Classification of Aerial Imagery using a Relational Convolutional Neural Network

THESIS

Submitted to the Department of Mathematics and Computer Science

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF ARTS

&

BACHELOR OF SCIENCE

Ryan Pacheco
Brendan Peltzer
2019

THESIS

Submitted to the Department of Mathematics and Computer Science

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF ARTS

&

BACHELOR OF SCIENCE

By

Ryan Pacheco

Brendan Peltzer

2019

Classification of Aerial Imagery using a Region Convolutional Neural Network

Author: ______________________________

Ryan Pacheco

Author: ______________________________

Brenden Peltzer

Approved: ______________________________

Dale Hamilton, Department of Mathematics and Computer Science, Faculty Advisor

Approved: ______________________________

Dr. Christian Esh, Department of History, Houghton College, Second Reader

Approved: ______________________________

Barry L. Myers, Ph.D., Chair, Department of Mathematics & Computer Science

Abstract

Classification of Aerial Imagery using a Region Convolutional Neural Network.
PACHECO, RYAN, PELTZER, BRENDAN, MYERS, DR. BARRY, and HAMILTON, DR. DALE (Department of Mathematics and Computer Science).

This project set out to use aerial imagery from Small Unmanned Aircraft Systems (sUAS) to train a Region Convolutional Neural Network (RCNN) to identify and label linear features. For this research, significant amounts of training data were generated using labelImg for rectangular object identification and labelMe for polygonal object detection. This training data was then used to retrain a RCNN to identify and label rail grades, mine tailings, hand stacks, dirt roads, and foundations. Several pre-trained models, including: ssd_mobilenet_v1_coco, faster_rcnn_inception_v2_coco, and rfcn_resnet101_coco were used as a starting point for retraining. Each of these models was designed to allow further retraining of the RCNN, however, each one had roadblocks that prevented successful retraining in this experiment. Several roadblocks were identified that caused valuable time to be wasted. Google Drive proved to be troublesome when attempting to move large amounts of data necessary for retraining. This led to valuable time being spent attempting to send data to and from Google's server that could have been spent further diagnosing retraining errors. To counteract this, an API was developed that would allow for training imagery to be stored easily on the NNU servers rather than Google Drive.

Acknowledgements – Ryan Pacheco

I would like to thank my parents Greg and Cindy, as well as my younger brother Kyle for supporting me during this project and throughout my academic career. Without their support and guidance I would not be in the position I am today. I would also like to thank my professors, Dr. Barry Myers and Dr. Dale Hamilton, who helped me learn what it meant to execute a long term computer science project as well as teaching me the fundamentals of computer science that have helped me be successful in classes and the workplace. I would also like to thank Dr. Christian Esh for everything he did to help me conduct research to see how this project could be aided by a historical perspective. Finally, I would like to thank the Zachary Garner, Nicholas Hamilton, Isaac Kronz, and Johnathan Branham who worked on FireMAP with me and assisted in the beginnings of this research. The groundwork they laid and the assistance they lent during this project was immensely helpful and encouraging.

Acknowledgements – Brendan Peltzer

I would like to thank my parents for supporting me during this project and throughout my academic career. Without their love and support, I would not be where I am today with the opportunities I have been given. I would also like to thank my professors, Dr. Barry Myers and Dr. Dale Hamilton, whose guidance and instruction facilitated my learning during this project as well as helped me develop skills that were pertinent to the completion of a project such as this. Finally, I would like to thank the other research students that worked on FireMAP both this past summer and those who worked in

previous summers. The groundwork they laid and the assistance they lent during this project was immensely helpful and encouraging.

Table of Contents

Abstract iii

Acknowledgements – Ryan Pacheco iv

Acknowledgements – Brendan Peltzer iv

Table of Figures viii

Background 1

  RCNN (CNN w/ SVM) – Brendan Peltzer 1

  Significance of Chinese Miners to the Frontier Story – Ryan Pacheco 5

Methodology - Brendan Peltzer 9

  Training Data 9

  Training Procedure 11

Results/Problems – Ryan Pacheco 11

Possible Solutions – Ryan Pacheco 15

Future Work – Ryan Pacheco & Brendan Peltzer 18

Conclusion – Brendan Peltzer 20

References 22

Apendix A 24

  tfREADME.txt 24

  Jupyter Classification Notebook 29

  Client.py 34

Server.py... 34

Table of Figures

Figure 1 2
Figure 2 3
Figure 3 4
Figure 4 10
Figure 5 14
Figure 6 16
Figure 7 16
Figure 8 16
Figure 9 17
Figure 10 17
Figure 11 17
Figure 12 18

Background

RCNN (CNN w/ SVM) – Brendan Peltzer

The project "Classification of Aerial Imagery Using a Region Convolutional Neural Network" was part of the summer research conducted at Northwest Nazarene University (NNU) as part of the Fire Monitoring and Assessment Platform (FireMAP) team helmed by Dr. Dale Hamilton. The goal of this project was to locate and classify linear features on an image. Small Unmanned Aircraft Systems (sUAS) were used to take aerial photos of the Boise National Forest at a height of 120 meters. Additionally, this project also sought to utilize a Region Convolutional Neural Network (RCNN) to locate and classify rail grades, mine tailings, hand stacks, dirt roads, and foundations.

A Support Vector Machine (SVM) is a classification algorithm used to separate data points into *N* number of features. The SVM uses a technique known as supervised machine learning. Supervised machine learning is an algorithm that uses pre-labeled training data for the training process . These algorithms classify an image by creating a hyperplane which divides the decision space between classes based upon which side of the hyperplane an unclassified object lands when placed in the decision space (Hamilton, 2019). For this project, the algorithm is being retrained to work with two dimensions at a time since each linear feature is being retrained separately. Our project resulted in the hyperplane being split into two halves. When retraining on dirt roads, the SVM will determine on which side of the hyperplane each pixel belongs: the side classified as "dirt road" or the side classified "not dirt road", denoted as a 1 or -1

respectively. A hyperplane separated the two-dimensional data which is shown as a line, as seen in Figure 1.

Misclassification can be reduced by identifying the optimal separating hyperplane-the separating hyperplane furthest from any training vectors-while still correctly separating the training vectors. The margin refers to twice the distance from the hyperplane to the nearest training vector. To achieve the greatest separation between classes and reduced misclassification, the separating hyperplane should contain the maximum margin possible between the two training vectors. The training vectors which lie on the hyperplanes margin edges are the support vectors. Once the optimal hyperplane is located, pixels with unknown class are placed into a vector within the decision space, after which the pixel's class can be determined by calculating which side of the optimal hyperplane the vector lies on (Hamilton, 2018).

