McMaster University

COMPENG 4TN4 - Image Processing

Liam Luimes

400322938

luimesl@mcmaster.ca

https://github.com/LuimesLiam/demosaicing

Project Phase 2
Image Processing Pipeline

Due Date: April 30^{th}, 2024

Contents

1 Introduction

Many processes go into the construction of images taken on a digital camera before the image can be used. This set of processes together is called the "image processing pipeline". This pipeline is implemented from scratch in this paper, mainly demosaicing (see Phase 1 for a more in-depth look), histogram equalization, and white balancing. All of these methods will be discussed in depth in this paper, as well as a few methods compared and contrasted.

This paper explores two main demosaicing methods, namely a linear regression method where two kernels are learned to interpolate the R, G, and B values from the four main raw Bayer patterns [1]. The other implementation is Deep demosaicing, where a convolutional neural network is used to learn how to properly reconstruct a full colour image. These implementations are trained on a simulated data set that is artificially injected with Gaussian noise to better reflect real-world data. Since both the methods here are implemented with a learning style approach, these methods learn how to denoise while demosaicing. Therefore, it is not necessary to implement a denoising filter before the demosaicing process.

This paper then implements a rough Contrast Limited Adaptive Histogram Equalization (CLAHE) method as another step in the image processing pipeline. This method is tested on both simulated low light data, and short capture images taken in low light settings.

This paper also implements a simple gray-world white balancing approach to the demosaiced image. This paper discusses the strengths and weaknesses of this simple approach, and what other methods could be used to improve the white balancing step. To test this method, images are hue-shifted, and then plugged into the white balancing algorithm.

Please note, the GitHub repository will remain private until after the completion of this class to not allow students in the same class to copy this work. .

2 Background

2.1 Image Processing Pipeline

The image processing pipeline serves as an essential framework for transforming raw data captured by a digital camera's sensor into a coherent, visually appealing, and usable digital image. This sequence of operations is meticulously designed to interpret and refine the light and colour data recorded by the camera into an accurate representation of the scene as perceived by the human eye. Without such processing, the raw data would remain a muddled array of pixels, far from the clarity and detail we expect in modern digital imagery.

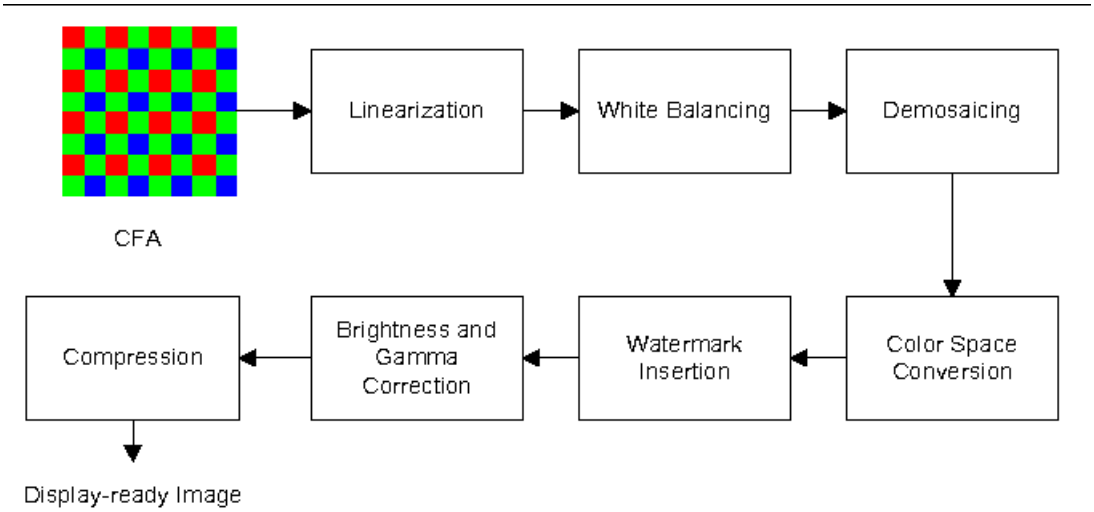


Figure 1: Image Processing Pipeline

The pipeline starts with a raw Colour Filter Array (CFA) image captured from the camera sensor array. This holds all the information about the scene but is not yet refined enough to make sense of it [2]. The raw data from the sensor is linearized to correct for the nonlinear response of the sensor and the optics [3]. This step ensures that the recorded pixel values are proportional to the light intensity [3]. Then either the white balancing or demosaicing can take place. White balancing is applied to ensure that objects that appear white in person are rendered white in the photo [3]. This compensates for the colour temperature of different light sources. The demosaicing process interpolates the raw data from the CFA to produce a full-colour image [4]. Since each pixel only records one colour, this step estimates the other two colours for each pixel by using information from the surrounding pixels [5]. Once the image has full-colour information, it's converted from the camera's colour space to a standard colour space [3]. This ensures the colours are displayed consistently across different devices [3]. This step will be neglected in this paper's implementation. Brightness adjustment is made to match the desired output level, and gamma correction is applied to adjust the luminance to suit the display's characteristics [3]. The image is then compressed to reduce the file size for storage or transmission. Once again, for the sake of simplicity of this paper, no compression will be attempted.

