Machine Learning and Neural Networks for Image Processing

Darren Au

University of Texas at Austin

Faculty Mentors

Dr. Michael Cullinan

Dr. Carolyn Seepersad

Eva Natinsky

Doug Sassaman

Abstract

This investigation aims to develop an algorithm to identify these feature-rich areas in an image. I began by creating and testing several filters and techniques to perform this task. I then selected the best option and used this algorithm to generate a dataset before putting the dataset into a machine learning model. The architecture of this model is based on UNet, a common image segmentation model typically used for medical images. Through training with a batch size of 1 and an epoch of 5, I achieved an accuracy of around 80% while testing the model on a validation set.

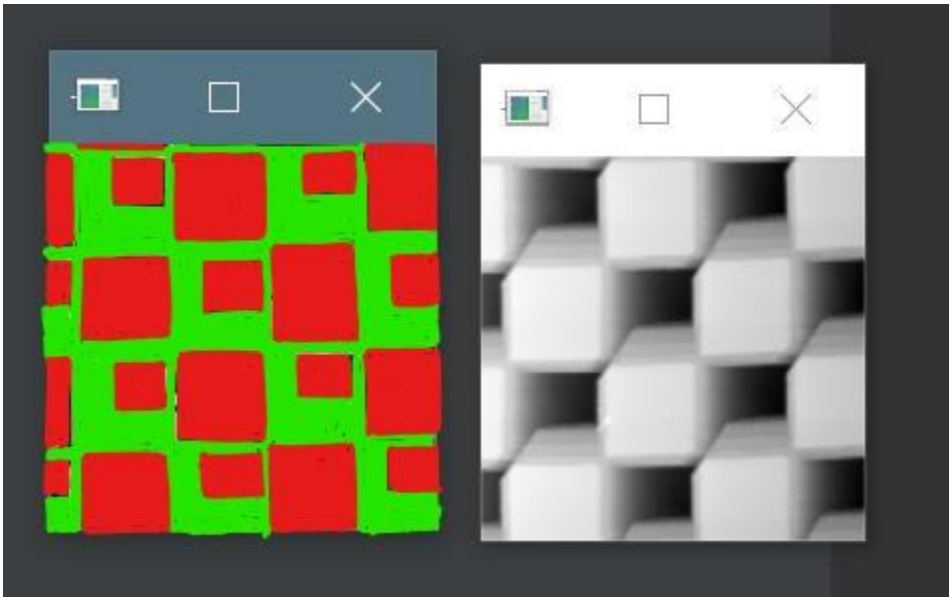
Introduction

Dr. Michael Cullinan's work involves working with nanomanufacturing systems. As part of his research, he needs to take atomic force microscopy (AFM) image scans to analyze his work. However, as this process is time-consuming, he was looking to identify feature-rich areas from a larger image before focusing on those areas at a higher resolution. The purpose of this project would be to accomplish that.

Methods

This project is predicated on the assumption that areas of high gradients are equivalent to feature-rich areas. From a human analysis standpoint, this has primarily appeared to be true. Consequently, most of the feature-rich detection is referred to as "edge detection" as well.

My graduate mentor, Eva Natinsky, began by giving me a template of what she envisioned the algorithm would do.



I began by researching methods to detect areas of high gradient. The most common result was using a “sobel filter”, which is a set of matrices convolved across the image. The matrices used can be seen below.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

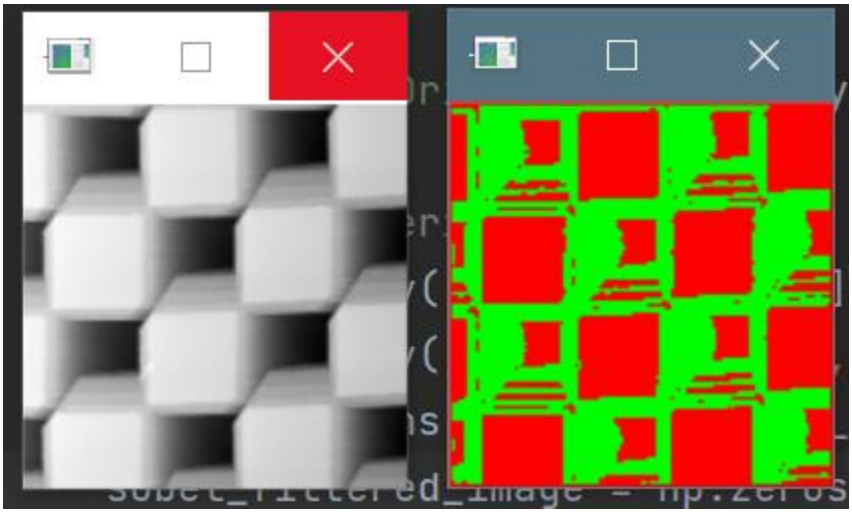
+1	+2	+1
0	0	0
-1	-2	-1

Gy

The two matrices are set up such that when convolved with the image, it will produce an output matrix whose magnitude depends on the gradient of the original image. Both matrices need to be

