Automated Planetary Terrain Mapping of Mars Using Image Pattern Recognition

Design Document
Version 2.0

Team Strata:

Sean Baquiro
Matthew Enright
Jorge Felix
Tsosie Schneider

Table of Contents

1 Introduction……………………………………………………………………….3
2 Architectural Overview…………………………………………………………..5
3 Module and Interface Descriptions……………………………………………….7
4 Implementation Plan……………………………………………………………12

1 Introduction

1.1 Purpose

The purpose of the Design Document is to provide a description of the Automated Terrain Mapping software system in order to deliver an understanding of the architecture and design of the system being built. This document will aid in software development by providing detailed knowledge of the different components of the system and how they interact.

1.2 Goals and Objectives

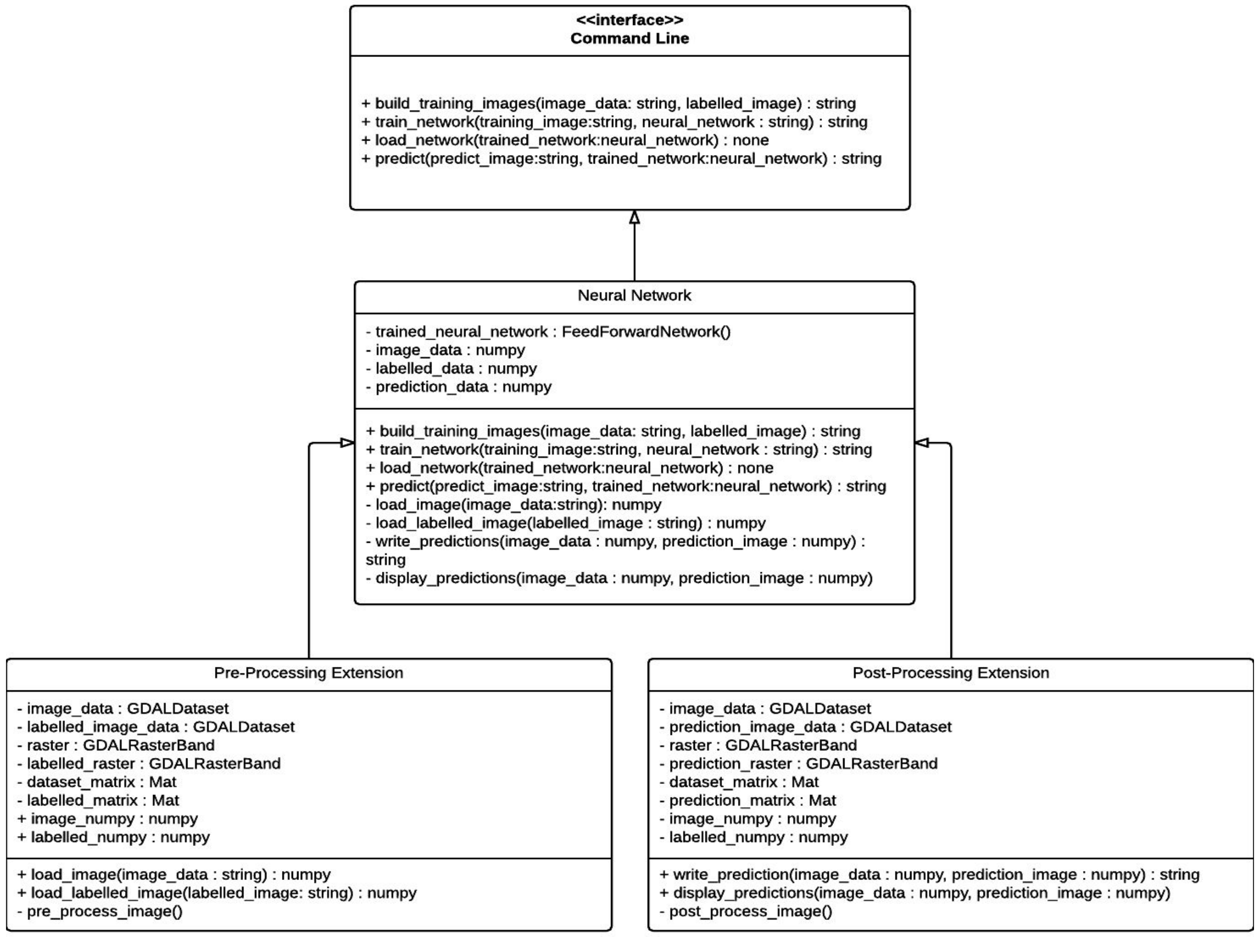
Team Strata is a Northern Arizona University Computer Science Capstone team that is working on building a computer software system proposed by USGS sponsor Ryan Anderson entitled “Automated Terrain Mapping of Mars using Image Pattern Recognition”. The computer program should allow human users to automate the task of identifying characteristic terrain types on Mars’ surface by loading HiRISE images into the system for analysis and training, have the neural network learn to recognize certain terrain types, then produce the results as a JP2 image. Current approaches to image recognition make essential use of machine learning methods, specifically the use of a convolutional neural network.