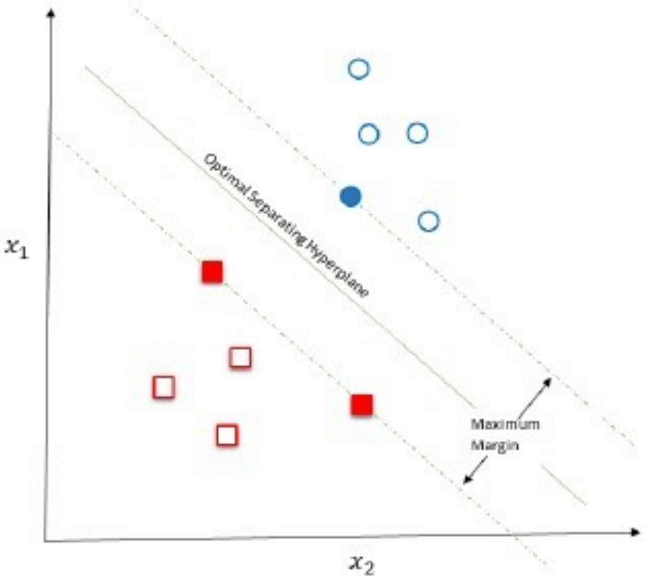

Figure 1: SVM Hyperplane

A Convolutional Neural Network (CNN) is another machine learning algorithm which is very computationally efficient. It is one of the most widely-used models for image-related problems because of how accurate it is. A CNN is made up of many layers that perform a series of convolution and pooling operations as seen in figure 2.

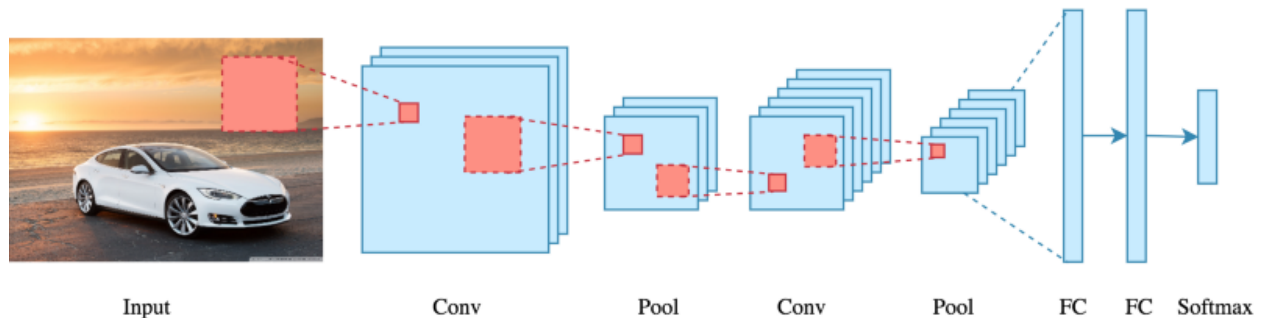

Figure 2: CNN Hidden Layers (Dertat, 2017)

Convolutional neural networks use three basic ideas: Local Receptive Fields, Shared Weights, and pooling (Nielsen, 2015). According to Nielsen, Local Receptive Fields is the input layer. Inputs for the RCNN are divided into groups and condensed into easily processable data. For this project, dividing the input for the RCNN would be taking an image with pixels, and groups those pixels together into discernible features that can be trained. Nielsen explains that for each feature you have, there are Shared Weights associated with it. Every feature has a weight in terms of how important it is to the CNN. In this case, a feature that includes a dirt road would have a greater weight than a feature that is labeled as “not dirt road”, which is also known as the convolution layer. Pooling layers simplify the output from the convolution layer. The pooling layer takes each feature map from the convolutional layer as input and outputs a single, simplified feature map. One form of pooling, known as max pooling, will look at the image and detect a feature based on the data from the convolutional layer but it does not concern

itself with exactly where the feature is in the image, rather it gives you a more broad target area that it can confirm the feature is located within (Nielsen, 2015). For this project, the CNN would be looking at a dirt road, and would surround the road with a box. This box would contain not only a dirt road, but also possibly trees, grass, shrubs etc. due to the fact that max pooling cares about the general area of the feature and not the exact location of the feature, in order to make a decision on the general areas that contain a given feature.

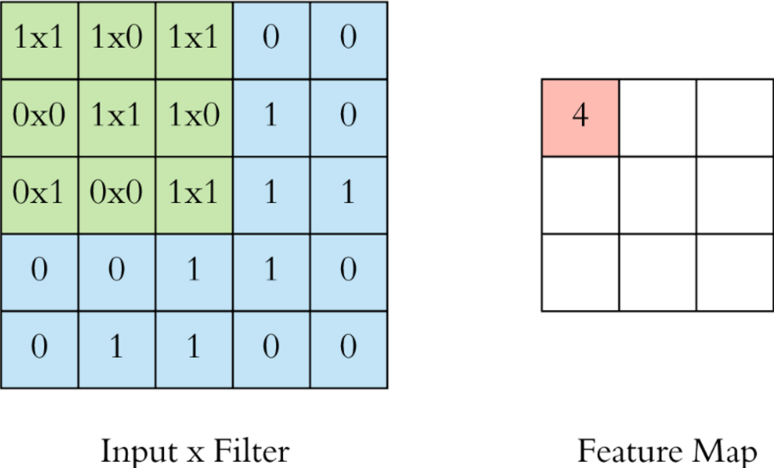

Figure 3: CNN Feature Map (Dertat, 2017)

Pooling is performed after a convolution operation. Pooling is usually used to reduce dimensionality of feature maps, reducing training time and helping to prevent overfitting. Dimensionality is referring to the number of features that are used to train the CNN. By reducing dimensionality, the CNN is able to avoid what is known as the curse of dimensionality. The curse of dimensionality is when too many features are used to train an CNN, which can lead to what is known as overfitting the classifier, and a decrease in accuracy. In the case of the most common type of pooling, max pooling is performed by sliding a window over the input and taking down the max value in that

window. The output of this layer is then used as the input for the next layer, the fully-connected (FC) layer (Dertat, 2017). At any given layer, the input used is the output generated by the previous layer.

