U-Net Machine Learning Model to Analyze Ice Formations on Pluto's Sputnik Planitia

Brooke Carmody
Instructor: Dr. Alan Howard
Department of Astronomy
University of Virginia

This thesis is submitted in partial completion of the requirements of the BA Astronomy Major
May 10th, 2024

U-Net Machine Learning Model to Analyze Ice Formations on Pluto's Sputnik Planitia

Abstract

This thesis details the process of building a machine learning model that classifies and labels the ice sublimation pits across Pluto's Sputnik Planitia. Using primarily generated data, I worked to build and train a U-Net model to run image analysis on these ice pits to be able to identify the pit shape and the center of the pit automatically. The U-Net, a variation on the Convolutional Neural Network, is a powerful architecture for working with low-dimensional data and with smaller data sets. [RFB15] U-Nets are both time-efficient and power-efficient models, even when trained on largely computer-generated or augmented datasets, making it the ideal architecture for building a model to analyze Pluto. By building on the data augmentation code from my advisor, Dr. Alan Howard, I generated sufficient ice pit images to train and tune the U-Net to be able to identify and label ice pits in new images.

1 Introduction

1.1 Geomorphology of Sputnik Planitia

Sputnik Planitia is a large, ice-covered basin with a high albedo, mostly located in Pluto's northern hemisphere but extending at least a little bit south of the equator. The planitia stretches approximately 1050 kilometers by 800 kilometers, making it the largest known glacier in the solar system. The geomorphology of Sputnik Planitia is absolutely fascinating and much more varied and active than could have been expected before the New Horizons Pluto flyby in 2015 [Laknd]. The basin is primarily composed of nitrogen ice, with smaller amounts of methane ice and carbon monoxide ice. [Moo+17] There is a distinct lack of impact craters visible, indicating the surface is young by planetary standards. Given the resolution limits of approximately 400 meters per pixel and the calculated probability of smaller Kuiper Belt Objects impacting Pluto, estimates put the surface of Sputnik Planitia at 10 million years or younger. [Tri16] Despite the lack of impact cratering, there are still a number of impressive features across the basin that make it so interesting to observe and analyze.

There are "cells" across Sputnik Planitia indicative of convection within the nitrogen ice that makes up the surface [BI18]. These convection cells are polygonal in shape, range in size from 10 to 40 kilometers across, and are likely several kilometers thick at their centers [McK+16]. They are formed as the under layers of ice are warmed by Pluto's interior heat, the warmed ice wells up in the center of the cells and pushes the previous top layer out towards the edges of the cells, and then at the edge of the cells the ice sinks into the margins to be recycled back into the convective process. The flow rate is estimated to be 7 cm per year, which is a similar scale to tectonic plate movement on Earth. There is also strong evidence for faster ice flows into Sputnik Planitia, including apparent glaciers flowing into the planitia from the adjacent highlands. [Umu+17] Along the eastern edge of the basin, dark features and patterns in the ice indicate that the N2 glacial features in the region are moving from the pitted highlands of the Tombaugh Regio into Sputnik Planitia. The basin is bordered by a series of mountain ranges along the western rim. These regions are chaotic and blocky, and are significantly elevated above the rest of the planitia, reaching up to 5 kilometers above the surrounding terrain. The al-Idrisi Montes marks the northernmost mountain range, the Zheng-He Montes and Bare Montes are the central ranges, and the Hillary Montes and Norgay Montes to the southwest of the basin mark the southernmost of this type of feature.

The feature that this thesis is investigating is the network of ice pits present across Sputnik Planitia [How21]. These pits are most frequent in the margins between the convection cells, and are currently believed to result from fracturing and sublimation of the nitrogen ice making up the majority of the basin. Some models of the evolution of the sublimation pits put the estimated surface age of Sputnik Planitia at approximately 180,000 years. The sublimation pits are relatively straightforward to identify with the naked eye as they cluster between convection cells and have a consistent shadow pattern due to their smaller open-

ing relative to their depth. Although identifying and classifying these pits with a computer model takes far more time to develop than simply using the naked eye, once the model is trained and ready to be deployed, it can easily be used on any new data we may acquire from Pluto from future missions and will be significantly faster than identifying the pits by hand.

1.2 Machine Learning Models for Image Analysis

There are a number of machine learning algorithms in use today that could be effective at analyzing planetary surface features. Support Vector Machines, or SVMs, have proven to be powerful models for doing object-based image analysis (OBIA) and classification tasks [TA08]. SVMs work by finding the hyperplane that best separates different classes in the feature space. In data with more dimensions, SVMs use designated kernel functions to transform the data into a higher-dimensional space where it can be more easily separated by a single hyperplane. SVMs do not require much computational power and are effective in high-dimensional space. Additionally, due to the range of kernel functions that work with SVMs, they are rather robust and versatile. That said, they are limited in that SVMs struggle with noisy data and require careful choosing and tuning of the kernel function and regularization parameters. SVMs also inherently do a binary classification, effectively sorting data to one side of the hyperplane or the other. This makes them a weaker model on complex, multi-class data. The images available of the ice pits on Pluto are relatively simple data and the total dataset is not that large, meaning SVMs could work well. However, because SVMs only do binary classification, they are likely not flexible enough for this type of project in the long run.

