Object Detection and Digitization from Aerial Imagery Using Neural Networks

by

William Malcolm Taff IV

A Thesis Presented to the
Faculty of the USC Graduate School
University of Southern California
In Partial Fulfillment of the
Requirements for the Degree
Master of Science
(Geographic Information Science and Technology)
University of Southern California

August 2018

To Casey and Frances.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge my wife, Casey, and baby Frances for their immense patience and flexibility throughout the entirety of the program. I would also like to thank Dr. Steven Fleming for his guidance and sage advice while developing my thesis. Further, I would like to recognize Dr. Yao-Yi Chiang and Dr. Ran Tao for graciously serving on my thesis committee and providing me insight into their fascinating work.

List of Abbreviations

ANN	Artificial neural network
AOI	Area of interest
API	Application programming interface
AWS	Amazon Web Services
CSS	Cascading Style Sheets
CSV	Comma Separated Values
CNN	Convolutional neural network
DSC	Dice similarity coefficient
GDAL	Geospatial Data Abstraction Library
GIS	Geographic information systems
HPC	High performance computing
HTML	Hypertext Markup Language
IDE	Integrated development environment
JSON	JavaScript Object Notation
NDVI	Normalized difference vegetation index
NDWI	Normalized difference water index
NGO	Non-governmental organization
OS	Operating system
OSM	OpenStreetMap
PIL	Python Imaging Library.
REST	Representational State Transfer
RDBMS	Relational database management system

RLE	Run length encoding
SAR	Synthetic aperture radar
SQL	Structured Query Language
UI	User interface
URL	Uniform Resource Locator
UX	User experience
VGI	Volunteered geographic information
WMS	Web Map Service
WMTS	Web Map Tile Service

Abstract

With the recent abundance and democratization of high-quality, low-cost satellite imagery comes the distinct need for a way to analyze and derive insight from this ever-growing torrent of data. Machine learning technologies and methods are now frequently applied to large datasets to accomplish such varied tasks as language translation, fraud detection, disease diagnosis, and automated driving. This project proposes a means to apply these same technologies to automatically detect and digitize features within satellite imagery. An end-to-end machine learning and web application framework was developed to detect, extract, and digitize arbitrary classes of geospatial features. This system is composed of a web user interface which allows users to source true-color satellite imagery and existent digitized feature data and subsequently use these data to train a machine learning model that will "learn" to automatically identify features within new imagery. This involved the development of both a web application user interface and a specific type of machine learning algorithm termed a neural network that has been shown to excel in image recognition tasks. Following the identification of these features from satellite imagery, features may be exported to a geospatial database for storage and further analysis. This system and provides the foundation for a significant retooling and augmentation of manual geospatial feature digitization workflows and creates new opportunities for geospatial analysis by deriving features from aerial images rapidly en masse.

Chapter 1 Introduction

Once purely the domain of militaries and international scientific bodies, high-quality satellite imagery is now readily available to a wide range of consumers. With democratization of this data into the private and public sectors, the ability to process and derive value from this data has become increasingly important. For many applications, the volume of imagery renders manual human identification and digitization of objects extremely onerous or entirely infeasible.

To sift through this torrent, many have turned to using a suite of technologies and techniques termed "machine learning," wherein complex algorithms are "taught" to detect patterns and derive insight from enormous datasets. One specific subset of machine learning technologies, artificial neural networks or simply neural networks, have been shown to produce state-of-the art results for a host of image processing tasks. From reading handwriting to identifying species of plants and animals to driving cars and creating art, neural networks are found at the heart of many of the latest advances in machine learning and image processing (Gatys, Ecker and Bethge 2015).

As with many of the latest technological advances, the learning curve to utilizing neural networks can be quite high for all but the most technical of users. Large amounts of "training" data are required to teach neural networks how to perform a particular image processing task. The network itself often requires fine tuning and the results may require extensive post-processing. To lower the barriers of entry for utilizing this powerful technology, a web application featuring an end-to-end machine learning solution was developed. The application, named U-Map after the U-Net neural network architecture that was used within the system, provided users with a holistic data sourcing, neural network training, and post-processing pipeline. Using this application, users were able to train a neural network model to detect and

automatically digitize geospatial features from satellite imagery. Through this, users with geospatial domain knowledge were empowered to use the latest machine learning technologies through a simple web application interface.

A plethora of technologies and methods are grouped under the moniker of machine learning, with applications varying from fraud detection to targeted marketing to help desk chatbots and autonomous vehicles. While these are very disparate applications, they share the same underlying principles. The key idea behind machine learning is that computer programs can be made to "act without being explicitly programmed" (Ng 2013). Within this thesis, a specific class of machine learning algorithms was used – "supervised learning" algorithms. Within supervised learning, a set of data and corresponding "labels" are both provided to an algorithm so that the algorithm can determine which attributes likely corresponded to a given label. An often-cited example of supervised learning is that of spam filtering, wherein algorithms are given "training" email data with a binary spam/not spam label for each email. The algorithm is then able to determine what commonalities exist between each class of email – suppose spam emails have numerous occurrences of the word "free" or frequent spelling errors (Géron 2017).

This same general principle was used within this thesis, with the system allowing users to select preexisting geospatial feature data from the open source OpenStreetMap (OSM) GIS dataset via a web application to serve as labels. This feature data was then combined with the Mapbox satellite imagery dataset to create a wholistic label and data training set. These two datasets were then passed to a machine learning algorithm that used this training data to "learn" to detect a specific class of geospatial features within satellite imagery. After this model was "trained," users were able to utilize the web application user interface to pass new satellite imagery to the machine learning model, which subsequently detected and digitized similar

features within the imagery. Following digitization, users could commit this data to a geospatial database for future retrieval and analysis.

1.1. Motivation

There is a confluence of advancements in machine learning technologies, satellite imagery, and geographic information systems (GIS) that have made this project feasible and served as motivation. These topics are discussed in the below sections.

1.1.1. The Burgeoning Field of Machine Learning

Machine learning is a broad term that encompasses many technologies within the field of computer science. Broadly speaking, most machine learning methodologies apply complex algorithms to problems that would be too time-consuming or simply impossible to solve with hand-written computer code or manual analysis and human workflows. To address this problem, many machine learning technologies center around the development of complex statistical models (Ng 2013).

Most machine learning technologies hold to the axiom that "the best way to predict the future is the examine the past." Large volumes of existent data are processed to produce extremely complex models – often with thousands of parameters. As such, these machine learning algorithms can be used to derive correlations between large numbers of data points. These models are used across all swathes of the public, private, and academic sectors. Machine learning is used to clear spam from inboxes, recommend movies, determine credit worthiness, drive cars, and diagnose disease.

1.1.2. A New Abundance of Low Cost Satellite Imagery

This project applied machine learning principles to the field of geographic information science. With the recent availability of high-quality, low-cost satellite imagery and commodity high performance computing (HPC) assets, this field has seen an explosion of interest. Governmental bodies, major corporations, and the open source community are all working towards a common objective – to make sense of and derive value from enormous amounts of satellite data.

In many applications, human processing and analysis is simply too expensive and time consuming to be feasible given such a bulk of information. DigitalGlobe, a private satellite imagery company, possesses over 90 petabytes (90,000 terabytes) of satellite imagery data alone (Hamilton n.d.). Planet, another private imagery company, operates a constellation of over 150 small, lower resolution imagery satellites providing high-frequency imagery updates. Provided weather and atmospheric conditions allow, the company claims to be able to provide daily imagery refreshes for the whole of the globe (Draper 2018) (Vance 2017). Without reliable automated means to sift through this imagery, valuable insights may be lost.

1.1.3. Combining Machine Learning and GIS

Throughout the academic body of knowledge, there is a long history of using remote sensing to detect and digitize features for analysis. Many have utilized the properties of multispectral imagery to identify buildings, water, vegetation, and other types of objects and landcover. There is a large suite of complex specialized algorithms, such as the Normalized Difference Vegetation Index (NDVI), that can be applied to derive meaning from this multispectral imagery. However, these algorithms are often required to be highly specific and excel at detecting only a small gamut of objects or phenomena within imagery.

With recent advances in machine learning technologies and an abundance of high-resolution, high-refresh rate satellite imagery, there are new opportunities for analyzing imagery. Many researchers and private organizations doing this work have pivoted to the analysis of true-color imagery due to the increased flexibility of being able to both apply more generic algorithms to detect numerous types of objects or phenomena, and to differentiate between objects or phenomena that have similar or identical multispectral signatures. For instance, using traditional techniques it would be challenging to automatically discern a baseball field from a lawn or field within multispectral imagery – both may be covered mostly with grass. Both classes of objects would share a common multispectral signature. However, using true-color imagery and new powerful machine learning algorithms, these objects can be easily separated and identified. The Terrapattern project uses these principles to allow the public to search seven major metropolitan areas using an aerial imagery tile as the query input. The software then returns any locations within that city that their machine learning model has determined to "look like" the input imagery – allowing users to locate intersections, pools, helipads, and many other classes of objects within the area (Levin, et al. 2016).

In a more complex implementation, the company Orbital Insights uses their machine learning algorithms to derive economic, human security, and social indicators. Using high-refresh rate imagery from Planet Labs, Orbital Insights has developed analytics on consumer spending using parking lot occupancy, construction growth by identifying laid foundations, and energy stockpiles by analyzing the size of shadows cast on the floating roofs of petroleum storage tanks (Orbital Insight 2017). None of these applications would be possible with multispectral imagery or traditional means of analysis, nor scalable if relying upon manual human intervention.

1.2. General Overview of the System

The system functions as an integrated end-to-end machine learning platform, allowing users to (1) source feature and imagery data, (2) train a neural network using these data, (3) use a trained neural network model to detect instances of objects within new satellite imagery, and (4) validate the output of the model through a defined workflow. The intended users of this system are, generally, those that require spatial feature data for a given area of interest for which there is limited existent feature data (Figure 1). The users are understood to serve in a geospatial data management role within their organization and have a sound understanding of geospatial data. The users are not presumed to have an in-depth understanding of machine learning nor neural networks. The only machine learning domain knowledge required of users is the foundational concept of training data. Selection of training data within the system was extremely intuitive, requiring a user to simply identify a class of geospatial feature to be used as training data and to select a geographic area from which to source these features. Default values were given for all other machine learning model parameters, providing the user with acceptable predefined values as well as an indication of appropriate ranges for these values.

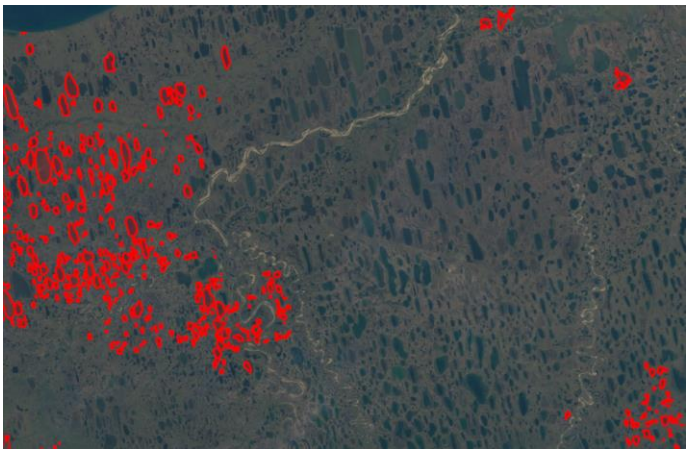


Figure 1 Existent OSM lake features in red over satellite imagery demonstrating the lack of existent OSM feature data, near Utqiaġvik (formerly Barrow), Alaska

1.2.1. System Functionality

A multipart solution was chosen to meet to goals of the application. First, a web application user interface (UI) was required for users to select training feature data within an area of interest (AOI) and allow users to review satellite imagery for this AOI. The web application UI was the main component of the "data pipeline" that was used to select training data that are passed to the backend machine learning framework. After the user passes the training feature data to the machine learning framework, the system automatically downloads high-resolution satellite imagery corresponding to the spatial locations of the training feature data. It is impractical and resource intensive to display high-resolution imagery directly within the web UI, so the users were merely shown imagery tiles from a web map tile service (WMTS), leaving the backend machine learning code to source this data. With the training feature data and high-resolution satellite imagery secured, the machine learning code, implemented in Python, iterates through the entire training dataset to develop a machine learning model that detects lakes within satellite imagery. After model development is completed on the backend, the system returns example model performance to the user via the web UI.

The user then is able to select a new AOI, presumably an AOI with limited existent feature data, and pass this AOI to the backend machine learning code. The code again sources high-resolution imagery for this AOI and iterates through all selected imagery and digitizes objects that have a high probability of being the same class of geospatial feature that was used to train the model. These objects are then returned to the user via the web UI, wherein the user may choose to commit these features to a geodatabase.

1.2.2. System Architecture

Scalability, ease of use, and user experience are three of the most important overarching non-functional requirements of this system. As such, it is exceedingly important that the system be as intuitive as possible. To this end, all of the infrastructure for this system is deployed on the web – the UI, the data sourcing pipeline, data pre- and post-processing, and output verification are deployed within the cloud. Many existent systems, such as Skynet and DeepOSM (see Chapter 2 Related Work), rely heavily on locally installed assets and manual retrieval of satellite imagery data.

In contrast, this system performs all processing remotely. The system is deployed wholly on Microsoft's Azure cloud computing platform and utilize data hosted by a number of vendors. The system is comprised of: (1) web application servers running on the .NET and Node.js frameworks within Azure, (2) WMTS provided by a number of vendors, (3) a Flask Python server for machine learning processing and execution of all Python code deployed on Azure, and (5) a Microsoft SQL Server database also hosted on Azure.

1.3. Structure of the Thesis Document

In the pages that follow, this thesis is divided into four primary sections. The Related Work chapter includes a review of existent literature on the subjects of remote sensing, machine learning, and a number of efforts combining these two disciplines. Chapter 3 - Application Requirements, details the objective of the project, system requirements, and design rationale. Chapter 4 - Components of the Final Application, details the process of developing the system. Within Chapter 5 - Results, selected UI screens from the system are given, along with the physical system architecture, data flows, pseudo code, and testing results. In Chapter 6 –

Conclusions, a summative discussion of the project is given along with challenges, any limitations, and future opportunities for improvement.

Chapter 2 Related Work

The following provides a review of existent academic literature as well as commercial and open source software systems that have influenced this project. Within the Literature Review section, the topics of remote sensing, machine learning, computer vision, and works regarding machine learning principles in the specific context of GIS will be discussed. The Existent Applications section features a review of select open source and proprietary systems using machine learning to derive geospatial data and insights from satellite imagery.

2.1. Literature Review

As this system spans the domains of GIS, remote sensing, computer vision, and machine learning, this review of related work is broad while also highly focused on select topics such as machine learning architecture and algorithm design. The Remote Sensing and Object Detection section discusses previous works attempting to detect objects or phenomena from aerial imagery. The Machine Learning and Computer vision section discussed the broader field of computer vision and several applications of this technology outside the domain of GIS. The Machine Learning and GIS section details a number of projects that attempted to utilize machine learning and computer vision technologies within the field of GIS.

2.1.1. Remote Sensing and Object Detection

There have been numerous previous attempts to automatically derive object information from remotely sensed satellite imagery. These efforts have varied in complexity, some using rather simple manual interventions, and others utilizing complex data processing techniques in attempts to identify natural and manmade features. An overwhelming majority of these earlier projects leveraged the unique properties of multispectral and hyperspectral imagery within their

analyses as opposed to using true-color images. Many researchers opt to use multispectral and hyperspectral imagery as the spectral signature of many types of landcover and objects differ greatly. They use these differences as an input to their classification processes (GISGeography 2017) (Emerson, Lam and Quattrochi 2005).

Jiang et al. (2014) used this technique in their attempts to create an "automated method for extracting rivers and lakes," which aimed to automatically extract water features from multispectral Landsat imagery. The authors created several permutations of a "water index" using several spectral bands to identify water features. Similarly, Lu et al. (2012) utilized the NDVI in their analysis to identify the conversion of Brazil's *cerrado* (landcover similar to a savanna) to agricultural and pastoral usage. Some researchers have taken this methodology one step further, integrating multispectral imagery with other visible characteristics of features when attempting to perform object detection. Emerson et al. (2005) used the Image Characterization and Modelling System (ICAMS) software package to create indices of pixel color variance, topographical object complexity, and spatial autocorrelation. These indices were then used within complex algorithms in an attempt to classify suburban landcover.

Geospatial object detection algorithms are not, however, strictly limited to the used of aerial imagery as some researchers have shown. In their paper "A General Approach for Extracting Road Vector Data from Raster Maps" (2013), Chiang and Knoblock used object detection and extraction algorithms to generate road data from existent raster maps. Due to the high color fidelity and range of many raster maps, particularly of those scanned from physical maps, the authors first binned similar colors found within the map into a significantly constrained color space. Following this, a user was prompted to select known road features within the map, as to identify different colors associated with road features. With this

information, the authors were able to extract all features within the raster matching the user defined road coloration scheme. Further automated post-processing was then performed to remove features that were found to be morphologically dissimilar to roads yet happened to share similar coloration within the raster. The authors then produced centerlines from the cleaned roads layer and performed subsequent post-processing to correct road intersections distorted during the generation of centerlines.

Unlike these researchers, true-color imagery aerial was used exclusively within this application as this architecture is inherently more extensible as objects with similar multispectral signatures may be differentiated and low-cost true-color imagery is more readily available on the marketplace. Further, by training a model with true-color imagery few adjustments to the core machine learning model would be required for the model to recognize other objects – the operator would merely have to train an additional model with examples of other object classes. The Terrapattern project follows this philosophy, having built an online platform and service for identifying hundreds of discreet categories of objects all using the same algorithm (Figure 2) (Levin, et al. 2016). However, these previous works, especially that of Emerson et al., do provide some insight into the logic of a "black box" machine learning algorithm (Emerson, Lam and Quattrochi 2005). It is highly likely that feature attributes such as color variance, geometry, and spatial relationships to other objects will be included in the hundreds or thousands of parameters that a trained model will utilize.

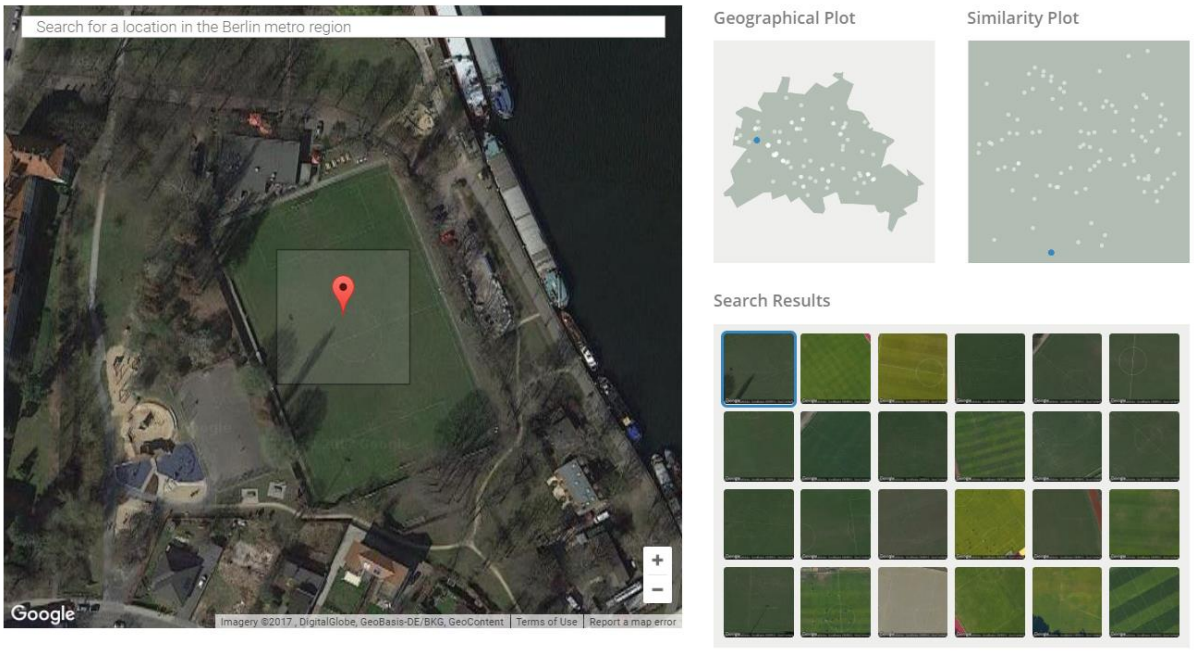


Figure 2 Searching for soccer pitches in Berlin using Terrapattern (Levin, et al. 2016)

2.1.2. Computer Vision and Machine Learning

Moving in tandem with progress in the broader field of machine learning, computer vision is also seeing a dramatic increase in both interest and capability. Today, computer vision is being used to drive cars, power surveillance systems, identify tumors, and unlock your cellphone. Driving these advances in computer vision is a specific type of machine learning architecture termed artificial neural networks (ANNs). While an in-depth discussion of ANNs is outside the scope of this paper, they operate by using large numbers of artificial neurons (essentially discreet algorithms) that each specialize in a specific task (e.g. identifying color or shape). When these neurons work together, they make an excellent tool for detecting complex relationships within data (Hijazi, Kumar and Rowen 2015) (Schmidhuber 2015).

Lee (2015) found neural networks to greatly outperform traditional image processing algorithms when used for robotic vision applications. The author began by developing a traditional object extraction algorithm which functioned by segmenting imagery based on the

perceived shape, coloration, texture, and similar attributes of objects within a scene. This algorithm was then used to allow an autonomous robot to locate objects within a room. Lee then performed the same experiment, this time utilizing a convolutional neural network (CNN) – a specialized ANN architecture which has been shown in research to excel at image processing. When the robot performed the same task using the CNN architecture, object detection performance increased by 33-50%.

Song and Yan (2017) also found CNNs to have distinct advantages over traditional image processing techniques. The authors compared traditional algorithms and CNNs when applied to image segmentation tasks; that is, separating distinct objects within imagery. Among the traditional image processing techniques discussed by the authors were threshold segmentation and edge detection segmentation algorithms. Threshold segmentation algorithms are by far the least complex and computationally intensive, operating by simply dividing an image into a number of classes based on the color value of every pixel within the image. The edge detection algorithms given by the authors segmented an image by identifying areas of local variance where attributes such color, brightness, texture, or shape abruptly change identifying that horizon as the edge of an object. Following the evaluation of these and other similar algorithms, the authors evaluated Google's DeepLab CNN architecture. Much as Lee, Song and Yan found that the CNN provided better results than the traditional computer vision approaches. Interestingly, the authors achieved the best results by first utilizing DeepLab and then post-processing the output using a traditional segmentation algorithm.

