Super-Resolution with Regression Tree Fields

Project Report

Jakob Kruse

Introduction

This document is a report of my project for the module “Masterpraktikum” within the Masters programme in Computer Science at Technische Universität Dresden.

The task I was given might be described like this: Adapt the existing RTF image restoration framework from J. Jancsary, S. Nowozin and C. Rother[2] to do Single Image Super-Resolution, building upon state-of-the-art methods by incorporating their results and possibly introducing suitable new features.

In the first part of the report, I will give a short overview of the two concepts making up the title of this work, Super-Resolution and Regression Tree Fields. Following that, I will talk about the changes I undertook within the RTF framework and their motivation. In the third part, I will describe the test setup as well as the various minor scripts used in the execution of the experiments. This will be followed by the results and, in the last part, a discussion of the outcome of this project, the lessons learned and the opportunities for future improvement.

Super-Resolution

When the term Super-Resolution is used in computer vision, it generally refers to some method that takes as input one or more images of a scene, and returns a higher resolution image of the same scene as output. To do so, it has to recover or reconstruct detail that is not explicitly present in the original image(s).

These methods can be divided into Multiple Image and Single Image Super-Resolution. The former usually operates on a set of similar images that only differ by sub-pixel offsets. In that case, the missing detail can be calculated with high confidence from the high frequency content of the images. This project, however, is about the latter case, in which only a single low resolution image is given and the missing detail has to be guessed using some form of prior knowledge.

One popular form of prior knowledge used is self-similarity. Intuitively, to find the high resolution version of a particular image patch, a similar patch of higher resolution in the same image is used. This has the obvious advantage of requiring no information from outside the input, but the results on images with few self-similar patches are bound to be poor.

Another approach is to substitute high resolution patches not from the same image, but from an example database. With sufficiently many (and diverse) example patches, this allows for very rich texture detail; but it has the drawback of always having to keep a large database around. Additionally, there is a danger of bringing in too much data from different contexts, thus destroying the identity of the original image.

A more general approach, then, is to use machine learning to train a graphical model that can find the most likely output based on how other images related to their (known) high resolution counterparts. Once the model is learned, the training data are not needed anymore and its application to new images tends to be very fast. As we will see, Regression Tree Fields fill this role exceptionally well.

Just as an aside, for this particular project, the height and width of the high resolution output image will always be twice those of the input image.

Regression Tree Fields

Regression Tree Fields are a very recent type of graphical model, having been introduced only in 2012 by J. Jancsary, S. Nowozin, T. Sharp and C. Rother [1]. At its core, an RTF can be seen as an extension of a Conditional Gaussian Random Field. CGRFs model the probability distribution over all possible output images $\boldsymbol{y}$ based on an input image $\boldsymbol{x}$ and their model parameters. The crucial difference is that RTFs are designed to be "non-parametric", i.e. there are no global model parameters. Instead, regression trees are used to specify the local model that is in effect at each position based on the surrounding image content.

To visualize how this is achieved, it helps to think of the model as a factor graph. There are different types of factors, as represented in **Figure 1(a)** for the example of a standard 8-connected graph. Each of these factor types is instantiated at every position in the image, yielding something like **Figure 1(b)**, and every factor instantiation contributes one energy term to the overall energy of the configuration. The configuration with the lowest total energy then corresponds to the most likely output image.

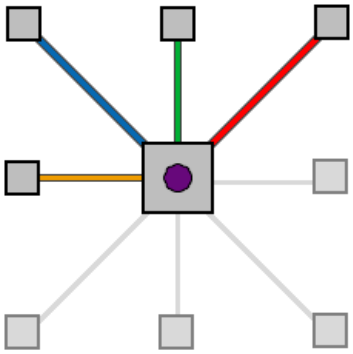

Figure 1(a) – The five factor types appearing in an 8-connected graph (3-by-3 neighbourhood)

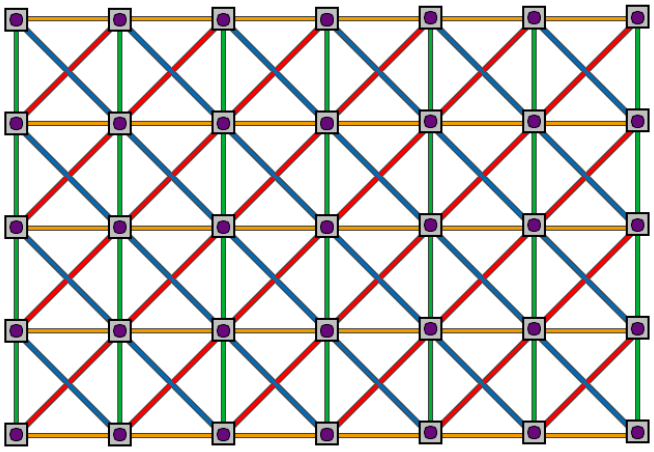

Figure 1(b) – Factor instantiation across the image

The local model that is used for a specific instantiated factor's energy term is given by a regression tree. This tree takes the local content of $\boldsymbol{x}$ in that factor's neighbourhood as input and, after a number of branching decisions, arrives at a leaf. This leaf stores the parameters for the local energy term that will be applied for the factor in question (see **Figure 2**). There is one regression tree for each factor type, and factors of the same type share the same tree. If too factors from the same type are instantiated in similar image contexts, they will therefore have the same local model and if their contexts differ significantly they will have different local models.

