Stan Marseille (4798740)
Mitchell Maassen van den Brink (4728068)
Jolle Verhoog (5171598)
Wouter Mooi (5128625)
Supervisor: Ayush Kulshreshtha
Programme: MSc Applied Mathematics/MSc Computer Science

Reproduction blog: Multi-view underwater image enhancement method via embedded fusion mechanism

Deep Learning (CS 4240) - April 14, 2024

---

1 Introduction

The use of deep learning is becoming more popular every day, as everybody has seen, for example, the increasing popularity of ChatGPT over the last few years. A specific branch of deep learning is image enhancement. Take for example the enhancing of ultrasound images [1], which is a method that is "cost-effective, flexible, and does not require ionizing radiation", according to the paper. Another paper that is cited over 3000 times is a paper where they aim at enhancing fingerprint images [2], where it is shown in an experimental setting that it "improves both the goodness index and the verification accuracy". A final example is the digital image enhancement and noise filtering by use of local statistics in [3], where the results show that the techniques developed in this paper are "readily adaptable to real-time image processing". Thus, this branch of deep learning has numerous applications and an ever-increasing potential.
In this blog, aim to reproduce the specific application of enhancing underwater images as done in [4]. These images are often distorted by "blurriness, colour distortion, and other degradation issues due to interference from light and complex underwater backgrounds". Thus, the authors came up with a neural network architecture that has the aim of enhancing these images: the multi-feature underwater image enhancement method via embedded fusion mechanism (MFEF). According to this paper, the main contributions are presenting a "three input-structure that obtains reconstruction results", propose a "novel multi-feature fusion (MFF) module that fully fuses the multi-view information streams, allowing multiple features to interact" and "design a pixel-weighted channel attention module (PCAM) that captures the interdependencies between channels and recalibrates the image's essential features according to the image's degradation level".
This reproduction was done on behalf of Lobster, a company in Delft, trying to map the seafloor. They use a robot that floats above the seafloor, taking a lot of pictures. When they combine these pictures, they get a map of the seafloor, which explains their interest in this technique. We will first give an outline of the architecture and how it is implemented in Python, with accompanying assumptions. Subsequently, we present the results and discuss them briefly in terms of what can be improved.

2 Methods and Architecture

Reproducing the result of the paper involved reproducing the architecture of the network, preprocessing mechanisms, and evaluation metrics as close as possible. The paper referred to publicly available datasets with images and reference images for training the network. However, no code was published by the authors. As such, we relied on various schematics and descriptions of the different modules, and the loss

functions. We reproduced the MFEF class, with its specific pipeline, and made three separate classes for the REM, PCAM and MFF modules. Also, we defined a loss function as described in the paper. At several stages we encountered difficulties as some architectural design descriptions led to persistent errors. In this section, we will discuss the different modules and justify design choices and their possible impacts where applicable.

2.1 Image preprocessing

To process our images through our modules we first need to preprocess the original images. The paper states clearly how the two offset pictures are created; one through white balance shift and one through contrast enhancement through CLAHE. The experimental values found by the researchers were exactly copied to our functions. Furthermore, the sizes of the images needed to be altered slightly. The model has a channel dimension of three and there is double down-/upsampling, so the pixel values should be divisible by twelve. This was done by finding the multiple of 12 closest to the original image, and then cutting off maximally 11 pixels on each side. For debugging under faster computing times, we also included a decrease variable that would further decrease the image size. This variable can be easily set to false when enough computing power is available. As a last step in the preprocessing, the images in the original folder are converted to four separate tensors. Three for the MFEF module and one for reference with the loss function.

2.2 Architecture of the MFEF module

The multi-feature underwater image enhancement method via embedded fusion mechanism (MFEF) is combining all the modules in one overarching architecture. The goal of MFEF is to improve underwater images.
The MFEF gets three images as input. The normal underwater image, the image with white balance and the image with contrast-limited adaptive histogram equalization (CLAHE), as explained in the preprocessing step. After running these images through the network, they all get concatenated and after a final convolutional layer, the final image is produced.
We ran into some issues when trying to reproduce the MFEF. As can be seen in figure 1, the result of one REM module gets concatenated with the result from the MFF module. This then gets put through another REM module, to which a pixel-wise addition is done with the input image. The problem is that these are not the same size. So, we assumed that we could add a tensor with multiple images stacked together to the result from the REM module.
Another assumption we made was on the dense connections before the PCAM module. These connections are not explained in the paper. Judging by the figure in the paper, we determined that each channel gets downsampled and the channels get connected by factor of downsampling. So, the original size is one channel, 2 times downsampled is one channel and the last channel is 4 times downsampled.
The last assumption is on the convolutional layer. There was nothing in the paper on this layer. So, we used a 3 by 3 kernel with padding 1 and 3 outgoing channels to get an RGB image back as a result.

