**Faculty of Computer Science** Chair of Computer Graphics and Visualization

**Master's Thesis**

# Improving JPEG Compression with Regression Tree Fields

Nico Schertler
Matriculation No.: 3573429

Submitted to the faculty of Computer Science at the Technische Universität Dresden in partial fulfillment of the requirements for the degree of

Master of Science (M. Sc.)

Supervisor
Prof. Dr. rer. nat. Stefan Gumhold

Dresden, 09.09.2014

## Task Assignment

Structured machine learning approaches like Markov random fields, conditional random fields, or regression tree fields (RTFs) have proven to be very successful in different image processing tasks such as segmentation, restoration, or de-blurring. The aim of this thesis is to analyze how RTFs can be utilized to improve image compression. The basic idea is to integrate the machine learning approach into an existing image codec by adding a prediction step. This prediction step is learned with ground truth computed from the uncompressed image. The idea is exemplarily studied for the JPEG-codec after the DCT transformation. For each DCT coefficient, an 8×8 subsampled coefficient image can be constructed and used to predict other coefficient images. The difference between original coefficient image and predicted one is then passed on to the standard coefficient coding of the JPG-codec. The results shall be compared to the original JPEG-codec as well as JPEG2000.

Obligatory Tasks

- Literature search on the state of the art in machine learning based image compression
- Short introduction to RTFs, available implementations and their usage
- Collection of several image data bases with different type of images
- Extension of an existing JPEG-library with flexible prediction step in encoding and decoding that allows prediction of coefficient images in arbitrary order
- Coupling of extended JPEG-codec with implementation of RTF
- Training of RTF from single image with RTFs of different complexity
- Evaluation of compression performance on variety of grayscale images with different RTF complexities

Optional Tasks

- Optimization of prediction order
- Support for multi-channel images
- Training of RTFs and evaluation of compression results for image collections
- Implementation of a compression scheme for RTFs
- Analysis of the best trade-off between number of training images and complexity of the trained and to be encoded RTFs

## Statement of Authorship

I hereby declare that the thesis *Improving JPEG Compression with Regression Tree Fields,* submitted to the examination board of the Faculty of Computer Science, has been composed by myself and describes my own work unless acknowledged otherwise in the text. I certify that no other than the mentioned sources have been used and that citations have been marked as such.

Dresden, 09.09.2014

Nico Schertler

## Abstract

Storing digital images is a common task in a variety of domains, ranging from family photo albums to medical imaging. Where many images need to be saved, efficient coding is essential. JPEG, as a well-established codec for still-image compression, is used in a variety of use cases. This thesis analyzes improvements on compression performance by introducing a prediction step through regression tree fields. RTFs are used to predict coefficients of JPEG's discrete cosine transform, which allows to encode only differences of small magnitude instead of the original coefficients. The adapted compression scheme's degrees of freedom (e.g. prediction order, RTF loss function, and encoding procedure) are analyzed with respect to their influence on compression performance. This thesis' results show that the general idea of predicting coefficient images can improve the compression performance significantly. Especially loss-specific optimization poses a great gain, which induced the development of an entropy-based loss function. Furthermore, the results are evident that RTF models are not suited for predictions in the frequency domain with respect to compression, which suggests further work with different models and image representations based on the developed scheme.

## Acknowledgements

I would like to express my very great appreciation to Prof. Dr. Stefan Gumhold for supervising this work and for his valuable ideas and comments during development. Advice given by Prof. Ph.D. Carsten Rother on the domain of machine learning has been a great help in understanding basic concepts and getting an overview of related technologies. Special thanks are extended to Dr. Jeremy Jancsary, who provided great support for his RTF implementation, which significantly helped me both use and adapt the code to my needs.

# Contents


## 1 Introduction 3

1.1 Assumptions ... 3
1.2 General encoding and decoding procedure ... 3
1.3 Thesis Structure ... 4

## 2 Related Work 5

2.1 Machine Learning for Still Image Compression ... 5
2.2 Related Tasks in Machine Learning ... 6

## 3 Fundamentals 7

3.1 JPEG Compression ... 7
3.1.1 Color Space Transformation ... 7
3.1.2 Downsampling ... 8
3.1.3 Discrete Cosine Transform ... 8
3.1.4 Quantization ... 10
3.1.5 Entropy coding ... 10
3.1.6 Decoding ... 10
3.2 Regression Tree Fields ... 11
3.2.1 Gaussian Markov Random Fields ... 11
3.2.2 Regression Trees ... 13
3.2.3 Regression Tree Fields ... 14
3.2.4 Available implementations ... 15

## 4 Improving JPEG with RTFs 17

4.1 General Setup ... 18
4.2 RTF Definition ... 18
4.3 Predictive Dependencies of Coefficient Images ... 19
4.4 Quantization and Entropy Coding ... 20
4.4.1 Adapted Quantization and Encoding ... 22
4.4.2 Encoding RTF Models ... 25
4.5 Loss Functions ... 26
4.5.1 Distance Functions ... 27
4.5.2 Entropy Loss ... 28
4.5.3 Comparison of Loss Functions ... 31
4.6 Eigenvalue Bounds ... 32
4.7 RTF Input Representation ... 33
4.8 Prediction Strategy ... 34
4.9 RTF Complexity ... 36
4.10 Compression of Image Data Sets ... 38
4.11 Comparison to RTF De-blocking ... 38

## 5 Implementation Details 39

5.1 Architecture ... 39

5.2 Implementation of Entropy Loss ..... 40
5.3 Selected Design Patterns and Structures ..... 41
5.3.1 Assembling Prediction Strategies ..... 41
5.3.2 Dynamic Serialization Order ..... 42
5.3.3 Updates of the View ..... 42

**6 Conclusions** 45

**References** 47

**List of Figures** 51

# 1 Introduction

The JPEG codec is a compression method for still images that enjoys great popularity since its standardization in 1992 [ITU92]. Due to its nature (more in chapter 3.1) it is best suited for smooth images with continuously changing colors. Although a lossless mode is part of the JPEG standard, most implementations only support lossy compression. Throughout this thesis, JPEG's lossless mode will not be examined further.

Since its development, the JPEG codec has been extended (e.g. by variable quantization [ITU96]), it has been used in derived image formats (e.g. in JPS for stereoscopic images), and improved, which led to further standards (e.g. JPEG2000 [ITU02]). This thesis' aim is to develop a novel compression scheme based on the original JPEG standard which incorporates a machine learning based prediction step during encoding and decoding in order to improve the compression ratio, maintaining image quality.

## 1.1 Assumptions

In designing an appropriate method, some assumptions about the codec environment were made. The entire process is distributed asymmetrically across encoder and decoder, which shall allow online decoding (i.e. in a sufficiently short time span). It is sufficient for the encoder to process offline with no or very weak time limits.

The image quality is measured on a per-pixel difference basis. While most machine learning methods – including the RTF model – are most suited for custom loss measures (e.g. SSIM for human perception [ZBSS04]), this thesis focuses on a difference-based measure to assess general-purpose compression.

## 1.2 General encoding and decoding procedure

The machine learning component is responsible for predicting parts of the image. A part can be a single channel, an output image of the discrete cosine transform, or any other component. Supervised learning is used to infer a model for given input and output (more on this later). Once a prediction model has been learned for an image part, this part does not have to be stored with the image, because it can be predicted from available sources. It is possible to store the quantized differences of original and predicted data to achieve a desired image quality.

In order to reconstruct the predictions, the model has to be available. Throughout this thesis, three different approaches for storing the model will be examined. The first approach involves saving the model directly in the image file (together with meta-data and differences). This is especially useful for single-image compression where a separate model is trained for each image. This might turn out to be inefficient if the trained models are very similar. In that case it is more reasonable to use a single model for several images, which will then be stored in a separate file. The primary use case for this method is the compression of image data sets where items are structurally similar. If this can be achieved successfully, it is possible to pre-calculate models for different image types and distribute them together with the codec. The encoding procedure would then include a classification step which allows to pick the correct model. The model itself does not need to be stored with the image data. It is sufficient to store a unique identifier.

The decoder will successively predict missing image parts, adding the difference information from the image file until the entire data have been restored. The encoder has to be aware of this procedure. If image part B is predicted from part A, and C is predicted from B, then the encoder needs to emulate the decoded part B before training the model for C. The decoded part B might be different from the original part due to the codec's lossy nature. If the encoder did not take this into account, an error would accumulate with every prediction.

The machine learning method which will be used in this thesis is the regression tree field (RTF [JNSR12]), one of the most recent approaches, which combines regression trees and conditional random fields. More information on the details of RTFs can be found in chapter 3.2. The author of the aforementioned paper provides a C++ implementation, which is used for this thesis after adapting it to the concrete needs.

## 1.3 Thesis Structure

The thesis is structured as follows. Chapter 2 introduces related work, especially regarding machine learning for image compression. Theoretical fundamentals of the JPEG codec and related machine learning methods are explained in chapter 3. Based on these fundamentals, chapter 4 step by step develops a compression scheme which incorporates the machine learning step. Various degrees of freedom are analyzed, and the optimal instance for each degree is inferred. Chapter 5 gives a short overview of the implemented application and chapter 6 closes with conclusions and future work.

# 2 Related Work

Data compression is tightly coupled to machine learning. In fact, compression can be regarded equivalent to intelligence [Mah06]. That's why data compression has become a widely used benchmark for general intelligence (e.g. in the Hutter Prize where the goal is to compress an English text [Hut06]). Machine learning methods – as they aim to behave intelligently – are, therefore, strongly related to compression. The following chapter introduces work that has been done in this field so far.

## 2.1 Machine Learning for Still Image Compression

Neural networks are one early form of machine learning. Jiang [Jia99] summarizes various methods for using them for image compression. The approach with the largest similarity to this thesis is the basic back-propagation neural network. For compression, each image is tiled with a fixed block size, and each block is compressed separately. The network consists of three neuron layers. Each neuron of the input layer represents the value of a single pixel. Therefore, this layer has as many neurons as the block has pixels (for single-channel images). The same correspondence is used for the third layer, the output layer. In between both layers a hidden layer is introduced which consists of a variable number of neurons, which are fully connected to each neuron of the input and output layer. The network's weights are trained for a set of training images, which produces a network that can be used to map an image block to itself. Compression is achieved through the hidden layer, which should consist of significantly fewer neurons than the input and output layer. A block is compressed by applying each pixel to the input layer and propagating the values to the hidden layer using the trained weights. The values of the hidden layer's neurons are then encoded with appropriate entropy coding. Decompression is achieved by applying the encoded values to the hidden layer and propagating to the output layer. Besides the values of the hidden layer, the decoder must be aware of the network's weights, which makes this method only applicable for compression of image sets.

More recent work utilizes support vector machines (SVM) to choose a minimum number of support vectors that allow to reconstruct the image. This approach has been researched in both the spatial and frequency domain. All methods use a similar approach. Firstly, the input image is transformed into a different representation (e.g. through DCT). The resulting representation usually consists of a number of coefficients that are used to train a regression SVM. The SVM is defined by its support vectors and their weights. Compression stores only these parameters instead of the original coefficients (significantly less). During decompression, the original coefficients are reconstructed by the SVM.

In [RK03], SVMs are used to reconstruct an image's discrete cosine transform (DCT). The results are superior to JPEG compression, yielding better image quality with a fixed compression ratio or better compression ratio with a fixed image quality. One specialty of this approach, which has also been researched in this thesis for RTF compression, is the preprocessing of the SVM's input. Instead of feeding in the bare DCT coefficients, their sign is left away (making every coefficient positive) and stored separately as a single bit. This preprocess makes the input more homogeneous, resulting in a better SVM performance.

A similar approach has been researched using discrete wavelet transform (DWT) with the haar wavelet by both [Ahm05] and [RYQB05]. The work of [RK03] has been improved in [GCG05] with an adaptive

insensitivity for the SVM, which is more appropriate for the frequency domain. This is based on the fact that some DCT coefficients have a smaller impact on the image quality and can thus be reconstructed with a lower accuracy. Although this resulted in better image quality, [CGGM08] have shown that in general a linear mapping from coefficients to their respective insensitivity does not satisfy statistical independence, suggesting non-linear mappings.

The previous methods are based on JPEG compression. In [FTM12], the use of SVMs in combination with the JPEG2000 codec has been researched. JPEG2000 uses a DWT, which makes this work similar to [Ahm05] and [RYQB05]. However the results outrank existing compression schemes.

## 2.2 Related Tasks in Machine Learning

Apart from the previously mentioned work, machine learning methods are rarely used for image compression. Many use cases aim to improve image quality. Thus, algorithms for those tasks can be seen as some compression because they allow to store the image with a lower quality (and less space) using standard compression schemes. The image's quality is then reconstructed (or at least improved) by post-processing steps.

Super-resolution is the process of improving an image's spatial resolution. Usually, this is applied where hardware requirements do not allow to acquire the image in a higher resolution. However, it could also be used to reduce an image's storage requirements. [KAV05] is an example of solving the super-resolution task using Markov Random Fields (MRF).

Because JPEG compression is known for its block artefacts, another common task for machine learning is de-blocking, which aims to remove these discontinuities. In order to reconstruct the image, a model is required which is aware of the image's structure. In [JNR12], this task was solved using a regression tree field. The results are still available and will be compared to this thesis' results in section 4.11.

When parts of an image have been lost, inpainting algorithms can reconstruct missing data. The primary use case for this technique is the digital processing of analogous material and data which have been retrieved through potentially lossy media (e.g. UDP transmissions of video). RTFs are also suitable for this task as shown in [JNSR12]. It is possible to systematically remove parts from the image in order to reduce its file size. However, the expected reduction is usually not high enough to use this technique as a compression scheme.

# 3 Fundamentals

The following chapter focuses on the theoretical background of involved technologies. A brief overview of the original JPEG codec is given, and regression tree fields are introduced. Variable steps and parameters which will be modified throughout the rest of this thesis will be highlighted accordingly.

## 3.1 JPEG Compression

As stated in chapter 1, this thesis does not consider the lossless specifications of the JPEG codec. Instead, only the lossy compression is analyzed.

The lossy variant of JPEG consists of two parts: the encoder, which converts an RGB image into a JPEG stream, and the decoder, which converts the JPEG stream back into an RGB image. The standard does not specify how this stream is arranged in a file. This is subject to the file format's specification, JPEG File Interchange Format (JFIF) being one of the most popular ones.

In order to compress an image, various steps are executed, which may be either lossy or lossless. Figure 3.1 shows an overview of all involved steps. The following sections introduce each step in more detail.

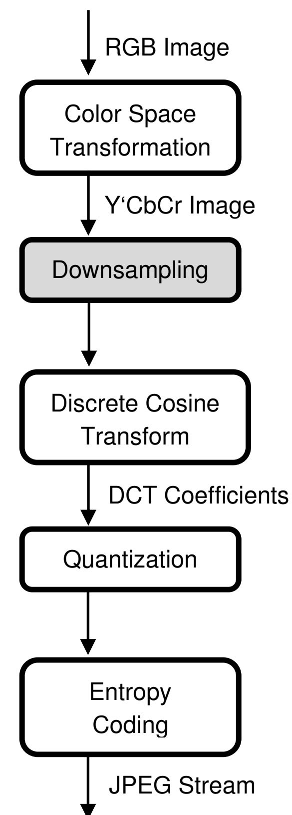



Figure 3.1 Overview of the JPEG encoder

### 3.1.1 Color Space Transformation

JPEG's aim is to compress an image in a way that introduces the least possible distortion in human perception. Therefore, the first step is to transform the image from the artificial RGB format to the Y'CbCr format, which also has three channels. However, they map more closely to human perception than RGB. The Y' channel represents a pixel's luma value, which can be considered to be its brightness. This channel roughly maps to perception by rods in the human eye. The other channels (Cb and Cr) represent chromaticity, which roughly map to cone perception.

For 8-bit RGB data, the conversion from to Y'CbCr can be expressed as a linear mapping [ITU14]:

$$\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 & 0 \\ -0.168736 & -0.331264 & 0.5 & 128 \\ 0.5 & -0.418688 & -0.081312 & 128 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ 1 \end{pmatrix}$$

Once the conversion has taken place, all further steps are executed separately for each channel.

