Lunar Surface Image Segmentation

George Washington University
Machine Learning II - DATS 6203_10
Group 5
Sahara Ensley, 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.

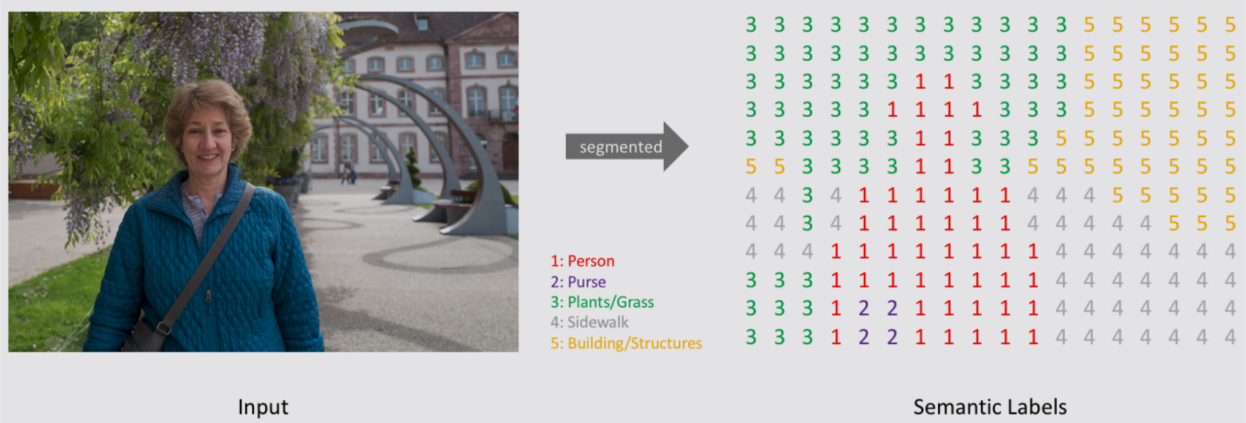

Figure 1: Overview of Semantic Segmentation[5]

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 2: Breakdown of Ground Truth Masks' Color Coding

2.3 Framework and Network Selection

In order to have maximum control over our model's architecture, parameters, and training loop, we decided to use Pytorch to complete our project.

In order to solve our problem we decided to use Ronneberger et. al (2015)'s U-Net model[2] to segment our images. A U-Net is a form of an encoder-decoder where the feature outputs from each block of the encoder structure are concatenated with the upsampling outputs of each block of the decoder.

The structure we used for our U-Net is as follows:

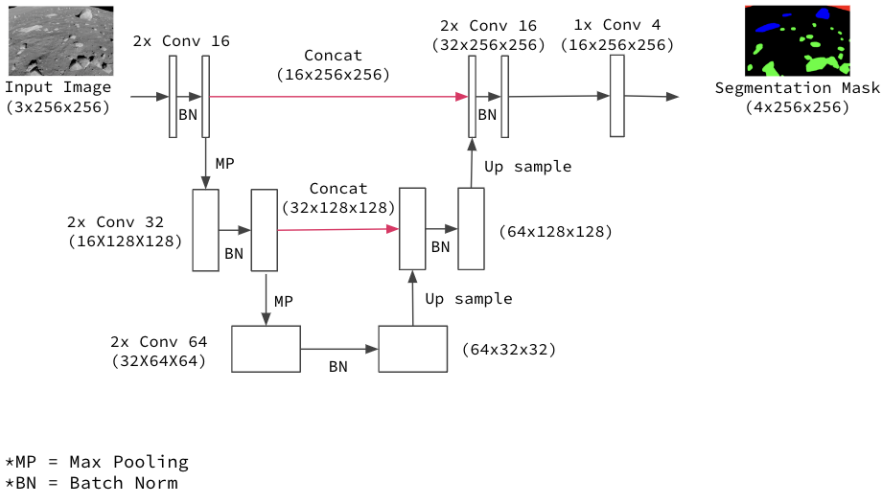

Figure 3: Outline of Custom U-Net model

The Encoder is the 'down' portion of the U-Net. Each block of the Encoder down samples the input by passing it through pooling layers following convolutional layers to identify features. These feature maps are then saved to be used by the Decoder. Conversely, the Decoder up-samples the output of the final Encoder layer and subsequent Decoder layers by passing it through transposed convolutional layers. Before every Decoder block the feature map of the same level Encoder block is concatenated to the input, this allows the Decoder to retain feature

information while upsampling the image back to its original size. The final step of the U-Net is to pass the output of the last Decoder block through a classification head made up of a single convolution layer with a softmax activation function.

We decided to go with a relatively simple structure in order to avoid overfitting our model and to reduce the necessary training time. The model is discussed in-depth in the Modeling section of our report.

We also plan on performing transfer learning and testing multiple different U-Nets with pre-trained models as the Encoders that connect to an untrained Decoder. We plan to use versions of ResNet, VGG, and MobileNet pretrained models and compare the outputs to see which model performs the best on this dataset. The number of parameters for each of the pretrained model are as follows.

Model	Parameters (M)
Custom	0.1
MobileNetv3 Large	5.4
ResNet 18	11
VGG 11	18

Figure 4: Parameter Count by Model

2.4 Contribution Goals

After reviewing other attempts at segmenting this dataset we noticed that almost every model created from scratch was only capable of identifying the ground and the sky of the image with no correctly identified rocks. One of the most successful attempts can be found here[8], this attempt used a VGG backbone as the encoder of their U-Net and got the most successful results. Our goal was to create our own U-Net, and then use transfer learning on multiple U-Nets with different pretrained encoder backbones to determine the most successful technique. We plan on leveraging the same python package that some of the most successful attempts used, Segmentation-Models-Pytorch. This package returns U-Nets with specified backbones and allows us to tune the hyperparameters and train on our own dataset. Since our goal was to compare backbones this was the most efficient way of creating multiple models.

3. Experimental Setup - Data Preprocessing

3.1 Data Acquisition

Kaggle Method - Initial Data Download
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 5: Scripts used for data acquisition from Kaggle.

Google Drive - Data Distribution
Alternatively, our team also uploaded the same dataset into a public Google Drive for easier data download and access without the need to configure the Kaggle API credentials beforehand. We will be using this secondary method for our main script as it simplifies the data acquisition process.

The following shows the scripts used for data download from our Google Drive distribution.

Python File	Description
google_drive_data_download.py	Script to download data from Google Drive link

Figure 6: Scripts used for data acquisition from Google Drive.

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 6: 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 7: 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 8: 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.

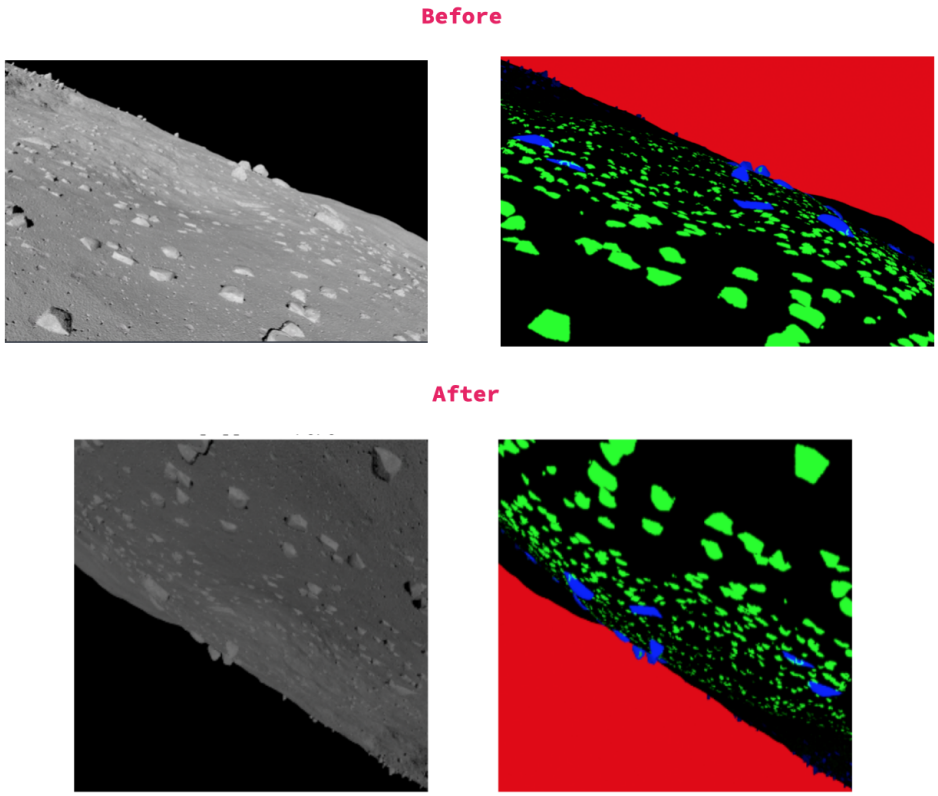

Figure 9: Example data augmentation before and after.