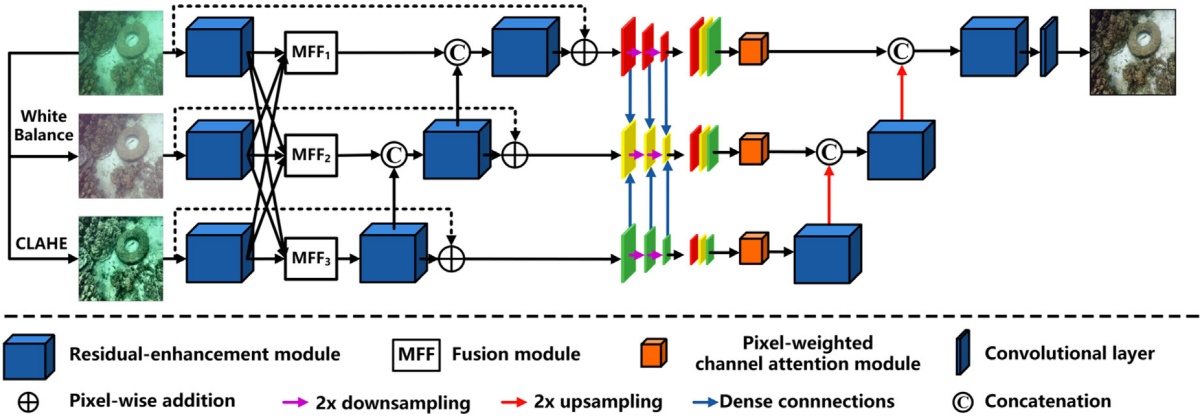


Figure 1: A schematic overview of the MFEF module.

2.3 Architecture of the Rem module

The REM Modules (Residual Enhancement Module) are positioned in three stages within the MFEF architecture, which are indicated by the blue boxes in figure 1. Firstly, at the initial feature encoding stage, secondly after the Multi Feature Extension (MFF) and thirdly at the final decoding stage. As described in the paper, in the first stage, the REM module is employed to extract encoded features from the three separate image streams: original, white-balance, and CLAHE (as they are already in tensor form). At the second stage, the REM Module is used to extract deeper features that emerged when the separate image streams interacted in the MFF module (MFF details are described in section 2.5). In the third stage, the REM is used with the multiscale decoder to reconstruct the resulting output.

The REM module is composed of two blocks with 4 convolutional layers and three GELU activations, alternating consecutively. In between the two blocks, the input is added to the output of the first block to form an intermediate result via pixel wise addition. All this can be seen in figure 2. The intermediate result is added to the final output via pixel wise addition again. For the convolutional layers, the kernels were set to 3x3 with a stride of 1. The GELU, or Gaussian Error Linear Unit activates both linear and non-linear patterns; this may contribute to more effectively capturing complex patterns in the image tensors. The number of convolutional layers (eight) also allow the model to capture complex patterns in the data, while balancing potential overfitting and vanishing gradient problems. The pixel wise addition allows for a direct flow of gradients upstream, refines features by combining the input with the output of the first and second block.

In the implementation, we used the PyTorch NN (Neural Net) module. The forward pass was straightforward and used consecutive operations. We used in place operators to promote efficient computing.