1.3 Scope

Anderson specifies that this software should remain an open source project that avoids any licensing costs. To Anderson, avoiding licensing cost is important because he wants the source code to be available to anyone. The open architecture is available on Github and will be handed to Anderson once completed. The software will consist of three major functions: (1) load HiRISE JP2 images for analysis, (2) train the neural neural network to recognize certain terrain types, and (3) produce an annotated JP2 image of the marked terrains. Some examples of the different terrain types and features that we are trying to identify include sand dunes, sinuous ridges, valleys, and canyons.

The convolutional neural network is a model with a large learning capacity that is controlled by varying their depth and breadth. They also make strong and mostly correct assumptions about the nature of images when it comes to pixel dependencies. The ability of multilayer back propagation networks to learn complex high dimensional nonlinear mappings from large collections of data set examples makes the convolutional neural network an obvious candidate for image and pattern recognition. With a multi-layer back propagation network, the network substantially realizes mapping function from input to output, and the mathematical theory has proved it has the function of realizing any complex nonlinear mapping. The convolutional neural network will serve as the main component to recognize said terrain types by feeding the neural network training data sets to learn from and then map the characteristics across an entire set of images. The program should use multiple co-registered orbital data sets acquired from the HiRISE (High Resolution Imaging Science Experiment) and CTX camera instruments equipped on the Mars Reconnaissance Orbiter. Co-registration of HiRISE images is achieved using manual tiepointing (A point in a digital image or aerial photograph that represents the same location in an adjacent image or aerial photograph) between the HiRISE image and CTX image. The HiRISE is able to photograph hundreds of targeted areas of Mars' surface in unprecedented detail. It is equipped with a telescopic lens that produces images with high resolutions which enable scientists to distinguish objects at around one meter in size from an altitude that varies between 200 to 400 kilometers above Mars. With a program that can automatically annotate a HiRISE image showing different terrain types faster than a human can, not only can we learn more about climate change and morphology on Mars, but we can possibly highlight future landing sites that can lead to more research opportunities.

2 Architectural Overview



<<interface>>
Command Line
+ build_training_images(image_data: string, labelled_image) : string
+ train_network(training_image:string, neural_network : string) : string
+ load_network(trained_network:neural_network) : none
+ predict(predict_image:string, trained_network:neural_network) : string
Neural Network
- trained_neural_network : FeedForwardNetwork()
- image_data : numpy
- labelled_data : numpy
- prediction_data : numpy
+ build_training_images(image_data: string, labelled_image) : string
+ train_network(training_image:string, neural_network : string) : string
+ load_network(trained_network:neural_network) : none
+ predict(predict_image:string, trained_network:neural_network) : string
- load_image(image_data:string): numpy
- load_labelled_image(labelled_image : string) : numpy
- write_predictions(image_data : numpy, prediction_image : numpy) : string
- display_predictions(image_data : numpy, prediction_image : numpy)
Pre-Processing Extension
- image_data : GDALDataset
- labelled_image_data : GDALDataset
- raster : GDALRasterBand
- labelled_raster : GDALRasterBand
- dataset_matrix : Mat
- labelled_matrix : Mat
+ image_numpy : numpy
+ labelled_numpy : numpy
+ load_image(image_data : string) : numpy
+ load_labelled_image(labelled_image: string) : numpy
- pre_process_image()
Post-Processing Extension
- image_data : GDALDataset
- prediction_image_data : GDALDataset
- raster : GDALRasterBand
- prediction_raster : GDALRasterBand
- dataset_matrix : Mat
- prediction_matrix : Mat
- image_numpy : numpy
- labelled_numpy : numpy
+ write_prediction(image_data : numpy, prediction_image : numpy) : string
+ display_predictions(image_data : numpy, prediction_image : numpy)
- post_process_image()