Assuming that these operations can be executed without loss in precision (e.g. due to floating point precision), this step is lossless. Since the precision loss is very small, and the input values are integer, the original RGB values can be retrieved by rounding the back transformation's result without any loss.

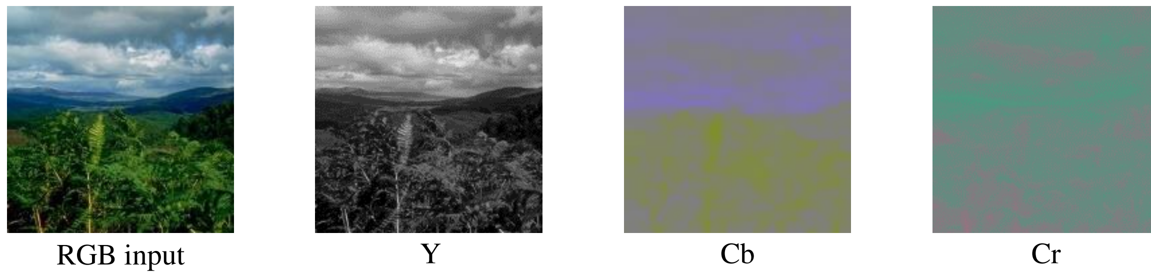



Figure 3.2 Result of color space transformation

### 3.1.2 Downsampling

Downsampling is an optional step in JPEG compression, which was introduced due to the human eye's anatomy. While there are very many rods (perceptors for brightness), the spatial density of cones (perceptors for chromaticity) is significantly smaller. This led to the assumption that there is no need to store an image's chromaticity channels in full size, but only a subsampled representation. There are various sampling schemes, such as 4:4:4 or 4:2:0. The numbers in this notation correspond to the luma sample rate, chroma sample rate in odd rows and chroma sample rate in even rows, respectively [Poy01]. However, because this thesis is focused on general-purpose compression, no downsampling is performed (which is equal to 4:4:4 sampling).

If downsampling is performed (other than the 4:4:4 sampling), then this step is lossy.

### 3.1.3 Discrete Cosine Transform

After re-sampling, the image is split in blocks of 8×8 pixels and transformed into frequency space. This is performed using discrete cosine transform (DCT, [ANR74]).

DCT is a special form of Fourier transform which decomposes a discrete real-valued function $f: \mathbb{N} \rightarrow \mathbb{R}$ into a sum of cosine functions of different frequencies. When the function's domain consists of $N$ elements, the re-composition (inverse DCT) can be expressed as:

$$f(x) = \frac{1}{N} c_0 + \frac{2}{N} \sum_{k=1}^{N-1} c_k * \cos\left[\frac{\pi}{N} k \left(x + \frac{1}{2}\right)\right]$$

The coefficient $c_k$ represents the weight of the $k$-th frequency, where $c_0$ stands for the zero-frequency (constant function). They can be calculated with the forward DCT as:

$$c_k = \sum_{x=0}^{N-1} f(x) \cos\left[\frac{\pi}{N} k \left(x + \frac{1}{2}\right)\right]$$

It can be observed that the first coefficient $c_0$ is the sum of all values, which is basically the unnormalized average. Therefore the zero-frequency coefficient represents the function's average value over its domain.

DCT's 1D-definition can be extended to two dimensions by applying 1D DCT to columns and rows after each other. For JPEG's standard block size of 8×8 pixels, 2D-DCT yields 64 coefficients, which can be used to re-compose the block as:

$$f(x,y) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C_u C_v\, c_{u,v} \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

$$C_u = \begin{cases} \frac{1}{\sqrt{2}} & u = 0 \\ 1 & otherwise \end{cases}$$

$$C_v = \begin{cases} \frac{1}{\sqrt{2}} & v = 0 \\ 1 & otherwise \end{cases}$$

Figure 3.3 shows plots of all 64 base functions (disregarding the normalization with $C_u$ and $C_v$). Before performing the DCT step, the pixels' values are shifted by the half of their range (usually by -128). If the constant scaling by $0.25$ is ignored, DCT yields real-valued coefficients in the range $[64 * (-128), 64 * 127] = [-8192, 8128]$. In the following, the coefficients will not be indexed with a two-dimensional index, but with a one-dimensional which is derived by linearizing the matrix in a row-major order:

$$c_{u,v} = c_{u+8v}$$

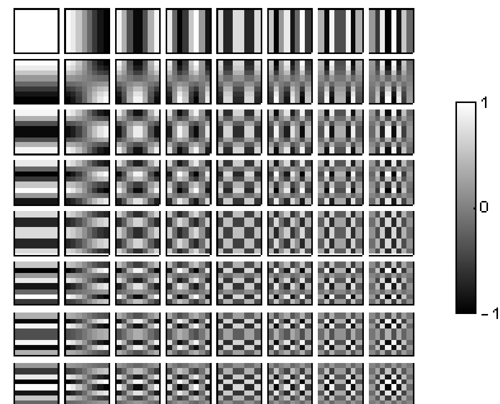



Figure 3.3 2D-DCT's base functions. Horizontal frequencies increasing from left to right; vertical frequencies increasing from top to bottom.

After the DCT step, each block is transformed into 64 coefficients. Thus, the image of one channel is converted into 64 coefficient images with sizes scaled down by 8. If the original image's size is not a multiple of 8, the missing pixels are filled with artificial data, e.g. by extrapolation.

Figure 3.4 shows the result of the DCT step for the previously introduced sample image's Y channel. Each of the smaller images represents the coefficients for one frequency. The top left image shows the coefficients for the zero frequency ($c_0$), whereas the bottom right image shows the maximum vertical and horizontal frequency ($c_{63}$). Different mappings from coefficient value to display color have been used in order to make relevant properties visible.

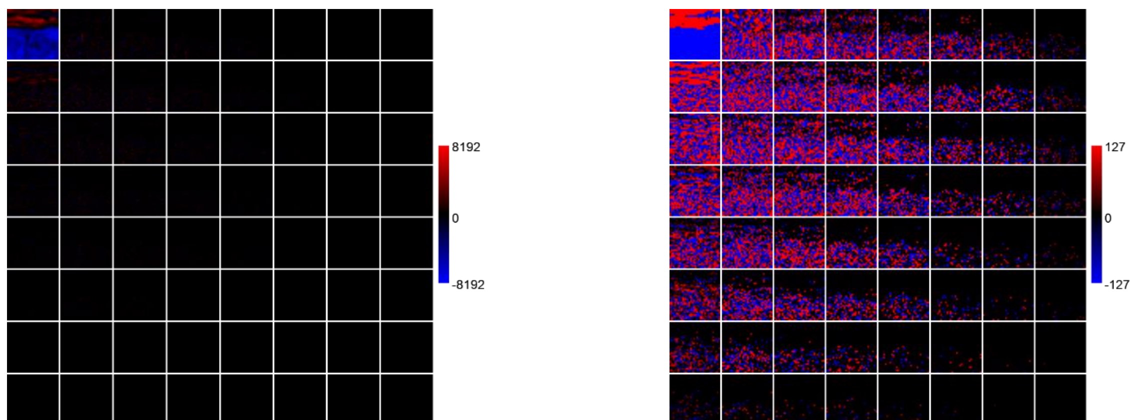



Figure 3.4 The sample image's coefficient images for the Y channel using different display mappings

It can be observed that the coefficient's absolute values decrease as the frequencies that they represent increase. This is caused by the image's overall smoothness, which is one condition for the JPEG codec to be efficient.

Another observation regards the coefficients' signs. While the signs are continuous in the zero-frequency image (because this is the down-scaled input image), all other frequencies do not show this property. There are small connected regions with the same sign, however in general it changes very frequently across the image.

In most implementations, both forward and inverse DCT are implemented efficiently using fast Fourier transform. The inverse DCT yields the original pixel values. Therefore, this step is lossless.

### 3.1.4 Quantization

The main compression step of the JPEG codec is the quantization of DCT coefficients. Each coefficient is scaled down by a constant factor and rounded to an integer representation. Usually, the quantization factors are calculated from the channel (i.e. Y channel quantization factors are different than Cb channel quantization factors) and the coefficient's frequencies (i.e. higher frequencies are quantized more). This results in a lot of coefficients becoming zero or at least equal, which reduces the overall entropy. Due to its nature, quantization is a lossy step. The quantization factors heavily impact compression performance.

This step is the first point at which the RTF-based approach differs from the original JPEG implementation. For one, instead of quantizing the DCT result, differences of predictions and original values are used (more on this later). Secondly, several quantization schemes are examined.

### 3.1.5 Entropy coding

The last step is to encode the quantized coefficients with the least possible amount of bits without further loss. Entropy coding is the tool used for this task. As a preprocessing step, a block's coefficients are re-arranged in a zig-zag order, beginning with low frequencies (with quite high values) and ending with the highest frequencies (with potentially low values due to quantization). This process has the effect that coefficients of similar values are situated next to each other, especially many coefficients with a zero value are at the end of the sequence.

The actual encoding uses a combination of run-length encoding (RLE) and a Huffman code. RLE is used to efficiently encode series of zeroes while the Huffman code is used to minimize the result's redundancy. More recent extensions to the JPEG standard also allow arithmetic coding instead of the Huffman code. However, throughout this thesis, Huffman compression is used.

### 3.1.6 Decoding

In order to decode a JPEG stream into an RGB image, all steps have to be inverted and executed in reverse order.

## 3.2 Regression Tree Fields

The regression tree field is a structural model that evolved from both the Markov random field and regression tree. The following chapter introduces the necessary theoretical background of involved concepts.

### 3.2.1 Gaussian Markov Random Fields

Markov random fields (MRF) can be used to model the probability distribution of random variables $Y_i \in \mathbb{R}^k$ which are associated with a graph's nodes. Figure 3.5 shows such a graph with vertices $V$ and undirected edges $E$.

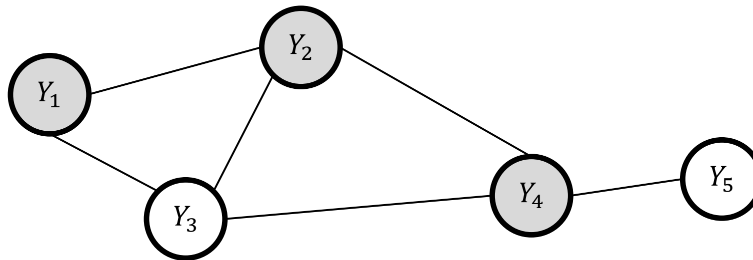



Figure 3.5 Sample graph with five random variables. The Markov blanket of $Y_3$ is highlighted in gray.

In order to represent an MRF, the graph has to fulfill the local Markov property [KS80], that is for the probability density function $\pi$:

$$\pi(Y_i = y_i | Y_k, k \neq i) = \pi(Y_i = y_i | Y_k, (i,k) \in E)$$

This means that a variable is stochastically independent of variables that are not connected by an edge. The set of connected vertices (i.e. the influencing variables) for a given variable $Y_j$ is called its Markov blanket.

If all variables are concatenated into the vector $Y$, the overall probability density may be expressed as a multivariate Gaussian [RH05]:

$$\pi(Y = y) = \sqrt{\frac{|Q|}{(2\pi)^n}} \exp\left(-\frac{1}{2}(y-\mu)^T Q (y-\mu)\right)$$

In the above definition, $n$ represents the number of variables, $Q$ is the positive definite precision matrix with determinant $|Q|$ and $\mu$ is the Gaussian's mean vector. The Markov property forces all entries of the precision matrix that don't correspond to an edge in the according graph to be zero, making the matrix sparse:

$$(i,j) \in E \Leftrightarrow Q_{i,j} \neq 0$$

The Gaussian's covariance matrix $\Sigma = Q^{-1}$ is typically dense.

Another property which is induced by the Markov property is the probability's factorization over maximum cliques. A clique is a fully connected subgraph. Therefore, the maximum cliques of the sample graph above are $cl = \{\{Y_1, Y_2, Y_3\}, \{Y_2, Y_3, Y_4\}, \{Y_4, Y_5\}\}$, yielding:

$$\pi(Y = y) = \prod_{c \in cl} \phi_c(Y_c)$$

The factors $\phi_c$ only depend on a small subset of the graph's variables and can be visualized in a factor graph. Factor graphs introduce a second form of nodes for the factors. Edges are then drawn between a factor and the variables it depends on. The sample graph yields the following factor graph:

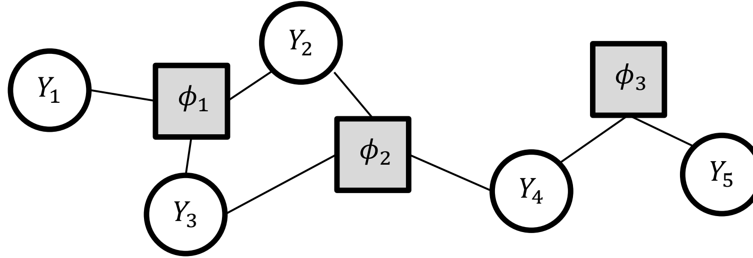



Figure 3.6 Sample factor graph

Usually, not the probability densities are modeled, but energy functions, which are derived from the negative log likelihood:

$$\begin{aligned} &-\log\left(\prod_{c \in cl} \phi_c(Y_c)\right) \\ &= \sum_{c \in cl} \log \frac{1}{\phi_c(Y_c)} \\ &\propto \sum_{c \in cl} \psi_c(Y_c) = E(y) \end{aligned}$$

If the potentials $\psi_c$ are conditioned on an observation $x$, the definition transforms into a conditional random field (CRF) [BKR11]:

$$\begin{aligned} E(y, x) &= \sum_{c \in cl} \psi_c(Y_c, x) \\ \pi(Y = y, x) &= \frac{1}{Z(x)} \exp\big(-E(y, x)\big) \end{aligned}$$

Here, $Z$ is the partition function that normalizes the probability density.

This model can be applied to images by associating each pixel with a separate random variable. Interpreting an input image as the observation $x$, the response image $y$ can be calculated as the probability's maximizer:

$$\begin{aligned} Y &= \arg\max_y \pi(Y = y, x) \\ &= \arg\min_y E(y, x) \end{aligned}$$

Such models can be defined successfully for various tasks such as image de-noising. Unary and pairwise factors can be computed efficiently and suffice in most cases. The image's natural connectivity can be used to derive the factor graph, although every other factor definition is possible as well. Figure 3.7 shows a sample factor graph for the standard 4-connectivity. Observations are included for reasons of completeness. Usually, they are omitted.

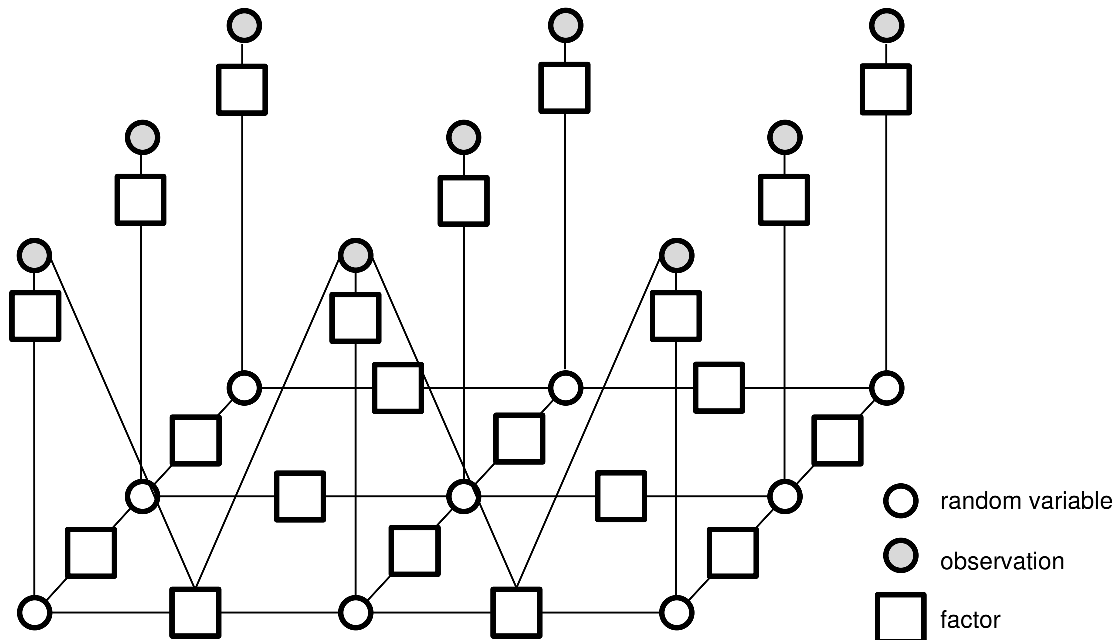



Figure 3.7 Factor graph for natural 4-connectivity of an image. Dependencies of pairwise factors on observations are only visualized for the front-most slice for clarity reasons.

