a*b* color space and returns $\frac{1}{\text{SSIM}} - 1$ as a score (also known as DSSIM for **dis**similarity), meaning 0 is a perfect score (no difference between reference and target) and any positive score is progressively worse without bound.

Unfortunately, our training metric and evaluation metric are not the same. During the training of the neural networks for the lossy compression method we used the mean sum of squared errors (MSE) along all three color channels as our metric. DSSIM is a global metric across all pixels (or all training samples in this case) and thus does not work with mini-batch learning, which would make training impractically slow. DSSIM is also relatively complex and it's unclear if it would be differentiable. Fortunately low loss in the MSE error generally correlates with a lower DSSIM score in our experiments.

4.1.1 A distribution for weights and biases

As a final step to finish encoding an image to a file we need to quantize the network parameters (see Section 3.2.2). As we wish to use equal frequency binning, we want to know the distribution of the parameters in our trained neural networks.

To that end we trained a network for each image in the True Color Kodak image test set [Cor19]. We used a simple configuration of four hidden layers of size 12 using the DCT method with a granularity of $n = 27$ that was found as working reasonably well using ad-hoc preliminary optimization without applying quantization.

After training each network we extracted all parameters from the network and saved them. Then these parameters were plotted and a distribution manually fitted to them, see Section 5.1. After this experiment the resulting quantization table was used for all other experiments.

4.1.2 Auxiliary input encoding

We performed an experiment how well each auxiliary input encoding method from Section 3.2.3 learned to approximate a 386×320 image of van Gogh's *Starry Night*. All tests were done using the same small (sum of weights ≈ 1 KiB) neural network configuration of Section 4.1.1, with the exception of the control (no auxiliary input encoding) which received an additional layer of neurons to make up for the lost extra connections for having less inputs.

4.1.3 Network shape and evaluation

We compare our method against MozJPEG 3.3 [Moz19] using the Kodak image test set. We target the 4 KiB file size for our optimization, and for that reason we scale the entire test set down 50% to 384×256 pixels thumbnails (all images in the test set have the same dimensions, although some are vertical), as JPEG does not perform well for larger images at tiny file sizes.

But before we can compare we must find a neural network configuration that performs well at this file size. There are two parameters to tune, the granularity n of the auxiliary inputs and shape of the neural network.

To find the best performing network we focused on the same single image (as training takes a long time) as in the last section and did a grid search for the above parameters. We picked the granularity as a value out of $n \in \{10, 30, 50, 70, 90\}$, the number of hidden layers out of $\{1, 3, 5\}$ and the distribution of neurons per layer as one out of 'flat' (all hidden layers have the same number of neurons) or 'tapered' (each hidden layer has 2/3rd of the neurons as the one before it). Since the number of bytes for the image is fixed, the number of neurons we can store is also fixed. We have $2n + 2$ (the original x, y and n auxiliary inputs for each) inputs and 3 outputs, a relation between the number of neurons in each layer and the total number of layers. This is enough information to fully deduce the network shape assuming we always want to use as many neurons as we're allowed to. Each configuration got 200 epochs of training after which it must output an image. After this grid search we did a second grid search to fine-tune the network shape and granularity with 500 epochs of training.

In our final evaluation we upped the training time to 1000 epochs and compressed each image in the test set using our method. We also repeatedly encoded each image using MozJPEG's quality parameter until the output image was just at or under our target file size of 4 KiB. We then computed the DSSIM score compared to the original reference image for both and compared them.

4.2 Lossless compression

Unlike lossy compression, lossless compression has a very simple, single dimensional goal: lower the compressed file size as much as possible while being able to recover the original data with perfect accuracy. The only metric is file size, and the lower the better.

However, there is still one extra hidden dimension of optimization: *which* images you are compressing. It is trivial to make an optimal compressor which only compresses fully red images - just encoding the width and height would be enough.

While a silly example, fundamentally it is an issue. Compression is all about recognizing patterns in your data and encoding those patterns rather than the original data. But every form of image might contain different patterns. Consider the following list of image subjects: human portraits, company logos, animals in fields, legal contract texts, Super Mario Brothers gameplay, paintings by Piet Mondriaan, Dutch passports, lakes. For every single one of these subjects you could make a dedicated compressor that vastly outperforms a generic compressor. A perfect image compressor would essentially contain a model of the entirety of human culture and behavior, with specialized routines for every plausible image subject.

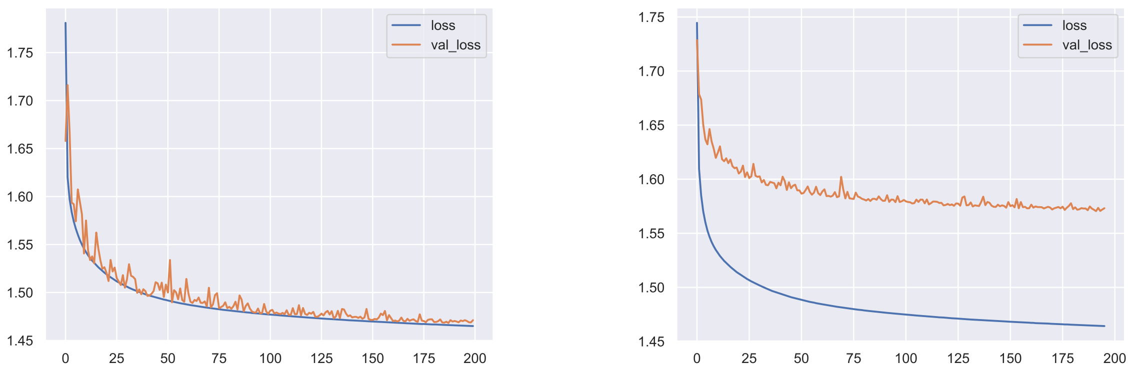
While worrying about a full model of the world is vastly exaggerated, it is not a stretch that any neural network solution trained on images of text might pick up patterns such letter shapes, words or even inter-word connections. Or that an algorithm optimized for artificially created company logos will not perform as well on generic nature photos and vice versa.