Figure 1: Architectural Overview Class Diagram

2.1 Overall System Structure

The hybrid python/C++ process pipeline of terrain pattern recognition can be divided into four constituent modules: (1) image pre-processing, (2) neural network processing, (3) image post-processing and (4) console interface. The neural network portion of the pipeline will utilize the PyBrain open source package while the image processing facilities will be C++ compiled python extension meant to increase efficiency by using high performance image processing libraries. Each module can be further divided into subprocesses necessary to present the desired interface output between process modules.

2.1.1 Console Interface

This module provides a simple interface to the user for interaction with the system pipeline. This is meant to filter user input and serve as the first layer of error checking in the system. The operations exposed by the user interface includes: building training images from provided files, training the neural network from previously built training images, loading a previously trained neural network from file, and having a loaded neural network make predictions on a provided image.

2.1.2 Neural Network

This is the module that will forward user commands for desired processing. The user commands presented to the neural network will be assumed to be correct. As such, a user that wishes to have a neural network predict the location of sand dunes will have loaded the appropriate neural network from file. The initial creation of a neural network will occur when this module is asked to train a neural network but none is loaded. User commands to build training images will be forwarded to the image pre-processing or image post-processing modules by way of methods exposed to the neural network module by the extension, as shown in Figure 1.

2.1.3 Image Pre-Processing

The image pre-processing module exposes only two functions to the neural network load_image and load_labelled_image these functions present the image data inside a numpy array-like object. The numpy array-like object is the standard method of interfacing the C++ strict data-typing with Python dynamic data-typing. The image processing of this module is a private method and occurs internally.

2.1.4 Image Post-Processing

The image post-processing module exposes only two methods to the neural network write_prediction and display_predictions. Both exposed methods require numpy data that was provided to the neural network from the pre-processing extension while the method that writes neural network predictions to disk returns a string that will be forwarded to the user. The image post-processing is handle internally.

2.2 System Interfaces

The neural network module will act as the driver class for the system, which will allow mutual exclusion of read and write extension functionality, the user interface console will simply be a module to error check user input and present system interaction options. Modules will interface through the passage of numpy array-like objects that maintain HiRISE raster images in a numpy array (Python data structure), the numpy data structure is needed in order to transfer data contained in C++ array data-structure to the

Python interpreter, which does not contain the array data-structure. The image pre-processing extension will return a processed image object based upon input parameters to distinguish between training and test datasets. The neural network module will use training data sets to build and save neural networks for future use. When test data is presented the module will use a previously trained neural network to create a numpy array filled with label predictions. The test image data and label predictions will then be given to the post-processing image extension so the information can be displayed for user inspection and writing to disk.

2.3 Constraints and Assumptions

Currently, the project is constrained by the amount of training data that has been provided by the sponsor. As of now we have the initial training dataset needed to begin implementation of the neural network that will make predictions about the “sand dune” terrain type. An underlying assumption about the terrain types is they are mutually exclusive. So as the pool of terrain training data increases the ability to augment training sets for previous training neural networks increases through the use of negative example i.e. patterns not to look for.

3 Module and Interface Descriptions

3.1 JP2 Image Processing and Data Extraction

In order for user machines to handle the large JP2 images used as input and training data, they must be processed and downsampled to a more manageable size. During the downsampling of a JP2, some data will be lost at the pixel level. In the downsampling process, pixels are grouped together and the average is taken over an area of pixels to create a new pixel value. After an image has been downsampled it will then be converted to a TIFF file and have the image data stored in an OpenCV matrix. A description of the tools we will be using to downsample and convert JP2 images can be found in section 3.1.1. The OpenCV matrix with the TIFF image data will be used as the input for the neural network. This data will then need to be transferred into a numpy array since the neural network will be set up using Python. Figure 2 shows how the tools are incorporated in order to process, convert, and extract data of an image.