### 3.2.2 Regression Trees

Regression trees [BFOS84] are binary decision trees, which can be used for non-parametric regression of functions with arbitrary domains and ranges $f: D \rightarrow R$. They consist of two node types: inner nodes represent feature tests and leaf nodes represent a result.

The tree's input is an element of the function's domain, which is firstly processed by the root. Feature test nodes have two children and represent functions $feat: D \rightarrow \{true, false\}$. If this function's result for the tree input is $true$, the control flow is passed to the left child, otherwise to the right child. In most cases (also in the implementation used in this thesis), the feature function is a threshold test of a feature response $r: D \rightarrow \mathbb{R}$ and $feat: i \mapsto r(i) \geq t$ for an arbitrary threshold $t \in \mathbb{R}$.

A leaf node contains a single element of $R$. If the control flow reaches a leaf node, its content is returned as $f$'s result. Because the tree has a finite number of leaf nodes, there can only be a finite number of return values, which results in a discretization of $R$, even if the original range is continuous. One way to overcome this limitation is the usage of multiple trees with slightly different parameters, creating a regression forest. Each tree within this forest is evaluated separately and the results are averaged to produce a near-continuous range. However, regression forests are not covered in this thesis.

Regression trees can be learned in a discriminative way by defining pairs of input and desired output. The tree depth significantly influences the regression's quality. A small depth results in few leaf nodes which yield a coarse approximation, while a deep tree with lots of leafs can reconstruct the function more accurately.

### 3.2.3 Regression Tree Fields

Regression tree fields [JNSR12] combine regression trees and conditional random fields as introduced in the preceding sections.

The likelihood of a certain result image $y$ for a given input x is expressed using a quadratic energy

$$E_W(y,x) = \frac{1}{2} y^T \Theta_W(x) y - y^T \theta_W(x)$$

Here, $W$ describes the set of model parameters. This includes, besides others, parameters which define the functions $\Theta_W$ and $\theta_W$. $\Theta_W$ is a function which maps an input to an $m|V| \times m|V|$-dimensional real-valued matrix ($m$ being the number of channels of the target image and $|V|$ being the number of pixels) and $\theta_W$ is a function which produces an $m|V|$-dimensional vector. Throughout this thesis, output images always comprise a single channel, therefore $m = 1$.

As before, this total energy is split into factors, which are grouped into factor types. Every factor is then instantiated from a type by offsetting it by its according pixel's position (the first pixel in the case of pairwise factors). In the example introduced earlier (Figure 3.7), there are three factor types: one unary type, one horizontal and one vertical pairwise type. This results in the following definition for the partial energy $E_{t,f,W}$ of a single factor:

$$E_W(y,x) = \sum_{t \in factor\ types} \sum_{f \in factors_t} E_{t,f,W}(y_{t,f}, x)$$

$$E_{t,f,W}(y_{t,f}, x) = \frac{1}{2} y_{t,f}^T \Theta_{t,W}(x) y_{t,f} - y_{t,f}^T \theta_{t,W}(x)$$

A factor's energy is only evaluated using a subset $y_{t,f}$ of the target image. For unary factors, these are $m$-dimensional vectors which represent a single pixel (the pixel which the factor is instantiated for). $\Theta_{t,W}$ yields an $m \times m$-dimensional matrix, and $\theta_{t,W}$ returns an $m$-dimensional vector. For pairwise factors, the vectors of the two corresponding pixels are concatenated into a $2m$-dimensional vector (which also increases the matrix' dimensionality).

Every factor of the same type shares the same functions $\Theta_{t,W}$ and $\theta_{t,W}$ with the rest of the type's factors. Both functions are represented by a single regression tree $X \rightarrow \mathbb{R}^m \times \mathbb{R}^{m \times m}$ or $X \rightarrow \mathbb{R}^{2m} \times \mathbb{R}^{2m \times 2m}$ for unary and pairwise types, respectively, where $X$ represents the set of input pixels. The set of model parameters $W$ essentially contains the parameters of these trees, including feature tests and leaf values.

The steps to calculate a factor's local energy are therefore:

1. Identify the according input pixel $x$
2. Identify the according factor type $t$
3. Identify the output vector (i.e. by concatenating two pixel vectors for pairwise factors)
4. Propagate $x$ down the $t$-th regression tree to get matrix $\Theta$ and vector $\theta$ (note that these are not functions anymore)
5. Evaluate $E = \frac{1}{2} y^T \Theta y - y^T \theta$

The total energy's minimizer with respect to output vectors Y is then the RTF's response for a given input image $X$. This minimizer can be calculated efficiently using iterative optimization methods, which are beyond the scope of this thesis.

Discriminative training of RTFs combines regression tree learning and parameter optimization. An RTF can be trained to minimize a custom loss-function, where the loss is a function $l: G \times Y \to \mathbb{R}$ which evaluates a prediction's quality with respect to a ground truth $G$. In most cases it is desirable to minimize the distance of predictions and ground truth images (the prediction should match the ground truth as closely as possible), which suggest the usage of distance functions, such as mean squared error (MSE) or mean absolute error (MAE) , as loss functions. However, because parameter optimization uses quasi-Newton methods, the loss function should be continuously differentiable while twice continuous differentiability is desirable.

RTF training is performed iteratively in several rounds, starting with a single root node for each factor type's regression tree and all factor instances are associated with this node. At the beginning of a round, the parameters (matrix $\Theta$ and vector $\theta$) of all leaf nodes are optimized to minimize the loss function. During this process, the eigenvalues of $\Theta$ are bounded by a custom range, which can be used to regularize the RTF (i.e. to prevent over-fitting and to allow the RTF to generalize). Furthermore, this constraint results in the local precision matrices to be positive-definite (cf. section 3.2.1).

The next step in RTF training is the split of leaf nodes. A number of available feature tests are sampled and evaluated with respect to their contribution to the loss function's gradient's norm (again projected in such way that the resulting matrices' eigenvalues are bounded). Assuming that a high gradient norm results in a quick optimization of the overall loss, the feature test which induces the highest projected gradient norm is chosen for the leaf node, making it a feature test node. The node's parameters are copied into its new children, which does not change the RTF's prediction. Higher prediction accuracy is achieved during parameter optimization in the next round. The factors contained in the node are then sorted in either of its children, based on the feature test's response.

This procedure is executed until a maximum tree depth is reached or until there are no more leafs which can be split (e.g. which contain only a single factor). A final parameter optimization finishes the RTF's training.

During training, there are essentially three parameters that can be used to control prediction quality: the loss function, the eigenvalue bounds, and the tree depth. All three parameters will be examined for their most appropriate values for image compression in the next chapters. The set of available feature tests and the number of sampled features also influence the prediction quality. However, these parameters are considered constant.

### 3.2.4 Available implementations

The authors of [JNSR12] provide full C++ code for their RTF implementation [MSR13], which will be used in this thesis. It is structured as a framework with various variation points for the previously introduced parameters, such as loss functions, image data sets, eigenvalue boundaries, etc. Minor adaptations were necessary, which will be explained in detail later.

When all variation points are assigned, the framework's two main functionalities are RTF training (for given pairs of input and ground truth images) and regression / prediction (given an input image).

# 4 Improving JPEG with RTFs

Figure 4.1 shows an overview of the modified codec. While in the original JPEG codec DCT's results go straight to quantization, the modified approach uses them to train an RTF model. This training must be aware of the decoding procedure and adapt input images accordingly. However, during development, this awareness has been removed for fast evaluation. Training RTFs is a very time-consuming task and by changing successive steps (e.g. quantization parameters) already calculated RTF models would be invalidated. Instead, training is considered a stand-alone step which is performed once, knowingly that the results will only approximate the actual codec performance. This allows adaptions of other steps without the need to re-train the model every time some parameters are changed. During development, which aims to find the most appropriate strategy for every step, this is considered a valid simplification. This behavior is consistent for every configuration and is unlikely to change the relative compression performance of different strategies compared to each other (i.e. the best configuration with this simplification is most likely the best configuration without it). When transferring development results into an actual codec, the code has to be changed slightly in order to incorporate the decoder awareness into the encoder.

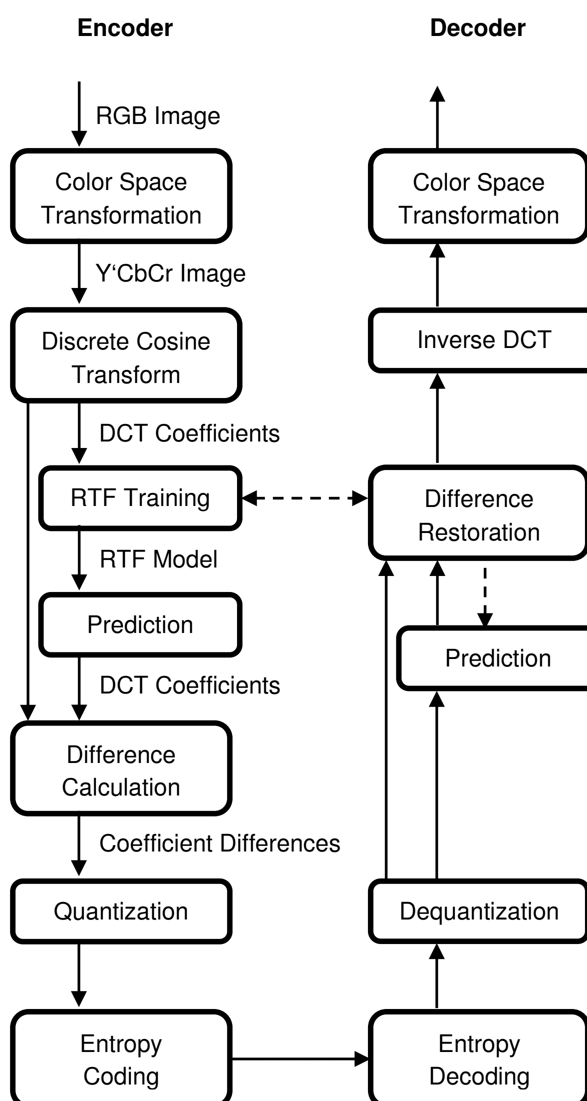



Figure 4.1 Modified codec scheme with incorporated prediction step

The difference calculation's result, which is equal to the prediction step's loss, is then passed on to coding. Because the coefficient differences may have different properties than the original coefficients, the coding methods may need to be adapted to these new properties.

Each of the following sections focuses on a separate step in the above scheme, closing with the resulting optimal strategy. Through this greedy approach, an overall configuration is gained which is considered to have an optimal compression performance in means of decoded image quality and necessary storage size.

## 4.1 General Setup

In order to optimize the compression strategy, a utility program has been developed that allows to exchange steps in the general compression scheme. This program is intended for development purposes and therefore keeps more data in memory than is necessary for compression, which allows a flexible utilization. After an image is loaded, it is converted into a 24 bit RGB representation (if the original data are in a different format) and cropped to a multiple of eight size. This avoids the need of filling in artificial data in the last row and column, allowing to focus on the actual compression. For the first two steps (color space transformation, DCT), code from libjpeg [IJG14] has been extracted and simplified. The remaining steps are custom implementations and can be chosen arbitrarily.

## 4.2 RTF Definition

One very distinctive property of RTFs is the definition of factors. The chosen RTF implementation supports both unary and pairwise factors, but no higher-order factors. In this thesis, their layout are defined by the integer parameter $factor\ type\ count$. Starting with a single unary factor type, each increment of the factor type count augments the factor graph with a pairwise factor type up to a densely 8-connected graph. Figure 4.2 shows this correlation.

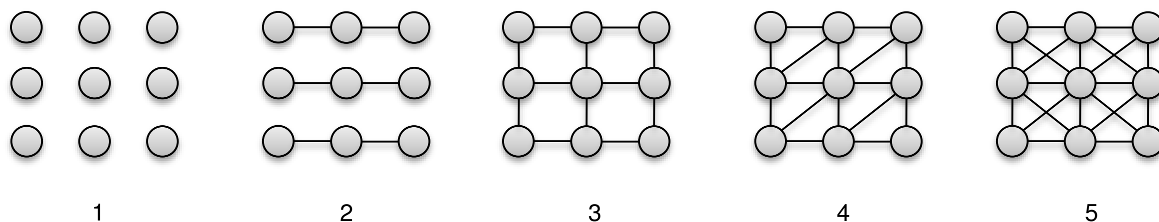



Figure 4.2 Definition of factor types based on the factor type count. Vertices represent pixels (random variables + unary factors). Edges represent pairwise factors.

The regression trees mainly use the features shipped with the RTF code. These are the following:

- Unary feature:
  Returns the value of a single pixel at the coordinate of the factor, translated by the feature's offset vector. Uses the channel specified by feature parameters.
- Pairwise feature:
  Returns the difference of two pixels' values. The first pixel is located at the factor position, translated by an offset. The second pixel is located either at the factor position, translated by a second offset (for unary factors) or at the factor's second pixel location translated by the second offset (for pairwise factors).

Additionally, two features have been implemented which return the factor's x or y coordinate. These two features are only activated in single-image compression because they are not discriminative when processing multiple images. E.g. in an image set with a single motive, viewed from multiple perspectives, the same local structure may appear at several image locations. Therefore, a factor's position is not expressive at all, leading to a loss in generality if used.

RTF training and prediction works best if all pixel values are normalized, i.e. they are in the range $[-1, 1]$ or $[0, 1]$. The process of normalizing input images will be discussed in a subsequent section. For

now it may be assumed that the normalization of pixel value $x$ is achieved by dividing by the greatest absolute value of the entire image:

$$normalize: \mathbb{R} \to \mathbb{R}: x \mapsto \frac{x}{d}$$
$$d = \max_{x \in image} \|x\|$$

## 4.3 Predictive Dependencies of Coefficient Images

When predicting a coefficient image, some images might be more suitable as the prediction's input than others. This can be caused by statistical correlation or higher-order relationships. In order to determine such dependencies, an experiment was conducted where every coefficient image of the Y channel (cf. Figure 3.4) was used to predict every other image. The prediction quality is measured with the peak signal to noise ratio (PSNR) which is defined as follows [ITU01]:

$$\begin{aligned} PSNR\ [dB] &= 10 \log_{10}\left(\frac{max^2}{MSE}\right) \\ &= 20 \log_{10} max - 10 \log_{10} MSE \\ MSE &= \frac{1}{n}\sum_{px}\left(prediction_{px} - original_{px}\right)^2 \end{aligned}$$

Here, $max$ is the range of pixel values. For 8 bit channels, this is usually 255. In the experiment, the actual ranges of original pixel values have been used. If the prediction results in an error of 1 for all pixels of an 8 bit channel, the resulting PSNR is $20 \log_{10}\left(\frac{255}{1}\right) \approx 48.1\ dB$. An average error of 0.5, which is perfectly re-constructible by rounding, yields a PSNR of $20 \log_{10}\left(\frac{255}{0.5^2}\right) \approx 60.2\ dB$. Generally, higher values represent better prediction qualities with a perfect prediction resulting in an infinite PSNR. In the experiment a single RTF with a maximum tree depth of 10 is trained for every prediction. MSE loss is used during training, which automatically also optimizes for PSNR [JNR12]. Figure 4.3 shows the experiment's results.

