DETECTION AND CLASSIFICATION OF CRACKS ON TRANSPORTATION INFRASTRUCTURE USING UAV BASED AERIAL IMAGERY

FINAL REPORT

Team: SDMAY20-13

Client: Muhammad Ahmad Siddique

Advisor: Dr. Halil Ceylan

Team Members:

Lauren Arner – Project Manager

Benjamin Ferreira – Testing Lead

Madi Jacobsen – Data Lead

Ian Seal – Reporting Lead

John Schnoebelen – Software Developer

Jack Temple – Lead Software Developer

Team Email: sdmay20-13@iastate.edu

Team Website: https://Sdmay20-13.sd.ece.iastate.edu

EXECUTIVE SUMMARY

ENGINEERING STANDARDS & DESIGN PRACTICES

Software

- PyTorch – Python library for machine learning
- TensorFlow – Python library for machine learning

Standards

- IEEE 12207 - Software life-cycle processes
- IEEE 29119-2015 - ISO/IEC/IEEE International Standard - Software and systems engineering--Software testing

SUMMARY OF DELIVERABLES

- Software that can process UAV photos and identify cracks
- Graphical user interface for client to easily process photos

APPLICABLE COURSES FROM IOWA STATE UNIVERSITY CURRICULUM

- COM S 309 – Software Development Practices
- COM S 319 – Construction of User Interfaces
- COM S 311 – Introduction to the Design and Analysis of Algorithms
- STAT 330 – Probability and Statistics for Computer Science

NEW SKILLS/KNOWLEDGE ACQUIRED NOT TAUGHT IN COURSES

- Machine learning using PyTorch library
- Image processing using Python
- Differences between cracks and joins in concrete
- Utilizing Iowa State University's High-Performance Computing (HPC)

CONTENTS

FIGURES AND TABLES

DEFINITIONS

GUI – Graphical User Interface

HPC – Iowa State University's High-Performance Computing System

UAV – Unmanned Aerial Aircraft; a drone that can be flown remotely

UI – User Interface

INTRODUCTION

ACKNOWLEDGEMENT

Our team would like to acknowledge and thank our client within Iowa State's Civil, Construction and Environmental Engineering department, Muhammad Ahmad Siddique, for the assistance he has provided throughout the project. From the beginning, Ahmad has been proactive in assisting our team with design specifications, equipment, and initial data. Throughout the project, Ahmad was available to answer questions and was willing to help us in whatever manner he could.

PROBLEM AND PROJECT STATEMENT

PROBLEM STATEMENT

Due to the weather fluctuations in Iowa and throughout the Midwest, concrete and other road surfaces are constantly changing causing potholes, cracks, and other problems that create hazardous and at times, undrivable road conditions. Currently the images or videos collected by UAV must be carefully examined by an operator to manually identify the cracks over long pavements

PROJECT STATEMENT

The purpose of our project is to provide a way to take photographs from an Unmanned Aerial Vehicle, process them, and output meaningful data that can assist users in identifying cracks in pavement and changes in cracks over time.

This has been accomplished by using machine learning algorithms and image processing. To do this, we trained a machine learning model by creating an artificial neural network that can identify whether a given image has a crack in it or not. The model was trained using a large dataset of open-source images of cracked/non-cracked pavement. Photos taken by the UAV, which were provided by our client, are then run through our machine learning model. During image processing, the algorithm detects whether a crack was detected or not. The output is an image where cracks identified will be shown.

In addition, we created a graphical user interface that will allow the client to use the algorithm as a program to select images and review the output data.

INTENDED USERS AND USES

This project is intended to be used by professionals such as construction engineers, researchers, and Department of Transportation employees and officials, as an additional resource to evaluate existing infrastructure including roads and bridges. This will assist in prioritizing repairs and maintenance to roads and bridges in critical conditions in order to ensure they stay safe and drivable.

OPERATIONAL ENVIRONMENT

The operational environment for this project can be looked at in two ways. The first environment would be the operational environment of the product we are creating. This product, a software, will be used in an office like environment where a desktop computer or laptop is available.

The second, and more variable, environment would be the one where the images are captured by the UAV. This environment is considered more important due to the requirements of photos that need to be taken. For best results using the algorithm, the photo captured by the UAV should have no shadows and no precipitation. While the overcast conditions will alter the pictures will still be able to run through image processing whereas precipitation could alter the photos beyond our capabilities.

