Lunar Surface Image Segmentation: Individual Report

George Washington University
Machine Learning II - DATS 6203_10
Group 5
Joshua Ting
GitHub Repo
Final Presentation

Table of Contents

1. Introduction

In an entirely fictional scenario, our team has been tasked by a space agency to assist in landing their lunar lander on the surface of the moon. Specifically, the agency requires assistance with developing software to automatically identify safe locations on the lunar surface free of large rocks which could be potential collison obstacles.

We are to perform this through analyzing lunar surface images, specifically, to perform semantic segmentation on images of lunar surfaces to identify large rocks that could cause catastrophic collisions during the landing of a lunar lander.

In this paper, we explore several methodologies to perform the task of image semantic segmentation. Specifically, we attempt to accurately predict the pixel classes of lunar surface images. Additionally in this paper, we outline the methodologies for:

- Data Acquisition and Storage
- Image Processing and Augmentations
- Exploratory Data Analysis
- Model Selection, Training, and Metrics Evaluation

2. Background

2.1 Semantic Segmentation

Semantic segmentation is a type of computer vision problem where each pixel of an image is assigned a class which represents parts of some object within that scene. The entire scene can have multiple different objects and therefore multiple different classes.

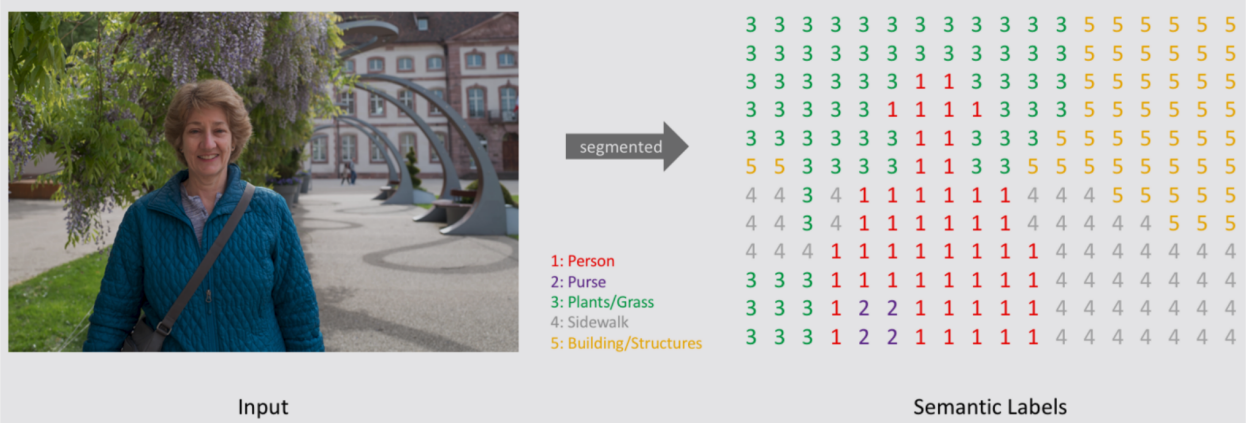

Figure X: Overview of Semantic Segmentation[5]

2.2 Data Description

Our dataset is sourced from [Kaggle's Artificial Lunar Landscape][7], originally posted by Romain Pessie about 3 years ago. The dataset consists of the following:

1. Rendered Lunar Images:
    a. 9,766 Images of Rendered Lunar Landscapes
    b. 9,766 Ground Truth Masks
2. Real Lunar Images for Testing:
    a. 36 Images of Real Lunar Images
    b. 36 Ground Truth Masks

Since real images of the moon are scarce, our space agency has provided photo realistic artificial renderings of lunar images as well as a much smaller subset of real lunar images, primarily to be used for testing purposes. The intent is to train on the rendered images and see how it performs on the real lunar images.

A ground truth mask has been generated for each of the rendered and real images. This process was likely done manually and was also provided with the dataset. These ground truth images will be our target labels. The ground truth masks are colored with the following color-code where each color represents a different class to be predicted.

Ground Truth Mask Color Code

Color	Scene Representation	RGB Value
Red	Sky	255,0,0
Green	Small Rocks	0,255,0
Blue	Large Rocks	0,0,255
Black	Unlabeled/Surface of Moon	0,0,0

Figure X: Breakdown of Ground Truth Masks' Color Coding

2.3 Framework and Network Selection

2.4 Contribution Goals

3. Experimental Setup - Data Preprocessing

3.1 Data Acquisition

To download the data from Kaggle, our team developed an object class to interface with Kaggle's API.

```
class KaggleAPI:
   '''
   Object to handle connection to Kaggle API to upload and download files.
   '''
```

The object has a method to specify a Kaggle dataset and will download to a specified location and unzips. The use of this method requires the creation of a Kaggle API account and saving the credentials in json format to a specified location. Please see our GitHub documentation for more information on this Kaggle credential authentication process.