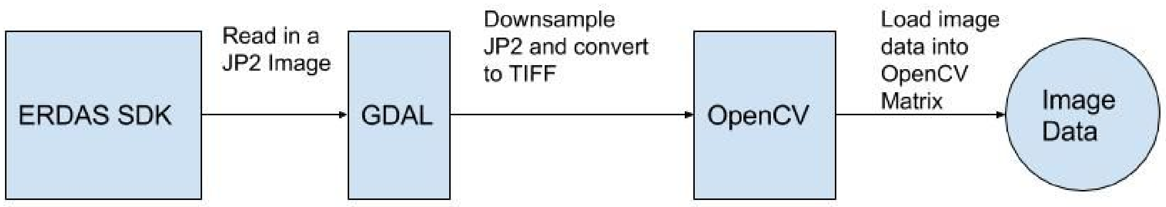

ERDAS SDK
Read in a JP2 Image
GDAL
Downsample JP2 and convert to TIFF
OpenCV
Load image data into OpenCV Matrix
Image Data

Figure 2: Processing JP2 Diagram

For the input of a JP2 image, the current goal for our team is to create a simple user interface that will allow a user to input a JP2 file for terrain mapping and another JP2 containing the training data for the neural network. We plan to accomplish this by using Python to ask for user input and require the user to input a valid location containing a JP2 file.

3.1.1 GDAL with ERDAS ECW

GDAL is a translator library for raster and vector geospatial data formats. In our system, “gdal_translate” is used as a utility to convert raster data between the JP2 and TIFF formats. The ERDAS ECW JPEG2000 SDK is used to add large image support to applications. It provides compression and enables use of very large images in the industry standard ECW (Enhanced Compressed Wavelets) image format and the standard JPEG 2000 format.

3.2 Setup and Neural Network Training

The next major component from the program will be to set up and train the neural network using Python. In order to set up the neural network, some small steps need to be completed first. The first step is processing the image with the training data. The training image will go through the same process as detailed above for the input data. Once the training and input data have been extracted into an OpenCV matrix, the data needs to be translated into a numpy array (See section 3.2.1). After the data has been translated successfully, we can then move on to setting up and training the neural network (See section 3.2.2 for more information about the neural network). Figure 3 below shows the basic steps for this component of the program.

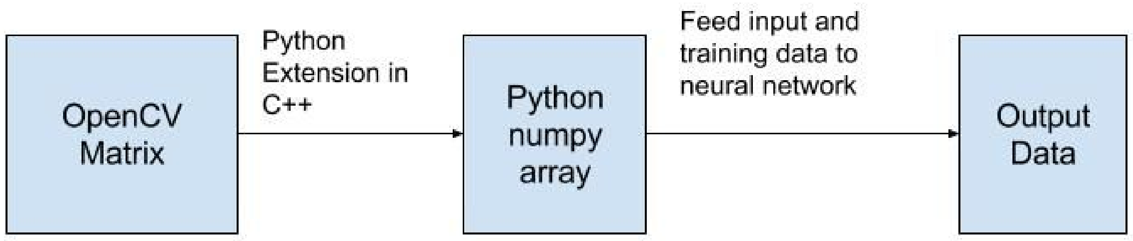


Figure 3: Neural Network Training

3.2.1 Python Extension in C++

A module created in C++ stores the TIFF image data in an OpenCV matrix. This data needs to be translated into a numpy array for use with the convolutional neural network. This extension is created without any third party tools. To support the extension the

Python API defines a set of functions, macros, and variables that provide access to most aspects of the Python runtime system.

3.2.2 Python Neural Network

A Convolutional Neural Network (CNN) will serve as the main component in the architecture. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. Unlike regular neural networks, the layers of a CNN have neurons arranged in three dimensions: width, height, and depth. The third dimension, depth, refers to the third dimension of an image. The training data will be annotated JP2 images of the feature we want the network to recognize, such as sand dunes. The testing data will be untouched HiRISE images of Mars' surface, and the job of the neural network is to learn to recognize the terrain features from the training data and accurately map them across the testing data. The architecture of the CNN will be a list of layers that transform the image volume into an output volume.