END PRODUCT AND DELIVERABLES

The end product and deliverable to the client will be a software that can intake a photograph of a road, identify cracks, and output relevant summary data such as percentage of photo identified as a crack and meta data related to the position of the UAV where the photo was taken. This software can be used to help those assessing infrastructure conditions and prioritize roads for maintenance and repair. This software will be broken into two parts: back and front end. The back end will be the algorithm and script that processes the image. The front end will be a GUI for client that allows them to easily select, process, and view the output of an image.

BACK END DELIVERABLES

The back-end deliverable will be algorithm used to train the dataset and created the image crack detection software. This is also tied in with the front-end deliverable, as the front end also facilitates part of the backend as well. There are two parts for the backend deliverable. One part is the training portion that is used to train the algorithm given a dataset of cracked/non-cracked concrete imagery. Whether or not the client will be able to run this successfully will depend on how much processing power is available, as this portion is quite resource intensive. The testing portion of the backend deliverable is what is tied in with the front end. This part is the result of the training portion. It takes an epoch, which depending on how accurate the epoch was trained will detect cracks based off the input images given on the front end.

FRONT END DELIVERABLES

The primary front-end deliverable will be a functional user interface in the form of a desktop application/executable file. We based the functionality of our UI on several functional requirements. First, the UI should let the user select a desired directory of drone imagery that they want to detect cracks on. Next, the UI should be able to send all necessary commands to the back-end crack detection software easily so that it can run those images through the model without the user having to input any complicated shell commands. During execution, the UI should display the progress of the program towards completion, and finally, output all logged data and images to a known location on the user's machine. In addition, the user interface will be easy to use and require no Python or coding knowledge.

PREVIOUS WORK AND LITERATURE

Detecting cracks using machine learning is a well-researched topic. *Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks* [1] goes extensively into the mathematics and logic behind using a convolutional neural network to detect cracks. We used the same dataset that was used in their research to train our project. *Road Crack Detection Using Deep Convolutional Neural Network* [2] is another research paper that we looked at. This research was specifically attempting to detect cracks in roadway, which provided useful insights. There are also many projects in

public repositories on GitHub that have tried to solve this problem as well. We have linked a few in the appendix that we used, referenced, and experimented with. There are two main differences between these projects and ours. The first is that ours is being trained for use on roadways. This means it will be attempting to detect cracks in asphalt as well as concrete. It will also be working with aerial photographs instead of close up pictures of concrete, which is what most of the previous models were created and tested for. The other difference is that our project will also be developed to include a GUI. This will allow users who are not comfortable or accustomed to using the command line to use the project.

PROJECT IMPLEMENTATION BREAKDOWN

TECHNOLOGY CONSIDERATIONS

The current technology used by researchers in identifying cracks in pavement involves printing off all the images and then circling by hand each crack that exists. This is very time consuming and should be automated. There have been attempts to use machine learning to detect cracks in pavement, however none of them are complete solutions. Most are not able to detect the difference between cracks in concrete and joints between different sections, and grass is frequently identified as cracks. Most examples we found also used datasets of around 40,000 images to train the program. In order for a model to be trained to a higher confidence level, there must be large datasets, which is hard to come by without manually creating/gathering the images for the set.

BACK END

DESIGN ANALYSIS

After looking at different options for machine learning algorithms, we originally decided that PyTorch would be a better option than its closest competitor TensorFlow. This is because of PyTorch being easier to use and having more documentation. We have also created a Python script that crops images down so that they can be easily analyzed by the software. During the first semester, we were able to get a working PyTorch project to detect cracks with limited accuracy. However, in January, the project stopped training on the HPC clusters. After extensive troubleshooting we decided to switch to a similar

TensorFlow project so that we could continue to attempt to improve our results. During this transition, we also began training and running the model on less powerful personal computers and discovered that the training time did not increase dramatically. Our results did not change dramatically with the switch; however the model was much more reliable and we ran into fewer issues.