A Region Convolutional Neural Network (RCNN) is a type of CNN that acts as a feature extractor which produces a 4096-dimensional feature vector as output (Gandhi, 2018). The extracted features are then fed into an SVM for classification. Essentially, the last layer of the CNN is replaced with an SVM for classification. This allows not just image classification but object detection. It determines if class(es) appear in an image and where they are located on the image. The fully-connected layer of the CNN has an output length that is variable. Taking different regions of interest from the image, a CNN can be used to classify the presence of the object within that region, but this is very slow and would create an immense number of regions. An RCNN can use selective search to extract just 2000 regions from the image instead of countless. “The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an SVM to classify the presence of the object within that candidate region proposal”(Gandhi, 1). For this project’s purposes, the RCNN was chosen over a CNN because it was predicted that an RCNN would both be faster and have higher accuracy than a traditional CNN thanks to the addition of the SVM on the last layer.

Significance of Chinese Miners to the Frontier Story – Ryan Pacheco

Ever since Frederick Jackson Turner presented his groundbreaking essay *The Significance of the Frontier in American History* in 1893, there has been a debate as to how accurate his claims are. Turner and his supporters argue the West was all about

the individual making something of themselves and the frontier represented that spirit. The Chinese miners of Idaho City represent the individuality that Turner is arguing which is demonstrated by Liping Zhu in his book *A Chinaman's Chance: The Chinese on the Rocky Mountain Mining Frontier.* Zhu addresses how Chinese immigrants fit into the frontier story, specifically in the mining town of Idaho City. According to critics of Turner's frontier thesis, the Chinese would have been taken advantage of rather than prospered in the frontier—due to the oppressive and opportunistic mining corporations. Zhu, however, paints a different picture, one that shows the Chinese embracing their new home and succeeding in ways that were unprecedented on the frontier.

In *The Significance of the Frontier in American History*, Turner argues, "the frontier is productive of individualism" (Turner, 1999). Basing his claim on the idea of an individual going into a new land and finding success in a new life, Zhu echoes this sentiment when he describes how the Chinese found massive success in Idaho City's mines. Zhu explains how the Chinese came to Idaho City in the summer of 1863 to lease mining claims from white landowners and how their success impacted the whole town. In the fall of 1865, the Chinese miners were stimulating the local economy in radical ways as they "brought business to local merchants, prospectors, and landlords who desperately needed tenants" (Zhu, 51). Prior to the arrival of the Chinese, Idaho City was in bad shape; a fire brought the town to its knees and nearly killed it. However, the excitement shown by the merchants at the miners' arrival demonstrates how the Chinese were making a new home for themselves in the Boise Basin. Zhu explains how "the merchants wanted the Chinese for business; the prospectors needed the Chinese for mine transactions; the landlords welcomed the Chinese as tenants; the state officers

looked to the Chinese for revenues" (Zhu, 53). The impact the Chinese miners had on Idaho City fits perfectly with Turner's frontier thesis. The Chinese moved into a new land and made something of themselves through their individual merits—shaping the experience of the frontier in Idaho City.

In addition to affirming Turner's belief that the frontier was all about the individual, Zhu provides a much-needed supplement to Turner's work. Turner explains in his book how the frontier moved in a series of the following interconnected phases: the traders' frontier, the ranchers' frontier, and the farmers' frontier. While these phases all describe a part of the frontier story, they neglect one major component of the move west, mining. Turner completely omits the mining that occurred in the frontier, focusing instead on the government's attempts to regulate the frontier. "The East has always feared the result of an unregulated advance of the frontier, and has tried to check and guide it" (Turner, 37). As well as the individuals who made the frontier, Turner examines how a large organizations' involvement in the West ultimately failed as attempts to "deprive the West of its share of political power were all in vain" (Turner, 38). This fits with Turner's claim that the individual made the frontier experience, not a collective group, as illustrated when the individuals of the West "transformed the democracy of Jefferson into the national republicanism of Monroe and the democracy of Andrew Jackson" (Turner, 35), by making America more about the individuals that Andrew Jackson catered to, the early settlers demonstrated just how much power the individualistic West had over the more communal East. However, Turner does fail to take into account the massive success large mining organizations experienced organizing individuals for collective benefit rather than individual gain in the frontier.

Realizing Turner ignored a crucial aspect of the frontier, Zhu examines how the Chinese mining in Idaho City was another phase of Turner's frontier. Zhu explains "if the thrust of U.S. history is a westward movement onto the frontier, as Frederick Jackson Turner alleges, the 5,000-year-old Chinese civilization can be viewed as a process of continuous expansion in all directions" (Zhu, 7). Zhu identifies how the Chinese move to America is just another phase in the frontier's life. Voluntarily moving to the United States, the Chinese miners worked as hard as possible. Leasing mining claims from white settlers, the Chinese saved the Boise Basin from an economic collapse. It is important to note, though, the Chinese miners did not do so as a collective unit, but rather saving the basin was a by-product of individual Chinese miners extracting so much gold from the previously abandoned mines (Zhu, 1997). Eventually growing their numbers to 4,274 (28.5% of Idaho's population), the Chinese miners are the epitome of the self-made individuals Turner claimed the frontier was made of. These Chinese miners are the ones who succeeded in the absence of a large mining corporation, instead of being crushed by one. The Chinese came to the frontier to improve their individual lives—like every other settler—and "the Chinese in [the Boise Basin] mining region were better off in many respects, particularly in terms of living conditions than their counterparts in China—better off, in fact than many whites in the United States" (Zhu, 91). The Chinese miners fit perfectly into Turner's theory the Frontier was made by individuals who were looking to find success and better lives in an unknown land.

Further complementing Zhu's work on the Chinese in the Boise Basin, FireMAP has allowed for machine learning and sUAS technology to gather new perspectives on the lives of the Chinese miners who once mined the hills of Idaho City. Zhu and Turner

argue the frontier was all about the individual and how the individual shaped the frontier they occupied. Using sUAS technology and machine learning, FireMAP will be able to not only observe these claims but document them as well. By using sUAS to get a different perspective on the land in Idaho City, researchers are able to identify mine tailings, hand stacks, linear features, and old Chinese mining camps that were previously only observable from the ground. Applying machine learning to these images will allow researchers to identify features they did not previously know about. Through the use of machine learning and sUAS's, researchers no longer will have to hypothesize about what shaped the frontier, but in the future they will be able to actively observe how the frontier was shaped by the individuals who occupied it, just as Turner and Zhu argue.