While these researchers used machine learning to address the "where" of objects within imagery, these algorithms did not address the "what." That is, they were identifying discreet objects within imagery, yet not identifying what the objects were. A subset of the image

processing field, termed semantic segmentation, addresses this issue. Within semantic segmentation, objects within imagery or video are identified and assigned a class identifier such a “tumor,” “nerve,” “road,” and the like. Through this, data is derived from the purely visual objects within the imagery or video – opening up many more opportunities for further processing or decision making (Shuai, Ting and Gang 2016).

One of the more popular of these semantic segmentation tools is the SegNet scene segmentation library, also utilizing a CNN architecture (Figure 3). SegNet was initially developed for the purpose of classifying objects within “road” scenes into semantic classes (e.g. roads, sidewalks, vehicle, pedestrians) in real time for use by self-driving cars (Badrinarayanan, Kendall and Cipolla 2017). Several GIS-related object detection projects, notably Skynet and DeepOSM, have used portions of SegNet within their code to perform segmentation tasks (Development Seed 2017) (Johnson 2017).

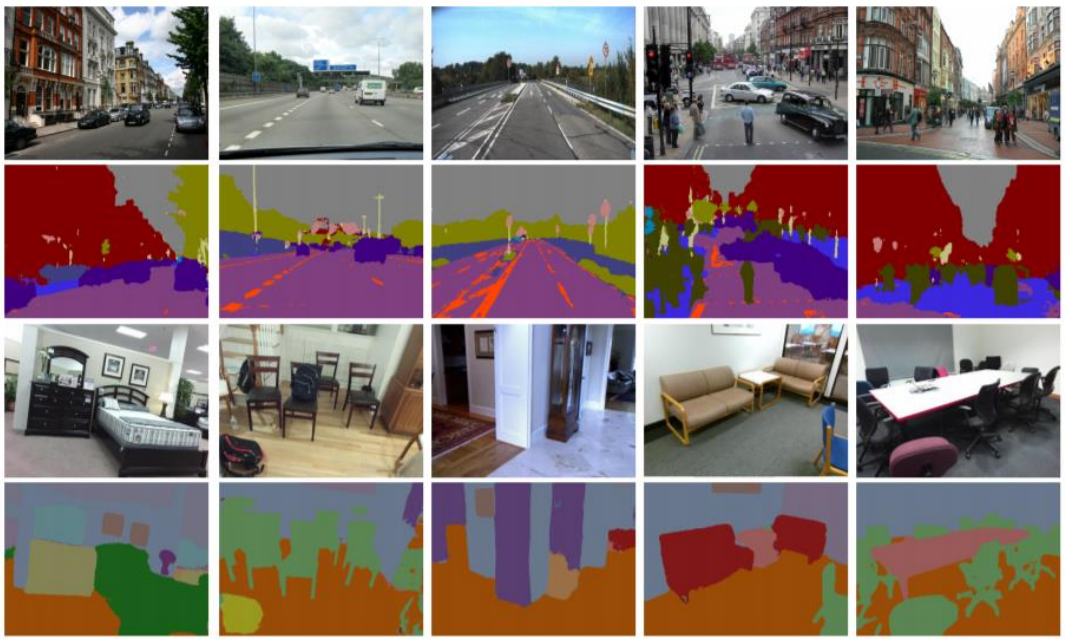


Figure 3 SegNet input and classified output (University of Cambridge n.d.)

2.1.3. Machine Learning and GIS

There is a growing interest in the integration of machine learning and GIS, with contributors in this field ranging from global corporations and the world's leading universities to talented hobbyist in the open source community. Facebook, for example, has used machine learning to detect, digitize, and commit updates to the OSM road dataset (Patel 2017). Facebook's algorithms are proprietary as are those utilized by most other companies in the private sector such as Descartes Labs, SpaceKnow, and Orbital Insight. Fortunately, there are many researchers within the academic and open source communities willing offer their methods and code to the public.

Similar to much previous work using aerial imagery, a number of researchers have applied machine learning methods to the analysis of panchromatic, multispectral, hyperspectral, and other advanced imagery formats. In their paper "Road Segmentation in SAR Satellite Images with Deep Fully-Convolutional Neural Networks" (2018), Henry, Majid and Merkle used CNNs to extract road networks from synthetic-aperture radar (SAR) data. The authors note challenges traditional methods face when distinguishing roads from railways, rivers, and other features that share similar visual and topographic profiles. To segment road features, the FCN8 CNN architecture was utilized, with manually annotated images (using data sourced from Google Maps) as training data. The authors flattened the SAR data, providing the neural network with grayscale two-dimensional images. While this inherently reduced the amount of information available to the model, the authors noted the main advantage of using SAR data for object detection to be the ability of SAR sensors to collect data regardless of prevailing weather conditions, unlike optical imagery which can be occluded by cloud cover.

Unlike these researchers, Iglovikov, Mushinskiy and Osin (2017) elected to provide all available data to their model when developing a segmentation algorithm for landcover. The

authors utilized 57 km^2 (57 imagery tiles each covering 1 km^2) of multispectral imagery gathered from the WorldView-3 satellite to develop their solution; the coastal/aerosol, blue, green, yellow, near-infrared, and short-wavelength infrared bands were available within the dataset. They further augmented this data by creating a reflectance index channel and a flattened RGB image. These data were then provided to a U-Net CNN which had been modified to learn from the additional depth of data, as the original U-Net was developed for grayscale imagery (Ronneberger, Fischer and Brox 2015). The authors were able to produce relatively accurate segmentation results, given the small training dataset of 57 images.

These examples notwithstanding, there is distinct bias within the open source community towards the use of true-color imagery over panchromatic, multispectral, hyperspectral, and other more advanced classes of imagery. This is likely due to two primary factors: true-color imagery is more freely available from common sources such as Google Maps and Mapbox, and existent machine learning algorithms can be more readily applied to true-color aerial imagery without extensive modification. One of the most prominent and often cited papers on true-color aerial imagery segmentation is Mnih's doctoral dissertation, "Machine Learning for Aerial Image Labeling" (2013). This paper has become a lodestar for many projects in the open source community. While Mnih has since patented his code and specific methods, this paper is still an extremely relevant discussion of general architecture, image pre- and post-processing, and model validation.

2.2. Existent Applications

There is a growing interest both within the commercial sector and the open source communities regarding the use of machine learning technologies to mine and derive insights from satellite imagery. These projects vary from single contributor open source applications to

initiatives lead by multinational non-governmental organizations (NGOs) and private corporations.

2.2.1. DeepOSM and Skynet

DeepOSM is an open source project that uses OSM data to train a model to detect and digitize road data using a neural network. The current work on DeepOSM is relatively limited in scope, focusing solely on roads and using the United States Department of Agriculture's National Agriculture Imagery Program (NAIP) aerial imagery as the only imagery data source. Further, following object prediction, predictions are used to overlay the source satellite imagery and rasterized as JPEG files limiting any future processing or analysis (Johnson 2017).

The Skynet project builds off the work of DeepOSM. Skynet is also trained towards road detection and digitization, but, unlike DeepOSM, Skynet is much more feature rich and is supported by the mapping and data visualization company Development Seed (Figure 4). In addition to a more complicated machine learning model, Skynet features simple tools for viewing the output of the model along with quality assurance and correction tools for post-processing (Development Seed 2017). Together, Skynet and DeepOSM had the greatest influence on the technical implementation of the project. The neural network design and data input/output pipelines within U-Map diverged significantly from this previous work, as the object detection functionality within U-Map was less limited in scope and neither DeepOSM nor Skynet output data directly into a data store, but the high-level designs have commonalities.

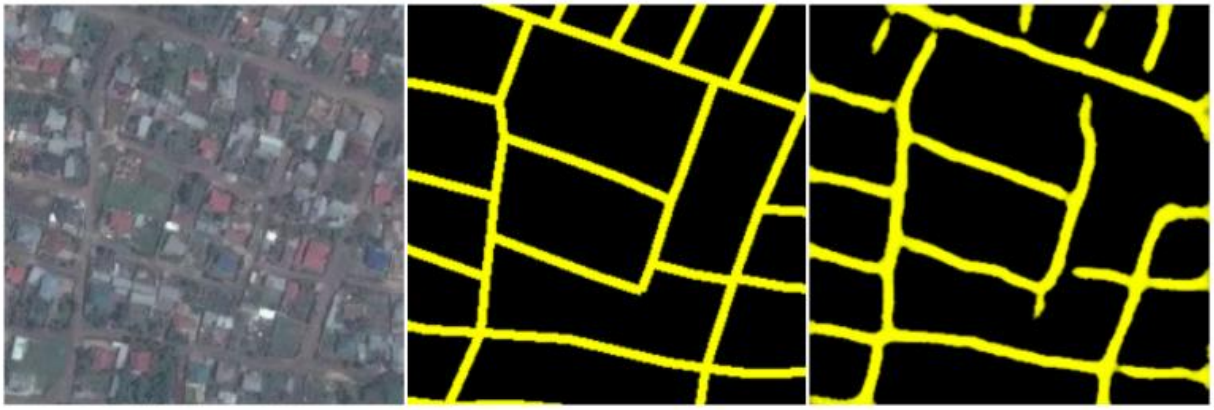


Figure 4 Skynet input, OSM data, and predictions near Goma, Democratic Republic of the Congo (Development Seed 2017)

2.2.2. Terrapattern

The open source Terrapattern project operates in a fundamentally different manner than the DeepOSM or Skynet systems. Terrapattern alternatively functions as a "image search" platform for satellite imagery. Instead of digitizing features, this application allows users to find visually similar satellite imagery tiles within several metropolitan areas. This application functions through the use of an image classification algorithm that assigns a class label to the entirety of any image, as opposed to DeepOSM and Skynet which assign road classification labels to individual pixels within a given image.

However, Terrapattern did illuminate several interesting use cases as this application featured hundreds of different classes within the system, in contrast to DeepOSM and Skynet which only functioned on a singular type of geospatial object. The Terrapattern system is still in the initial testing phases, but the creators referenced several "inspiration" use cases pioneered by other researchers that lead the design of the system. Among these was the ability to track deforestation within the Amazon and to monitor conflict in South Sudan by detecting instances

of burned structures (Terrapattern 2016) (Monitoring of the Adean Amazon Project 2018) (Al Achkar, et al. n.d.).

2.2.3. Orbital Insights

Orbital Insights is perhaps the most "talked about" start-up in the satellite imagery and machine learning space. Orbital Insights partners with DigitalGlobe and Planet to source high refresh rate imagery and derives advanced analytics from this imagery. These analytics and data are applied across a wide swathe of sectors ranging from defense, agriculture, energy, and finance (Scoles 2017).

For instance, the company has used their imagery and analytics platform to develop leading indicators for retail sales by automatically detecting the number of vehicles within department store parking lots day-to-day during busy shopping seasons – giving investors and analysts an early edge on the market (Figure 5). In another widely reported example, Orbital Insights used object detection algorithms to locate all of the floating roof oil storage tanks – massive tanks used to store tens of thousands of barrels of oil – in China. Having identified all of these objects, the satellite imagery was further processed to provide analysts with much more accurate intelligence regarding the oil storage capacity of one of the world's largest economies (Vance 2017).

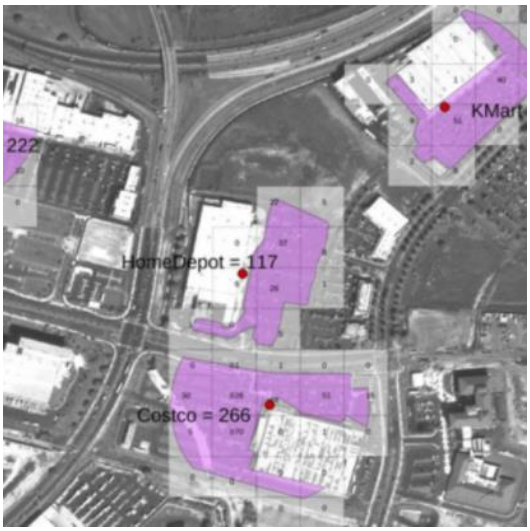


Figure 5 Parking lot vehicle counts derived from aerial imagery (Orbital Insight 2017)

The company releases little information regarding the detailed architecture of their system, and the little that can be inferred from press releases and the company's official blog are of little technical use; they merely state that they are using deep learning, neural networks, and computer vision to drive their analytics. However, the interest in this company and their early measurable success did serve as inspiration for this system, as well as pointed to some potential future use cases.

Chapter 3 Application Requirements

This chapter presents the core objectives and requirements of the application along with the high-level design considerations that went into the development of the application and underlying infrastructure. The Application Goals and User Requirements sections give an overview of the general purpose of the application and its functionality and workflow. The Functional Requirements section details the specific operations the application performs and how a user interacts with the application to generate a desire output. The Design Principles and Choices section presents the crucial technical infrastructure of the application and explains the decision-making processes that lead to the final architecture of the application.

3.1. Application Goals

The goal of the application was to provide a means for the rapid training and utilization of a machine learning model which detects and digitizes objects from satellite imagery. To realize this goal, the system provided a user-friendly web UI to allow users to select geospatial features to be used for training a machine learning model. The system then fully automated the preprocessing of training data to transform a user's selected geospatial features into an input format suitable for model training. The system then trained a machine learning model to detect the same type of geospatial features as provided by the user. Users were able to review the performance of the model and adjust any parameters necessary to increase the model's detection accuracy.

Following the satisfactory training of a machine learning model for a given type of geospatial feature, users could then pass new satellite imagery to the trained model which would predict instances of a given type of geospatial feature within the imagery. Users could then review the predicted instances of features against ground truth imagery and commit detection

predictions to a database for future processing or retrieval. Through this system, users could, in effect, automate a significant portion of their geospatial feature digitization workflow. Figure 6 provides a high-level conceptual workflow of the system, from training data selection to prediction validation.

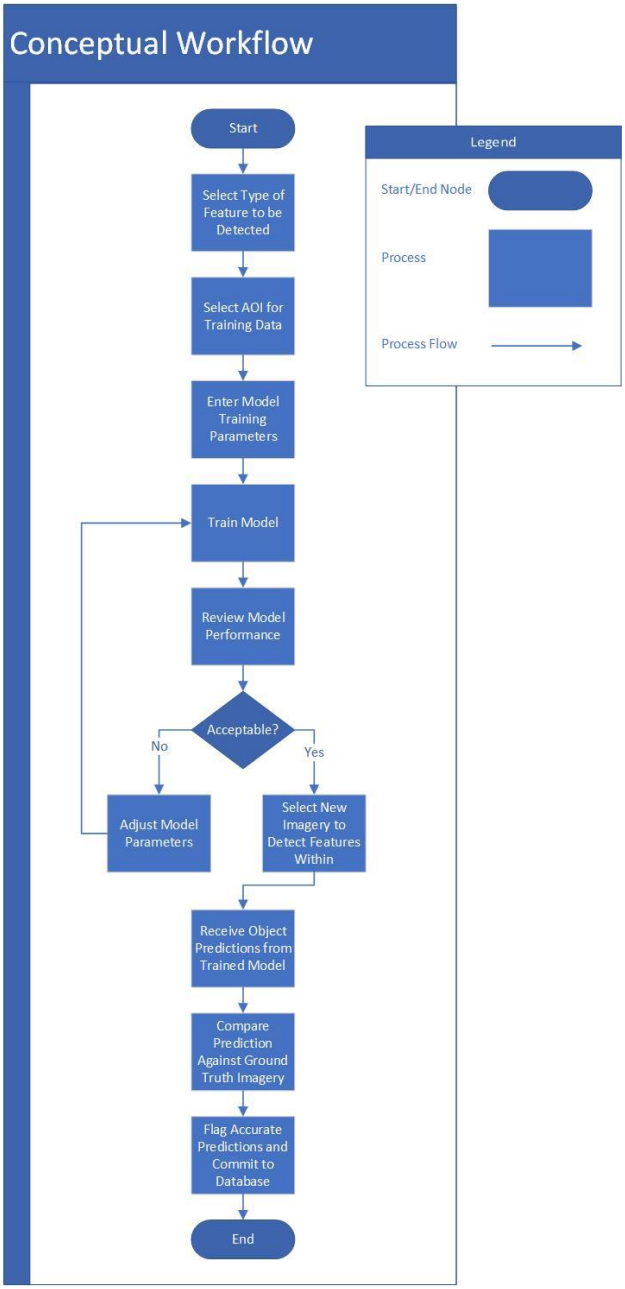


Figure 6 Conceptual workflow of the application

3.2. User Requirements

The intended audience of the solution were technical GIS users such as GIS data managers and analysts. The userbase was understood to have a fundamental understanding of geospatial data, data management, general database concepts, and related topics. As the user would be empowered to generate and commit spatial data to a database, it was assumed that the user was competent and comfortable in this role. Further, the intended audience was presumed to have a need for the bulk, automated digitization of geospatial features.

From an end user perspective, the selection of training data and training imagery was required to be as fluid as possible, with minimal manual intervention or sourcing of data and imagery. Configuration of the machine learning model should likewise require little intervention and be performed wholly through a web user interface – as opposed to a command line or the direct editing of code. Similarly, when utilizing a trained model, passing novel imagery to the model must be performed exclusively from a web user interface, with no manual preprocessing of satellite imagery. The user should receive the model prediction results via a user-friendly web interface and have the ability to select high-probability object detection instances and commit these results to a database.

3.3. Functional Requirements

The below requirements represent the high-level functionality that the system must perform. These requirements span from general navigation and usability of the system to core workflow functionality such as selecting training data, generating a machine learning model, and validating model prediction output.

3.3.1. Map Interaction and Navigation

The user must be able to pan and zoom throughout the map within the web UI. To speed the user's workflow identifying AOIs, the map should include the ability to perform geocode searches for specific locations. Further, to aid in the selection of training data the user should be able to select from a number of general reference basemaps including satellite imagery, streets, and topographic basemaps.

3.3.2. Select Training Data

The user must be able to enter OSM query parameters using the Overpass API (application programming interface) query language. The user may enter the type of data, the key, and the value in the pattern of: *datatype["key" = "value"]*. The user must then be able to select one or more rectangular AOIs. The system will then load all OSM features that match the provided query parameters within each AOI. The system will then load all Mapbox satellite imagery tiles within each AOI.

3.3.3. Model Training

The user must be able to enter model training parameters following the selection of training data. The system will generate masks of any features matching the user's query parameters within the selected AOIs. The system will consume the AOI bounds and feature masks and download high-resolution satellite imagery for the AOIs. These data will then be preprocessed to create a holistic set of training data to be used to train a machine learning model.

Given the potential bulk of training data and the time required to train a machine learning model, training data preprocessing and model training tasks will be performed by the system asynchronously on the backend servers as opposed to within the web UI. The user must be able to monitor the status and progress of these processes.

3.3.4. Review and Adjust Machine Learning Performance

After the model has been trained, the model will be automatically validated against the dataset. The user will be returned the model performance within the web UI. If model performance is not sufficient, the user should be able to retrain the model with adjusted parameters (e.g. increased training iterations) or an increased number of training examples.

3.3.5. Predict Features

After a model has been trained and optionally adjusted, the user should be able to select the machine learning model to be used and one or more AOIs for the machine learning model to detect and digitize features within. The user should be able to select one or more rectangular AOIs and pass the AOIs to the system. The system will consume the AOI bounds and download high-resolution satellite imagery for the AOIs. The system will iterate through the imagery and detect likely instances of the type of object to be detected. The system will then convert these predictions into geospatial objects. These digitized objects will be returned to the user for review via the web UI.

3.3.6. Commit Feature to Database

As there may be variability in model prediction performance, it is required that users perform a final validation step within the system workflow to validate object predictions returned by the machine learning model. Following prediction of objects from new imagery, the user must be able to review and select features to be committed to a geodatabase.

3.4. Design Principles and Choices

The system architecture of the solution can be broken down into the following core components. The technologies chosen for the respective component and the rationale for this

choice are given below. A detailed description of the implementation of the below components may be found in section 4.1.

3.4.1. Platform – Web Application

To provide the rich user experience, scalable architecture, and rapid development this project required, a web application was the appropriate platform. Thick client applications are inherently operating system dependent, rely on the user's own hardware, and require much more extensive debugging and configuration to ensure a consistent end user experience. This application was required to be deployed via the web and, as this is a technical application, no efforts were made to accommodate mobile browsers. Similarly, no efforts were made to support older or niche browsers – only the most recent releases of Chrome, Firefox, and Microsoft Edge browsers were supported.

3.4.2. System Infrastructure – Microsoft Azure

Cloud hosting was the most logical choice for this system as it allowed for the simple and rapid creation and configuration of all required backend components including: a web application server, web application framework, Python web server, and a database server. Servers can be created and managed via a simple, intuitive web portal with very little overhead. Microsoft Azure was chosen over Amazon Web Services as Azure is more tightly integrated with the primary programing framework of the system, Microsoft .NET.

3.4.3. Web Application Framework – Microsoft .NET C#, Node.js

Microsoft .NET was chosen as the primary framework for serving the web application. .NET is a robust application framework with all of the features required for enterprise-class application development. The Visual Studio integrated development environment (IDE) required

to develop in .NET includes tightly integrated source control and cloud hosting within the Azure cloud environment.

Node.js was used exclusively to run open source geospatial packages that are only available for the Node.js framework. Node.js is an extremely popular open source application framework with a large installed user base.

3.4.4. Web User Interface – JavaScript, Leaflet

JavaScript is the de facto standard for developing interactive web user interfaces. Leaflet is a robust and extensible JavaScript web mapping library with a bulk of the functionality required prebuilt. For additional functionality, there is a large community of developers that create plugins and any additional functionality can be added through custom code.

3.4.5. Imagery – Mapbox

The online mapping platform Mapbox was used to source imagery used to train the machine learning model. Mapbox collates imagery datasets from a number of providers and makes these data available through a web API. For the lowest resolution imagery (zoom levels 0-8), de-clouded imagery from the NASA MODIS satellite program are provided. For intermediate zoom levels (9-12), Landsat 5 and Landsat 7 imagery are used. For the highest resolution imagery (zoom levels 13 and greater), various sources are used dependent upon area (e.g. Digital Globe, NAIP, open source). All imagery provided by Mapbox through the API is color-corrected, "optimized," and compiled into a single source (Mapbox n.d.).