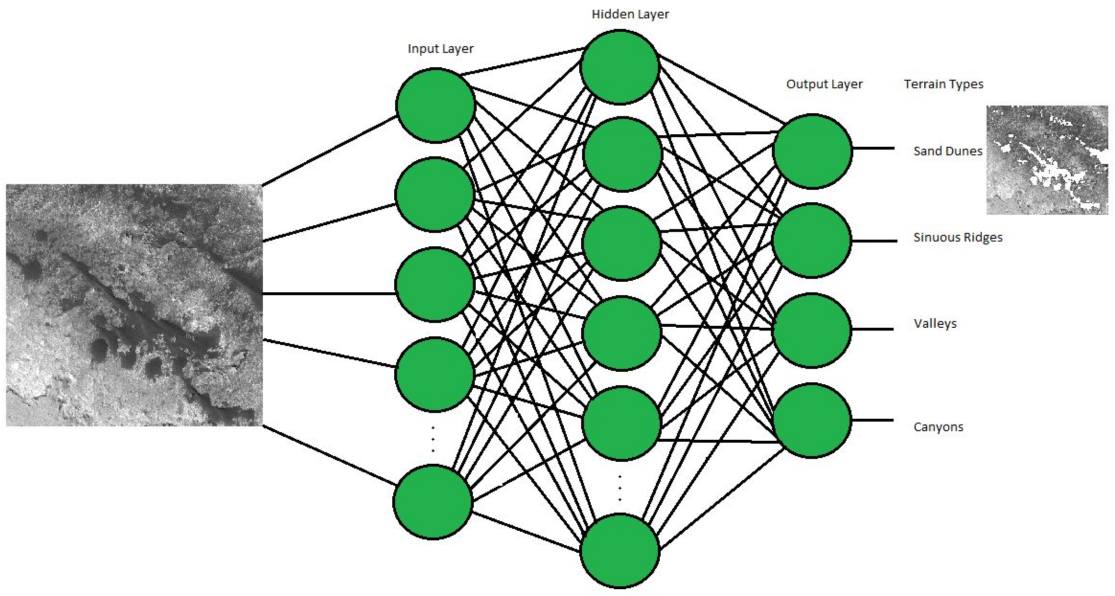


Figure 4: Convolutional Neural Network Diagram

3.3 Processing Output Data into JP2

The final step to complete the objective of this project is to process the output data back into JP2 format, which ultimately results in a fully annotated image. This step is essentially the opposite of the first step of feeding the data into the neural network.

3.4 Use Case

Figure 5 shows the interactions between the user and system role. The main user will be able to input a JP2 image to have the terrain mapped, input the JP2 with the marked terrain (training data), and see the annotated output image. The system will handle the pre-processing of the input images, train the neural network, and generate the output data.

