3SK3 Project 2:

Divesh Chudasama, chudasad
chudasad@mcmaster.ca
2021-04-10

1. Introduction:

In this project the basic concept of machine learning is explored using a simple multilinear regression algorithm.The process of training to generate a ‘model’ showcases some of the performance advantages of this approach. Similar to the first project we take a mosaic image that would come from a CFA and demosaic it into a full rgb image. The objective is to have better performance and compare to matlab’s demosaic function.

2. Algorithm:

The general idea is to follow the principle of regression where we use data points to approximate a gradient (linear function) that has the lowest error between each point. This helps prevent overfitting like we saw in the brute force interpolation approach in project 1.

For this design we want to look at each pixel position on a mosaic image and use the neighbouring pixels to generate a multi linear regression for the center pixel. We have 4 sets of groups or patches, with patch A we train the green and blue values missing. With patch B and C we train the red and blue missing values. And with patch D we train the red and green missing pixels. We collect all the same patches and use them to train against a reference image that is a full rgb version of the same image. Here is how we complete the actual calculation of the coefficients, where A is all the columized patches in the image, c are the linear coefficients and G is the green values in the rgb image:

$$c = (A^T A)^{-1} A^T G$$

The patches follow in a certain order across the mosaiced image, here is a layout of it using the A,B,C,D naming for each patch:

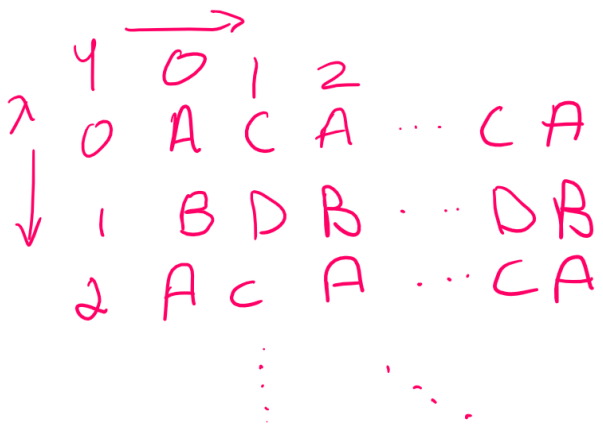

Figure 1: patch order

3. Implementation:

3.1. Patches:

The language of choice for this project was also done in python like in project 1. For the process of training a few functions needed to be created in order to pull all data from the mosaiced image. I created a “patchify” function that takes a padded version of the original image and goes pixel by pixel and extracts each block. The minimum patch size I tested with was a 5x5, but I made this a parameter for the patchify function. I will later discuss in the results section the outcome of varying the patch size. This block/patch is then converted into a column using numpy’s column function and then stored in an array corresponding to that patch. This process requires a large number of iterations in a nested for loop, to optimize this process and reduce the training time. I split the patchify process into 4 set corners of the image. Each corner is patchified and the data is then combined into 1 set of arrays used to complete the regression model. In practice when training the provided tree.png the brute force patchify took almost 3 minutes, with the optimization it was done in 10-20 seconds.

The type of patch is determined by the position that the pixel is in, as shown in figure 1. I check the x and y coordinates (row, column) and take mod 2 of both, if it's 0 and 0 its patch A.

x%2	y%2	patch
0	0	A
1	0	B
0	1	C
1	1	D

Once the patches are stored and the RGB references are also stored, the coefficients can then be calculated using numpy's matrix operations this becomes straightforward. Inorder to reuse them I store the coefficients in a csv file that can then be loaded into the demosaicing file later.

3.2. Demosaicing:

Once we have the multilinear coefficients, in order to demosaic any image we simply loop through in the same way we need when we were patchifying and calculate the respective missing colour pixel. Use the weights/coefficients and the neighbouring pixels (patch) to do this. Another key point that helped clear up some artifacting was defining bounds for the pixel values, there were a fair few that exceeded the 0-255 limit. I clip those to prevent any strange colour artifacting in the final image.

3.3. Project Structure:

The file structure contains data provided (in/out) as well as a utils folder containing helper files and the main directory with the train and demosaic scripts. Within the util folder I have a support file which contains some basic functions for converting an image to a mosaic, padding an image dynamically, clipping output between 0-255, and a mean squared error function. The regression file contains some support function for completing the training. This would include a function that performs the matrix operations to generate the coefficients and the patchify function.

- ❏ data/
  - ❏ in/
    - ❏ bayer/
      - ❏ Mosaic images
    - ❏ Images …
  - ❏ out/
    - ❏ Output image files
- ❏ utils/
  - ❏ support.py
  - ❏ regression.py
- ❏ train.py
- ❏ demosaic.py
- ❏ weights.csv

4. Running Program:

4.1 Training

To train, within the main directory you may run the following. It requires 1 argument:

- input rgb image path (used as reference for training)

```
$ python train.py "<input file path>"
```

4.2 Demosaicing

Within the main directory you can run the following command in the terminal. It takes 2 arguments:

- input rgb image path
- desired output image path

```
$ python demosaic.py "<input file path>" "<output file path>"
```

5. Experiment and Results:

Using the provided test data as a benchmark for training and demosaicing I recorded the error in each image. In addition to the provided images I also used a personal image that was also used in the last project so I can refer back to the improvements in the overall output. Then I increased the patch size to see the impact of the error and finally also tested in matlab.

Figure 2: patch size 5x5

Figure 3: patch size 7x7

Figure 4: patch 9x9

In the images We can see some checker boarding in the 5x5 image, but this tends to reduce as the patch size increases.

5.1. Tests

Image	Error			
	Patch 5x5	Patch 7x7	Patch 9x9	Matlab mosaic()

Lenna.png	235.6706	116.7192	75.4842	19.5114
kodim05.png	69.3950	61.3668	67.7992	60.2215
kodim07.png	38.2685	28.7887	28.3983	22.9461
kodim21.png	44.9917	32.3196	30.4985	34.2974
lion.png	48.5383	31.6986	25.0830	17.3215
tree.png	79.4640	69.3021	66.3137	78.4965
kodim23.png	94.6313	50.5736	38.8237	17.6087
raptors.jpg	57.6384	39.6905	40.1324	(no mosaic to use)

5.2 Comments:

As can be seen, the error across each image reduces as the patch size increases, since we are also increasing the number of coefficients going from 25 to 49 to 81. This however still shows that the matlab does outperform in overall in some cases. These values may also vary depending on which image is used to train with, I have used tree.png in all cases and have noted an improvement in most images if I use the raptors.jpg instead. This can be due there being more information and colour in that image, but it also is a lot larger and takes longer to train. Also it can be noted that in comparison to the previous project the performance has improved in regards to overall accuracy (error reduced) and also the demosaicing runs considerably faster.

5.3 Retrained:

Image	Error
Lenna.png	37.4966
kodim05.png	40.8099
kodim07.png	14.0412
kodim21.png	25.2858
lion.png	18.9420

tree.png	99.2970
kodim23.png	17.0072
raptors.jpg	15.5015

When retrained using the raptors.jpg at a 9x9 patch size we see all the images tested improve in error except for the tree.png. It can also see that it performs better than the matlab function.

Conclusion:

The use of a linear regression algorithm has shown promise and further provided performance improvements over the generic brute force (naive implementation). This opens the door to further train the current model and enhance the accuracy while still keeping the same run time.

References:

Project document:
https://avenue.cllmcmaster.ca/d2l/le/content/376196/viewContent/3098734/View

Project slides:
https://avenue.cllmcmaster.ca/d2l/le/content/376196/viewContent/3096453/View