Methodology - Brendan Peltzer

Training Data

Training data for the RCNN, was generated using LabelImg (tzutalin). This program is a graphical image annotation tool written in Python (shown in figure 4) that allows a user to label training images, specifying areas of an image to be annotated by creating a bounding box around it. That annotation is then given a label and is saved as an XML file. That XML file must be converted to a CSV file which is then used by Tensorflow, an open source machine learning library for research and production, to generate a TFRecord file. The TFRecord format is a simple format for storing a sequence of binary records. For this project, dirt roads were chosen as the first feature to retrain. The decision to train on dirt roads was made because it was easier for the undergraduate researchers who were recruited to assist in the labeling of training data

to identify what is dirt road or not dirt road than it is to identify other archaeological features.

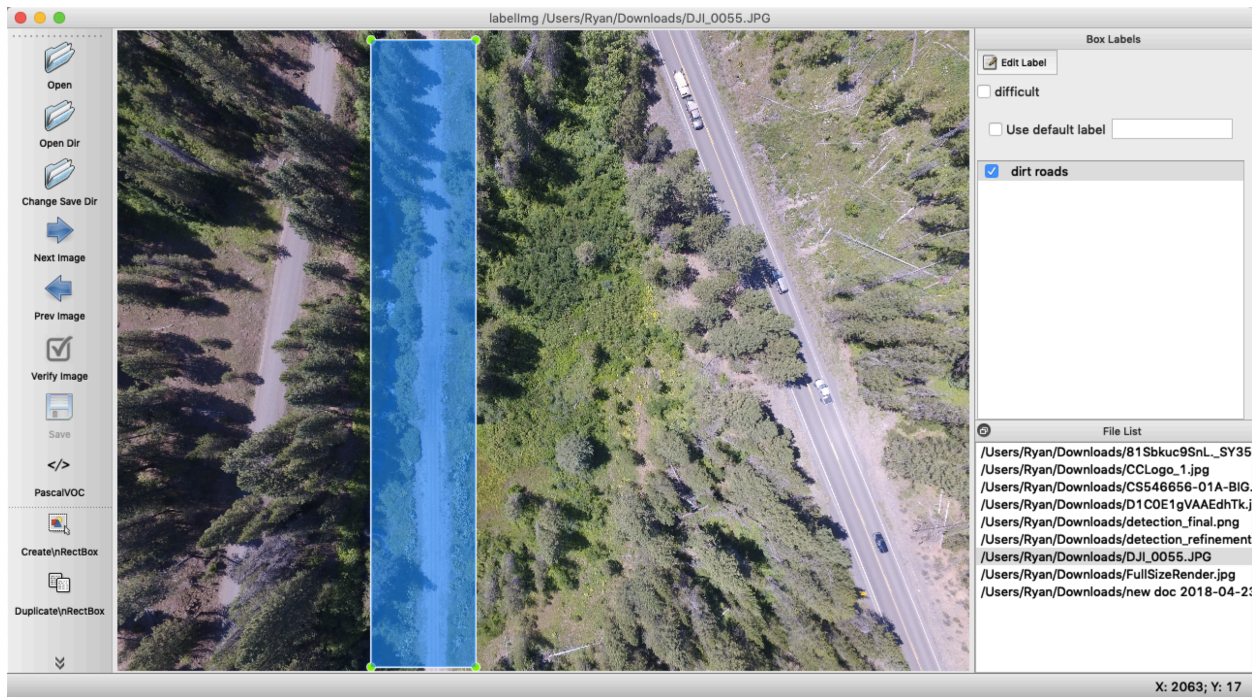

Figure 4: LabelImg Training Data Generator

For this project, several hundred training images were labeled using LabelImg, a tool to classify the different linear features that the algorithms will be retrained for. Three students were drafted to assist in the creation of training data as part of their semester project for a Spatial Analysis (COMP 3230) course at Northwest Nazarene University. These students provided the opportunity to experience project management in a unique way. The Spatial Analysis students had group had a leader who was the main communicator during this project. Instructions were relayed to him on the proper procedure for creating and documenting training data for this project with the direction that the information was to be shared with his other group members. This proved to be a great lesson learned because between the time that the instructions were shared and the time training data creation began, some misunderstandings took place in the group.

This is a great reason why project managers should not rely on people they manage to do their job for them. The proper way to go about this should have been to communicate with each member of the group individually and have them personally state their interpretation of the rules to ensure that the message being sent was the same one being received by the team members. This goal could have been achieved by way of a written document, which could be referred to by the group members in the future in case they had forgotten any details about the procedure. Miscommunications happen, but it is the job of the project manager to ensure that all members understand the process and clear up any misconceptions there may be.

Training Procedure

Before retraining, the labeled training data was compiled and used to teach the RCNN to distinguish between the different linear features. In this instance, each linear feature was trained independently from one another. Meaning that at any given time each training data image has only one label attached to it, which could be dirt roads, rail grades, hand stacks, foundations, or mine tailings for this project. Approximately 10% of the training data is separated from the rest of the group and is used as testing data for classifier validation. While training data is integral to the retraining process, testing data is utilized after retraining is complete, determining how accurately the algorithm classified the linear features.

Results/Problems – Ryan Pacheco

In the end, RCNN was not successfully retrained and was not able to identify dirt roads in sUAS imagery. While the retraining process was underway, several issues with

the current model training pipeline were identified. These problems are: over-reliance on Google Drive, multiple parties developing training data which led to inconstant training data, desire to use the Artificial Intelligence (AI) workstations graphical processing unit (GPU) to attempt to speed up training, and software version requirements at times being outdated. Significant time was invested to correct these issues, taking time away from actually retraining the RCNN, and correcting any problems that occurred during training.

Right from the start, Google Drive proved to be a serious roadblock to getting to the point where retraining the RCNN was possible. This is due to the fact that all the data needed to create training data is only stored on Google Drive. While Google Drive is a convenient way to store data in one central location that is easily accessible to many members of a research team, its limitations quickly became apparent when attempting to retrieve gigabytes of high resolution imagery off the Google Drive server onto a local machine where it can then be used for creating training data, and retraining the RCNN. During the initial days of research, five days were spent attempting to get all the necessary data off Google Drive. This process required attempting to download the images directly from the server and trying to set up Google file stream on a local machine, which would have allowed for Google Drive to act like another local directory on the local machine, allowing for the easy copying of data to and from Google Drive. In the end, a student had to be found who already had Google file stream installed, and the images had to be copied from their “G:” directory onto an external storage device. Due to the data not being stored also on any local passport drive, Google Drive proved

to be a major roadblock in the process as it made getting the necessary data for retraining much more difficult to obtain.