applied as one determines the gradient in the x-direction while the other does this for the y-direction. The two gradient matrices are then combined to a total magnitude that is determined with the distance formula of sqrt(x^2-y^2). By doing this, the resulting output is an image of the gradient magnitudes of the original image.

This image is then converted into a binary image, where values after a certain threshold are set to green while everything else is set to red.

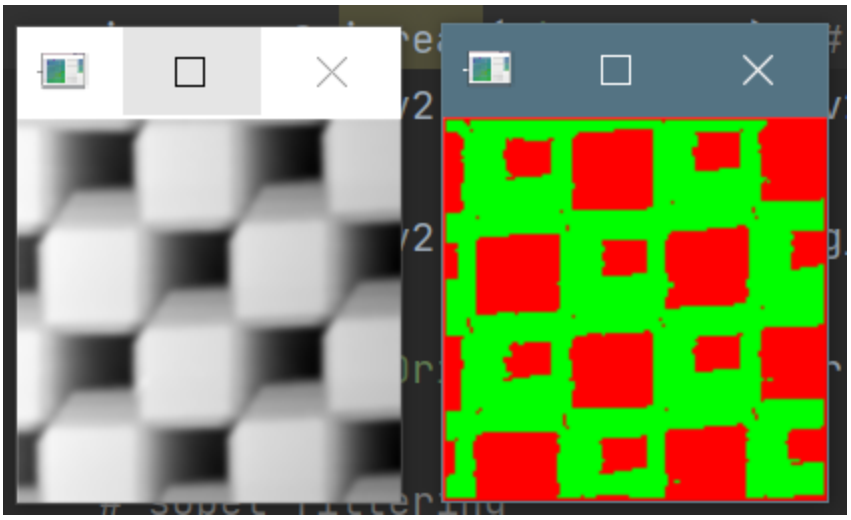


This was the 1st iteration of the algorithm and, coincidentally enough, also the one that ended up being used. There were many attempts afterward to improve the algorithm; Specifically, attempts to emulate the template given by Eva. However, all of the changes had tradeoffs. All these iterations were created primarily using OpenCv and NumPy libraries. OpenCv would allow for most of the image manipulation while NumPy was used to manipulate the matrices. Each of the attempts can be found in this drive folder[1]

[1] https://drive.google.com/drive/folders/1slHtSwUTrUPhkwh4MwEjK-W-jR-Vbian?usp=sharing

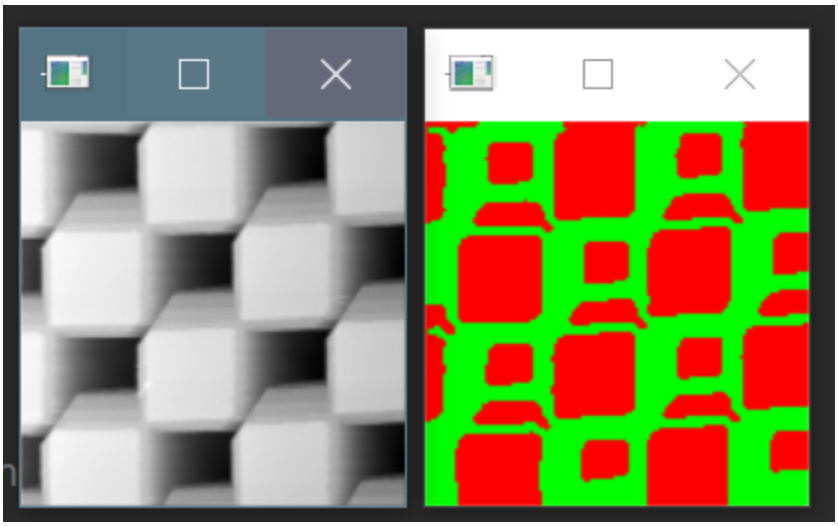
v1.0.1: multiplied gy by 5

This was just toying around with the filter. Evidently, the final result was very close to the "mask" that I was aiming for, but it would only work for this specific picture