The imagery provided through the API was of a generally high quality for most areas. Per the Mapbox terms of service, tracing geospatial vector objects from imagery provided by the service is permissible for non-commercial purposes (Mapbox 2018). As this system was, in

effect, tracing the imagery through an automated means, Mapbox was determined to be the most suitable source of satellite imagery.

3.4.6. Machine Learning Framework – Python

Python and R are the two standard-bearers in the machine learning space. Judging from the existent literature, industry trends, and the open source community, Python is the more popular language for machine learning applications. On the popular code sharing website GitHub, Python was the second most popular language of 2017 accounting for approximately 17% of all code contributions to the site. R accounted for less than .2% (GitHut n.d.). Further, some of the most popular machine learning libraries such as TensorFlow, PyTorch, scikit-learn, and Keras utilize Python making it the rational choice.

3.4.7. Machine Learning Architecture – U-Net

Selection of an appropriate machine learning architecture was perhaps the most crucial decision point in the development of the application. The first step was to determine the appropriate algorithm for satisfying the application requirements. Several different classes of image processing algorithms were reviewed including: image recognition, object detection and localization, object segmentation, and sematic segmentation.

Image recognition algorithms function by classifying imagery into semantic categories based on the content of the image. As can be seen in Figure 7, the example images are categorized into the most likely correct class as determined by the model (TensorFlow 2018). While this type of algorithm has many applications, with Google Images search and Terrapattern as prime examples, these types of algorithms were not appropriate for this system.

Figure 7 Image recognition output from the AlexNet model implemented in TensorFlow (TensorFlow 2018)

Object detection and localization algorithms were found frequently within other geospatial machine learning products, namely the Orbital Insight platform. This class of algorithm identifies and typically marks instances of classes of objects within imagery (dos Santos 2017). Having the ability to detect instance of objects and their location makes these algorithms applicable in a wider range of scenarios than pure image classification algorithms and allow for much more additional processing such as counting the number of objects and degerming the spatial relationships between objects. These algorithms have many practical applications ranging from face detection for digital cameras, video surveillance, and image search (Rey 2017). However, as can be seen in Figure 8, a bounding box surrounding an instance of an object is not sufficient for object digitization – only detection.

Figure 8 Orbital Insight demonstration of object detection and localization of airliners (Orbital Insight 2017)

For object digitization, another class of algorithms termed segmentation algorithms, were found to be the most suitable. Broadly, segmentation algorithms attempt to deconstruct an image into its component parts – e.g. separating the background from the foreground, text from a page, individual objects within a photo (Figure 9) (Shi and Malik 2000). A significantly more complex offshoot of these algorithms, semantic segmentation algorithms, go one step further and attempt to assign semantic classification to segmented objects (Figure 10) (Cremers 2012). Given this, a semantic segmentation algorithm was chosen for this system.

There are a number of semantic segmentation algorithms, varying by architecture, performance, and popularity. The Fully Convolutional Networks (FCN), DeepLab, SegNet, and U-Net architectures appeared to be the prominent in the existent literature as well as within the machine learning community, as judged by the number of blog posts and repositories on the code sharing website GitHub. Each of these architectures was evaluated on several key criteria including existent documentation, active open source projects on GitHub (Table 1), and the open source community consensus regarding computational intensity, performance, and suitability for

small training datasets. An indication of the community consensus regarding these architectures was gathered, in part, from information on the data science competition website Kaggle. Four separate competitions and their respective submissions were reviewed. The objective of these competitions ranged from segmenting vehicles from a background scene, segmenting nerves from ultrasound imagery, segmenting cell nuclei, and segmenting multispectral satellite imagery. Within each of these competitions, a substantial amount of submissions utilized the U-Net architecture (Kaggle 2018).

Table 1 Number of Python projects on GitHub by architecture (April 2018)

Architecture	Number of Python Projects on GitHub
U-Net	662
FCN	500
SegNet	91
DeepLab	84

As such, the U-Net neural network architecture was chosen as the best semantic segmentation solution for this system. As U-Net was originally developed to address biomedical use cases having limited training data (such as electron microscope images of human nerve cells), the architecture was optimized to perform well with a small number of training examples. In one example given by the authors, U-Net achieved satisfactory segmentation performance using only 30 training images (Ronneberger, Fischer and Brox 2015).

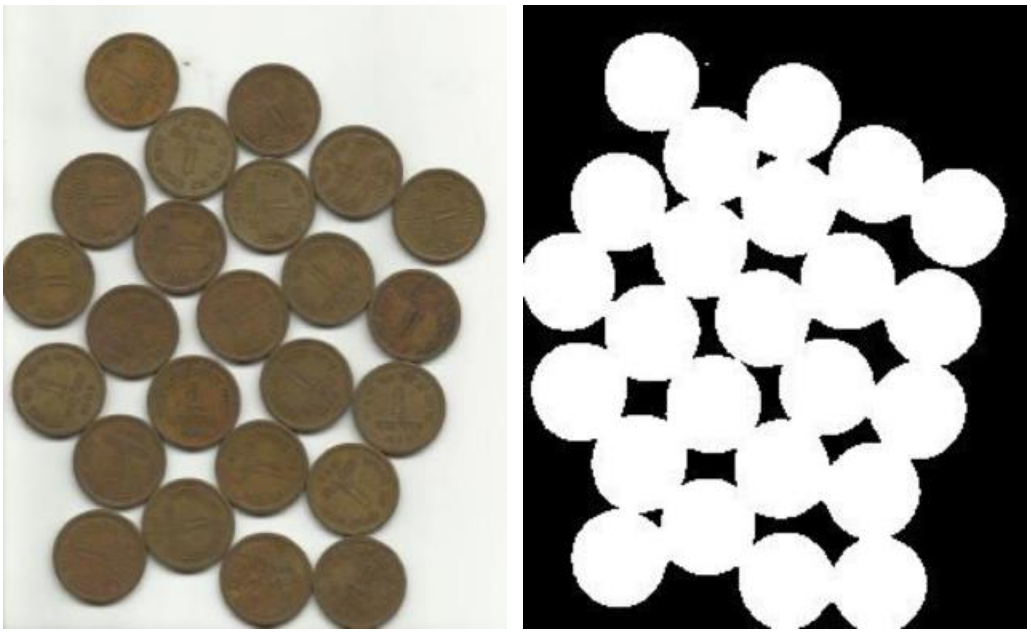


Figure 9 Image segmentation input (left) and output (right) from OpenCV (OpenCV n.d.)

Figure 10 Semantic segmentation input (left) and classified output (right) (Cremers 2012)

3.4.8. Python Web Framework – Flask

Flask is the de facto standard Python web framework for small to medium sized projects. Flask allows Python code to be deployed within a web environment and allows for the rapid creation of APIs for passing data to and receiving data from Python scripts running on a server.

3.4.9. Geodatabase – Microsoft SQL Server

For the scope of this system, geospatial database functions were not explicitly required. All geospatial processing occurred either within web UI, the .NET and Node.js web applications, or Python code. As such, it was sufficient to store geospatial objects in a standard Structured Query Language (SQL) database in the open source GeoJSON format. GeoJSON is represented as a specially formed textual string and can be stored in a database within a standard character field of a table.

Standard relational SQL databases as well as object-oriented and NoSQL databases were considered for use within this system. Given that all data stored within the system is inherently textual in nature, a standard SQL database was deemed to be sufficient for the purposes of this system. Due to tight integration with .NET, Visual Studio, and Azure, the Microsoft SQL Server 2017 relational database management system (RDBMS) was used as the database for this system.

Chapter 4 Components of the Final Application

Following requirements definition and the selection of core architectural components, the development of the application commenced. The system was designed as a wholly online user experience, with no software installation or configuration required by the end user. All system infrastructure, including servers and databases, was deployed on the Microsoft Azure cloud platform. All JavaScript, .NET, and Node.js development for the system was done within Microsoft's Visual Studio 2017 IDE on Windows 10. Due to certain Python package operating system dependencies, all Python code was developed within Microsoft's Visual Studio Code IDE on the Ubuntu 16.04 Linux distribution. All source control was managed via Git on Visual Studio Team Services.

4.1. System Infrastructure and Design

The system was composed of five primary components: a .NET web application, a Node.js API, a machine learning model with data pre- and post-processing pipelines implemented in Python, a Flask Python API, and a SQL Server database instance. All of these services were deployed on virtual servers within the Azure cloud environment. The following describe the high-level functionality and implementation of these components. Detailed processing flows and the technical development of each respective component may be found in sections 4.3-4.8.

4.1.1. .NET Web Application

The .NET web application provided the user interface and interactivity of the system. Further, all requests to other web services were initiated from the web application. Within Visual Studio, the ASP.NET MVC framework template was chosen due to this specific framework's

suitability for handling both client-side and server-side processing. That is, ASP.NET MVC excels at both serving up both rich JavaScript-enabled web pages as well as performing more intensive processing on the server itself.

All client-side web pages were designed using the Bootstrap front-end framework with a large number of JavaScript libraries used to provide interactivity within UI. Bootstrap is a popular framework of HTML elements, Cascading Style Sheets (CSS), and JavaScript which gives developers the ability to rapidly create attractive user-friendly web pages using prebuilt styled layouts and elements such as modal windows, form components, and the like. The most recent release of Bootstrap, Bootstrap 4, was used during development. To provide additional visual styling, the Material Design for Bootstrap front-end framework was included along with the standard Bootstrap 4 assets.

For mapping and geospatial functionality within the UI, the Leaflet JavaScript library was used along with a large number of auxiliary plugins. Leaflet is an open source mapping framework with an active online community of users, which proved invaluable when tailoring the product to this project's specific needs. In addition to the base Leaflet library, in excess of ten Leaflet plugins were used to provide additional functionality within the UI. The functionality of these plugins included the addition of geocoding services, polygon tools for working with AOIs, and extensible buttons, among others.

The web application was deployed within Microsoft Azure as an App Service within the Microsoft Azure Portal. Through the use of an App Service, all of the underlying infrastructure required to serve the web page was provided by the Azure platform. There was no need to create a web server for hosting the application, no need to register a URL domain, nor any of the other server setup and configuration tasks. The Azure platform allowed for the web application to be

published to the App Service wholly through the development IDE, Microsoft Visual Studio 2017.

4.1.2. Node.js API

For manipulating tile data for export, the open source tilebelt Node.js package was required. Tilebelt is provided by the web mapping company Mapbox as a set of tools for working with WMTS tiles. This library allows a developer to convert tile bounds to GeoJSON, to request WMTS tiles within a given bounding box, to request a tile for a given point, and a number of other useful features. Tilebelt is only offered on the Node.js framework, and no similar tools written in JavaScript or C# could be located.

To utilize the functionality of tilebelt, a Node.js web service was created as a separate project within Visual Studio. The Node.js service acts as a simple API to accept requests from the web UI to perform a small range of tile functions. Requests were passed to the API and returned in the form of JavaScript Object Notation (JSON) objects. The Node.js API was published as an App Service within Microsoft Azure just as the .NET web application.

4.1.3. Python Machine Learning Model, Data Preprocessing, and Post-processing

The U-Net semantic segmentation machine learning model, with requisite pre- and post-processing pipelines, was implemented in Python to provide the core training and prediction workflows of the application. Data pre- and post-processing functionality required the use of a large number of auxiliary Python libraries for transforming and manipulating data into a format and structure suitable for consumption for the U-Net model. As with all semantic segmentation models, U-Net requires two primary inputs – training labels and training data, both in the form of image files. The Rasterio, Geospatial Data Abstraction Library (GDAL), and Python Imaging Library (PIL) Python libraries were used to transform geospatial OSM features selected by the

user into rasterized image files. Python's native urllib URL library was used to download satellite imagery corresponding to the training labels to be used as training data for the model. For the data post-processing and exporting pipeline, Rasterio, GDAL, and PIL were used to essentially invert the data preprocessing functions – transforming rasterized image files output by the machine learning model into geospatial features.

An open source PyTorch implementation of U-Net developed by a GitHub contributor named Tuatini Godard was used as the foundation of the machine learning model itself (Godard 2017). The model was originally developed for use in a semantic segmentation challenge on Kaggle, which tasked contributors with segmenting images of vehicles from a photo studio background (Figure 11). This PyTorch implementation was chosen over other implementations in Keras, Tensorflow, and other machine learning libraries due to its relative speed in training the model as well as the number of helper functions (such the ability to visualize model training and performance, image manipulation and augmentation functions, and generic data loading and export functions). These factors greatly decreased development time, removing the need to code these features and the implementation of U-Net. However, a nontrivial amount of development was required to reengineer and tune the architecture to accommodate this application's vastly different use case.

The one significant drawback to implementing any neural network architecture in PyTorch was the lack of support for Nvidia's CUDA graphical processing unit (GPU) platform on Microsoft Windows. The CUDA platform allows frameworks such as PyTorch to use the power of GPUs to train neural networks. While performance differences between training on a central processing unit (CPU) versus a GPU vary based on the application, benchmarking has shown GPUs to perform upwards of twenty times faster when training object detection and

localization neural networks as compared to CPUs (Lawrence, et al. 2017). As such, to achieve acceptable performance when training the model all development and testing of the U-Net model was performed on a Linux Ubuntu 16.04 personal computer equipped with an Nvidia GeForce GTX 970 GPU with 4GB of GDDR5 graphics memory.

Figure 11 Input image (left) and segmented output (right) of Godard's original model (Godard 2017)

4.1.4. Flask API

To allow data to be passed from the web UI to the backend Python code, a Flask API was created as an Azure App Service. The creation of the Flask API was relatively trivial when compared to the development of the machine learning model itself. The Python functions created for data pre- and post-processing and the machine learning model where developed to accept a number of parameters within their function signatures, allowing for a simple migration to a Flask deployed API. The parameters for a given function simply had to be mapped to the appropriate routing configuration within Flask to allow for REST requests sent to the Flask API from the web UI to be passed to the appropriate Python function.

4.1.5. SQL Server Database

A SQL Server database instance was used to store data generated by the system. Machine learning model parameters were stored within the Model table, using the model's name as a unique identifier. Similarly, the Prediction table was used to store prediction requests submitted by the user. GeoJSON objects generated within the Node.js API and Python code were stored within a GeoJSON table along with the type of the data, be it training data, model validation output, or model predictions. Records within the GeoJSON tables were related via foreign key relationships to the Model and Prediction tables. Further, the status of data pre and post-processing, model training, and predictions tasks were stored within the Model and Prediction tables respectively allowing the user to monitor these processes as they progressed within the Python code.

4.2. External APIs

Several open source and commercial APIs were used within the system. These were used to display web maps within the UI, perform geocode lookups for the convenience of the user, and to source vector data and satellite imagery required for training machine learning models.

4.2.1. OSM Tiles

The OSM basemap was used for general reference and identification of features within the web UI (Figure 12). The API was freely available from an OSM tile server, offered with an open source license. JavaScript and Leaflet were used to incorporate the WMTS into the UI.

While the integrity of information captured within the dataset is generally considered to be high-quality, there was a large reporting bias apparent within the OSM feature data that was used to render the basemap. As all OSM data are volunteer geographic information (VGI), features surrounding population centers, transportation networks, and recreational areas within

the Western world were better represented within the data than areas that are presumably of less interest to those creating the data, such as large swathes of the developing world and wilderness areas that see little human activity.

Figure 12 OSM tiles, Anchorage, Alaska

4.2.2. Esri World Topographic Tiles

The Esri topographical basemap was used for general reference and identification of features within the web UI (Figure 13). The API was freely available from an Esri tile server, offered with a proprietary license limiting the use of the data. Downloading of the data and offline storage for processing and analysis was not permitted. JavaScript and Leaflet were used to incorporate the WMTS into the UI.

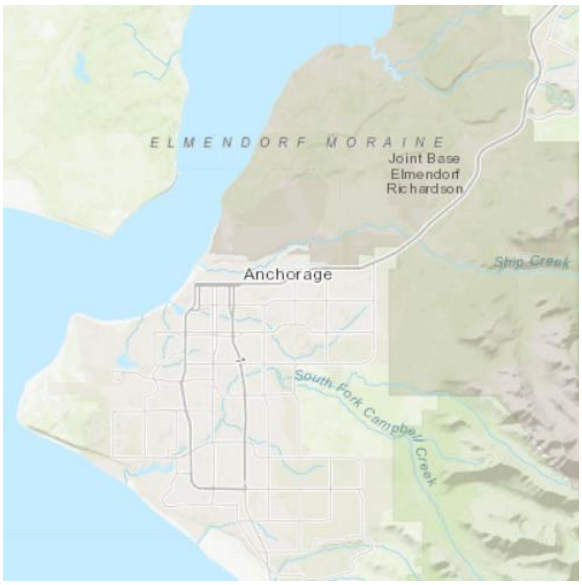


Figure 13 Esri World Topographic tiles, Anchorage, Alaska

4.2.3. Esri World Imagery Tiles

The Esri imagery basemap was used for general reference and identification of features within the web UI (Figure 14). The API was freely available from an Esri tile server, offered with a proprietary license limiting the use of the data. Downloading of the data and offline storage for processing and analysis was not permitted. JavaScript and Leaflet were used to incorporate the WMTS into the UI.

The imagery was of a very high quality throughout the entirety of the zoom range. The imagery was a mosaic, compiled from different sources. Dependent upon location, imagery was available at zoom levels 0-19 (corresponding to scales of 1:500 million to 1:1,000, respectively).

Figure 14 Esri World Imagery tiles, Anchorage, Alaska

4.2.4. Mapbox Satellite Tiles

The Mapbox satellite imagery basemap was used to display imagery for user selected AOI within web UI (Figure 15). The API was available from a Mapbox server offered with a proprietary license limiting the use of the data. However, for non-commercial purpose caching and tracing the data was permissible under the Mapbox terms of service (Mapbox 2018). A Mapbox subscription was required to access data through the API. Subscriptions are offered at different pricing tiers, dependent upon the number of required requests. The free pricing tier, allowing for 50,000 map views per month, was sufficient for the scope of this system. JavaScript and Leaflet were used to incorporate the WMTS into the UI.

The imagery was of a very high quality throughout the entirety of the zoom range. The imagery was a mosaic, compiled from different sources. Dependent upon location, imagery was available at zoom levels 0-19 (corresponding to scales of 1:500 million to 1:1,000, respectively).

Figure 15 Mapbox Satellite tiles, Anchorage, Alaska

4.2.5. OSM Features

The OSM features may be selected by the user within the web UI (Figure 16). Features were further used to train machine learning models. The API was freely available from an OSM server, offered with an open source license. Data was provided in the non-standard OSM XML format via the JavaScript OSM Overpass API. The open source osmtogeojson JavaScript module was used to convert OSM XML data to standard GeoJSON within the code of the web user interface and the Node.js web service.

OSM feature data attributes were defined using key-value pairs. Key-value pairs can be created and assigned arbitrarily by the contributor submitting the data, but contributors do tend to hold to certain standard conventions. The primary attributes of interest for this project were the "primary feature" tags which describe the object that is being represented. Examples of primary feature key-value pairs are: "building: commercial," "shop: electronics," "natural: glacier," and "natural: water." Additional metadata key-value pairs, such as the source imagery from which the feature was digitized and when the data were created was often included for reference.

While the integrity of information captured within the dataset is generally considered to be of high-quality, there was a large reporting bias apparent within the OSM feature data. As all OSM data is VGI, data surrounding population centers, transportation networks, and recreational areas within the Western world were better represented within the data than areas that are presumably of less interest to those creating the data, such as large swathes of the developing world and wilderness areas that see little human activity.

When compared to proprietary datasets (such as Google Maps or Bing Maps) OSM contained significantly fewer recorded features in many areas (Figure 17). However, OSM's data was freely available for download, offline analysis, and use in derivative works – which proprietary feature datasets were not.

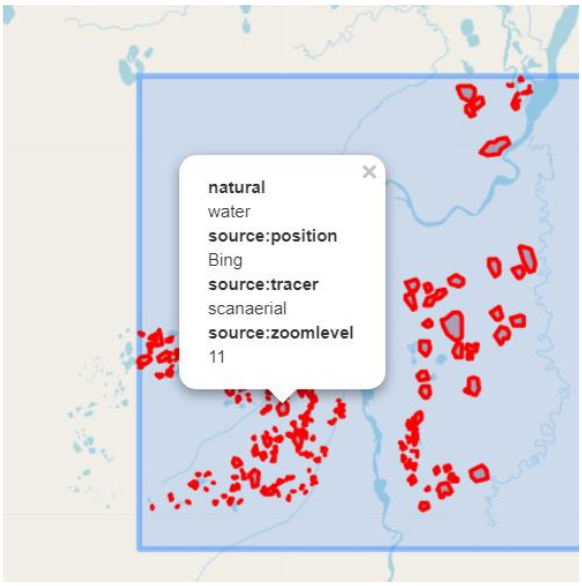


Figure 16 OSM features, North Slope, Alaska

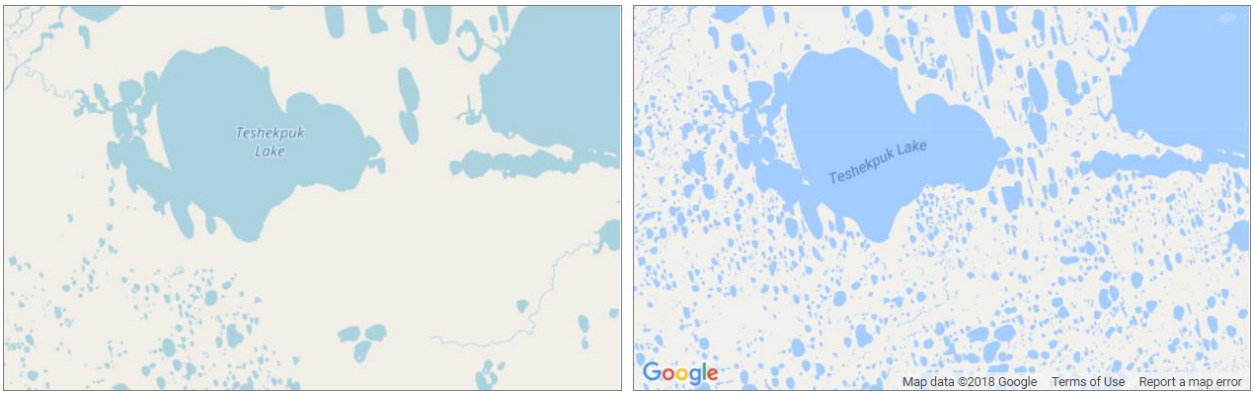


Figure 17 OSM feature data (left) compared against Google Maps feature data (top right) (GEOFABRIK 2017)

4.2.6. Bing Locations API

The Bing Locations geocoding API was used within the web UI to allow users to quickly search for and have the map centered on a queried location. The API was available from a Microsoft server using a free API key. There was a limited, but sufficient number of requests permitted from a single free API key. The open source JavaScript leaflet-control-geocoder package was used to integrate the Bing Locations API into the web UI.