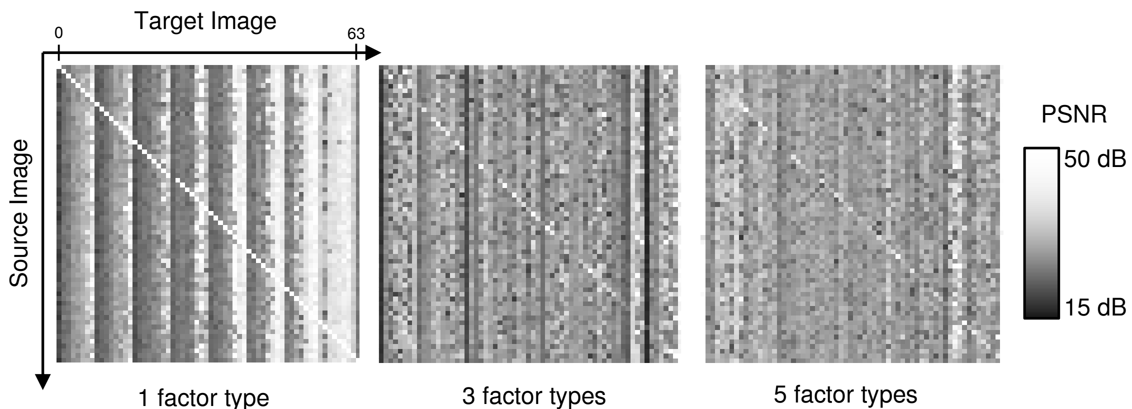



Figure 4.3 Results of the predictive dependencies experiment for different RTF complexities. Every pixel stands for a pair of input coefficient image (vertical axis) and target image (horizontal axis). The pixel color represents the prediction quality.

The first observation that can be made is that some coefficient images can be predicted more easily than others (the according columns are brighter). In the case of a single unary factor, this applies especially to high-frequency coefficients.

The second observation refers to single columns. Most columns produce uniform prediction qualities with only few irregularities. This leads to the conclusion that the choice of the respective prediction source does not (or only slightly) affect the prediction quality for the majority of coefficient images. However, there are some irregular columns or irregular pixels within a single column which this conclusion cannot be applied to. The irregularities' appearances do not follow a strict rule across images and finding them would require a full search, which is extremely time-consuming. Therefore, the above hypothesis is extended to all image pairs, ignoring irregularities.

These results are consistent across channels and images (i.e. not only specific for this sample image), allowing to set up an initial prediction scheme for compression: For every channel, the first coefficient images (zero-frequencies) are stored unchanged. For each remaining coefficient image, an RTF is trained which uses the three zero-frequency images as input (resulting in a three-channel input image). With this scheme, $3 * 63 = 189$ RTFs are used for prediction and three coefficient images have to be stored, along with the differences for predicted images.

## 4.4 Quantization and Entropy Coding

In order to evaluate the compression performance, initial methods for the last two steps are necessary. This section focuses on their initialization, as well as their optimization.

As introduced in chapter 3.1, the JPEG standard uses a combination of RLE and Huffman coding, which may be used directly for encoding coefficient differences. This scheme's suitability is assessed approximately by interpreting the coefficient differences (for coefficients 1 thru 63) and the source coefficients (for coefficients 0) as DCT coefficients. Through inverse DCT, an image is generated which is then encoded with an implementation of the JPEG encoder (in this case with the Microsoft .Net CLR implementation). This is possible because the differences are in the same range as the original coefficients. Because the zero-frequency coefficients are unchanged and differences are very small, it is very likely that the resulting image's pixels do not exceed the range of possible pixel values. The work-around of re-constructing an image through IDCT introduces another lossy quantization step, which has the effect that the coefficients and coefficient differences are changed slightly before being passed to JPEG's internal quantization and entropy coder. Therefore, the result can only approximate the actual performance of the original encoder. However, a quick evaluation is possible because it is not necessary to extract the relevant code from a JPEG implementation.

After encoding the image, it is immediately decoded to an RGB representation and the difference to the original is evaluated. Two measures are used to evaluate the image quality: PSNR (as introduced before with a peak signal of 255) and root mean squared error (RMSE, [RK03]), which is defined as the MSE's root:

$$RMSE = \sqrt{MSE}$$

Both measures have different advantages. PSNR can visualize changes of errors which are already very small. I.e. the PSNR raise of halving an error is a constant, independently of the error value. For RMSE those changes are smaller the smaller the original error value is, which makes them barely recognizable

for very small errors. On the other hand, the RMSE of a perfect reconstruction is 0, which is representable in a diagram. The according PSNR of $\infty \ dB$ is not suitable for visualization.

In the assessment, RTFs for each non-zero-frequency coefficient image of the already introduced sample image are trained for MSE loss. Figure 4.4 and Figure 4.5 show the results of this procedure.

(a) (b) (c) (d)

Figure 4.4 Results of using JPEG encoding after prediction. (a) The re-assembled image from coefficient differences that is sent to the JPEG encoder. (b-d) Decoded images for different quality levels (b) 1 (c) 15 (d) 100.

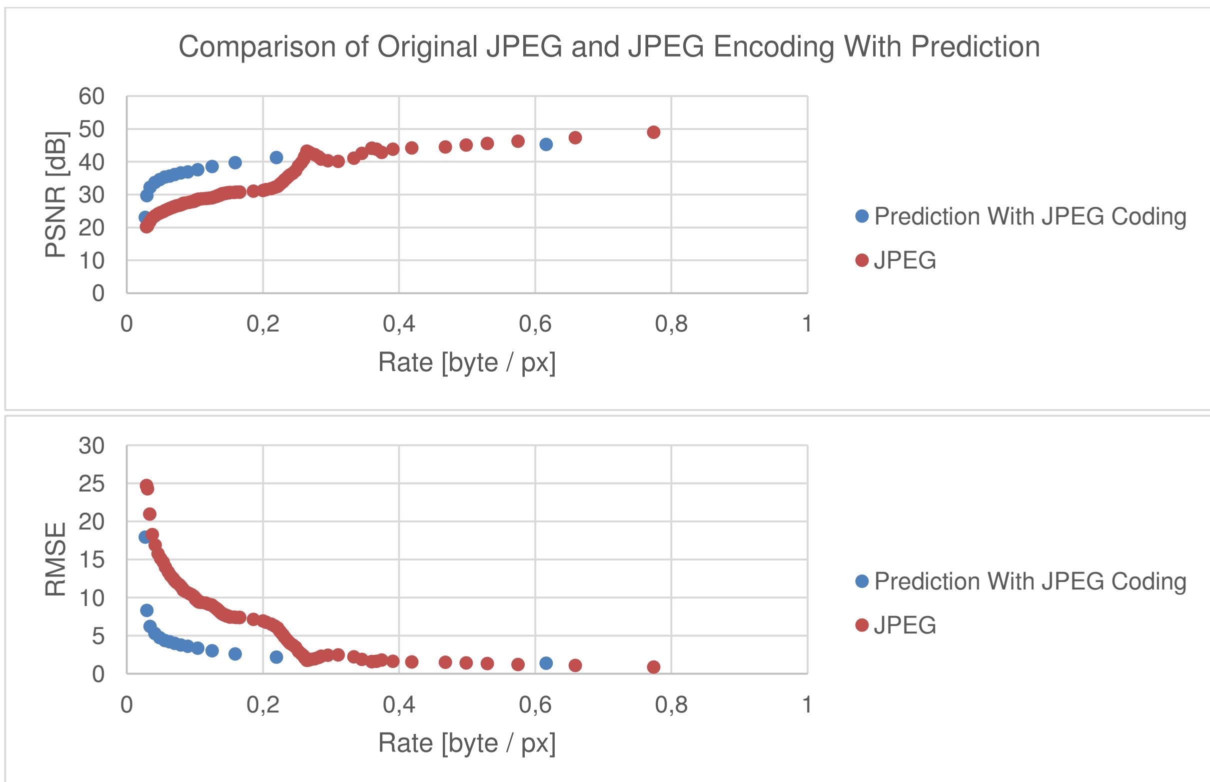



Figure 4.5 Rate distortion diagrams of using JPEG encoding after prediction. The rate represents the storage size which is needed in average to store a single pixel. Raw data require a rate of 3 bytes per pixel.

These results show that the compression ratio can be increased significantly, especially for low bitrates. However, the rate does not contain the storage required for saving the RTF models, which would require about additional 80 bytes per pixel for the chosen RTF complexity, thus making this scheme highly unsuitable for compression. Minimizing the size of RTFs is subject to other parts of the algorithm (most of all to the prediction strategy). Therefore, the additional size which arises from those models is ignored during the quantization and encoding's optimization.

### 4.4.1 Adapted Quantization and Encoding

Although the previous method yielded promising results, this section focuses on a custom implementation of quantization and encoding which is more suited for differences. Data quantization is performed similarly to JPEG's approach, which involves dividing the data by a quantization factor and rounding it, which produces a 16-bit signed integer:

$$quantize_f(d) = round\left(\frac{d}{f}\right)$$

JPEG uses different quantization factors for different coefficients and channels in order to remove irrelevant (for a human eye) high-frequency contents. This contradicts the general-purpose nature of this thesis. Furthermore, experiments have shown that the differentiation of quantization coefficients based on the coefficient frequency leads to a decrease in compression performance (in means of storage size and image quality). Instead, this differentiation is based on whether a coefficient image is stored as original values (for the zero-frequency images) or differences (all other images). The reason for that decision is the assumption that the overall error which results from an error in a difference image is far less than from an error in a non-difference image because difference images are only used to correct the predictions slightly. Non-difference images do not have predictions and should, therefore, be re-constructible as accurately as possible.

Both quantization factors are expressed as bit depths using the following formula:

$$f(bitdepth) = 2^{14-bitdepth}$$

The provided bit depth represents the number of bits that are needed to represent the full range of possible values after quantization. The DCT coefficients' original range $[-8192, 8128]$ (cf. section 3.1.3) requires 14 bits if rounded to integers. Therefore, a bit depth of 14 yields a quantization factor of 1. This change of representation is not necessary but makes parameterizing the algorithm more convenient.

After quantization, the entire data are passed into per-symbol RLE and Huffman coding [Huf14]. While the original JPEG applies these coding schemes to single blocks, this custom implementation encodes the entire data in a single pass. This allows to arrange data more efficiently (with respect to RLE), but comes with the requirement to hold the entire image in memory, which is already the case because a single DCT step yields pixels for all coefficient images. Therefore, the calculation of a single complete coefficient image has the side-effect that all other images are calculated as well.

Figure 4.6 visualizes this scheme's results. It achieves yet better compression ratios than the previous JPEG-based encoding scheme. High bit depths even allow perfect reconstruction of the input image. However, the visualized data does include neither the size of RTF models nor the Huffman tree which are both needed for decoding.

In the remainder of this thesis, such families of encoding schemes will be referred to with a single representative data line (e.g. the 10-bit line in Figure 4.6) for reasons of clarity. Usually, parameterizations with lower non-difference bit rates yield a better compression performance at low rates, whereas higher bit rates achieve better results at high rates, which has to be kept in mind when interpreting such diagrams. Furthermore, because PSNR diagrams allow a cleaner visualization (except their inability to visualize perfect reconstructions), RMSE diagrams will only be used if necessary.

The suggested encoding scheme has been optimized with respect to several factors, such as quantization factor calculation and the order in which the coefficients are serialized into a stream. The final stream order serializes coefficient images according to their channel and then in a zig-zag order according to their index (i.e. channel/index order starts with: 0/0, 1/0, 2/0, 0/1, 1/1, 2/1, 0/8, 1/8, 2/8 …). Coefficient images are serialized row-wise. This roughly sorts coefficients in descending order, which is optimal for RLE.

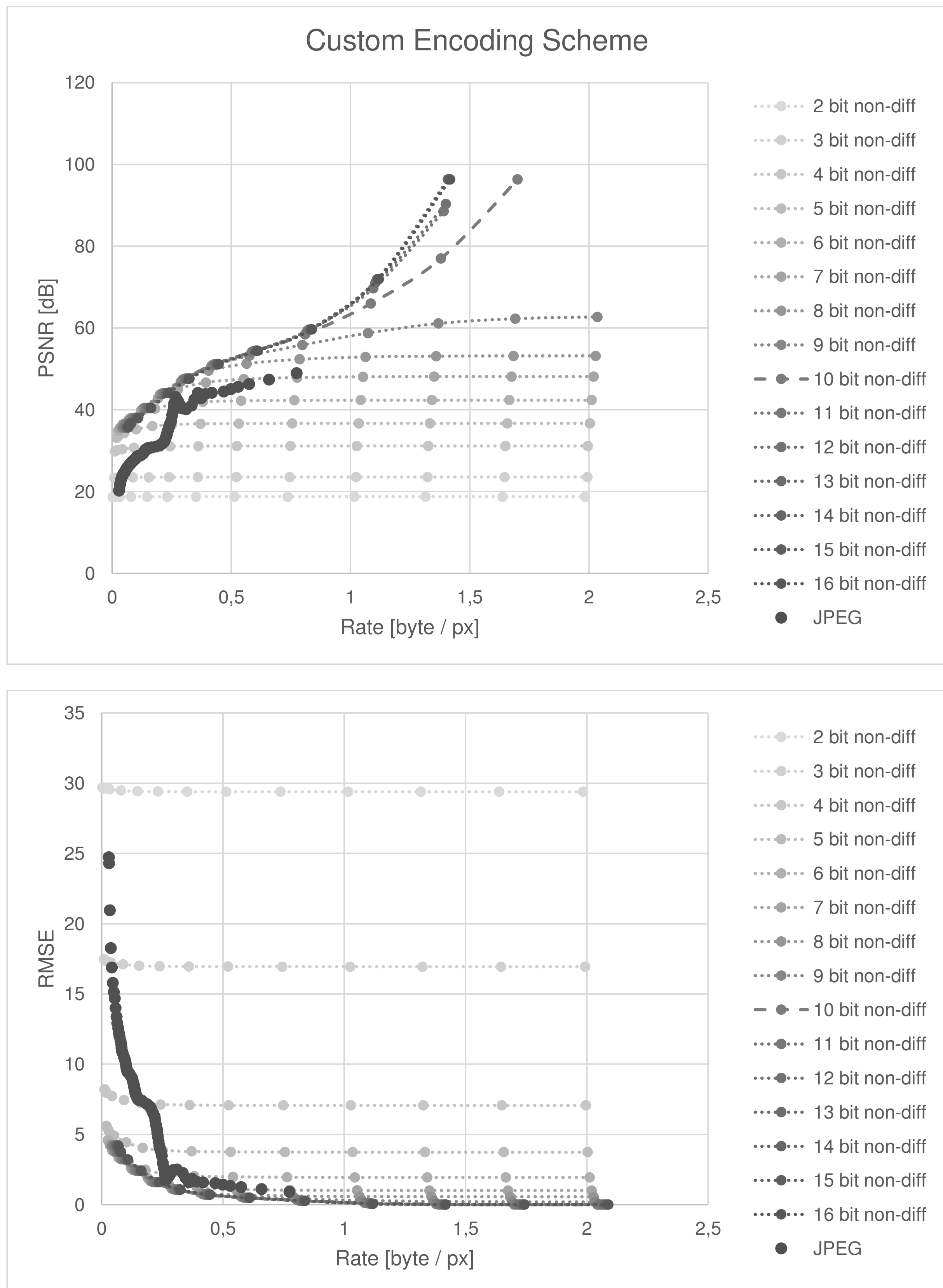



Figure 4.6 Rate distortion diagrams for custom encoding scheme. Data points with equal bit depth for non-difference images are connected by dotted lines. These interpolation lines are only used to connect the data points visually; they are not based on actual data.

### 4.4.2 Encoding RTF Models

The decoder needs to perform a prediction step, which requires the complete RTF model. Most model parameters can be considered constant (e.g. feature definitions) or can at least be stored in the file header with very little space requirements (e.g. factor type count). Regression trees do not fall within either class. The following section focuses on encoding such trees efficiently.

The used RTF implementation allows to serialize the trained RTF model in ASCII form, which is obviously not efficient. This text serialization is parsed back into an internal tree representation which only contains necessary data. Such representations can be re-serialized into the original ASCII form in order to send them to the RTF implementation for prediction. Additionally, the internal representation can be serialized in a dense binary form which is appropriate for storage.