The following shows the scripts used for performing data augmentation.

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

Figure 10: 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 11: 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.

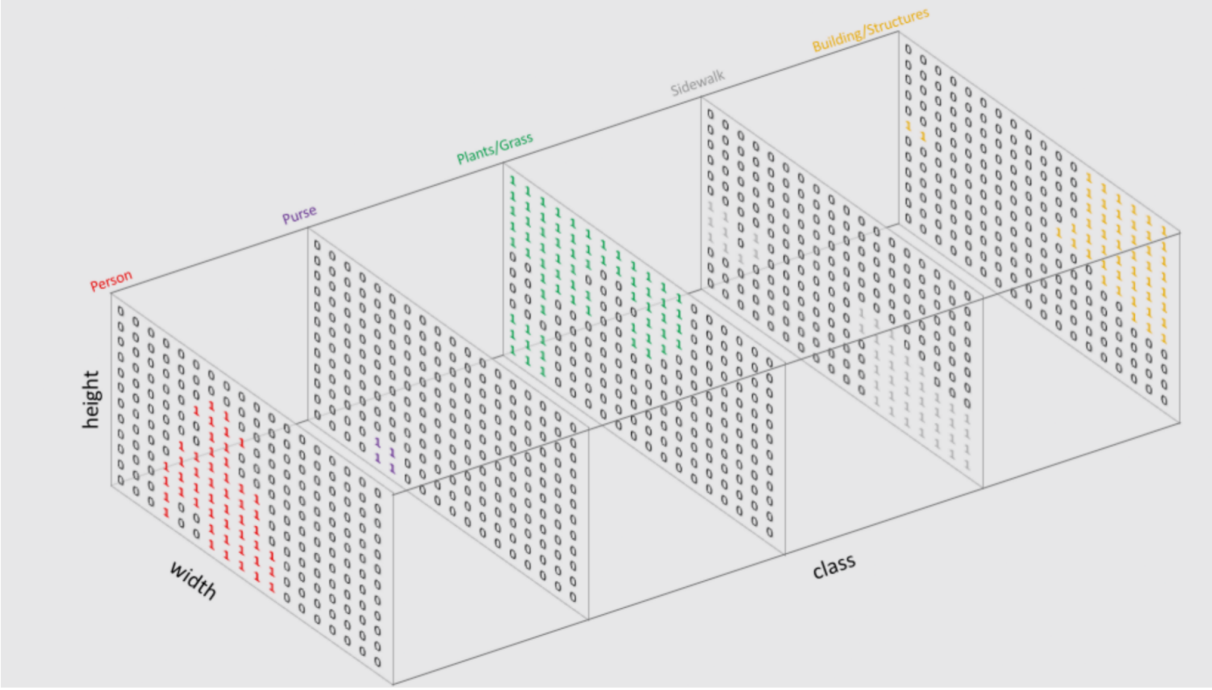

Figure 12: Example of Semantic Mask One Hot Encoding[5]

The following shows the scripts used for performing image processing.

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

Figure 13: 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 14: 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 15: 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

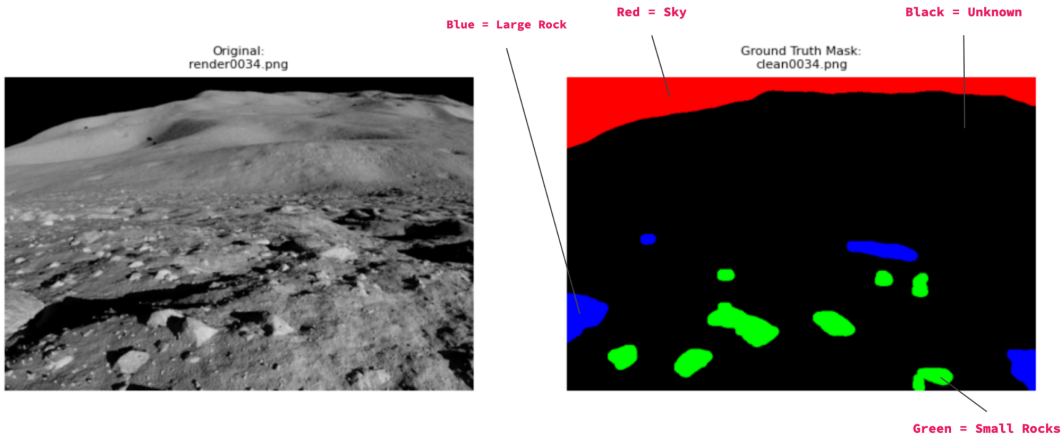

Figure 16: Sample Image I - Rendered Lunar Image and Associated Ground Truth Mask

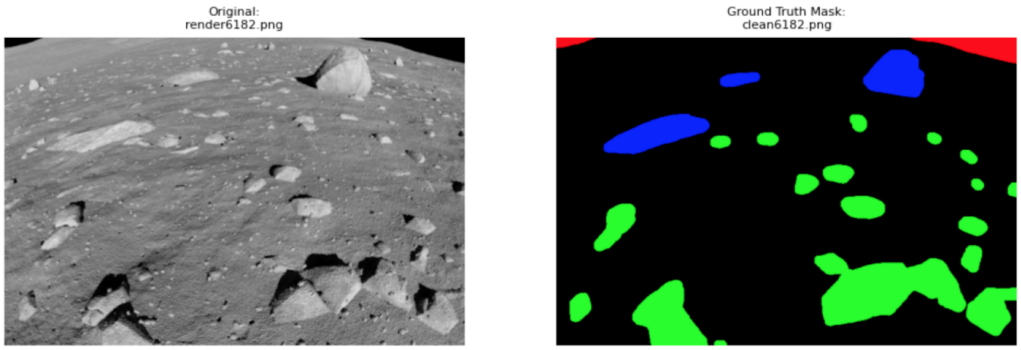

Figure 17: Sample Image II - Rendered Lunar Image and Associated Ground Truth Mask

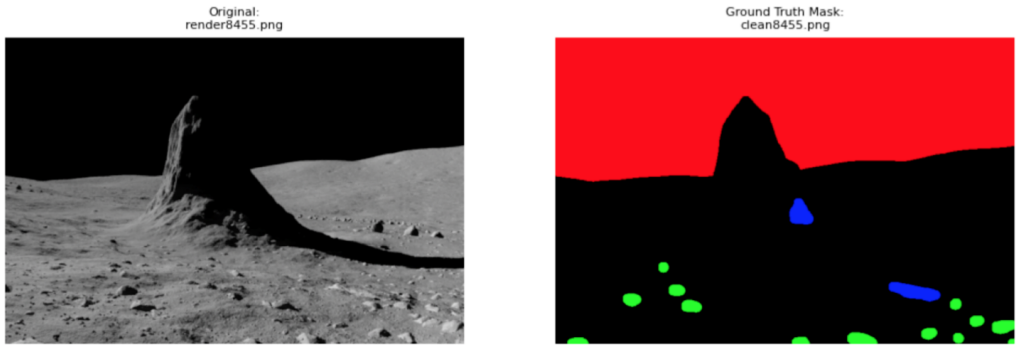

Figure 18: Sample Image III - Rendered Lunar Image and Associated Ground Truth Mask