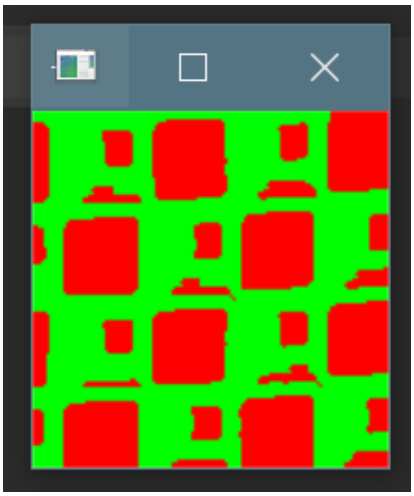
v2.0.0: used contour detection

To reduce the smaller specs of red, I added a contour detection built into OpenCV that would draw a contour around all the red areas. It would then detect the red areas and only draw them if they met a certain threshold. This resulted in less noise but still missed detecting the small trapezoid below the smaller square as an area of high gradient



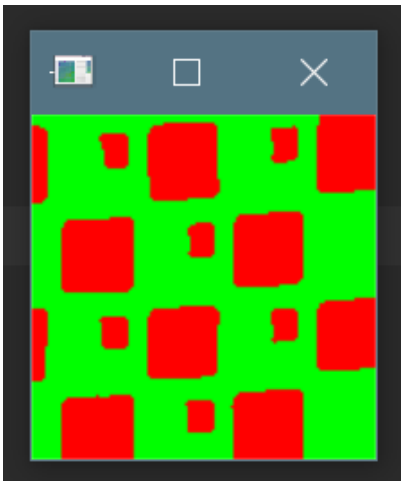
v2.0.1: variable threshold

I attempted to make the threshold vary based on the average pixels so that it wasn't hardcoded and overfitted for that specific image. The results weren't that successful in making the algorithm work for multiple images though



v2.0.2: changed so that value based on median instead of mean

The threshold was now based on the median value of the pixels, so the ratio of red and green pixels should in theory, be constant. Results once again are seen below.



v3.0.0: output to excel file

Nothing in terms of output changed from v2.0.2 but now, the python file writes each of the gradient and actual values of the image to an excel file. I was hoping to use this to help debug and find a new approach to the algorithm

It was also at this point that the deadline for the project was approaching, and my attempts at altering the algorithm weren't seeing much progress. Each change would try to do something new, but it would also lead to issues with other images, sort of like whack-a-mole were trying to address one will lead to something else popping up. Eva and I discussed and decided to move forward with the machine learning aspect of the assignment.

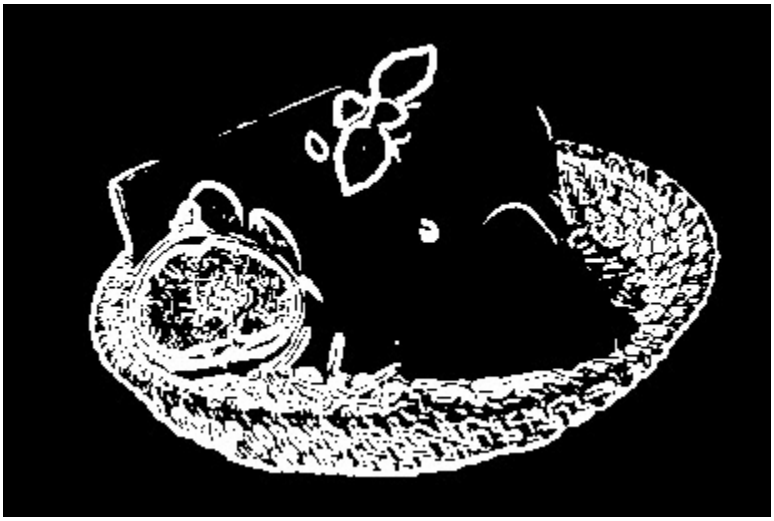
This would involve choosing one of the algorithms to form the dataset. I ended up choosing the 1st iteration, as despite its shortcomings, was the most stable and consistent among all the images.

Moving onto the Machine Learning portion of the project, the first task was to create a dataset. It was relatively easy to convert the previous code to create the dataset; it simply involved parsing a folder of images, running the algorithm, and writing it to a folder. I found a file of a thousand images online and ran that to create the output. The program also had to be altered such that the output mask was now in black and white instead of red and green.