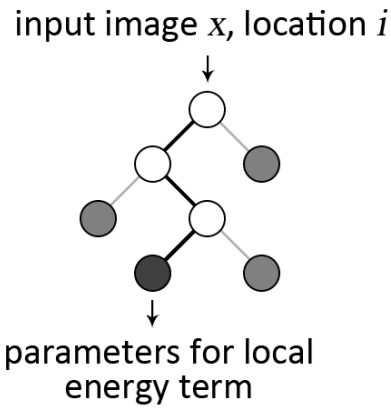

Figure 2 – Regression tree sketch

The parameters stored in the trees' leaves and the branching criteria of the regression trees have to be found by training. It was shown in [2] that training by minimization of the empirical risk with relation to a (differentiable) loss function is feasible and even beneficial. Without going into too much detail, the joint training of both tree structure and values in the leaves works as follows: Starting with trees of depth 1, the parameters in the leaves are optimized and the leaves then split in such a way that the next parameter optimization can yield a lower overall loss. These steps are iterated until the desired maximum tree depth is reached.

The resulting model achieves very good prediction times and outperforms all of the compared methods in [2] in terms of the underlying loss function.

Changes to the RTF framework

Compared to the overall complexity of the RTF code, the adaptations necessary for it to handle Super-Resolution were relatively straightforward. In essence, I had to accommodate for the difference in size between input and output images.

Since the original framework already supports multidimensional output, this turned out to be easier than initially anticipated. This is because internally, the RTF does not map images comprised of pixels to other images, but fields of input vectors to fields of output vectors. The size of those vectors can be chosen freely, as long as suitable methods for conversion of the input and ground truth images (and subsequent saving the output images) are given. In fact, the original input vectors are already bolstered up with a lot of additional data, like responses from a filter bank and local image noise levels.

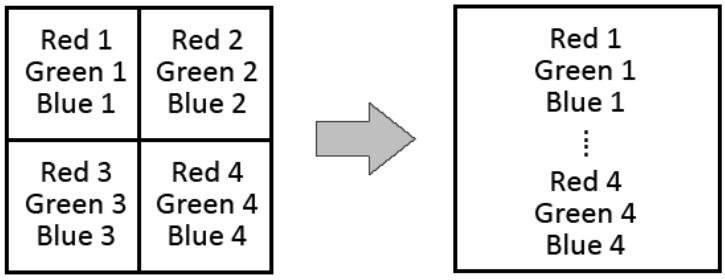

Figure 3 – Four-in-one output pixel vector

The first change I made, then, was to set the size of the output vectors to four times its previous value in `Dataset.h`, so each output vector would contain the RGB components of four output pixels (see **Figure 3**). The method for saving output images was changed accordingly, as was the method for loading ground truth images, which are represented in the same way.

```
static void SaveGroundTruthImagePNG(const ImageRefC<UnaryGroundLabel>& ground,
const std::string& path)
{
  ImageRef<unsigned char, 3> img(ground.Width() * 2, ground.Height() * 2);

  for( int y = 0; y < ground.Height(); ++y )
    for( int x = 0; x < ground.Width(); ++x )
      for( int c = 0; c < 3; ++c )
      {
        // Sub-pixel 1
        *(img.Ptr(2*x, 2*y) + c)         = (unsigned char) (std::max(0.0f,
std::min((float) ground(x,y)[c], 1.0f))*255.0f + .5f );

        // Sub-pixel 2
        *(img.Ptr(2*x + 1, 2*y) + c)     = (unsigned char) (std::max(0.0f,
std::min((float) ground(x,y)[c + 3], 1.0f))*255.0f + .5f );

        // Sub-pixel 3
        *(img.Ptr(2*x, 2*y + 1) + c)     = (unsigned char) (std::max(0.0f,
std::min((float) ground(x,y)[c + 6], 1.0f))*255.0f + .5f );

        // Sub-pixel 4
        *(img.Ptr(2*x + 1, 2*y + 1) + c) = (unsigned char) (std::max(0.0f,
std::min((float) ground(x,y)[c + 9], 1.0f))*255.0f + .5f );
      }

  Utility::WritePNG(img, path);
}
```