Regression trees contain two node types: inner nodes and leaf nodes. Although the ASCII serialization contains the same data for all nodes, these are not necessary for prediction. Inner nodes only require feature information and leaf nodes only require the parameters of the local quadratic model as follows:

- Inner nodes:
    - Feature Type (one of four possible types)
    - Channel
    - Offset 1 (2D-vector in the integer range $[-15, 15]$)
    - Offset 2 (2D-vector in the integer range $[-7, 7]$)
    - Threshold (real-valued number)
- Leaf nodes:
    - $\theta$ (real-valued vector; 1D for unary types, 2D for pairwise types)
    - $\Theta$ (real-valued matrix; 1×1 for unary types, 2×2 for pairwise types)

Trees are serialized using a depth-first traversal. In that procedure, all nodes require an additional field which specifies if the node is an inner node or leaf node. This field is serialized as the first bit of a node. For inner nodes, the first offset is only necessary if the node represents a unary or pairwise feature, the second offset is only necessary in the case of pairwise features. Therefore, the size of serialized inner nodes varies between five and seven bytes, which is visualized in Figure 4.7.

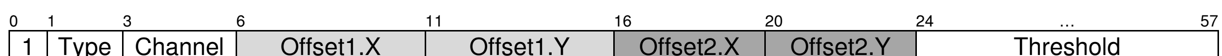


Figure 4.7 Data layout of inner nodes

Integer components that allow negative values are encoded with a shift bias of half their range. Real-valued components are encoded as 32 bit floating point numbers (IEEE 754 [IEEE87]).

Leaf nodes consist completely of real-valued components (two for unary trees and six for pairwise trees), which are encoded as 32 bit floating point numbers as before. However, the first bit, which defines if the node is a leaf, must be zero. According to IEEE 754, a float's first bit is its sign, which is zero for positive numbers. Due to this, any leaf's first number must be forced to be positive. This is achieved by adding a small amount to every leaf's first component. This number is determined in a pre-processing pass as the minimum of all leaves' first components. It is rounded off to an integer value and stored in the stream before the actual tree serialization as a single byte. On deserialization, this bias is read back and reverted for every leaf.

For RTFs with a single unary factor type, this scheme can be reduced further. There are no inter-dependencies of random variables in the resulting field, and each variable is influenced only by its own unary energy which is defined in the form of a one-dimensional quadratic function [JNSR12]:

$$E_{t,f,W}(y,x) = \frac{1}{2}\Theta_{\mathrm{t,W}}(x)y_{t,f}^2 - \theta_{t,W}(x)y_{t,f}$$

The minimizer of the overall energy (with respect to $y$) can be calculated locally as follows:

$$0 = E_{t,f,W}{}'(\hat{y}) = \Theta_{t,W}\hat{y} - \theta_{t,W}$$

$$\frac{\theta_{t,W}}{\Theta_{t,W}} = \hat{y}$$

Predictions are always the result of the division of the leaf's vector and matrix (both 1-dimensional for unary trees). Therefore, it is not necessary to store both components, but only their quotient. On deserialization, this quotient is used as the vector component, setting the matrix to 1. This changes the RTF but preserves its solution. This procedure cannot be applied to RTFs with pairwise factors because their solutions cannot be calculated locally.

An RTF may contain more than one unary factor type. This can be used to mimic regression forests. In this case (still without pairwise types), the local energy and minimizer is calculated as follows:

$$E(y) = \sum_i \left(\frac{1}{2}\Theta_i y^2 - \theta_i y\right)$$

$$E'(y) = y\sum_i \Theta_i - \sum_i \theta_i$$

$$\hat{y} = \frac{\sum_i \theta_i}{\sum_i \Theta_i}$$

This implies that it is not sufficient to store only quotients for those trees.

If serializations constructed in such way are processed by gzip for further compression, the result is bigger than the original input. This suggests that the developed scheme produces a very dense packing of the data with little redundancy.

Huffman trees are encoded likewise. Inner nodes contain a single bit; leaf nodes contain the has-children bit along with their according code words, rendering the bias shift superfluous.

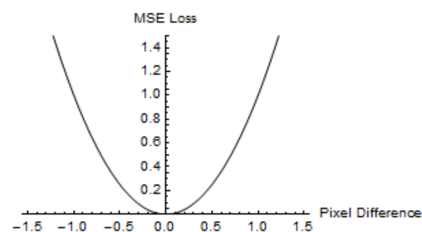
## 4.5 Loss Functions

Specific loss functions are used in the RTF training phase. They evaluate a prediction's quality with respect to a ground truth with a single scalar value. The chosen function influences the training process in two ways. When the parameters of leaf nodes are optimized, the loss function is the objective to minimize. Additionally, when selecting the split feature, the loss function's gradient is evaluated. This allows the loss function to influence prediction results significantly. An optimal loss function produces coefficient differences that require the least possible amount of storage to encode. The following section discusses a variety of functions and their applicability for compression.

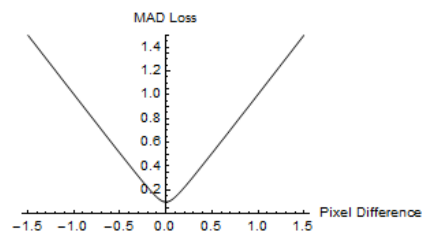
### 4.5.1 Distance Functions

Two loss function families have been examined. The first one is the family of distance functions. With these functions, optimization is performed with the ultimate goal to predict the original image as accurately as possible. They have their minimum at the point of zero-difference, i.e. where prediction and ground truth match exactly. This optimization objective is coherent with compression because zero-differences can be encoded efficiently. Even if most differences are non-zero but very small, efficient encoding is possible through quantization. Deviations from the ground truth are penalized differently by each loss function. Difference functions are evaluated separately per pixel. The overall result is computed as the average of pixel differences. Through normalization, all pixel values are in the range $[-1,1]$, allowing differences in the theoretical range $[-2,2]$.

MSE (mean squared error) is used very frequently in image processing because of its mathematical properties (efficiently computable, continuously differentiable). Furthermore, its optimization can be performed efficiently, using the average of differences. Near-zero differences yield a very small penalty, while big differences are penalized more strongly. This loss function is already implemented in the RTF distribution.

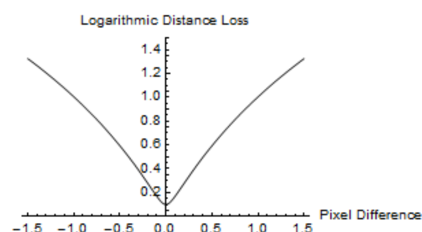



While MSE loss is appropriate to prevent outliers (through their large penalty), a uniformly increasing penalty might be more desirable for compression. The MAE (mean absolute error) would be a good choice for this. However, the absolute value function is not continuously differentiable at the point of zero-difference, making it unsuitable for optimization. To overcome this shortcoming, [TCAF07] proposed to introduce a small $\epsilon$, resulting in a differentiable function (MAD). In the plot, $\epsilon = 0.01$ in order to visualize its function. In the real implementation, it is much smaller. This function is also pre-implemented:



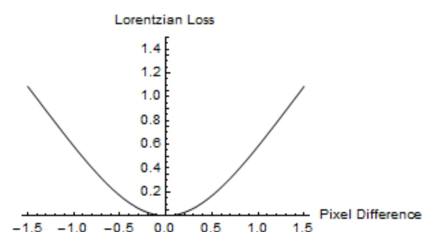

$$MAD(d) = \sqrt{d^2 + \epsilon}$$

After prediction, the differences are quantized to an integer. The number of bits required to represent any positive integer (without leading zeros) depends on its value as $bits = \lceil \log_2(value + 1) \rceil$. This suggests the usage of a logarithmic distance function in order to minimize the total amount of bits. Like for MAD loss, a small epsilon has been introduced to make the function differentiable. This function is not pre-implemented:




$$LogDist(d) = \sqrt{(\log_2(|d| + 1))^2 + \epsilon}$$

A pre-implemented logarithm-based loss function is the Lorentzian error. This function is considered robust against outliers and has proven successful in a variety of image processing tasks [Tap07]:




$$Lorentzian(d) = \log\left(1 + \frac{1}{2}d^2\right)$$

Figure 4.8 Loss function plots

### 4.5.2 Entropy Loss

The entropy coding step after prediction and quantization aims to encode the data with little redundancy, i.e. as closely to its information entropy as possible. The Shannon entropy is a measure for the information gain of each symbol in a data stream [TKS94]. It defines a lower bound for the size of any lossless direct code (such as the used Huffman code). The entropy can be calculated from the relative frequencies $f$ of each symbol $s$:

$$H = -\sum_{s \in S} f_s \log_2 f_s$$

The frequency of any quantized coefficient difference (bin $b$) for a given pixel distribution $P$ can be calculated as the number of pixels which produce this difference after quantization:

$$f_b(P) = \frac{1}{n} * \left|\left\{px \in P \mid quantize\left(diff_{px}\right) = b\right\}\right|$$

Assuming that the Huffman code introduces a constant redundancy, minimizing the coefficient differences' entropy directly maps to minimizing the required storage size. If the coefficient images can be predicted perfectly (i.e. with zero-difference), the resulting entropy is minimal. That's why difference based loss function can be considered as a special case of entropy minimization. However, the general entropy minimization approach aims to distribute the coefficient differences in an optimal way. This includes the case where all differences are equal, but non-zero, which also yields minimum entropy without the need for perfect predictions.

In order to calculate the entropy, the training process has to be aware of the normalization and quantization parameters. Both are factors and can be combined to a single factor which can be used to transform normalized to quantized coefficient differences.

Information entropy is only defined for discrete data, which is achieved through rounding in the quantization process. However, discrete data are not suitable for the quasi-Newton optimization used during training. Therefore, a continuous, differentiable approximation of the resulting entropy has to be found.

This is achieved by adapting the frequency calculation. In the original approach, each quantized sample point increments the count for the bin in which they lie. Even when the sample is changed slightly, it is very likely that it will be in the same bin again, which results in a zero-derivative. This problem can be avoided by assigning an extent to each sample so it will increase the relative count of several bins instead of a single one. Then, the bin's count is not an integer anymore but a real number because fractions of a sample's extent may overlap a single bin. This extent can be constant (i.e. a box function), but can also be adjusted by choosing an appropriate window function (i.e. bins close to the sample are influenced more than bins farther away). The frequency calculation changes accordingly:

$$f_b(P) = \frac{1}{n} \sum_{p \in P} \int_{\underline{b}}^{\overline{b}} w\left(x - quantize\left(diff_p\right)\right) dx$$

Where $\underline{b}$ and $\overline{b}$ are the bin's lower and upper bound, respectively and $w: \mathbb{R} \to \mathbb{R}$ is the window function. Furthermore, the quantization function does not include the rounding anymore. Figure 4.9 visualizes this process for a sample distribution of 16 quantized values and a triangle window function.

Figure 4.9 Adapted frequency calculation for entropy loss. Left: Sample distribution for triangle window; each peak represents a single sample's weight. Right: the resulting frequencies for each bin.

The final entropy-based loss function is calculated based on this real-valued frequency distribution, making the loss continuous. Besides being continuous, the loss function should be (desirably twice) continuously differentiable. The RTF implementation expects the loss function to provide its per-pixel gradient with respect to the prediction. This derivative can be calculated as follows:

$$
\begin{aligned}
g_p &\coloneqq \text{ ground truth value for pixel } p \\
pr_p &\coloneqq \text{prediction for pixel } p \\
q &\coloneqq \text{quantization factor} \\
f_b(P) &\coloneqq \text{frequency of bin } b \text{ for pixels } P \\
W(x) &\coloneqq \text{antiderivative of w(x)} \\
B &\coloneqq \text{set of all bins}
\end{aligned}
$$

$$
\begin{aligned}
f_b(P) &= \frac{1}{n}\sum_{p\in P}\int_{\underline{b}}^{\overline{b}} w\left(x - \frac{g_p - pr_p}{q}\right)dx \\
&= \frac{1}{n}\sum_{p\in P}\int_{\underline{b}-\frac{g_p-pr_p}{q}}^{\overline{b}-\frac{g_p-pr_p}{q}} w(x)dx \\
&= \frac{1}{n}\sum_{p\in P}\left[W\left(\overline{b} - \frac{g_p - pr_p}{q}\right) - W\left(\underline{b} - \frac{g_p - pr_p}{q}\right)\right] \\
\frac{\partial f_b(P)}{\partial pr_{\hat{p}}} &= \frac{1}{n}\left[\frac{\partial}{\partial pr_{\hat{p}}}\sum_{p\in P\setminus\hat{p}}\left[W\left(\overline{b} - \frac{g_p - pr_p}{q}\right) - W\left(\underline{b} - \frac{g_p - pr_p}{q}\right)\right]\right. \\
&\qquad \left. + \frac{\partial}{\partial pr_{\hat{p}}}\left(W\left(\overline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right) - W\left(\underline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right)\right)\right] \\
&= \frac{1}{n}\left[0 + \frac{\partial}{\partial pr_{\hat{p}}}\left(W\left(\overline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right) - W\left(\underline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right)\right)\right] \\
&= \frac{1}{nq}\left[W'\left(\overline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right) - W'\left(\underline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right)\right] \\
&= \frac{1}{nq}\left[w\left(\overline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right) - w\left(\underline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right)\right] \\
H(P) &= -\sum_{b\in B} f_b(P)\log_2 f_b(P)
\end{aligned}
$$

$$\begin{aligned}\frac{\partial H(P)}{\partial pr_{\hat{p}}} &= -\sum_{b\in B}\left[\frac{\partial f_b(P)}{\partial pr_{\hat{p}}} * \log_2 f_b(P) + f_b(P) * \frac{\partial f_b(P)}{\partial pr_{\hat{p}}} * \frac{1}{\ln 2 * f_b(P)}\right] \\ &= -\sum_{b\in B}\left[\frac{\partial f_b(P)}{\partial pr_{\hat{p}}}\left(\log_2 f_b(P) + \frac{1}{\ln 2}\right)\right] \\ &= \frac{1}{nq}\sum_{b\in B}\left[\left(w\left(\underline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right) - w\left(\overline{b} - \frac{g_{\hat{p}} - pr_{\hat{p}}}{q}\right)\right)\left(\log_2 f_b(P) + \frac{1}{\ln 2}\right)\right]\end{aligned}$$

This derivative can be calculated efficiently by ignoring bins that are not influenced by the respective pixel. This also avoids the calculation of $\log_2 0$ for a zero-frequency. In this case, the first factor in the sum (difference of window functions) is also zero, yielding zero for the entire term.

If another sample is added to the sample distribution of Figure 4.9, the entropy changes continuously, which is visualized by Figure 4.10. It can be observed that the entropy is minimal if the additional sample is added to the bin with the highest frequency. Therefore, entropy loss training tries to produce coefficient differences that already often exist.

The derivative's continuity is caused by the window function's continuity. However, because the window function is not continuously differentiable, the resulting entropy function is not twice continuously differentiable. This suggests the usage of a different window function. The window function should have the following properties.

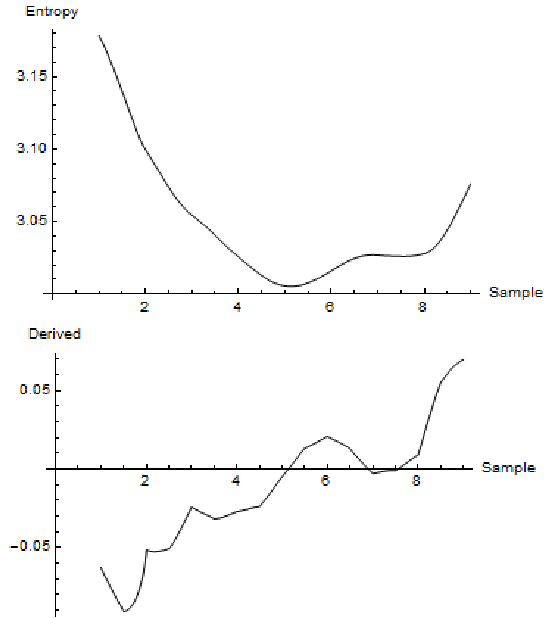



Figure 4.10 Entropy change if another sample with a certain value is added.

It should allow local evaluation, i.e. for a given window width $e$:

$$w(x) = 0\ \forall\ x < -\frac{e}{2} \vee x > \frac{e}{2}$$

It should be non-negative. This property asserts that the frequency is always positive for influenced bins (allowing the calculation of the logarithm).

$$w(x) \geq 0\ \forall -\frac{e}{2} \leq x \leq \frac{e}{2}$$

It should be normalized. Although this property is not necessary, it simplifies the code because frequencies can be normalized with the number of pixels:

$$\int_{-\frac{e}{2}}^{\frac{e}{2}} w(x)dx = 1$$

Additionally, it should be continuous and at least once continuously differentiable.