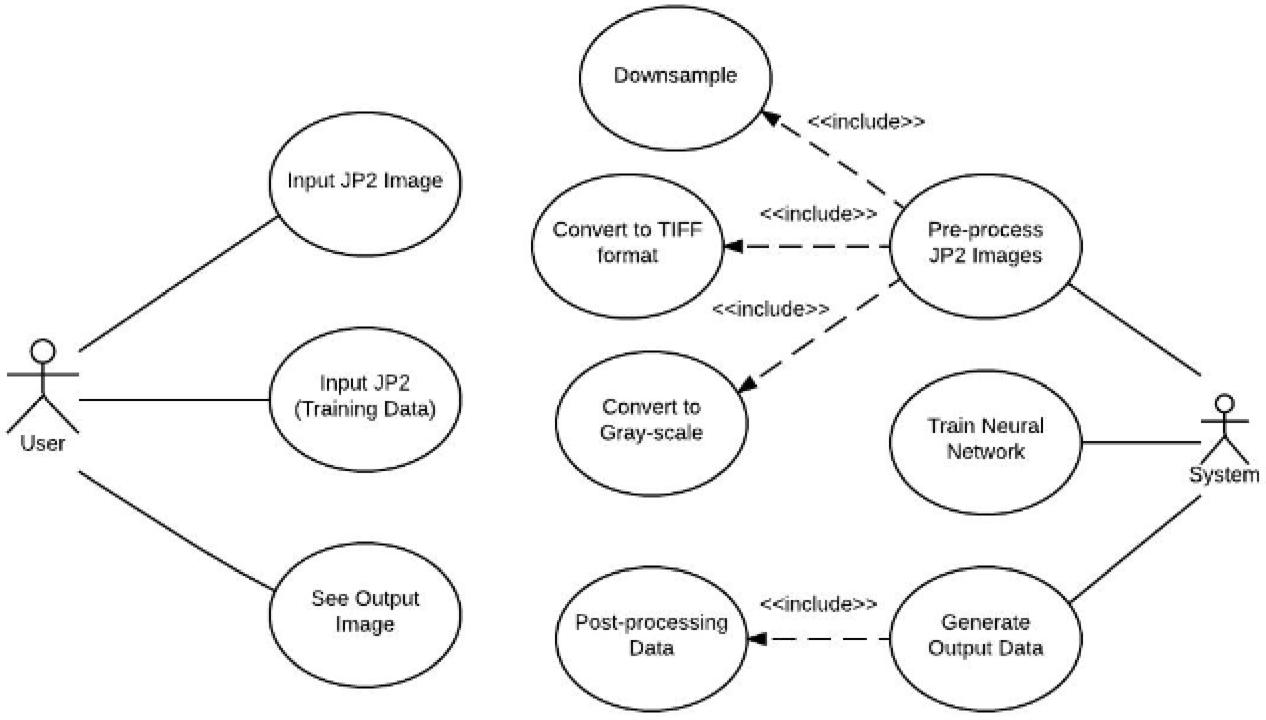


Figure 5: Use Case Diagram

3.5 Activity Diagram

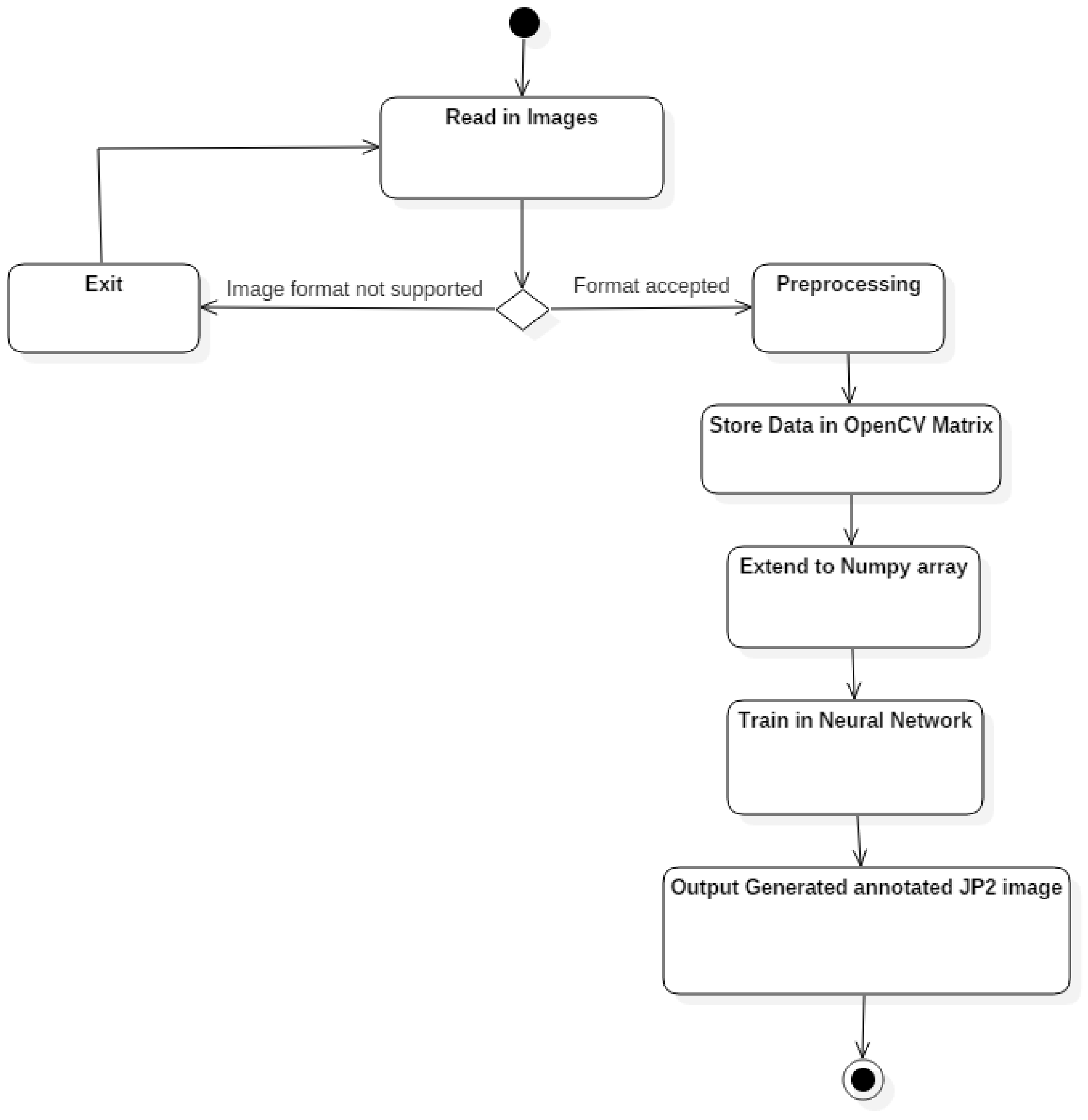


Figure 6: Activity Diagram

4 Implementation Plan

The implementation phase will be divided into six different subphases where each subphase will focus on a specific component of the program. The last three phases will focus on testing of the entire system, writing up the documentation and user guide, and UGRAD presentation of the project. The schedule (See Figure 7 below) also contains the start date of each phase and the expected time of completion for each individual phase.