Another model that shows promise for planetary image analysis is a Convolutional Neural Network, or CNN. CNNs are deep learning algorithms used for data processing in a grid-like pattern, making them especially well suited for image analysis[LSD15]. A Convolutional Neural Network consists of convolutional layers, often accompanied by pooling layers, fully connected layers, and normalization layers. The convolutional layers apply a series of learnable filters to the input, helping the network identify various features such as edges, textures, or objects. CNNs are relatively easy to start building due to their widespread use in all computer-based fields resulting in a large number of Python libraries and packages. They are incredibly useful in doing image analysis for their ability to automatically learn spatial hierarchies of features from images and for their efficiency in dealing with high computational complexity due to shared weights in convolutional layers and downsampling in pooling layers. However, CNNs require a large amount of labeled training data and have a tendency to overfit on smaller datasets, making it difficult to finish building a working CNN model in a field without much data. Due to the lack of high-resolution images of Sputnik Planitia, a CNN model would be difficult to train without serious data augmentation. CNNs are also prone to requiring serious computational power, often outside the scale of a single laptop or PC, which can make them costly and inefficient for smaller projects like this.

A potential variation on the Neural Network architecture is the transformer, a model originally developed for dealing with sequential data like natural language data but recently adapted for image analysis [Vas+17]. The core of the transformer model is the attention mechanism, which allows the model to weight the input data with different levels of importance and then focus on the data most relevant to the task at hand. In a Vision Transformer, or ViT, an image is divided into patches; the patches are then flattened and linearly embedded, with position embeddings added to maintain positional information [Dos+21]. Transformers are really good at capturing global contexts and long-range dependencies, which would be beneficial for analyzing the ice of Sputnik Planitia. However, like many more powerful models, transformers are incredibly computationally expensive and often require huge amounts of training data, which unfortunately does not yet exist for Pluto. Despite these limitations, it is important to note that transformers are growing in popularity in image analysis due to being so effective at analyzing intricate patterns and relationships in data.

Another variation on a Convolutional Neural Network is the U-Net architecture [RFB15]. U-Nets were designed for medical image segmentation but extrapolate well to other types of data. The U-Net architecture is often represented as a symmetric "U" made up of an encoder on the left that captures context and a decoder on the right that enables precise localization. The encoder extracts features at a series of contracting scales, and then the decoder constructs the output images from the extracted features through a series of expanding layers. U-Nets also take advantage of skip connections during the decoding phase, combining low-level feature maps with higher-level feature maps to help with precise localization. U-Nets are powerful in that they are efficient relative to the low number of required input parameters and they work very well on smaller datasets like those of Pluto. With that in mind, U-Nets struggle with large images and are very tuned in to the quality of the training data. Even the barest hint of bad data in the training set is enough to mess up a U-Net.This causes issues with larger models, but given the artificial nature of the dataset I am training my model on, this is a low risk relative to the pros of the U-Net architecture.

2 Methodology

2.1 Data Generation and Pre-Processing

The first step in building any machine learning model is to acquire and prepare the data. Due to the scope of this project and the lack of available data from Sputnik Planitia, much of this first step involved data augmentation. I accomplished this by building from base code provided by my advisor to write the *make_hole* function. This function creates a TIFF file and a PGM file to simulate a simplified singular ice pit like those found on Sputnik Planitia by creating a digital elevation model of a hole, calculating slope and aspect, then using these to calculate the shading of each pixel. This function only creates one pit per

file, but is very flexible and allows for many different shapes and sizes of these ice pits. The variables *mx* and *my* help determine where the pit is centered on each image relative to the bottom left corner, *max_slope_deg* determines the slope of the pit walls, and *radius_x* and *radius_y* determine the x- and y-radius of the pit such that differing values for these radii can make the pits very circular or very oblong. The most important variables for simulating the real-world conditions of Pluto's surface are *sun_elevation_deg* and *sun_azimuth_deg*; these set the elevation and azimuth, respectively, of the light source illuminating the pit image just as the sun does on Pluto. These are helpful for humans to better visualize how these pits would look in real life, and they are helpful for the model to better understand how the patterns of light and shadows within the pits interact to create the shape of the whole pit.