Saving a ground truth image

To capitalise on one of the strengths of the RTF model, which is building on the results of other state-of-the-art methods, I incorporated the output of K. I. Kim and Y. Kwon's Super-Resolution approach [3]. To do so, I applied the MATLAB code they provide on their project page to my dataset and changed the image loading method to append the pixel values from their output to the corresponding RTF input vectors. The size of the input vectors was increased accordingly.

```
ImageRef<InputLabel> LoadInputImage(const size_t idx) const
{
  ImageRef<InputLabel> ret;
  std::string path = InputImagePath(idx);

  if( path.back() == 'g' || path.back() == 'G' )
    ret = LoadInputImagePNG(path);
  else
    ret = LoadInputImageDLM(path);

  ImageRefC<UnaryGroundLabel> kk2010_output =
LoadGroundTruthImagePNG(GetImagePath(idx, "kk2010"));

  for( int y = 0; y < ret.Height(); ++y )
    for( int x = 0; x < ret.Width(); ++x )
      for( int c = 0; c < 12; ++c )
        ret(x,y)[kk2010_output_index(c)] =
static_cast<float>(kk2010_output(x,y)[c]);

  return ret;
}
```

Loading an input image with appended state-of-the-art-results

I also added an additional split criterion for the regression trees to the learning algorithm in `Feature.h`. This so-called binning feature virtually partitions the image into five by five equally distributed bins and, for a given location, returns the number of the bin that location falls into (see **Figure 4**). That means a tree can take into account in which part of the image a factor was instantiated when determining its parameters. The more similar the images in a dataset are in their layout, the more useful this feature should become.

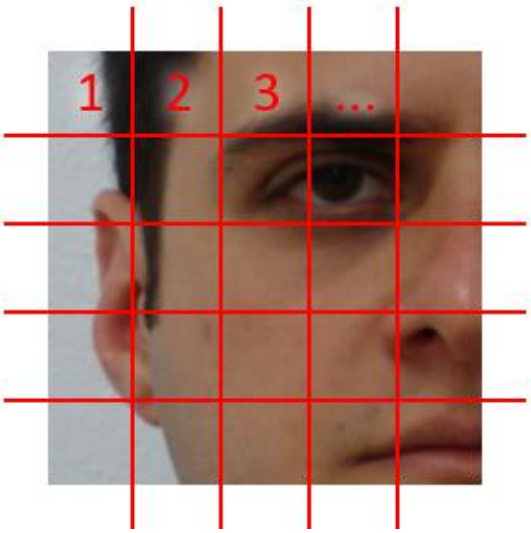

Figure 4 – Binning feature

Finally, I made a few minor changes to the method `Compare()` in `Superresolution.h`, which computes the loss over a set of pairs of output and ground truth images with respect to a specific loss function, to get more data out of it in a useable format. There was also a small string comparison bug in the way this method is called, which I fixed.

Test setup

For testing the resulting machine learning system, I used portrait photographs from the FEI Face Database [4]. This database contains 200 sets of photos with each set consisting of 14 angles/expressions of the same face. I used the neutral front view from each set, ending up with 200 images of very similar layout. Each of these I cropped to a specific 150 by 150 pixel region containing the most important face features to keep the computational effort low while maintaining the level of detail. From these ground truth images, I created the set of input images by simply downscaling by a factor of two (with default bicubic interpolation).

All of this was done with the MATLAB script `build_dataset.m`.