4.3. High-Level Processing Flow

The usage of the application can be split into two primary use cases – training and validation, and object prediction. High level descriptions of these use cases are provided within this section. A more granular description of system processes and data inputs and outputs are detailed in sections 4.5-4.8

As shown in Figure 18, the training and validation use cases encompasses all processing flows required to generate a fully trained model. The workflow begins with the selection of training data by the user and the entry of select model training parameters. Following the submission of these data and parameters to the system, a large amount of data preprocessing is performed by the system prior to model training. The machine learning model is then trained

using this processed data and a select number of geospatial feature predictions are returned to the user for evaluation within the web UI. If the user finds the performance of the model to be satisfactory, the user could then predict instances of similar features within new imagery. If performance is not adequate, the user could select larger amounts of training data, different training data, or adjust the model parameters.

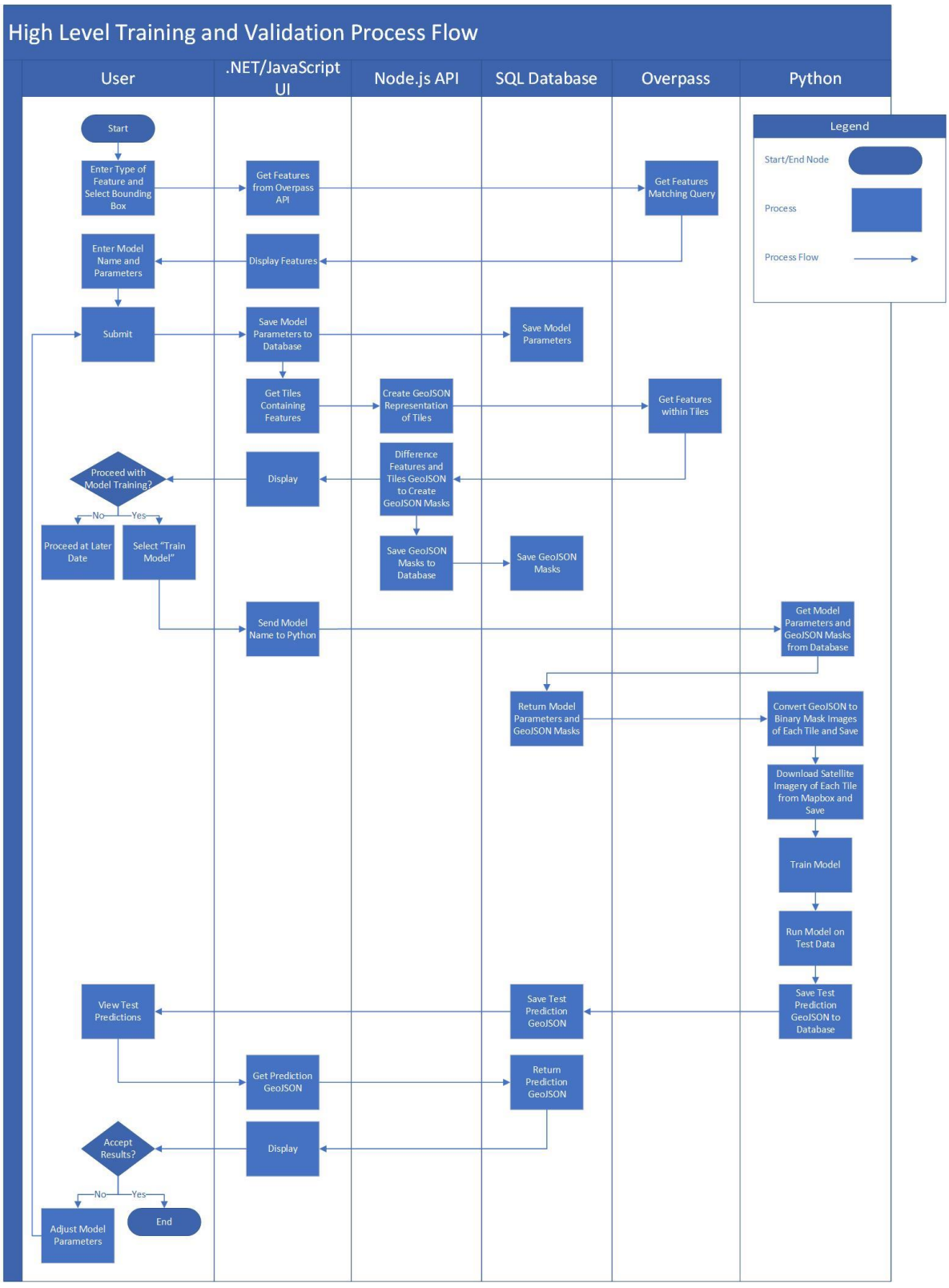


Figure 18 High Level Training and Validation Process Flow

Once one or models have been trained via the training and validation process, the user is able to predict objects within new satellite imagery through the prediction workflow (Figure 19). Within this workflow, a user would begin by selecting an AOI bounding box of new imagery from within which the user wishes to detect instances of a given class of geospatial feature. The user would then select an appropriate model that had been trained on the same class of feature. Following the submission of these parameters to the system, imagery from within the AOI is downloaded and the trained model is used to predict occurrences of the type of feature on which the model was trained. These predictions are then returned to the user to be validated against ground truth imagery or other basemaps within the web UI.

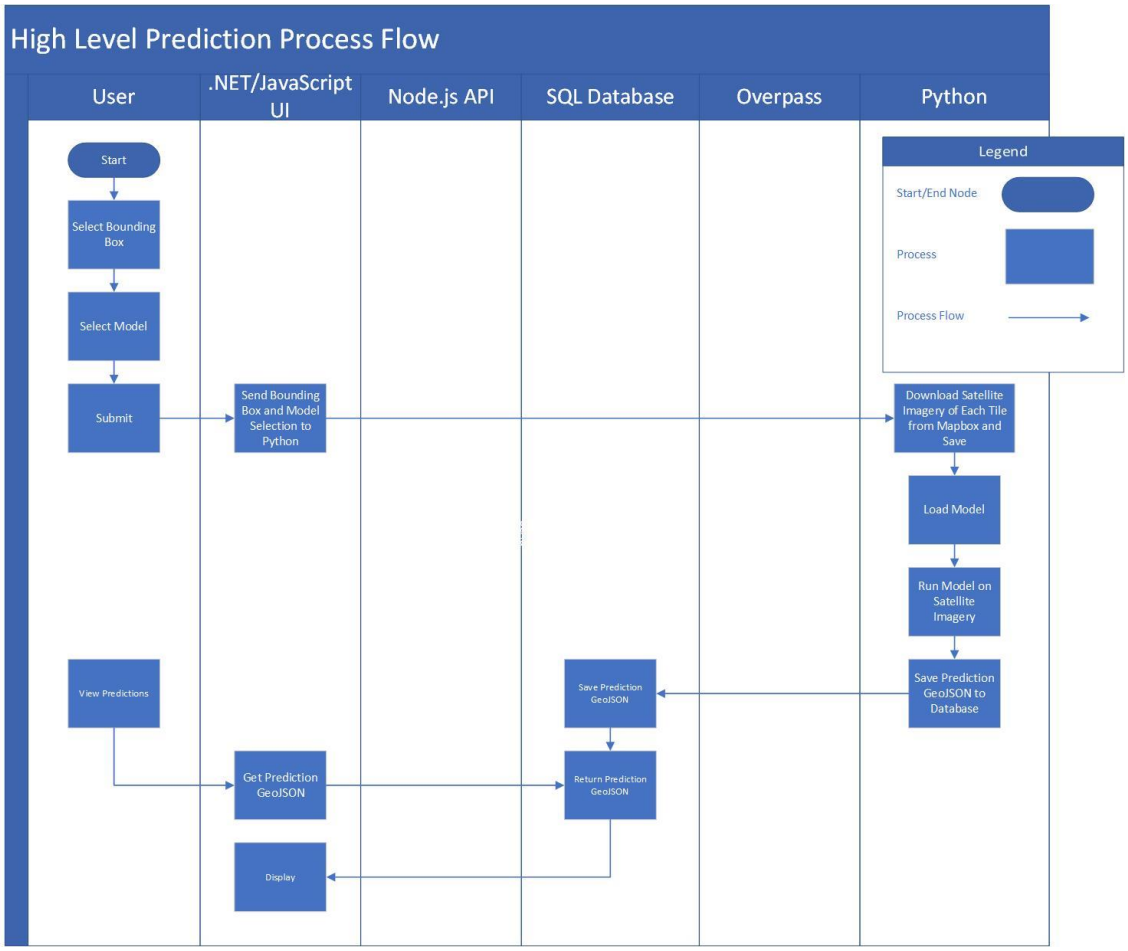


Figure 19 High Level Prediction Process Flow

4.4. Data Model

The data model for the system consisted of three primary tables: Model, GeoJSON, and Prediction (Figure 20). The Model table contained the basic data describing the model along with the most recent status of model processing and a timestamp of the most recent status. GeoJSON objects related to models were stored within the GeoJSON table via a model's unique name identifier. GeoJSON objects were stored as a text string within a character field in the table. The Type field described the type of the GeoJSON record within the system. There were four distinct type identifiers:

1. Training Mask
2. Validation Prediction
3. User Prediction
4. Committed User Prediction

The Training Mask type was used to denote GeoJSON mask records that were generated when a user selects training data within the web UI. The Validation Prediction type was used to denote object predictions automatically generated following model training. The User Prediction type was used to denote object predictions generated by the user from a trained model. The Committed User Prediction type was used to denote object predictions that the user has reviewed against ground truth imagery and wished to mark as "validated" within the system.

The Prediction table was used to store user prediction requests. A prediction was related to the model used for the prediction via the model's unique name identifier. The prediction threshold supplied by the user, the most recent status of prediction processing and a timestamp of the most recent status were also included. A prediction is related all GeoJSON objects created

during a user requested prediction via a foreign key relationship with the GeoJSON table using a prediction's unique ID field.

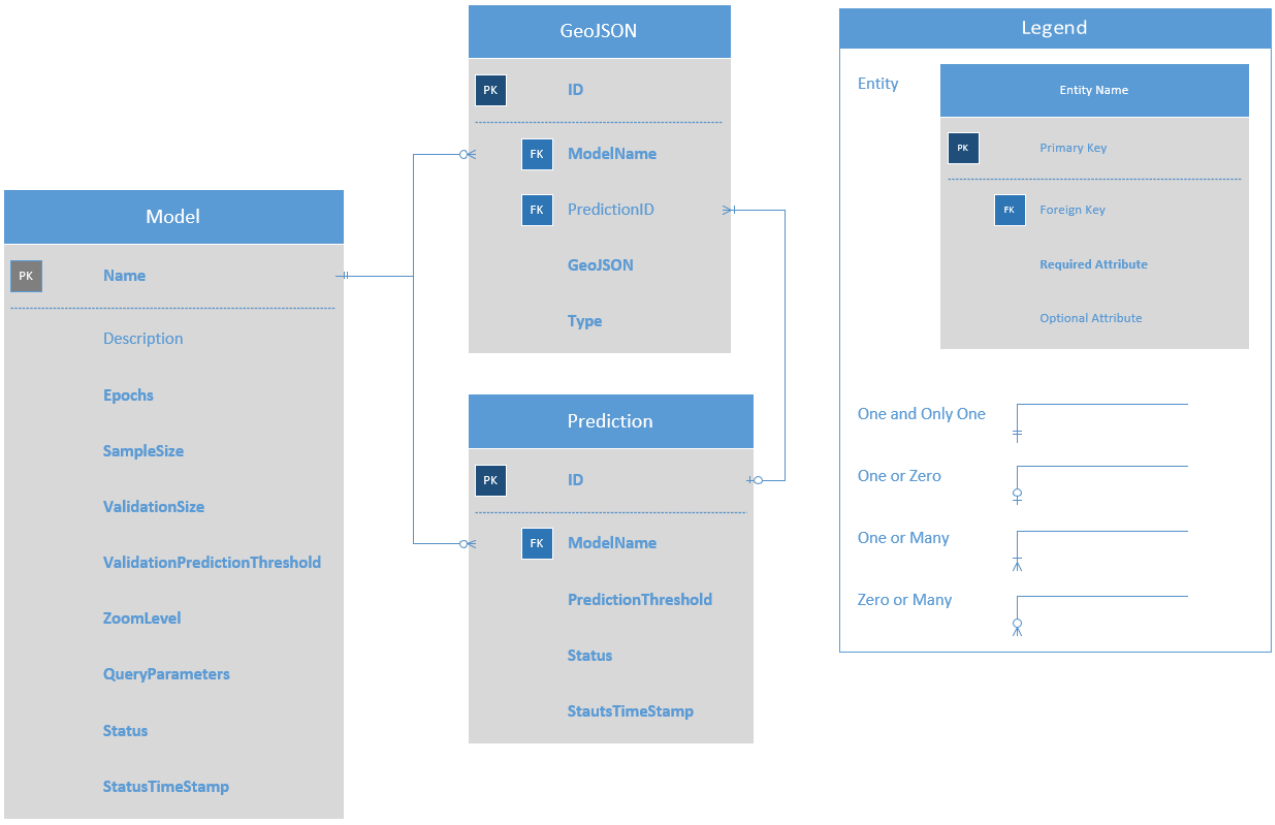


Figure 20 Data model

4.5. Training Data Sourcing and Preprocessing

The two primary inputs for the training the system were (1) OSM feature data representing geographical features and (2) true-color raster satellite imagery from Mapbox corresponding to the vector data. For sourcing both of these data, the user was provided with an easy to use web UI. However, extensive processing was required to source and further transform these data into a format that was consumable by the machine learning model.

4.5.1. Sourcing of OSM Feature Data

As discussed previously, OSM feature data were source from the Overpass API through the web UI. As opposed to the standard OSM API, the Overpass API is optimized for querying and viewing data rather than editing and contributing features. For the purposes of this system, the Overpass API required a specially formatted query providing both the types of features to be returned and an AOI in the form of a bounding box within which the API would search for features. The Overpass API was exposed as a representational state transfer (REST) web service. Query parameters are appended to the Uniform Resource Locator (URL) in the below format.

overpass-api.de/api/interpreter?data=[out:json][timeout:15];(*data* ["*key*" = "*value*"](*bounding box coordinates*);(*further query parameters);*); out body;>;out skel qt;

To source feature data, datatype-key-value tuples and a bounding box supplied by the user were used to query the Overpass API through JavaScript in the web UI. Data were returned in Overpass XML syntax. The open source osmtogeojson JavaScript package was then used to transform these data into standard GeoJSON to be displayed using Leaflet within the web UI.

When the user had selected their desired amount of training data, they then entered various machine learning model parameters and submitted these parameters to the system. Upon submission, the system saved the model and model parameters to a SQL database for future retrieval using C# code. The system then looped through all WMTS tiles within the user's supplied bounding box that contained any part of a returned geospatial feature using JavaScript and Leaflet, creating an array of unique WMTS x, y, z tile identifiers.

Following this, the system looped through this array of tile identifiers and passed the previously supplied query parameters and each tile identifier to a system internal Node.js API. The Node.js code then created a geospatial polygon object representation of the tile using tilebelt and queried the Overpass API using the tile's bounding box and the query parameters to find all

matching objects within that tile. A difference operation was used to create a mask of the features within the tile (Figure 21). For line data, an arbitrary 25-foot buffer operation was applied to allow these features to be unmasked within the resultant output (Figure 22). The GeoJSON mask was then simultaneously returned to the web UI to be displayed to the user and saved to a SQL database along with a relation to the model that the mask would be used to train. These data within the SQL database are later processed within Python code to create a binary mask image from the GeoJSON.

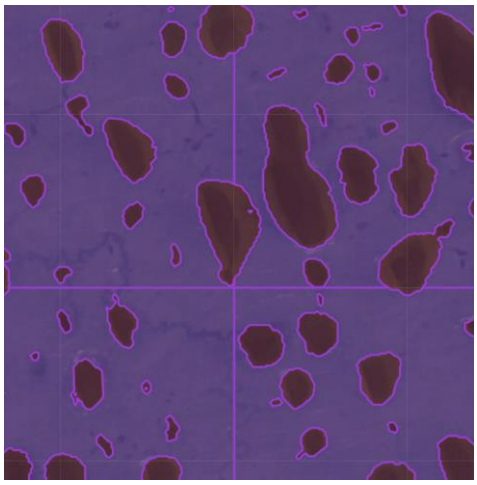


Figure 21 GeoJSON masks of lakes on Alaska's North Slope

Figure 22 GeoJSON mask of streets near Memphis, TN

4.5.2. Creation of Binary Masks and Sourcing of Mapbox Satellite Imagery

When the user wished to proceed with training a model, the model name identifier was passed from the web UI to a system internal Flask Python API for additional processing and the eventual training of a machine learning model. At each step, the status of the processing operation was logged to the SQL database to allow a user to follow the processing progress through the web UI.

Within the Python code, the model name was used to query the SQL database GeoJSON table to return all GeoJSON tile masks relating to the model. Each row of GeoJSON data returned from the database was then processed. The GeoPandas Python library was used to parse and convert the GeoJSON string into a geospatial GeoSeries object. This object was then converted to a NumPy array and saved to server storage as a binary mask GIF file using the popular Matplotlib charting library and PIL image processing library. Following this, Mapbox satellite imagery was downloaded for each tile using the Mapbox REST API and saved to server storage.

Figure 23 OSM feature data converted to binary mask and corresponding Mapbox satellite imagery (OpenStreetMap 2018) (Mapbox 2018)

4.6. Machine Learning Model

The machine learning components of the system were divided into two distinct processing flows. Firstly, the model training workflow served to generate new predictive models from input OSM feature data and corresponding satellite data. This functioned by feeding this dataset, along with other parameters, into a "classifier" function that would train the neural network to predict instances of similar objects within new imagery. The second processing flow, the object prediction workflow, used a previously trained model to perform the prediction function on new imagery.

4.6.1. Model Training

Following all data sourcing and preprocessing, the U-Net neural network could be trained on the user's selected data. The main model training Python function was designed to take the unique name of a model as a function parameter. The supplied model name was then used to query the SQL database to retrieve the full set of model parameters provided by the user through the web UI. The model name, sample size, and validation size parameters were used to create an array containing the filenames and locations of the training images and masks to be used to train the model. Training image and masks were stored in a standardized structure, allowing the code to select data based on the model name parameter. Once images were loaded, manipulations were performed on the images to augment training dataset. The manipulations applied random degrees of rotation, flipped images horizontally, and applied scaling transformations. The purpose of this was twofold – this both expanded the dataset by creating additional training data from a limited set of training data and introduced an additional degree of visual variability into the dataset.

Following this a number of callback functions were utilized which were used to log the status of training, visualize model training performance, and the like. Custom functions were created to log high-level processing steps to the SQL database for retrieval through the web UI. The TensorBoard library was used create verbose logs of model training and to visualize model training performance on the development Ubuntu workstation (Figure 24). TensorBoard was extremely useful when reengineering and retuning the original model and while developing the data preprocessing pipeline as a means to quickly diagnose coding errors and potential optimizations. However due to architectural constraints, this information was not exposed to application end users.

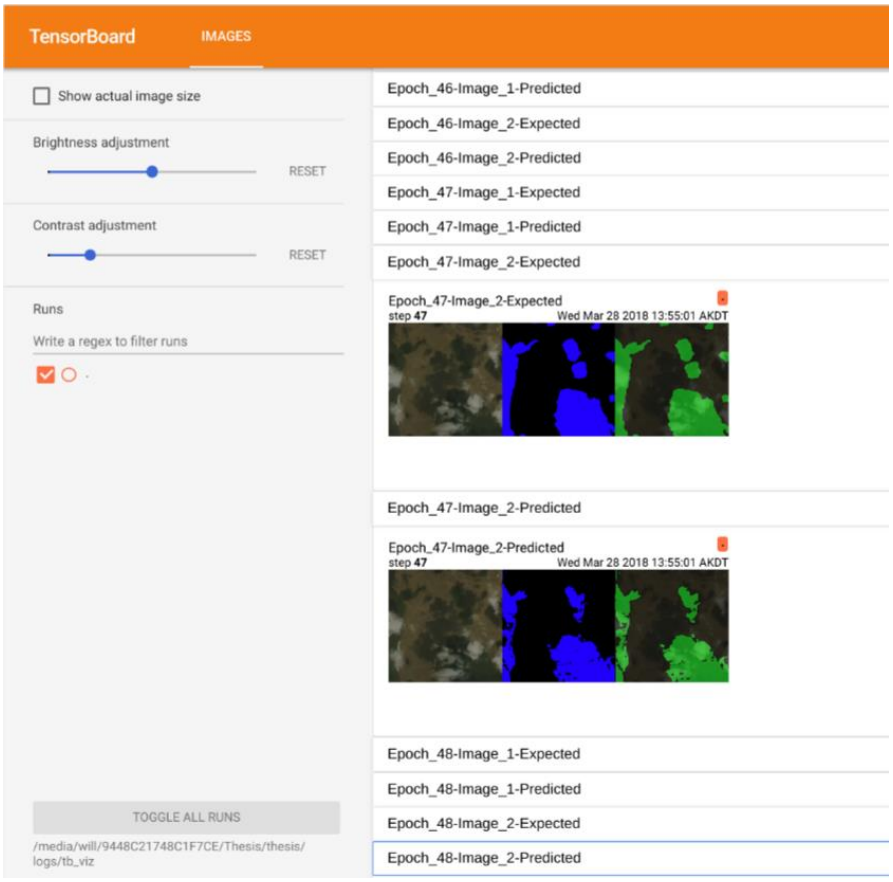
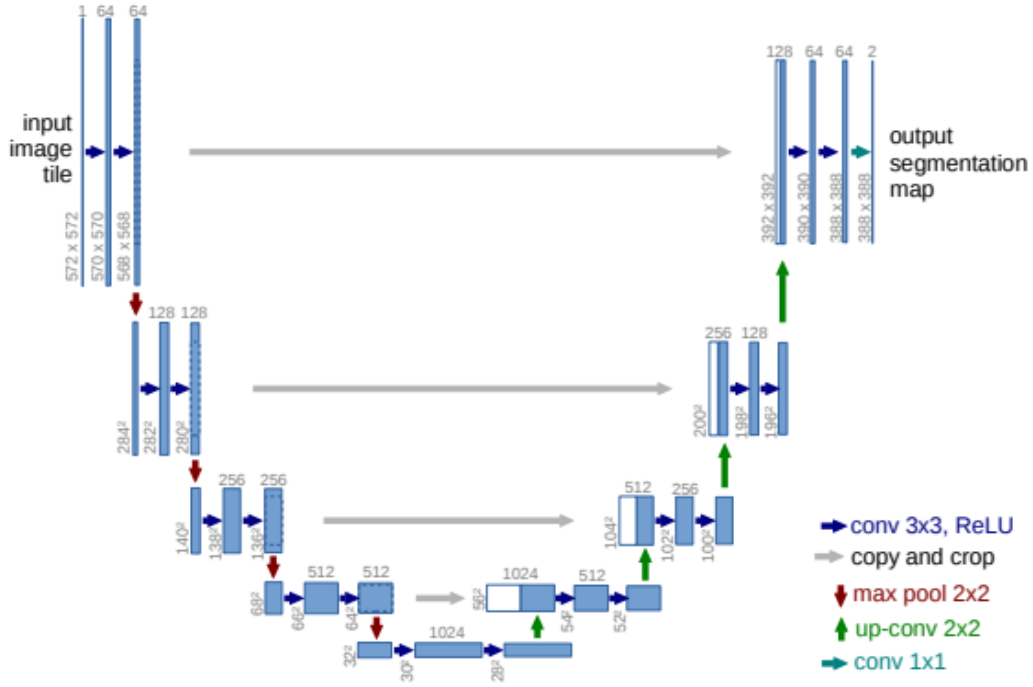


Figure 24 TensorBoard visualization of prediction performance. Top image shows expected prediction from binary masks, bottom image shows actual prediction after 47 training epochs.