Once the pit image has been generated, the program then labels the image with the relevant information. With these pits I was only looking for two structures per image, as with the more simple data there is not much need to look deeper for the model to train correctly. The first structure of note is the pit itself and the second is the center of the pit. Relative to the surrounding flat area in the generated images, the pit can be considered the region of the image with a gradient magnitude higher than the threshold hyperparameter. To find the gradient magnitude for each pixel in the image, I calculated the forward differences of both x and y to find the gradients along x and along y, and then used these to calculate the total gradient of each pixel.

$$\vec{\nabla} image_{i,j} = (\text{gradient}_x, \text{gradient}_y)$$

where

$$\text{gradient}_x = \begin{cases} image_{i,j} - image_{i,j-1} & \text{for } j = 1, 2, \ldots, \text{cols} - 1 \\ image_{i,0} - image_{i,\text{cols}-1} & \text{for } j = 0 \end{cases}$$

and

$$\text{gradient}_y = \begin{cases} image_{i,j} - image_{i-1,j} & \text{for } i = 1, 2, \ldots, \text{rows} - 1 \\ image_{0,j} - image_{\text{rows}-1,j} & \text{for } i = 0 \end{cases}$$

To find the gradient magnitude for each pixel, I found the square root of the sum of the squares of both the x and y gradients.

$$magnitude_{i,j} = \sqrt{(\text{gradient}_x)^2 + (\text{gradient}_y)^2}$$

Once I had the gradient and magnitude, I then tuned the threshold hyperparameter until the *find_crater* function found the whole crater and nothing more or less.

To find the center of the crater, I wrote the find_center function to find the point in the image with the minimum divergence. The center of the pit would be the lowest elevation in the image and thus the divergence at that point would be near zero and significantly lower than elsewhere on the image. I calculated

the divergence using the forward difference of the gradient rather than of the image as I had previously done when finding the gradient.

$$\text{divergence}_{i,j} = -\left(\text{forward difference in x of gradient}_x\right) - \left(\text{forward difference in y of gradient}_y\right)$$

which simplifies to:

$$\text{divergence}_{i,j} = \text{div}_x + \text{div}_y$$

Once I had found the divergence for each pixel, I used the unravel_index function in NumPy to convert the array of divergence values into a tuple of x/y coordinate arrays and then sort by value to return only the lowest divergence value.

With the pit images generated and the pit center and pit itself identified, I used Matplotlib to visualize three graphics: the elevation, the shading based on slope and light source position, and the crater and center labeling. The image plots act as confirmation that the pits are being generated and labeled correctly; the shaded image, Figure 1, and the label image, Figure 2, are then saved as both TIFF and PGM files to two separate directories, one for the image and one for the labels. Both files have the same name as that makes it significantly easier to pair them to each other when actually training the model. To generate enough images for a training dataset, I then ran make_hole an arbitrary number of times with each of the variables set to randomize within a certain range. This created almost 400 unique pit images with corresponding labels to work with. For the sake of consistency and convenience, I worked almost exclusively with the TIFF versions of the generated data.

The next step was to normalize and standardize all the TIFF files. Normalizing the data involves scaling all the pixel values down in each image to fall between 0 and 1 inclusive, and standardizing the data involves making all the images the same dimensions by augmenting the smaller images with the same neutral values already present at the edges of their axes.

$$\text{normalized}_{i,j} = \frac{\text{img\_gray}_{i,j} - \min(\text{img\_gray})}{\max(\text{img\_gray}) - \min(\text{img\_gray})}$$

The normalized image data were then saved to a new directory of normalized images to pull from when implementing the model.

From there, the final step before training and tuning the model is to split the data into the training, validation, and testing datasets. It's relatively standard practice to assign 80% of the data to training and 10% to validation, and the 10% is to be used for further testing once the model is appropriately tuned. To ensure further randomization, I used the NumPy random number generator in the split_tiff_dataset function to randomly shuffle the image into a list and assign the first 80% of the image data to be used for training. I then split the remaining data by assigning the next 10% to the validation dataset and the final 10% to the testing dataset. With the data processed and split, I then moved into the model itself.

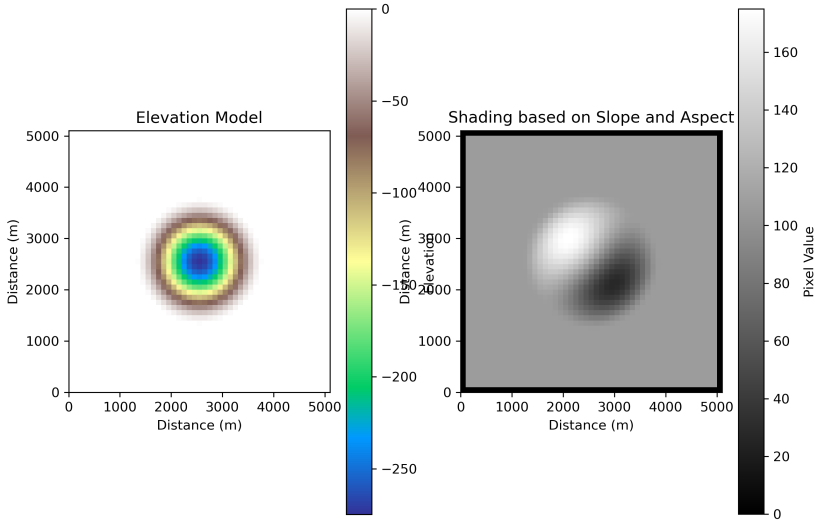

Figure 1: Generated ice pit with 20° slope, sun elevation at 25°, sun azimuth at 45°, and x- and y-radius equal at 1250 pixels. Left shows the elevation and right shows shading relative to the sun.