Another issue encountered during research was inconsistent training data being developed. This was due to the use of students from the class “Introduction to Spatial Analysis.” It was very difficult to meet with this group due to schedule conflicts, resulting in only one member of the group attending meetings. This member would then go and pass the information along to his other group members. Miscommunications occurred as a result, including how to name training data once it was generated, which led to several pieces of training data being useless due to not being able to identify which flight the data was from. Another issue with using this group was due to the timeline of their project for spatial analysis the quality needed for good training data had not yet been established and passed onto the spatial analysis students before their work on the training data had completed. The inconsistent training data resulting from this collaboration is one possible cause for the RCNN failing to retrain properly.

Retraining the RCNN required certain software to be installed in the machine used for retraining, specifically TensorFlow. TensorFlow is necessary for retraining the RCNN, but many different versions of TensorFlow exist. At the time of retraining TensorFlow version, 1.12.0 was the latest version. This version of the software was required to retrain the RCNN using the latest features of TensorFlow. However, despite using the latest version of TensorFlow, the official RCNN code provided with the pertained models for processing new images contained an error that identified TensorFlow version 1.12.0 as an earlier version of TensorFlow incompatible with the classification process of the model as shown in figure 5.

```
if tf.__version__ < '1.4.0':
  raise ImportError('Please upgrade your tensorflow installation to v1.4.* or later!')
```

Figure 5: TensorFlow Classification Script Version Restriction Error

Figure 5 shows how the classification code was looking for a version of TensorFlow greater than version 1.4.0. This code was not designed to handle a version of TensorFlow greater than 1.9.0, as the Import Error in figure 5 would be displayed when running TensorFlow 1.12.0. Due to this error, TensorFlow was uninstalled after the install retraining of the RCNN and reinstalled with version 1.9.0. This meant that different versions of TensorFlow were being used for training and classifying. This use of different versions of TensorFlow could be one of the causes for the RCNN retraining failure. Eventually, the error condition was removed from the classification code so TensorFlow version 1.12.0 could be used for classification, however, it is unclear if removing the code in figure X led to an error in classifying the images or if the retraining process was broken elsewhere. Figuring out which version of TensorFlow is necessary for the entire process of retraining is critical to the future success of this research.

The final major issue encountered while retraining the RCNN was the reliance of the GPU version of TensorFlow. This was chosen due to its ability to speed up the retraining process from a week on the CPU version, to 24 - 48 hours on the GPU version. However, the GPU version of TensorFlow was much more complicated to install than the CPU version. In order to utilize the GPU Nividia graphics drivers known as CUDA and cuDNN needed to be installed on the AI workstation. The documentation provided made it unclear which version of CUDA and cuDNN were required, leading to the improper drivers being installed, and the AI workstation having to be rebooted multiple times as the correct version was installed. Also, the latest version of

TensorFlow-GPU required the latest versions of CUDA and cuDNN to be installed. This was not specified in the documentation at the time but has since been updated. In all it took roughly two weeks to get the AI workstation configured properly to utilize TensorFlow-GPU, and during the setup new graphics drivers had to be installed on the AI workstation. The setup of the AI workstation was estimated to be no more than two days, so the two weeks spent on getting the driver situation worked out put the research significantly behind schedule and contributed to less time being available to further investigate why retraining was not successful.

Possible Solutions – Ryan Pacheco

Throughout this research project ideas as to how to counteract the problems encountered were executed. The one possible solution that was developed was the first steps of an Application Programing Interface (API) that would allow for Google Drive to be replaced as the primary storage location of FireMAP's data. The API currently allows for a client machine to send an image to a server, where a copy of that image is then stored for future reference. The API is written using the Python programing language due to its flexibility to function on a machine independent of the host OS. This allows for the client script to be ran on OSX, Windows, or Linux without the need to edit the core code. Using a library known as Flask, the API is able to send and receive curl requests that contain a JavaScript Object Notation (JSON) object of an image converted into 64-bit bytecode. As seen in figure 6, the client side of the application is what converts the image into bytecode.

```
with open(image_file, "rb") as image_file:
    encoded_image = base64.b64encode(image_file.read())
```

Figure 6: Client.py

Figure 7 then shows how the bytecode generated in Figure 6 is then converted to a string format and stored in a JSON object.

```
string_image = encoded_image.decode('utf-8')
image_dict = {}
image_dict["image"] = string_image
json_image = json.dumps(image_dict)
```

Figure 7: Client.py

The bytecode is stored in a JSON object because JSON is a format that can be posted to a server using an HTTP Post command. After the JSON object is created figure 8 shows how an HTTP request is made to the server and the servers response code is returned to the client to let the user know that the transfer was successful.

```
headers = {'Content-Type' : 'application/json'}

response = requests.post(url, data=json_image, headers=headers)
print(response)
```

Figure 8: Client.py

A header is made for the HTTP request that lets the server know to look for a JSON object, the JSON object containing the bytecode of the image is bundled up as the requests data object, and the server's url is then provided in the format of "http://0.0.0.0:5001/save_img" which is specified earlier in the client script and stored as the "url" variable.

The API also consists of a server component that establishes endpoints, and brings the server online for the client script to interface with. Figure 9 shows how the server is brought online.

```
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5001)
```

Figure 9: Server.py

The server is brought up on any available port on the machine where the server is running, also the server is brought up to use the local IP address of the machine where it is hosted. Figure 10 shows how the endpoint that saves the image is defined.

```
@app.route('/save_img', methods=["POST"])
```

Figure 10: Server.py

Figure 10 shows how an endpoint known as "/save_img" is defined and can be pinged by going to "http://0.0.0.0:5001/save_img." Next, the JSON object needs to be retrieved and decoded by the server, shown in Figure 11.

```
def save_img():
    json_data = request.get_json()
```

Figure 11: Server.py

Once the JSON data is retrieved from the HTTP Post, Figure 12 shows how a new image is then generated on the server, and the bytecode stored in the JSON object is used to recreate the original image on the server.

```
new_image = open("test_image.png", "wb")
image_data = base64.b64decode(json_data["image"])
new_image.write(image_data)
new_image.close()
return("200")
```

Figure 12: Server.py

Once the bytecode is decoded on the server a 200 code is sent back to the client to indicate a successful transfer of the data.

The API currently only supports sending one image at a time to a server, and does not currently retrieve images once they are stored on the server. Given these limitations the API currently operates as a starting point that allows for more endpoints to be added to provide increased functionality. Once feature complete, this API should allow for FireMAP to easily store image data on the Computer Science servers, eliminating the need to use Google Drive as the primary storage location of all data. This allows Google Drive to serve as an important backup if the Computer Science servers ever go down and the API with it.