With that in mind we limit our training and testing data to the same Kodak image data set used in the lossy experiments, knowing full well that any model generated on it is incomplete compared to a more large-scale varied data set including text, digital art, etc.

To evaluate our changes to FLIF we compare it to unmodified FLIF using both with default settings. Additionally we also compare it to PNG compressed using Ken Silverman's PNGOUT utility which is very slow but gets much smaller files than an ordinary PNG compressor.

Unlike the lossy compression, the lossless compression trains a single model for all images. Thus if we trained on the same images we use to evaluate our model we can not extrapolate this data to other images, as the neural net might remember the original data rather than learn generalized patterns. For that reason we split the Kodak test images into two equally sized sets: one for training and one for evaluation.

Using quick ad-hoc testing of various combinations as a starting point we settled on two hidden layers with 200 and 50 neurons. We let it train for as long as the validation loss improved, however it didn't seem like there was an end in sight as the validation loss closely followed the training loss, despite splitting our available training data into disjoint training and validation samples as seen in Figure 4.1.



(a) Validation set from disjoint samples from same images as training set.

(b) Validation set strictly from different images than training set.

Figure 4.1: The difference between using the same images and different images for training and validation datasets, even when the used samples from the images are disjoint.

The issue is that from the same image many similar or even identical sample contexts plus labels will be extracted. If you then shuffle and split that training data as usual in machine learning you will still end up with very similar distributions in the training and validation data. To combat this we used samples from the

Kodak evaluation images as our validation data. This does mean that we can no longer ethically use the Kodak image set for evaluation, as doing hyperparameter optimization for our validation loss would cause indirect fitting on the evaluation data set. So instead our final evaluation will be done on an entirely unrelated corpus from the FLIF authors they call `wikipedia-photos` [Pho15] which consists of 49 downsampled images of very high resolution redistributable photographs from Wikipedia Commons.

Knowing that overfitting could be an issue, with the same network shape we tested if we could combat it using the regularization methods dropout and batch normalization. Then, with the basic methods and hyperparameters figured out, we found the optimal network shape by testing various shapes in a hand-guided search. Each tested shape was trained for 200 epochs and evaluated.

Finally the best network shape was selected (taking into account prediction runtime performance) and used in our final evaluation.

Chapter 5

Results

5.1 A distribution for weights and biases

For our lossy compression method our input and output nodes have a domain of $[-1, 1]$ and we use tanh as our activation function, which also has an output domain of $[-1, 1]$. This appears to have a side effect that the trained neural networks tend to be 'well-behaved'. After performing the experiment from Section 4.1.1 we ended up with Figure 5.1. The values are centered around 0, and rarely have a magnitude greater than 1.

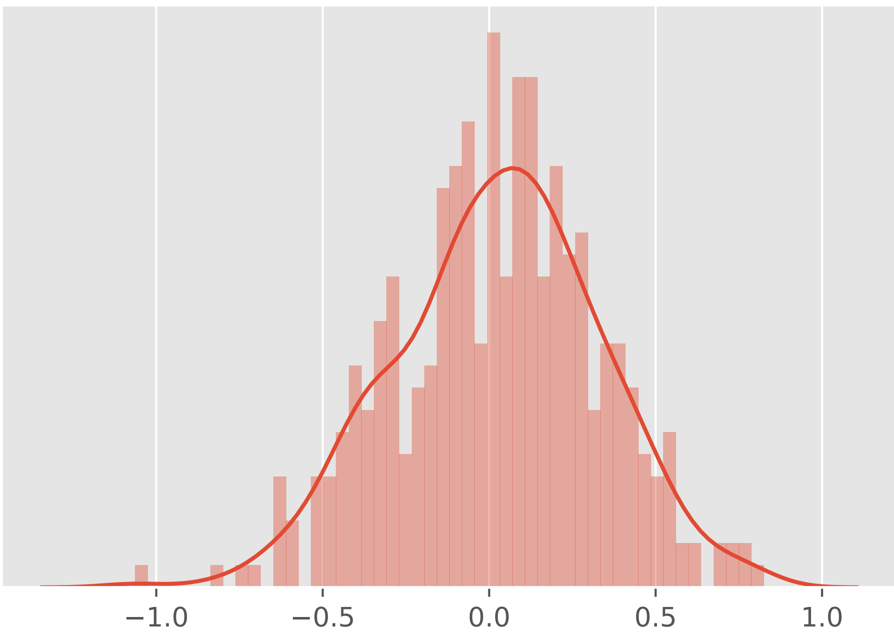


Figure 5.1: Distribution of network weights and biases.

It's a bit awkward that there's an even amount of 8-bit unsigned integers, meaning we either don't have an exact representation of zero (unacceptable), accept two representations of zero (wasted information) or have a slightly different ranges for negative and positive values. We chose the latter option.

After some rough fitting we chose the Gaussian distribution with $\mu = 0$, $\sigma = \frac{1}{2}$ and spread the 8-bit unsigned integers such that integer $i \in \{0, \dots, 255\}$ corresponds to value $\mathrm{CDF}^{-1}\left(\frac{i+1}{258}\right)$. This gives the following possible quantized values:

$$\begin{bmatrix} -1.33134 & -1.21019 & -1.13460 & -1.07849 & -1.03336 & -0.99536 & \\ \dots & -0.00972 & -0.00486 & 0.00000 & 0.00486 & 0.00972 & \dots \\ & 0.96236 & 0.99536 & 1.03336 & 1.07849 & 1.13460 & 1.21019 \end{bmatrix}$$

This is exactly what we want - an exact zero representation, more precision around zero and a range slightly bigger than one.