2.2 Demosaicing

Demosaicing is a process used in the digital image processing pipeline to reconstruct a full-colour image from Bayer patterns. A digital camera will collect the data for the photo using a colour filter array (CFA). This is essentially an array of photo sensors that have red, green, and blue filters to allow for those colours to only pass through and hit the photo sensors [5].

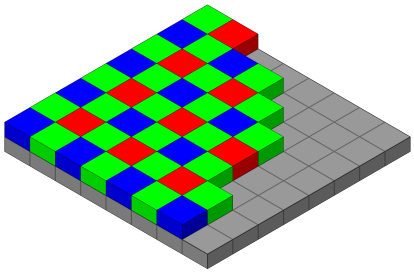


Figure 2: CFA

This raw image will have a mosaic of colours that appear discrete and uncoordinated. To get colour from this CFA image, we need to perform the demosaicing task in the pipeline. This demosaicing task can be done with various approaches, some of which are implemented in this paper. The main idea is to predict the missing R, G, or B value given the pixels in near proximity to the center missing pixel [4].

Figure 3: Raw CFA vs. Demosaiced Image

CFA typically comes in four pattern configurations- RGGB, BGGR, BGRG, and GRBG [4]. The demosaicing algorithm will have to know what pattern is being used before it can demosaic the image. The programs implemented in this paper have options to select between these four patterns.

2.3 White Balancing

White balancing ensures that colours appear natural and accurate in the image, regardless of the lighting conditions under which the photo was taken. It corrects for colour casts caused by different light sources (e.g., daylight, incandescent, fluorescent) that may introduce unwanted colour tints [6]. One way to do this is to implement the "gray world approximation". The Gray World approach assumes that in an average scene, the average colour of the image should appear as a neutral gray. By adjusting the colour channels such that their average becomes gray, it aims to correct for any colour casts present in the image [7]. While it's a simple and intuitive method, it may not always produce accurate results in scenes where the assumption of an average gray world does not hold true, such as scenes dominated by a single colour or scenes with strong colour casts. Nonetheless, it serves as a basic and effective starting point for white balancing in many applications.

2.4 Histogram Equalization

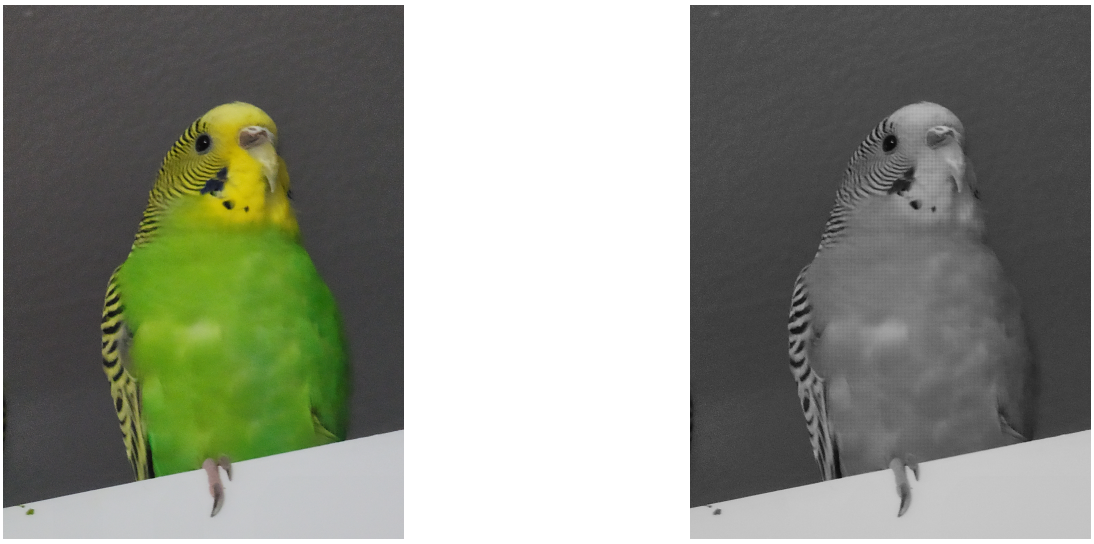
Histogram equalization is a technique used to enhance the contrast and visibility of an image by redistributing the intensity values of pixels [8]. It works by stretching the intensity range of the image histogram to cover a wider range of values, thereby increasing the dynamic range of the image and improving its overall appearance. Contrast Limited Adaptive Histogram Equalization (CLAHE) is a variant of histogram equalization that operates locally on small regions of the image. It divides the image into tiles or regions and applies histogram equalization independently to each tile. This approach prevents over-enhancement of local details by limiting the contrast enhancement within each tile, thus avoiding the introduction of artifacts while still improving the overall contrast of the image [9].

3 Data Set

This section covers the data sets used in training and evaluation. Much of the data used in this paper are simulated data, i.e. full colour images are converted to a Bayer pattern CFA image, Gaussian noise is applied, then darkened or hue shifted. In conjunction with simulated data, raw images from an existing data set were also used. See each section below for more details regarding the respective techniques.

3.1 Mosiacing Algorithm

Four different mosaic pattern modes can be selected. Each selected pattern specifies a different arrangement for sampling colour channels from the original RGB image. Each function works by iterating through the pixels of the RBG image and assigning colour values to the corresponding pixels of the mosaiced image according to the mode. Modes can be selected between CFA pattern types RGGB, BGGR, GBRG, and BGRG. See 7.1 for a code snippet, or "sim.py" in phase 2 repository for code.