Future Work – Ryan Pacheco & Brendan Peltzer

While the goal of this project was not met, a significant amount of progress has been made and a multitude of lessons were learned throughout the duration. This project has great potential and can be continued by future students in more ways than one. One such opportunity is to make a more complete version of the API. This new version of the API would need to have several additional features than those available in the current version of the API. The API would need to be able to transfer more than one image at a time, and even a whole directory, from the client to a server. The API would need to be able to transfer the entire contents of a directory from a client to a server. Also, it is imperative that the API allows for the retrieval of data from the server to the client. Finally, the API would need to be hosted on the NNU Computer Science servers so it can be accessed as a static IP address which can be hit from any machine.

As it stands now, access to the Computer Science servers is blocked on NNU's main network, so permissions must first be set in place for a student to be able to successfully push an image to the server. Once a connection is established, this API could serve as the main storage method for all training data. This would be a far more reliable and faster way of storing images than Google File Stream. However, it is important to note this is not to diminish the usefulness of Google File Stream. Despite the new API, Google File Stream could still be used as a backup and has the advantage of being accessible anywhere not just on NNU's campus. One possibility could also be to set up a Google File Stream folder on the Computer Science servers that the API could place images into. This folder could then sync with Google Drive automatically, allowing for data located on the server to always be accessible from Google Drive should the API go down.

Another project that can be adapted, would be to figure out exactly was is preventing the RCNN from retraining properly. There is a high likelihood that a cause preventing the RCNN from retraining is purely hardware based. As of the time of writing this thesis, the AI workstation that was used to perform retraining has continued to deteriorate in functionality. Currently, no work can be completed without the Operating System (OS) crashing and causing the entire machine to restart. It is possible that during the configuration stage of this machine, some mistakes were made that caused incompatibilities with drivers or other software that prevents all the tools from being utilized properly. There are many conditions that could cause the OS to crash unexpectedly and cause the machine to not function as expected in regards to retraining and general usability. To continue with this research, the next step would be

to either re-image the workstation and install all needed software from scratch or to attempt retraining on a separate machine.

Conclusion – Brendan Peltzer

Overall, this project was a difficult one with a lot of setbacks, and a lot was learned that can be applied in the future. While we did not get the results we had hoped for, significant progress was made. We were able to get the RCNN running successfully but it was not able to retrain in a way where we got the outputs that we were looking for. There were many hours spent troubleshooting the software and dealing with hardware issues on our workstation. We got the code to a point where it should be able to successfully classify roads but we are not getting the outputs we are expecting, namely the bounding boxes are missing. It could be an issue with the configuration of hardware and its drivers. . It is also possible that we simply overlooked something in the code where a fresh set of eyes could come in and fix it right away. Ultimately, we were not able to complete this project due to the time constraints. In the future someone else will be able to pick this project up and get it working provided enough time and effort. Once the RCNN can successfully classify dirt roads, it can easily be tweaked to classify any and all of the other linear features we mentioned in our proposal. Rapid turnaround can be achieved due to the fact that once a model is successfully retrained, it can be retrained using the same methodology, but with different training data.

Several hundred training images were labeled during this project. Someone who picks this up as a project in the future will have a great head-start with all the resources we have compiled. In addition to using LabelImg, LabelMe was used to create training images as well(Wkentaro). The difference between the two is LabelMe is for rectangles,

but with LabelImg you can use points to create any polygonal shape you want. The application of this is for a Mask-RCNN which is another machine learning classifier we wanted to attempt to use for retraining before we ran into the problems with this one. This is another project that a student could pick up in the future, and if they did they would already have a great head state just based on the number of training images we labeled alone.

Google Drive and Google File Stream both caused headaches and wasted time during this project. We want to be efficient as possible, so struggling for hours with Drive and having it fail multiple times while trying to upload and download data is not ideal. If someone wanted to spend more time finishing the API we started, they would have a very viable alternative to Google File Stream, which could still be used as a backup in case all other storage options are damaged or inaccessible.

Overall, we are disappointed with the outcome of this project but we are optimistic about what this project could provide in the future. Anywhere from one to three senior projects could potentially be done with the work that has been started here. While we ran out of time before we could find success, we believe that a talented developer could take this project and turn it into a home run with enough effort and dedication.

References

Dertat, A., & Dertat, A. (2017, November 08). Applied Deep Learning – Part 4: Convolutional Neural Networks. Retrieved from https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Gandhi, R., & Gandhi, R. (2018, June 07). Support Vector Machine – Introduction to Machine Learning Algorithms. Retrieved from https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

Gandhi, R., & Gandhi, R. (2018, July 09). R-CNN, Fast R-CNN, Faster R-CNN, YOLO – Object Detection Algorithms. Retrieved from https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

Hamilton, D; Pacheco, R; Myers, B; Peltzer, B, (2019) “kNN vs. SVM: a Comparison of Algorithms”, *Fire Continuum Conference: Preparing for the Future of Wildland Fire;* May 21-24, 2018; Missoula, MT Proceedings RMRS-P. Fort Collins, CO: U.S. Department of Agriculture, Forest Service, Rocky Mountain Research Station. Online.

Turner, F. J., White, R., Riley, G., Worster, D., Limerick, P. N., Malone, M. P., . . . Etulain, R. W. (1999). *Does the Frontier Experience Make America Exceptional?* Boston: Bedford/St. Martins.

Tzutalin. (n.d.). LabelImg is a graphical image annotation tool and label object bounding boxes in images. Retrieved from https://github.com/tzutalin/labelImg

Wkentaro. (n.d.). Image Polygonal Annotation with Python. Retrieved from https://github.com/wkentaro/labelme

Zhu, L. (1997). A Chinaman's Chande: The Chinese on the Rocky Mountain Mining Frontier. Colorado: University Press of Colorado.

Apendix A

tfREADME.txt

```
Get object detection API to work in windows 10- Tensorflow
*** NOTE: every time you change a environment var, make sure to exit command prompt and open new one ***
*** '>' denotes command line

**** TO DO: in models\research\ can run >python setup.py and should make it so
        you do not have to move folders into models\research\ ****


Download python 3.6.5 64 bit
        select add to PATH
        Install Now

install tensorflow
        > pip install tensorflow
                        or
        > pip install tensorflow-gpu

install dependencies
        > pip install numpy
        > pip install Cython
        > pip install pillow
        > pip install lxml
        > pip install jupyter
        > pip install matplotlib
        > pip install pandas

clone object detection api repositor
        > cd "to wherever"
        > git clone https://github.com/tensorflow/models.git

add PYTHONPATH to environment variables
        open Environment Variables
        under User Variables select New...
        type PYTHONPATH for variable name
        type path to models\research\ for variable value
        type path to models\research\slim for variable value
        select Ok
        add %PYTHONPATH% to PATH

go to where you cloned models and create objectDetect folder
        ex:
```

Documents
-> models
-> objectDetect

install protoc
go to https://github.com/google/protobuf/releases
Download protoc-3.4.0-win32.zip (the newer versions don't work in windows)
unzip
copy protoc.exe in protoc-3.4.0-win32 -> bin
open \models\research\
paste protoc.exe
> cd to 'wherever'\models\research\
> protoc object_detection/protos/*.proto --python_out=.