The following shows the scripts used for data download from Kaggle.

Python File	Description
`KaggleAPI.py`	`Object to handle connection to Kaggle API to upload and download files.`
`kaggle_download.py`	`Script to download Kaggle datasets`

Figure X: Scripts used for data acquisition.

3.2 Splitting Data

Our team split the dataset into the following splits:

Dataset Split	Type	Percentage of Type
Train	Rendered	49%
Validation	Rendered	21%
Test - Rendered	Rendered	30%
Test - Real	Real	100%

Figure X: Splitting dataset into training, validation, and testing.

We decided to have 2 sets of testing datasets; one for the rendered image type which is the same type of data as the training and validation dataset while the other is of real lunar images

taken on the moon. This second testing set will help us determine how generalizable our model could be going from training on rendered data to being deployed in production for moon missions.

The following shows the scripts used for splitting data into training, validation, and 2 testing datasets - one for rendered images and one for real images..

Python File	Description
`TrainTestSplit.py`	`Functions to correctly organize dataset.`
`kaggle_download.py`	`Script to download Kaggle datasets`

Figure X: Scripts used for splitting data.

3.3 Data Augmentation

In order to prevent overfitting, our team also implemented data augmentation techniques on the training dataset.

```
def data_augmentation(self, image, mask):
    '''
    Function to perform data augmentation
    '''
```

When performing data augmentation, we have to pay careful attention that we do not interfere with the pixel location in the image that is associated with the pixel location in the ground truth mask. Meaning, the same pixel classification in the image and in the mask will still need to be the same relative location after data augmentations are applied. The following data augmentations were applied randomly:

Technique	**Applied On**	**Random Chance**
Horizontal Flip	Images and Masks	20%
Vertical Flip	Images and Masks	20%
Brightness -Jitter	Images Only	50%
Contrast - Jitter	Images Only	50%
Saturation - Jitter	Images Only	50%
Hue - Jitter	Images Only	50%

Figure X: Random data augmentation techniques applied.

The following images show an example of the data augmentation performed for the image and associated mask. Notice how the relative pixel locations do not change in the image and mask after augmentation.

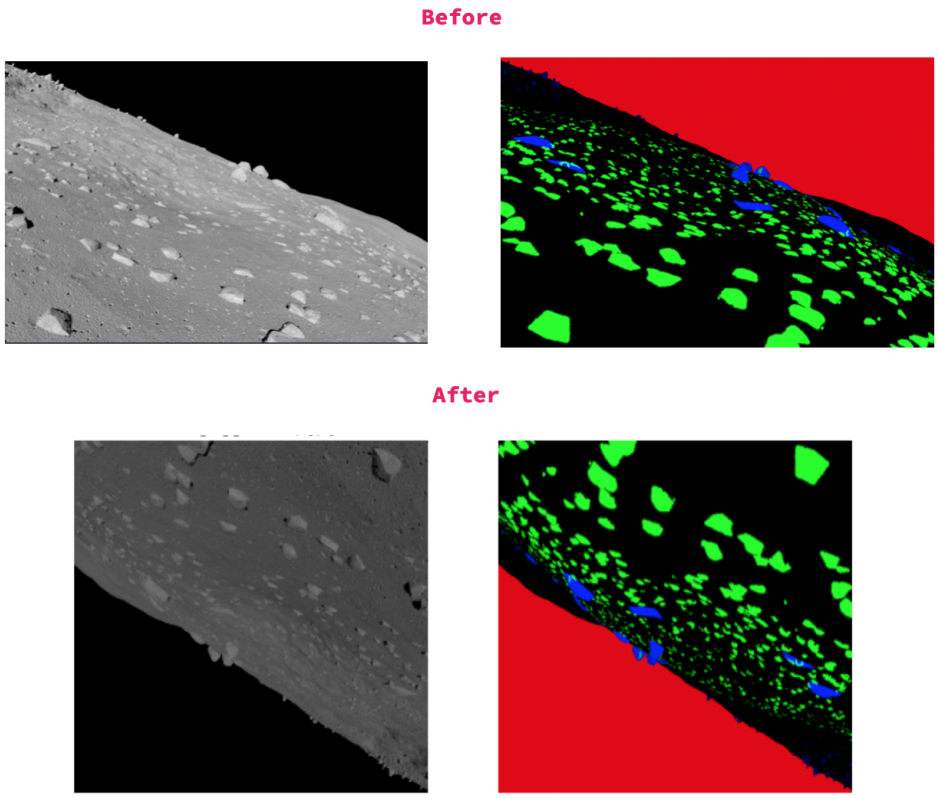

Figure X: Example data augmentation before and after.