(a) Ground Truth

(b) CFA Simulation

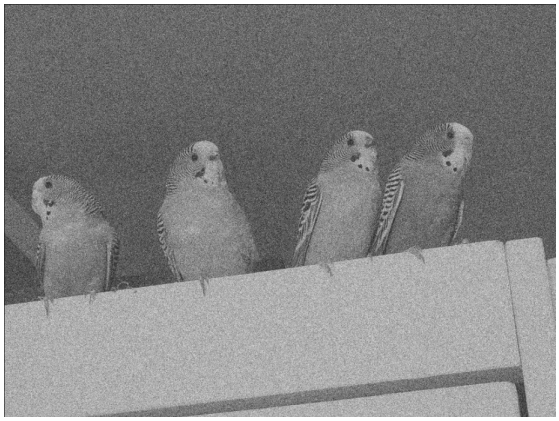
Figure 4: CFA simulation and ground truth

Gaussian noise is then added to each colour channel, varying by different random strengths. The different standard deviations are applied to each colour channel, as green tends to have the least noise, red the second least, and blue has the most noise [10]. Each of these standard deviations is randomly generated around their respective base S.D (i.e. green is 1, so between 0.9 and 1.1). This randomness allows the dataset to more realistically model the real world. See 7.2 for a code snippet, or "sim.py" in phase 2 repository for code.

Below is shown a demo of the noisy data, ground truth, one with normal noise (i.e. S.D around 1,2,4 to R, G, and B respectively) and exaggerated noise with each S.D multiplied by 100.

(a) Ground Truth

(b) CFA Normal Noise



(c) CFA Times 100 Noise

Figure 5: CFA simulation with noise

3.2 Existing Data Set

Other data was used, particularly from an existing open-source data set provided for this course project. This dataset consists of long and short-exposure image pairs. These images were taken in low light settings, so this data will be used to test and train for histogram equalization. The long-exposure images will be used as ground truth, and the short-exposure images will be used as inputs. These images are all 16-bit, where as the methods implemented in this paper rely on inputs being 8-bit images, so these images will be scaled down to 8-bit.
The dataset used in this paper is linked here: `https://paperswithcode.com/dataset/sid`

4 Demosaicing

Here are two demosaicing methods used in this paper. Other methods have been explored in Project Phase 1 where they were compared and ranked. This paper focuses on learning methods, so the linear regression model used in phase 1 is applied here, as well as a new neural network model was built specifically for this paper. Both models will be covered below.

4.1 Linear Regression

One approach to demosaicing is to convoluted a filter over the raw CFA image to interpolate pixel values. To get the best matrix for interpolating the image, linear regression can be used. In this approach, a 5x5 matrix for A and B is initialized with random numbers. These matrices serve as starting points and will be refined with linear regression. Various methods were tested when implementing linear regression, but a simple model using PyTorch regression was implemented. Earlier attempts implemented training until a threshold gradient vector length was researched, but it was found this overtrained the model and was too cumbersome to train, so that method was abandoned.

4.1.1 Regression Model

The regression model uses randomly initialized 5x5 (or any shape the user chooses) convolves it over the image and stores the output. If the user has the option to input a custom matrix as a starting place. The model internalized its components to begin the regression. It also pads the image to allow for the model to convolve using the "same" settings so that the output image will be the same size as the input image. This model uses the PyTorch built-in SDG optimizer. The image is then compared to ground truth using a mean square error loss function from the PyTorth libraries. The model will continue to train for a default of 6000 epochs or until an error threshold is met. The linear regression training is implemented as seen in "linear_regression_V2.py".

4.1.2 Demosaicing

This demosaicing algorithm is the simplest so far. It simply applies the learned A and B matrices to the image via a convolution to their respective colour channels, then stacked the results as a output.

```
R_m, G_m, B_m = masks_CFA_Bayer(CFA.shape, "RGGB")
R = CFA * R_m
G = CFA * G_m
B = CFA * B_m

red = convolve2d(R,RK,'same')
green = convolve2d(G, GK,'same')
blue =convolve2d(B,  RK,'same')
```

To further improve the image quality, a Gaussian filter can be applied if the user wishes.

```
smooth =  np.array([
[1,  2,  1],
[2, 16,  2],
[1,  2,  1]]
)/28

red = convolve2d(red,smooth,'same')
green= convolve2d(green,smooth,'same')
blue = convolve2d(blue,smooth,'same')
```

The simplicity of this algorithm means it's much faster than multiple convolutions as in the other algorithms, however may experience some undesired artifacts or aliasing.

4.1.3 Using The Program

To use the program is simple. The main demosaicing algorithm is in the "demosaic_lin_reg.py" file. To run the demosaicing algorithm simply use the below function imported from file mention. The function definition is as such:

```
demosaic(CFA, RK, GK, smoothing= False, norm=True, display=False,
    balanced=False, he=False, pattern ='RGGB' )

    returns output image
```

- CFA
  - The input image to de demosaiced.
- RK
  - The matrix used to interpolate the Red and Blue values
  - This matrix can be loaded from the "outputs/matrix" director.
- GK
  - The matrix used to interpolate the Green value
  - This matrix can be loaded from the "outputs/matrix" director.
- Smoothing
  - Apply a Gaussian filter to smooth the image
- norm
  - Apply a normalization to the output image
- display
  - Built in display function to show output image before returning
- balanced
  - Built in gray world white balancing
- he
  - Built in histogram equalization
- pattern
  - The type of Bayer pattern the image is

4.1.4 Results

This model was trained and run on Bayer CFA patterns RGGB, BGGR, GRBG, and GBRG. The output matrices are recorded as an example and shown here with an example output demosaiced image.