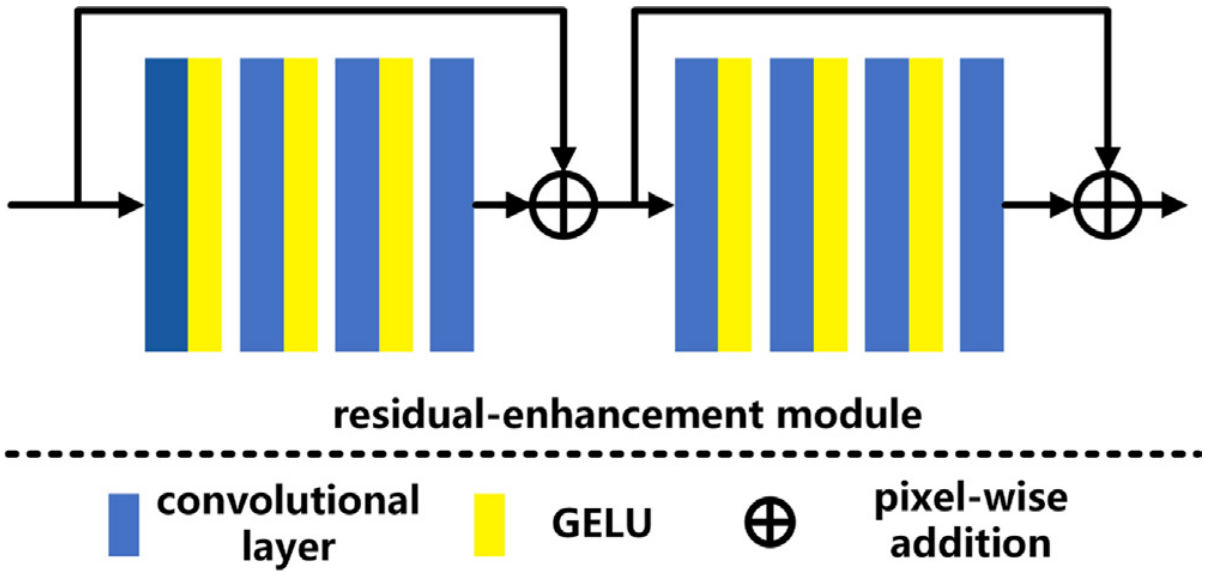


Figure 2: A schematic overview of the Rem module.

2.4 Architecture of the PCAM module

The Pixel-weighted channel attention module (PCAM), according to the paper, is introduced to 're-calibrate critical features' [4]. Through different operations features are assigned different 'levels' of importance to address the varying quality of images that the model may be exposed to. The PCAM is situated in each separate image stream before the final concatenations and REM stages, as shown in figure 1. As such, important features may be extracted and emphasized before an image if forwarded to the last (REM) stages of the network.

The forward pass consists of a series of convolutions and activations. A 1x1 convolution and 3x3 convolution run in parallel, to learn more detailed relations between spatial information (structure and relations between parts of the image) when the results are multiplied pixel wise. Following this, Global Average Pooling (GAP) is then introduced to let the network obtain more information about features with relevant information in all channels. In addition, various residual connections are used, these add the input features to outputs, preventing features from vanishing. A schematic overview of the module can be seen in figure 3.

In the development of this module, various functions from the PyTorch NN module were used. These allowed us to efficiently initialize the layers and implement the forward pass while preserving all the model parameters for backpropagation. Extra precaution was taken to use in-place operations and introduce little different parameters to minimize computing costs and reduce RAM load.

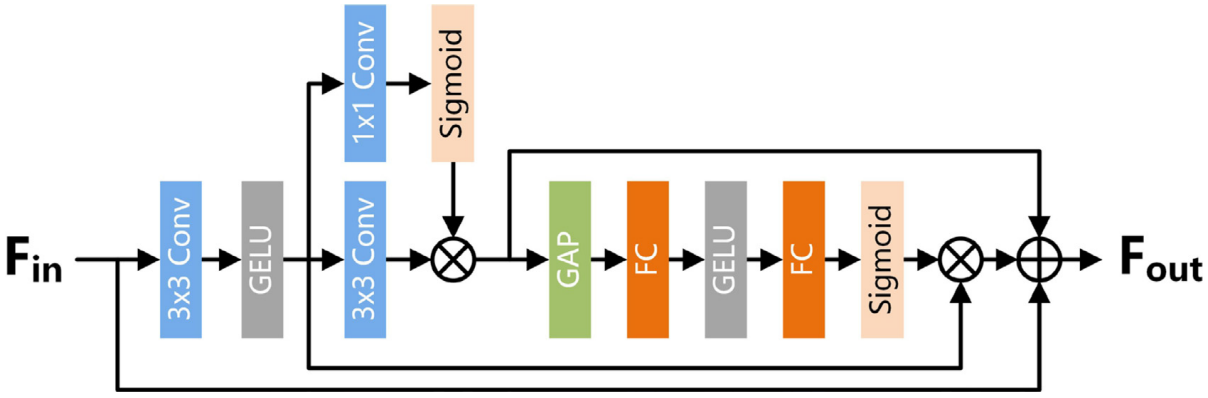


Figure 3: The architecture of the PCAM module.