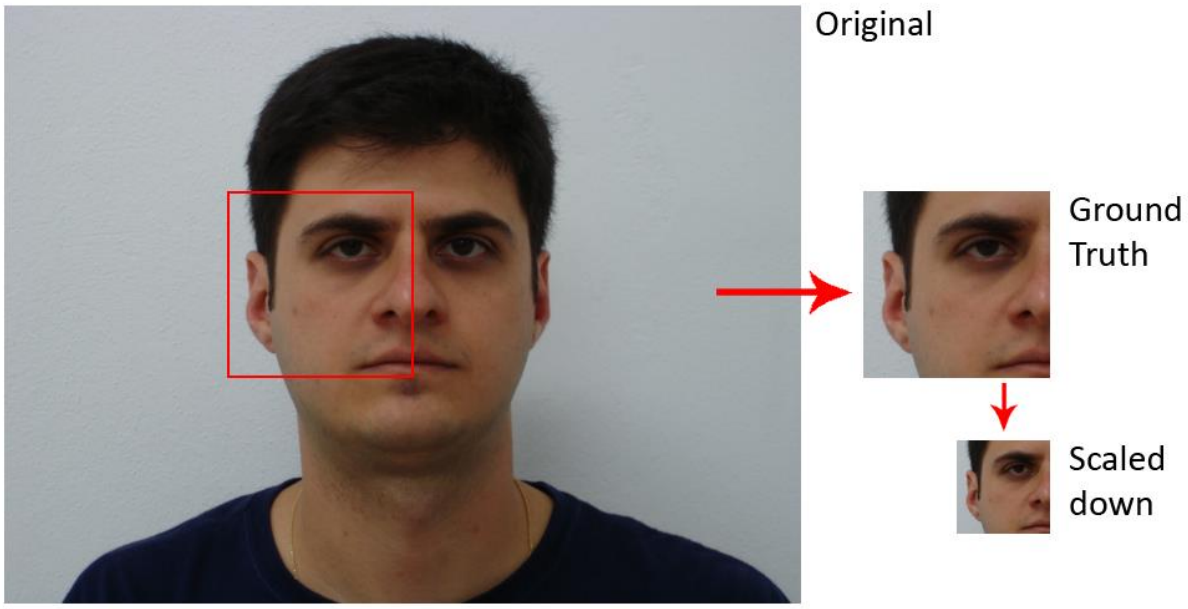

Figure 5 – Dataset preparation

I then did two test runs on this set, one optimising the model for the *Peak Signal-to-Noise Ratio (PSNR)* loss function and the other optimising for *Structural Similarity (SSIM)*. Each time, I measured the average loss between predictions and ground truth with respect to both of these loss functions and additionally to the *Information Content Weighted Structural Similarity (IWSSIM)* function, which is an extension of *SSIM*. I then compared the results to the average losses of the super-resolution method from K. I. Kim and Y. Kwon [3] as well as naive scaling with nearest-neighbour, bilinear and bicubic interpolation.

Because of the strong hardware requirements of the joint learning algorithm, I was only able to use 60 images for training as using more let the programme crash. Even so, it quickly exceeded the testing machine's 8GB of main memory and thus took very long to compute. For this reason, the depth of the regression trees was also limited to 5 instead of the desirable 10 or more, and the binary factors drawn from a 5 by 5 neighbourhood.

Only when running without the incorporated state-of-the-art results from [3], training with 100 images was possible. The results of this (when optimising for *SSIM*) are also given.

Overall, a lot of small scripts and batch files were used to prepare the data, run the tests and collect the results, all of which will be included in the appendix for reference.

Results

The primary objective measure for the performance of a Super-Resolution method is the amount of similarity, or absence of differences, between the generated high resolution images and the ground truth. Since the latter is known for each of the images, any function that compares two images may be used to obtain such a measure. Because the RTF runs were optimised for *PSNR* and *SSIM* respectively, I used these functions for comparison with the other methods.

Figures 6(a) and **(b)** show the average values of the two functions, over all the test images, for each of the tested Super-Resolution methods. For *PSNR*, possible values theoretically range from 0 (images are completely independent) to infinity (images are completely identical). For *SSIM*, the boundaries are -1 and 1, respectively. In both cases higher values indicate better results.