Several performance variables were then defined within the code as the final input into the neural network. The Python code detected the number of CPU cores available on the host machine, if an Nvidia CUDA compatible GPU was available, and then a hardcoded "batch size" variable was assigned within the code. The batch size indicates to the neural network the number of images to process within memory concurrently; the higher the batch size, the faster model training will complete all other variables being equal. On the development machine with 4GB of graphics memory, three 512x512 pixel images could be batched without failure. This batch size could be increased if a higher end GPU was utilized.

Following assignment of these variables, a new instance of the U-Net neural network was created within the code. The neural network developed by Godard closely follows that of the original U-Net paper. While an in-depth discussion of the U-Net architecture is outside of the scope of this work, it is important to note several features of the network. Firstly, U-Net was designed as a convolutional neural network, a subclass of neural networks that have been shown to excel at image recognition and classification tasks (Hijazi, Kumar and Rowen 2015). Further, the eponymous u-shaped architecture serves to upsample the output to a high-resolution segmentation map (a binary mask of the features that were predicted), visualized on the right-hand side of Figure 25. Another interesting feature of the architecture are the "copy and crop" functions denoted by grey arrows within Figure 25. This function allows the network to use the contextual information from the corresponding higher resolution feature maps on the left-hand "contracting" path of the network (Ronneberger, Fischer and Brox 2015).



U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Figure 25 Excerpt from the original U-Net paper describing the architecture of the neural network (Ronneberger, Fischer and Brox 2015)

Following instantiation of the neural network, the optimization algorithm for use within the network was defined. Broadly, the type of optimization algorithm used serves to help minimize the prediction error within the network (Walia 2017). PyTorch has a large number of optimization algorithms defined within the torch.optim package, many of the gradient descent family. As the original U-Net authors cited the use of a gradient descent optimizer (implemented in the Caffe framework), several of these optimizers were tested (Ronneberger, Fischer and Brox 2015). Both the SGD and RMSprop PyTorch optimizers were tested extensively, with RMSprop providing the most satisfactory performance.

The training imagery, masks, callbacks, system performance variables, neural network, and user supplied epochs value were then passed to a "classifier" function which used these inputs to train a machine learning model. Broadly, the classifier functioned by passing the training imagery and masks to the neural network and training the model over a number of epochs. In each epoch, the satellite imagery and corresponding binary masks were fed into the network providing both the input data (images) and pixel-level labels (binary masks). The behavior and performance of the model was updated with each successive epoch and performance metrics and sample predictions were logged for display within TensorBoard (Figure 24).

Figure 26 Mapbox satellite imagery (left), binary mask generated from OSM data (middle), and masked satellite imagery (right) (Mapbox 2018) (OpenStreetMap 2018)

Following the training of the model through all of the given epochs, validation imagery was then passed to the model to demonstrate model performance. The validation imagery was a subset of the provided training data that was not used within model training, meaning that the model had no opportunity to "learn" the validation imagery as it was never provided with that imagery nor the corresponding binary mask. As such, the validation imagery was the least biased set of data on which to gauge the performance of the model (Brownlee 2017). To generate the validation predictions, validation imagery was input into the object prediction workflow.

4.6.2. Object Prediction

To generate object predictions from validation imagery or an AOI supplied by the user the process was very much the same. The only material difference being that validation imagery was taken as a subset of user selected training data where predictions from a new AOI required the downloading of the imagery tiles from Mapbox. For predictions from a new AOI, the Flask API was passed an array containing: the name of the trained model to be used, the prediction threshold, the unique WMTS tile identifiers of the top left and bottom right tiles in the AOI, and the zoom level of the imagery the user had selected. These values were used to download all imagery tiles within this AOI bounding box for the given zoom level.

From here both validation imagery and new satellite imagery were passed to a custom "predictor" function. This function would first load the correct trained model from server storage using the model name parameter. The predictor would then generate an array containing the filenames and locations of imagery to be processed. This array was then passed to the trained model to predict new instances of a given type of object within the new imagery. The model produced a 512x512 probability mask for each input image as a NumPy array, with values ranging from 0 to 1 for the model's percentage confidence in the correct prediction of a given pixel within the mask (Figure 27). A thresholding function was then used to filter out pixels below the user's supplied threshold value. All remaining pixels within the mask that were not filtered out were then converted to a value of 1.

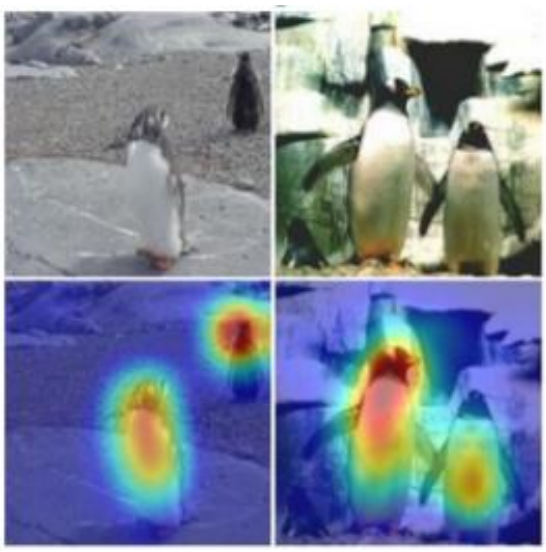


Figure 27 Example class activation map, visualizing the function of a probability mask. In this example, red values would be prediction confidences closer to 1 and blue values closer to 0. (Zhou, et al. 2016)

Following the thresholding function, the binary mask arrays were then converted to the run-length encoding (RLE) format and stored in a comma separated values (CSV) file for storage and further processing. Within the RLE format scheme, images were compressed down into a single string of numbers making the storage of arbitrarily large dataset much more convenient and manageable. After conversion, prediction RLE data was then passed to the data post-processing pipeline for georeferencing of predictions and eventual display to the user.

4.7. Date Post-processing

Following the generation of predictions from the machine learning model, the non-georeferenced prediction data was transformed into georeferenced spatial objects. As stated, the model output predictions into a compact RLE CSV file with one row of RLE data representing a single tile. Each row within the CSV file contained two columns – one column storing the tile's WMTS x, y, z identifier, the other storing an RLE string of the encoded prediction. To convert these RLE strings into spatial objects, the RLE CSV file was first opened and parsed into a

NumPy array, with each array element corresponding to a line in the RLE CSV file. Each line of the NumPy array was then processed using PIL to generate a binary image from the RLE data. Each binary image was saved to the server's local storage, using the tile's WMTS identifier as the file name.

The GDAL package was then used to create a georeferenced GeoTIFF file from each binary image. The pygeotile package was first used to create a geospatial object representation of the tile, taking the tile's WMTS identifier as parameters. From this, the tile's bounds were passed to the GDAL Translate method to produce a spatially referenced black and white GeoTIFF image of the tile. The Rasterio package was then used to create a binary "NoData" mask layer from the GeoTIFF image. This NoData mask included only predictions from the tile, those portions of the GeoTIFF that appear white within the image. Using the NoData mask, the GDAL Polygonize method was used to create a GeoJSON MultiPolygon object representation of the tile.

As the machine learning model operated as a pixel-level classifier, the GeoJSON objects created by GDAL Polygonize could be extremely complex, especially around the borders of objects. At the edges of object predictions, the classifier appeared to struggle with appropriately predicting and classifying pixels in this transitionary zone. As can be seen in Figure 28, the GeoTIFF images appeared "fuzzy" around the edges of object predictions (white), with non-classified (black) pixels interspersed throughout. When these pixels were polygonised, this was reflected within the GeoJSON objects. While this was the true reflection of the machine learning model's output, this greatly increased the file size and complexity of the GeoJSON objects – frequently increasing storage greater than tenfold.

To mitigate this issue, a number of approaches were tried. Primarily, edge detection algorithms within the OpenCV computer vision library were used in an attempt to simplify the edges of the binary masks before conversion to GeoTIFF. It proved exceedingly challenging within these algorithms to achieve shape simplification while maintaining the desired edge fidelity. Further, there was a desire to convey to the user the model's classification uncertainty along these object edges; over simplification or smoothing of these boundaries would withhold this information from the user. In the end, a "brute force" approach was used wherein all patches of classified (white) pixels less than an arbitrary size of five pixels by five pixels were removed from the GeoJSON objects. Through this technique, storage size was greatly reduced while maintaining edge fidelity and discernable model uncertainty.

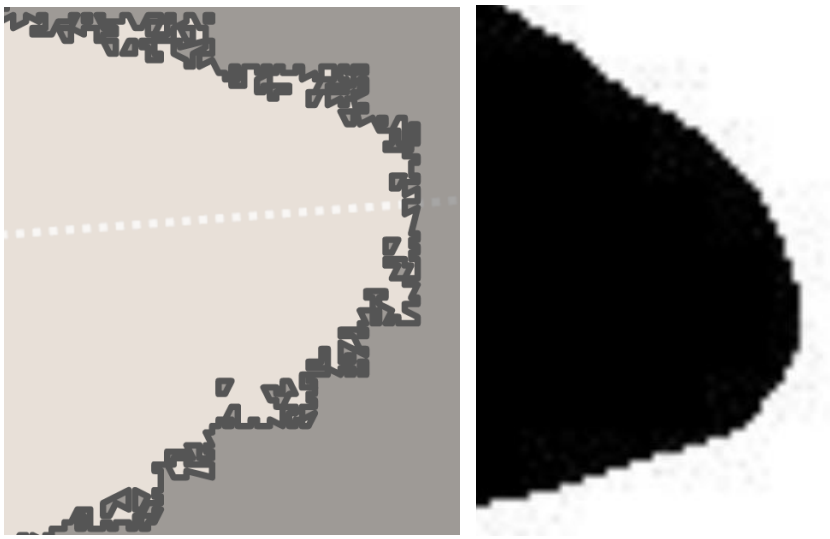


Figure 28 Example GeoJSON complexity (left) and errant pixel classification (right)

Subsequent to post-processing, all GeoJSON objects were saved to the database for display to the user within the web UI. The GeoJSON objects were related to the generating model through a model name identifier foreign key.

4.8. Prediction Validation and Storage

Following a user's prediction request through the web UI, data preprocessing, object detection, and data post-processing, the user was able to view object predictions within the web UI. Prediction data was displayed much in the same way as OSM feature data within the web UI as semi-opaque polygons overlaying a basemap. However, these data were sourced from the system's SQL database via a query of all GeoJSON data relating the selected prediction instance. Custom code was added to extend the Leaflet JavaScript package allowing users to click on features within the web UI to flag these features as "validated" (Figure 29). When a feature has been validated, the Type field of the GeoJSON record within the SQL database was updated to reflect the user's validation of that feature.

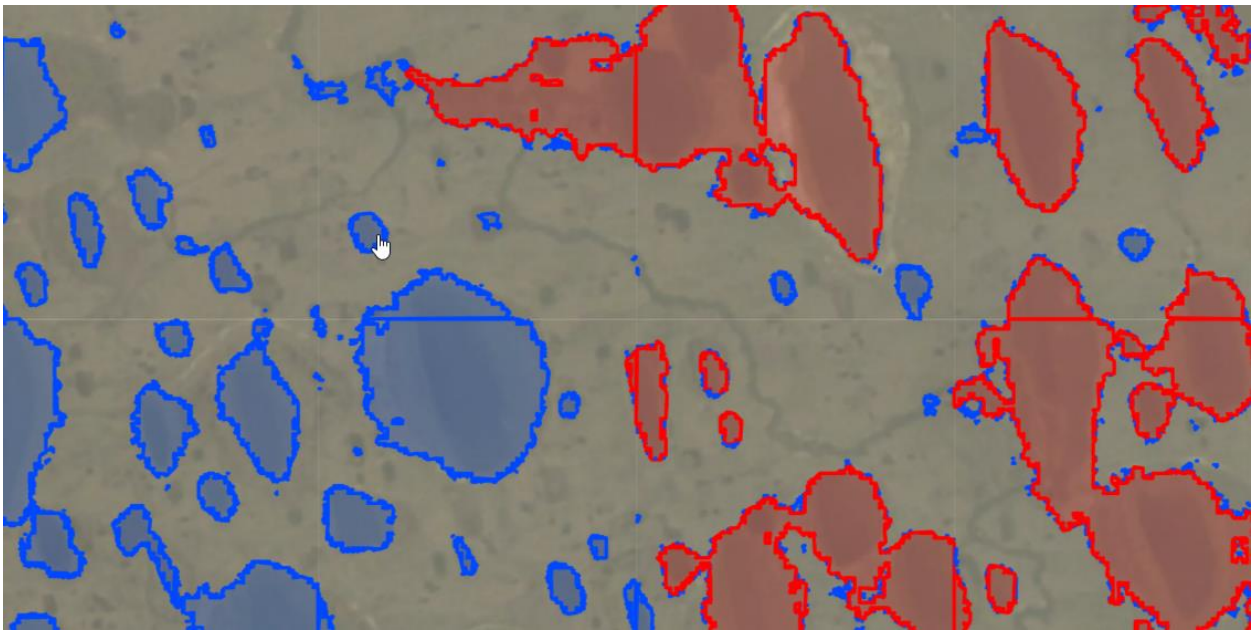


Figure 29 Predictions (blue) and validated predictions (red)

Chapter 5 Results

This chapter demonstrates the results of the project including a number of the web application UI screens, the machine learning model output, model performance, and the resultant digitized geospatial features. The reader will progress through a typical application workflow, using the web application to select training data, model creation and validation, and passing novel satellite imagery to the model to generate digitized feature output.

5.1. Application Organization and Workflow

The application was composed of four interactive webpages – Train, Admin, Review, and Detect. The Train page allowed the user to source model training data and enter model training parameters. The Detect page allowed users to select novel satellite imagery to pass to trained models to receive object detection predictions. The Admin page allowed the user to view the processing status of models that were generated by the system, adjust model training parameters to retrain a model, and to access the Review page wherein the user could view training data used to train models, model prediction performance, and predictions generated from models. The Review page also allowed the user to validate object detection instances from predictions against ground truth imagery and flag these validated instances within the database. The user workflow, delineated by webpage, is given in Figure 30. Detailed descriptions of the workflow processes are provided in sections 5.2-5.4.

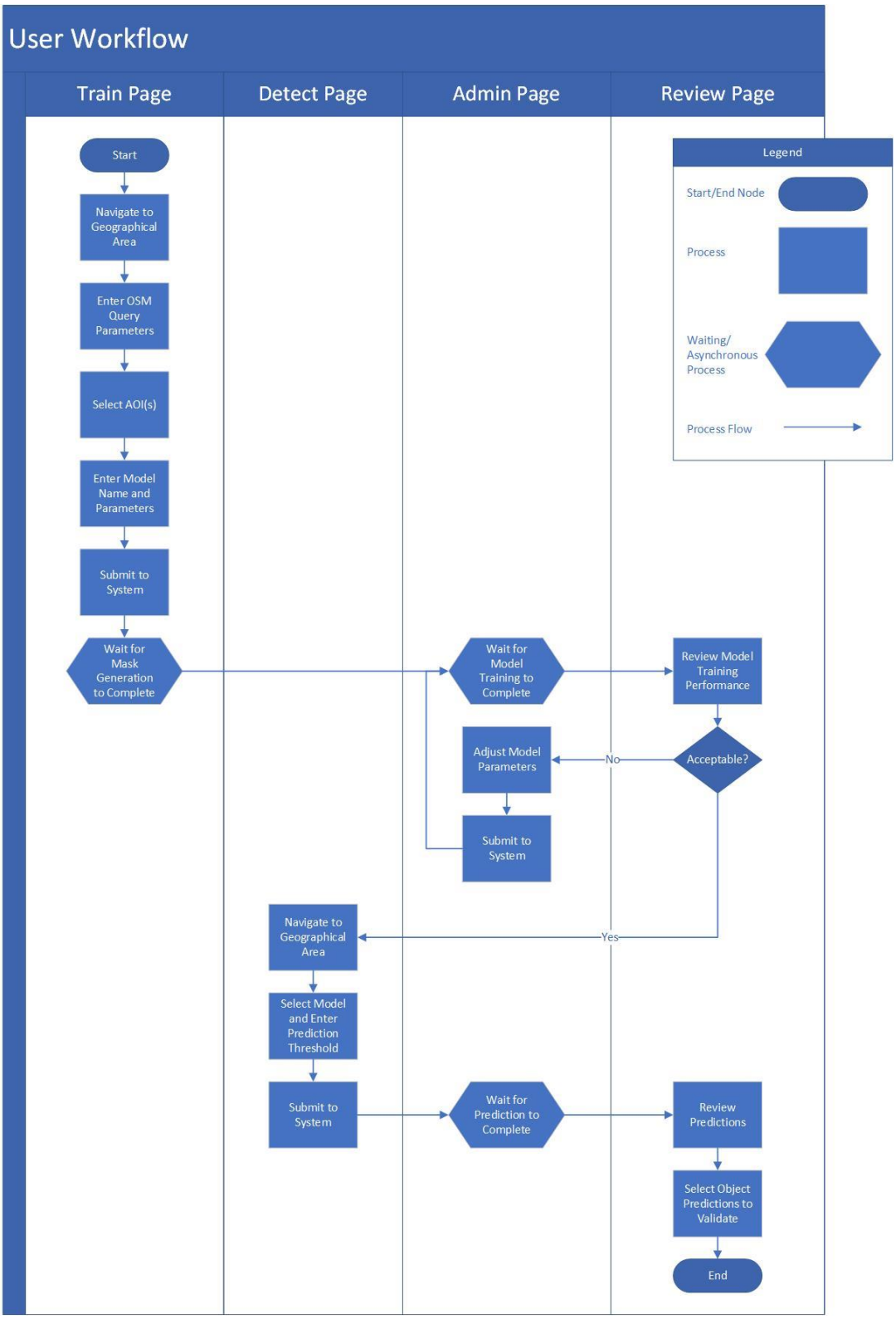


Figure 30 User workflow of the application by webpage

5.2. Selection of Training Data and Model Parameters

The Train page served as the main landing page of the application and allowed the user to query OSM data, view satellite imagery overlain with OSM data, select appropriate OSM data for model training, provide model parameters, and initiate data preprocessing and model training. The below describes the UI, user workflow, and data processing flows of the Train page.

5.2.1. General Navigation

The Train page focuses around a full window web map and the user is given a number of interactive navigation and visualization options (Figure 31). Users may pan and zoom, select a number of overlay layers for display, change the opacity of overlay layers, and perform a geocode search to locate a point of interest on the map (Figure 32).