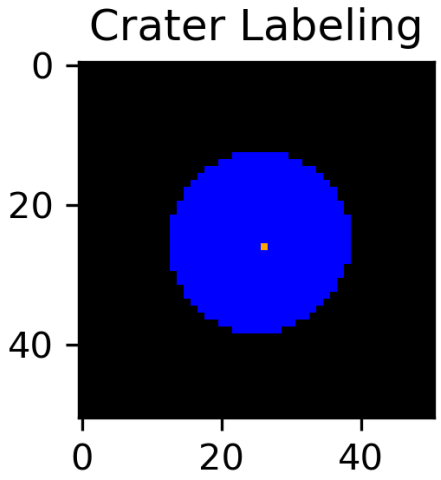

Figure 2: Ice pit label. Blue shows the extent of the pit, orange is the center of the pit.

2.2 U-Net Architecture

The U-Net as an architecture builds upon the Fully Conventional Network work of Long, Shelhamer, and Darrell who adapted classification networks available at the time such as AlexNet and GoogLeNet into fully convolutional networks and then transferred their learned representations by fine-tuning to the segmentation task. Ultimately, they were able to produce accurate and detailed segmentations by combining semantic information from deep layers with appearance information from shallow layers [LSD15]. The U-Net was then created by Ronneberger, Fischer, and Brox who were able to present a network and training strategy that relied on the strong use of data augmentation to use the available annotated samples more efficiently. The architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization all done in successive layers [RFB15]. This architecture has been used in many medical segmentation contexts since it's inception but as shown in this thesis, its potential uses are widespread and varied.

3 Results and Reflection

The most time-consuming part of this sort of modeling is normally pre-processing the data, and my U-Net model was no exception. Due to the lack of Pluto data in general and, more specifically, the lack of well-sorted and well-labeled data for Sputnik Planitia's ice pits, I spent most of my programming time trying to generate and label clean, noise-free data that wouldn't skew my model during training and didn't spend as much time as I would have liked on coding the actual model itself. The U-Net model runs and trains successfully, but I have yet to achieve the level of accuracy and success I was aiming for. Some of this may be in how I set up the convolutional layers and I strongly suspect some of it is in not having tuned the hyperparameters to their optimal states yet. I would have liked to have had more time to focus on this project this year and I expect I will continue to work on my U-Net model for a little while longer, at least until I have reached a level of accuracy I find more satisfactory.

References

[TA08] A. Tzotsos and D. Argialas. "Support Vector Machine Classification for Object-Based Image Analysis". In: *Object-Based Image Analysis: Spatial Concepts for Knowledge-Driven Remote Sensing Applications*. Springer, 2008, pp. 663–677. DOI: 10.1007/978-3-540-77058-9_36.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *arXiv:1411.4038* (2015).

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *arXiv:1505.04597* (2015).

[McK+16] W. B. McKinnon et al. "Convection in a volatile nitrogen-ice-rich layer drives Pluto's geological vigour". In: *Nature* 534.7605 (2016), Article 7605. DOI: 10.1038/nature18289.

[Tri16] D. E. Trilling. "The Surface Age of Sputnik Planum, Pluto, Must Be Less than 10 Million Years". In: *PLoS ONE* 11.1 (2016), e0147386. DOI: 10.1371/journal.pone.0147386.

[Moo+17] J. M. Moore et al. "Sublimation as a landform-shaping process on Pluto". In: *Icarus* 287 (2017), pp. 320–333. DOI: 10.1016/j.icarus.2016.08.025.

[Umu+17] O. M. Umurhan et al. "Modeling glacial flow on and onto Pluto's Sputnik Planitia". In: *Icarus* 287 (2017), pp. 301–319. DOI: 10.1016/j.icarus.2017.01.017.

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". In: *arXiv:1706.03762* (2017).

[BI18] P. B. Buhler and A. P. Ingersoll. "Sublimation pit distribution indicates convection cell surface velocities of 10 cm per year in Sputnik Planitia, Pluto". In: *Icarus* 300 (2018), pp. 327–340. DOI: 10.1016/j.icarus.2017.09.018.

[Dos+21] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *arXiv:2010.11929* (2021).