4.1.5 Output Matrices

RGGB

$$A = \begin{bmatrix} -0.00222192 & 0.00441482 & 0.00694215 & -0.05285718 & -0.00718026 \\ -0.0828956 & 0.00156459 & 0.23053804 & -0.004256 & -0.06093271 \\ 0.00176693 & 0.4623801 & 0.9856857 & 0.46688917 & 0.02074038 \\ -0.06303059 & 0.0118576 & 0.26344866 & -0.01037153 & -0.09590347 \\ -0.00933838 & -0.04106568 & 0.01058885 & -0.02172916 & -0.00552835 \end{bmatrix}$$

$$B = \begin{bmatrix} -1.6567669e-03 & 6.4156048e-02 & 1.3691127e-02 & -6.9852732e-02 & -1.7359730e-02 \\ 6.4814158e-02 & 3.5407442e-01 & 4.4436640e-01 & 1.8543483e-01 & 1.6577134e-04 \\ 1.8057724e-02 & 5.0642484e-01 & 9.5124918e-01 & 5.2959335e-01 & 3.7711043e-02 \\ -1.2587789e-02 & 2.3014025e-01 & 5.0957519e-01 & 2.8803253e-01 & 2.5533281e-02 \\ -1.7352087e-02 & 8.6155478e-03 & 3.5944175e-02 & 5.5804588e-03 & -2.0759631e-02 \end{bmatrix}$$

BRRG

$$A = \begin{bmatrix} 8.8047152e-03 & -1.3397861e-01 & -3.0274736e-02 & 1.6118576e-01 & -7.5177802e-03 \\ 1.0765021e-01 & 1.0780372e-01 & 2.7534381e-01 & 8.1826636e-04 & -2.1394865e-01 \\ -1.0232225e-01 & 2.0207496e-01 & 9.0032148e-01 & 4.3023455e-01 & 9.6992619e-02 \\ -1.6738902e-01 & 4.4565454e-02 & 3.3860767e-01 & -2.3990756e-02 & 4.0859189e-02 \\ 9.4118891e-03 & 9.5164590e-02 & 5.4547351e-02 & -1.2540291e-01 & -5.7237130e-02 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.01877008 & 0.03925351 & 0.0979136 & -0.01226373 & -0.06716131 \\ 0.01294238 & 0.18150336 & 0.5634988 & 0.31833664 & -0.10302378 \\ 0.01881674 & 0.36633795 & 0.8596393 & 0.5343497 & 0.08629613 \\ 0.03207455 & 0.18017216 & 0.41418955 & 0.31058827 & 0.06377131 \\ 0.02798093 & -0.01822321 & 0.01742865 & 0.14097276 & -0.01318185 \end{bmatrix}$$

GRBG

$$A = \begin{bmatrix} -3.2880850e-02 & -5.1339932e-02 & 3.6212612e-02 & 8.8264540e-02 & 2.3581486e-03 \\ -1.1228259e-03 & 7.7020624e-03 & 2.5487125e-01 & 1.2820997e-02 & -5.8575712e-02 \\ 2.2712197e-02 & 3.8152602e-01 & 8.8551021e-01 & 1.8844926e-01 & 2.5799152e-02 \\ -1.5419854e-01 & 3.3655711e-02 & 3.7573346e-01 & 1.3149377e-03 & -8.7610405e-04 \\ -9.1066081e-03 & 3.4600355e-02 & 3.1932961e-02 & -6.1877400e-02 & -2.0866711e-02 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.03579692 & 0.15680078 & 0.05175698 & 0.04043378 & 0.06121258 \\ -0.03162151 & 0.23945199 & 0.4323217 & 0.30452368 & 0.07642691 \\ 0.3196559 & 0.36402962 & 0.6927807 & 0.53642434 & 0.15219675 \\ 0.2374772 & 0.474933 & 0.71950555 & 0.34725773 & -0.04373903 \\ -0.08441237 & 0.21221964 & 0.22678946 & -0.02877011 & -0.10052066 \end{bmatrix}$$

GBRG

$$A = \begin{bmatrix} 0.0273015 & -0.02541049 & -0.02291533 & 0.02603629 & -0.03728085 \\ -0.05174419 & 0.02760636 & 0.29843616 & 0.0682372 & -0.06839382 \\ -0.02932595 & 0.3233795 & 0.9233628 & 0.3867395 & 0.00500673 \\ 0.0118042 & 0.08482323 & 0.30386928 & 0.01560433 & -0.10365652 \\ -0.02815492 & -0.09627023 & -0.03000841 & 0.0112351 & 0.01556513 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.04845999 & 0.09117012 & 0.10936663 & -0.0077738 & 0.02255463 \\ 0.09364557 & 0.21055977 & 0.4567533 & 0.37662357 & 0.00561293 \\ 0.01926018 & 0.4343485 & 0.68812054 & 0.38444617 & 0.03213771 \\ -0.06692138 & 0.34802485 & 0.47294632 & 0.0554525 & 0.01446032 \\ 0.04451301 & 0.11907248 & 0.17561613 & 0.0377758 & -0.04305771 \end{bmatrix}$$

4.1.6 Bird Demosaicing

The result of this interpolation is visually very close to the ground truth image. The mean square error to ground truth is 1.7863e-3, which is relatively good for such a simple demosaicing process described in section 4.1.2. This demosaicing was done on a simulated noisy image with standard deviations of around 1, 2, and 4 for R, G, and B respectively.