A window function which fulfills these properties is the Hann window [Har78]:

$$w(x) = \begin{cases} \frac{1}{e}\left(1+\cos\left(\frac{2\pi x}{e}\right)\right) & -\frac{e}{2} \le x \le \frac{e}{2} \\ 0 & otherwise \end{cases}$$

$$\int w(x)\, dx = \begin{cases} -0.5 & x < -\frac{e}{2} \\ \frac{x}{e} + \frac{1}{2\pi}\sin\left(\frac{2\pi x}{e}\right) & -\frac{e}{2} \le x \le \frac{e}{2} \\ 0.5 & x > \frac{e}{2} \end{cases}$$

Figure 4.11 shows the changed behavior of the continuous entropy function if the Hann window is used. The overall entropy is smaller as opposed to the triangle window because the Hann window is more focused on its center. Both samples (triangle and Hann window) use a window width of 3.

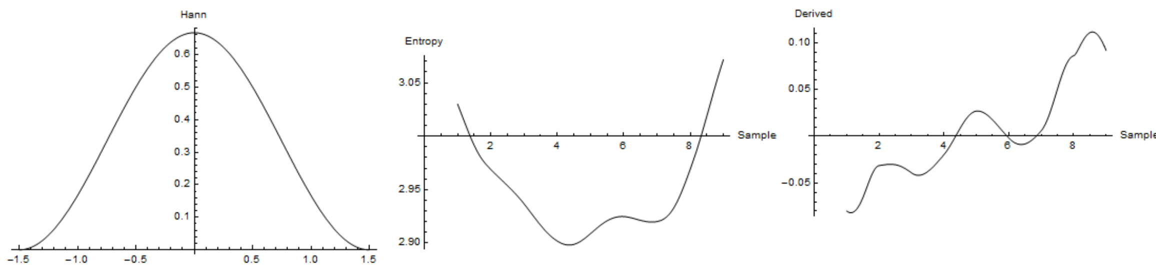



Figure 4.11 Hann window and resulting entropy

The approach described above allows to train a single RTF for optimal entropy of quantized differences. However, prediction schemes may include several RTFs (e.g. the initial scheme uses 189 RTFs). All differences are encoded together. Therefore, only the total entropy of all differences is a measure for the prediction quality, which is not automatically minimized by minimizing partial entropies.

In order to minimize the overall entropy, subsequent trainings have to be linked to each other. This is achieved by transferring the final frequencies to the next training round, which in turn calculates the actual frequencies from those and the distribution of its own prediction. Latter trainings then aim to produce differences that already exist in previous trainings. One problem of this approach is that the impact of trainings on the final entropy decreases from round to round, rendering the last rounds virtually powerless. A possible solution is to emulate a simultaneous training where all predictions have equal influence. This is achieved by using a single RTF to predict all 64 coefficient images of a channel. The maximum tree depth is increased by 6 to maintain fairness of comparison. These first six levels could be used to choose the correct subtree ($2^6 = 64$). Although this is most likely not what the training procedure will produce, such construction should perform at least as well as 64 separate RTF which are trained together for a single entropy. Details on how multiple inputs and outputs are handled by the RTF stage are described in a subsequent section.

### 4.5.3 Comparison of Loss Functions

The previous section introduced several loss functions, which all perform differently with respect to compression. Figure 4.12 shows the final compression results if all functions are used to compress the same image. The rate does include neither the RTF size nor the size of Huffman trees. All representatives encode non-differences with 10 bit. Training of entropy RTFs has been performed with the according difference bit depth. The complete data set can be examined on the accompanying CD.

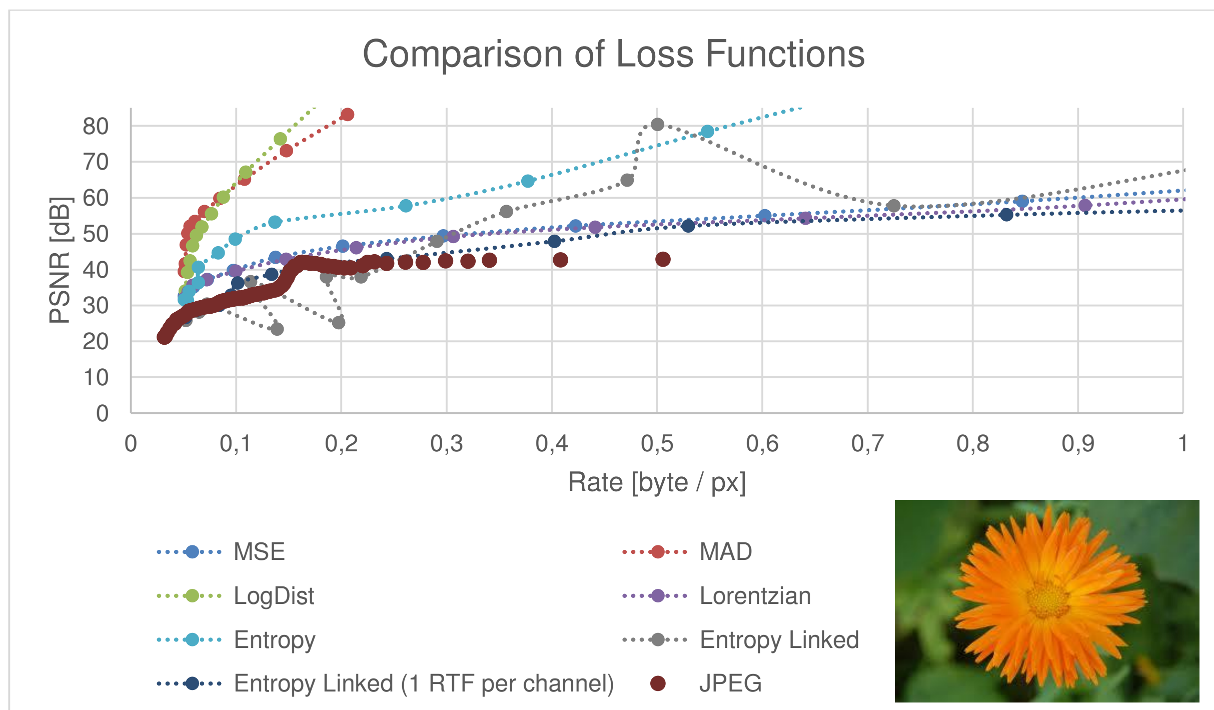



Figure 4.12 Comparison of Loss Functions. The sample image (bottom right) will be used for subsequent experiments.

There are obviously two loss functions which are superior to the other ones: MAD and LogDist. While MAD loss produces improved image quality for small rates, logarithmic distance loss is preferable for very high qualities. Entropy loss poses an improvement as opposed to MSE loss but does not achieve the performance of MAD and LogDist. Surprisingly, the improvements on entropy loss discussed in the previous section (linking frequencies, simultaneous prediction) perform worse than the original implementation where each prediction is optimized separately. These results are reproducible qualitatively for other images.

If high image qualities are desired, lossless compression might be more appropriate. Therefore, the rest of this thesis focuses on MAD loss.

## 4.6 Eigenvalue Bounds

One parameter set which has not been considered so far is the set of eigenvalue boundaries. Eigenvalue bounds constrain the eigenvalues of the leaves' matrix parameters to a particular interval. In theory, a tighter interval has two effects. Predictions become less distinctive, i.e. the possible error for the training data set may increase, but the RTF should generalize better for unseen data. This property is only important for scenarios where general RTFs are pre-installed and not trained for a specific data set. Secondly, the time required for training changes. Although training time varies strongly, in average a tighter bound results in faster training. Both effects are contrary. For compression, a small prediction error is necessary (loose interval) and little training time is desirable (tight interval). Figure 4.13 shows how different eigenvalue bounds influence compression performance (again 10-bit non-difference encoding is used).

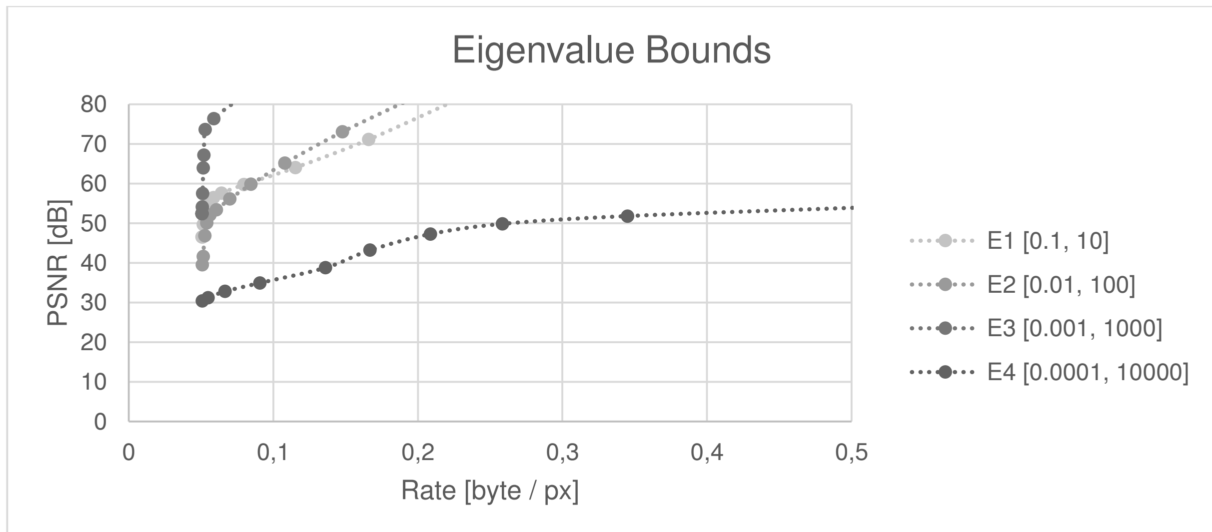



Figure 4.13 Comparison of compression performance for different eigenvalue bounds

In this case, training for the E3 interval is even faster than for the E2 interval (12:32 min vs. 15:47 min). For different images, the training time varies significantly. However, the E3 interval always yields the best compression performance, which is, therefore, used for subsequent experiments.

## 4.7 RTF Input Representation

RTFs are trained in a discriminative way, i.e. based on pairs of input and output images. Output images are single one-channel coefficient images, the type of input images can vary according to the used prediction scheme. In most schemes, input images consist of one or several coefficient images. Each source image forms a separate channel. Two artificial channels are added to every input image in order to identify the target image. All pixels within one of these channels are equal and represent the target image's coefficient index and channel. This approach allows to predict several target images from the same input image.

Both input and output images are normalized. Three normalization methods have been examined for their suitability for compression. Each method requires additional parameters to be stored in order to reconstruct the normalization and de-normalization on decoding.

The simplest form of normalization is division. For each coefficient image – both source and target images – it is necessary to store the divisor. In order to keep this additional space as small as possible, the divisor itself is quantized and stored as a single byte. The coefficient range requires a maximum divisor of 8192. When distributed over the 256 states of a byte, a quantization factor of 32 is necessary. The normalization method is then:

$$normalize_d(v) = \frac{v}{32 * (d+1)}$$

The divisor $d$ can be calculated from the image's maximum absolute value:

$$\frac{max}{32(d+1)} \leq 1$$
$$\left\lceil \frac{max}{32} - 1 \right\rceil = d$$

The second normalization method introduces a bias shift to make all values positive:

$$normalize_{d,s}(v) = \frac{v + 32s}{64(d+1)}$$

Both parameters are stored as a byte and require a total of two bytes per image.

The last method is based on [RK03]. It stores the coefficients' signs in a separate bit vector and performs a division. This makes the coefficients more uniform with the price of additional storage. This method requires $\frac{\#px}{8} + 1$ additional bytes per image.

$$normalize_{d}(v) = \frac{|v|}{32 * (d+1)}$$

Figure 4.14 shows the results of these three normalization methods.

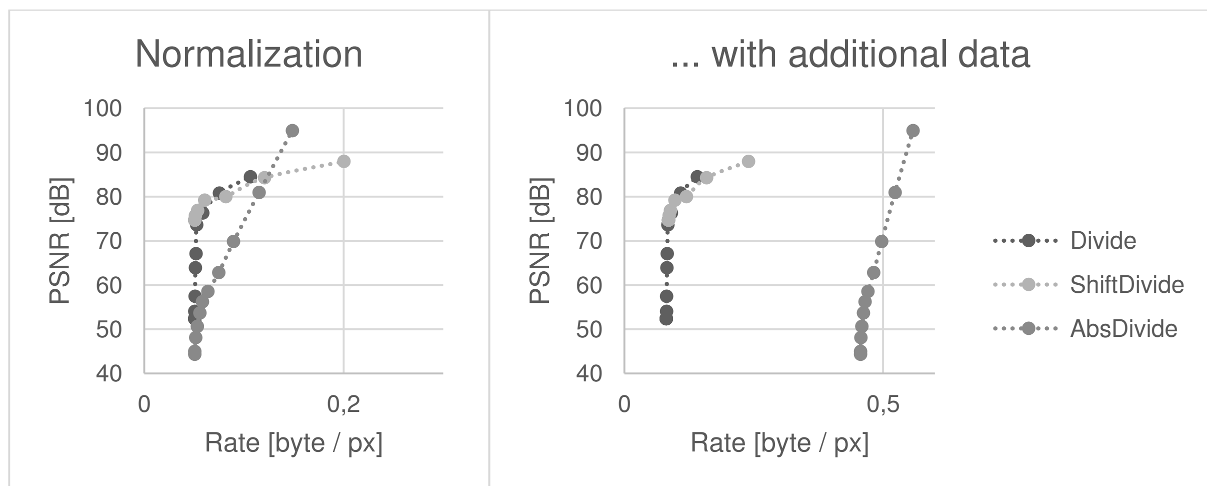



Figure 4.14 Comparison of normalization methods. Left: Rate includes only coefficient differences. Right: Rate also includes additional data (normalization parameters, Huffman trees).

For MAD loss, AbsDivide does not pose an improvement of compression performance. This is different for other loss function like MSE. However, its requirement to store the signs makes it an ineligible candidate. Divide and ShiftDivide normalization produce similar results, even for smaller tree depths. Because the shift does not result in a significant improvement, it is considered superfluous. Therefore, the optimal normalization method is the simple Divide normalization.

## 4.8 Prediction Strategy

The previous sections assumed that a separate depth-10 RTF for each non-zero-frequency coefficient image is trained. This strategy would require about additional 80 bytes per pixel to store the RTF models, which is far too much. This size can be reduced by decreasing the maximum tree depth, by defining fewer factor types, and by using fewer RTFs. The following section focuses on developing an alternate prediction scheme which uses less RTFs. Additionally to varying the prediction order, the maximum tree depth is decreased to 7. This reduces the required size of a single RTF from approximately 0.4 bytes per pixel to 0.05 bytes per pixel (in the sample image).

All prediction schemes have in common that the three zero-frequency coefficient images are stored as non-differences while all other parts are predicted. The following schemes have been examined with respect to their compression performance:

The initial scheme uses separate RTFs (189 in total) to predict each coefficient image from the three zero-frequency images (C0 images). This strategy is appropriate if all images are fundamentally different and share no commonalities. However, storing the RTF models requires a huge amount of memory. Therefore, this scheme is only suitable – if at all – if the RTF size can be distributed over a lot of pixels, i.e. for very big images or image data sets. This scheme will be referred to as *C0ToAll*.

The *row-wise* scheme builds upon the *C0ToAll* scheme for coefficient images with a zero horizontal frequency. However, each coefficient row of a channel (i.e. all images with the same vertical frequency) are predicted with a single RTF, using the three zero-frequency images and the respective predecessor (i.e. coefficient image 3 uses the three zero-frequency images and its predecessor image 2). This makes a total of $3 * (7 + 8) = 45$ RTFs.