[How21] A. D. Howard. "Pervasive Ice-Related Erosion of Mid-Latitude Martian Craters". In: 1149 (2021).

[Laknd] E. Lakdawalla. *Pluto updates from AGU and DPS: Pretty pictures from a confusing world*. The Planetary Society. Retrieved January 30, 2024, from https://www.planetary.org/articles/12211538-pluto-updates-from-agu. nd.

For Creating Simple Training Data

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os
import random
```

In [ ]:
```python
def save_pgm(image, filename, max_val=255):
    """
    Save an array to a PGM file.

    Parameters:
    - image: 2D numpy array of pixel values.
    - filename: String, the name of the file to save the image to.
    - max_val: Integer, the maximum pixel value (typically 255 for 8-bit images).
    """
    # Ensure the pixel values are within the correct range
    image = np.clip(image, 0, max_val)

    # Open the file for writing
    with open(filename, 'wb') as f:
        # Write the PGM header
        f.write(bytes(f"P5\n{image.shape[1]} {image.shape[0]}\n{max_val}\n", 'ascii'))

        # Write the pixel data
        if image.dtype != np.uint8:
            image = image.astype(np.uint8)
        image.tofile(f)
```

In [ ]:
```python
def forward_difference_x(image):
    rows, cols = image.shape
    d = np.zeros((rows,cols))
    d[:,1:cols-1] = image[:,1:cols-1] - image[:,0:cols-2];
    d[:,0] = image[:,0] - image[:,cols-1];
    return d

def forward_difference_y(image):
    rows, cols = image.shape
    d = np.zeros((rows,cols))
    d[1:rows-1, :] = image[1:rows-1, :] - image[0:rows-2, :];
    d[0,:] = image[0,:] - image[rows-1,:];
    return d

def compute_gradient(image):
    #finds the gradient of each pixel in both x and y directions
    gradient_x = forward_difference_x(image)
    gradient_y = forward_difference_y(image)
    return gradient_x, gradient_y

def compute_divergence(gradient_x, gradient_y):
    #finds the divergence in each pixel
    div_x = -forward_difference_x(gradient_x)
    div_y = -forward_difference_y(gradient_y)
    return div_x + div_y
```

In [ ]:
```python
def convert_pgm_to_tiff(pgm_path, tiff_path):
    # Open the PGM file using Pillow
    with Image.open(pgm_path) as img:
        # Save the image as a TIFF file
        img.save(tiff_path, format='TIFF')
```

In [ ]:

```python
def find_center(elevation):
    gradient_x, gradient_y = compute_gradient(elevation)
    divergence = compute_divergence(gradient_x, gradient_y)

    # Find the point with the minimum divergence (assumed to be the center)
    center_idx = np.unravel_index(np.argmin(divergence), divergence.shape)
    return center_idx

def find_crater(elevation):
    gradient_x, gradient_y = compute_gradient(elevation)
    magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    # Define crater as regions where the gradient magnitude is significantly above the threshold
    threshold = np.percentile(magnitude, 5)  # Adjust threshold as necessary
    rim = magnitude > threshold
    return rim
```

In [ ]:

```python
def make_hole(mx=51, my=51, pixel_size=100.0, max_slope_deg=20, sun_elevation_deg=25,
              sun_azimuth_deg=45,radius_x=None, radius_y=None, data_dir='.', label_dir = '.'):

    # Constants
    radian = np.pi / 180.0
    pi = np.pi

    # Set default radii if not specified
    if radius_x is None:
        radius_x = pixel_size * 0.25 * (mx - 1)
    if radius_y is None:
        radius_y = pixel_size * 0.25 * (my - 1)

    xmid = pixel_size * 0.5 * (mx - 1)
    ymid = pixel_size * 0.5 * (my - 1)
    max_slope = np.tan(radian * max_slope_deg)
    sun_elevation = radian * sun_elevation_deg
    sun_azimuth = radian * sun_azimuth_deg
    zenith = radian * 90

    # Allocate elevation array
    elevation = np.zeros((mx, my), dtype=np.float32)

    # Create an elliptical hole
    for j in range(my):
        y = j * pixel_size
        for i in range(mx):
            x = i * pixel_size
            # Elliptical distance calculation
            ellipse_dist = ((x - xmid) ** 2 / radius_x ** 2) + ((y - ymid) ** 2 / radius_y ** 2)
            if ellipse_dist <= 1.0:
                rad_radius = pi * np.sqrt(ellipse_dist) / 2.0
                elevation[i, j] -= pixel_size * (1.0 - np.cos((pi / 2.0 - rad_radius)) ** 2) / max_slope