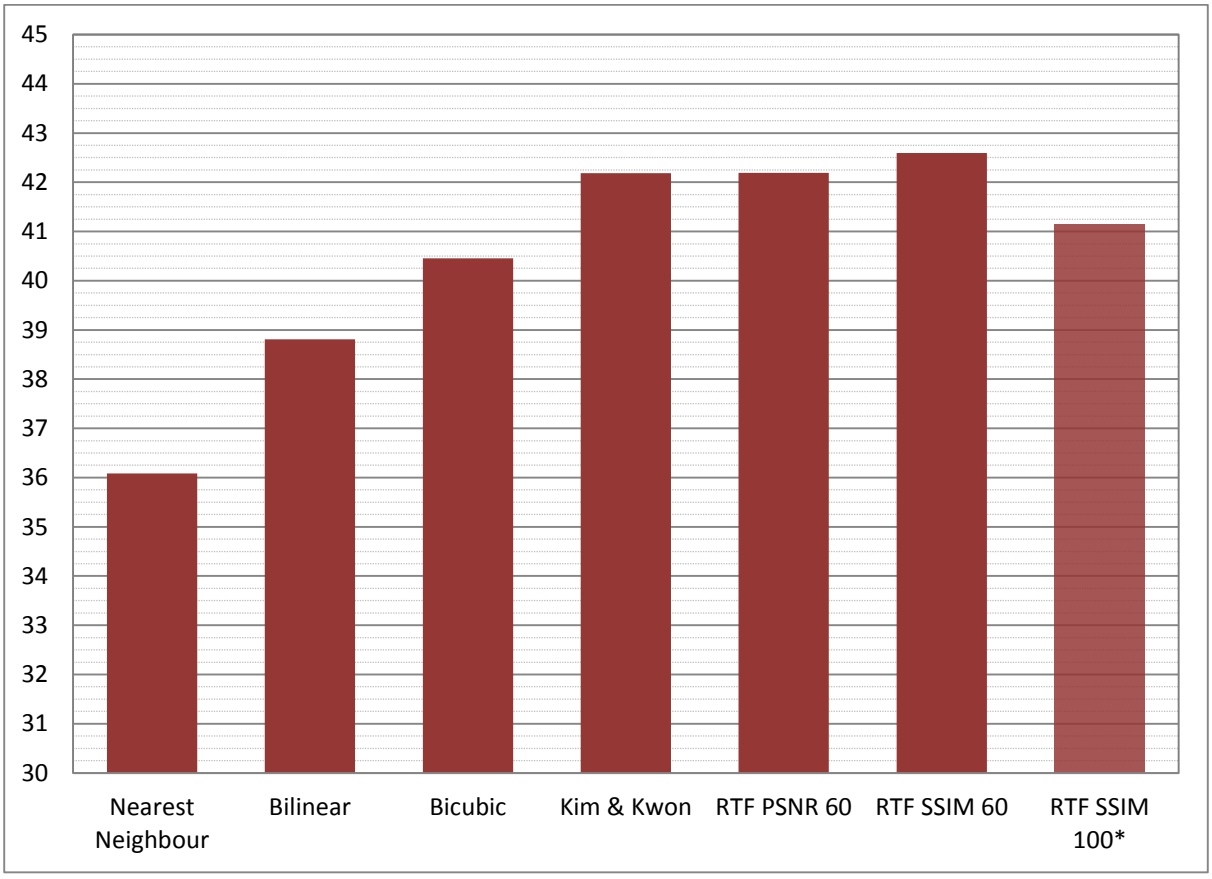

Figure 6(a) – Results compared by means of Peak Signal-to-Noise Ratio (PSNR): higher is better