2.5 Architecture of the MFF module

According to [4], the MFF module aims to fuse information flow of features in various stages. These differences in information flow occur because of the various ways that the images have been preprocessed. This namely yields different values for image brightness, colour, and details. Thus, the MFF module is trying to let the details that can be obtained from the various preprocessing methods interact and merge. In the MFEF architecture of figure 1, it can be seen that it is located after the first stage of Rem modules, which have been introduced before.

The various outputs of the first stage of REM modules are denoted by FB_1, FB_2 and FB_3. These inputs first undergo a 1×1, 3×3 and 5×5 convolution layer, with stride 1, which aims to merging characteristics from different picture views to keep unique and useful properties. After each of these convolutions, one layer of the GELU function is applied to handle how different features work together. A residual connection is made to avoid the vanishing of gradients and extract the actual features. These 3 streams are then concatenated in the channel dimension and a channel shuffle is applied. Because the different parts of the features are strongly connected, this shuffle makes sure all kinds of features work together well. Finally, a last 3×3 convolutional layer is applied, which yields the output. A complete overview of the module can be seen in figure 4.

All the several layers were available to us in the PyTorch NN module, which is convenient for reasons that have been mentioned before. Firstly, note that we assumed that after every convolutional layer, the output had to be padded. This made sense to us as it allowed for concatenation later on in the module. Secondly, the PyTorch NN module had a standardized method to apply a channel shuffle. The first problem arose when the paper did not specify what the number of groups were that the channel shuffle needs as argument. We assumed this was 3, as this made sense with the number of different inputs. The second problem was that the channel shuffle obstructed the complete MFEF module in making a backward pass. We therefore left the channel shuffle out as it was not possible due to time constraints to make this backward pass ourselves.

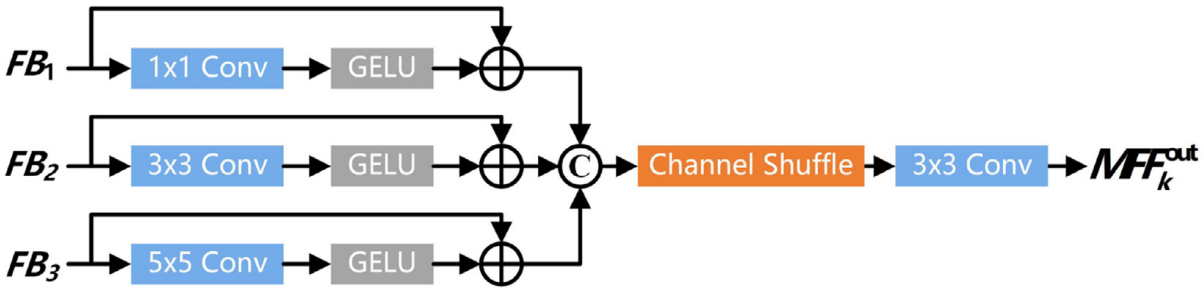


Figure 4: The architecture of the MFF module.

3 Pipeline

Now we will walk through the code we made to reproduce our paper. First, we imported all necessary libraries to make our code run. Then we could define all modules we explained earlier. To train our network, we needed the training data. Luckily, the training data was made available by the writers of the original paper. So, we could download these photos and upload them to our workspace. But this is a lot of data to process, and we quickly realized that we ran out of RAM, when trying to train our network. To speed up the training process, we also wanted it to be able to run on GPU. That is why we pre-processed all the images before training, as explained before, and made tensors of all the data. This is more efficient because the CPU does not have to do it during training. Now, we could just iterate over the data without using too much working memory. This data is then split into training and testing data, which is used to train our model. After training, we could use our model to see if it can improve underwater images. The input of our model, as explained before, is three images. After running these through our network, we are left with a tensor. This tensor can be converted back to an RGB image.