(a) Ground Truth Bird

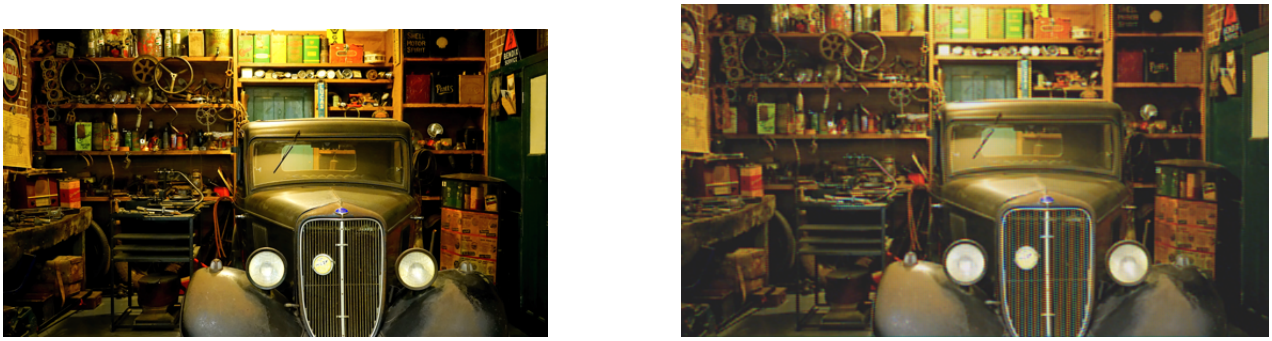
(b) Demosaiced Bird

Figure 6: Linear Regression Demosaic

4.1.7 Truck Demosaicing

The results of this more complicated image seem relatively good. This demosaicing was done on a simulated noisy image that was not in the training dataset with standard deviations of around 1, 2, and 4 for R, G, and B respectively. The pixel artifacts in the tool in the background and chrome areas still exists, which is expected due to the simplicity of the algorithm. This image is more complex compared to the one in figure 6.

The mean square error of this result is 3.946e-2.



(a) Ground Truth Truck

(b) Demosaiced Truck

Figure 7: Linear Regression Truck Results

4.1.8 Extreme Case

To test the power of this learning model, the linear regression model will be trained on a small data set of 10 very noisy images. The images have a average standard deviation of 20,40,80 for R,G,B respectively. The model is then tested on a image with the same simulated S.D, but one the model has not trained on.

Results
The results are surprisingly well, with a fantastic visual appearance, and with a mean square error of 5.63e-3 to ground truth.

(a) Extreme Noise

(b) Demosaiced

Figure 8: Linear Regression Extreme Noise

4.2 Deep Demosaicing

This section uses a Convolutional Neural Network (CNN) to demosaic the raw Bayer image. This model is trained on the noisy simulated data and stores an output model that can be used for demosaicing other images.

4.2.1 CNN Model

The CNN model is a simple but effective deep neural network model. The model can be found in the 'model.py' file in the phase2 directory. The model defaults to 6 layers and 64 channels. The model is built using PyTorth libraries. Included in this file is the "createModelFunction" which is based on models developed in this course. This function initializes the model and returns the model. A model is created in the training process, as well as in the inference process.

4.2.2 Training

The training takes place in the "train.py" file in the phase2 directory. The main training used in this project is done through the 'train_model_with_early_stopping() function. This will initialize a model, create a dataset from the CFA images and ground truth images, and then begin training through epochs. The training will prematurely end if a threshold is met. To actually train the model, all the user has to do is call the "train(path)" function from this file. It will handle all the initialization and data set creation, and provide a directory path where two other directories reside. These two directories must be named "ground_truth" and "mosaiced" which contain their respective images.

4.2.3 Demosaicing

Demosaicing using this framework is done in the file "DeepDemosaic.py". This file handles everything related to the inference of the network. To actually demosaic an image using this, call the function "inference(bayer_img,trained_model, dim, stride)". This will return a demosaiced image.

4.2.4 Results

Here the demosaicing results using the Deep Demosaicing method will be discussed and compared with the results for the linear regression model.

4.2.5 Bird Demosaicing

The result of this deep demosaicing is visually very close to the ground truth image. This demosaicing was done on a simulated noisy image that was not in the training dataset with standard deviations of around 1, 2, and 4 for R, G, and B respectively.
The mean square error to ground truth is very low at 9.1594e-4, which is very close to the ground truth image. This is an improvement of a factor of 2 compared to the linear regression model in section 4.1.