TRAINING THE ALGORITHM

For the algorithm to run, it first must be trained. The model does this by taking the images from the dataset and splitting them. In the dataset of 40,000 images, 30,000 of them are used to "train" the model by telling it whether the image contains a crack or not. After it has been trained, the remaining 10,000 images used in the dataset are used to test the model's accuracy. Once the test images have been run, an epoch is returned. An epoch is the confidence level that the model can train itself to. This is how well the model can correctly choose whether a training image is cracked or not crack. The higher the epoch, the more "confident" the model is at correctly identifying the images as cracked or not.

For the confidence level we were able to get an epoch accuracy of 97% using the PyTorch model. However, during January, the HPC stopped training the PyTorch model. After multiple attempts to try and solve the issues, we decided to try using a similar model using TensorFlow and train it on personal computers. Using this model, we were also able to get approximately the same confidence level.

RUNNING THE ALGORITHM

When running the algorithm, algorithm will choose the model with the highest epoch score. The model works by breaking down each input image into 128 x 128-pixel squares starting at the top left of the image. Each square is run through the model as a separate image and classified as a crack or not. If the image is a crack, the model moves on to the next grid square. If the image has been identified as having no crack, it masks it in black.

Once all the individual grid squares have been processed, they are reassembled to create the input image again but with the black overlay hiding those squares that are identified not having a crack.

FRONT END

DESIGN ANALYSIS

When setting out to design a functional user-interface for our client to run our crack detection software, we realized we needed to find a suitable GUI framework to work with. After spending some time researching different frameworks, we decided upon PyQt5 due to its simplicity and ease of use for Python applications. PyQt5 is a set of Python bindings for Qt5, which is a group of libraries used to create simple/modern user interfaces for programs. QT also has a tool to help format the PyQt5 UI widgets called QT Designer, which we used to produce .ui files – inputted as a sort of "style-sheet" for our code.

To work towards our final product, we designed a GUI prototype with a focus on functionality over appearance. This prototype has two buttons – one to let the user choose a directory of images, and another to run those images through the back-end's crack detection model. The UI shows the file path of the inputted files, as well as of the outputted images so the user can easily confirm where both are stored while using the app.

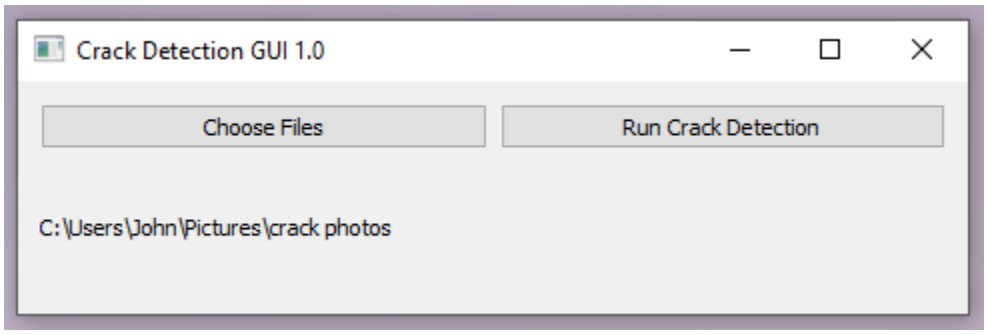

Figure 1. Prototype UI

To get the user interface to communicate with the back-end software, we had our code send OS commands to directly start up the crack detection software on the selected directory of images. We decided on this because after a lot of research we could not find a good way to maintain the command-line argument format when importing the back-end code as a Python module into our front-end code.

Finally, our prototype will be converted into a full desktop application (.exe file) that can be installed and run by the user without installing Python or TensorFlow. This will be the final front-end product and will contain a user manual and installation guide to ensure ease of use.

TESTING & RESULTS

This section will discuss the testing process and results for both front and backend. The back end section details the process used to get the true accuracy of the algorithm along with a breakdown of results. The front end section details the criteria used for designing the user interface provided to the customer.

BACK END

In order to get the true accuracy of the algorithm, the images that were processed with the algorithm were compared to control images where the cracks were human identified. This section breaks down the process we used along with the results.

PROCESS

The process to compare results is straight forward. For every test image used, a control image was created. This image would be divided into 128 x 128-pixel grid squares and each square was identified as either having or not having a crack by hand.

Once an image was processed through the algorithm, it was then compared to the control image by hand to see if the algorithm correctly identified whether or not a crack was present.

To see the full detailed testing process, see Appendix 2. Detailed Testing Process.