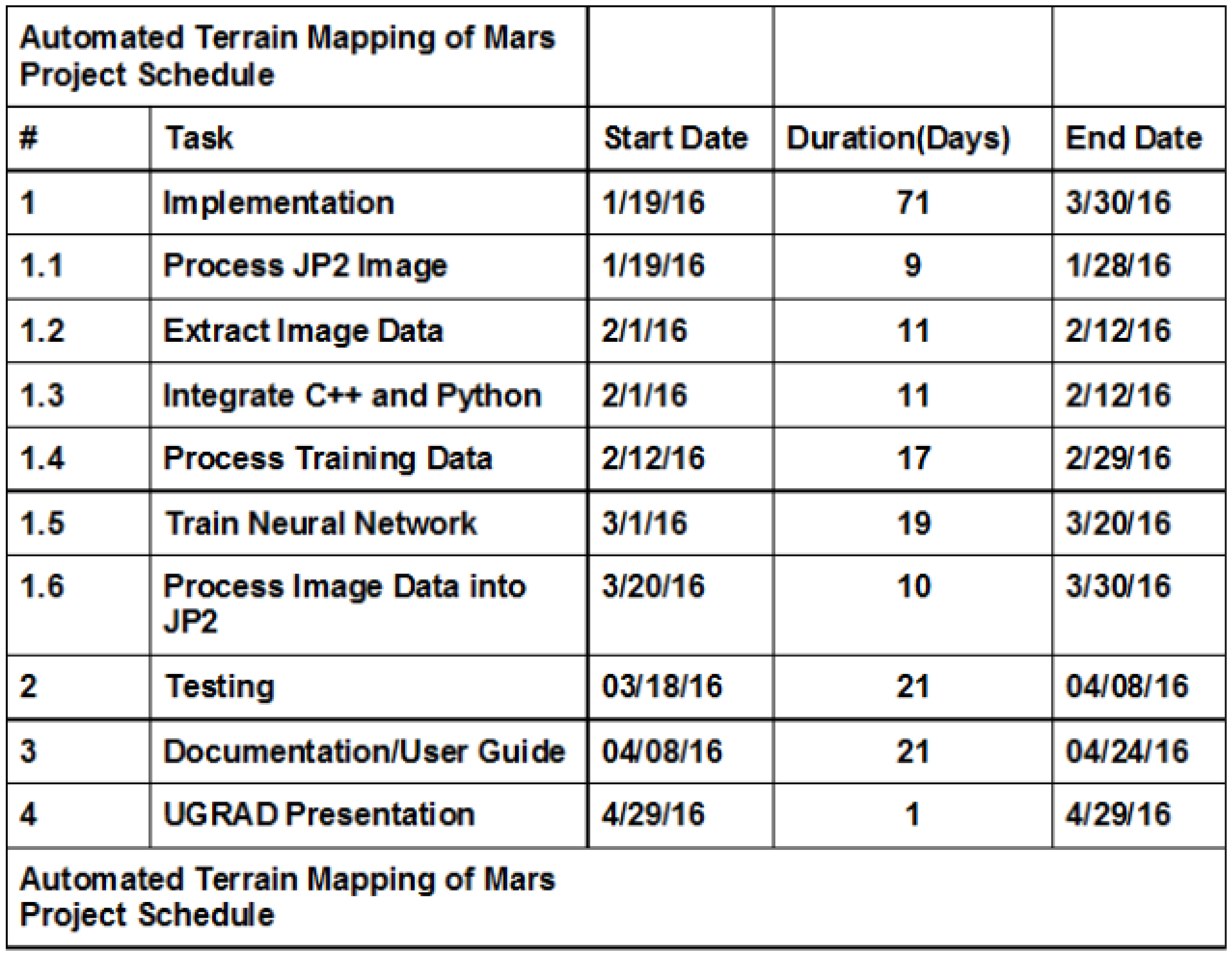
The first part of the implementation will require us to correctly process a JPEG 2000 (JP2) image file (Task 1.1). In this subphase we will use the necessary APIs to downsample a large JP2 file in order to work with a manageable data size and convert it to a TIFF file. Using the proper programming tools we will be able to implement the downsampling and converting functionality, allowing us to use large data files from the HiRISE and CTX camera instruments as our training and input data for the neural network.