3.1 Method of training

For the training of the network, we made use of CUDA to train locally with GPU support. In [4], the following loss function was used

$$L = L_{l_1} + 0.05 L_{per}, \tag{1}$$

where the first term denotes the $L1$ loss and the second term is called the perceptual loss function. The $L1$ loss has is implemented to keep the sharp artefacts in borders and empirical image quality. The perceptual loss is defined as

$$L_{per} = \sum_{x=1}^{H} \sum_{y=1}^{W} |\phi_j(J)(x,y) - \phi_j(R)(x,y)|, \tag{2}$$

where ϕ_j represents the jth convolutional layer in the VGG-16 network pretrained on the ImageNet and J and R are the measured reconstruction result and ground truth, respectively. This absolute difference is then summed up over all the pixels of the pictures. The $L1$ loss is a standard method in the PyTorch NN module, making it easy to implement. However, we had to make a separate class for the perceptual loss, as this was not available as a standard function in PyTorch. Luckily, we could import the VGG-16 network, making it relatively easy to realize this loss function.
As an optimizer for the model parameters, we used the Adaptive Moment Estimation Weight Decay Optimizer (AdamW), as [4] states it requires fewer memory resources and solves the problem of large-scale data and parameter optimization. Just as in the paper, we set the learning rate to 0.0001. We additionally used a learning rate scheduler (something that the paper did not use), to increase performance and reduce training time. This scheduler decreases the learning rate according to a predefined schedule as the number of epochs passes.

4 Results

After completing the reproduction of the network's architecture and testing its ability to forward pass an image, we trained the network using the 'raw' image dataset and the accompanying reference images.

4.1 Training results

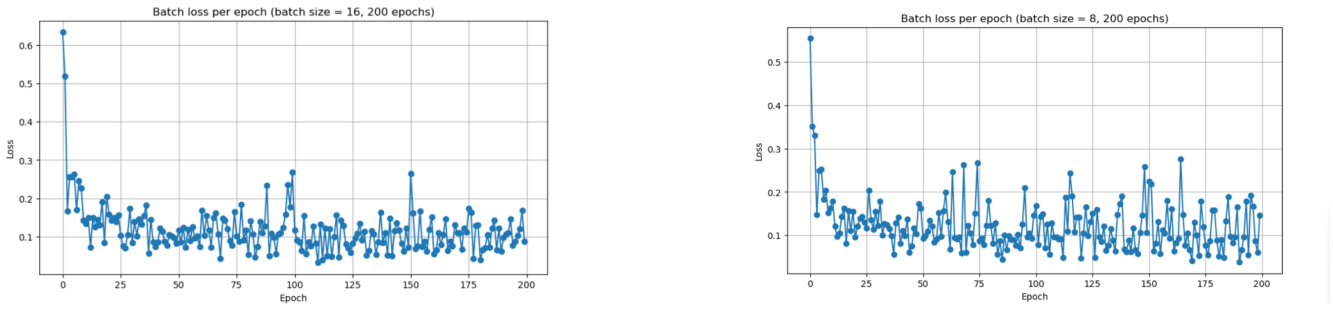
Using the training loop as described, several training efforts were made. We implemented a method in the image preprocessing pipeline that allowed us to resize the images to a lower resolution. In this way we were able to incrementally increase and vary: the image resolution, image dataset size, number of epochs, batch size and learning rate. This ensured we could test and evaluate both our computing power and the effects of these parameters on the network's performance. The process was along the lines of the following:

1. We started with 200 raw and reference images (400 in total), used a batch size of 16, a learning rate of 0.001, 200 epochs and a resolution of 360x360. We concluded that the training loss converged quickly. The iteration time per batch was around 30 seconds.

2. Because of the apparent convergence, the number of epochs was maintained, but the image resolution adjusted to the original image size. Because some of the raw images and accompanying reference images did not have the same resolution, we encountered an error in the training loop. The loss was undefined as a result of the different tensor lengths. As a result, we decided to set the image resolution to 720x720, where most images were around 800 to 1200 pixels in height or width. We acknowledged that this decision may have a significant impact on the models' performance on certain, or all images, but we were unable to implement a better fitting solution due to the time constraints of this project.

3. In the training experiments that followed, we experimented with the batch size, when training on 400 and 700 images (per tensor). Logically, a smaller batch size led to a shorter iteration time per batch. With a batch size of 8, the time per batch was about 15 seconds. When attempting to process larger batches of 32 or more, the GPU was unable to complete 1 iteration after 5 minutes. As such, that training was stopped.

