Lunar Surface Image Segmentation: Individual Report

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

Table of Contents

1.0 Introduction

Our project was based on a dataset of images of the surface of the moon. Space travel is an incredibly precise, difficult, and most of all expensive, endeavor. Going to the moon, or any other planet, requires finding an optimal location to land whatever craft is being flown there. In order to do this, rock and solid ground need to be identified. While we can imagine this is mostly a job done by people, a remote operated or autopilot craft needs to be able to identify rocks and other obstacles in its way when scanning the surface of the moon.

This is one of many use cases of an algorithm that can identify rocks from a landscape. It can be applied to geology research, undersea machines, or drones on Earth. Given this problem statement, we set out to develop a deep learning algorithm to segment an image of the surface of the moon and identify large rocks, small rocks, and the sky.

In order to solve this problem we knew we needed two main components, the model, and the data loader. We split the work up mainly along this line with Josh working on the data loader and me working on the modeling aspect. We also split the smaller tasks amongst each other. I worked a lot on the EDA and the main script and smaller functions like the TrainTestSplit, while Josh worked on all the data downloading and model storage interaction with our code. It was, I believe, an even split of the work. We also each interacted with the other aspects. Josh helped with testing some of the models and I worked with editing part of the data loaders when necessary so we both interacted with all parts of the project.

1.1 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.

1.3 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.

2. Individual Work

2.1 Introduction

Like mentioned previously. My main contribution to the project was model design and training. The development of this part of the project proved to be very challenging. I started with trying to simply get a working model that could train and output a segmentation mask. This model was incredibly simple and the structure was taken mostly by combining online sources. Once I had a working model that I understood I was able to go back and refine the structure and individual aspects. I tuned the parameters until it stopped improving and I was happy with the design.

I then moved onto the pre-trained model design. I tried for many days to write a pre-trained backbone U-Net from scratch before discovering the python package that returned pre-designed U-Nets with pre-trained backbones. I was able to get those models to train and then began the process of adjusting the hyperparameters until I was happy with the design. Following this I was able to run the full training code and visualize the results.

2.2 Custom Model Structure

We began our modeling by creating a custom U-Net model (Fig. 1). 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 was made up of a single 2-dimensional convolutional layer with a *softmax* activation function.

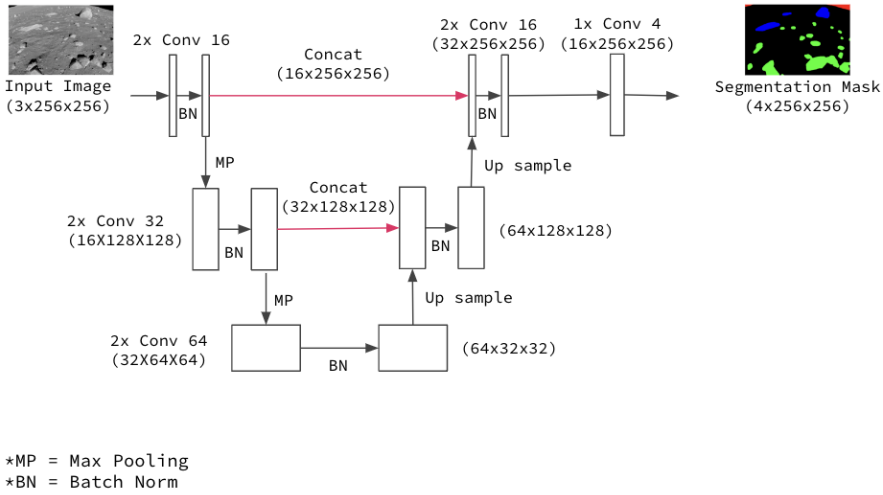

Figure 1: Outline of Custom U-Net model

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 2: Parameter Count by Model

2.3 Metrics and Algorithm Choices

Deciding the metrics and algorithms to use was done in part through research and in part to trial and error hyper parameter tuning.

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).

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.

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]

2.3 Weights and Hyperparameters

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 layers 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

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 3: 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 4: 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.

2.4 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 5: 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 6: 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 7: 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.

2.5 Other Contributions

Any other functions not explicitly discussed previously that I worked on are as follows, however a majority of my work has already been discussed.