(a) Ground Truth Bird

(b) Deep Demosaiced Bird

Figure 9: Deep Demosaic - Birds

4.2.6 Truck Demosaicing

The result of this deep demosaicing is visually very close to the ground truth image. This demosaicing was done on a simulated noisy image that was not in the training dataset with standard deviations of around 1, 2, and 4 for R, G, and B respectively. This image is more complex compared to the one in figure 9 as there are many narrow edges and chrome areas which may lead to aliasing and artifacts as seen in the linear regression section 4.1
The mean square error to ground truth is very low at 4.1865e-3, which is very close to the ground truth image. This is an improvement of 9.44 times compared to the linear regression model in section 4.1.

(a) Ground Truth Truck

(b) Deep Demosaiced Truck

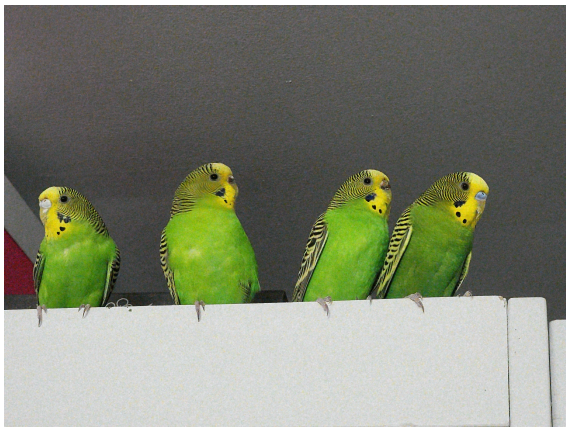
Figure 10: Deep Demosaic - Truck

4.2.7 Extreme Case

To test the power of this learning model, the model will be trained on a small data set of 10 very noisy images. The images have a average standard deviation of 20,40,80 for R,G,B respectively. The model is then tested on a image with the same simulated S.D, but one the model has not trained on.

Results
The results are surprisingly well, with a fantastic visual appearance, and with a mean square error of 4.284e-3 to ground truth.

(a) Extreme Noise



(b) Demosaiced

Figure 11: Deep Demosaic Extreme Noise

4.2.8 Raw Data

The model was trained on a small number of low light raw images taken from the dataset outlined in 3.2. In training, the model is compared to the long exposure image, thus the trained model will not only demosaic, it will also adjust lighting to some degree. The test images below were not included in the training dataset.

(a) Raw Low Light Image

(b) Long Exposure Ground Truth

(c) Long Exposure Ground Truth

Figure 12: Deep Demosaic

The demosaiced image is not of the best quality, but do greatly enhance the brightness of the image.

5 Histogram Equalization

Implemented here is a very simple CLAHE histogram equalization. CLAHE divides up the image into sections and computes a histogram equalization over each block [9]. To reduce an extreme difference in adjacent blocks, an overlap is defined. CLAHE aims to reduce amplification contact by clipping the values based on predefined values in the histogram before computing the CDF. [9].

5.1 CLAHE Implementation

First, the code converts the RGB image into a LAB colour space using OpenCV's 'cv2.cvtColor' function. The result is then split into L, A, and B channels using 'cv2.split'. The tile height and width are then calculated by using the user input tile size. The overlap is then calculated using the user input as well.

```
lab_image = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
l_channel, a_channel, b_channel = cv2.split(lab_image)

tile_height = l_channel.shape[0] // grid_size[0]
tile_width = l_channel.shape[1] // grid_size[1]

overlap_height = int(tile_height * overlap_ratio)
overlap_width = int(tile_width * overlap_ratio)

l_channel_clahe = np.zeros_like(l_channel)
```

The program uses nested loops to iterate over each tile grid while considering the overlap. The regions of the L channel corresponding to the current tile are calculated, then the tile is extracted from the L channel. The histogram is then computed using NumPy 'np.histogram' function.

```
y_start = i * tile_height - min(i, 1) * overlap_height
y_end = (i + 1) * tile_height + min(grid_size[0] - i - 1, 1)
    * overlap_height
x_start = j * tile_width - min(j, 1) * overlap_width
x_end = (j + 1) * tile_width + min(grid_size[1] - j - 1, 1)
    * overlap_width

tile = l_channel[y_start:y_end, x_start:x_end]

hist, _ = np.histogram(tile.flatten(), bins=256, range=[0, 256])
```

The fundamental application of CLAHE is applied by then clipping the histogram to limit the maximum pixel intensity by using the defined helper function [9]. After this, the CDF can be computed using a helper function CDF helper function). Then the program maps the pixel values of the current tile using the computed CDF, equalizing the histogram within the tile and enhancing contrast

```
hist, _ = np.histogram(tile.flatten(), bins=256, range=[0, 256])
hist_clipped = clip_histogram(hist, clip_limit)
cdf = compute_cdf(hist_clipped)
tile_equalized = np.interp(tile.flatten(), np.arange(256), 255
    * cdf).reshape(tile.shape)
l_channel_clahe[y_start:y_end, x_start:x_end] = tile_equalized
```

The image is then stacked and converted back to RGB colour space and returned using OpenCV functions 'merge' and 'cvtColor' respectively.

5.2 CLAHE Results

At user input settings 8x8 tile with 7 overlap and clipped at 2.0. To test this CLAHE implementation, a image intensity is scaled down to only 30% of its intensity. The algorithm is then applied, to which the

results recover the colour image. The results aren't perfect, which a comic-like look to the image, but still the image was mostly recovered.