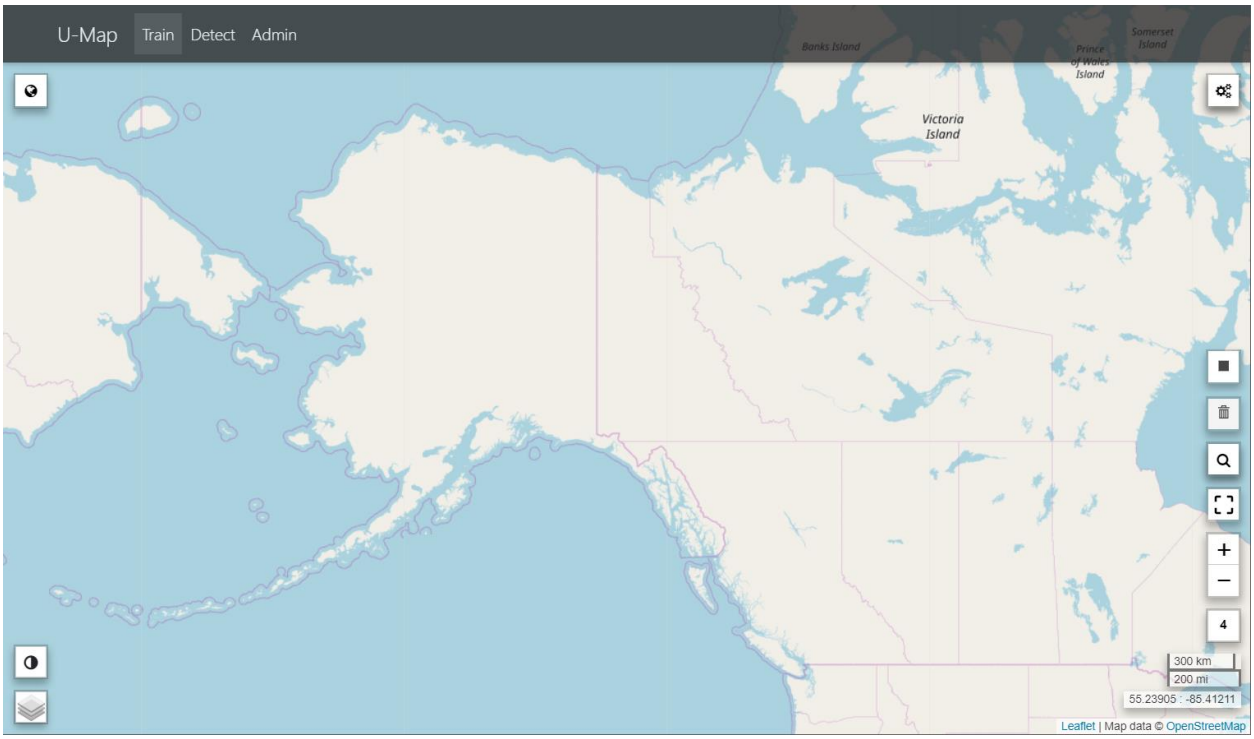


Figure 31 The Train landing page

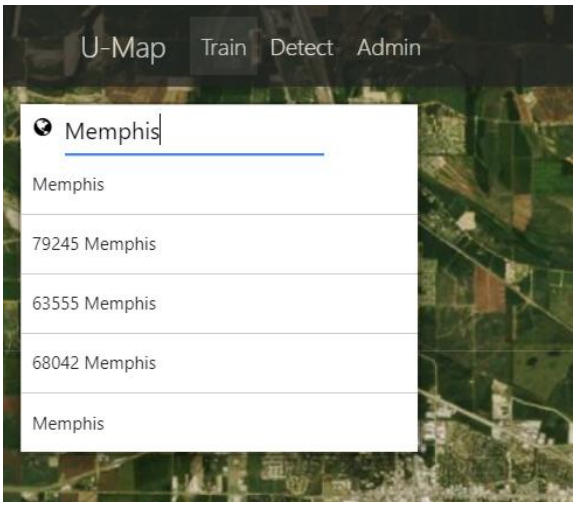


Figure 32 Performing a geocode search with an imagery overlay layer visible

5.2.2. Query Parameter Entry

Users may enter any number of OSM Overpass API query parameters and use these parameters to search for features within an AOI on the map. Users must enter query parameters in the format of *datatype["key" = "value"]* and may enter multiple parameters by entering each datatype-key-value tuple on a new line within the Query Parameters modal input form. Users may reset the query parameters to the default value using the Reset button. A link to the official OSM Overpass API language guide is provided for convenience (Figure 33).

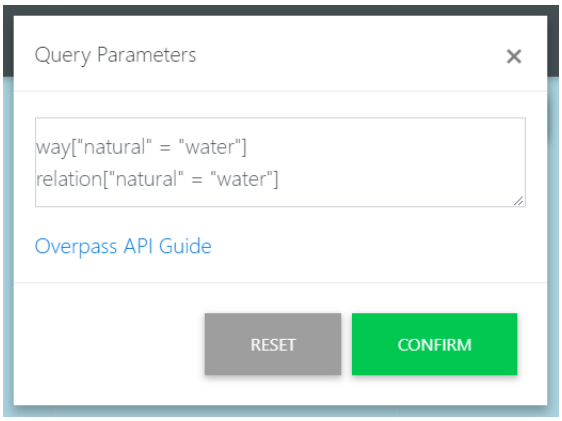


Figure 33 The Query Parameters modal form which allowed users to enter Overpass API queries for types of geospatial features

5.2.3. Selecting Features by Area of Interest

Using a rectangular selection tool, users may select one or more AOIs within the map. Features matching the users' given query parameters within the AOI will be returned to the user within the web UI. Users can inspect the features' OSM tags by clicking on the feature within the map. Mapbox satellite imagery tiles corresponding to the selected AOI will be loaded to provide the user with a ground truth with which to compare features returned from the OSM database (Figure 34). The user would wish to select an AOI or AOIs with a number of existent features adequate for model training.

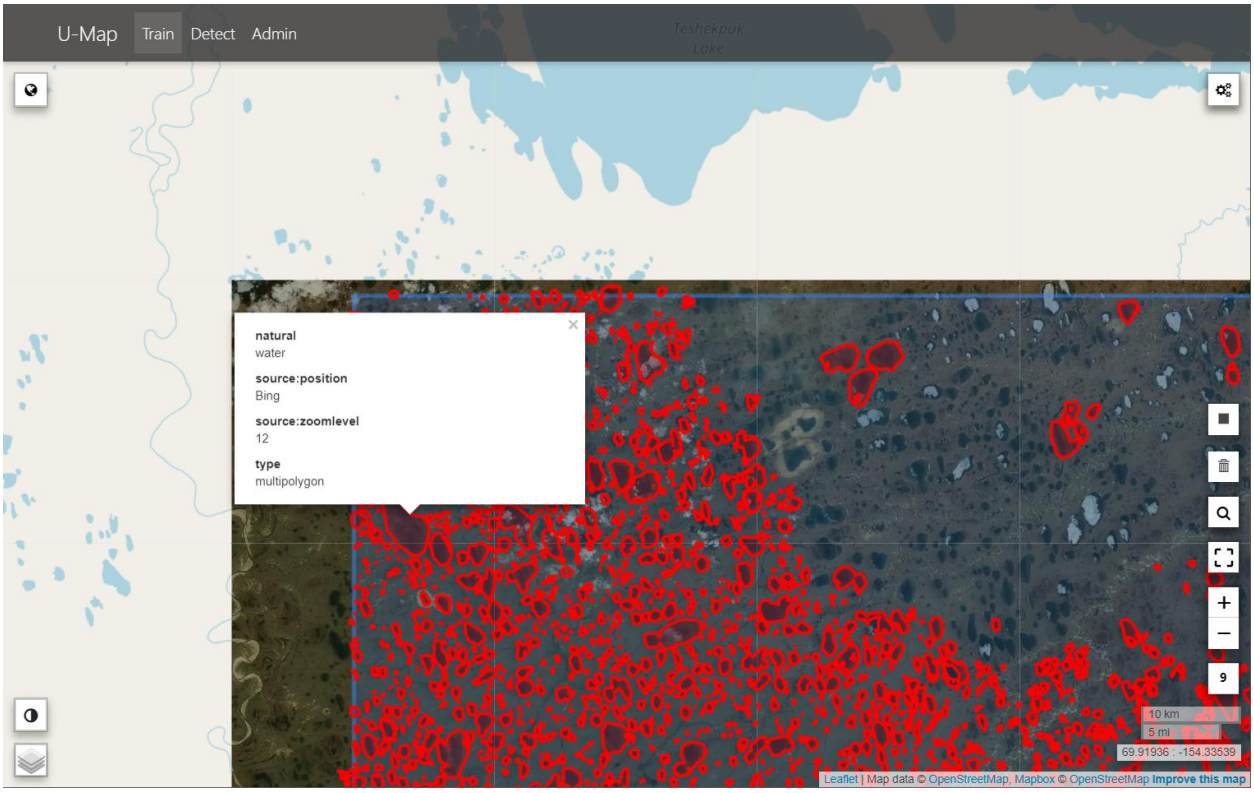


Figure 34 The Train page with an AOI selected and features and satellite imagery loaded

5.2.4. Model Parameters

When a user is satisfied with the selected feature data, they will then proceed to enter several parameters for the new model they wish to generate (Figure 35). The user will be prompted to enter

1. A name for the model that will be used to identify it in the future
2. A verbose description of the model
3. The number of epochs, or number of times the machine learning model will be shown the training data
4. The sample size that gives the percentage of the dataset, less the validation size, that should be used to train the model
5. The validation size that gives the percentage of the data that should be used validating the performance of the model
6. The validation prediction threshold, which limits the returned prediction data for the model validation to only pixels that model has assigned a confidence value greater than the threshold (e.g. .5 returns pixels that the model has determined to have at least a 50% chance of being the desired object)

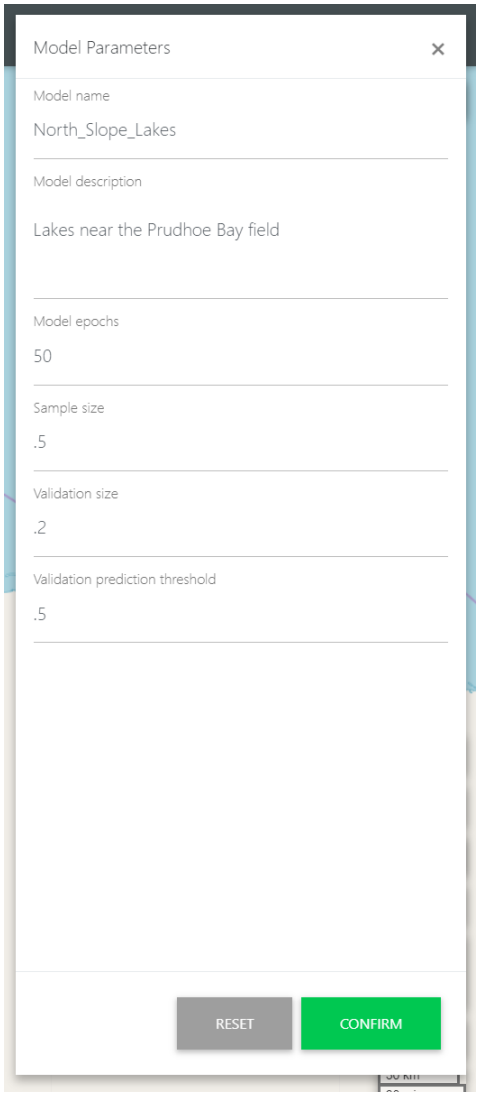


Figure 35 The Model Parameters modal form which allowed users to enter parameters to be used during model training

5.2.5. Creation of GeoJSON Masks

Following the user's confirmation of the model parameters, the system will generate GeoJSON representations of binary masks for all tiles within the AOI that contain OSM features matching the user's query parameters (Figure 36). Upon creation of all GeoJSON masks, the user's provided model name is passed to the Flask API and Python code for further processing and model training.

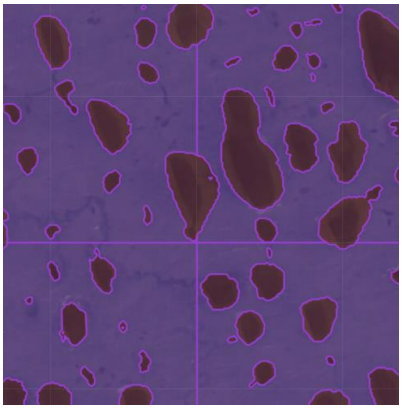


Figure 36 GeoJSON binary masks in the web UI

5.3. Model Training and Evaluation

The Python code exposed via the Flask API performed all of the substantive backend processing for this system. This includes data preprocessing, model training, model validation, object prediction from trained models, and data post-processing for display within the web UI. Due to the computational intensity of these processes, all processing occurs within the Python code asynchronously from the user's browser and interaction. The user may monitor the progress of these processes and evaluate model performance through the Admin page. The below describes the user workflow and data processing flows during data preprocessing and training of the U-Net model.

5.3.1. Model Training

Following the selection of training data, the user may view the progress of data preprocessing from the Admin page. Within the Admin page, the user is presented with the Models grid displaying the parameters of all models stored within the system (Figure 37). Within this, the user can view

1. The model's name
2. The latest status

3. The timestamp of the latest status
4. The number of epochs
5. The sample size
6. The validation size
7. The zoom level
8. The description
9. The OSM query parameters

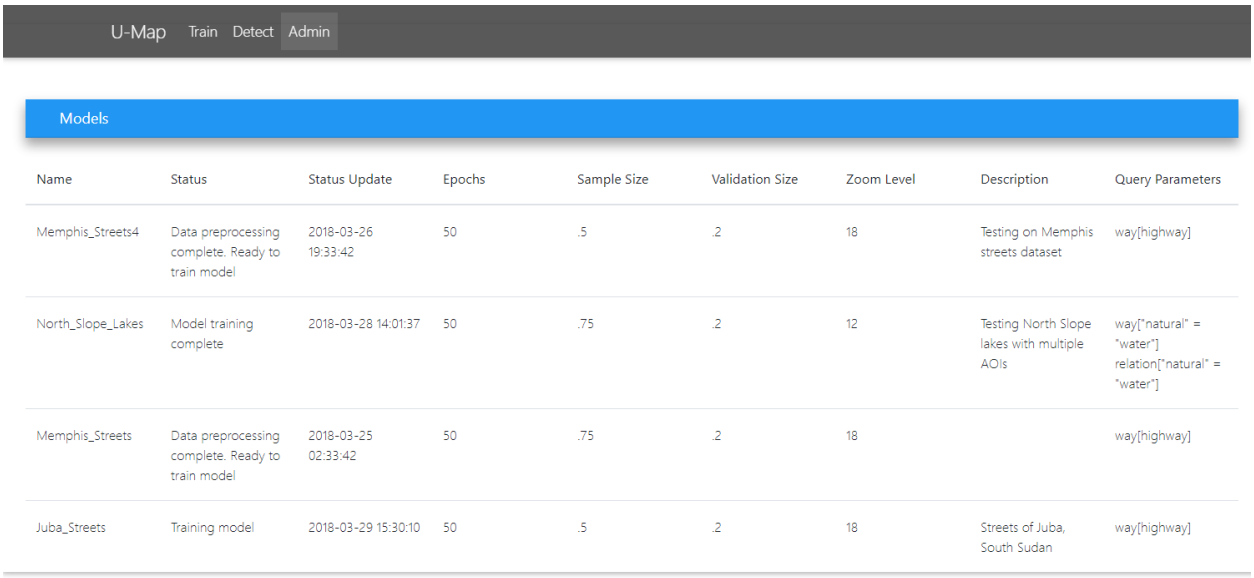


Figure 37 The Models grid displaying all models within the system

By clicking on the row of the model in the Models grid, the user can view further information about the model's status, as well as view the model's training masks and predictions. The user is presented with a workflow visualization detailing the progress of the model through the asynchronous preprocessing and model training phases (Figure 38). As the data preprocessing and model training phases progress, different options will be made available to the

user. Dependent upon progress, the View Training Masks and View Predictions buttons will be activated within the modal window (Figure 39).

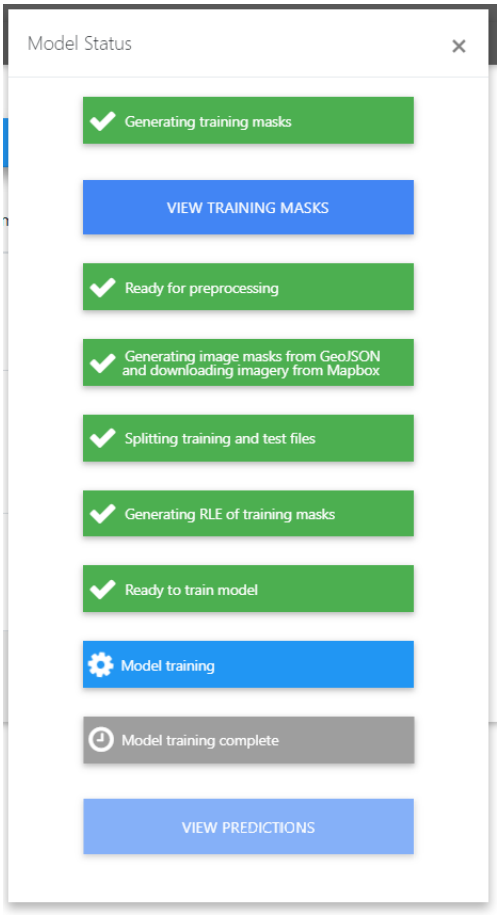


Figure 38 The Model Status modal displaying a model that is currently being trained

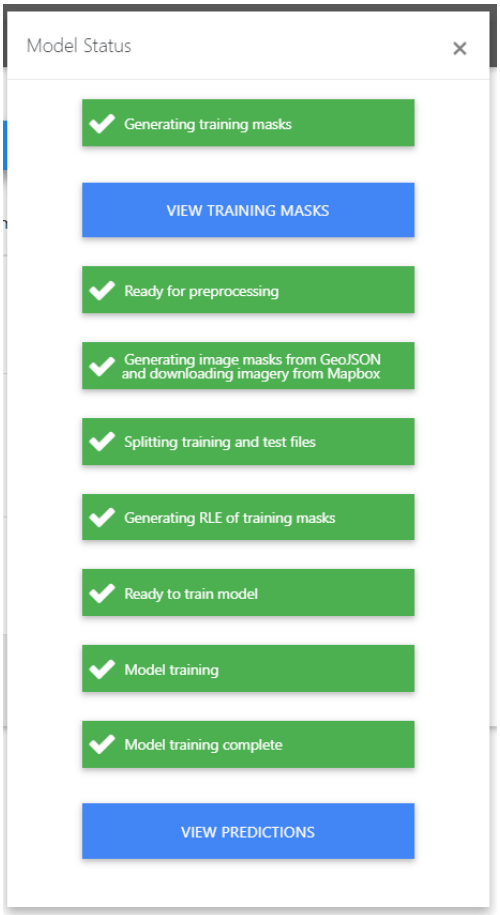


Figure 39 The Model Status modal displaying a trained model

When the user clicks the View Training Masks button within the modal window, they will be navigated to a map displaying all of the GeoJSON masks that were used as an input to train the machine learning model. When the user clicks the View Predictions button, the Predictions window will open with a grid displaying the test predictions generated following model training (Figure 40). By clicking on the row in the grid, they will be navigated to a map displaying all of the training object predictions. In the following screen the user may review model training output against existent OSM geospatial data as well as satellite imagery ground truth (Figure 41). If the user is satisfied with the performance of the model, they will proceed to

the Detect page to pass novel imagery to the model for object prediction. If not, they will retrain the model with different parameters and/or greater amounts of training data.

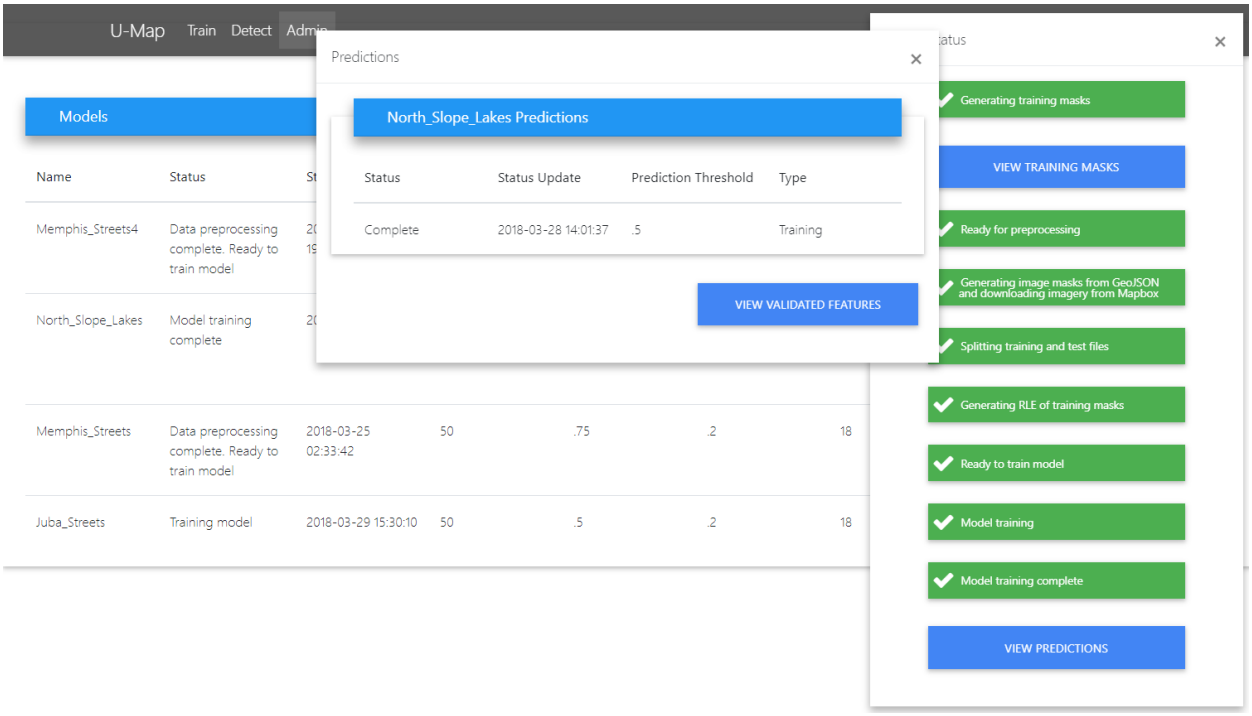


Figure 40 The Predictions window displaying a training prediction generated from a model

Figure 41 Existent OSM data (red) and validation predictions (blue) overlaying Mapbox satellite imagery (Mapbox 2018)

5.4. Predicting Objects from Imagery

The Detect page served as the users' portal for selecting novel imagery from which to predict instance of a given object. Within this page, users could select AOIs for novel imagery, select the trained model they wished to use (the type of object they wished to detect), and enter the prediction sensitivity threshold for the returned predictions. The below describes the UI, user workflow, and data processing flows of the Detect page.

5.4.1. AOI Selection

The Detect page was built off of the Train page and features all of the same UI/UX elements. However, the user utilizes these tools in a much different way. When using the rectangular selection tool, the user is selecting an AOI from which to generate predictions of object occurrences. As such, the user may wish to select an AOI that contains little existent geospatial vector data. As can be seen in Figure 42, the selected AOI shows a large number of lakes in the satellite ground truth imagery but very few are represented in the OSM dataset. Performing this analysis may be done in two different ways: (1) the user may enter the correct OSM query parameters for the type of object they wish to predict and select an AOI to view existent OSM feature data within that AOI or (2) display an imagery layer on the map and reduce the imagery layer opacity to compare features within the OSM features basemap with the satellite imagery layer.