*** if installed tensorflow-gpu then do 'install cuda' and 'install cuDNN' else skip ***
install cuda
go to https://developer.nvidia.com/cuda-toolkit-archive
select the latest version of CUDA
download AND install Base Installer
THEN download and install Patch's in sequential order
add C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\LATEST_VERSION\bin
and C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\LATEST_VERSION\lib\x64 to PATH
Wouldn't hurt to restart computer

install cuDNN
go to https://developer.nvidia.com/cudnn
click Download cuDNN (may have to setup account)
select I agree to terms
select archive
select Download the latest version of cuDNN for CUDA LATEST_VERSION -> cuDNN LATEST_VERSION Library for Windows 10
from nvidia:
The following steps describe how to build a cuDNN dependent program. In the following sections:
your CUDA directory path is referred to as C:\Program Files\NVIDIA GPU
Computing Toolkit\CUDA\LATEST_VERSION
your cuDNN directory path is referred to as <installpath>
1. Navigate to your <installpath> directory containing cuDNN.
2. Unzip the cuDNN package.
cudnn-LATEST_VERSION-windows7-x64-v7.zip
or
cudnn-LATEST_VERSION-windows10-x64-v7.zip
3. Copy the following files into the CUDA Toolkit directory.

a) Copy <installpath>\cuda\bin\cudnn64_7.dll
to
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\LATEST_VERSION\bin.
b) Copy <installpath>\cuda\ include\cudnn.h
to
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\LATEST_VERSION\include.
c) Copy <installpath>\cuda\lib\x64\cudnn.lib
to
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\LATEST_VERSION\lib\x64.

test if install worked
in models\research\
> python object_detection/builders/model_builder_test.py
**** If you are using tensorflow-gpu and get this error:
ImportError: DLL load failed: A dynamic link library (DLL) initialization routine failed.
Then go back a version in tensorflow by using:
>pip install --upgrade --ignore-installed tensorflow-gpu==1.5
****

create images
Download prebuilt binaries of labelImg from https://github.com/tzutalin/labelImg
This will create bounding boxes of images in pascal voc format
In objectDetect folder (created above), create directory 'images'
Put images to classify in images directory
When using labelImg save xml (with coordinates of bounding boxes) in images directory
Within images directory create train and test folder:

```
objectDetect
    ->images
        ->train
        ->test
```

COPY about 5-10% of images along with matching xml annotations into test and COPY the rest into train

change xml to tfrecords
Create 'data' directory in objectDetect

```
objectDetect
    ->images
    ->data
```

go to https://github.com/datitran/raccoon_dataset
Copy 'xml_to_csv.py' and 'generate_tfrecord.py' and put them in the objectDetect folder
In 'xml_to_csv.py' change main function to:

```
def main():
```

```
                for directory in ['train', 'test']:
                        image_path = os.path.join(os.getcwd(),
'images/{}'.format(directory))
                        xml_df = xml_to_csv(image_path)
                        xml_df.to_csv('data/{}_labels.csv'.format(directory), index=None)
                        print('Successfully converted xml to csv.')
```

> cd 'wherever'\objectDetect
> python xml_to_csv.py

In 'generate_tfrecord.py':

On line #29, change row_label == 'your label'
If multiple records, make elif row_label == 'your 2nd label' return 2
Return 0 is a placeholder, don't use it
Also, the command line command is in the comments at the top,

> cd 'wherever'\objectDetect
> python generate_tfrecord.py --csv_input=data/train_labels.csv --output_path=data/train.record
> python generate_tfrecord.py --csv_input=data/test_labels.csv --output_path=data/test.record

get model and config file

pre-trained models can be found at:

https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

their corresponding config file can be found:

'wherever'\models\research\samples\configs

put the model and config file into 'wherever'\objectDetect
use 7-zip TWICE to extract model from .tar and .gz
In config file:

Under model funct change
    num_classes: "your num of classes"
Under train_config change
    fine_tune_checkpoint: "modelfolder/model.ckpt"
Under train_input_reader
    input_path: "data/train.record"
    label_map_path: "data/object-detection.pbtxt"
Under eval_input_reader
    input_path: "data/test.record"
    label_map_path: "data/object-detection.pbtxt"

Move config file into training directory

objectDetect
    ->training
        ->"whatever config file you picked"

In data folder create 'object-detection.pbtxt"

objectDetect
    ->data

->object-detection.pbtxt
Inside object-detection.pbtxt
```
item {
    id: 1
    name: "your label here"
}
```

move folders to api folder
from "wherever"\objectDetect
Copy directories:
data
"whatever model you picked"
images
training
Paste these into "wherever"\models\research\object_detction

start training
> cd 'wherever'\models\research\object_detction
> python train.py --logtostderr --train_dir=training\ --pipeline_config_path=training\"whatever config file you picked".config
"python train.py --logtostderr --train_dir=training\ --pipeline_config_path=training\faster_rcnn_inception_v2_coco.config"
*** if you get error:
ValueError: Tried to convert 't' to a tensor and failed.
Error: Argument must be a dense tensor: range(0, 3) - got shape [3], but wanted []
go into "wherever"\models\research\object_detection\utils\learning_schedules.py
and change:
```
rate_index = tf.reduce_max(tf.where(tf.greater_equal(global_step, boundaries),
                                     range(num_boundaries),
                                     [0] * num_boundaries))
```
into
```
rate_index = tf.reduce_max(tf.where(tf.greater_equal(global_step, boundaries),
                                    list(range(num_boundaries)),
                                     [0] * num_boundaries))
```

to export inference graph:
run from "wherever"\models\research\object_detection
> python export_inference_graph.py --input_type image_tensor --pipeline_config_path training/faster_rcnn_inception_v2_coco.config --trained_checkpoint_prefix training/model.ckpt-2150 --output_directory smoke_inference_graph

to run ai detection:
run from "wherever"\models\research\object_detection

> jupyter notebook

navigate to demo or sample something, change the sample input to your own input and run