The following shows the scripts used for performing data augmentation.

Python File	Description
`ImageProcessor.py`	`Object to handle all processing of images/data.`

Figure X: Scripts used for data augmentation.

3.4 Image Processing

Image processing is performed with the help of the *ImageProcessor()* object class which has methods for performing certain image processing techniques before input into the model for training.

```
class ImageProcessor:
    '''
    Object to handle processing of images.
```

```
    '''
```

The following shows the main processing methods used for the images and/or the ground truth masks.

Step	Method	Description
1	`cv2.resize(img_loaded, (self.imsize, self.imsize))`	`Resize images and masks to 256x256.`
2	`image = self.one_hot_encode(image, class_map)`	`One Hot Encode ground truth masks.`
3	`final_img = self.rescale(image)`	`Rescales the pixels of images and masks to be between 0 and 1.`
4	`img_tensor = torch.from_numpy(img_loaded).float()`	`Convert numpy arrays to torch tensors.`
5	`mask_tensor = mask_tensor.permute(2, 0, 1)`	`Reorders the ordering of the dimensions to have channels first.`
6	`reverse_one_hot_encode(self, img, class_map=None)`	`Function to reverse one hot encode 4 class channel to 3 channel RGB mask.`

Figure X: Image processing steps.

A key image processing technique is to turn the ground truth images from RGB channels to one hot encoded channel, meaning each channel will now represent one class with a pixel value of 1 if that pixel represents that class and a value of 0 if it does not. Our masks have 4 classes so they will be turned from 3 RGB channels into 4 class channels before being fed into the model for training. The diagram below demonstrates an example of this process.

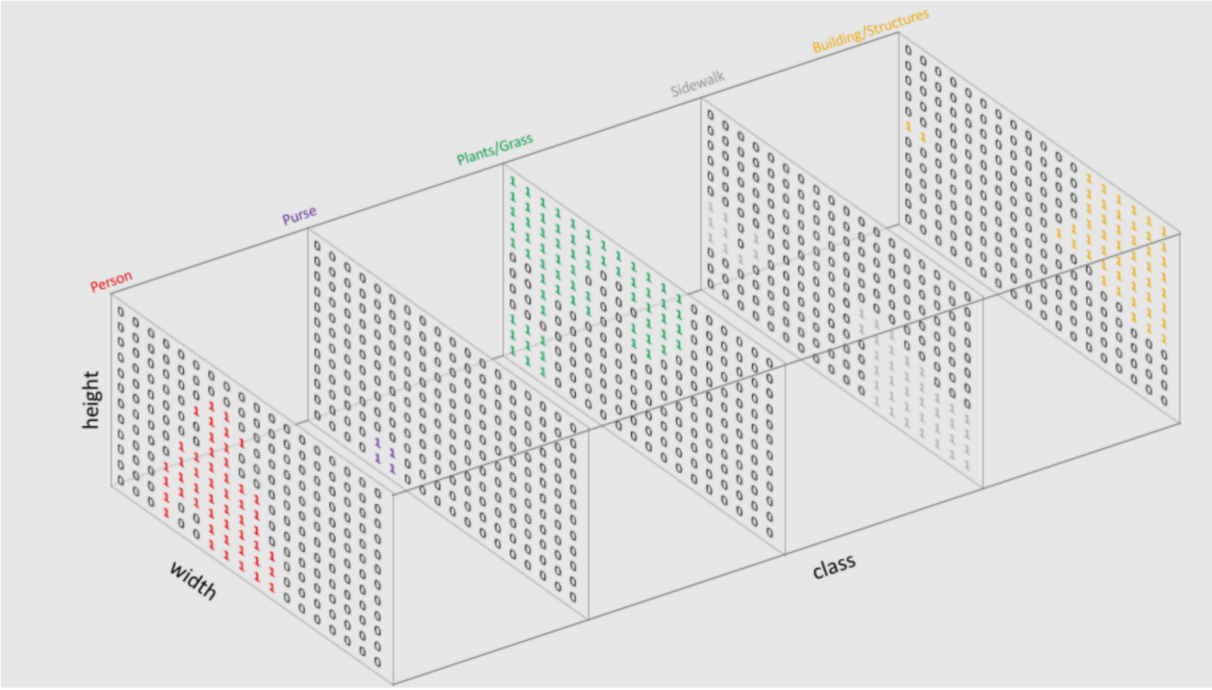

Figure X: Example of Semantic Mask One Hot Encoding[5]

The following shows the scripts used for performing image processing.