4. We noticed that the model converged balanced during the 200 epochs (as was specified in the paper), when the batch size was set to 8 or 16 with a learning rate of 0.001. As such, we decided to train the model using 730 images (total set size was 890 images), the rest was retained for testing and evaluation. The results of these are discussed in the 'Discussion' section. In addition, two plots with the batch loss per epoch during training for batch size 8 and batch size 16 are included and can be seen in figures 5a and 5b.



(a) Batch size: 8

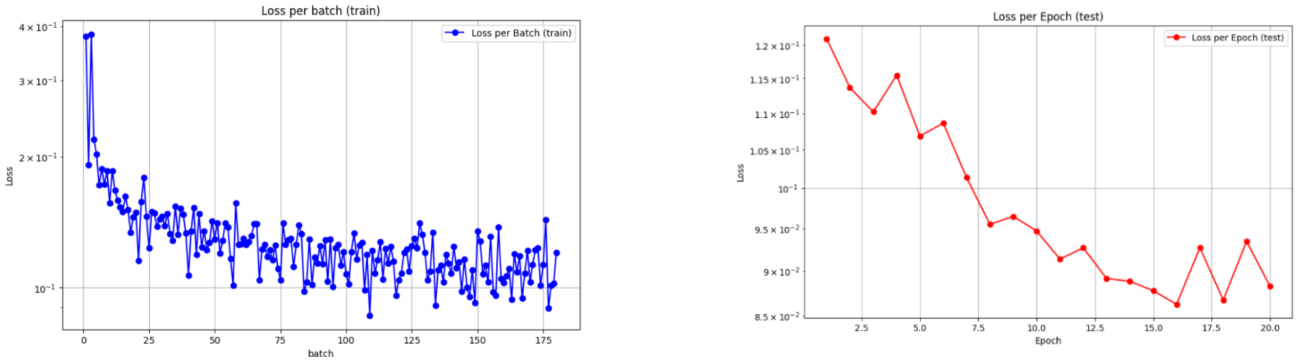
(b) batch size: 16

Figure 5: Training losses with batch size 8 (left) and batch size 16 (right)

5. Also, a learning rate scheduler (StepLR, PyTorch) was implemented to incrementally decrease the learning rate as the number of iterations increased. The initial learning rate was still 0.001, but was reduced by a factor of 10 every 25 iterations. Considering the complexity of the model, decreasing the learning rate at higher iterations may reduce overshooting while maintaining stability and performance. Overall, leading to a lower loss, and hopefully providing better generalization.

4.2 Test results

The training results seemed excellent, even when the resulting pictures are unsatisfactory. A hypothesis was that the model might be overfitting, but judging by the test results, this was not the case. As can be seen by computing the loss on unseen test data: both the training and test loss goes down rapidly after each epoch. The results of the losses can be seen in figures 6a and 6b. These losses are also way lower than the losses in the original paper, so we suspect there might be something wrong with our calculation of the loss.



(a) Training loss after each batch.

(b) Test loss after each epoch.

Figure 6: Training losses (left) accompanied by test loss after each epoch (right).

4.3 Visual results/actual outputs

By running our trained model on pictures of the test set and visualizing the results, we wanted to evaluate our actual outputs. When we did this with our most trained model, it seemed all outputs had a preference for the red channel, as can be seen in figure 7 in the appendix. The images were not blurry and the features of the original seemed to be preserved, and arguably there were more defined edges of objects within the image. We thought the dataset might have an under-representation of the red channel, explaining the deposition towards it. Thus, we calculated the mean and variance of each channel in the whole dataset and normalized all pictures with these. When retraining the model with these pictures, the output images seemed to all have a disposition towards the green channel, which can be seen in figure 8 in the appendix. All in all, the pictures were successfully passed through the network, with some features being more distinguishable in the output, but the output images seemed to have a disposition towards one of the colour channels.