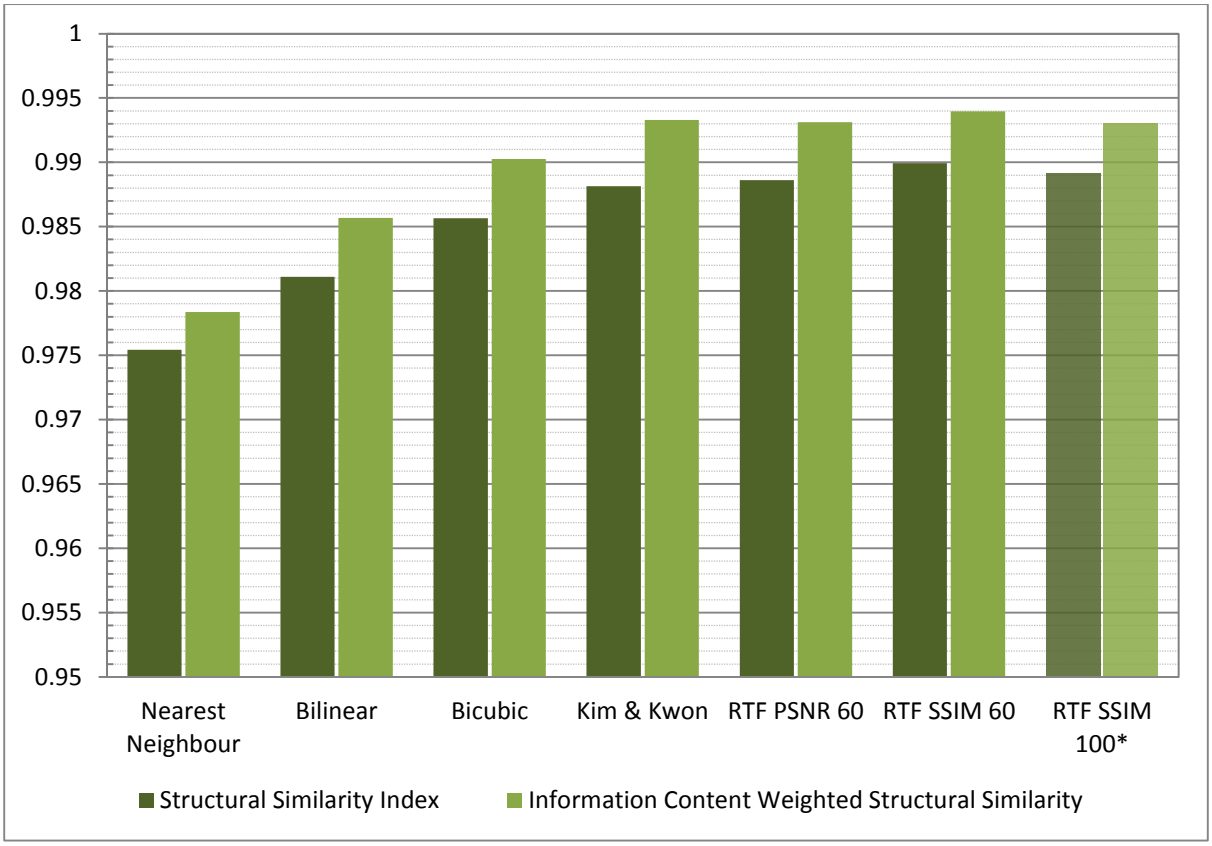

Figure 6(b) – Results compared by means of SSIM and ISSIM: higher is better

In both charts, *Nearest Neighbour*, *Bilinear* and *Bicubic* stand for the simple interpolation methods as implemented in MATLAB. *Kim & Kwon* stands for the results from [3], while *RTF PSNR 60* and *RTF SSIM 60* stand for the RTF runs optimised for *PSNR* and *SSIM* respectively, each trained with the first 60 images from the dataset. *RTF SSIM 100** stands for the RTF run optimised for *SSIM* with 100 training images, but lacking incorporated state-of-the-art results and the new binning feature. In each but the last case, the set of test images consisted of images 61 through 200 of the dataset, while in the last case it was images 101 through 200.

As can be seen, the quality of the naive scaling methods rises with their complexity in both charts, but is clearly limited. The state-of-the-art results from [3] pose a significant improvement, in both cases almost extrapolating the line given by the previous methods. In comparison, *RTF PSNR 60* achieves the same average *PSNR* and *SSIM* values and, interestingly, slightly better *IWSSIM* values than [3]. What is more surprising, though, is that *RTF SSIM 60* outperforms both of them not only for the *SSIM* variants, but also for the *PSNR* measure. This result was confirmed by other RTF runs on 50 training images (not shown here).

It is also worth noting that the “pure” RTF approach *RTF SSIM 100** performed just as well as the method from [3] when comparing *SSIM* values, for which it was optimised. The power of the model therefore lies not only in the incorporated results.