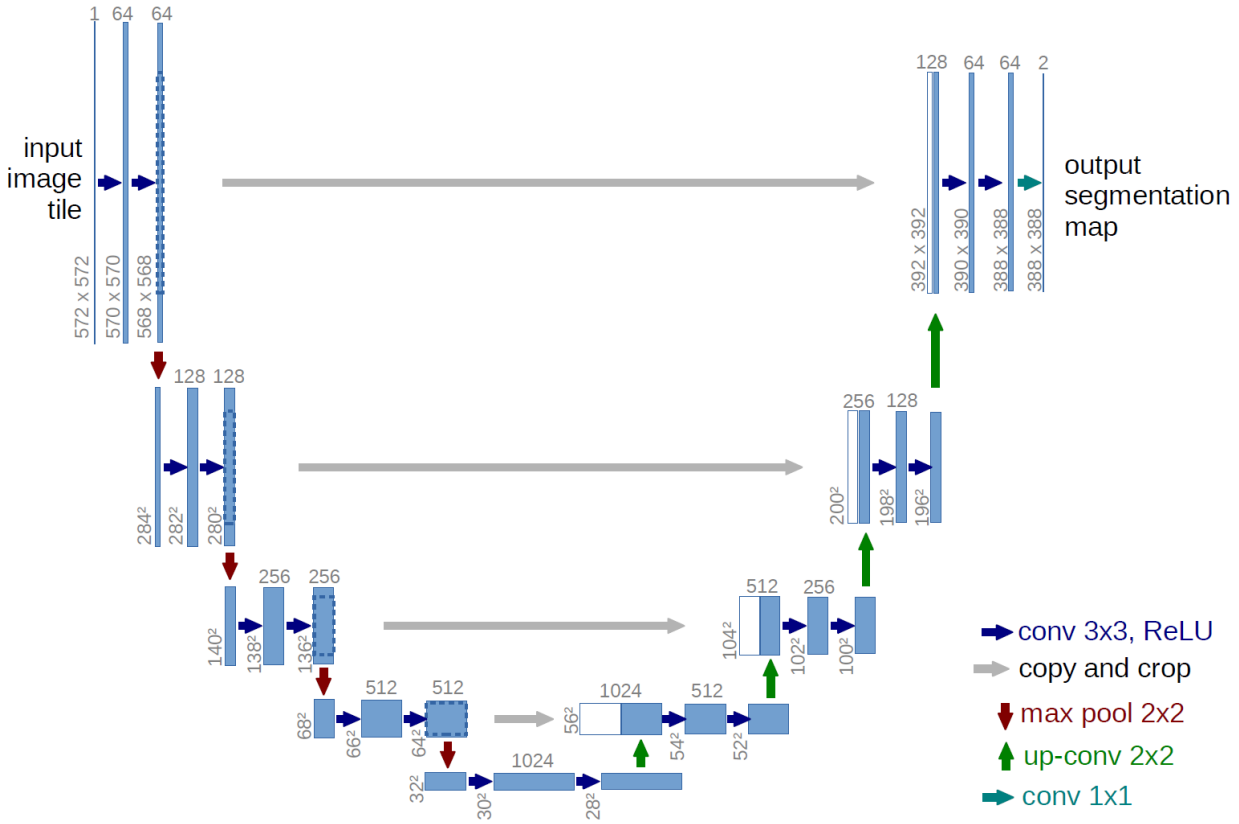
An example of one of the images produced by the algorithm is seen below.

Original image:

Binary Image



The next process was finding a machine learning model to toss the dataset into. After quite a bit of research, I determined that the specific problem of machine learning was image segmentation. The most common architecture used is called UNet. As seen in the image below, it gets its name from its U shape and is typically used on medical images to detect tumors.



However, in our case, it would detect areas of high gradient. It is different than normal architectures because it includes downsampling and upsampling, allowing the model to not only figure out what features are there but also where it is located. Normal models typically only use downsampling is because a very common machine learning problem is image classification, which only requires knowledge of what the image is, and consequently, only downsampling.

The architecture for UNet was found on GitHub, created by aladdinpersson[2]. I chose PyTorch over Tensorflow as Eva and Dr. Cullinan's work is currently in PyTorch, allowing for easier

[2] https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/image_segmentation/semantic_segmentation_unet

integration if it is used. Despite being premade, I still ran into quite a few problems while trying to implement the model.