5 Discussion

The undertaking of reproducing the multi-view underwater image enhancement method via embedded fusion mechanism (MFEF) offered a unique insight into the challenges of reproducing publications, where the exact details or the pseudocode is not provided. We also encountered these challenges of non-available code and vague descriptions of certain modules and processes. These issues underscore the importance of comprehensive documentation and sharing of resources. We tried to navigate these gaps by making educated assumptions, like resizing images, and sometimes omitted complex functionalities like the backward

pass of channel shuffling.
The proposed MFEF architecture appeared to be complex, but our own written code seemed to yield results, at least in part. In the earlier stages of implementing the MFEF and MFF module we encountered problems with the channels shuffle layer during backward passes, gradients could not be assigned to the right features. Due to time constraints, we decided to surpass this layer. Even though this made the network functional, we realized this choice has had an impact on its capability to learn interdependencies in between the channels. In addition, we persistently encountered problems as a result of the differences in image resolutions between the raw (features) and reference images. As some couples had differences in their resolutions, the image tensor dimensions were not always equal, resulting in loss measurement errors. To mitigate this problem, we resized images to 720 x 720 pixels. This choice resolved the error but removed parts of the features, which also impacted the effectiveness of training and probably overall generalization.
In the MFEF, a lot of steps were lacking proper documentation. That resulted in us having to make big assumptions on how the tensors were concatenated or added to each other. This has a big influence on the results of our model, which is an explanation of why the network performs poorly.
As for the results, we ran into some issues with colour channel dominance in the output images. We believe that our adjustments to the network with respect to the original contributed to this, but also suspect other factors may have played a role. Finding the origins of certain outcomes was difficult due to the complexity of the model and the time constraints. We tried to address these issues through normalization and model adjustments, but these measures were to an extent unsuccessful. We managed to significantly reduce both training and testing losses, suggesting that our model was successful to a certain degree. However, we also think there might be an issue with the loss function as our losses were lower than the original paper, while our output images looked worse. If we had more time, we would explore more advanced techniques in balancing colour channel sensitivity and further refine the loss functions to better capture perceptual quality.
In conclusion, the reproduction of the paper was not completely successful. We followed the methodology in the paper closely, but we think we were missing some knowledge about image enhancement, and the paper also lacked information for us to fill in these gaps. Our work emphasizes and demonstrates that exact reproductions of academic findings may prove difficult when the authors design choices are not completely unambiguous.

A Appendix

A.1 Division of work

Stan:

1. Implementation of REM and PCAM module
2. Training loop design and implementation
3. Testing and debugging architecture before implementation
4. Training experiments and model training
5. Blog post: REM, PCAM, Training loop, part of discussion, and small other parts

Mitchell:

1. Implementing the MFF module
2. Partly creating the train and test iterations.
3. Writing the parts of the blog about the MFF module and the method of training
4. Converting the blog to LaTeX.

Wouter:

1. Making the MFEF.
2. writing the blog parts of MFEF, pipeline and test results.

Jolle:

1. Creation of preprocessing functions.
2. Normalization functions.
3. Loss function and visualization.
4. Testing, debugging and visualizing the training of the model.