RESULTS

The results we had for the true accuracy of the images varied greatly. There were a lot of factors that played into the ability of the algorithm to correctly identify whether a crack was present.

The following table shows the results for several of the processed images.

Image	Road	Angle	True	False	True	False	Total	Total	Negative	Positive

	type		Positive (Pavement Dot)	Positive (Pavement)	Negative (black)	Negative (black dot)	(552)	Accuracy (%)	Accuracy (%)	Accuracy (%)
8	asphalt	askew	117	246	160	29	552	50.18	84.66	32.23
11	asphalt	askew	124	247	172	9	552	53.62	95.03	33.42
7	asphalt	askew	129	417	5	1	552	24.28	83.33	23.63
10	asphalt	askew	169	117	231	35	552	72.46	86.84	59.09
FALL_RUN	asphalt	straight	185	104	184	79	552	66.85	69.96	64.01
9	asphalt	straight	202	115	197	38	552	72.28	83.83	63.72
5	concrete	straight	7	330	123	92	552	23.55	57.21	2.08
3	concrete	straight	48	379	125	0	552	31.34	100.00	11.24
2	concrete	straight	52	369	131	0	552	33.15	100.00	12.35
1	concrete	askew	71	397	77	7	552	26.81	91.67	15.17
6	concrete	straight	85	249	205	13	552	52.54	94.04	25.45
4	concrete	straight	104	182	259	7	552	65.76	97.37	36.36
Askew Average (%)	Straight Average (%)	Asphalt Average (%)	Concrete Average (%)			Overall Accuracy		47.74	86.99	31.56
45.47	46.54	56.61	38.86							

Table 1. Accuracy Data Table

In the table we can see the variance of the accuracy. While the results, vary greatly, it is important to consider some of the factors that may have contributed to the results such as the angle of the photo and type of pavement. Although they cannot explain all the inaccuracies, they can account for some. Below are some of the observations we made that while processing these results:

1. The transition from grass to pavement was considered a crack by the algorithm
2. The algorithm had trouble distinguishing between grooving in concrete and actual cracks. (One possible reason for this is the algorithm is looking at each 128x128 image by itself and not the full image)
3. The transition from pavement to gravel is considered a crack (also counted as a crack in the control images)
4. The rock in gravel and asphalt was usually considered a crack by the algorithm

5. Grass itself, along with automobiles, is also counted as a crack by the program, this can be fixed with more test images and training.

As seen in Table 1, the overall average accuracy rate is 47.74%. The program's overall negative accuracy rate is at 87.00%, which is an excellent number to build from when training the program. The final positive accuracy rate hits low at 31.56% due to the multitude of false positive examples given above, such as grass, automobiles, and concrete grooving. Obtaining a multitude of test images with these false positive examples would have been our next step. Originally, it was thought that the angle the image was taken from would drastically skew the accuracy rate. After processing the images, it is obvious that there is only a small difference in the between straight on and slightly askew images, which are at 49.35% and 45.47% accuracy respectively. When comparing crack detection accuracy on concrete versus asphalt, asphalt has a dramatically higher accuracy rate of 56.61% compared to concrete's 38.86%. This is believed to be due to the grooving in the concrete test images.

Overall, these results were lower than expected especially due to the high epoch rating, approximately 97%, that was used when processing these images. However, this data does give extensive information about what type of photos will work best going forward.

FRONT END

Even though our front-end is a smaller portion of our overall project, we still tested abnormal use cases to provide our client with a polished, functional end product.

PROCESS

The process we used for testing the front-end was manual-based testing, where the graphical screens are manually checked in conformance with our deliverable's requirements for the project.

CHECKLIST

We used the following checklist to create a polished user-interface experience:

- Check all the GUI elements for size, position, width, length, and acceptance of characters or numbers.

- Check execution of the intended functionality of the application using the GUI
- Check for errors when images are incorrectly
- Check for clear demarcation of different sections on screen
- Check font used in an application is readable
- Check the alignment of the text is proper
- Check the positioning of GUI elements for different screen resolution.

RESULTS

The results of the testing were straightforward, as we could visualize any problems that arose when going through the checklist above, such as, alternating different combinations of window size to check for alignment of the text issues. If needed, we would alter the code as needed and then re-test using the same scenario. Although this is a tedious way of testing, it worked well for us because of the few specific tasks that our UI needed to accomplish.