Python File	Description
EDA.py	Ran EDA code and saved files so the EDA notebook could be executed
EDA_figures.ipynb	Jupyter Notebook to visualize data before modeling
TrainTestSplit.py	Split data into necessary groups and copied them into necessary folders for modeling
utils.py	Utility functions for testing and debugging

Figure 8: Auxiliary Scripts

3. Results

The results of our project can be divided into 2 sections. The results from our custom U-Net and the results from our pre-trained models. These results are also compared against each other to come to a conclusion about the ideal model to choose for the segmentation problem we are trying to solve.

3.1 Custom U-Net

When looking at any of our models I first plotted the training loss and metrics to get an idea of how the model performed over time.

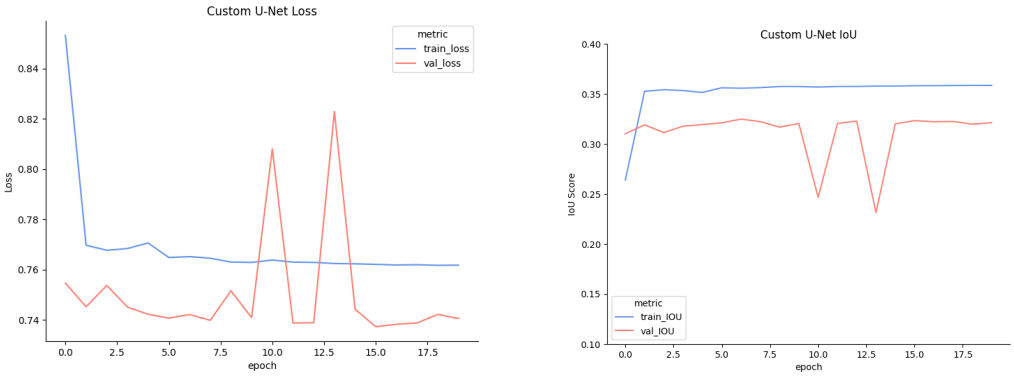

Figure 9: Custom U-Net Training

It's immediately clear from Fig. 9 that the custom U-Net that we developed did not train effectively. The validation loss did not drop significantly after the first few epochs, suggesting that the model is underfitting the dataset. We believe this is because the model we created was not complex enough to accurately solve the segmentation problem we had to solve.We can look at example predictions to get an idea of what's occurring in the model.

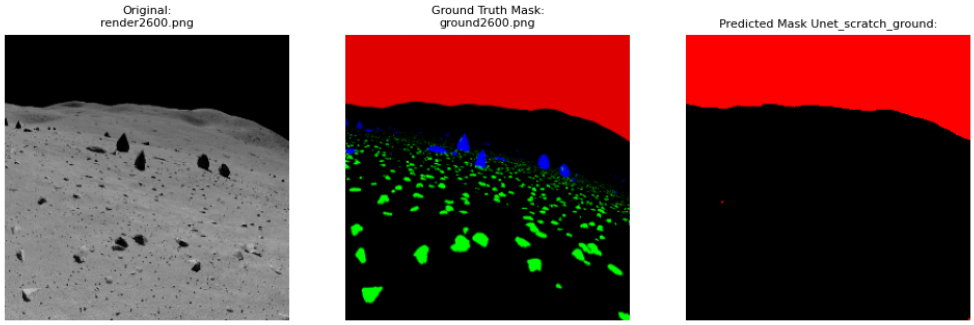

Figure 10: Custom U-Net Prediction

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.

3.2 Pre-trained Models

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.