5.2 Auxiliary input encoding

It is evident from Figure 5.3 that providing auxiliary inputs greatly speeds up learning and leads to a much better end result. The DCT method was by far the best performing method, so our later experiments all used it.

Interestingly the soft thermometer method performed ever so slightly worse than the hard clipping one, although the performance difference is minimal. To our human eyes however the soft thermometer method is a clear winner however, as it causes less jarring artifacts as seen in Figure 5.4. In our eyes this reasserts caution against blind reliance on a metric for computer vision comparison.

We also tested using polar coordinates in addition to the four methods above, however found that it never significantly improved performance, but did create a visual artifact due to its point discontinuity, as seen in Figure 5.2.

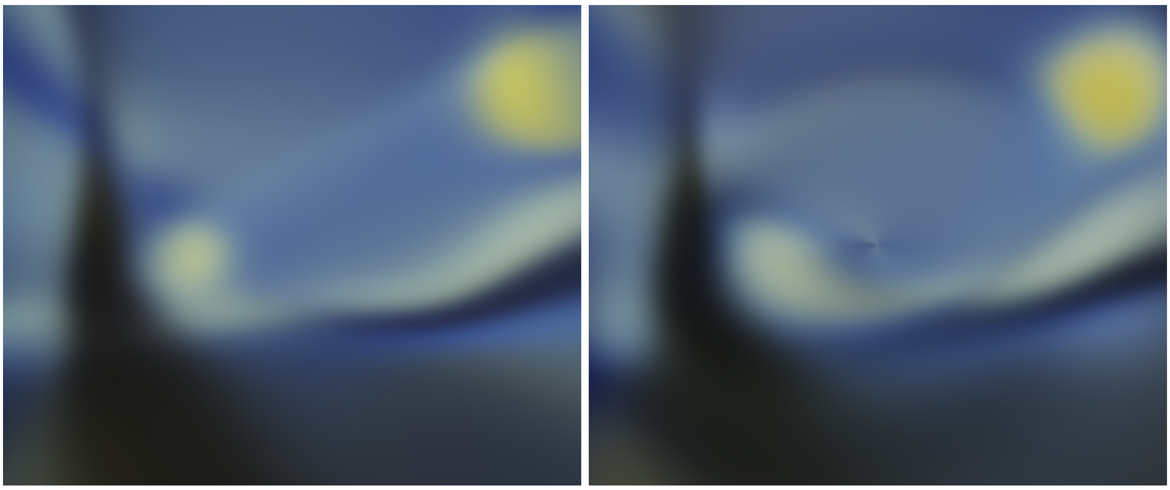


Figure 5.2: Close-up of the control method trained for 1000 epochs respectively without and with polar coordinates as auxiliary inputs.