CLOSING MATERIAL

CONCLUSION

In conclusion, the purpose of our project was to create a tool that could intake multiple images of roads, process them, and output results that would help users identify and prioritize infrastructure that needs maintenance and repair.

The first goal of meeting 80% accuracy was not met. The program was trained with images of cracks (positive) and images of unbroken asphalt and concrete (negative). Overall, the final accuracy rate was determined to be 47.74%, which did not meet our initial goal. Some trends were noticed while processing the images that may allow us to better train the program. When an obstacle, such as a car or pole, is tested, it will typically come back as a false positive (preferred over false negative because of how the program is trained). One obstacle we did not foresee being analyzed as a crack was grass. This highly skews the accuracy rates in some images due to the altitude the images were taken. Altitude determines if the image is a majority grass or a majority road in many cases. We expected the skew of an image due to the angle it was taken to

have a dramatic impact, but numbers so far show very little difference when the image is straight on compared to askew. The program tends to do dramatically better when identifying cracks on asphalt roads compared to concrete. This may be due some concrete roads have a grooving throughout them that causes the program to return false positives.

Overall, this program is not ready for use by the clients due to the low accuracy rate, but the UI makes it easily learnable for the client once the program detects cracks, crack type, and pavement type more accurately.

REFERENCES

[1] Cha, Young-Jin & Choi, Wooram & Buyukozturk, Oral. (2017). Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks. Computer-Aided Civil and Infrastructure Engineering. 32. 361-378.10.1111/mice.12263.

[2] L. Zhang, F. Yang, Y. Daniel Zhang and Y. J. Zhu, "Road crack detection using deep convolutional neural network," 2016 IEEE International Conference on Image Processing (ICIP), Phoenix, AZ, 2016, pp. 3708-3712

APPENDICES

Concrete Data Set -

https://data.mendeley.com/datasets/5y9wdsg2zt/2#file-c0d86f9f-852e-4d00-bf45-9a0e24e3b932

GitHub Repositories -

- https://github.com/pytorch/examples/tree/master/mnist
- https://github.com/pytorch/examples/tree/master/imagenet
- https://github.com/warmspringwinds/pytorch-segmentation-detection
- https://github.com/fyangneil/pavement-crack-detection
- https://github.com/satyenrajpal/Concrete-Crack-Detection
- https://github.com/Sarthakdtu/Road-Cracks-Detection-Neural-Network

APPENDIX 1. OPERATION MANUAL

INSTALLATION GUIDE

Since the UI prototype never ended up fully converted into an executable desktop application, the program as is cannot be run without having Python3, PyQt5, and all back-end dependencies installed on the local machine. This guide will help lay out the installation steps so our current product can run on any system.

INSTALLATION STEPS

I. Clone Project Repository onto Local Machine

II. Install Python 3:

1. Download the Latest Python 3 Release from python.org
2. Run the installer by double-clicking on the downloaded file
3. Make sure to check the box that says "Add Python 3.x to PATH"
4. Click "Install Now" and finish the installer

III. Install Required Dependencies: (use "pip install" in a shell window)

- PyTorch
- MatPlotlib
- Numpy
- h5py
- torchvision
- Pyqt5

OPERATION

Operating the application is quite simple if you have successfully installed everything listed above. This section will outline how to start up the application and then how to run our crack detection software on a chosen directory of images.

STARTING THE APPLICATION

1. Navigate to the cloned repository "sdmay20-13" on your local machine
2. Navigate to the folder named "gui" where you will see the app gui.py
3. Start the application by either double-clicking gui.py or by entering "python gui.py" into your command line
4. The program should start up in a few seconds

RUNNING CRACK DETECTION

Once the gui.py program has started up, it will look similar to Figure 1 below with only the two buttons labelled "Choose Files" and "Run Crack Detection".