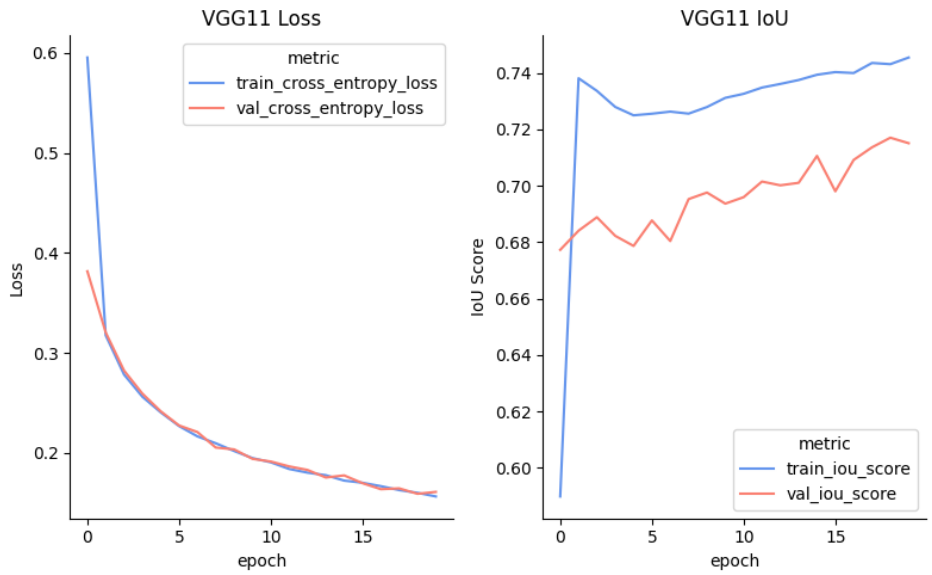

Figure 11: VGG11 Training History

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

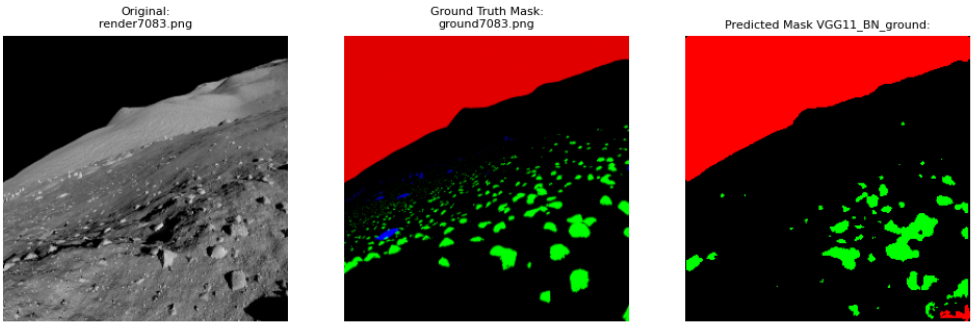

Figure 12: VGG11 Example Prediction

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

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

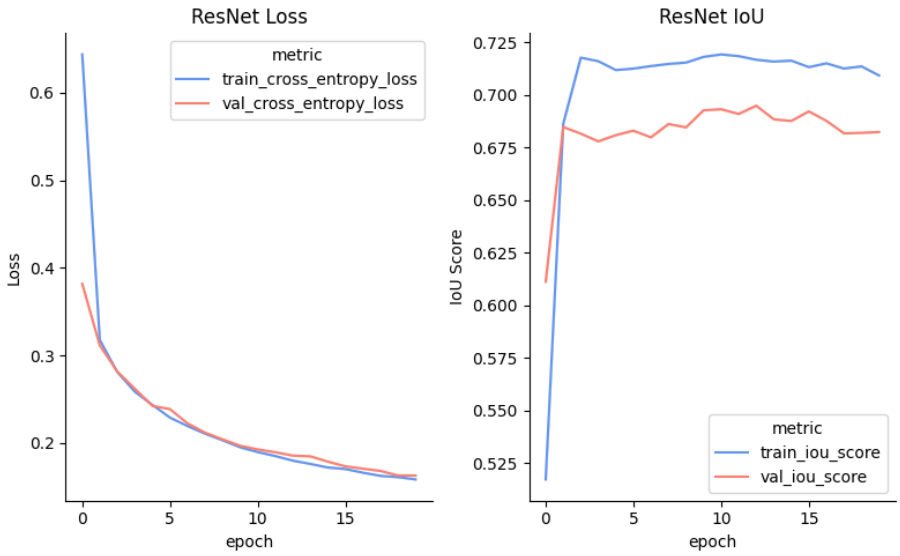

Figure 13: ResNet18 Training History

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

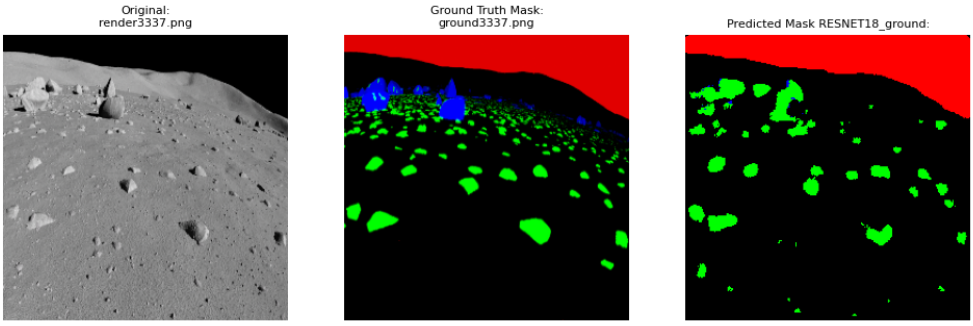

Figure 14: ResNet18 Example Prediction

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.