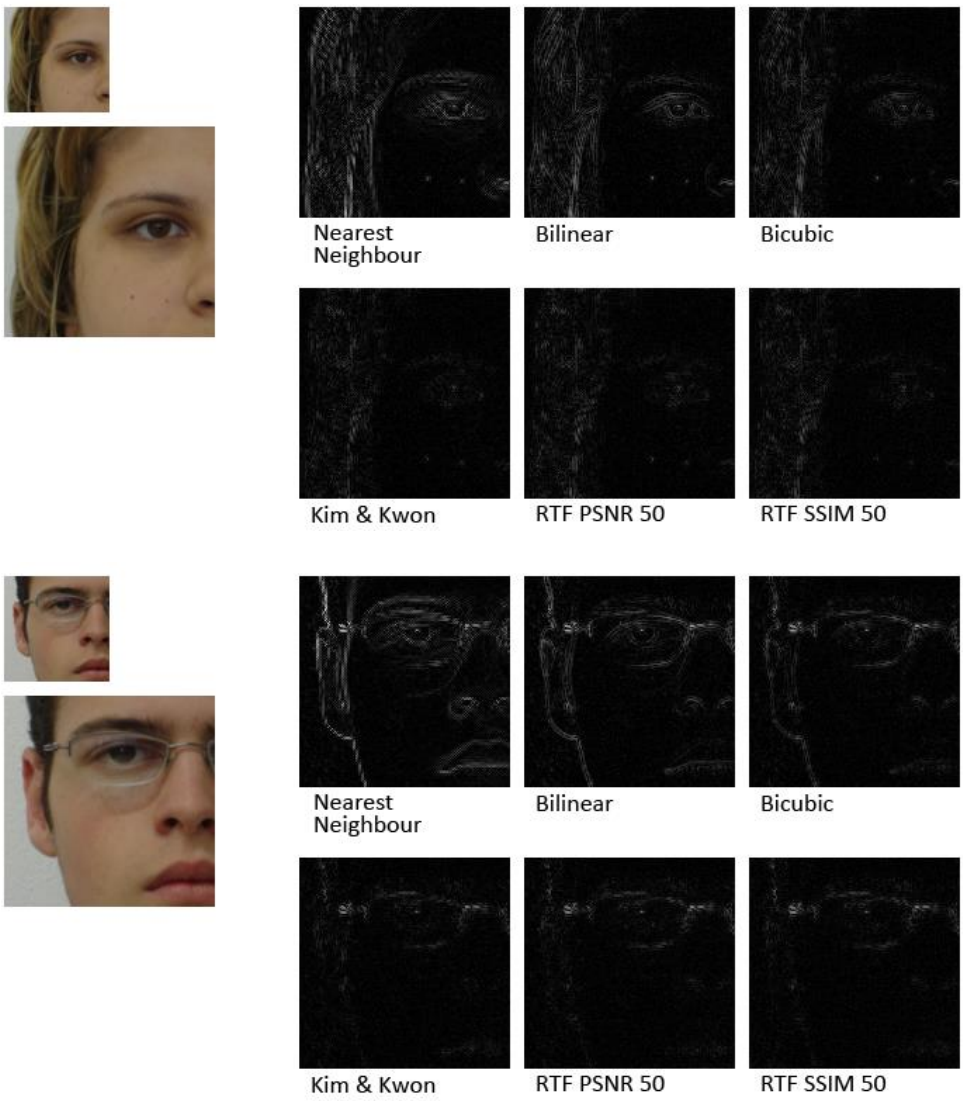

Figure 7 – difference images for the compared methods

Figure 7 shows, for two images from the test set, the input image, the ground truth image and the (somewhat lightened up) greyscale difference images between outputs from all 6 compared methods and the ground truth. Lighter colours mean more deviation from the ground truth. Note that for these images, results from the test run with 50 training samples were used, but the visual difference should be negligible.

It is interesting to see that in some places the output of *Kim & Kwon* [3] appears darker than the output of e.g. *RTF SSIM 50*, which performed better according to the measured loss values. This may have to do with the fact that difference images are closely related to the *mean squared error*, which in turn is closely related to the *PSNR* measure. When optimising for *SSIM*, the absolute pixel differences between output and ground truth are not necessarily minimised, since they are not the only feature for human perception of image differences.