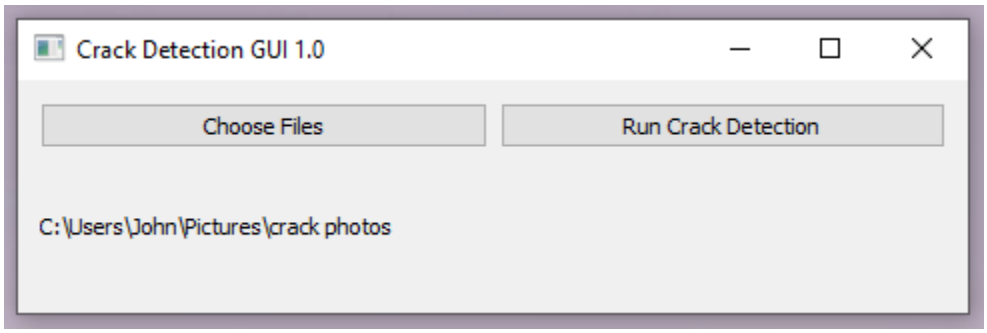

Figure 1. Prototype UI

The steps to run images through our crack detection model are as follows:

1. Click the "Choose Files" button on the left. This will let you browse through your file system.
2. Navigate to the directory that contains your set of desired input images and click "Select Folder".
3. The filepath for your selected directory will be shown under the leftmost button so you can confirm the location before moving on.
4. Click the "Run Crack Detection" button to start inputting the images through the model.
5. Text will appear below the rightmost button after crack detection has completed showing the output directory's filepath.
6. Repeat steps 1 through 5 on additional sets of images if needed.

APPENDIX 2. DETAILED TESTING PROCESS

The purpose of this document is to outline the testing process to verify the true accuracy of the algorithm. The process itself is broken down into two sub-processes: creating control algorithm and comparison of images

CREATING CONTROL IMAGES

In order to compare data, we first needed to create a set of control data for the image we were using to test the algorithm. This was done in a multi-step process.

1. Each image being used had a grid overlay placed on top of it to divide it into 128x128 pixel boxes.

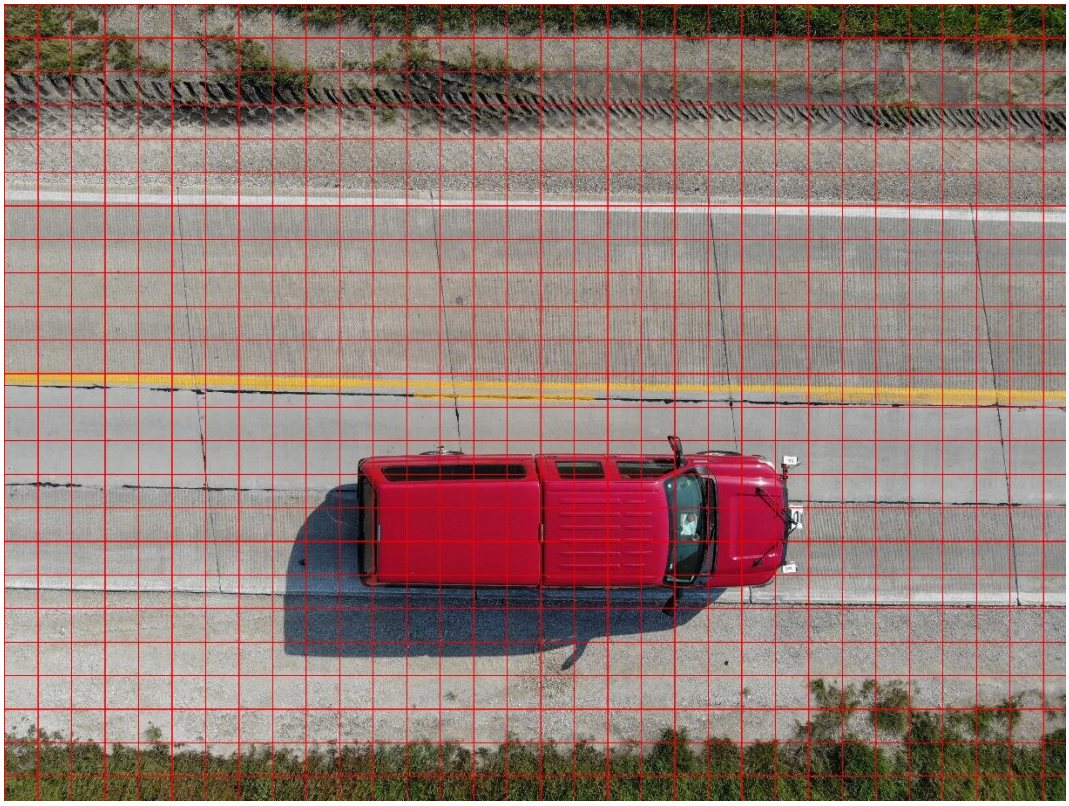

Figure 2. Image with 128x128 grid overlay

2. For each image, every 128x128 pixel square that contained a crack were marked with a pink square. When marking images, the following guidelines were used:

a. Transitions from grass to pavement were not counted as cracks

Figure 3. Example of grass to road (not counted as crack)

b. Grid squares where there is a road edge and a gravel should are counted as crack