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

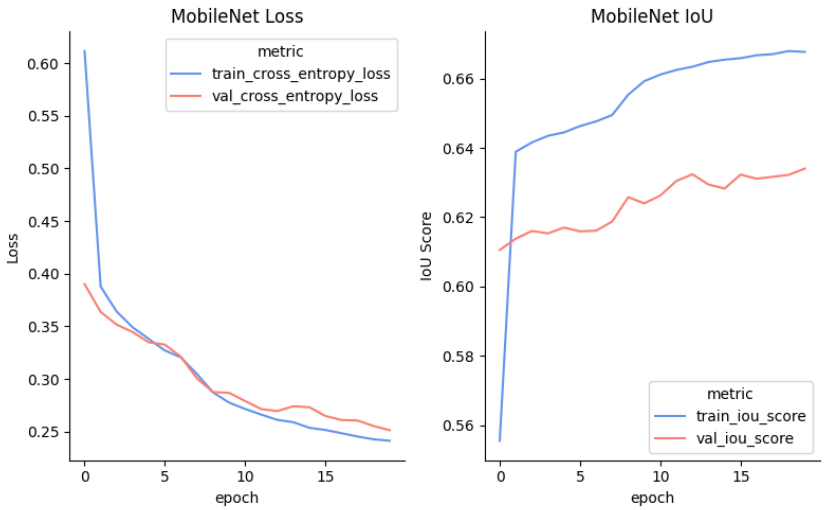

Figure 15: MobileNetv3 Training History

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

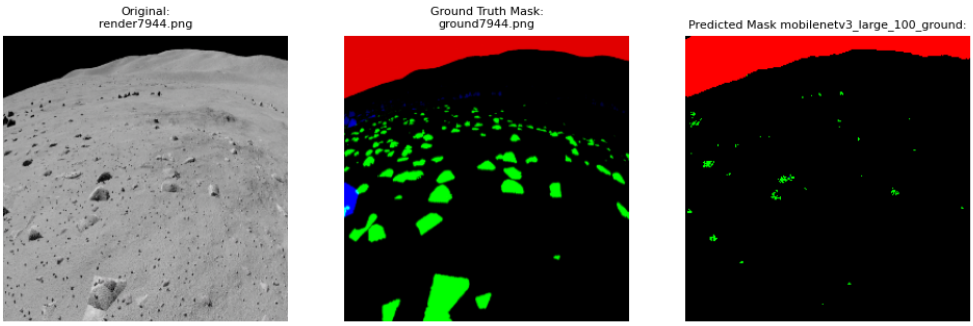

Figure 16: MobileNetv3 Example Prediction

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.

3.3 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.