B Visual results

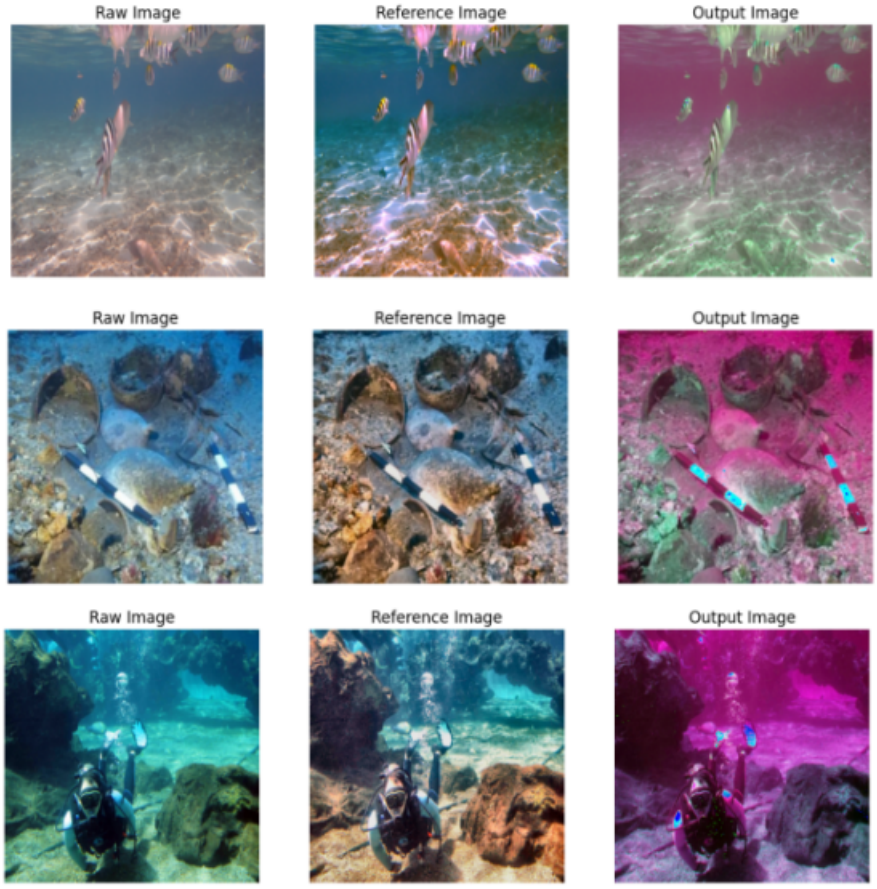


Figure 7: Some visual results, where it can be seen that the red channel is dominating.

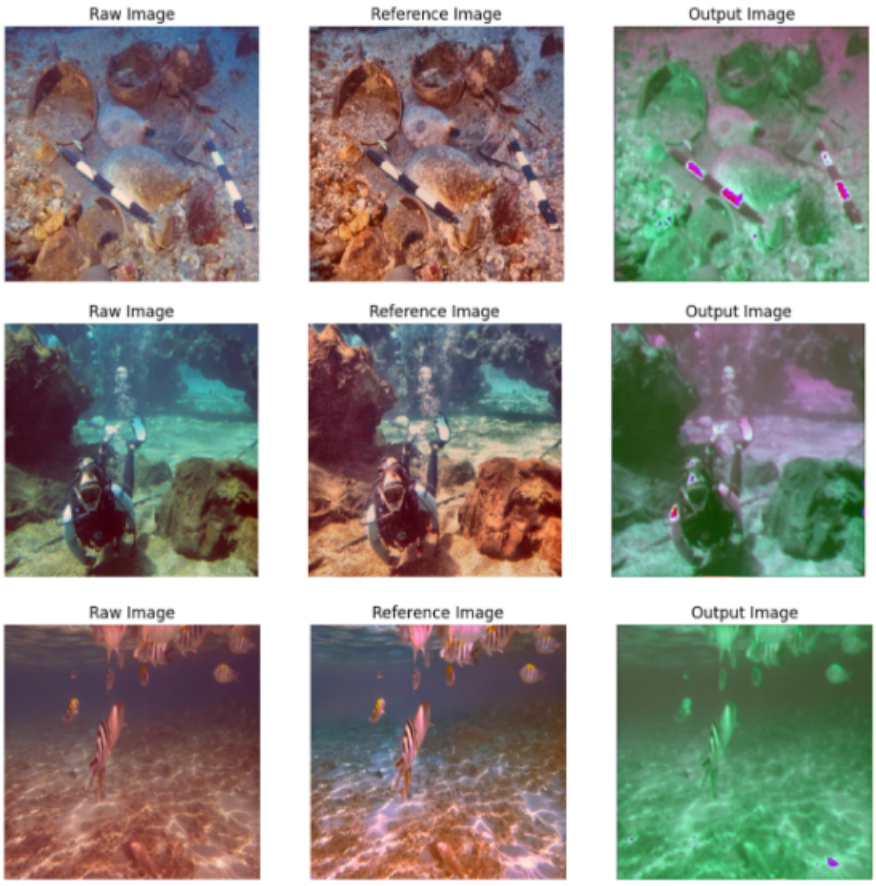


Figure 8: Some visual results after retraining, where it can be seen that the green channel is dominating

C Link to GitHub

The code used in this blog can be found here.

References

[1] S. H. C. Ortiz, T. Chiu, and M. D. Fox, "Ultrasound image enhancement: A review," *Biomedical Signal Processing and Control*, vol. 7, no. 5, pp. 419–428, 2012.

[2] L. Hong, Y. Wan, and A. Jain, "Fingerprint image enhancement: algorithm and performance evaluation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 20, no. 8, pp. 777–789, 1998.

[3] J.-S. Lee, "Digital image enhancement and noise filtering by use of local statistics," *IEEE transactions on pattern analysis and machine intelligence*, no. 2, pp. 165–168, 1980.

[4] J. Zhou, J. Sun, W. Zhang, and Z. Lin, "Multi-view underwater image enhancement method via embedded fusion mechanism," *Engineering applications of artificial intelligence*, vol. 121, p. 105946, 2023.