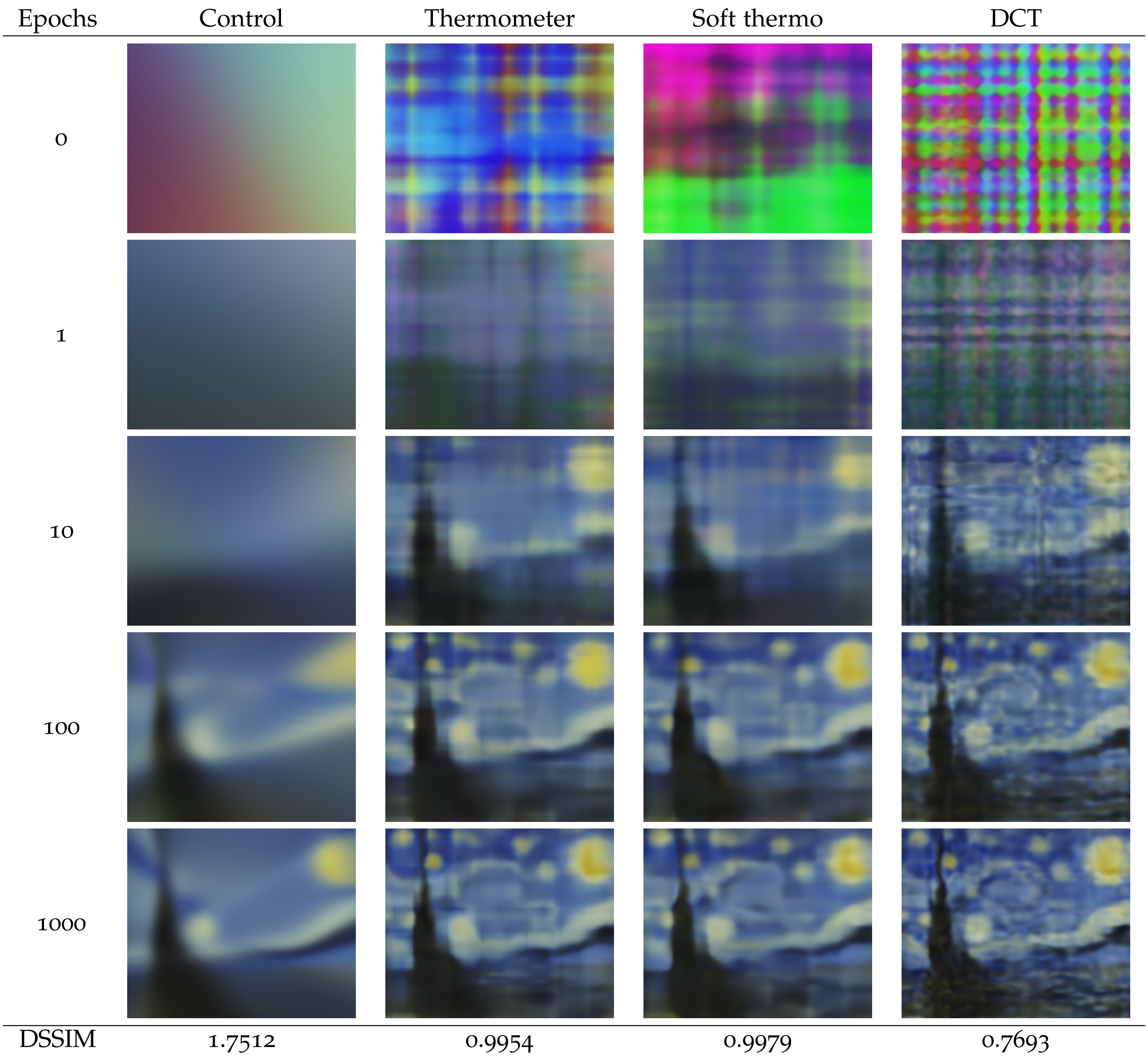


Figure 5.3: Auxiliary input encodings compared. The images listed at zero epochs of training were generated right after the networks were initialized with random weights. They provide an insight in the way the network outputs an image. The listed DSSIM scores are for the final model trained for 1000 epochs.

Figure 5.4: Close-up of the final iteration of the (soft) thermometer methods from Figure 5.3.

n	Tapered	Layers	DSSIM
10	No	1	0.6224
10	No	3	0.4879
10	No	5	0.4442
30	No	1	0.3076
30	No	3	0.3019
30	No	5	0.3001
50	No	1	0.2852
50	No	3	0.2637
50	No	5	0.2803
70	No	1	0.3093
70	No	3	0.2807
70	No	5	0.2904
90	No	1	0.3516
90	No	3	0.2881
90	No	5	0.3048
10	Yes	3	0.4784
10	Yes	5	0.4082
30	Yes	3	0.2784
30	Yes	5	0.2662
50	Yes	3	**0.2399**
50	Yes	5	0.2537
70	Yes	3	0.2624
70	Yes	5	0.2471
90	Yes	3	0.2828
90	Yes	5	0.2808

n	Tapering	Layers	DSSIM
50	1/2	3	0.2264
50	1/2	4	0.2336
50	1/2	5	**0.2243**
50	2/3	3	0.2263
50	2/3	4	0.2284
50	2/3	5	0.2247
50	3/4	3	0.2317
50	3/4	4	0.2318
50	3/4	5	0.2248
60	1/2	3	0.2355
60	1/2	4	0.2251
60	1/2	5	0.2301
60	2/3	3	0.2346
60	2/3	4	0.2329
60	2/3	5	0.2250
60	3/4	3	0.2337
60	3/4	4	0.2254
60	3/4	5	0.2332
70	1/2	3	0.2385
70	1/2	4	0.2456
70	1/2	5	0.2382
70	2/3	3	0.2341
70	2/3	4	0.2324
70	2/3	5	0.2564
70	3/4	3	0.2390
70	3/4	4	0.2318
70	3/4	5	0.2385

Table 5.1: Grid searches to find a good neural network configuration for 4 KiB images.

5.3 Lossy compression topology and evaluation

From the grid search as described in Section 4.1.3 it has become clear that tapering with a granularity of 50 to 70 is desirable as seen in Table 5.1. To do the final fine-tuning we did a second grid search this time limiting the granularity to $[50, 60, 70]$, number of hidden layers to $[3, 4, 5]$ and adding a new parameter, the tapering factor. With possible values $[1/2, 2/3, 3/4]$ it determines how much smaller each hidden layer is compared to the previous one. We also upped the training time to 500 epochs.

From that we find that the best performing configuration is a granularity of $n = 50$ with a tapering factor of $1/2$ and 5 hidden layers. This results in a neural network with hidden layer sizes of $[32, 16, 8, 4, 3]$, with a total number of 4023 network weights/biases. Using this network we evaluated our method, giving Table 5.2.

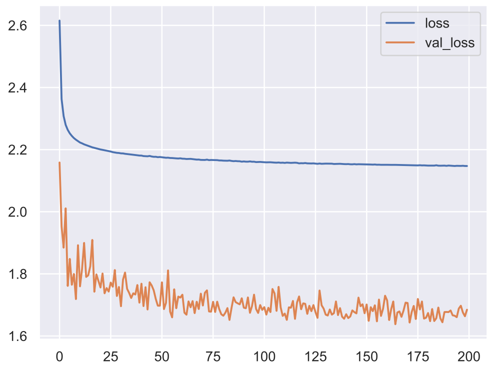
Unfortunately, our lossy compression method gets beaten across the board by JPEG at a target file size of 4 KiB. On a positive note we do notice that our method has roughly twice the DSSIM score as JPEG, trailing it as JPEG's score goes up and down for various different images, with a correlation coefficient of $R^2 = 0.918$. This means that when JPEG finds it easy to accurately encode an image, our method also has an easier time. In other words, it's on the right track. Perhaps with future research and other techniques the gap could be closed.

For a visual inspection most images used in our comparison can be found in Appendix A along with the 4 KiB JPEG competitors and the lossless reference images.