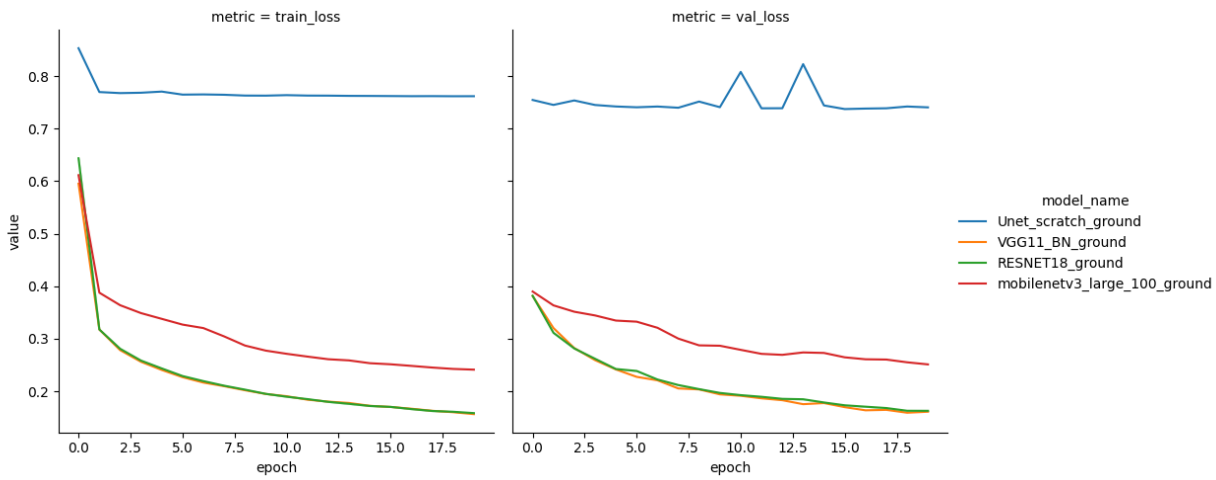

Figure 17: Model Training Comparison

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.

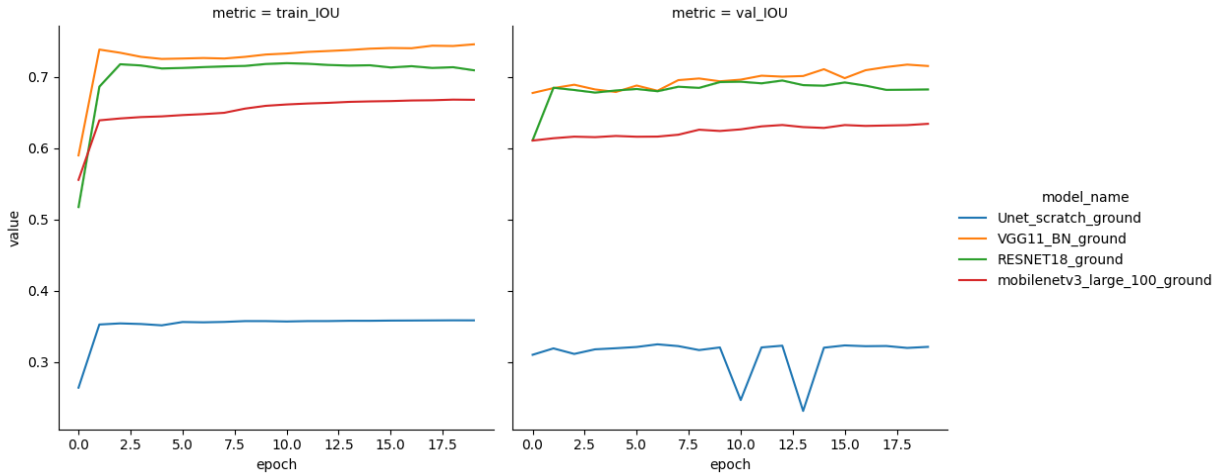

Figure 18: Model Metric Comparison

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 the running these models on the testing dataset we see the same results.

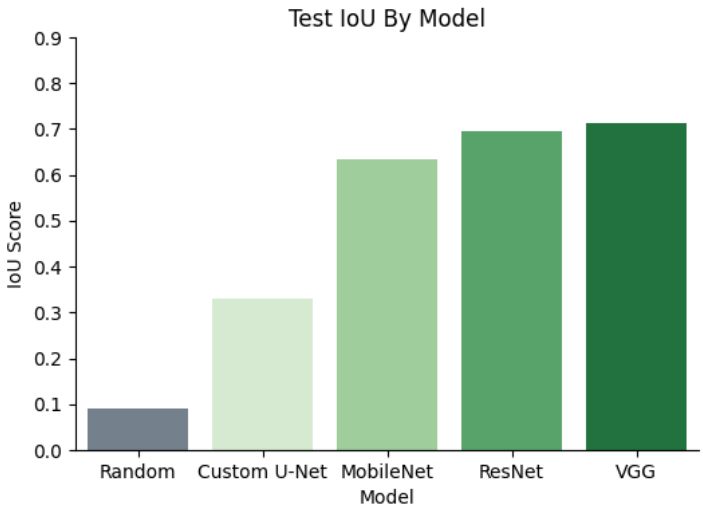

Figure 19: Model Testing Comparison

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.