Python File	Description
`ImageProcessor.py`	`Object to handle all processing of images/data.`

Figure X: Scripts used for image preprocessing.

3.5 Custom DataLoader

A custom data loader is developed in order to wrap the loading of an image, perform data augmentations, and data preprocessing in batches during training time.

```
class CustomDataLoader:
    '''
    Object to handle data generator.
    '''
```

The CustomDataLoader class has a *__getitem__()* method that loads the index of the image and mask, performs data augmentation if required, and the pre-processing steps. It does it for each of the training, validation, and testing datasets.

```
    def __getitem__(self, idx):
```

```
        '''
        Params:
            self: instance of object
            idx (int): index of iteration
        Returns:
            img_tensor (pt tensors): processed image as tensors
            mask_tensor (pt tensors): processed masks as tensors
        '''
```

The following shows the scripts used for the custom data loader.

Python File	Description
`CustomDataLoader.py`	`Object to handle data generators.`

Figure X: Scripts used for custom data loader.

3.6 Plotters

Finally, our team developed a *Plotter()* class in order to handle all of our plotting needs. We especially liked to use it as a sanity check at each step within our data processing, training, and evaluation process that our processes are working as intended. The various plots of the image, mask, and channel breakdowns throughout this report and our presentation were created using the methods within this class.

```
class Plotter:
   '''
   Object to handle plotting of images.
   '''
```

Python File	Description
`Plotter.py`	`Object to handle all plotting of images/ground truth masks.`

Figure X: Scripts used for plotting.

4. Exploratory Data Analysis

4.1 Example Images

Our team started off our EDA process by visually exploring our dataset. We did this by plotting the original images and their associated ground truth masks side-by-side in order to help us better understand the dataset that we are working with. Below are a few examples of the rendered images, real images, and their associated ground truth masks.

Rendered Images Samples

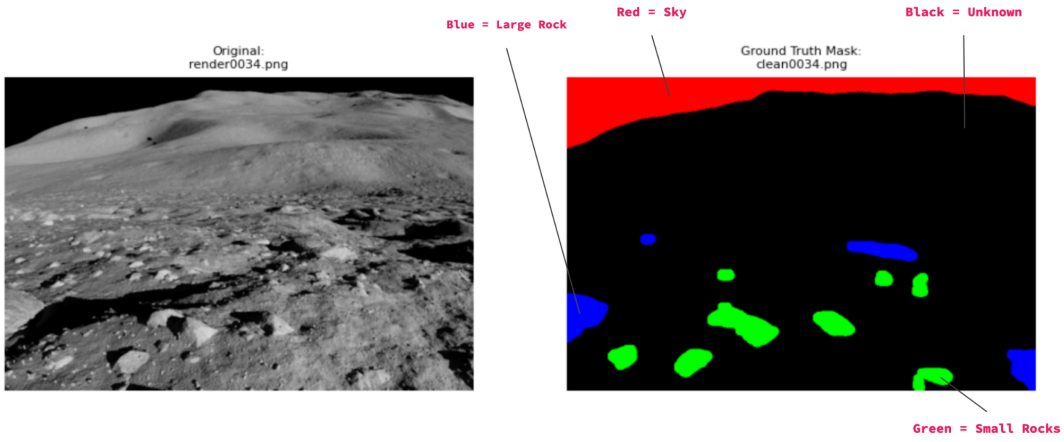

Figure X: Sample Image I - Rendered Lunar Image and Associated Ground Truth Mask

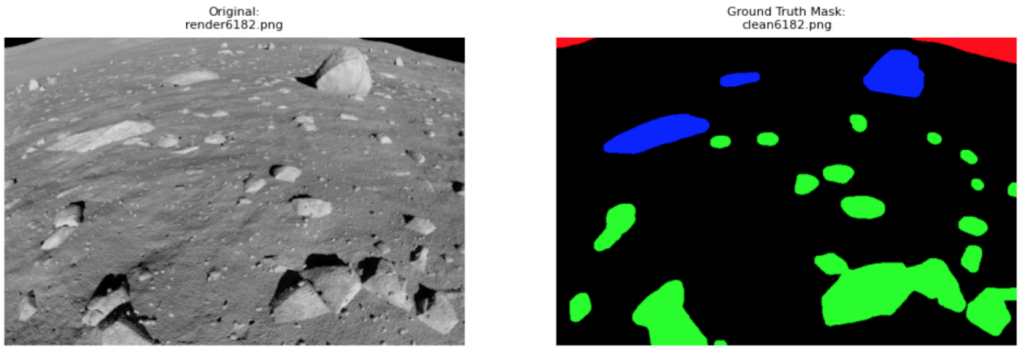

Figure X: Sample Image II - Rendered Lunar Image and Associated Ground Truth Mask