Real Images Samples

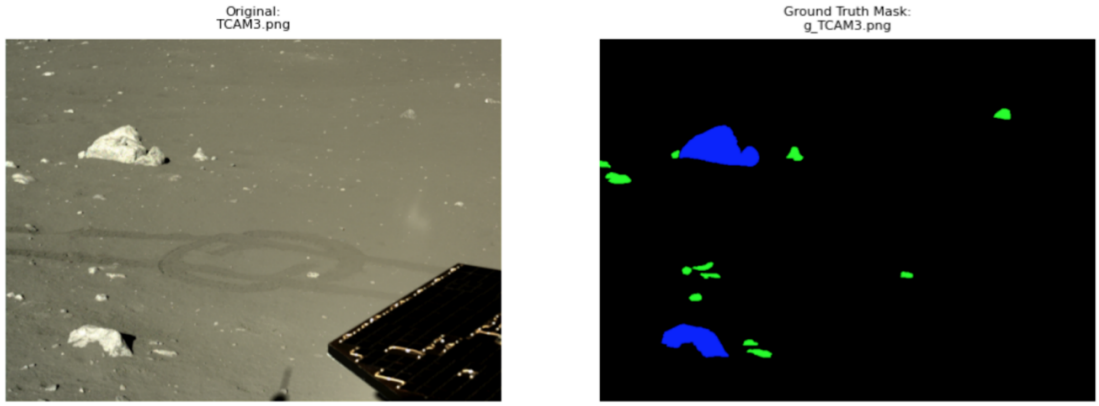

Figure 19: Sample Image IV - Real Lunar Image and Associated Ground Truth Mask

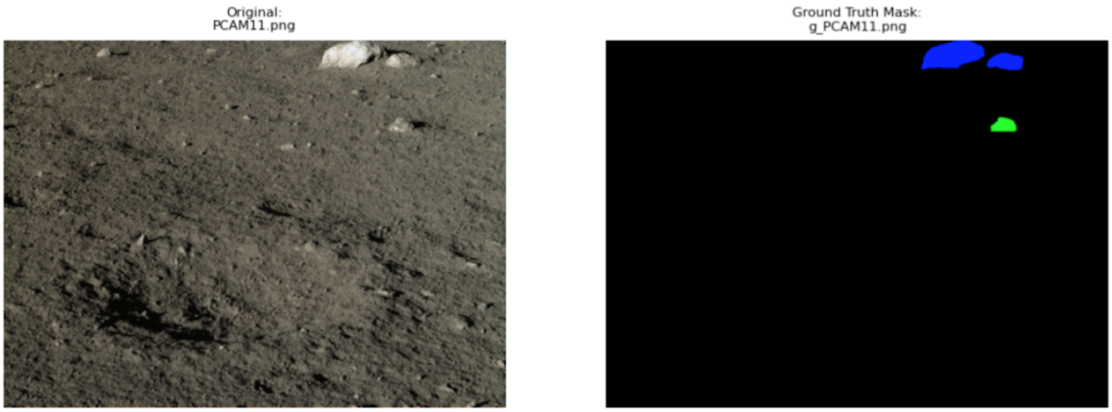

Figure 20: Sample Image V - Real Lunar Image and Associated Ground Truth Mask

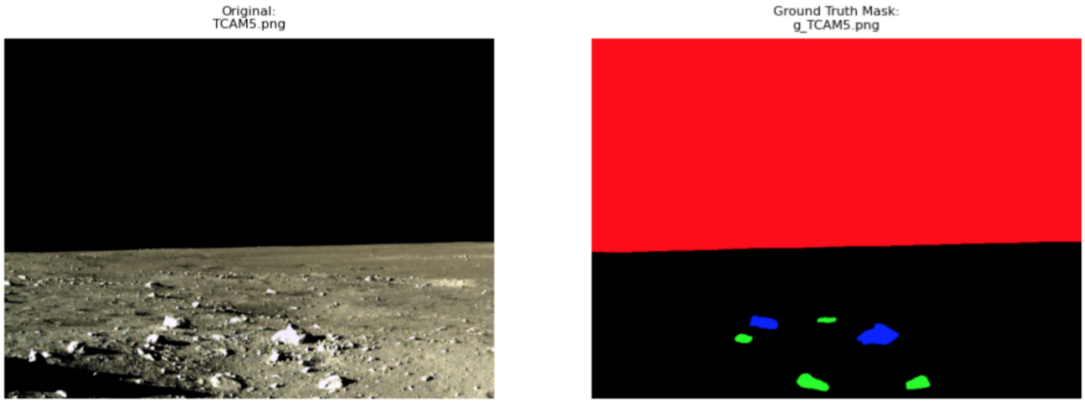

Figure 21: Sample Image VI - Real Lunar Image and Associated Ground Truth Mask

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.

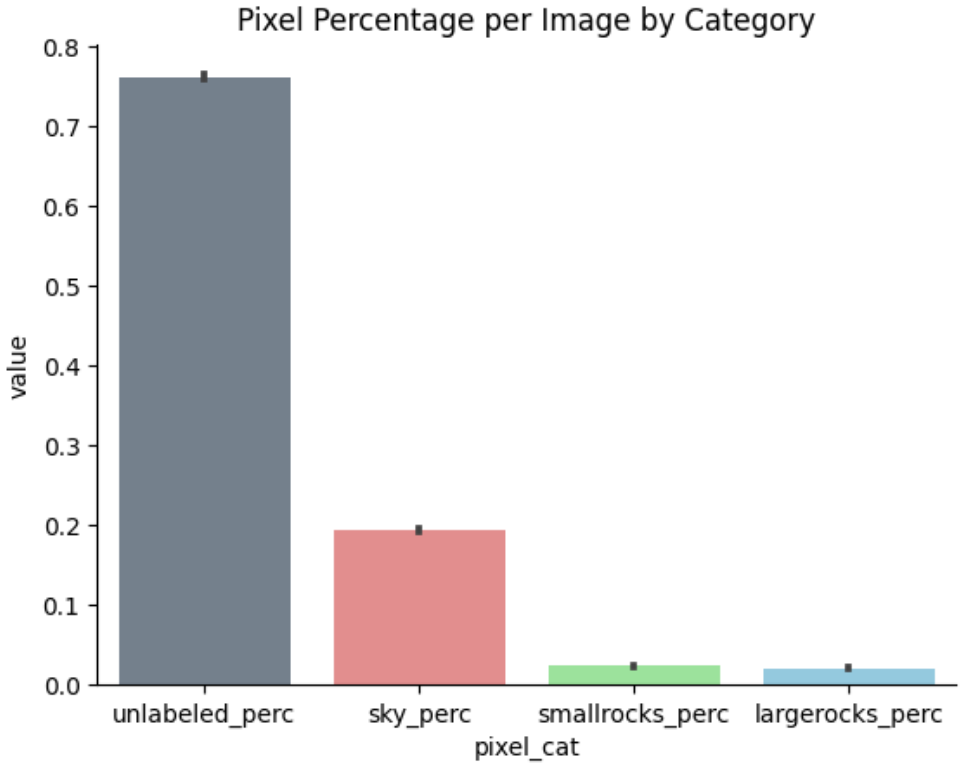

Figure 22: Class Imbalance Breakdown - Count of Pixels

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.

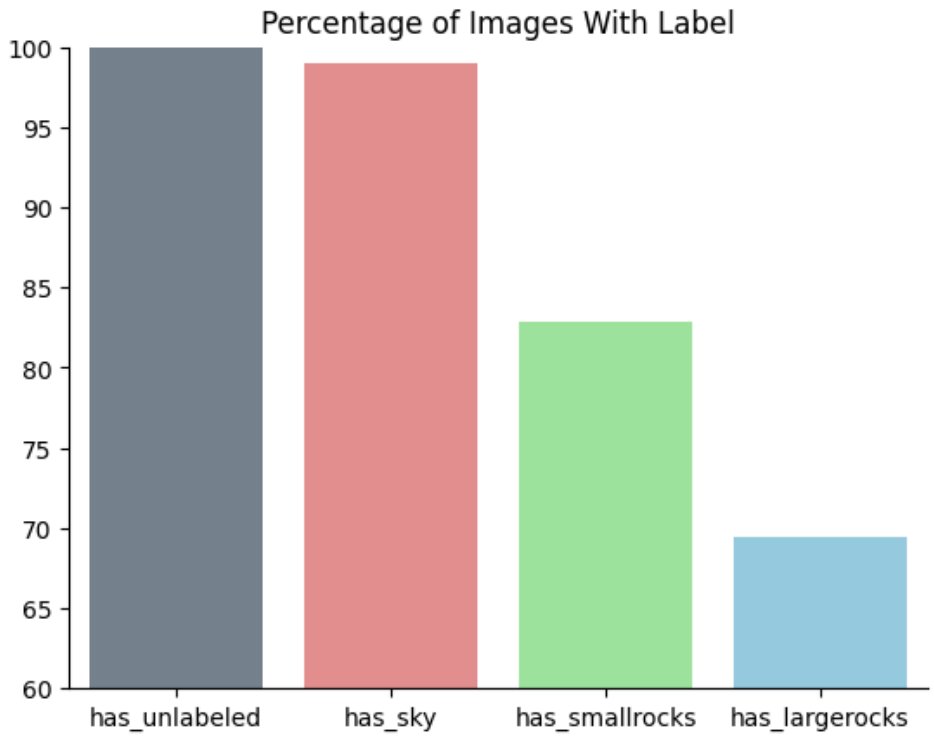

Figure 23: Class Imbalance Breakdown - Percentage of Images with Specific Label

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.