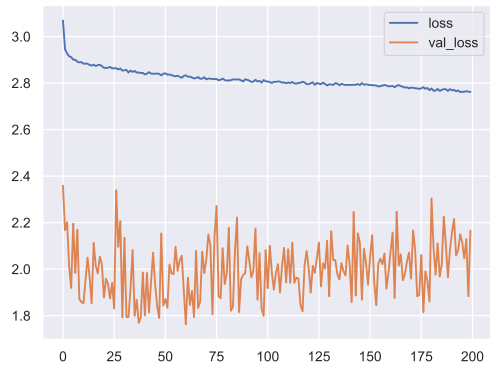
Image	Our method	JPEG	Image	Our method	JPEG
1	0.1851	0.1292	13	0.2910	0.1521
2	0.0610	0.0420	14	0.2120	0.1169
3	0.1012	0.0400	15	0.1094	0.0548
4	0.1042	0.0585	16	0.0786	0.0497
5	0.3551	0.1810	17	0.1194	0.0667
6	0.1426	0.0773	18	0.2541	0.1195
7	0.2043	0.0767	19	0.1566	0.0674
8	0.2844	0.1350	20	0.0740	0.0310
9	0.0941	0.0427	21	0.1196	0.0625
10	0.1199	0.0540	22	0.1575	0.0745
11	0.1259	0.0704	23	0.1025	0.0459
12	0.0693	0.0363	24	0.2194	0.1026

Table 5.2: DSSIM scores on compressing the Kodak test set images to 4 KiB using our lossy compression method and MozJPEG.

5.4 Lossless compression



(a) Training with dropout.



(b) Training with batch normalization.

Figure 5.5: Both dropout and batch normalization increased noise in the validation metric and overall reduced performance.

We have tried to use regularization methods such as dropout and batch normalization after our ReLU (except in the final layer) to reduce the amount of overfitting a network might do, however both methods did more harm than good (see Figure 5.5). They added a lot of noise to the training process and trained substantially slower. The training speed was so slow that an acceptable training time (less than 3 hours for the 12 images in our training set) could not be achieved such that the final model would be an improvement over one without these regularization methods. We conjecture that their questionable benefit in this case is due to our input size ($\approx$ 5.6 million samples from the 12 images) vastly outmatching our network size although it's possible batch normalization doesn't play nice with our loss function.

We found that two hidden layers was optimal (adding more barely increased performance and made training a lot slower), and kept increasing the size of the network until the network started overfitting and validation performance started reducing. However that never happened with the sizes we tested, instead we hit another soft limit: processing speed. The amount of training time becomes worse, but more importantly the inference speed also becomes linearly worse as the number of weights in the neural network increases. Since this neural network has to be evaluated for every single pixel in the decompressor if it is prohibitively slow the compression method is not practical. Considering the results we got we decided that the neural network with

Layers	Weights (×1000)	Loss	Val. loss
[50]	1	1.5949	1.6705
[50, 50]	4	1.5190	1.6146
[50, 50, 50]	7	1.5014	1.6023
[50, 50, 50, 50]	9	1.4937	1.6007
[50, 50, 50, 50, 50]	12	1.4878	1.5967
[100]	3	1.5380	1.6250
[200]	6	1.5070	1.6008
[400]	12	1.7133	1.6161
[100, 50]	8	1.4912	1.5902
[200, 50]	16	1.4636	1.5699
[400, 50]	31	1.4254	1.5422
[100, 100]	13	1.4759	1.5805
[200, 100]	26	1.4510	1.5640
[400, 100]	51	1.4127	1.5384
[800, 100]	103	1.3827	1.5190
[200, 200]	46	1.4290	1.5507
[400, 200]	92	1.3988	1.5322
[800, 200]	183	1.3704	1.5149
*[800, 400, 200]	423	1.3523	1.5150
*[800, 400, 200, 100]	443	1.3332	1.5104

Table 5.3: Differences between various neural network layouts after training 200 epochs. Entries marked with an asterisk were too slow at training and only received 100 epochs of training. The best iteration based on validation loss was chosen for each entry, although this iteration was universally found in the last handful of iterations.

two hidden layers of respectively 400 and 50 neurons was a good trade-off between speed and space savings.

The final benchmark as seen in Table 5.4 shows that our method does compress better than unmodified FLIF. It is never significantly bigger and often a bit smaller for a total 1.24% reduction in file size across the entire corpus. However our method is also roughly 20 times slower than unmodified FLIF, meaning that for most real-world scenarios our method is impractical.

Image	Our	FLIF	PNG	Our	FLIF	PNG
01	1067006	1067006	1537493	100.0%	100.0%	144.1%
02	1216150	1239658	1701731	100.0%	101.9%	139.9%
03	1138620	1143021	1494379	100.0%	100.4%	131.2%
04	1743966	1743965	2256292	100.0%	100.0%	129.4%
05	1726238	1752268	2117213	100.0%	101.5%	122.6%
06	1555800	1568842	2175386	100.0%	100.8%	139.8%
07	1177434	1215689	1917308	100.0%	103.2%	162.8%
08	938693	938692	1311191	100.0%	100.0%	139.7%
09	1161001	1195283	1659647	100.0%	103.0%	142.9%
10	1485529	1485528	1795603	100.0%	100.0%	120.9%
11	1474418	1515417	1819304	100.0%	102.8%	123.4%
12	1647736	1694280	2160786	100.0%	102.8%	131.1%
13	1849817	1849816	2344318	100.0%	100.0%	126.7%
14	1402084	1457176	1752873	100.0%	103.9%	125.0%
15	1302361	1302360	1844464	100.0%	100.0%	141.6%
16	767176	767175	980767	100.0%	100.0%	127.8%
17	1314957	1314956	1663182	100.0%	100.0%	126.5%
18	1688409	1688409	2127549	100.0%	100.0%	126.0%
19	1621096	1619987	1743628	100.1%	100.0%	107.6%
20	1473365	1484647	1926133	100.0%	100.8%	130.7%
21	1788741	1797573	2213937	100.0%	100.5%	123.8%
22	1615206	1623320	1998054	100.0%	100.5%	123.7%
23	1181203	1192436	1521455	100.0%	101.0%	128.8%
24	1324802	1345501	1903409	100.0%	101.6%	143.7%
25	1577536	1641609	2171838	100.0%	104.1%	137.7%
26	1624055	1655052	2102574	100.0%	101.9%	129.5%
27	1600418	1600417	2015306	100.0%	100.0%	125.9%
28	1413228	1431549	1644849	100.0%	101.3%	116.4%
29	1719861	1724670	2057595	100.0%	100.3%	119.6%
30	1486281	1522494	1859526	100.0%	102.4%	125.1%
31	1415198	1415198	1886674	100.0%	100.0%	133.3%
32	1020982	1024288	1327392	100.0%	100.3%	130.0%
33	1242026	1242026	1570824	100.0%	100.0%	126.5%
34	1113707	1132659	1502611	100.0%	101.7%	134.9%
35	1201545	1229893	1612342	100.0%	102.4%	134.2%
36	1441092	1470970	1812512	100.0%	102.1%	125.8%
37	1476776	1480975	1810568	100.0%	100.3%	122.6%
38	807867	807866	1086308	100.0%	100.0%	134.5%
39	1447202	1483288	1943348	100.0%	102.5%	134.3%
40	1245195	1256812	1619469	100.0%	100.9%	130.1%
41	931839	952387	1277331	100.0%	102.2%	137.1%
42	1261508	1288795	1793853	100.0%	102.2%	142.2%
43	1452183	1543604	1959903	100.0%	106.3%	135.0%
44	1258308	1286309	1803513	100.0%	102.2%	143.3%
45	1011012	1011011	1300872	100.0%	100.0%	128.7%
46	1380139	1396750	1684127	100.0%	101.2%	122.0%
47	1355655	1355655	1706395	100.0%	100.0%	125.9%
48	1277410	1291214	1787373	100.0%	101.1%	139.9%
49	1809431	1830284	1903025	100.0%	101.2%	105.2%
Total	67232262	68078780	87206230	100.0%	101.3%	129.7%