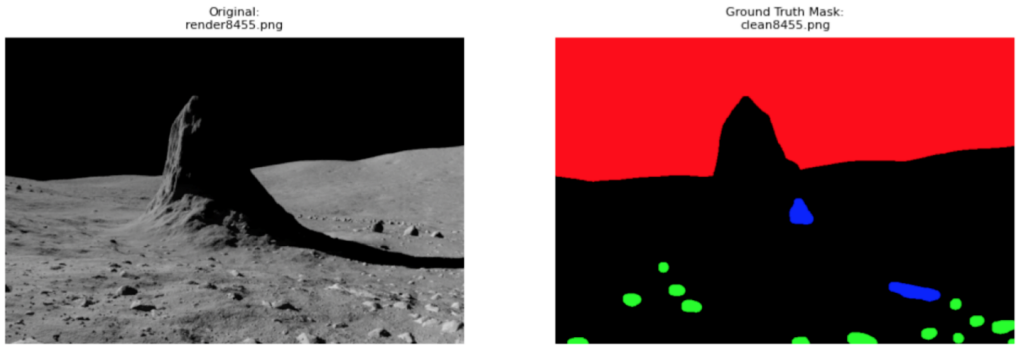

Figure X: Sample Image III - Rendered Lunar Image and Associated Ground Truth Mask