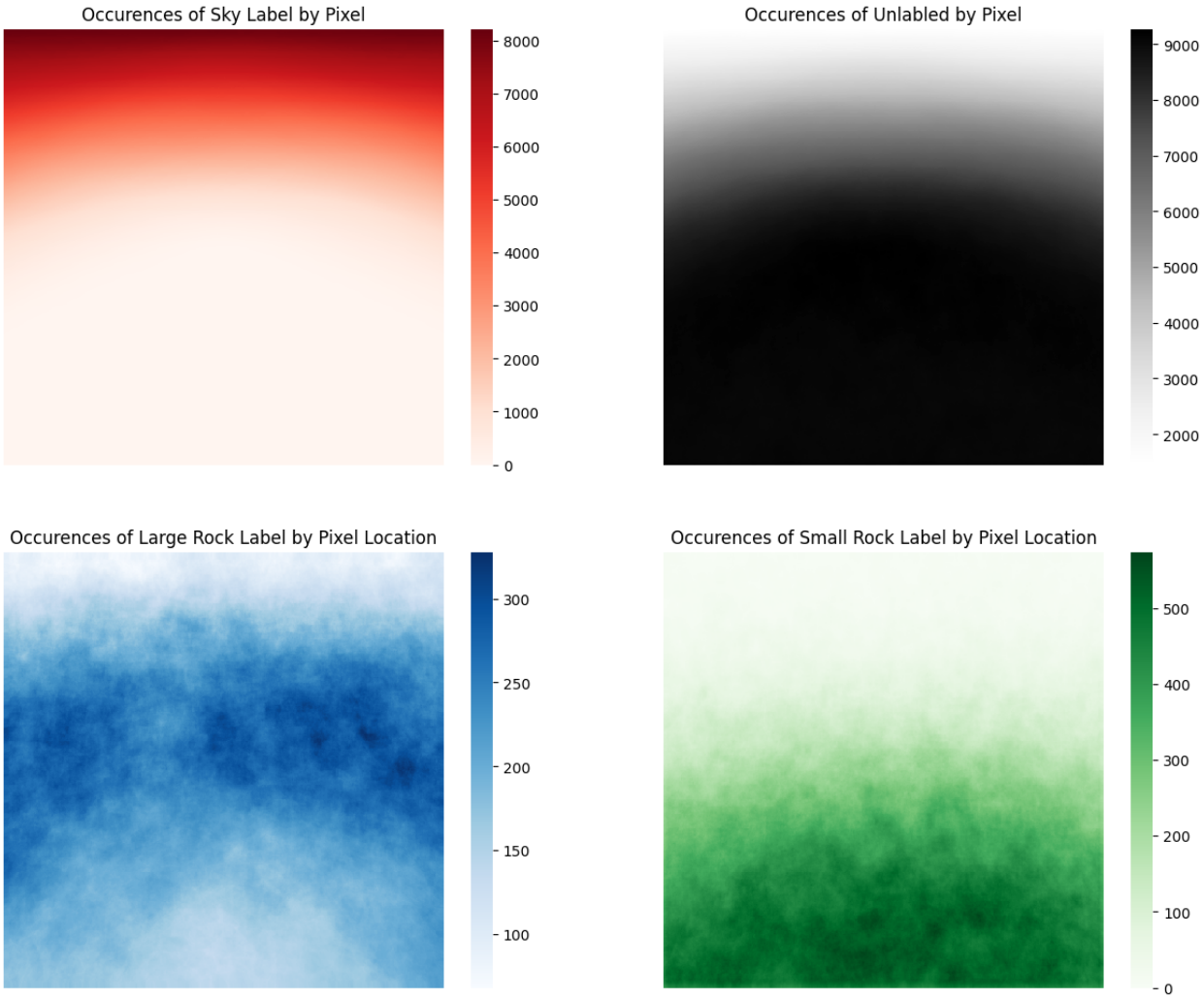

Figure 24: Class Occurence Intensity by Pixel Location

5. Experimental Setup - Modeling

5.1 Introduction

The motivation behind our modeling was to see if we could have created a custom U-Net that was able to segment the lunar images successfully. We then trained multiple U-Nets with pre-trained Encoders and untrained Decoders. To do this we created 2 unique python classes, one for our custom U-Net and one for the pre-trained models. These classes acted as wrappers around our 2 model types. In order to get our pretrained models we used a python package called *Segmentation_Models_Pytorch* which was compatible with multiple different pretrained models, three of which we trained and compared against each other and our own model.

5.2 Custom Model Structure

We began our modeling by creating a custom U-Net model (Fig. 2). Each of the Encoder and Decoder blocks consisted of 2 2-dimensional convolutional layers with a *ReLU* activation and a BatchNorm layer between the first convolutional layer and the activation. Between each Encoder block there was a MaxPooling layer and a Dropout layer. Before every Decoder block there was a 2-dimensional transposed convolutional layer that upsampled the input. In addition there was a crop function called that ensured the feature maps were the correct size to be concatenated with the upsampled input. There was then a concatenation function called to combine the feature maps from the same level of the Encoder with the inputs to the Decoder block. The final classification head of our U-Net contained a single 2-dimensional convolutional layer with a *softmax* activation function.

5.3 Metrics

We decided to use Jaccard's Index to determine the effectiveness of our model. Jaccard's Index, also known as the Intersection over Union (IoU) metric, is a similarity measure that determines how much overlap there is between the predicted mask and the true mask.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Equation 1: Jaccard's Index Calculation[9]

We chose this metric over a simple accuracy calculation because it's more representative of the correct mask than accuracy is. This is especially true for our dataset because of the class imbalance between the unlabeled and sky classes and the rocks classes. If only 10% of an image has a label, and our model incorrectly classified the entire 10%, the accuracy could still be as high as 90% while Jaccard's Index would be significantly lower (actual number depending on specific classification).

5.4 Optimization Algorithm

We chose to use Adam as our optimization algorithm for our custom U-Net. We chose this because of its fast convergence time. We knew we weren't going to train for many epochs so we wanted a trusted and efficient optimization algorithm. For our pre-trained models we chose to use SGD instead of Adam because the backbones were already trained.

5.5 Loss

We chose to use Cross Entropy Loss for this problem because we are working with a multi class classification problem. Since there are 4 possible labels that a pixel can have (sky, big rock, small rock, unlabeled/ground) we could not use Binary Cross Entropy loss. The formula for Cross Entropy Loss is as follows:

$$L(\widehat{y},\ y) \ = \ \ -\sum y\ *\ log(\widehat{y})$$

Equation 2: Cross Entropy Loss Function[14]

5.6 Weights

The weights of both our custom U-Net models and the pre-trained models were initialized. We did this to make the training more efficient and hopefully allow the model to converge faster. We used *He Initialization* to initialize the weights of every convolutional layer of our Encoder-Decoder, while the pretrained models had the weights initialized for the decoders only. We chose this method because our activation function for the individual encoding and decoding blocks was *ReLU*. It's calculated by choosing a random value from a normal distribution with a mean of 0 and a standard deviation that changes depending on the number of inputs to that layer.

$$weight \ = \ Gaussian(mean \ = \ 0.0,\ std \ = \ \sqrt{2/n})$$

Equation 3: Weight Calculation

5.7 Hyperparameters

The hyperparameters for our custom model were the same for every block of our Encoder and Decoder (if applicable in multiple locations). Our hyperparameters were chosen either out of best practice or trial and error tuning.

Hyper Parameter	Value
Kernel Size (Block Layers)	3x3
Kernel Size (Classification Head)	1x1
Padding	'same'

Pooling	MaxPooling
Pooling Kernel Size	2x2
Dropout Level	0.2
Learning Rate	0.001

Figure 25: Model Hyperparameters

The hyperparameters for the pre-trained model were the same except for the following values.

Hyper Parameter	Value
Pooling	Average Pooling
Padding	1

Figure 26: Pre-trained HyperParamters

These values were different because the python package that provided the pre-trained model did not allow us to tune these values. However, we know from reading their documentation that they tuned them when creating the package so we trust their values.

5.8 Class Setup

In order to make our training code more efficient and easy to read we created 2 custom classes that acted as model wrappers, allowing us to call single functions for training and testing.