The RTF count can be reduced further with the observation that respective coefficient images of different channels are correlated. Therefore, a single RTF can be used to predict all coefficient images of a color channel, using the three zero-frequency images and the partner of the luminance channel. The luminance channel is predicted like in the *row-wise* scheme with the modification that the first column (images with zero horizontal frequencies) is predicted with a single RTF from their predecessors. This results in a total of $1 + 8 + 2 = 11$ RTFs, which is why this scheme will be referred to as *11RTFs*.

The initial *C0ToAll* scheme can be modified to use fewer RTFs for the same predictions, i.e. either a single RTF per channel (where images are predicted in the same way as before) or a single RTF for the entirety of coefficient images.

As stated in the introductory chapter, coefficient images are not smooth like the original image. This might cause RTFs to perform better on plain images than on coefficient images, especially if pairwise factor types are used. That's why it might be more efficient to predict the color channels in the spatial domain rather than the frequency domain once the luminance channel has been restored. The luminance channel is predicted like in the *11RTFs* scheme, also yielding a total of 11 RTFs.

Figure 4.15 shows how the different prediction strategies perform with respect to compression performance. Again, each graph is the 10-bit-non-difference representative of the whole family of compression parameters.

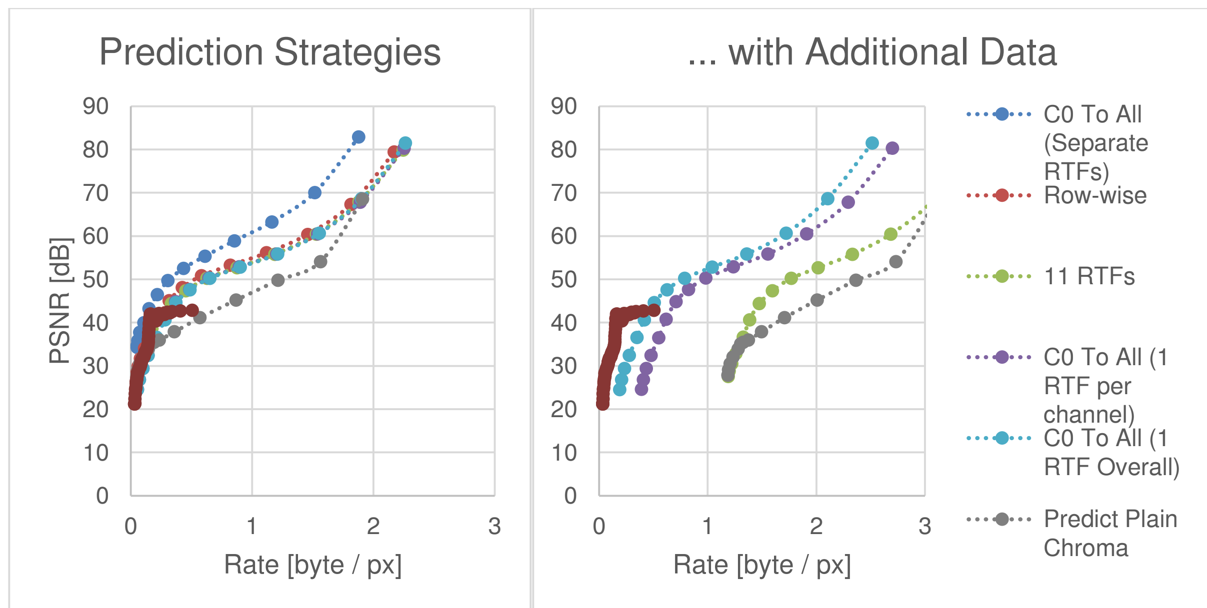



Figure 4.15 Comparison of prediction strategies. Left: Rate includes only coefficient data. Right: Rate includes normalization data, Huffman tree, and RTF models. The graphs for *C0ToAll* and *Row-wise* are not on this chart because the large number of RTFs induces high rates (about 20 and 5 bytes per pixel respectively).

Two properties can be observed in these charts. Disregarding additional data, only the original approach with separate RTFs per coefficient image represents an improvement over JPEG. In fact, all other strategies produce predictions that have large differences to the original. These differences are often in the same range as the original coefficients. In these cases, RTF prediction does not pose an improvement because the differences' entropy is nearly equal to the non-differences and therefore can't be encoded more efficiently.

On the other hand, the number of RTFs significantly increases the rate. Only if the RTF approach proves to scale well (i.e. prediction quality is preserved for a larger number of pixels and constant RTF complexity), it may be useful for compression at all. Each coefficient image of the sample has 748 pixels; the tree depth of 7 allows a maximum of 64 leaves per tree, which result in average-quality predictions (i.e. prediction is not perfect, but differences are significantly smaller than the original pixels). The previous experiment with a maximum tree depth of 10 (512 leaves max) yielded far better results. This represents almost a 1:1 mapping from pixels to leaves in the regression trees, which does not support the well-scaling assumption.

## 4.9 RTF Complexity

Before analyzing the scaling behavior of the RTF approach, the following section examines how the RTF complexity (maximum tree depth and factor type count) influences the prediction quality. Both parameters are reflected directly in the storage requirements of an RTF.

More factor types densify the conditional random field, which allows the modeling of inter-pixel dependencies. Furthermore, both training time (encoding) and testing time (decoding) increase as more factor types are present. Figure 4.16 visualizes how the number of factor types influences compression performance.

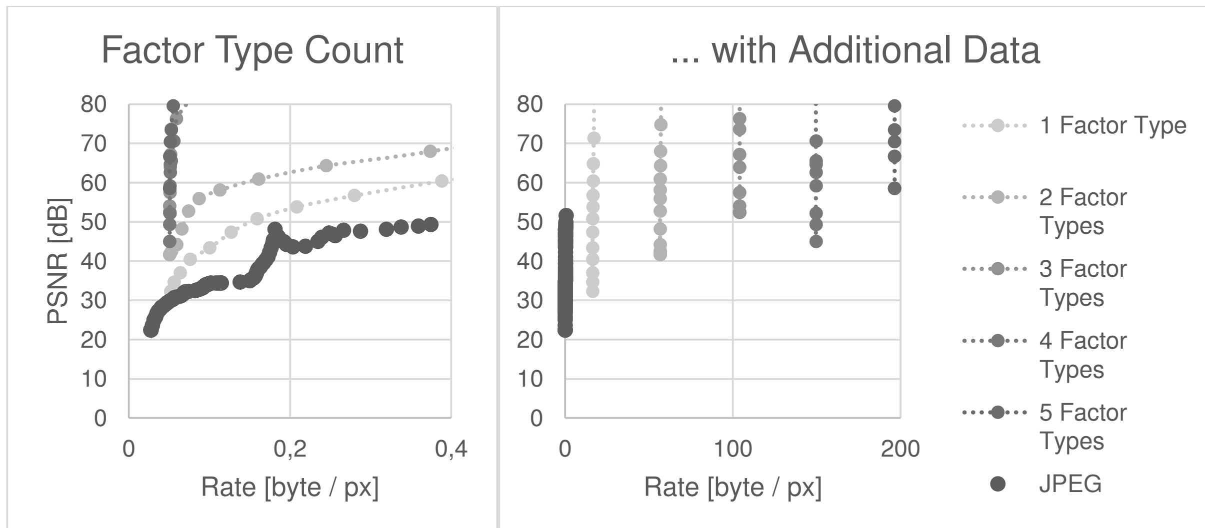



Figure 4.16 Comparison of different RTF complexities (factor type count)

Experiments with smaller trees have shown that RTFs with four and five factor types further improve the compression performance (compared to three factor types), which is not obvious in the diagram. One reason for this is that the additional trees can store more data.

Regression trees for the RTF are trained level by level and the parameters $\theta$ and $\Theta$ are stored at each node. This allows to revert the training process, i.e. for a given RTF it is easy to infer another RTF with shorter trees by simply cutting off nodes deeper than a pre-defined depth. The resulting RTF will behave equally to the original RTF if this had been trained only up to the pre-defined depth. While this behavior is not relevant in compression practice, it allows to evaluate different tree depths quickly. Figure 4.17 shows this evaluation's result.

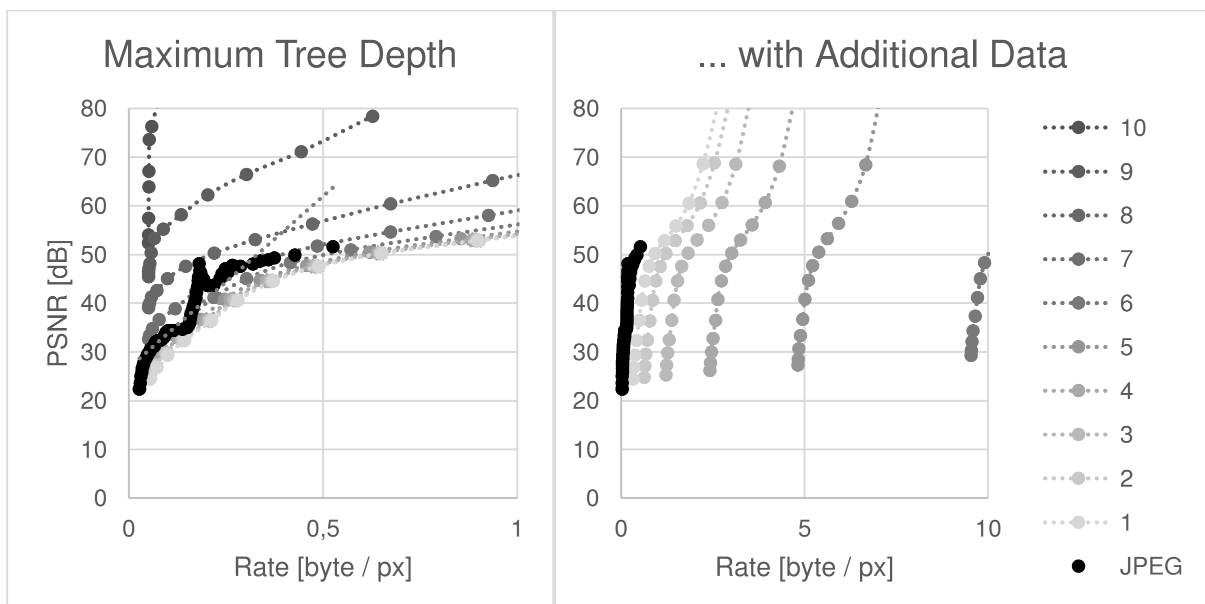



Figure 4.17 Comparison of different maximum tree depths for a three-factor-type RTF

This experiment shows that the prediction quality decreases quickly as the RTF complexity is reduced. The smallest complexity which yields usable predictions is the depth-8 RTF which allows a maximum of 384 leaves in all three trees (for 748 pixels). However, all complexities – even the depth-1 RTF –

need far too much storage for the RTF model. If additional data is considered, deeper trees increase the overall rate monotonically, regardless of the improved predictions.

## 4.10 Compression of Image Data Sets

The method to compress image data sets is almost identical to single-picture compression. The only difference is that the training step is provided with coefficient images from each image of the set simultaneously. This approach optimizes the RTF for the mean error across all images if an error loss function is used. For entropy loss, it is necessary to calculate the frequencies for each image separately, which is not implemented yet.

In order to evaluate the scaling-behavior of the RTF prediction, an image set of 100 similar images (all showing landscapes) has been encoded. The previous section did not yield a definite optimum for the number of factor types, so the following experiment will use a three-factor-types RTF, which should be able to model all necessary dependencies. Furthermore, the only strategy which has proven to improve the compression performance is used, which trains a separate RTF for each coefficient image. This also allows coefficient images of different frequencies to have fundamentally different properties (regarding RTF prediction) because each RTF specializes on a single frequency.

One coefficient image in this set consists of 1024 pixels. Previous experiments have shown that good predictions are achieved with a 1:1 mapping from pixels to leaves. Assuming that this ratio is independent of the number of images, this suggests the usage of a depth-9 to depth-10 RTF (with a maximum of 768 and 1536 leaves for all three trees, respectively) for this experiment. If this assumption is wrong, and the ratio refers to the total pixel count (102,400 in this case), a depth-16 RTF would be more appropriate.

Indeed, this experiment's results do not support the well-scaling assumption. The depth-10 RTF produces unusable predictions, whereas the predictions of a depth-15 RTF are of average to good quality, which leads to the conclusion that images in the frequency domain do not share valuable features that can be used by the regression trees. Instead, almost every pixel of every image needs a separate path in the RTF to be predicted correctly (or at least nearly correctly). However, this basically outsources an image's data into the RTF, which does not pose a real model in the sense of machine learning.

## 4.11 Comparison to RTF De-blocking

In chapter 2 the application of JPEG de-blocking has been introduced [JNR12], which also represents a kind of image compression with the help of RTFs. Through the usage of RTFs, this approach is a direct competitor to the method developed in this thesis. The following section compares both methods with respect to compression.

If the RTF models can be considered available (i.e. without the need to store them explicitly), the RTF-based compression poses a significant improvement and outperforms the de-blocking approach. However, de-blocking uses an RTF model which generalizes even to images that are not used during training. Hence, this approach scales very well, which is not the case for the RTF-based compression. Therefore, if the size of RTF models is considered, de-blocking is the only of both methods that can improve JPEG compression.

# 5 Implementation Details

In order to evaluate the compression performance for the various parameter combinations, an application has been developed which can assemble an overall scheme from atomic steps. The following chapter focuses on this program's implementation.

## 5.1 Architecture

The entire application consists of three components. The transformations library is a dynamically linked C++ library which handles low-level conversions such as DCT and color space transformations. The RTF management is also a C++ library. It provides methods to train RTFs and perform predictions using the RTF implementation from [JNSR12]. It is also responsible for converting input data into the correct data structures and for custom loss functions. The GUI and high-level logic are contained in a C# executable which in turn uses the aforementioned libraries via Platform Invoke. High-level logic includes code for assembling the final compression scheme and performing evaluations.

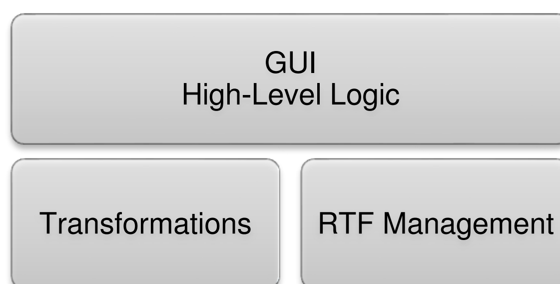



Figure 5.1 Overview of the system architecture

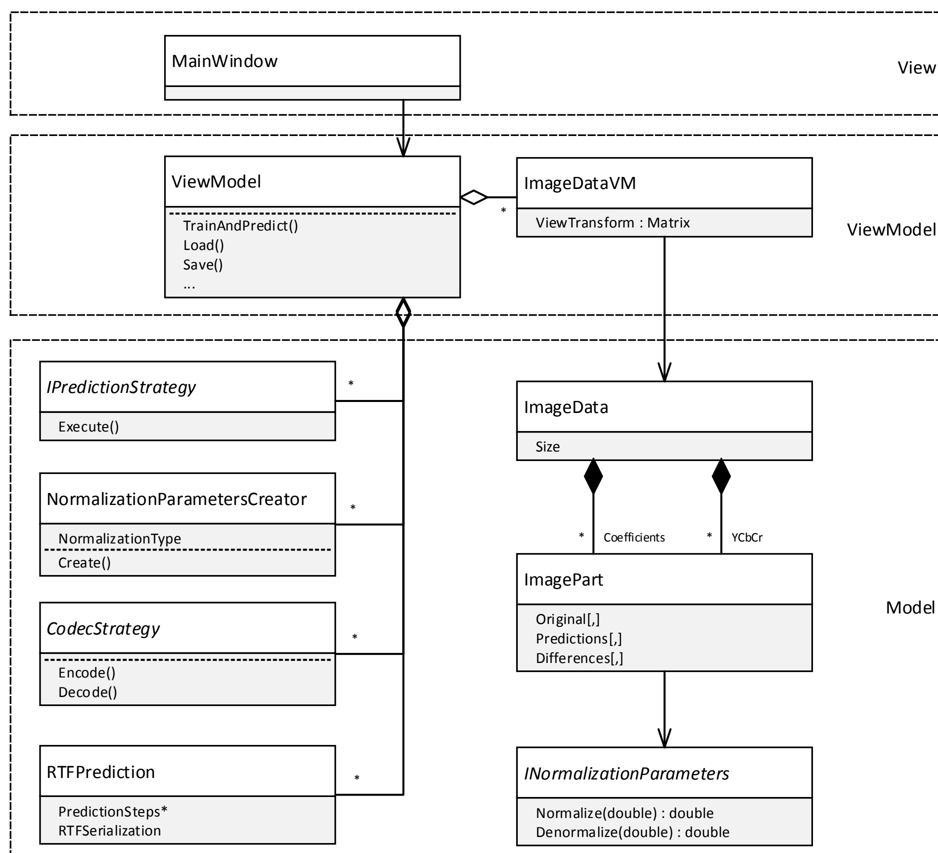



Figure 5.2 Reduced class diagram of the GUI and high-level logic component

The GUI and high-level logic component obeys to the Model-View-ViewModel (MVVM) design pattern, which is a variation of the MVC pattern. The view model's task, which is situated between the view and the model, is to provide the model's data and functionality in a way that can be handled by the view. One main difference between MVVM and MVC is that in MVVM the view specifies which view model to use, whereas in MVC the controller specifies which view is presented to the user.