    # Allocate pixel array
    pixel = np.zeros((mx, my), dtype=np.int32)

    # Calculate pixel values based on slope and aspect
    for j in range(1, my - 1):
        for i in range(1, mx - 1):
            aa, bb, cc = elevation[i-1, j-1:j+2]
            dd, ee, ff = elevation[i, j-1:j+2]
            gg, hh, ii = elevation[i+1, j-1:j+2]
            dzdx = (-(gg + 2.0 * hh + ii) + (aa + 2 * bb + cc)) / (8.0 * pixel_size)
            dzdy = (-(cc + 2.0 * ff + ii) + (aa + 2 * dd + gg)) / (8.0 * pixel_size)
            slope = np.arctan(np.sqrt(dzdx ** 2 + dzdy ** 2))
            aspect = np.arctan2(dzdy, dzdx)
            pixel[i, j] = int(255.0 * ((np.cos(zenith - sun_elevation) * np.cos(slope)) +
                                       (np.sin(zenith - sun_elevation) * np.sin(slope) * np.cos(sun_azimuth - a


    # Find the crater center and rim
    center = find_center(elevation)
    crater = find_crater(elevation)

```

```
    # Create a label image
    label_image = np.zeros((elevation.shape[0], elevation.shape[1], 3), dtype=np.uint8)
    label_image[crater] = [0, 0, 255]  # Blue
    label_image[center] = [255, 165, 0]  # Orange

    # Visualization
    plt.figure(figsize=(15, 7))
    plt.subplot(1, 3, 1)
    plt.imshow(elevation, cmap='terrain', extent=[0, mx * pixel_size, 0, my * pixel_size])
    plt.colorbar(label='Elevation')
    plt.title('Elevation Model')
    plt.xlabel('Distance (m)')
    plt.ylabel('Distance (m)')

    plt.subplot(1, 3, 2)
    plt.imshow(pixel, cmap='gray', extent=[0, mx * pixel_size, 0, my * pixel_size])
    plt.colorbar(label='Pixel Value')
    plt.title('Shading based on Slope and Aspect')
    plt.xlabel('Distance (m)')
    plt.ylabel('Distance (m)')
    plt.show()


    plt.subplot(1, 3, 3)
    plt.imshow(label_image)
    plt.title('Crater Labeling')


    # Check if output directory exists, create if not
    os.makedirs(data_dir, exist_ok=True)
    os.makedirs(label_dir, exist_ok=True)

    # Save the PGM image using PIL
    pgm_path = os.path.join(data_dir, f'hole_mx{mx}_my{my}_slope{max_slope_deg}_el{sun_elevation_deg}_az{sun_az
    image = Image.fromarray(pixel.astype('uint8'), 'L')
    image.save(pgm_path)
    convert_pgm_to_tiff(pgm_path, os.path.join(data_dir, f'hole_mx{mx}_my{my}_slope{max_slope_deg}_el{sun_eleva
    print(f"Image saved to {pgm_path}")

    # Save the label image as PGM using PIL
    label_path = os.path.join(label_dir, f'hole_mx{mx}_my{my}_slope{max_slope_deg}_el{sun_elevation_deg}_az{sun_
    label_pgm = Image.fromarray(label_image, 'RGB')
    label_pgm.save(label_path)
    convert_pgm_to_tiff(label_path, os.path.join(label_dir, f'hole_mx{mx}_my{my}_slope{max_slope_deg}_el{sun_el
    print(f"Label image saved to {label_path}")


# Example usage
make_hole(data_dir='RawPits', label_dir = 'RawLabels')
```

In [ ]:
```
make_hole(radius_x = 500, radius_y = 1500, data_dir='RawPits', label_dir = 'RawLabels')
```

In [ ]:
```
for i in range(0,400):
    mx_rand = 20 + int(random.random() * 40)
    my_rand = 20 + int(random.random() * 40)
    slope_rand = 5 + int(random.random() * 40)
    az_rand = int(random.random() * 360)
    elev_rand = 20 + int(random.random() * 55)
    xrad_rand = 500 + int(random.random() * 1000)
    yrad_rand = 500 + int(random.random() * 1000)
    make_hole(mx = mx_rand, my = my_rand, max_slope_deg = slope_rand, sun_elevation_deg = elev_rand,
              sun_azimuth_deg = az_rand,radius_x = xrad_rand, radius_y = yrad_rand,
              data_dir='RawPits', label_dir = 'RawLabels')
```

Installs and Imports

In [ ]:
```
pip install torch torchvision numpy matplotlib
```

In [ ]:
```
import os
import tifffile as tiff
import imagecodecs
import matplotlib.pyplot as plt
import numpy as np
import cv2
from skimage.color import rgb2hsv, rgb2gray, rgb2yuv
from scipy import ndimage
from scipy.ndimage import sobel
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
```

Read & Preprocess Files

In [ ]:
```
def normalize_image(image):
    """
    Normalize pixel values to [0, 1].

    Parameters:
        image (numpy.ndarray): Input image.