The breakdowns of the custom classes we created are as follows:

Class	Description
`class Down(nn.Module):`	Created the Encoder structure for our custom U-Net
`class Up(nn.Module):`	Created the Decoder structure of our custom U-Net
`class UNet_scratch(nn.Module):`	Created the final U-Net model, linked the Encoder and Decoder together and added the classification head
`class Model:`	Wrapper class meant to be used around any instance of a Pytorch model
`class Pretrained_Model:`	Wrapper class meant to be used around any instance of a pre-trained U-Net from the python package *Pretrained-Models-Pytorch*

Figure 27: Model Classes

The breakdown of the functions for our ***Model*** class are as follows:

Function	Description
`__init__(self, model, loss, opt, scheduler, metrics, random_seed, train_data_loader, val_data_loader, test_data_loader, real_test_data_loader, device, base_loc = None, name = None, log_file=None):`	Initialized the Model object with necessary hyperparameters, training utilities, and data loaders. Also set up the history of the model for plotting results
`def run_training(self, n_epochs, save_on = 'val_IOU', load = False):`	Ran the training loop for the model, saved and loaded the model as necessary
`def run_test(self):`	Ran prediction tests on the testing dataset and saved results
`def predict(self, img):`	Predicts mask of a single image
`def plot_train(self, save_loc)`	Plots training learning curves
`def save_model(self, epoch):`	Saves current model state
`def load_latest_model(self, device):`	Loads the best model with the same name as the current model

Figure 28: Model Class Setup

The breakdown of the functions and scripts for our ***Pretrained Model*** class are as follows:

Function	Description
`def __init__(self, backbone, encoder_weights, activation, metrics, LR, loss, device, train_data_loader, val_data_loader, test_data_loader, base_loc, name = None):`	Initialized the Pretrained Model object with necessary hyperparameters, training utilities, and data loaders. Also set up the history of the model for plotting results
`def run_training(self, n_epochs, load = False):`	Ran the training loop for the model, saved and loaded the model as necessary
`def run_testing(self):`	Ran prediction tests on the testing dataset and saved results
`def save_model(self, epoch):`	Saves current model state
`def load_latest_model(self, device):`	Loads the best model with the same name as the current model

Figure 29: Pre-trained Model Class Setup

In addition to the classes that we created we wrote multiple utility functions to aid in our training and testing scripts. Please see our GitHub documentation for detailed descriptions about those functions as well as any of the functions or scripts mentioned above.

6. Results

6.1 Testing and Model Evaluation

Custom U-Net

The first model we evaluated was our custom U-Net model that we designed and trained. We first looked at the loss over training and validation.

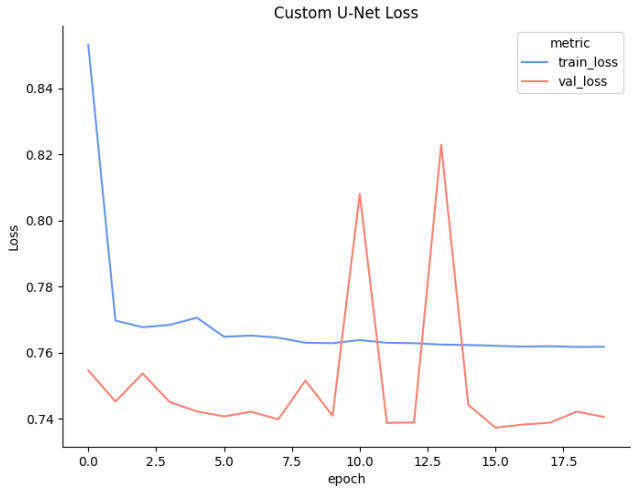

Figure 30: Custom U-Net Loss

We can see from this plot that the loss did not decrease much after the first few epochs, with a couple sharp upticks of the validation loss halfway through. We believe this is because our model was not complex enough to solve the segmentation problem in our dataset. We can see this further by examining the Jaccard's Index metric we used to evaluate our model.

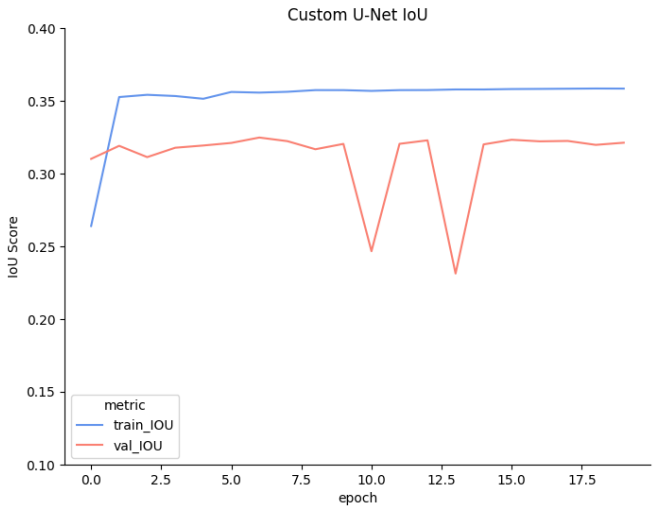

Figure 31: Custom U-Net IoU

We can see a similar trend as the loss, with the model not improving substantially after the first few epochs. We can look at example predictions to get an idea of what's occurring in the model.

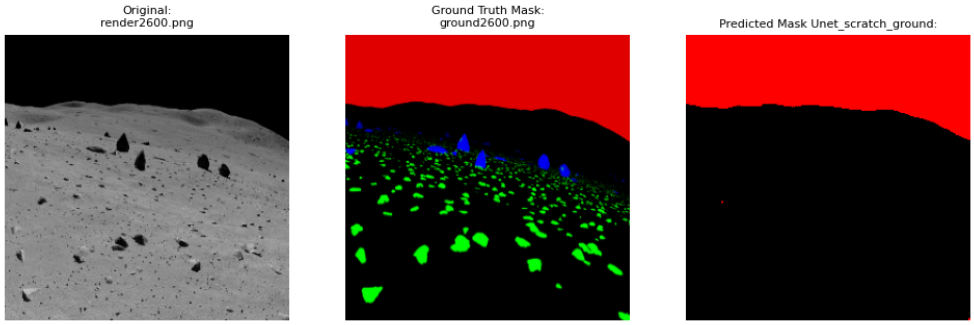

Figure 32: Custom U-Net Prediction

We can see from this example that the model is only learning how to predict the sky and the ground and never learning the rock classifications. This was a problem that occurred with many of the previous attempts we had seen on this dataset.

VGG11 Pretrained Backbone U-Net

After looking at our custom model, we can look at the same features in our U-Nets with pre-trained backbones. Beginning with the VGG11 backbone we can see that both the loss and IoU show improved learning from epoch to epoch.

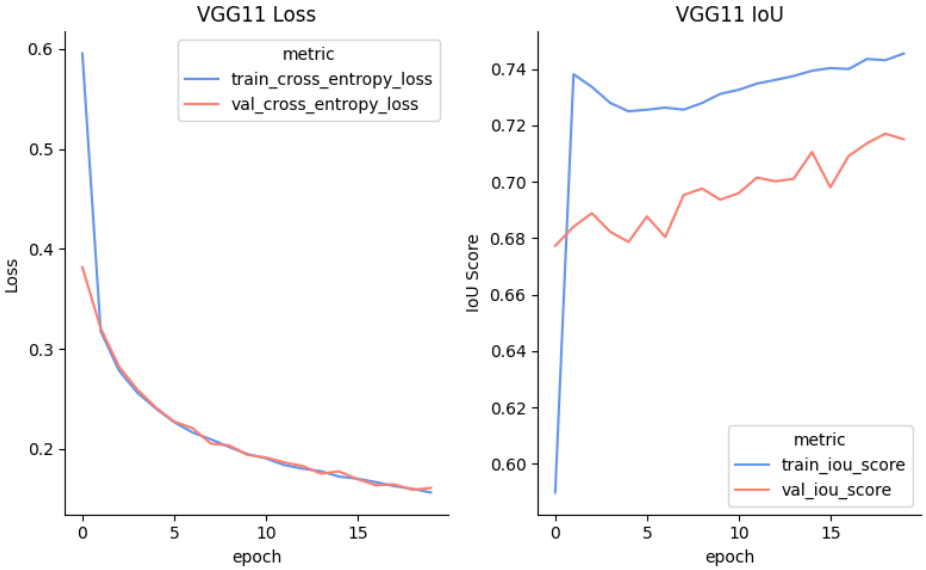

Figure 33: VGG11 Training History

We can see improved training in comparison to our custom model and this is exemplified by looking at an example mask prediction.