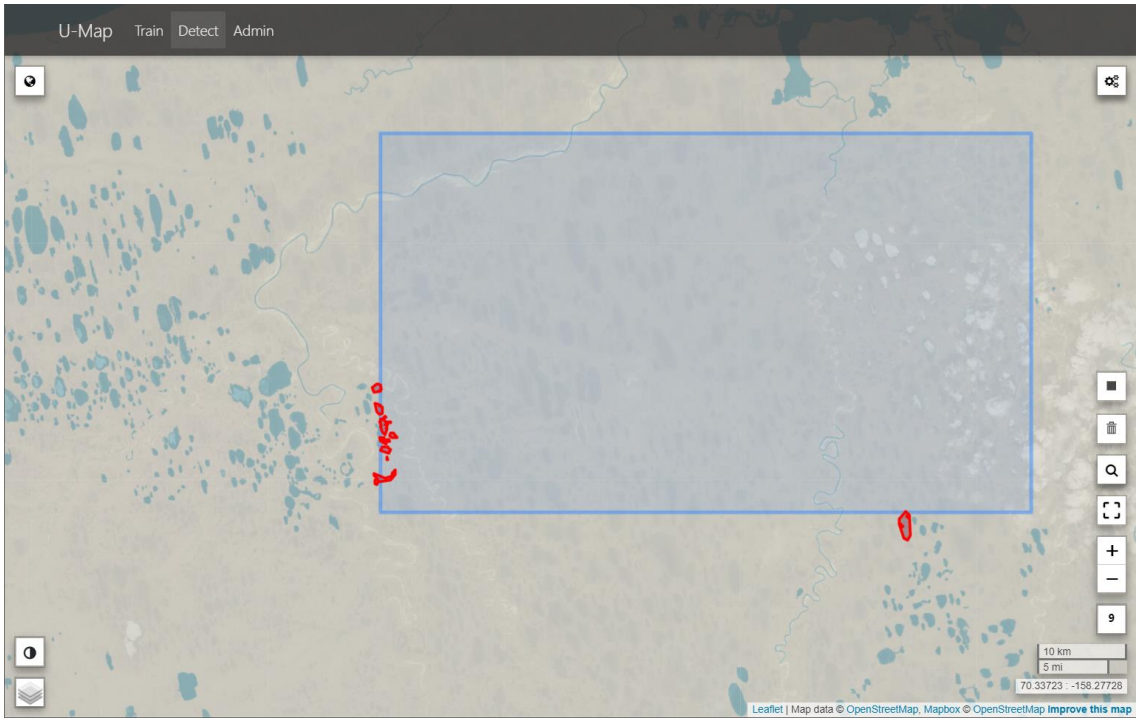


Figure 42 Detect page with an AOI selected and imagery layer opacity reduced to show little existent feature data within the AOI

5.4.2. Model Selection

Following the selection of an AOI, the user may then select the model that is to be used to predict object occurrences within the given AOI. The user is first presented with a dropdown list of available models (Figure 43). Following selection of the desired model, fields within the form are populated with the parameters that were used to train the model. The imagery zoom level used to train the model is provided, as the model will perform best when provided with imagery of the same zoon level from which it was trained. The user is then prompted to enter the prediction threshold for detection. The prediction threshold will limit the returned data to only instances in which the model has a classification confidence that meets or exceeds the user's given input (e.g. a prediction threshold of .5 will only return pixels that model has determined to have a 50% or greater likelihood of being the given object).

When the user has selected their desired AOI and entered the prediction threshold value, they may then proceed by clicking the Detect button within the Model Selection form. This will then pass the given AOI, model, and prediction threshold parameter to the Flask API to generate predictions asynchronously.

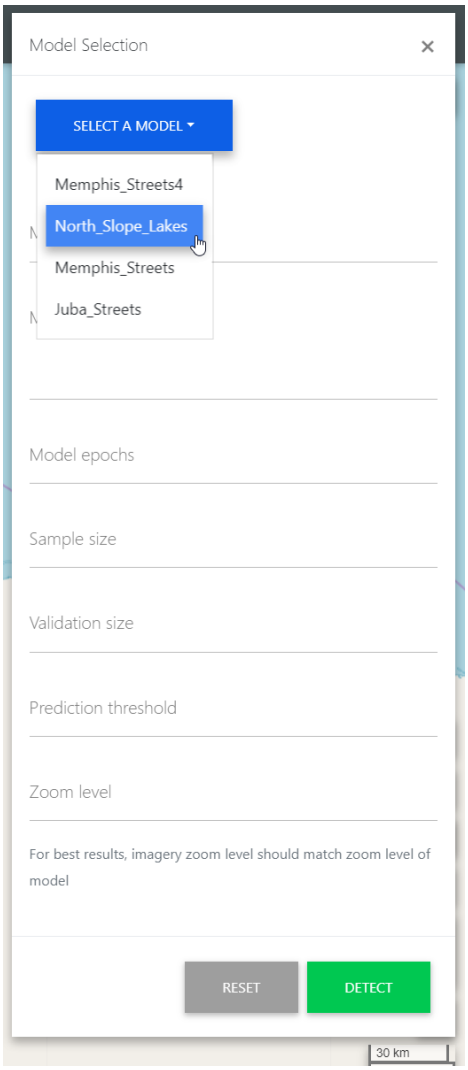


Figure 43 The Model Selection modal form

5.4.3. Viewing Predictions

While Python code is processing the novel imagery within the AOI the user has selected, the user may view the progress of this processing from the Admin page. Much in the same way

as the user views the processing status during the model generation phase, the user selects the desired model within the Models grid, opening the Model Status window on the right side of the screen. The user would then click the View Predictions button to open another model window containing a grid of all predictions generated by the given model. The user will be presented with the training predictions generated immediately following model training as well as any user generated predictions (Figure 44). If the prediction has not yet completed processing, the user will be shown the current processing step within the grid. If processing has completed, the user may click on the row of the prediction within the grid to view the output of the prediction.

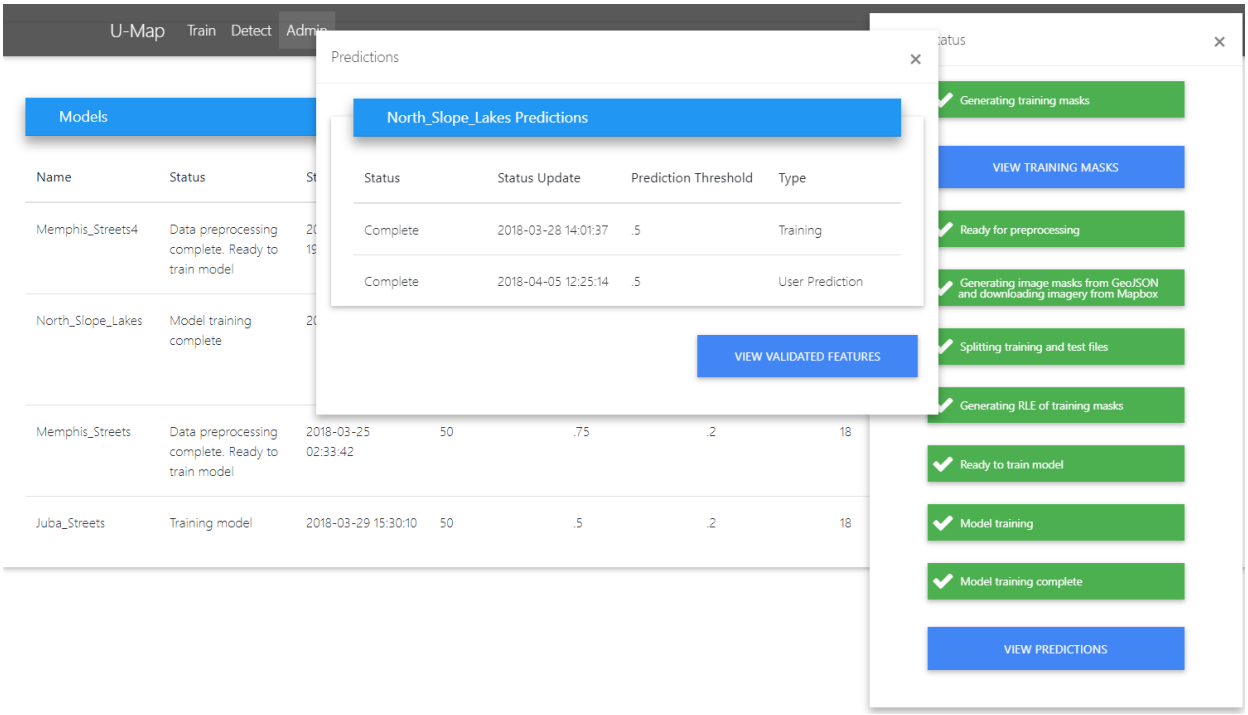


Figure 44 The Predictions window displaying all predictions generated from a model

The user is then navigated to the Review page to evaluate the output of the model for the given prediction AOI. The user may wish to compare this prediction output against satellite imagery ground truth layers and any existent OSM data within the AOI (Figure 45).

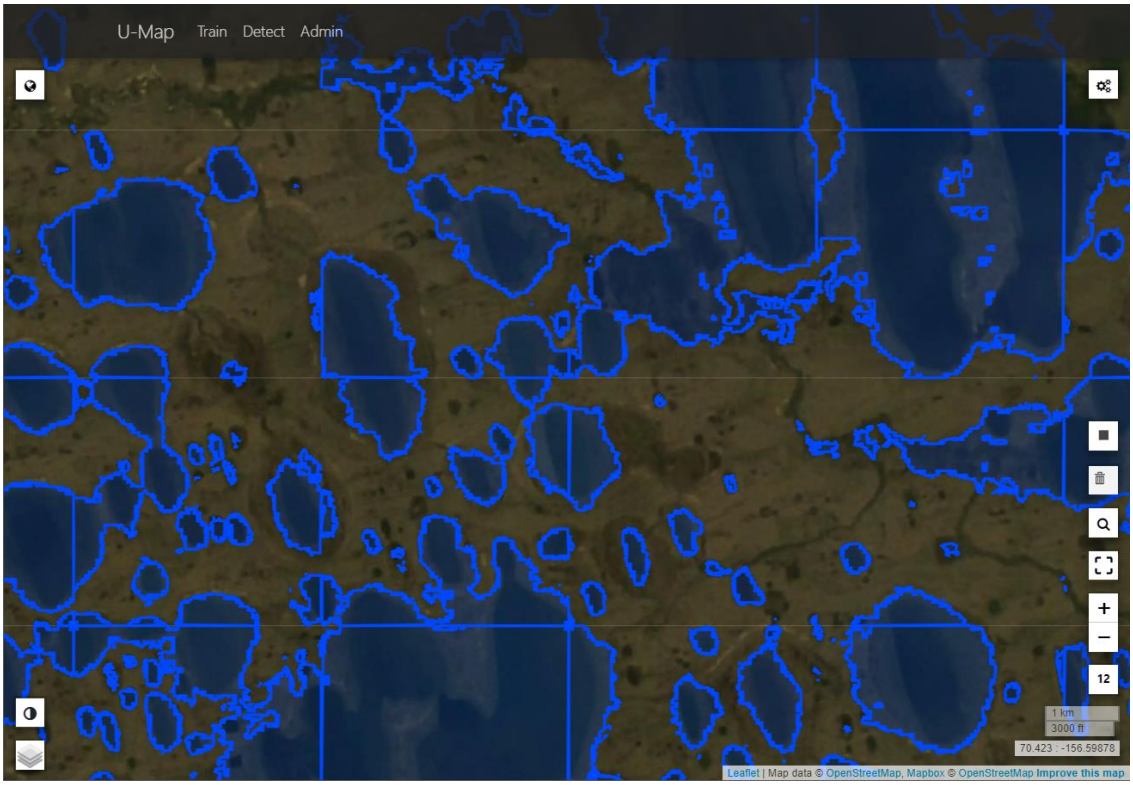


Figure 45 Object predictions (blue) over Mapbox satellite imagery layer

5.4.4. Validating Object Predictions

When a user has validated object predictions against ground truth imagery, they click on the given object within the web UI to mark the prediction as "validated" within the system. The object then becomes highlighted in red within the UI (Figure 46). When objects are clicked within the UI, the GeoJSON object is immediately validated within the system, with no need to manually commit the validation selections. To view these validated predictions following review, the user would return to the Admin page, open the Predictions grid for a given model and click the View Validated Features button to view all validated features from all predictions for the given model (Figure 44).

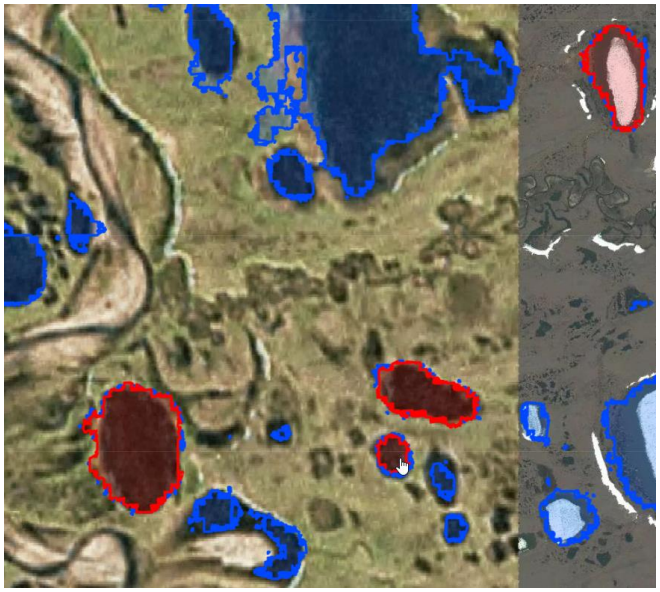


Figure 46 Validating object predictions (red) within the web UI

5.5. Testing and Evaluation

The application was tested using a unit testing strategy, with each application component tested independently before the system was tested as an integrated whole. For each component, the inputs and outputs were validated against expected outcomes, the inter-system and inter-module interfaces were tested, error handling was implemented, and efforts were made to maximize performance. The below details the testing and evaluation of the system's primary components.

5.5.1. Web UI

Testing of the web UI was mostly performed by navigating through the application, performing various user workflows, performing out of workflow operations to ensure system stability, and the like. The testing of the actual UI components was straight forward, given the immediate visual feedback provided by the UI. The Google Chrome browser's built-in Developer Tools were used for testing and debugging of UI HTML and JavaScript components.

The only major point of failure within the web UI was within the interfaces to other services, namely the Node.js API. These interfaces had to be extensively tested to ensure that endpoints on both sides of the system were receiving the expected data and that the data received through the interfaces was processed properly and any errors were handled gracefully within the system. As the system performed very little actual processing within the UI, performance rarely a concern.

5.5.2. Node.js API

The Node.js API performed a large amount of the geospatial processing for the data preprocessing pipeline. Complicating matters, it was challenging to visualize the geospatial operations that were being executed within the API code. Frequently the only solution for thorough testing was to insert breakpoints within the code and inspect the generated GeoJSON objects within third-party tools, such as the geojson.io web GeoJSON viewer.

Further, it appears as though the tilebelt Node package that was used within the application was not optimized to handle large volume geospatial operations. Performing difference operations on GeoJSON objects was often time consuming, with complex objects processing at a rate of only 1-2 difference operations per second. Further, the tilebelt methods did not handle unexpected types of geometry (such as lines or points) gracefully leading to this logic having to be coded in manually.

5.5.3. Python Preprocessing, Machine Learning Model, and Post-processing

The Python code required by far the greatest amount of testing and validation within the system. Within the data preprocessing flow, multiple Python libraries were required to convert GeoJSON objects into suitable mask images, often without clear intermediary datatypes between the packages defined. One package which was used to generate the binary masks, Matplotlib,

produced different output depending on defined dots per inch of the host machine's display settings – all of this leading to very controlled and defined testing procedure.

When configuring and testing the machine learning workflow itself, great care was taken to ensure system variables were comparable between the development and deployment machines. The performance variables defined within the model training process tended to raise unhandled exceptions in the code if system hardware was not sufficiently powerful to handle a given operation. Debugging aside, tuning was the most time-consuming testing task for the machine learning model. For a large area of training data (approximately 1000 km^2 and 1000 training images), training for 50 epochs took approximately 1.5 hours on a workstation with an Nvidia GeForce GTX 970 GPU with 4GB of GDDR5 graphics memory (Table 2). Several days of effort were spent tuning the input data and model hyperparameters to achieve satisfactory model performance.

Table 2 Testing system configuration

CPU	Intel Core i5-4690K 3.5GHz
Hard Drive	Samsung 850 EVO SATA III SSD
RAM	Kingston HyperX FURY 1866MHz DDR3 16GB (2x 8GB)
GPU	Gigabyte Nvidia GeForce GTX 970 G1 GDDR5 4GB

To validate model performance, the model training workflow was performed on approximately ten different geographical areas, using several different classes of geospatial features. Table 3 provides six training scenarios generated during the testing of the neural network and classifier. Two distinct areas and geospatial features were chosen for this testing: lakes in Alaska's North Slope region and roads in Juba, South Sudan. The testing scenarios were structured by first sourcing training data from an AOI and training a model for 50 epochs, the default value within the system. This generated both validation predictions as well as descriptive

statistics of model performance which could be viewed within the TensorBoard visualization tool (Figure 47).

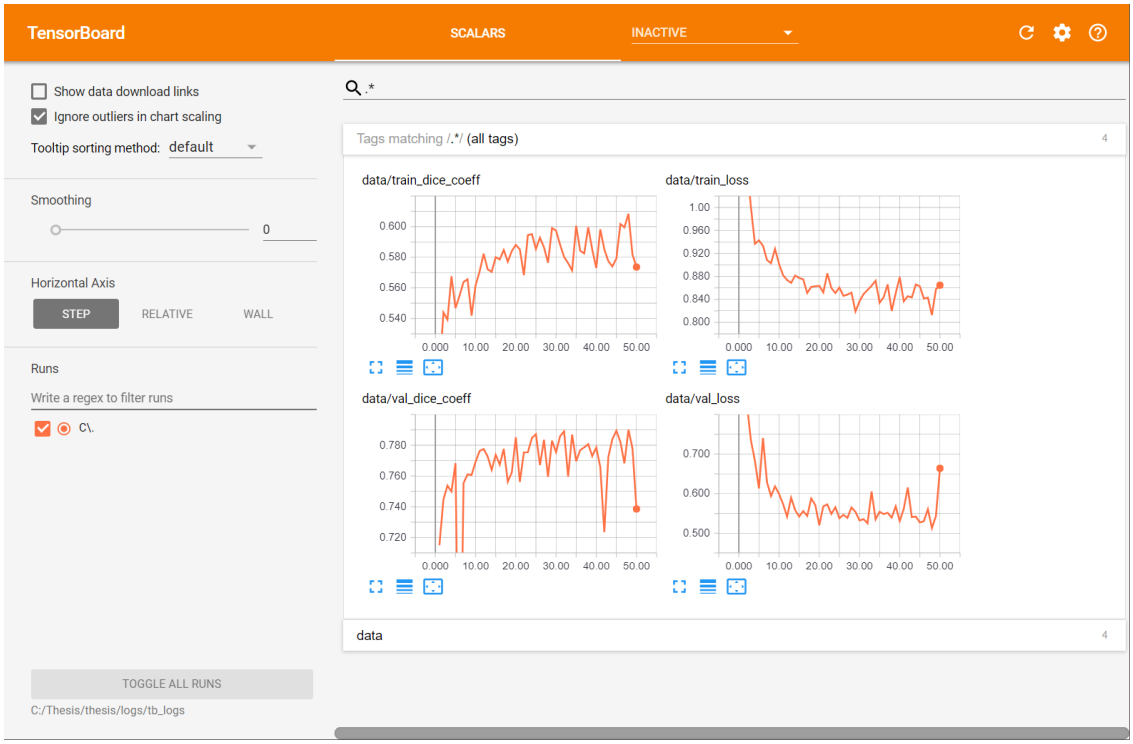


Figure 47 TensorBoard visualizing descriptive statistics of model performance during training

The Dice similarity coefficient (DSC) and a cross-entropy loss function were then used to measure the pixel-level classification accuracy of the trained model. Zou et al. (2004) described the DSC as a measure of the "spatial overlap between two sets of segmentations." The DSC is defined as a value between 1 and 0, with 1 representing total overlap and 0 indicating no overlap at all. In short, the higher the DSC, the better model predictions align with known geospatial features. The cross-entropy loss function compares the predicated probability of a pixel being a given class against the actual label of that pixel (Berman, Triki and Blaschko 2016). For example, if a model were to predict that a pixel had 16% probability of being of a geospatial

feature and training data showed that pixel to be within a geospatial feature (100% probability), the resultant cross-entropy value would be high.

These values were recorded along with the amount of training data and the length of time required to train the model. Two subsequent training scenarios were then performed – one increasing the number of epochs, another providing more training data by increasing the size of the AOI while leaving the number of epochs unchanged. Table 3 shows that increasing the number of training epochs provided relatively small improvements in both the DSC and cross-entropy measures, with only an approximately 1-2% increase in performance of both measures for a 54-77% increase in training time. However, increasing the amount of training data available to the model provided much greater improvement in performance. By increase the amount of training data provided to the North Slope Lakes model by 123%, a 10% improvement in the DSC and a 25% improvement in the cross-entropy loss were seen. Improvements were also seen within the Juba Streets model when increasing the amount of training data, however model performance continued to lag behind that of the North Slope Lakes model, particularly regarding the cross-entropy measure. This was likely due to the model struggling to differentiate the primarily dirt streets from other dirt areas sharing a similar appearance (Figure 48).

Table 3 Select model training statistics

Training Images	Versus Baseline	Epochs	Training Time (min.)	Versus Baseline	DSC	Versus Baseline	Cross-Entropy	Versus Baseline
North Slope Lakes								
898	--	50	89	--	0.7901	--	0.5129	--
898	--	75	137	54%	0.8036	2%	0.4998	-3%
1,999	123%	50	125	40%	0.8658	10%	0.3862	-25%
Juba Streets								
1,256	--	50	111	--	0.7222	--	0.7821	--
1,256	--	75	196	77%	0.7389	2%	0.7644	-2%
1,976	57%	50	144	30%	0.7775	8%	0.7029	-10%

Figure 48 OSM features over Mapbox satellite imagery of Juba, South Sudan demonstrating the visual similarity of streets and other non-street landcover (OpenStreetMap 2018) (Mapbox 2018)

The data output by the neural network classifier was not inherently spatially referenced, nor of a suitable vector format to be displayed within the web UI. Similar to the preprocessing pipeline, data post-processing functions required the use of numerous Python libraries with

intermediate interfaces connecting the functions. The GDAL library proved particularly difficult to test given the lack of verbose error messaging within the library. Further, the output from the GDAL.polygonize function required nontrivial manipulations following initial performance testing. Due to the aforementioned GeoJSON complexity, extensive amounts of data cleaning was required to achieve satisfactory performance and processing times.