Multiple images can be loaded into the application. For each image an `ImageData` object is created, which holds the image in both spatial and frequency representations. The `ImageDataVM` wraps these data in a view model and adds GUI-specific properties, such as a transformation matrix to modify the visualization.

`RTFPrediction` objects store trained RTF models along with the predictions for which they are used. The models are represented in the serialized form of the RTF library. However, they can be easily converted to internal tree structures with the `RTFReader` and back to serializations with the `RTFWriter`.

The variable steps of the compression schemes are objectified in several classes. Prediction strategies implement the `IPredictionStrategy` interface (strategy pattern) and define in which order and from which sources image parts are predicted. The actual prediction process is executed by the `ViewModel` using inversion of control.

During prediction, a set of normalization parameters (`INormalizationParameters`) is created for each `ImagePart`. Normalization parameters are used to normalize image data to a $[-1, 1]$ range before prediction. Which type of normalization parameters is created is defined by the `NormalizationParametersCreator` (reflective factory pattern).

Objects of type `CodecStrategy` (strategy pattern) define how the prediction results are encoded in the final data stream, i.e. which data are serialized and how they are encoded.

The above architecture's primary design goal is flexibility in terms of compression scheme variability. Performance and memory efficiency are subordinate. This allows to store data suboptimally (e.g. image data in various representations) if this increases flexibility (different prediction strategies use different image representations).

## 5.2 Implementation of Entropy Loss

The RTF implementation requires every loss function to define an empty tag class (usually in the `Loss` namespace) and the actual implementation (usually in the `Loss::Detail` namespace). Additionally a template specialization of the `LossDispatcher` is necessary. E.g. for MSE loss:

```
namespace Loss {
	class MSE;
	namespace Detail {
		template <…> struct MSE   {
			static … Objective(…) { … }
			static void Gradient(…) { … }
		}
	}
}
```

For entropy loss, a different structure has been chosen: The tag class is not empty anymore but contains necessary data for the loss function (such as quantization factors and frequencies of previous predictions). The implementation is split into two structures – one untemplated struct which defines generic functionality (such as the window function and frequency convolution) and the default templated struct. This distinction is necessary because both the templated implementation and the tag class need to access the generic functionality of the untemplated implementation. Although it would be possible to define the generic functionality in the tag class, it is considered cleaner to keep the tag class as slim as possible (with only meta-data).

In order to perform the frequency calculation, the definition of the window function `double window (double)` and its indefinite integral `double windowIntegral(double)` is necessary. The convolution is calculated incrementally by evaluating distinct values of the integral function. The definition of the derivative is not necessary because the loss function's gradient can be calculated from the original window function.

Furthermore, the RTF implementation has been adapted to include a pre-processing step for loss functions. This pre-processing step is used to calculate the total frequencies of all images before calculating the entropy and gradient. If the frequencies were not calculated in a pre-processing step, the objective function and gradient function could only access the frequencies of their respective images and previous images. I.e. the gradient calculation for the first image would not consider frequencies in the second and subsequent images.

## 5.3 Selected Design Patterns and Structures

The flexibility requirement strongly suggests the usage of design patterns where appropriate. The following section focuses on a selection of utilized patterns beyond the already introduced strategy and factory patterns.

### 5.3.1 Assembling Prediction Strategies

The `ViewModel`'s `TrainAndPredict()` method takes a list of prediction steps as the first argument, where a `PredictionStep` consists of a number of source images (defined by coefficient index and channel) and a target image. This list is a complex type whose construction needs to be performed carefully. This suggests the usage of the builder pattern (actually a variation of the original pattern). The builder pattern allows to construct complex types step by step, enforcing protocol constraints (e.g. order of data, value checks etc.) Additionally, the builder pattern is most suitable for a fluent interface, which allows a convenient usage. The prediction step builder's interface is defined as follows:

```
class PredictionMaker {
	PredictionMaker AddSources(params int[] coeffChannels);
	PredictionMaker AddTarget(int coefficient, int channel);
	PredictionStep[] Make();
}
```

This class can be used as follows:

```
TrainAndPredict(new PredictionMaker()
	.AddSources(0, 0, 0, 1).AddTarget(1, 0)
	.AddSources(0, 2, 1, 0).AddTarget(2, 0).Make(), …);
```

This defines an RTF that is used to predict two coefficient images. The image 1/0 (index/channel) is predicted from the images 0/0 and 0/1; image 2/0 is predicted from 0/2 and 1/0. During construction, each input is checked for validity. Furthermore, the builder ensures that every target has sources and that every source has a target.

### 5.3.2 Dynamic Serialization Order

All codec strategies – except the JPEG encoder – need to serialize input data into a stream before further compression (RLE, Huffman) can be applied. The order in which data are linearized significantly influences compression performance. A solution that can construct a serialization in an arbitrary order dynamically is essential to evaluate the compression performance without the need to re-compile.

Usually, the code for a hard-coded stream order looks as follows:

```
for (int channel = 0; channel < 3; ++channel)
	for (int c = 0; c < 64; ++c)
		for (int y = 0; y < height; ++y)
			for (int x = 0; x < width; ++x)
				serializedData[i++] = structuredData[x, y, c, channel]
```

A change of the order would require re-arranging the `for` statements. Instead, the serial index (formerly `i++`) can be calculated explicitly by an external function from the structured position vector `[x, y, c, channel]`. If only sequential serializations of each parameter were allowed (e.g. arrange all `c`'s from 0 through 63), this method would be required to calculate a dot product with pre-calculated coefficients. However, more advanced serializations are possible (e.g. zig-zag serialization), which cannot be calculated with a simple dot product. In order to allow such orders, the index function is calculated on demand in a recursive way. A function produced in this way can be used by the codecs to serialize data:

```
streamOrder = StreamOrder.Construct(width, height, StreamComponent.Channel,
          StreamComponent.CoefficientZigZag, StreamComponent.X,
          StreamComponent.Y);

…
serializedData[streamOrder(x, y, c, channel)] = structuredData[x, y, c, channel]
```

The number of stream components that are passed to this method is variable, i.e. the same method can be used to construct orders for data of arbitrary dimensionality.

### 5.3.3 Updates of the View

The MVVM pattern strongly relies on data binding to display data in its views. This means that data is not assigned to display controls directly (e.g. text for text boxes). Instead, only the data's location is specified. This allows lazy evaluation of data; i.e. UI data need only be calculated if there is a control to display them.

Another advantage of this approach is that UI controls do not need to be updated manually. Most data objects provide an event that is raised whenever the object's content is changed. By applying a variation of the observer pattern, bindings can update controls automatically when their data source changes.

Additionally, these events can be used independent of views. E.g. the `ViewModel` watches all RTFs for changes. Whenever an RTF is changed (e.g. through tree reduction), the view model initiates a re-prediction of involved targets (loose coupling). Furthermore, some statistics are updated (e.g. maximum tree depth and RTF storage size).

# 6 Conclusions

This thesis has shown that JPEG compression can be improved significantly by encoding differences if image parts can be predicted reliably. Furthermore, the results demonstrated that RTF models are not suited to predict images in the frequency domain as there are no or few usable features in the input images. Appropriate preprocessing steps or different image representations might change the RTF's suitability, which is subject to further research.

One strength of the RTF model is loss-specific training. Indeed, the choice of loss functions has great influence on the compression performance. A novel form of entropy optimization has been developed in this thesis. Although this loss function turned out to be outperformed by MAD and LogDist loss in its current form, more favorable parameterizations (e.g. window width) and better integration in the optimization process (avoidance of local optima) might increase the according compression performance. Further research on this topic may result in a loss function which is superior to others with respect to compression.

The developed application is optimized for flexibility in terms of software design, which makes it most appropriate as a basis for ongoing research efforts. This includes – besides the aforementioned topics – the analysis of different machine learning models and different types of media (such as volumes and 3D meshes).

## References

[Ahm05] Ahmed, R. Wavelet-based Image Compression using Support Vector Machine Learning and Encoding Techniques. *Proceedings of Computer Graphics and Imaging.* 2005.

[ANR74] Ahmed, N. and Natarajan, T., Rao, K.R. Discrete Cosine Transform. *IEEE Transactions on Computers.* 1974, Vols. C-23, 1.

[BFOS84] Breiman, Leo, et al., et al. *Classification and Regression Trees.* s.l. : Wadsworth Publishing, 1984. 0-412-04841-8.

[BKR11] Blake, Andrew, Kohli, Pushmeet and Rother, Carsten. *Markov Random Fields for Vision and Image Processing.* s.l. : MIT Press, 2011. 978-0-262-01577-6.

[CGGM08] Camps-Valls, Gustavo, et al., et al. On the Suitable Domain for SVM Training in Image Coding. *The Journal of Machine Learning Research.* 2008, Vol. 9.

[FTM12] Fazli, Saeid, Toofan, Siroos and Mehrara, Zahra. JPEG2000 Image Compression Using SVM and DWT. *International Journal of Science and Engineering Investigations.* 2012, Vol. 1, 3.

[GCG05] Gomez-Perez, G., Camps-Valls, G. and Gutierrez, J. Perceptual adaptive insensitivity for support vector machine image coding. *IEEE Transactions on Neural Networks.* 2005, Vol. 16, 6.

[Har78] Harris, F. J. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE.* 1978, Vol. 66, 1.

[HK06] Hyndmana, Rob J. and Koehler, Anne B. Another look at measures of forecast accuracy. *International Journal of Forecasting.* 2006, Vol. 22, Issue 4.

[Huf14] Hufmann Code Implementation. [Online] [Cited: 08 14, 2014.] http://rosettacode.org/wiki/Huffman_coding#C.23.

[Hut06] Hutter, Marcus. Prize for Compressing Human Knowledge. [Online] 2006. [Cited: 07 29, 2014.] http://prize.hutter1.net/.

[IEEE87] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. *ACM SIGPLAN Notices.* 1987, Vol. 22, Is. 2.

[IJG14] Independent JPEG Group. Independent JPEG Group. [Online] 2014 Jan. [Cited: 08 10, 2014.] http://www.ijg.org/.

[ITU01] International Telecommunication Union. *J.144 : Objective perceptual video quality measurement techniques for digital cable television in the presence of a full reference.* 2001.

[ITU02] —. *T.800 : Information technology - JPEG 2000 image coding system: Core coding system.* 2002.

[ITU14] —. *R BT.2020 : Parameter values for ultra-high definition television systems for production and international programme exchange.* 2014.

[ITU92] —. *T.81 : Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines.* 1992.

[ITU96] —. *T.84 : Information technology - Digital compression and coding of continuous-tone still images: Extensions.* 1996.

[Jia99] Jiang, J. Image compression with neural networks – A survey. *Signal Processing: Image Communication.* 1999, Vol. 14, Issue 9.

[JNR12] Jancsary, Jeremy, Nowozin, Sebastian and Rother, Carsten. Loss-specific training of non-parametric image restoration models: A new state of the art. *12th European Conference on Computer Vision (ECCV).* 2012.

[JNSR12] Jancsary, Jeremy, et al., et al. Regression tree fields – An efficient, non-parametric approach to image labeling problems. *25th IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2012.

[KAV05] Kasetkasema, Teerasit, Arora, Manoj K. and Varshney, Pramod K. Super-resolution land cover mapping using a Markov random field based approach. *Remote Sensing of Environment.* 2005, Vol. 96, 3-4.

[KS80] Kindermann, Ross and Snell, J. Laurie. *Markov Random Fields and Their Applications.* s.l. : American Mathematical Society, 1980. 0-8218-5001-6.

[Mah06] Mahoney, Matt. Rationale for a Large Text Compression Benchmark. [Online] Aug 2006. [Cited: 07 29, 2014.] http://cs.fit.edu/~mmahoney/compression/rationale.html.

[MSR13] Microsoft Research. Regression Tree Fields. [Online] Sep 2013. [Cited: 08 04, 2014.] http://research.microsoft.com/en-us/downloads/2a5e65d4-85a8-4579-b5d8-38831d36d8a2/default.aspx.

[Poy01] Poynton, Charles. *Digital Video and HD: Algorithms and Interfaces.* 2001. 978-1558607927.

[RH05] Rue, Havard and Held, Leonard. *Gaussian Markov random fields; theory and applications.* s.l. : Taylor & Francis Goup, 2005. 1-58488-432-0.

[RK03] Robinson, J. and Kecman, V. Combining support vector machine learning with the discrete cosine transform in image compression. *IEEE Transactions on Neural Networks.* 2003, Vol. 14, Issue 4.

[RYQB05] Jiao, Runhai, et al., et al. SVM Regression and Its Application to Image Compression. *Advances in Intelligent Computing. Lecture Notes in Computer Science.* 2005, Vol. 3644.

[Tap07] Tappen, M.F. Utilizing Variational Optimization to Learn Markov Random Fields. *IEEE Conference on Computer Vision and Pattern Recognition. CVPR.* 2007.

[TCAF07] Tappen, M.F., et al., et al. Learning Gaussian Conditional Random Fields for Low-Level Vision. *IEEE Conference on Computer Vision and Pattern Recognition. CVPR.* 2007.

[TKS94] Han, Te Sun and Kobayashi, Kingo. *Mathematics of Information and Coding.* s.l. : American Mathematical Society, 1994. 978-0821842560.

[ZBSS04] Wang, Zhou, et al., et al. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing (Volume:13, Issue: 4 ).* 2004.

# List of Figures

Figure 3.1 Overview of the JPEG encoder ... 7
Figure 3.2 Result of color space transformation ... 8
Figure 3.3 2D-DCT's base functions ... 9
Figure 3.4 The sample image's coefficient images for the Y channel using different display mappings ... 9
Figure 3.5 Sample graph with five random variables. ... 11
Figure 3.6 Sample factor graph ... 12
Figure 3.7 Factor graph for natural 4-connectivity of an image. ... 13
Figure 4.1 Modified codec scheme with incorporated prediction step ... 17
Figure 4.2 Definition of factor types based on the factor type count. ... 18
Figure 4.3 Results of the predictive dependencies experiment for different RTF complexities. ... 19
Figure 4.4 Results of using JPEG encoding after prediction. ... 21
Figure 4.5 Rate distortion diagrams of using JPEG encoding after prediction. ... 21
Figure 4.6 Rate distortion diagrams for custom encoding scheme. ... 24
Figure 4.7 Data layout of inner nodes ... 25
Figure 4.8 Loss function plots ... 27
Figure 4.9 Adapted frequency calculation for entropy loss. ... 29
Figure 4.10 Entropy change if another sample with a certain value is added. ... 30
Figure 4.11 Hann window and resulting entropy ... 31
Figure 4.12 Comparison of Loss Functions. ... 32
Figure 4.13 Comparison of compression performance for different eigenvalue bounds ... 33
Figure 4.14 Comparison of normalization methods ... 34
Figure 4.15 Comparison of prediction strategies. ... 36
Figure 4.16 Comparison of different RTF complexities (factor type count) ... 37
Figure 4.17 Comparison of different maximum tree depths for a three-factor-type RTF ... 37
Figure 5.1 Overview of the system architecture ... 39
Figure 5.2 Reduced class diagram of the GUI and high-level logic component ... 39