Table 5.4: Final lossless benchmark on the FLIF Wikipedia photo corpus [Pho15]. Size given in bytes and relative to best for each image.

Chapter 6

Conclusion and future research

From our lossy method we've seen that a neural network *itself* can be considered a compressed approximation of data by storing its weights, which is a technique that we have not seen before in the context of data compression. While in of itself this technique does not beat JPEG for image compression, we do see that just structure imposed by the neural network manages to compress information at a similar rate of a traditional algorithm. Perhaps with future research of other neural structures or auxiliary input methods the gap could be closed or applications could be found in other compression tasks.

Additionally we've seen that auxiliary input encoding can make a huge difference in learning rates and final model quality. This re-asserts the importance of feature engineering, but even feature synthesis that would in theory not be necessary.

Our lossless work can be considered a proof of concept that FLIF can be improved by introducing a stronger prediction scheme, however it did not improve as much as we'd hoped. There are three avenues for future research in our approach that we see.

The first is simply more data from a more varied corpus. Our method only achieved a 1.24% file size reduction on an entirely unrelated corpus, but on only the validation portion of the Kodak corpus we achieved a 3.13% file size reduction. And these results are using samples from merely 12 sample images. When trained for a very long time on a very large corpus we could likely see these numbers improve.

The second avenue is to use a much larger context than 5×5 and using convolutional neural networks instead of traditional feedforward networks. We expect this plus a larger corpus combined to have a significant positive effect.

Finally, FLIF uses not only the pixel prediction but also simple features based on the pixel context to generate bit probabilities using tree learning. Perhaps these simple features could also be replaced with learned ones using auto-encoders.

Bibliography

[ABC⁺16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[BI18] Paul Andrei Bricman and Radu Tudor Ionescu. Coconet: A deep neural network for mapping pixel coordinates to color values. *CoRR*, abs/1805.11357, 2018.

[C⁺15] François Chollet et al. Keras. `https://keras.io`, 2015.

[Cor19] Kodak Corporation. Kodak Photo CD Photo Sampler. `http://www.cs.albany.edu/~xypan/research/snr/Kodak.html`, 2019. [Online; accessed 01-July-2019].

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[DN13] James Demmel and Hong Diep Nguyen. Fast reproducible floating-point summation. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 163–172. IEEE, 2013.

[HCS⁺17] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, January 2017.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[HTT19] HTTP Archive. Page weight report. `https://httparchive.org/reports/page-weight?start=2016_01_01&end=2019_01_01`, 2019. [Online; accessed 11-August-2019].

[ISO04] ISO/IEC. 15948:2004 – Portable Network Graphics (PNG): Functional specification. Standard, International Organization for Standardization, Geneva, CH, March 2004.

[JTL+17] Feng Jiang, Wen Tao, Shaohui Liu, Jie Ren, Xun Guo, and Debin Zhao. An end-to-end compression framework based on convolutional neural networks. *CoRR*, abs/1708.00838, 2017.

[Les19] Kornel Lesiński. Image similarity comparison simulating human perception (multiscale SSIM in Rust). `https://github.com/kornelski/dssim`, 2019. [Online; accessed 26-June-2019].

[Moz19] Mozilla. Mozilla JPEG Encoder Project. `https://github.com/mozilla/mozjpeg`, 2019. [Online; accessed 01-July-2019].

[MS03] Henrique Malvar and Gary Sullivan. Ycocg-r: A color space with rgb reversibility and low dynamic range. *ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q*, 6, 2003.

[Obe07] Erick L Oberstar. Fixed-point representation & fractional math. *Oberstar Consulting*, 9, 2007.

[Pho15] Various Photographers. FLIF Wikipedia Commons image corpus. `https://github.com/FLIF-hub/benchmarks/tree/cee4aee713671db085649d3b7c35f391a552c13b/test_images/wikipedia-photos`, 2015. [Online; accessed 10-July-2019].

[Str99] Gilbert Strang. The discrete cosine transform. *SIAM Rev.*, 41(1):135–147, March 1999.

[SW16] Jon Sneyers and Pieter Wuille. Flif: Free lossless image format based on maniac compression. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 66–70. IEEE, 2016.

[TVJ+16] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. *CoRR*, abs/1608.05148, 2016.