Figure 4. Example of road to gravel transition (counted as crack)

3. Once all the cracks were identified for an image, they were rechecked.

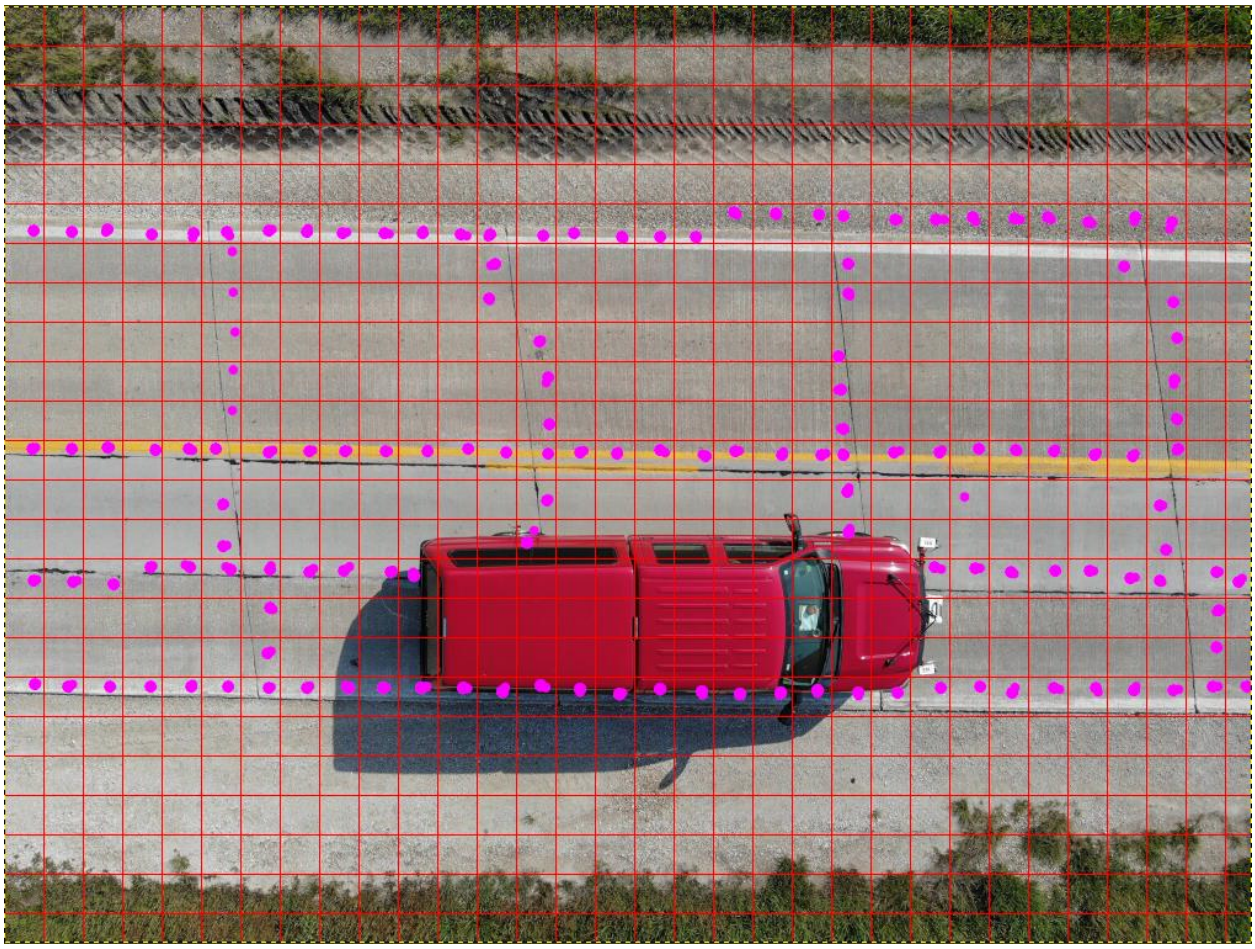

Figure 5. Control Image

COMPARISON OF ALGORITHM RUN IMAGES AND CONTROL IMAGES

Once an image has been run through the algorithm the following steps were used to verify the true accuracy of the algorithm

1. 1. A grid overlay is created from each control image where each grid square is 128x128 pixels and each dot corresponds with a crack located within the

respective grid square.