(a) Simulated Dark Image

(b) CLAHE Recovered Image

Figure 13: CLAHE- Simulated Low Light

To further test this algorithm, it was applied to real world data taken from the data set outlined in section 3.2 [11]. The implemented histogram will be compared to the results of the OpenCV2 implementation for CLAHE. These histogram equalizations were done on RAW images before any processing, each with a clipping limit of 50.0.

(a) Raw Dark Image

(b) CLAHE Recovered Image

(c) OpenCV2 CLAHE Recovered Image

Figure 14: CLAHE- Real Life Low Light

Visually comparing the results of CLAHE Recovered image above, the custom implementation is works very well implemented on the raw data, very comparable to the OpenCV2 CLAHE implementation.

This custom algorithm was then applied to the demosaiced raw data in section 4.2.8. The clipping limit was set to 2.0 for this test case.

(a) Demosaiced Image

(b) CLAHE Recovered Image

6 White Balancing

Gray World white balancing is a fundamental technique in digital imaging aimed at correcting colour casts and ensuring faithful colour reproduction in images [7]. Rooted in the assumption that the average colour in a scene tends towards a neutral gray, this method adjusts the colour channels of an image to achieve a balanced, natural appearance. By computing the average colour across each channel and scaling the colour values accordingly, Gray World white balancing aims to eliminate unwanted colour biases caused by varying lighting conditions [7].

6.0.1 White Balancing Before or After Demosaicing?

White balancing before or after demosaicing is a long debated topic. There are pros and cons to white balancing to each approach. This paper implements white balancing after demosaicing. This was done because white balancing before demosaicing can amplify noise in noisy images. This paper focuses on the application of the image processing pipeline to noisy images, so the main reason white balancing is done post-demosaicing is to avoid further image distortions before demosaicing. Another reason why white balancing is implemented post-demosaicing is so this white balancing algorithm can be generalized and used on any image, not just raw CFA images.
However, there are some potential side effects of white balancing post-demosaicing. Interpolating the raw data before applying white balance adjustments can potentially lead to a loss of detail or introduction of artifacts, especially in areas of high contrast or colour saturation.

6.1 Gray World Implementation

To perform a simple gray world white balancing approach, the program first ensures the image is in a float 32 format with a simple cast. Then the average of each channel is calculated using Numpy mean function. These are then averaged together to get the gray world's definition of "white" [7].

```
# Convert the image to float32 to avoid overflow during calculations
img_float = image.astype(np.float32)

# Calculate the average color of the image across each color channel
avg_r = np.mean(img_float[:,:,0])
avg_g = np.mean(img_float[:,:,1])
avg_b = np.mean(img_float[:,:,2])

# Calculate the average gray value
avg_gray = (avg_r + avg_g + avg_b) / 3.0
```

Then the scaling factor is calculated for each channel. Each scale factor is computed by dividing the average gray value by the corresponding average colour value. These scale factors will be used to balance the colours. The scaling factor is then applied to balance the colours. Then the result is clipped in case if some pixels are scaled beyond 255, and then it ensures the results are in uint_8 format.

```
scale_r = avg_gray / avg_r
scale_g = avg_gray / avg_g
scale_b = avg_gray / avg_b

# Apply the scale factors to balance the colors
img_balanced = img_float * [scale_r, scale_g, scale_b]

# Clip the values to ensure they are within the valid range [0, 255]
img_balanced = np.clip(img_balanced, 0, 255)

# Convert the image back to uint8 format
img_balanced = img_balanced.astype(np.uint8)
```

6.2 Gray World Results

To apply this white balancing algorithm, the image is colour shifted to show the workings of the algorithm. The result is a white balanced image, which colours we expect to see.

(a) Colour Shifted Birds

(b) Birds White Balanced

Figure 16: White Balance Results

Figure 17: Truck White Balance

The results can vary in proper white balancing quality. When a image has little to no white in the scene, the results can have a slight hue shift. This can be seen in figure 17 as there is a very slight blue hue shift. However, figure 16b has a large white contribution, so the white balancing shift is relatively accurate. Some ways to make this white balancing better is to implement human input to declare what section should be white. This would give the program a definition on what should be white and can correct from there [12].

7 Code References

7.1 Mosaic Code

One of the four functions for making a CFA mosaic pattern from a image used for simulation. See Github repo file "sim.py" for the rest of the functions.

```
def bggr_mosaic(rgb_image):
rows, columns, _ = rgb_image.shape
mosaiced_image = np.zeros((rows, columns), dtype=np.uint8)

for col in range(columns):
    for row in range(rows):
        if col % 2 == 0 and row % 2 == 0:
            mosaiced_image[row, col] = rgb_image[row, col, 0]  # Red
        elif col % 2 == 0 and row % 2 == 1:
            mosaiced_image[row, col] = rgb_image[row, col, 1]  # Green
        elif col % 2 == 1 and row % 2 == 0:
            mosaiced_image[row, col] = rgb_image[row, col, 1]  # Green
        elif col % 2 == 1 and row % 2 == 1:
            mosaiced_image[row, col] = rgb_image[row, col, 2]  # Blue

return mosaiced_image
```