Real Images Samples

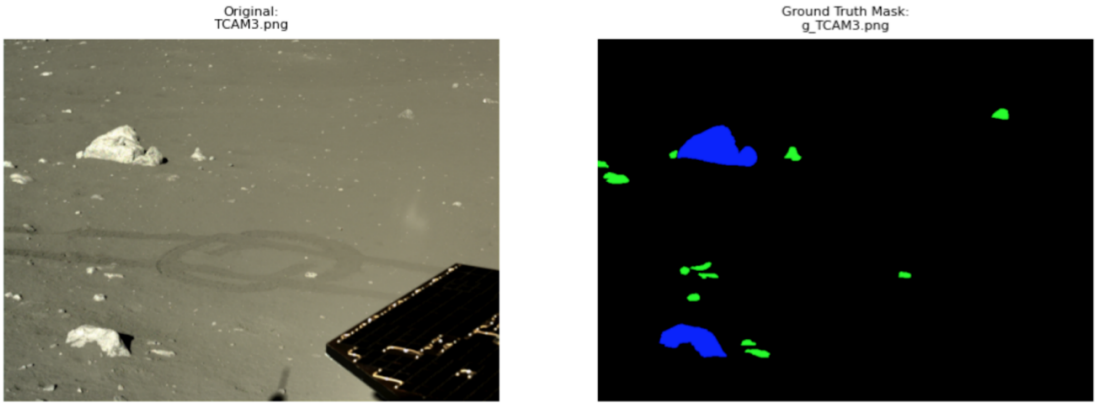

Figure X: Sample Image IV - Real Lunar Image and Associated Ground Truth Mask

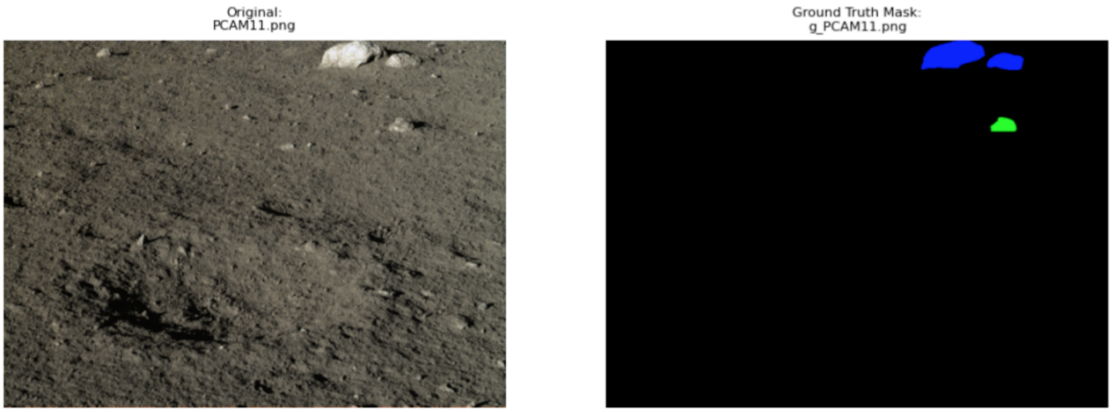

Figure X: Sample Image V - Real Lunar Image and Associated Ground Truth Mask

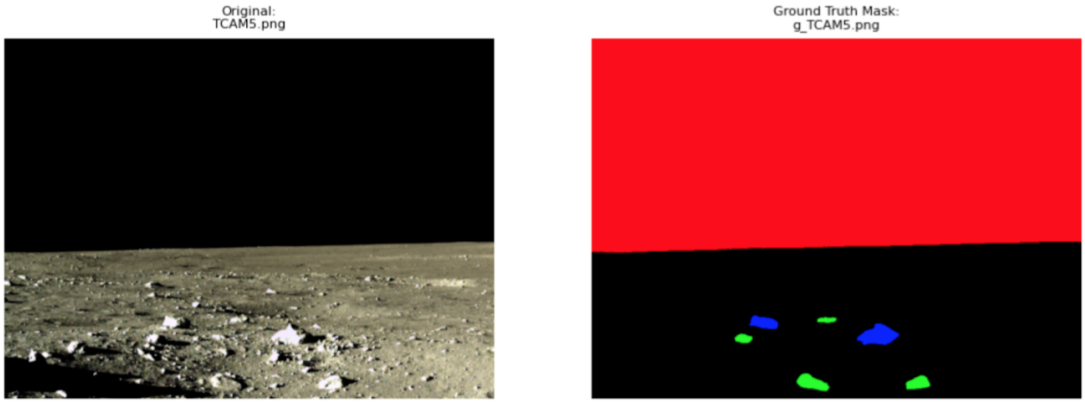

Figure X: Sample Image VI - Real Lunar Image and Associated Ground Truth Mask

4.2 Class Breakdowns

Next in EDA, our team explored the class balance by looking at the pixel percentage by each of the classes for the entire dataset. In the following figure, we observe that the majority class is from the “Unlabeled” class, followed by the “Sky”, then “Small Rocks”, and “Large Rocks” respectively. This is important to note as correctly classifying the small and large rocks is the key objective of our project.

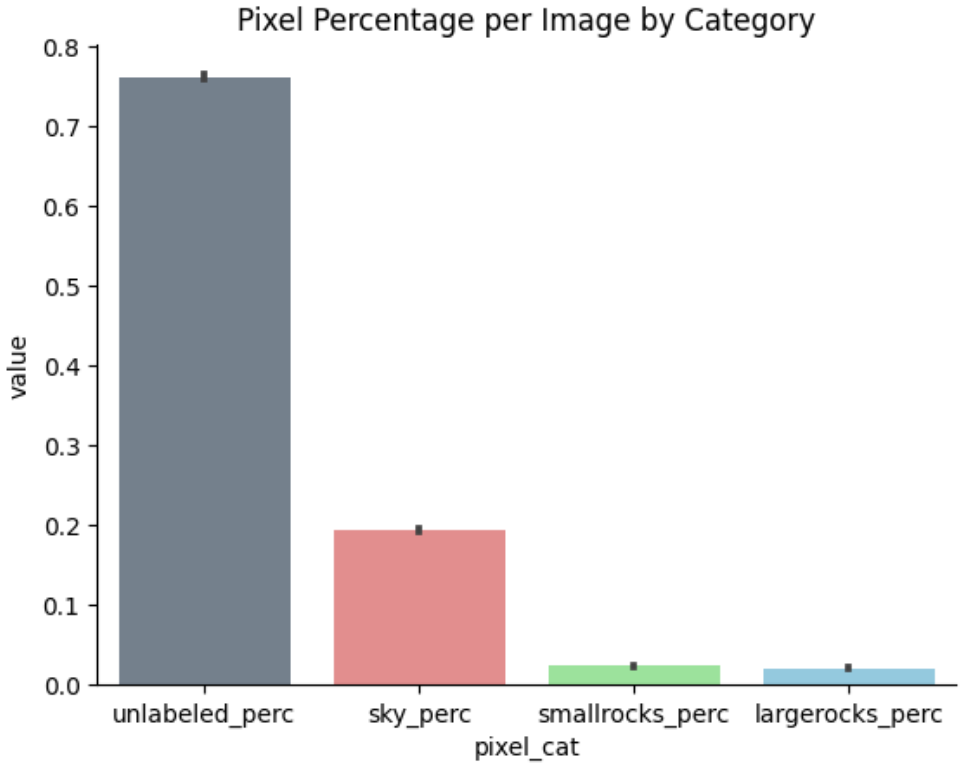

Figure X: Class Imbalance Breakdown - Count of Pixels

Additionally, our team also explored the percentage of images with a particular class label occurrence. “Unlabeled” and “Sky” occurred in almost every image within the dataset while “Small Rocks” and “Large Rocks” occurred less at under 85% and under 70% each.