Jupyter Classification Notebook

```
# coding: utf-8

# # Object Detection Demo
# Welcome to the object detection inference walkthrough!  This notebook will walk you step by step through the process of using a pre-trained model to detect objects in an image. Make sure to follow the [installation instructions](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/installation.md) before you start.

# # Imports

# In[34]:


import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile
import cv2

from collections import defaultdict
from io import StringIO
from matplotlib import pyplot as plt
from PIL import Image

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")
from object_detection.utils import ops as utils_ops

if tf.__version__ < '1.4.0':
  raise ImportError('Please upgrade your tensorflow installation to v1.4.* or later!')


# ## Env setup

# In[35]:
```

```
# This is needed to display the images.
get_ipython().run_line_magic('matplotlib', 'inline')


# ## Object detection imports
# Here are the imports from the object detection module.

# In[36]:


from utils import label_map_util

from utils import visualization_utils as vis_util


# # Model preparation

# ## Variables
#
# Any model exported using the `export_inference_graph.py` tool can be loaded here simply by changing `PATH_TO_CKPT` to point to a new .pb file.
#
# By default we use an "SSD with Mobilenet" model here. See the [detection model zoo](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md) for a list of other models that can be run out-of-the-box with varying speeds and accuracies.

# In[37]:


# What model to download.
MODEL_NAME = 'smoke_inference_graph_new'

# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'

# List of the strings that is used to add correct label for each box.
PATH_TO_LABELS = os.path.join('training', 'object-detection.pbtxt')

NUM_CLASSES = 1


# ## Download Model
```

```
# ## Load a (frozen) Tensorflow model into memory.

# In[38]:


detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
  with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')


# ## Loading label map
# Label maps map indices to category names, so that when our convolution network predicts `5`, we know that this corresponds to `airplane`.  Here we use internal utility functions, but anything that returns a dictionary mapping integers to appropriate string labels would be fine

# In[39]:


label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(label_map,
max_num_classes=NUM_CLASSES, use_display_name=True)
category_index = label_map_util.create_category_index(categories)


# ## Helper code

# In[40]:


def load_image_into_numpy_array(image):
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape(
      (im_height, im_width, 3)).astype(np.uint8)


# # Detection

# In[41]:


# For the sake of simplicity we will use only 2 images:
# image1.jpg
```

```
# image2.jpg
# If you want to test the code with your images, just add path to the images to the
TEST_IMAGE_PATHS.
PATH_TO_TEST_IMAGES_DIR = 'test_images'
TEST_IMAGE_PATHS = [ os.path.join(PATH_TO_TEST_IMAGES_DIR, '_
({}).jpg'.format(i)) for i in range(1, 29) ]

# Size, in inches, of the output images.
IMAGE_SIZE = (12, 8)


# In[42]:


def run_inference_for_single_image(image, graph):
  with graph.as_default():
    with tf.Session() as sess:
      # Get handles to input and output tensors
      ops = tf.get_default_graph().get_operations()
      all_tensor_names = {output.name for op in ops for output in op.outputs}
      tensor_dict = {}
      for key in [
          'num_detections', 'detection_boxes', 'detection_scores',
          'detection_classes', 'detection_masks'
      ]:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
          tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(
              tensor_name)
      if 'detection_masks' in tensor_dict:
        # The following processing is only for single image
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
        # Reframe is required to translate mask from box coordinates to image coordinates and fit
the image size.
        real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [real_num_detection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [real_num_detection, -1, -1])
        detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
            detection_masks, detection_boxes, image.shape[0], image.shape[1])
        detection_masks_reframed = tf.cast(
            tf.greater(detection_masks_reframed, 0.5), tf.uint8)
        # Follow the convention by adding back the batch dimension
        tensor_dict['detection_masks'] = tf.expand_dims(
            detection_masks_reframed, 0)
      image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')
```

```
    # Run inference
    output_dict = sess.run(tensor_dict,
                    feed_dict={image_tensor: np.expand_dims(image, 0)})

    # all outputs are float32 numpy arrays, so convert types as appropriate
    output_dict['num_detections'] = int(output_dict['num_detections'][0])
    output_dict['detection_classes'] = output_dict[
      'detection_classes'][0].astype(np.uint8)
    output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
    output_dict['detection_scores'] = output_dict['detection_scores'][0]
    if 'detection_masks' in output_dict:
     output_dict['detection_masks'] = output_dict['detection_masks'][0]
  return output_dict


# In[43]:


counter = 0
for image_path in TEST_IMAGE_PATHS:
  counter = counter + 1
  image = Image.open(image_path)
  # the array based representation of the image will be used later in order to prepare the
  # result image with boxes and labels on it.
  image_np = load_image_into_numpy_array(image)
  # Expand dimensions since the model expects images to have shape: [1, None, None, 3]
  image_np_expanded = np.expand_dims(image_np, axis=0)
  # Actual detection.
  output_dict = run_inference_for_single_image(image_np, detection_graph)
  # Visualization of the results of a detection.
  vis_util.visualize_boxes_and_labels_on_image_array(
    image_np,
    output_dict['detection_boxes'],
    output_dict['detection_classes'],
    output_dict['detection_scores'],
    category_index,
    instance_masks=output_dict.get('detection_masks'),
    use_normalized_coordinates=True,
    line_thickness=17)
  image_np = cv2.resize(image_np, (4000, 3000))
  image_np = cv2.cvtColor(image_np, cv2.COLOR_BGR2RGB)
  cv2.imwrite(

'C://Users/BlakeJohanson/Documents/FireMAP/CNNdata/smoke/smoke_{}.png'.format(counter)
,
```

```
        image_np)
```

Client.py

```
import base64
import json
import requests
import sys

image_file = sys.argv[1]
print(image_file)
url = "http://0.0.0.0:5001/save_img"

with open(image_file, "rb") as image_file:
    encoded_image = base64.b64encode(image_file.read())

string_image = encoded_image.decode('utf-8')
image_dict = {}
image_dict["image"] = string_image
json_image = json.dumps(image_dict)
headers = {'Content-Type' : 'application/json'}

response = requests.post(url, data=json_image, headers=headers)
print(response)
```

Server.py

```
from flask import Flask, request
import json
import base64

app = Flask(__name__)


@app.route('/save_img', methods=["POST"])
def save_img():
    json_data = request.get_json()
    new_image = open("test_image.png", "wb")
    image_data = base64.b64decode(json_data["image"])
    new_image.write(image_data)
    new_image.close()
    return("200")

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5001)
```