5.5.4. Full System

The integrated application was tested as a whole by stepping through the documented user workflow numerous times and fixing coding errors and tuning performance as needed. During final acceptance testing, the application no longer raised any errors to the end user and performance was satisfactory given GPU processing constraints. The application was designed to perform asynchronous processing and, given this, the recorded times for data preprocessing, model training, and post-processing were found to be acceptable. It should be noted that the model training phase of the user's workflow was by far the most time-consuming aspect of utilizing the application.

Following the training of a model, which varied in time depending on the amount of input data and epochs provided by the user, predicting imagery from the model was found to be exceedingly performant. Prediction times were found to average approximately .3 seconds per imagery tile input, meaning that a 1000 km^2 area (at zoom level 12) could be predicted in a five-minute timespan.

Chapter 6 Conclusions

This chapter summarizes the development of the U-Map application, challenges within the development process, and any limitation within the current implementation of the application. Further, a discussion of additional applications of U-Map and potential opportunities for future work are given.

6.1. Summary

The U-Map application provides an intuitive means for users to source data for any arbitrary type of geospatial features from the open source OSM geospatial dataset and combine these data with satellite imagery from Mapbox to train machine learning models for object prediction. The application allows users to monitor the asynchronous training of machine learning models and to evaluate performance of models all from within a web user interface. Users are then able to use trained machine learning models to predict occurrences of geospatial features within novel satellite imagery and validate the predictions through a simple workflow.

Given the ever-increasing volume of imagery produced by Earth observation satellites, it crucial that tools and methods for deriving *information* from the torrent of data continue to be developed. Machine learning technologies have been shown, within U-Map and other research, to be suited to this task. Though the field is still in its infancy, great progress has already been shown – with researchers using machine learning to slow deforestation, plan better cities, and combat crime (White 2018) (Raad 2017) (Rudin 2013). With further refinement of the underlying algorithms and the development of end-user tools, a much larger number of capable subject matter experts will be empowered to harness the full power of machine learning technologies to help solve some of the world's biggest problems.

6.2. Challenges in Development

The major challenges in development centered mainly around the scope of the application and the number of technical components required to realize the whole system. A web UI, two custom developed APIs, and a SQL database ultimately required the development of upwards of twenty unique interfaces between the components, leading to extensive testing and validation of inputs and outputs from these interfaces. One major discovery during the testing of these interfaces was the inability of the Flask API to handle large amounts of GeoJSON data within a REST request. This lead to a rearchitecture of the data preprocessing pipeline, wherein GeoJSON masks were stored within the SQL database as opposed to being directly sent to the API via a web request.

While the interfaces between system components were relatively well defined, the functioning and even selection of an appropriate machine learning model was much less so. Significant amounts of research were frontloaded into the project to ensure that the correct neural network architecture was chosen to meet the requirements of the application. Once the U-Net architecture was selected and an open source base implementation was identified, tuning and reengineering the U-Net neural network and associated programming presented a large learning curve, as well. Further, the computational intensity of model training created a painstaking debugging and tuning process.

6.3. Limitations

The U-Map application was designed and developed on a workstation with a consumer grade GPU, without access to a high-end installed GPU or high-performance cloud GPU resources. As such, the model training workflow takes upwards of an hour to process a moderate number of tiles over a small number of epochs. Deploying the application with access to more

powerful GPU compute resources would greatly increase the performance of the application. Further, as model training will always be an asynchronous task give current technology, it could be useful to expose TensorBoard visualizations to the end users through the application to provide them with greater insight into the model's performance, provided they were knowledgeable enough in the domain to interpret the information appropriately.

Regarding the functionality of the system, geospatial output from U-Map is currently limited to MultiPolygon datatypes with no functionality in place to generate line or point data from predictions. Polygons are the only logical output from the neural network, as it functions as a pixel-level classifier, but post-processing functions could likely be integrated to transform polygon data into line or point data, dependent on the desires of the user.

6.4. Future Work

There are a number of future enhancements that could expand on the utility and usability of the U-Map application. Firstly, integrating additional data sources into the application could greatly increase the value of system. Conceivably by integrating other sources of geospatial feature data, the quality and quantity of training data would vastly increase. OSM data, being VGI, is not subject to the same quality control and data refresh processes seen within many organizations' own geospatial or data management functions. With better spatial ground truth data, model prediction accuracy would likely improve. Further, the ability to integrate more timely satellite imagery would expand the use cases of the application. In instances where the situation on the ground is rapidly evolving, such as natural disasters or in security and defense applications, the ability to rapidly detect and digitize features on the ground could prove a huge boon to an organization's effectiveness.

If perpetually updated imagery was integrated into the system, it is conceivable that there could be value in redesigning the system to largely remove the human element from the prediction workflow. In instances where "good enough" predictions are sufficient, such as access control or persistent monitoring of an area, an automated workflow could be designed wherein the system performs automatic object detection within a given AOI upon every imagery refresh. This could provide users with data-driven alerts when visible change is detected on the earth's surface (e.g. new encampments detected within a contested region, deforestation within a conservation area).

In addition to additional data, internal machine learning processes could be further enhanced as well. Training data could be augmented through additional data transformation procedures. Within the original U-Net paper, the authors applied elastic deformations to input training images and masks to both increase the amount of training data by creating new training images and masks from the transformed data and to increase model predictive performance by exposing the model to more possible object appearances (Figure 49) (Ronneberger, Fischer and Brox 2015). This strategy would likely translate well from biomedical imagery to satellite imagery considering the near infinite shapes and configurations of roads, lakes, rivers, and the like.

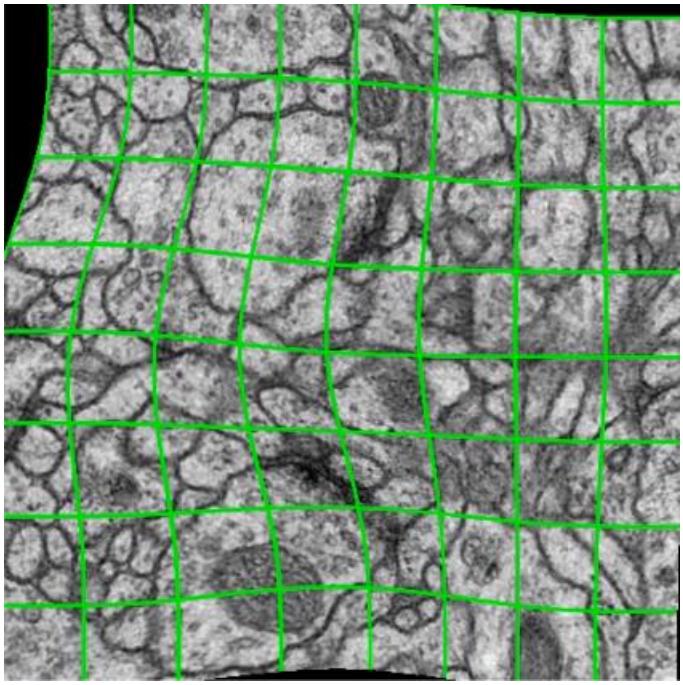


Figure 49 Elastic deformation of biomedical imagery (Ronneberger, Fischer and Brox 2015)

Rearchitecting the system to allow for models to be training on multiple classes of geospatial features is another large opportunity for future improvement. Currently the system is architected to allow for a model to be trained to detect a single type of geospatial feature. If this was reengineered to allow for multiclass training, a whole host of new use cases could be presented. With multiclass object detection, the efficiency of individual workflows could be greatly increased, allowing the user to select an AOI and any number of classes of objects to detect within the imagery or to perform land use classification.

References

Al Achkar, Ziad, Isaac L. Baker, Brittany L. Card, and Nathaniel A. Raymond. n.d. "Satellite Imagery Interpretation Guide Intentional Burning of Tukuls." Guide.

Arnab, Anurag, and Philip H.S Torr. 2017. "Pixelwise Instance Segmentation with a Dynamically Instantiated Network." *2017 IEEE Conference on Computer Vision and Pattern Recognition.* Honolulu: IEEE.

Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. 2017. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Scene Segmentation." *IEEE Transactions on Pattern Analysis and Machine Intelligence* PP (99): 1-1. Accessed September 10, 2017. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7803544.

Benn, D. I., T. Bolch, K. Hands, J. Gulley, A. Luckman, L. I. Nicholson, D. Quincey, S. Thompson, R. Toumi, and S. Wiseman. 2012. "Response of debris-covered glaciers in the Mount Everest region to recent warming, and implications for outburst flood hazards." *Earth-Science Reviews* 114 (1-2): 156-174. Accessed September 24, 2017. http://www.sciencedirect.com/science/article/pii/S0012825212000451.

Berman, Maxim, Amal Rannen Triki, and Matthew B. Blaschko. 2016. "The Lovasz-Softmax loss: A tractable surrogate for the optimization of the intersection-over-union measure in neural networks." *The 19th International Conference on Artificial Intelligence and Statistics.* Cádiz. 1-14.

Bradski, Gary, Adrian Kaehler, and Vadim Pisarevsky. 2015. "Learning-Based Computer Vision with Intel's Open Source." *Intel Technology Journal* 119-130. Accessed September 15, 2017. http://web.a.ebscohost.com/ehost/detail/detail?vid=0&sid=bc0083a1-51de-4541-83b5-4bcba1718917%40sessionmgr4007&bdata=JnNpdGU9ZWhvc3QtbGl2ZQ%3d%3d#AN=17208536&db=cph.

Brownlee, Jason. 2017. *What is the Difference Between Test and Validation Datasets?* July 17. Accessed April 1, 2018. https://machinelearningmastery.com/difference-test-validation-datasets/.

Chiang, Yao-Yi, and Craig A. Knoblock. 2013. "A General Approach for Extracting Road Vector Data from Raster Maps." *International Journal of Document Analysis and Recognition* 33-81.

Cremers, Daniel. 2012. "Optimal solutions for semantic image decomposition." *mage and Vision Computing* 476-477.

Development Seed. 2017. *Skynet - Machine Learning for Satellite Imagery.* Accessed August 30, 2017. https://developmentseed.org/projects/skynet/.

dos Santos, Leonardo Araujo. 2017. *Artificial Inteligence.* GitBook.

Draper, Robert. 2018. "They Are Watching You—and Everything Else on the Planet." *National Geographic*, February. https://www.nationalgeographic.com/magazine/2018/02/surveillance-watching-you/.

Emerson, Charles W., Nina Siu-Ngan Lam, and Dale A. Quattrochi. 2005. "A comparison of local variance, fractal dimension, and Moran's I as aids to multispectral image classification." *International Journal of Remote Sensing* 26 (8): 1575-1588. Accessed

September 16, 2017. http://www.rsgis.envs.lsu.edu/docs/Emerson-Lam-Quat-IJRS-2005.pdf.
Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. 2015. "A Neural Algorithm of Artistic Style."
GEOFABRIK. 2017. *Map Compare.* Accessed September 10, 2017. http://tools.geofabrik.de/mc/#9/70.7677/-156.4569&num=4&mt0=mapnik&mt1=google-map&mt2=here-map&mt3=google-satellite.
Géron, Aurélien. 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* 1st. O'Reilly Media.
—. 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly.
GISGeography. 2017. *Spectral Signature Cheatsheet – Spectral Bands in Remote Sensing.* February 1. Accessed September 24, 2017. http://gisgeography.com/spectral-signature/.
GitHut. n.d. *GitHut 2.0.* Accessed Marh 14, 2018. https://madnight.github.io/githut/#/pull_requests/2017/4.
Godard, Tuatini. 2017. *Practical image segmentation with Unet.* October 2. Accessed September 20, 2017. https://tuatini.me/practical-image-segmentation-with-unet/.
Google. 2017. *Google Maps.* Accessed September 24, 2017. https://www.google.com/maps/@70.6526779,-155.0450154,224626m/data=!3m1!1e3.
—. 2017. *How Machine Learning with TensorFlow Enabled Mobile Proof-Of-Purchase at Coca-Cola.* September 21. Accessed September 21, 2107. https://developers.googleblog.com/2017/09/how-machine-learning-with-tensorflow.html.
—. 2016. *Train an Image Classifier with TensorFlow for Poets - Machine Learning Recipes #6.* June 30. Accessed September 2, 2017. https://www.youtube.com/watch?v=cSKfRcEDGUs.
Guo, Li, Nesrine Cehata, Clément Mallet, and Samia Boukir. 2011. "Relevance of airborne lidar and multispectral image data for urban scene classification using Random Forests." *ISPRS Journal of Photogrammetry and Remote Sensing* 56-66. Accessed September 13, 2017. http://lareg.ensg.ign.fr/labos/matis/pdf/articles_revues/chehata_etal_IJPRS10.pdf.
Hamilton, Ryan. n.d. *Fun facts about DigitalGlobe satellites.* Accessed September 15, 2017. http://blog.digitalglobe.com/geospatial/fun-facts-about-digitalglobe-satellites/.
Henry, Corentin, Seyed Majid, and Nina Merkle. 2018. "Road Segmentation in SAR Satellite Images with Deep Fully-Convolutional Neural Networks." *IEEE Geoscience and Remote Sensing Letters.*
Hijazi, Samer, Rishi Kumar, and Chris Rowen. 2015. "Using Convolutional Neural Networks for Image Recognition." *Candece Design Systems.* Accessed September 23, 2017. https://ip.cadence.com/uploads/901/cnn_wp-pdf.
Howat, I. M., S. de la Pena, J. H. van Angelen, J. T. M. Lenaerts, and M. R. van der Broeke. 2013. "Expansion of meltwater lakes on the Greenland Ice Sheet." *The Cryosphere* 201-204. Accessed September 24, 2017. https://www.the-cryosphere.net/7/201/2013/tc-7-201-2013.pdf.
IBM. 2017. "10 Key Marketing Trends for 2017." *IBM.* Accessed September 23, 2017. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN.
Iglovikov, Vladimir, Sergey Mushinskiy, and Vladimir Osin. 2017. "Satellite Imagery Feature Detection using Deep Convolutional Neural Networks: A Kaggle Competition."

Jean, Neal, Marshall Burke, Michael Xie, W. Matthew Davis, David B. Lobell, and Stefano Ermon. 2016. "Combining satellite imagery and machine learning to predict poverty." *Science* 353 (6301): 790-794. Accessed September 22, 2017. http://science.sciencemag.org/content/353/6301/790.full.

Jiang, Hao, Min Feng, Yunqiang Zhu, Ning Lu, Jianxi Huang, and Tong Xiao. 2014. "An Automated Method for Extracting Rivers and Lakes from Landsat Imagery." *Remote Sensing* 5067-5089. Accessed September 14, 2017. http://www.mdpi.com/2072-4292/6/6/5067/htm.

Johnson, Andrew L. 2017. *DeepOSM.* Accessed August 20, 2017. https://github.com/trailbehind/DeepOSM.

Kaggle. 2018. *Kaggle.* Accessed April 25, 2018. https://www.kaggle.com/.

Lawrence, John, Jonas Malmsten, Andrey Rybka, Daniel A. Sabol, and Ken Triplin. 2017. "Comparing TensorFlow Deep Learning Performance Using CPUs, GPUs, Local PCs and Clod." *Proceedings of Student-Faculty Research Day, CSIS, Pace University.* 1-7.

Lee, Andy. 2015. *Comparing Deep Neural Networks and Traditional Vision Algorithms in Mobile Robotics.* Swathmore College.

Levin, G., D. Newbury, K. McDonald, I. Alvarando, A. Tiwari, and M. Zaheer. 2016. *Terrapattern: Open-Ended, Visual Query-By-Example for Satellite Imagery using Deep Learning.* May 24. Accessed September 12, 2017. http://www.terrapattern.com/.

Long, Johnathan, Evan Shelhamer, and Trevor Darrell. 2017. "Fully Convolutional Networks for Semantic Segmentation." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 640-651.

Lu, Dengsheng, Scott Hetrick, Emilio Moran, and Guiying Li. 2012. "Application of Time Series Landsat Images to Examining Land-use/Land-cover Dynamic Change." *Photogrammetric Engineering & Remote Sensing* 747-755. Accessed September 16, 2017. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4201056/.

Mapbox. 2018. Accessed March 14, 2018.

—. 2018. *Legal.* March 15. Accessed April 1, 2018. https://www.mapbox.com/tos/.

—. n.d. *Satellite imagery.* Accessed April 25, 2018. https://www.mapbox.com/help/how-satellite-imagery-works/.

Mishra, Kshitij, and P. Rama Chandra Prasad. 2015. "Automatic Extraction of Water Bodies from Landsat Imagery Using Perceptron Model." *Journal of Computational Environmental Sciences* 2015: 1-9. Accessed October 21, 2017. https://www.hindawi.com/archive/2015/903465/.

Mnih, Volodymyr. 2013. "Machine Learning for Aerial Image Labeling." PhD Thesis, Graduate Department of Computer Science, University of Toronto. Accessed August 30, 2017. https://www.cs.toronto.edu/~vmnih/docs/Mnih_Volodymyr_PhD_Thesis.pdf.

Monitoring of the Adean Amazon Project. 2018. *Monitoring of the Adean Amazon Project.* Accessed January 15, 2018. http://maaproject.org/en/.

Ng, Andrew. 2013. *Andrew Ng.* Accessed November 20, 2017. http://www.andrewng.org/courses/.

OpenCV. n.d. *Image Segmentation with Watershed Algorithm.* Accessed April 6, 2018. https://docs.opencv.org/3.3.1/d3/db4/tutorial_py_watershed.html.

OpenStreetMap. 2018. Accessed March 26, 2018. https://www.openstreetmap.org.

Orbital Insight. 2017. *Orbital Insight.* Accessed September 20, 2017. https://orbitalinsight.com/.

Patel, Drishtie. 2017. *OSM @ Facebook.* August 19. Accessed September 2, 2017. http://2017.stateofthemap.org/2017/ai-assisted-road-tracing-for-openstreetmap/.
Pelletier, John D. 2005. "Formation of oriented thaw lakes by thaw slumping." *Journal of Geophysical Research.* Accessed September 24, 2017. http://onlinelibrary.wiley.com/doi/10.1029/2004JF000158/full.
Raad, Mansour. 2017. *A new business intelligence emerges: Geo.AI.* April 18. Accessed April 28, 2018. https://www.esri.com/about/newsroom/publications/wherenext/new-business-intelligence-emerges-geo-ai/.
Rey, Javier. 2017. *Object detection: an overview in the age of Deep Learning.* August 30. Accessed April 6, 2018. https://tryolabs.com/blog/2017/08/30/object-detection-an-overview-in-the-age-of-deep-learning/.
Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. 2015. *U-Net: Convolutional Networks for Biomedical Image Segmentation.* Freiburg, Germany: Computer Science Department and BIOSS Centre for Biological Signalling Studies.
Ros, German, Ramos, Sebastian, Manuel Granados, Amir Bakhtiary, David Vazquez, and Antonio M. Lopez. 2015. "Vision-based Offline-Online Perception Paradigm for Autonomous Driving." *IEEE Winter Conference on Applications of Computer Vision WACV2015.* http://adas.cvc.uab.es/elektra/enigma-portfolio/semantic-segmentation/.
Rudin, Cynthia. 2013. *Predictive Policing: Using Machine Learning to Detect Patterns of Crime.* August 22. Accessed April 28, 2018. https://www.wired.com/insights/2013/08/predictive-policing-using-machine-learning-to-detect-patterns-of-crime/.
Schmidhuber, Jurgen. 2015. "Deep Learning in Neural Networks: An Overview." *Neural Networks* 85-117.
Scoles, Sarah. 2017. "How AI Could (Really) Enhance Images from Space." *Wired.* October 31. Accessed May 31, 2018.
Shi, Jianbo, and Jitendra Malik. 2000. "Normalized Cuts and Image Segmentation." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 888-904.
Shuai, Bing, Liu Ting, and Wang Gang. 2016. "Improving Fully Convolution Network for Semantic Segmentation." Accessed May 31, 2018. https://arxiv.org/pdf/1611.08986.pdf.
Song, Yuheng, and Hao Yan. 2017. "Image Segmentation Algorithms Overview." 1-6.
TensorFlow. 2018. *Image Recognition.* January 27. Accessed April 6, 2018. https://www.tensorflow.org/tutorials/image_recognition.
Terrapattern. 2016. *About Terrapattern.* Accessed September 25, 2017. http://www.terrapattern.com/about.
University of Cambridge. n.d. *SegNet.* Accessed September 10, 2017. http://mi.eng.cam.ac.uk/projects/segnet/.
Univesity of Arizona. 2005. *Growth Secrets Of Alaska's Mysterious Field Of Lakes.* June 28. Accessed September 24, 2017. https://www.sciencedaily.com/releases/2005/06/050627233623.htm.
Unnikrishnan, Ranjith, Caroline Pantofaru, and Martial Hebert. 2007. "Toward Objective Evaluation of Image Segmentation Algorithms." *IEEE Transactions on Pattern Analysis and Machine Intelligence* (IEEE) 29 (6): 929-944. http://ieeexplore.ieee.org/abstract/document/4160946/.
Vance, Ashlee. 2017. "The Tiny Satellites Ushering in the New Space Revolution." *Bloomberg Businessweek*, June 29.

Walia, Anish Singh. 2017. *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent.* June 10. Accessed April 1, 2018. https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f.

White, Topher. 2018. *The fight against illegal deforestation with TensorFlow.* March 21. Accessed April 28, 2018. https://blog.google/topics/machine-learning/fight-against-illegal-deforestation-tensorflow/.

Yu, Shiqi, Sen Jia, and Chunyan Xu. 2017. "Convolutional neural networks for hyperspectral image classification." *Neurocomputing* 219: 88-98. Accessed September 24, 2017. http://www.sciencedirect.com/science/article/pii/S0925231216310104.

Zhou, Bolei, Agata Kholsa, Agata Lapedriza, Aude Olivia, and Antonio Torralba. 2016. "Learning Deep Features for Discriminative Localization." *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Zou, Kelly H., Simon K. Warfield, Aditya Bharatha, Clare M.C. Tempany, Michael R. Kaus, Steven J. Haker, William M. Wells III, Ferenc A. Jolesz, and Ron Kiknis. 2004. "Statistical Validation of Image Segmentation Quality Based on a Spatial Overlap Index." *Academic Radiology* 178-189.