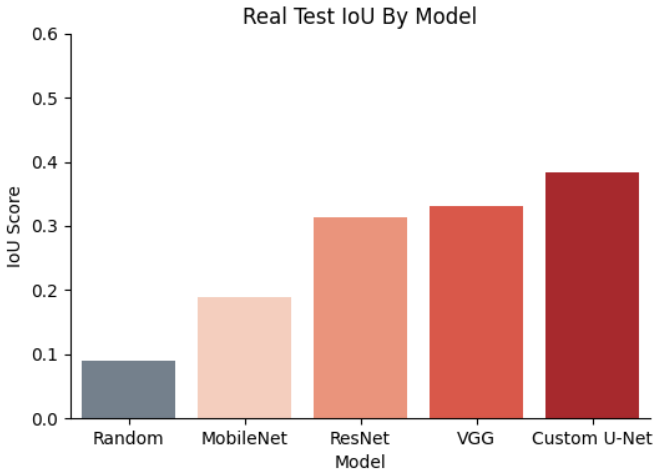

Figure 20: Model Comparison on Real Images

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.

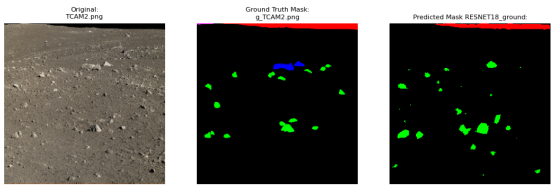

Figure 21: ResNet Prediction Example 1

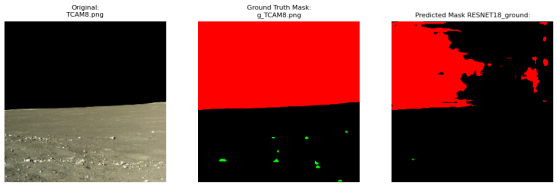

Figure 22: ResNet Prediction Example 2

It's clear from Fig. 21 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.

4. Conclusions

Our group succeeded in our goal of creating a custom U-Net and comparing it to multiple different types of pre-trained U-Net backbones. We were able to successfully segment the images with up to ~70% accuracy using an Intersection over Union metric. Our custom U-Net model did not approach the pre-trained model accuracy, or out perform any of the previous attempts to create a custom designed U-Net to segment this dataset. The issue that we ran into with the model only identifying the sky was a common one that we were not able to surpass. We believe that this issue stems from our model's simplicity. While we were preparing for a model that overfitted the dataset, we ended up with a model that severely underfitted the dataset. In the future we believe that creating a more complex model would improve its accuracy and hopefully overcome the challenge of only identifying the ground and sky.

Our pre-trained models performed similarly to previous attempts on this dataset and we determined that the VGG11 model was the best model to use on this dataset. We believe this is due to the fact that it was the largest model we trained. Given this, in the future using an even larger model could prove to be more successful. In the end our results confirmed what other attempts had shown, given that the most successful attempts also used a VGG model to segment this data.

Furthermore, running a full hyperparameter search in order to tune the best model (VGG11) we believe also could have provided a lot more accuracy. Given our time constraints we weren't able to run a full search and this did impact our ability to choose hyperparameters.

We both learned a lot about the process of image segmentation, the creation and design of U-Nets and computer vision projects in general. I also learned a lot about structuring training loops and creating efficient class structures to make training go as smoothly as possible. In the future I think if I was going to focus on a specific pre-trained backbone I would like to create my

own model class rather than use a python package for the base model in order to make the training loop fully customizable.

5. Code Percentage

In order to get an initial functioning U-Net training I copied 60 lines of code from the basic UNet structure found here. However, once I understood how it worked I completely re-wrote it so it made more sense for me to read and so I could customize it efficiently. However the basic structure of an Encoder class and a Decoder class and specifically the main function of the U-Net class was modeled off of that code. I also used 4 lines that I found here in order to do weight initialization which I used twice for a total of 8 lines.

All in all I wrote a total of 1450 lines of code. Taking into account the 60 lines from the initial structure and counting that I rewrote 50 of those lines, and the 8 that I used for the weight initialization my final code percentage that I got from the internet comes out to:

$$\%\ copied\ =\ \frac{60-50+8}{1450}\times\ 100\ =\ 1.2\%\ copied\ code$$

Equation 4: Model Comparison on Real Images

6. 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. https://segmentation-modelspytorch.readthedocs.io/en/latest/index.html
16. https://amaarora.github.io/2020/09/13/unet.html
17. Weight Initialization