[Wal91] Gregory K. Wallace. The jpeg still picture compression standard. *Commun. ACM*, 34(4):30–44, April 1991.

[WBSS04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13:600–612, April 2004.

[XWCL15] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

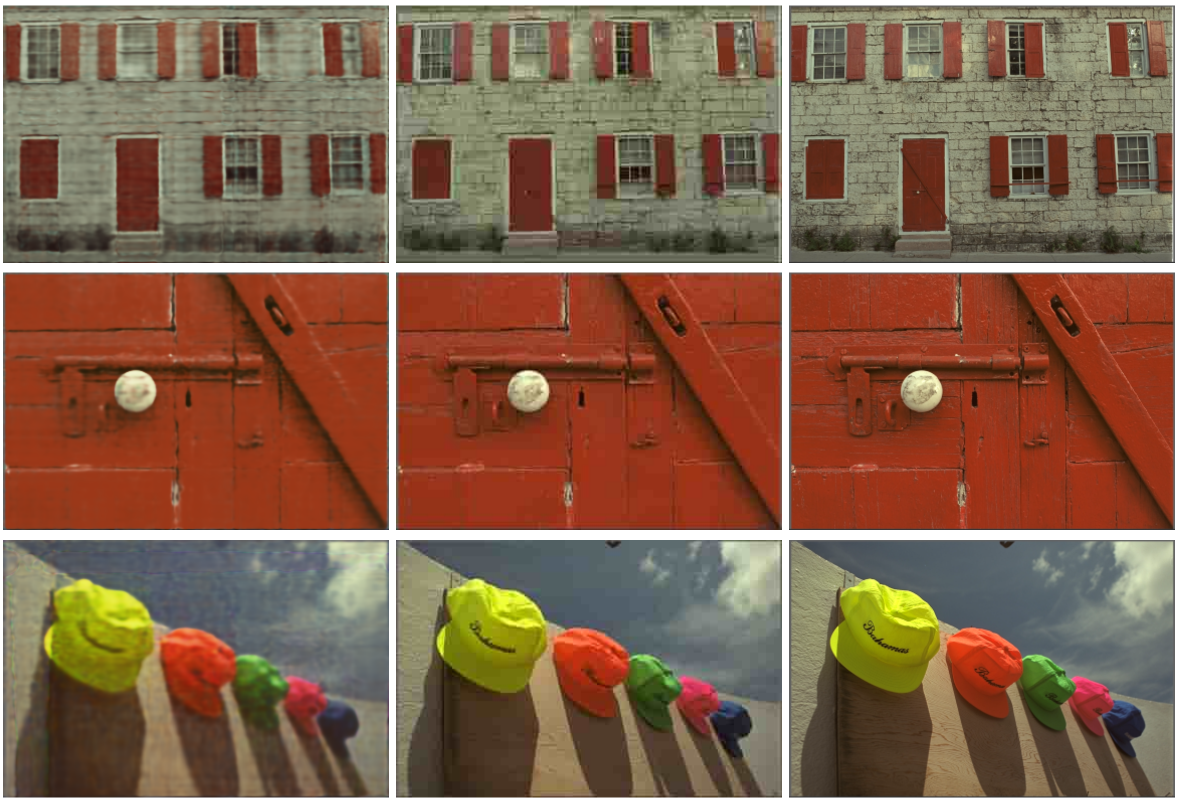
[YC99] Yunho Jeon and Chong-Ho Choi. Thermometer coding for multilayer perceptron learning on continuous mapping problems. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*, volume 3, pages 1685–1690 vol.3, July 1999.

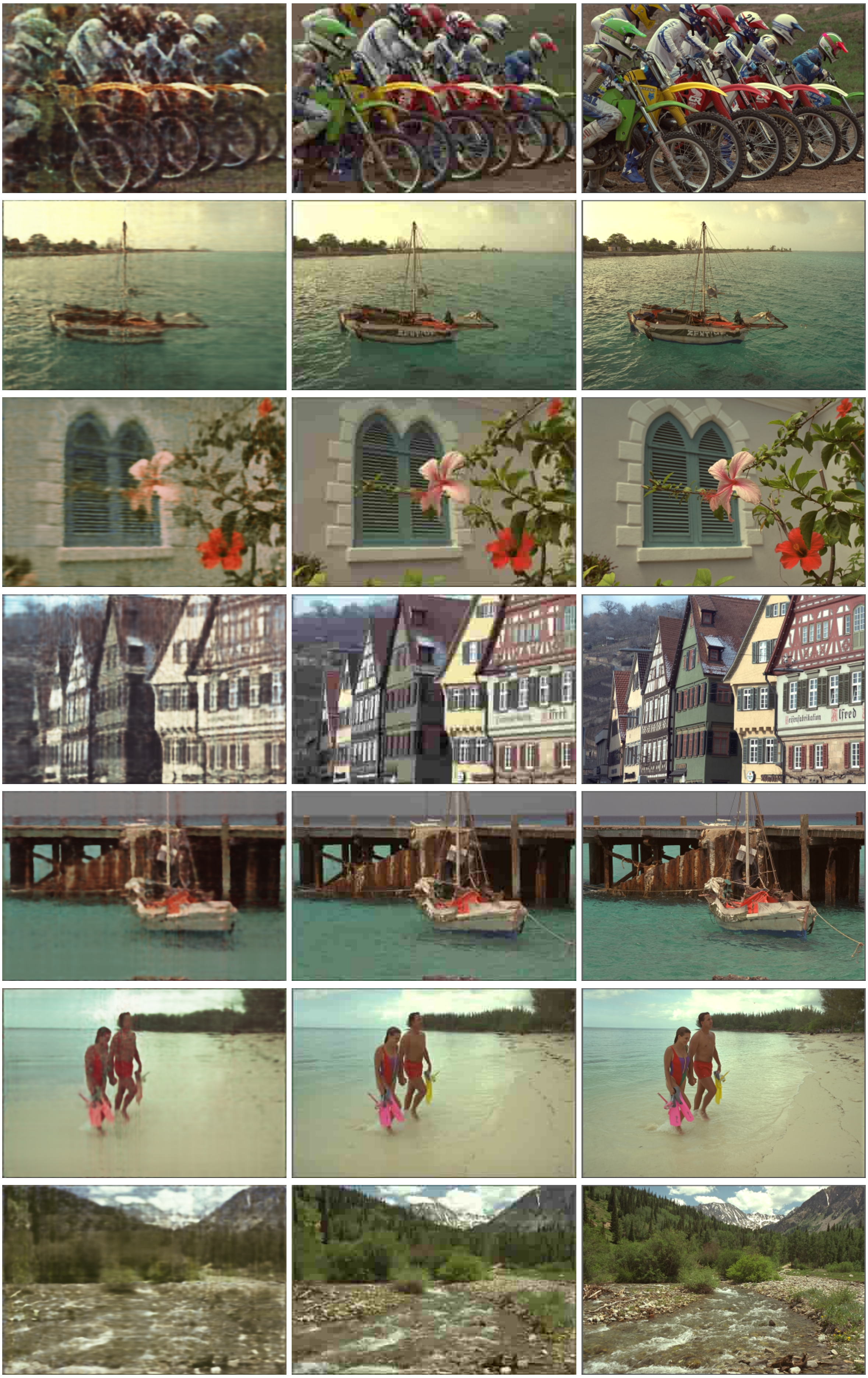
[Zei12] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