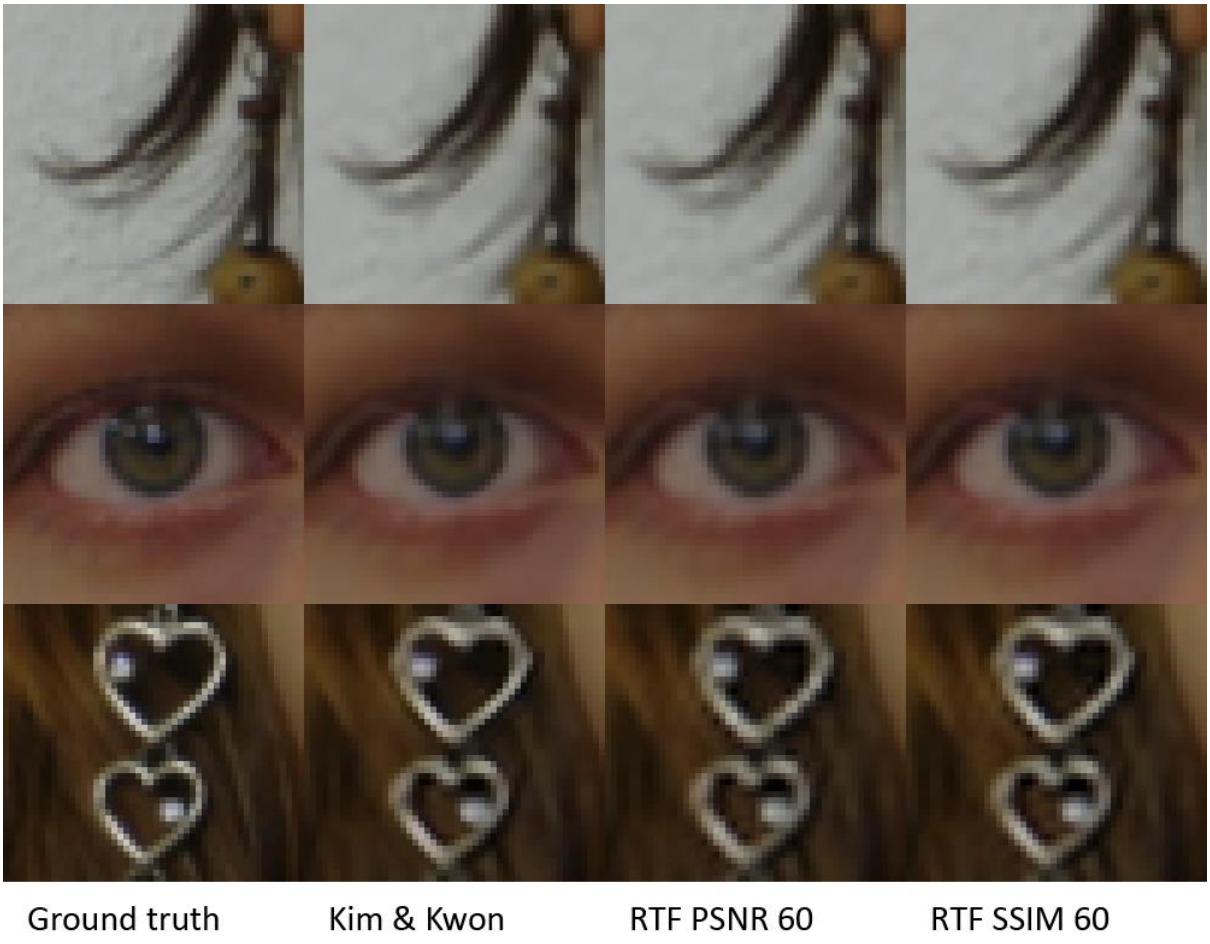

Figure 8 – Detail comparison

In **Figure 8**, some magnified regions of interest are shown to allow better comparison of detail. It is quite evident that the *RTF PSNR 60* results are blockier and less sharp than the others, despite using the *Kim & Kwon* results as part of their input. Between the latter and the *RTF SSIM 60* results, there is practically no visible difference in the upper two rows. The bottom row is a special case, as it contains a type of structure that does not appear in the training data. Since RTF rely on their training for the prediction, they perform visibly worse than [3] on new data like this.

The complete results for all test runs, including output images and measurements, are given in the appendix.

Discussion

Although the scope of the tests was somewhat limited, it could be shown that Regression Tree Fields are generally capable of Super-Resolution and could even improve upon state-of-the-art methods.

The main drawback here was that the 8 GB of main memory on the workstation used for test runs were not enough for the very demanding RTF training algorithm. Even with the comparatively small dataset, memory filled up quickly and this slowed the calculations down significantly. Without this limit, a sample size larger than 60 and, more importantly, a tree depth larger than 5 would have been feasible. It was described in [2] that a desirable tree depth might be 8, 9 or even higher, provided the data support this without risk of overfitting. More memory and processing power would also allow for additional inputs from other methods to be utilised.

As it is, I tried to incorporate results from one additional state-of-the-art method [5], but running the provided MATLAB code on my dataset caused the machine to completely freeze after a few hours of computation. Nonetheless, there is a wide range of existing Super-Resolution methods, and assuming enough power for the computations, any incorporated results should serve to make the RTF approach stronger by giving it more options to choose from.

As for the newly introduced binning feature, it turned out that this is quite difficult to track. The learning algorithm samples features at random when looking for split criteria for the regression trees, so it is hard to tell to what extent a specific feature influenced the result. To check this influence empirically, a large number of training procedures on the same data, half of them with and half without the feature, would have to be performed and the average results compared. This poses an enormous computational effort, which I am not sure would be worthwhile.

In addition to the partly trivial possibilities for improvement described above, the RTF framework also has support for a cascade-like mode called “stacking”. Several RTFs are trained in layers, each taking as input the output of its predecessor with the aim of correcting any remaining errors compared to the ground truth. A setup like this can be quite powerful. If used here without further changes to the code, though, it would double the size of the image with each round, which is not the desired effect. Adjusting this behaviour so that only the first round changes the image size might provide another increase in performance and could be the subject of a future project.

Similarly interesting but outside the scope of this project would be the extension to other scaling factors than 2. This would probably require a new approach altogether, as the direct mapping of input to (multiple) output pixels done here would not work for general scaling factors.

Lastly, the success of RTFs as a whole depends on the quality of the loss functions in use. Even if, under idealised circumstances, the optimal configuration and thus the most likely output image is found, the metric for that is still set by the loss function and might not line up with the perception of a human observer. Therefore, the development of better (yet still differentiable) loss measures would raise the bar on what RTFs can achieve and in doing so improve on the results reported here.

Sources

[1] J. Jancsary, S. Nowozin, T. Sharp and C. Rother: **Regression Tree Fields – An Efficient, Non-parametric Approach to Image Labeling Problems.** *2012 IEEE Conference on Computer Vision and Pattern Recognition, pp. 2376-2383, 2012*

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

[3] C. E. Thomaz and G. A. Giraldi: **A new ranking method for Principal Components Analysis and its application to face image analysis.** *Image and Vision Computing, vol. 28, no. 6, pp. 902-913, June 2010* [only Face Database]

[4] K. I. Kim and Y. Kwon: **Single-image super-resolution using sparse regression and natural image prior**. *IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 32, no. 6, pp. 1127-1133, 2010*

[5] W. T. Freeman and C. Liu: **Markov Random Fields for Super-resolution and Texture Synthesis**. *Advances in Markov Random Fields for Vision and Image Processing, c. 10, MIT Press, 2011*

Appendix

Provided with this document:

- the dataset used
- results of all experiments carried out
- the adapted source code of the RTF framework
- some .bat files and MATLAB scripts created in the process