Problem 1:

The model was training on my CPU as opposed to my GPU. This resulted in far longer training speeds. Resolving this involved a lot of trial and error with installing Nvidia CUDA drivers

Problem 2:

The predictions the model was outputting were largely fully white images. This issue was a result of me switching the binary values. In other words, instead of having a black canvas with white mask values, I had a white canvas with black mask values, causing the model to predict a fully white canvas.

These were the two largest issues besides small debugging to ensure the image path reading and writing were correct.

The model was run with a batch size of 1 and 5 epochs. After each iteration, it saves the model as a .pth.tar file.

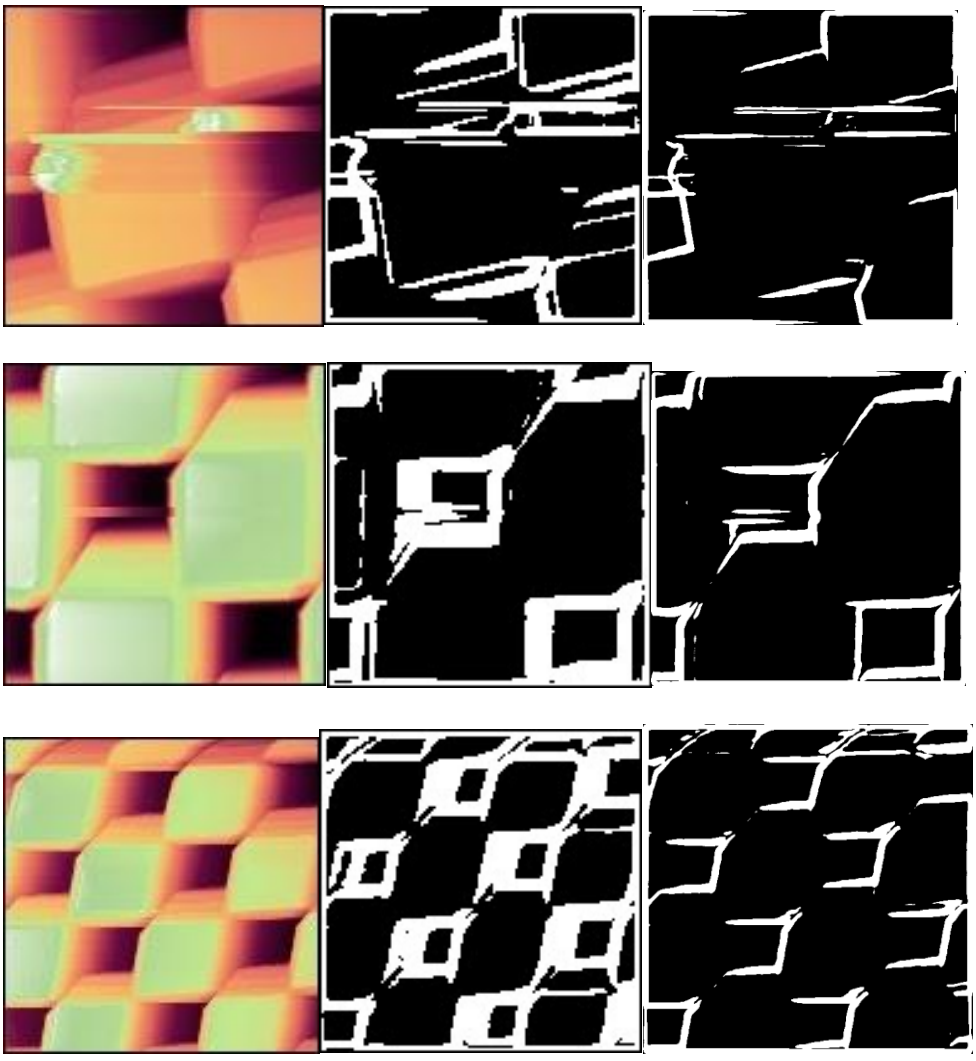
Results

Below are the results from some of the validation sets.

1st column: origins image

2nd column: algorithm-generated mask

3rd column: predicted mask



When the predictions are created, they are also tested for accuracy by determining the percentage of pixels correctly predicted by the model. The average among the validation set was ~80%.

Conclusion

While the model is evidently imperfect, this percentage can be raised by increasing the epochs or increasing the dataset. Despite these imperfections, the purpose of this project was largely to serve as a proof of concept for creating a machine learning model to detect areas of the high gradient; the outcome was pretty successful in accomplishing that. The results are uploaded to GitHub[3]

[3] https://github.com/darrenau03/FIRE-Machine-Learning