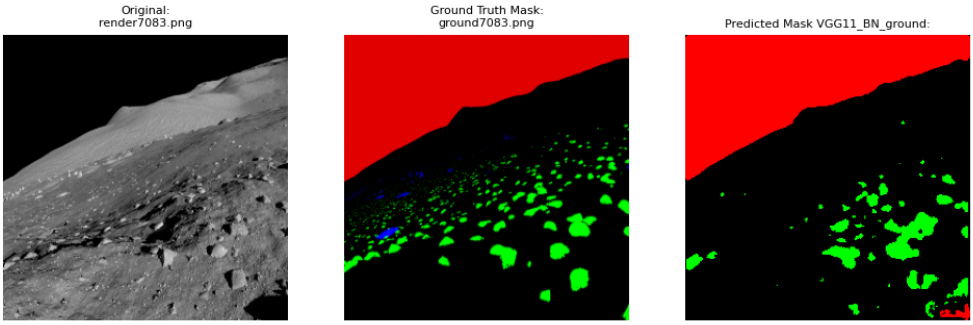

Figure 34: VGG11 Example Prediction

This example shows the ability of the VGG model to identify both rocks and sky, unlike our custom model.

ResNet18 Pretrained Backbone U-Net

Our ResNet18 backbone model trained very similarly to our VGG11 model.

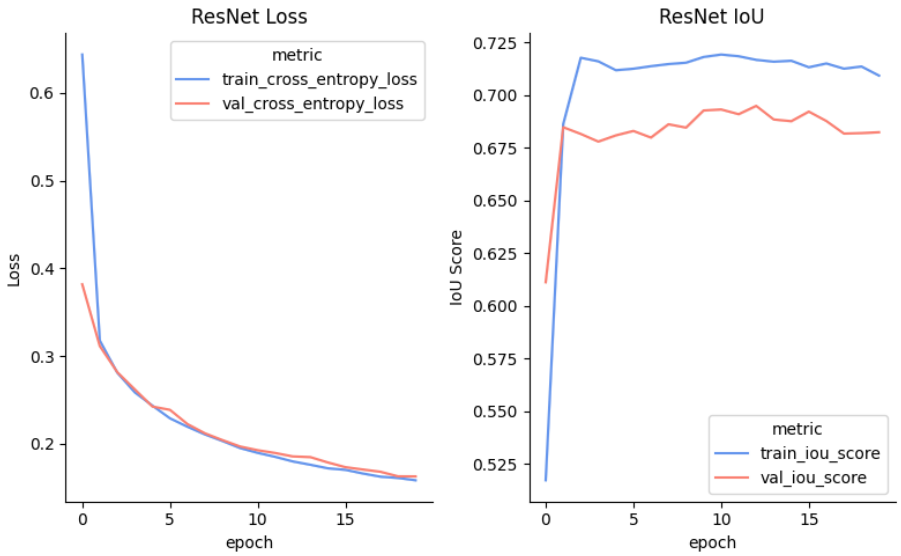

Figure 35: ResNet18 Training History

We can see similar evidence of learning continuing through the epochs along with a higher IoU compared to our custom model.

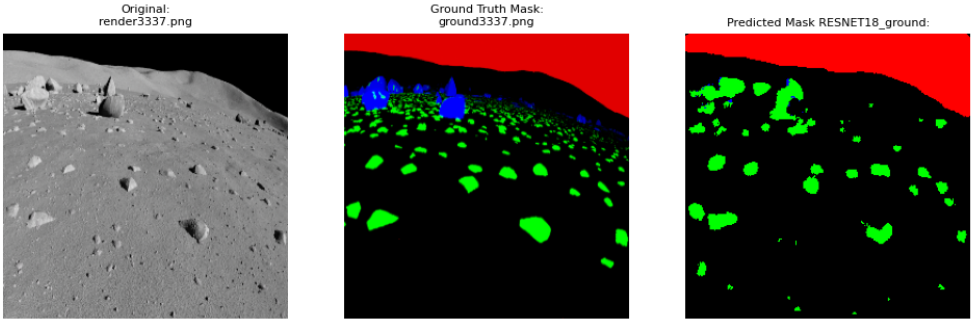

Figure 36: ResNet18 Example Prediction

From this example we can see that the ResNet also learned to identify the sky and small rocks (green) while struggling with the large rocks (blue). This continues to be similar to the VGG model.

MobileNetv3 Pretrained Backbone U-Net

The final model we trained and tested was the MobileNetv3 model.

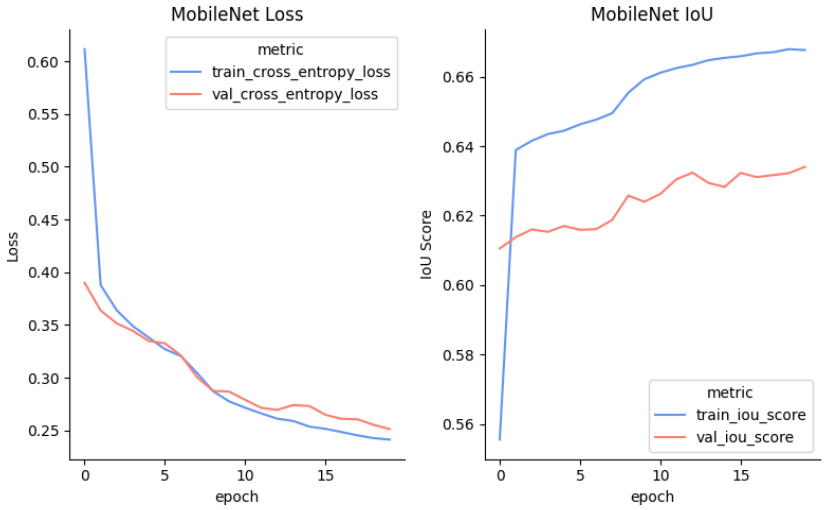

Figure 37: MobileNetv3 Training History

The MobileNet showed evidence of learning similar to the other pretrained models we used.

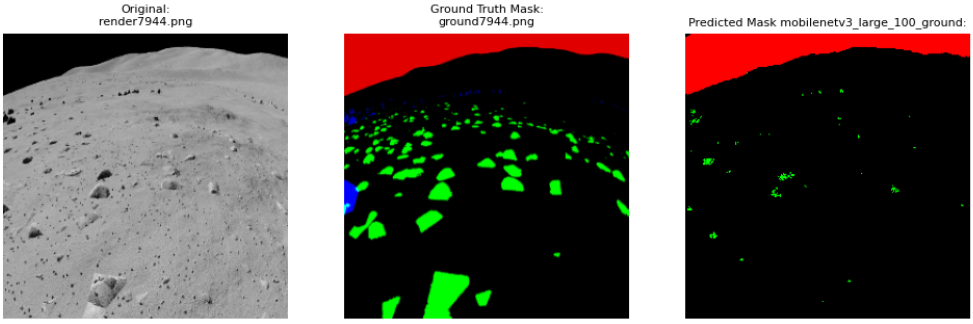

Figure 38: MobileNetv3 Example Prediction

This example prediction makes it very clear that this model is not performing as well as the other pretrained models we looked at, while it's capable of picking up some small rocks unlike our custom model, the rocks are not completely identified like they are with the VGG or ResNet models.

6.2 Model Comparison

In order to get a complete picture of what model performed best, we looked at direct comparison of loss values during training across all 4 models.

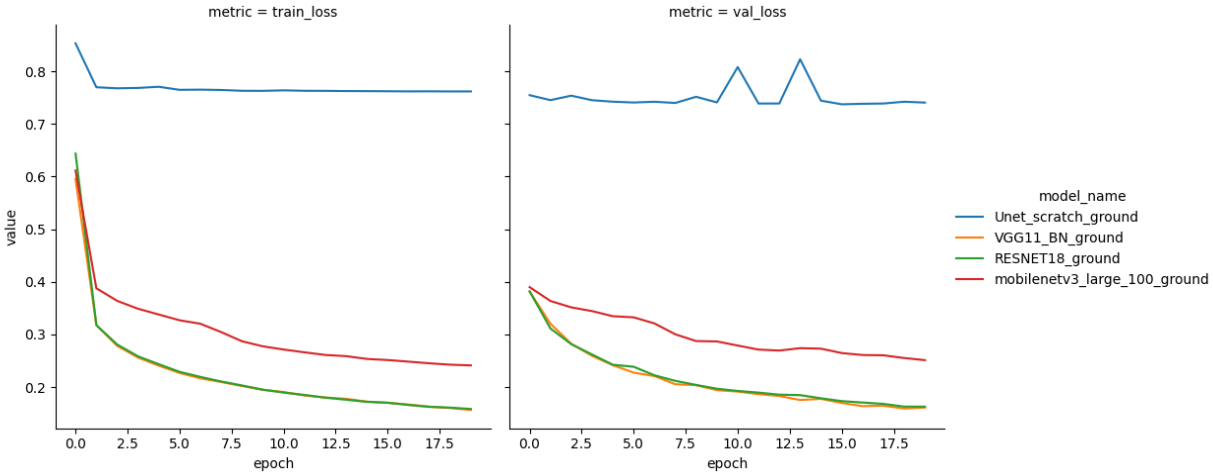

Figure 39: Model Training Comparison

In this view it's easy to see just how much more effective the pre-trained models were during training than our custom model. In addition, the VGG and ResNet models converged to a lower loss quicker than the large MobileNet model.