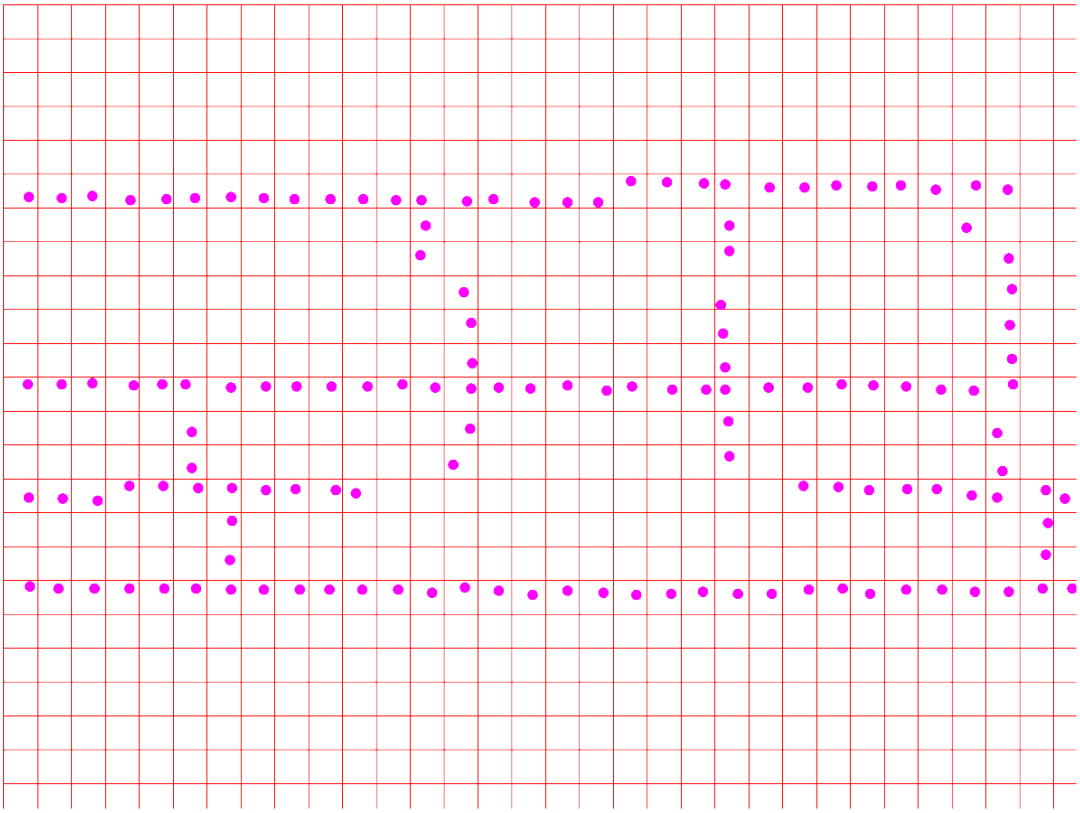

Figure 6. Grid overlay created

2. Once a transparent layer for a photo has been created, the output photo was then loaded into gimp with the layer placed over it.

Figure 7. Grid overlay on top of image processed through algorithm

3. After the two images were combined, each grid square was counted into one of the following categories.**

   True positive – Unmasked grid square with pink dot

   True negative – Black grid square with no dot

   False positive – Unmasked grid square with no pink dot

   False negative – Black grid square with pink dot

   ***Grid squares that only had the overlay and no output (checkered), were not included in the results since no output was produced*

4. To gather the true accuracy of the algorithm for each image, the following equation was used:

$$\text{Image Accuracy} = \frac{\#\text{ True Positives} + \#\text{ True Negatives}}{\#\text{ Total Grid Squares}}$$