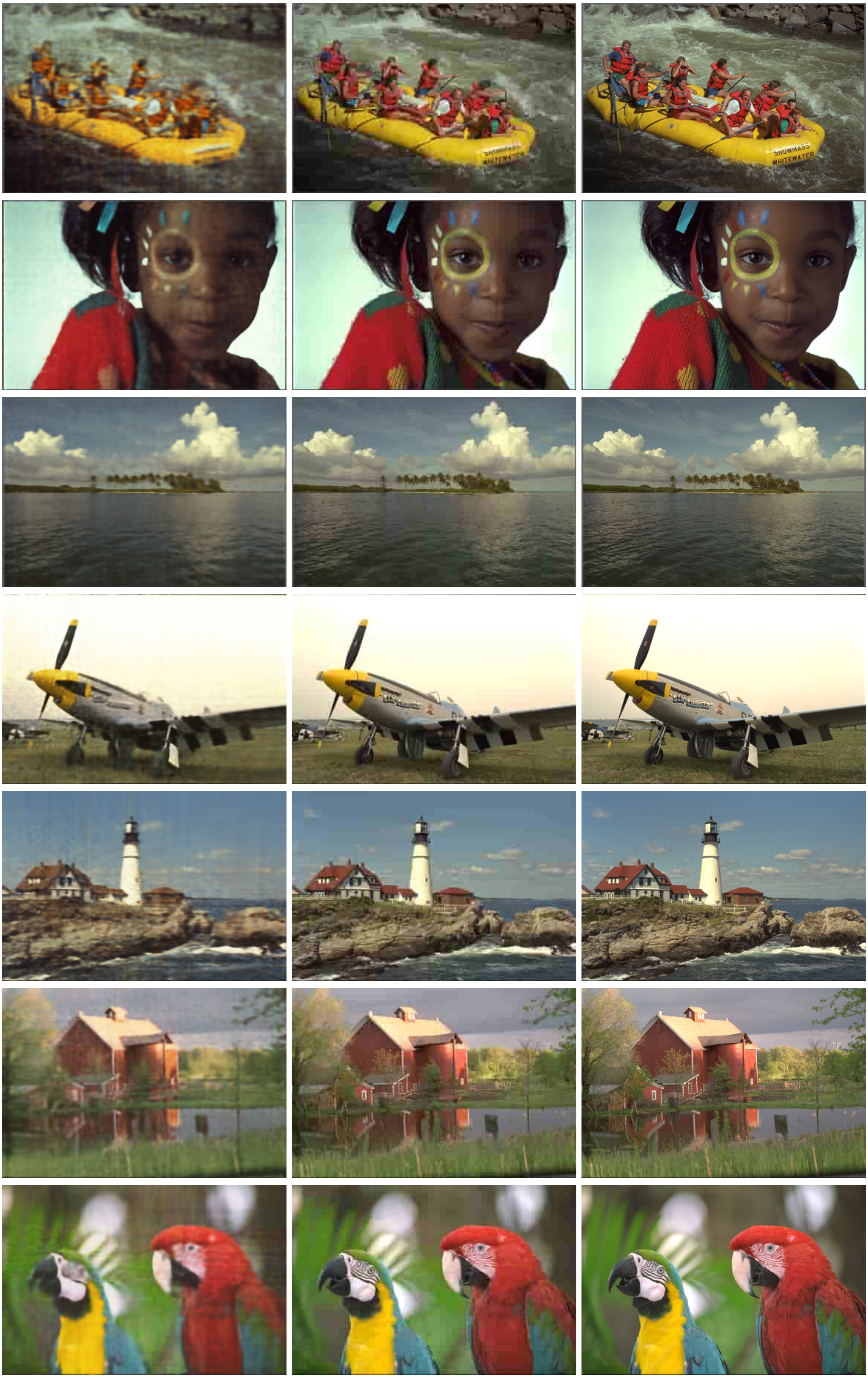
Appendix A

Comparison between our lossy method and JPEG at 4 KiB

These are the images associated with the experiment in Table 5.2. From left to right we see our method, JPEG and the reference image. For space reasons some images were omitted.






SNOWMASS
WHITEWATER