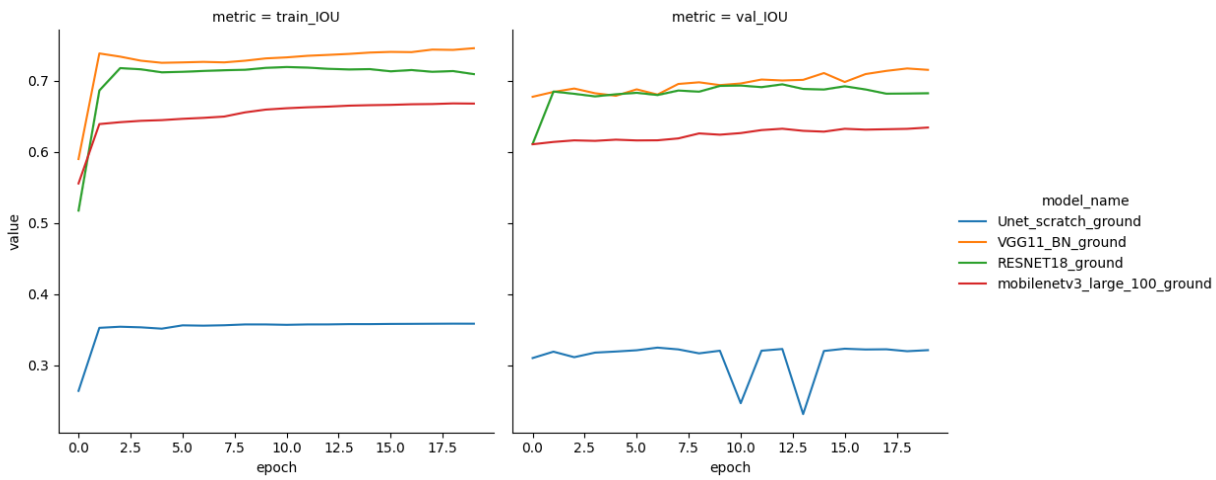

Figure 40: Model Metric Comparison

The view of our IoU metric over training continues to highlight the degree that the pre-trained models outperformed our custom model. It also highlights a slight difference between the ResNet model and VGG model towards the end of our 20 epoch training cycle. The ResNet model began to show a decrease in IoU values while the VGG and MobileNet values continued to rise.

When we plot the results of running these models on the testing dataset we see the same results. In order to confirm that the pre-trained models did in fact out perform our Custom U-Net we ran a Mann-Whitney U Test on the IoU values from our testing set. When comparing the output of the VGG model to our Custom model we did get a p value less than 0.05 (*p=6.6e-10*) rejecting the null hypothesis that the medians of the two distributions are the same. We had statistically significant results when comparing the outputs of all models except for the Custom and the MobileNet (*p=0.17*) we believe this is due to the fluctuations of both of these models.

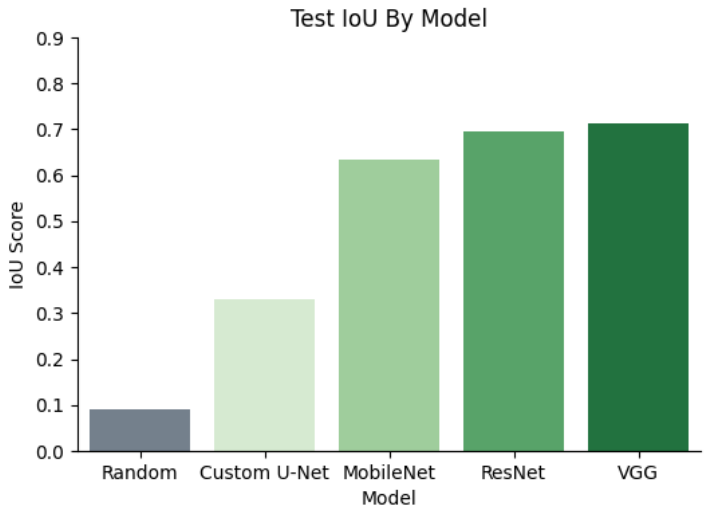

Figure 41: Model Testing Comparison

We also have access to a very small set of real lunar images, rather than the generated images that we have been training and testing on. We noticed that none of the high performing models created by other people had generated statistics for the testing dataset. When we ran all of our models on this dataset we got interesting results.

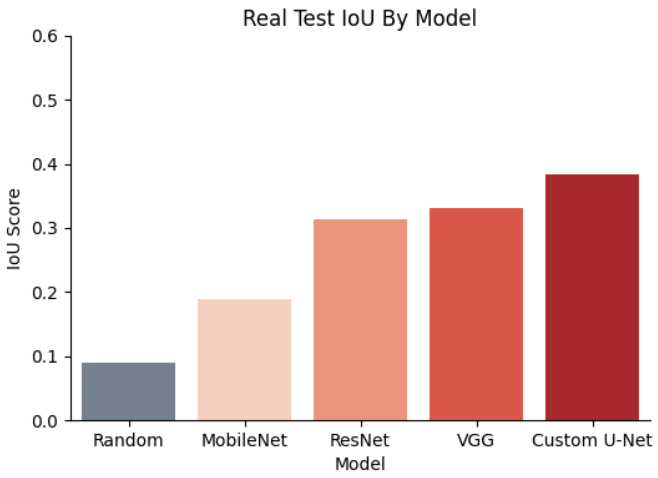

Figure 42: Model Comparison on Real Images

When tested on the real lunar images our model became just as accurate as the VGG and ResNet models, even out performing its own average IoU score on the rendered images. However, there were less than 40 real lunar images used for testing, so images that were difficult to segment had a much higher impact on the average IoU score for the model. For example, these were 2 true lunar image segmentation attempts from the ResNet model.

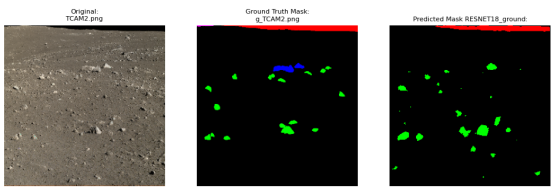

Figure 43: ResNet Prediction Example 1

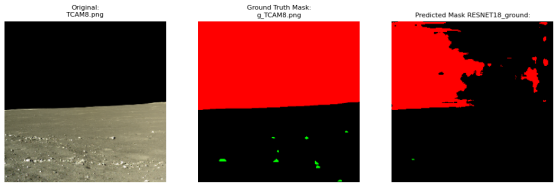

Figure 44: ResNet Prediction Example 2

It's clear from Fig. 43 that the ResNet model is still capable of identifying rocks in an image, however the occurrence of images on the right impacts its overall performance. We believe that the real lunar images under perform on all the pre-trained models because they look significantly different than the rendered images. In addition, it's clear from the image on the left that there are more rocks in the original image than are labeled. The ResNet model is identifying rocks that the ground truth mask is ignoring. This suggests that the model's IoU metric on the real lunar images is not necessarily indicative of the model's performance. We ran a Mann-Whitney U Test on the outputs of all the models once again and in this case the only two models that did not reject the null hypothesis ($p<0.05$) were the VGG and ResNet models.

6.3 Trained Model Distribution

Our team uploaded our trained models to a public Google Drive folder for simple access and download by others. We then created a function to automatically download the trained models into the project repository.

```
def download_trained_models(trained_models_url):
    '''
    Function to download trained models from Google Drive link
    '''
```

Python File	Description
`trained_model_dl.py`	`Script to download trained models from Google Drive link`

Figure 45: Scripts used for trained model download.

7. Conclusion

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.

As can be seen from the results section, the most promising model is the U-Net with the VGG11 pre-trained backbone. This remains consistent with what previous successful attempts at this project had been. This result makes sense due to the fact that the VGG11 model was the largest model that we used as a pre-trained backbone. This problem seemed to suffer more from underfitting than overfitting, so it makes sense that the most complex pre-trained model ended up performing the best on this dataset.