    Returns:
        numpy.ndarray: Normalized image.
    """
    img_gray = rgb2gray(image)
    return (img_gray - np.min(img_gray)) / (np.max(img_gray) - np.min(img_gray))

def display_image(img, title="Image"):
    plt.imshow(img, cmap='gray')
    plt.title(title)
    plt.colorbar()
    plt.show()
```

In [ ]:

```python
def normalize_all_tiff(raw_directory, norm_directory, raw_label_directory, label_directory):
    # Ensure the normalized directory exists
    if not os.path.exists(norm_directory):
        os.makedirs(norm_directory)
        print(f"Created directory for normalized images: {norm_directory}")

    if not os.path.exists(label_directory):
        os.makedirs(label_directory)
        print(f"Created directory for image labels: {label_directory}")

    # Loop through all files in the raw images directory
    for filename in os.listdir(raw_directory):
        # Check if the file is a TIFF image
        if filename.lower().endswith('.tif') or filename.lower().endswith('.tiff'):
            # Create a new filename with the suffix "NORM"
            # base, ext = os.path.splitext(filename)
            new_filename = filename
            new_file_path = os.path.join(norm_directory, new_filename)

            # Check if the normalized file already exists
            if os.path.exists(new_file_path):
                print(f"Normalized file already exists: {new_filename}")
                continue  # Skip this file if the normalized version already exists

            # Construct the full file path of the original file
            file_path = os.path.join(raw_directory, filename)

            # Read the TIFF image
            image = tiff.imread(file_path)

            # Normalize the image
            normalized_image = normalize_image(image)

            # Save the normalized image to the normalized directory
            tiff.imsave(new_file_path, normalized_image)
            print(f"Normalized image saved as: {new_filename}")

    for filename in os.listdir(raw_label_directory):
        # Check if the file is a TIFF image
        if filename.lower().endswith('.tif') or filename.lower().endswith('.tiff'):
            new_filename = filename
            new_file_path = os.path.join(label_directory, new_filename)

            # Check if the normalized file already exists
            if os.path.exists(new_file_path):
                print(f"Image label already exists: {new_filename}")
                continue  # Skip this file if the normalized version already exists

            # Construct the full file path of the original file
            file_path = os.path.join(raw_label_directory, filename)

            # Read the TIFF image
            image = tiff.imread(file_path)

            # Normalize the labels
            normalized_image = normalize_image(image)

            # Save the normalized labels to the normalized label directory
            tiff.imsave(new_file_path, normalized_image)
            print(f"Image label saved as: {new_filename}")
```

In [ ]:

```python
# Test read a TIFF file
tiff_name = 'new03a071.tif' # replace with your test file name
img = tiff.imread(tiff_name)
img_gray = rgb2gray(img) # confirm greyscale

# Confirm dimensions of image array and existence of values in image array
print("Tiff Shape: ", img.shape)
print("Tiff Values: ", img)

# Normalize greyscale image
img_norm = normalize_image(img_gray)

# Plot normalized image
plt.figure()
plt.title('Normalized Image')
plt.imshow(img_norm, cmap='gray')
```

In [ ]:

```python
# Normalize and save all TIFF files in the dataset
directory = "RawPits"
norm_dir = "NormPits"
label_dir = "NormLabels"
raw_labels = "RawLabels"
normalize_all_tiff(directory, norm_dir, raw_labels, label_dir)
```

Data Split

In [ ]:

```python
def split_tiff_dataset(directory, train_size=0.8):
    """
    Split a directory of TIFF files into training and validation sets.

    Parameters:
    - directory (str): Path to the directory containing TIFF files.
    - train_size (float): The proportion of the dataset to include in the train split.

    Returns:
    - train_files (list): Paths of TIFF files in the training dataset.
    - val_files (list): Paths of TIFF files in the validation dataset.
    """
    # List all TIFF files in the directory
    files = [os.path.join(directory, f) for f in os.listdir(directory) if f.lower().endswith('.tif') or f.lower(

    # Shuffle the list of files to ensure random splitting
    np.random.shuffle(files)

    # Calculate the index for splitting
    split_idx = int(len(files) * train_size)

    # Split the files into training and validation sets
    train_files = files[:split_idx]
    val_files = files[split_idx:]

    return train_files, val_files
```

In [ ]:

```python
train_set, temp_set = split_tiff_dataset(norm_dir, train_size=0.8)

ten_percent = int(len(temp_set) / 2)

valid_set = temp_set[:ten_percent]
test_set = temp_set[ten_percent:]

print (len(train_set), len(valid_set), len(test_set))
```

Build and Load Model

In [ ]:

```python
# Define & build model
class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)

class UNet(nn.Module):
    def __init__(self, input_channels=1, output_channels=1):  # Assuming 1 input channel and 1 output channel fo
        super(UNet, self).__init__()
        self.down1 = DoubleConv(input_channels, 64)
        self.down2 = DoubleConv(64, 128)
        self.down3 = DoubleConv(128, 256)
        self.down4 = DoubleConv(256, 512)
        self.maxpool = nn.MaxPool2d(2)
        self.up1 = nn.ConvTranspose2d(512, 256, 2, stride=2)
        self.conv1 = DoubleConv(512, 256)
        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.conv2 = DoubleConv(256, 128)
        self.up3 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.conv3 = DoubleConv(128, 64)
        self.conv_final = nn.Conv2d(64, output_channels, 1)

    def forward(self, x):
        x1 = self.down1(x)
        x2 = self.down2(self.maxpool(x1))
        x3 = self.down3(self.maxpool(x2))
        x4 = self.down4(self.maxpool(x3))
        x = self.up1(x4)
        x = self.conv1(torch.cat([x, x3], dim=1))
        x = self.up2(x)
        x = self.conv2(torch.cat([x, x2], dim=1))
        x = self.up3(x)
        x = self.conv3(torch.cat([x, x1], dim=1))
        x = self.conv_final(x)
        return x

```

In [ ]:

```python
# General combo loss function
class ComboLoss(nn.Module):
    def __init__(self, alpha=0.5, beta=0.5):
        super(ComboLoss, self).__init__()
        self.alpha = alpha
        self.beta = beta
        self.dice_loss = DiceLoss()
        self.bce_loss = torch.nn.BCEWithLogitsLoss()

    def forward(self, inputs, targets):
        return self.alpha * self.dice_loss(inputs, targets) + self.beta * self.bce_loss(inputs, targets)
```

In [ ]:

```python
# Dataset loader
class PitDataset(Dataset):
    def __init__(self, data_dir, labels_dir, transform=None, target_transform=None):
        """
        Args:
            data_dir (string): Directory with all the data images.
            labels_dir (string): Directory with all the label images.
            transform (callable, optional): Transform to be applied on a sample.
            target_transform (callable, optional): Transform to be applied on the target (label).
        """
        self.data_dir = data_dir
        self.labels_dir = labels_dir
        self.transform = transform
        self.target_transform = target_transform
        self.data_filenames = [f for f in os.listdir(data_dir) if f.endswith('.tif') or f.endswith('.tiff')]

    def __len__(self):
        return len(self.data_filenames)

    def __getitem__(self, idx):
        # Load data image
        data_img_path = os.path.join(self.data_dir, self.data_filenames[idx])
        label_img_path = os.path.join(self.labels_dir, self.data_filenames[idx])

        # Load and convert images
        data_image = tiff.imread(data_img_path).astype(np.float32)  # Convert numpy array to float32
        label_image = tiff.imread(label_img_path).astype(np.float32)  # Convert numpy array to float32
        # print(data_img_path, label_img_path) # to confirm labels and data are matching correctly

        # Apply transformations to data and label images if any
        if self.transform:
            data_image = self.transform(data_image)
        if self.target_transform:
            label_image = self.target_transform(label_image)

        return data_image, label_image

# Validate Data Loader Output as needed
# for images, labels in dataloader:
#     print(type(images), images.shape)  # This should print <class 'torch.Tensor'> and the shape
#     outputs = unet_model(images)  # Ensure images are passed as input
```

In [ ]:

```python
# Transformations for the data images
data_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((256, 256)),  # Resize for consistent image size
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.float()),  # Ensure tensor is float32
])

# Transformations for the label images
label_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((256, 256)),  # Ensure label image is same size as data image
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.float()),  # Ensure tensor is float32
])
```

In [ ]:

```python
# Usage
data_dir = norm_dir
labels_dir = label_dir
dataset = PitDataset(data_dir, labels_dir, transform=data_transform, target_transform=label_transform)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

data, labels = next(iter(dataloader))
print(data.dtype, labels.dtype)  # Should output torch.float32 for both
```

In [ ]:

```python
# Function to display images
# Check to make sure everything's aligned before continuing
def show_images(images, labels, num_images=4):
    fig, axes = plt.subplots(1, num_images, figsize=(15, 5))
    for i in range(num_images):
        ax = axes[i]
        img = images[i].permute(1, 2, 0)  # Convert from CxHxW to HxWxC for plotting
        lbl = labels[i].permute(1, 2, 0)
        ax.imshow(img[:, :, 0], cmap='gray')  # Show greyscale image
        ax.imshow(lbl, alpha=0.5)  # Overlay label with transparency
        ax.axis('off')
    plt.show()

# Load one batch of data
data, labels = next(iter(dataloader))
show_images(data, labels)

```

In [ ]:

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
unet_model = UNet().to(device)

for images, labels in dataloader:
    images, labels = images.to(device), labels.to(device)
    outputs = unet_model(images)
```

In [ ]:

```python
print(outputs)
```