The next step in the implementation process will focus on two subphases simultaneously. Team members will be assigned to either extract the image data that will be used as the input for the neural network (Task 1.2) or begin integrating C++ and python using an API (Task 1.3). The team members working on extracting image data will focus on using the TIFF image to correctly load the image into a data structure that will be used as the main input for the neural network. The team members working on the C++ and Python integration will be required to find and learn how to use a suitable API to integrate the two languages. Python we will be used to implement the neural network and integrate the image data processed in C++ into a suitable Python data structure.

The fourth subphase will be started once the prior subphases have been completed. This subphase will focus on processing the training data obtained from our sponsor, Dr. Ryan Anderson (Task 1.4). This step will follow the same process as the input data and will be used to train the neural network. Once we have processed and extracted the image data from both the input and training data, we will move onto training the neural network (Task 1.5). In this subphase will be focusing on using Python to load the image data processed using C++ into Python code and setting up the neural network.

The last subphase will focus on getting the output data from the neural network and processing the image data back into a JP2 image file (Task 1.6). This will allow us to see how accurate the neural network was at finding similar terrain types on the inputted data. This subphase will occur at the same time as the subphase involving the training of the neural network.

After the implementation phase our team will focus on testing and writing the documentation for the program. This phase will occur during the last month of the development cycle. Note that since our team is using an agile development method, we will be individually testing components during the implementation phase as well. Below is an updated project schedule table containing an overview of the implementation phase of the Automated Terrain Mapping of Mars using Image Pattern Recognition system.



Automated Terrain Mapping of Mars Project Schedule				
#	Task	Start Date	Duration(Days)	End Date
1	Implementation	1/19/16	71	3/30/16
1.1	Process JP2 Image	1/19/16	9	1/28/16
1.2	Extract Image Data	2/1/16	11	2/12/16
1.3	Integrate C++ and Python	2/1/16	11	2/12/16
1.4	Process Training Data	2/12/16	17	2/29/16
1.5	Train Neural Network	3/1/16	19	3/20/16
1.6	Process Image Data into JP2	3/20/16	10	3/30/16
2	Testing	03/18/16	21	04/08/16
3	Documentation/User Guide	04/08/16	21	04/24/16
4	UGRAD Presentation	4/29/16	1	4/29/16
Automated Terrain Mapping of Mars Project Schedule				

Figure 7: Team Strata Project Schedule