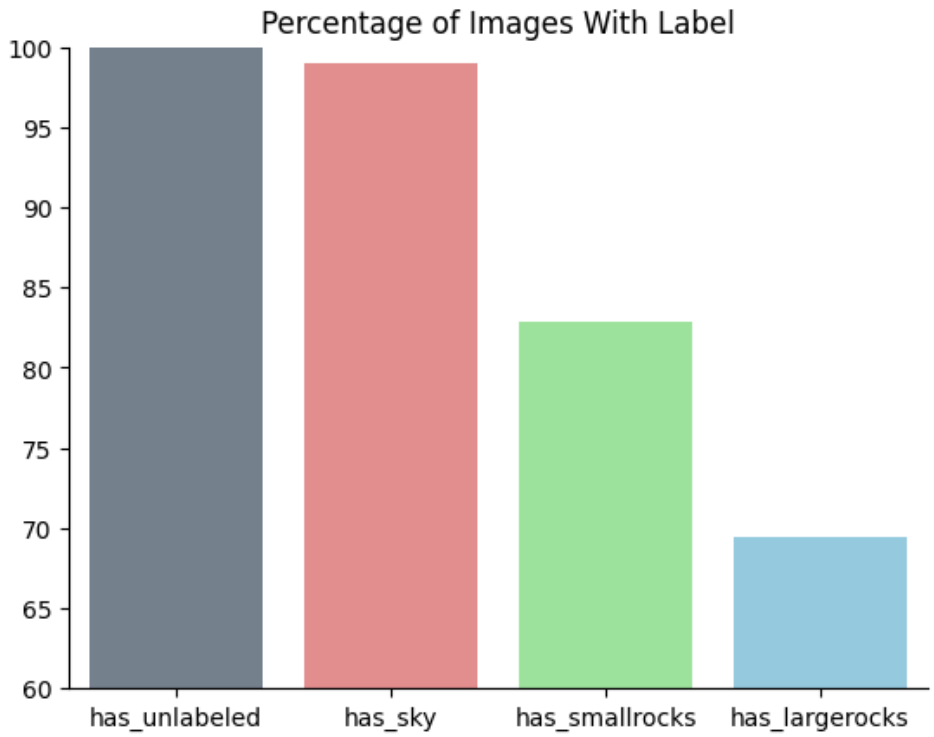

Figure X: Class Imbalance Breakdown - Percentage of Images with Specific Label

4.3 Class Organization

Lastly in EDA, our team wanted to explore where within the borders of an image were the highest occurrences of each class label by pixel location. The following figure shows for each of the 4 classes, where were the pixel locations that the respective class occurs the most, or to put simply, the class intensity by pixel location within the borders of the image.

It’s interesting to note that the sky is largely most common in the ⅓ of the image while the unlabeled background is generally most common in the lower ⅔ of the image. Meanwhile for rocks, the larger rocks are most common in the middle of the image while smaller rocks are in the bottom half of the image. The reasoning being that larger rocks are usually also tower and therefore take up more vertical space while smaller rocks are usually smaller and scattered across the ground which generally is at the bottom half of an image.