For our custom built U-Net we believe we could significantly improve accuracy by making the model more complex since it's currently underfitting to the data. This could be done by expanding the blocks that make up the encoder decoder structure to include more convolutional layers, or by increasing the number of blocks present in both sides of the U-Net.

We also believe that focusing the models on testing well on the true lunar images would be an important future direction. In order for this to be successful there would need to be a larger testing set of these images, and we believe that we would need to tailor our data augmentation to match those images rather than the rendered images.

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
15. Segmentation Models Pytorch Documetional
16. UNet Pytorch Implementation
17. Weight Initialization
18.

9. Appendix

9.1 Random Seed

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

9.2 Computer Listing

The Python packages that we used are in the *requirements.txt* file.

```
absl-py==1.2.0
aiohttp==3.8.1
aiosignal==1.2.0
arrow==1.2.2
asttokens==2.0.8
astunparse==1.6.3
async-timeout==4.0.2
attrs==19.3.0
Automat==0.8.0
aws-lambda-builders==1.18.0
aws-sam-translator==1.46.0
awscli==1.25.81
backcall==0.2.0
backports.zoneinfo==0.2.1
binaryornot==0.4.4
blessings==1.7
blinker==1.4
blis==0.7.8
boto3==1.24.46
botocore==1.27.80
cachetools==5.2.0
catalogue==2.0.8
certifi==2019.11.28
chardet==3.0.4
charset-normalizer==2.1.0
chevron==0.14.0
click==7.1.2
cloud-init==22.2
colorama==0.4.3
command-not-found==0.3
configobj==5.0.6
constantly==15.1.0
cookiecutter==2.1.1
```

```
cryptography==2.8
cupshelpers==1.0
cupy-cuda116==10.6.0
cycler==0.11.0
cymem==2.0.6
cytoolz==0.12.0
datasets==2.4.0
dateparser==1.1.1
dbus-python==1.2.16
debugpy==1.6.3
decorator==5.1.1
defer==1.0.6
dill==0.3.5.1
distro==1.4.0
distro-info===0.23ubuntu1
docker==4.2.2
docopt==0.6.2
docutils==0.16
dunamai==1.12.0
ec2-hibinit-agent==1.0.0
efficientnet-pytorch==0.7.1
en-core-web-sm @
https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-3.4.0/en_co
re_web_sm-3.4.0-py3-none-any.whl
entrypoints==0.3
et-xmlfile==1.1.0
executing==1.1.0
fastrlock==0.8
filelock==3.7.1
Flask==1.1.4
flatbuffers==1.12
fonttools==4.34.4
frozenlist==1.3.1
fsspec==2022.7.1
gast==0.4.0
gitdb==4.0.9
GitPython==3.1.27
google-auth==2.10.0
google-auth-oauthlib==0.4.6
google-pasta==0.2.0
gpustat==0.6.0
grpcio==1.47.0
```

```
h5py==3.7.0
httplib2==0.14.0
huggingface-hub==0.8.1
hyperlink==19.0.0
idna==2.8
importlib-metadata==4.12.0
incremental==16.10.1
install==1.3.5
ipykernel==6.17.1
ipython==8.5.0
iteration-utilities==0.11.0
itsdangerous==1.1.0
jedi==0.18.1
jellyfish==0.9.0
Jinja2==3.1.2
jinja2-ansible-filters==1.3.2
jinja2-time==0.2.0
jmespath==0.10.0
joblib==1.1.0
jsonpatch==1.22
jsonpickle==1.5.2
jsonpointer==2.0
jsonschema==3.2.0
jupyter_client==7.4.5
jupyter_core==5.0.0
kaggle==1.5.12
keras==2.9.0
Keras-Preprocessing==1.1.2
keyring==18.0.1
kiwisolver==1.4.4
langcodes==3.3.0
language-selector==0.1
launchpadlib==1.10.13
lazr.restfulclient==0.14.2
lazr.uri==1.0.3
leveldb==0.201
libclang==14.0.6
linecache2==1.0.0
loguru==0.6.0
macaroonbakery==1.3.1
Markdown==3.4.1
MarkupSafe==2.0.1
```

```
matplotlib==3.5.2
matplotlib-inline==0.1.6
mlxtend==0.21.0
more-itertools==4.2.0
mpmath==1.2.1
multidict==6.0.2
multiprocess==0.70.13
munch==2.5.0
murmurhash==1.0.7
nest-asyncio==1.5.6
netifaces==0.10.4
networkx==2.8.5
nltk==3.7
numpy==1.23.1
nvidia-ml-py3==7.352.0
oauthlib==3.1.0
opencv-python==4.6.0.66
openpyxl==3.0.10
opt-einsum==3.3.0
packaging==21.3
pandas==1.4.3
parso==0.8.3
pathspec==0.9.0
pathy==0.6.2
pbr==5.4.5
pexpect==4.6.0
pickleshare==0.7.5
Pillow==9.2.0
platformdirs==2.5.3
plumbum==1.7.2
preshed==3.0.6
pretrainedmodels==0.7.4
prompt-toolkit==3.0.30
protobuf==3.19.4
psutil==5.9.1
pure-eval==0.2.2
py-cpuinfo==8.0.0
pyarrow==9.0.0
pyasn1==0.4.2
pyasn1-modules==0.2.1
pycairo==1.16.2
pycups==1.9.73
```

```
pydantic==1.9.1
pydotplus==2.0.2
Pygments==2.12.0
PyGObject==3.36.0
PyHamcrest==1.9.0
PyJWT==1.7.1
pymacaroons==0.13.0
pymongo==4.2.0
PyNaCl==1.3.0
pyOpenSSL==19.0.0
pyparsing==3.0.9
pyphen==0.12.0
pyRFC3339==1.1
pyrsistent==0.15.5
pyserial==3.4
pyspellchecker==0.6.3
python-apt==2.0.0+ubuntu0.20.4.7
python-dateutil==2.8.2
python-debian===0.1.36ubuntu1
python-slugify==6.1.2
pytz==2022.1
PyYAML==5.3.1
pyyaml-include==1.3
pyzmq==24.0.1
questionary==1.10.0
regex==2021.9.30
requests==2.25.1
requests-oauthlib==1.3.1
requests-unixsocket==0.2.0
responses==0.18.0
rsa==4.7.2
s3transfer==0.6.0
sacred==0.8.2
scikit-learn==1.1.2
scipy==1.9.0
screen-resolution-extra==0.0.0
seaborn==0.11.2
SecretStorage==2.3.1
segmentation-models-pytorch @
git+https://github.com/qubvel/segmentation_models.pytorch@fdb024ba33753e127dce07662dc1
4cce27fd5bd3
serverlessrepo==0.1.10
```

```
service-identity==18.1.0
simplejson==3.16.0
six==1.14.0
smart-open==5.2.1
smmap==5.0.0
sos==4.3
spacy==3.4.1
spacy-legacy==3.0.9
spacy-loggers==1.0.3
srsly==2.4.4
ssh-import-id==5.10
stack-data==0.5.1
sympy==1.10.1
systemd-python==234
tensorboard==2.9.1
tensorboard-data-server==0.6.1
tensorboard-plugin-wit==1.8.1
tensorflow-estimator==2.9.0
tensorflow-gpu==2.9.1
tensorflow-io-gcs-filesystem==0.26.0
termcolor==1.1.0
testresources==2.0.0
text-unidecode==1.3
textacy==0.12.0
thinc==8.1.0
threadpoolctl==3.1.0
timm==0.4.12
tokenizers==0.12.1
tomlkit==0.7.2
toolz==0.12.0
torch==1.12.1
torchmetrics==0.10.3
torchtext==0.13.1
torchvision==0.13.1
tornado==6.2
tqdm==4.64.0
traceback2==1.4.0
traitlets==5.4.0
transformers==4.21.1
Twisted==18.9.0
typer==0.4.2
typing-extensions==3.10.0.0
```

```
tzlocal==3.0
ubuntu-advantage-tools==27.8
ufw==0.36
unattended-upgrades==0.1
unittest2==1.1.0
urllib3==1.26.11
wadllib==1.3.3
wasabi==0.10.1
watchdog==2.1.2
wcwidth==0.2.5
websocket-client==1.3.3
Werkzeug==1.0.1
wget==3.2
wrapt==1.14.1
xkit==0.0.0
xxhash==3.0.0
yarl==1.8.1
zipp==1.0.0
zope.interface==4.7.1
```