7.2 Gaussian Noise Code

```
def add_color_specific_gaussian_noise(mosaiced_image, std_green=1, std_red=2, std_blue=
mosaiced_image_float = np.float32(mosaiced_image)
std_green += random.uniform(-0.1, 0.1)
std_red += random.uniform(-0.1, 0.1)
std_blue += random.uniform(-0.1, 0.1)

noise_green = np.random.normal(0, std_green, mosaiced_image.shape).astype(np.float32)
noise_red = np.random.normal(0, std_red, mosaiced_image.shape).astype(np.float32)
noise_blue = np.random.normal(0, std_blue, mosaiced_image.shape).astype(np.float32)

noised_image = np.zeros_like(mosaiced_image_float)

rows, cols = mosaiced_image.shape
for i in range(rows):
    for j in range(cols):
        if mosaic_pattern == 'rggb':
            if i % 2 == 0 and j % 2 == 0:  # Red
                noised_image[i, j] = mosaiced_image_float[i, j] + noise_red[i, j]
            elif i % 2 == 1 and j % 2 == 1:  # Blue
                noised_image[i, j] = mosaiced_image_float[i, j] + noise_blue[i, j]
            else:  # Green
                noised_image[i, j] = mosaiced_image_float[i, j] + noise_green[i, j]

noised_image_uint8 = np.clip(noised_image, 0, 255).astype('uint8')

return noised_image_uint8
```

7.3 Train Linear Regression V2

```
def learn(input_images, kernels, ground_truths=None, gamma=0.0000003, name=None):
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

input_tensors = [torch.tensor(img, dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(d

ground_truth_tensors = [torch.tensor(gt, dtype=torch.float32).to(device) for gt in grou

kernels_tensors = [torch.tensor(k, dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(d
for j in range(2):
    for i, (input_tensor, kernel_tensor, ground_truth_tensor) in enumerate(zip(input_te
        kernel_tensor.requires_grad = True
        optimizer = optim.SGD([kernel_tensor], lr=gamma)
        for epoch in range(6000):
            optimizer.zero_grad()

            output = F.conv2d(input_tensor, kernel_tensor, padding='same')
            output = output.squeeze(0).squeeze(0)
            gt_for_loss = ground_truth_tensor[:, :,j ]

            loss = F.mse_loss(output, gt_for_loss)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Epoch {epoch}, Loss: {loss.item()}, {j}")

            if (loss.item() < 2):
                break
        # fig, ax = plt.subplots(1, 2, figsize=(12, 6))
        # ax[0].imshow(output.detach().cpu().numpy())
        # ax[0].set_title("Output Image")
        # ax[1].imshow(gt_for_loss.cpu().numpy())
        # ax[1].set_title("Ground Truth Image")
        # plt.show()
    optimized_kernel = kernel_tensor.detach().cpu().numpy().squeeze()
    np.save(f'outputs/matrix/{name[j]}_V2', optimized_kernel)
    print("DONE")
```

References

[1] K. Jeong, S. Kim, and M. G. Kang, "Multispectral demosaicing based on iterative-linear-regression model for estimating pseudo-panchromatic image," Jan 2024.

[2] G. Jeon, S.-J. Park, A. Chehri, J. Cho, and J. Jeong, "Color image restoration technique using gradient edge direction detection," pp. 101–109, 2010.

[3] Y. Liu, P. Liu, Z. Zhuang, E. Chen, and F. hong Yu, "A color image processing pipeline for digital microscope," vol. 8419, 2012.

[4] P. A. Cheremkhin, V. V. Lesnichii, and N. V. Petrov, "Use of spectral characteristics of dslr cameras with bayer filter sensors," *Journal of Physics: Conference Series*, vol. 536, p. 012021, Sep 2014.

[5] R. Lukac and K. Plataniotis, "Color filter arrays: Design and performance analysis," *IEEE Transactions on Consumer Electronics*, vol. 51, p. 1260–1267, Nov 2005.

[6] T. Akazawa, Y. Kinoshita, S. Shiota, and H. Kiya, "Three-color balancing for color constancy correction," *Journal of Imaging*, vol. 7, p. 207, Oct 2021.

[7] E. Lam, "Combining gray world and retinex theory for automatic white balance in digital photography," in *Proceedings of the Ninth International Symposium on Consumer Electronics, 2005. (ISCE 2005).*, pp. 134–139, 2005.

[8] W. A. Mustafa and M. M. M. A. Kader, "A review of histogram equalization techniques in image enhancement application," *Journal of Physics: Conference Series*, vol. 1019, p. 012026, jun 2018.

[9] P. Musa, F. A. Rafi, and M. Lamsani, "A review: Contrast-limited adaptive histogram equalization (clahe) methods to help the application of face recognition," in *2018 Third International Conference on Informatics and Computing (ICIC)*, pp. 1–6, 2018.

[10] G. Jeon and E. Dubois, "Demosaicking of noisy bayer-sampled color images with least-squares luma-chroma demultiplexing and noise level estimation," *IEEE Transactions on Image Processing*, vol. 22, pp. 146–156, 2013.

[11] C. Chen, Q. Chen, J. Xu, and V. Koltun, "Learning to see in the dark," pp. 3291–3300, 06 2018.

[12] I. Boyadzhiev, K. Bala, S. Paris, and F. Durand, "User-guided white balance for mixed lighting conditions," *ACM Transactions on Graphics*, vol. 31, p. 1–10, Nov 2012.