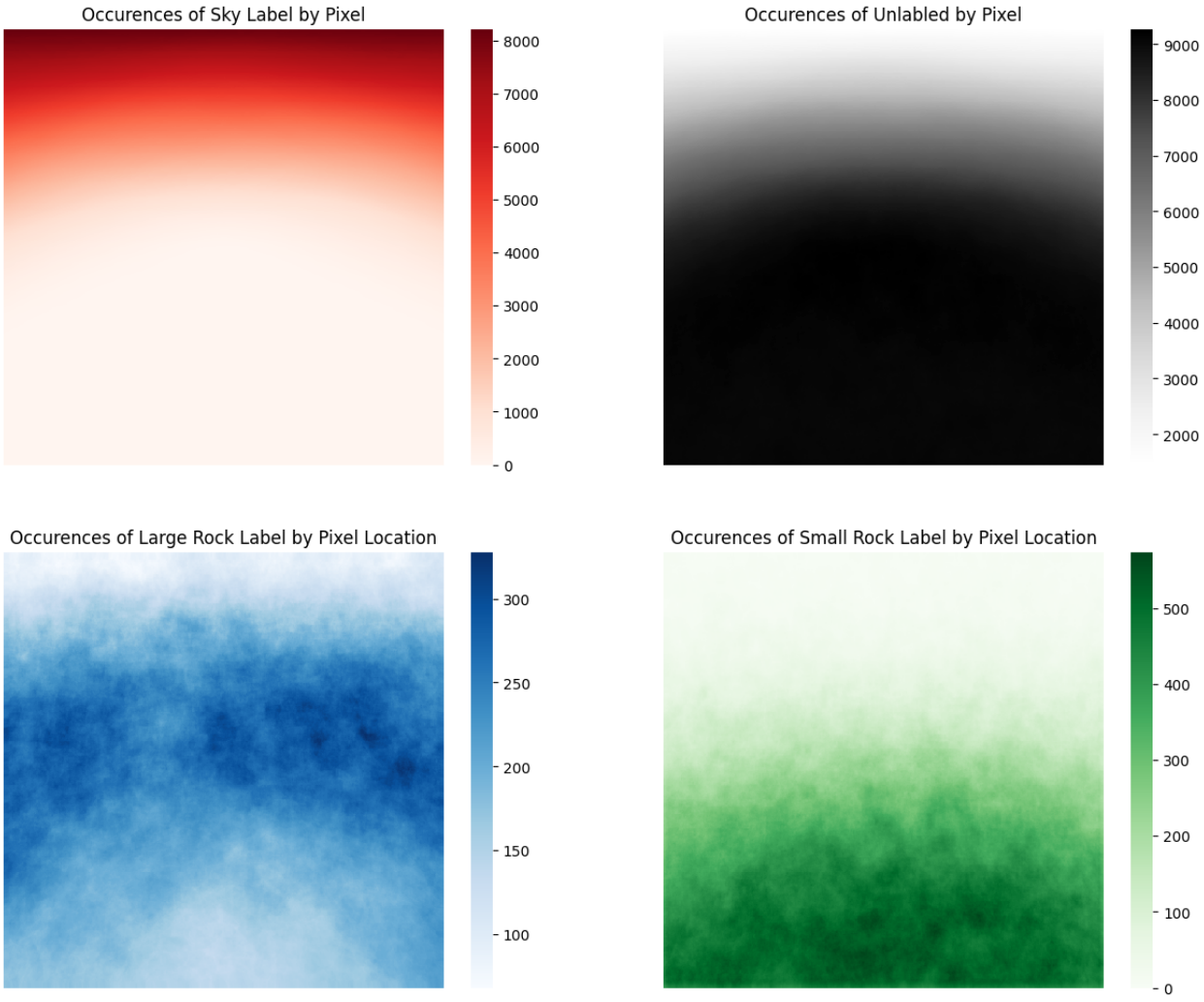

Figure X: Class Occurence Intensity by Pixel Location

5. Experimental Setup - Modeling

5.1 Introduction

5.2 Custom Model Structure

5.3 Metrics

5.4 Optimization Algorithm

5.5 Loss

5.6 Weights

5.7 Hyperparameters

5.8 Class Setup

6. Results

6.1 Testing and Model Evaluation

6.2 Model Comparison

6.3 Code Percentage

74%.

7. Conclusions

Our group succeeded in our objective to train and compare several models to perform the task of semantic segmentation on images of the Lunar surface. Our group learned a lot from working on this project in how to tackle not just a semantic segmentation task but also generally a computer vision task since many of the processing techniques and core modeling ideas are similar between the various tasks within the computer vision domain.

8. References

1. Jonathan Long et. al (2014) - Fully Convolutional Networks for Semantic Segmentation
2. Ronneberger et. al (2015) - U-Net: Convolutional Networks for Biomedical Image Segmentation
3. Artificial Lunar Landscape Dataset on Kaggle
4. Lunar Surface Image - thespaceacademy.org
5. An Overview of Semantic Segmentation
6. Stanford CS231: Detection and Segmentation
7. Kaggle - Artificial Lunar Landscape Dataset
8. Kaggle - Artificial Lunar Landscape Dataset - Silver Notebook
9. Jaccard Index
10. Understanding and Visualizing ResNets
11. Architecture and Implementation of VGG16
12. MobileNet v3
13. Metrics to Evaluate Semantic Segmentation
14. Cross Entropy Loss

9. Appendix

9.1 Random Seed

Random seed used in the modeling phase was *42*.

9.2 Computer Listing