



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**LINE OF SIGHT ANALYSIS USING A FEEDFORWARD
NEURAL NETWORK AND ONE-METER RESOLUTION
DIGITAL ELEVATION MODEL (DEM) MAP DATA**

by

John M. Grant

September 2020

Thesis Advisor:
Co-Advisor:

Brian M. Wade
Joshua H. Gordis

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2020	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE LINE OF SIGHT ANALYSIS USING A FEEDFORWARD NEURAL NETWORK AND ONE-METER RESOLUTION DIGITAL ELEVATION MODEL (DEM) MAP DATA			5. FUNDING NUMBERS	
6. AUTHOR(S) John M. Grant				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) The importance of maximizing one's Line of Sight (LOS) while minimizing enemy LOS is of critical importance in war. LOS between an observer and a target exists if a straight-line vector between the observer and target is not intersected by terrain. Many sensors and kinetic or non-kinetic weapons and enablers require intervisibility between the shooter and target for employment. A means to analyze a terrain map and determine one's LOS would aid route planning onboard aircraft to minimize exposure to ground-based sensors. Furthermore, most LOS programs are computationally expensive to run at scale, making any such analysis on board small aircraft generally unavailable to analyze a large terrain set or to analyze many LOS vectors between formations of sensors/shooters and targets. An LOS machine-learning estimate may solve this problem by reducing computational time, allowing a large number of LOS calculations to be performed with relatively small computation resources found on a laptop. Rapid and computationally efficient LOS calculations would aid warfighters in either maximizing their LOS (such as for an anti-aircraft missile placement) or minimizing their LOS (such as for a vulnerable helicopter needing to hide from potential enemies). The goal of this work is to determine whether such a machine-learning model can reduce the computation time for a large set of LOS calculations as compared to traditional LOS calculation methods with minimal loss in accuracy.				
14. SUBJECT TERMS line of sight, neural network, machine learning, terrain analysis, path planning, Python, Keras, DEM			15. NUMBER OF PAGES 131	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**LINE OF SIGHT ANALYSIS USING A FEEDFORWARD NEURAL NETWORK
AND ONE-METER RESOLUTION DIGITAL ELEVATION MODEL (DEM)
MAP DATA**

John M. Grant
Civilian, Department of the Navy
BS, California State Polytechnic University Pomona, 2011
MS, University of Nevada, Reno, 2012

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN ENGINEERING SCIENCE
(AEROSPACE ENGINEERING)**

from the

**NAVAL POSTGRADUATE SCHOOL
September 2020**

Approved by: Brian M. Wade
Advisor

Joshua H. Gordis
Co-Advisor

Garth V. Hobson
Chair, Department of Mechanical and Aerospace Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The importance of maximizing one's Line of Sight (LOS) while minimizing enemy LOS is of critical importance in war. LOS between an observer and a target exists if a straight-line vector between the observer and target is not intersected by terrain. Many sensors and kinetic or non-kinetic weapons and enablers require intervisibility between the shooter and target for employment. A means to analyze a terrain map and determine one's LOS would aid route planning onboard aircraft to minimize exposure to ground-based sensors. Furthermore, most LOS programs are computationally expensive to run at scale, making any such analysis on board small aircraft generally unavailable to analyze a large terrain set or to analyze many LOS vectors between formations of sensors/shooters and targets. An LOS machine-learning estimate may solve this problem by reducing computational time, allowing a large number of LOS calculations to be performed with relatively small computation resources found on a laptop. Rapid and computationally efficient LOS calculations would aid warfighters in either maximizing their LOS (such as for an anti-aircraft missile placement) or minimizing their LOS (such as for a vulnerable helicopter needing to hide from potential enemies). The goal of this work is to determine whether such a machine-learning model can reduce the computation time for a large set of LOS calculations as compared to traditional LOS calculation methods with minimal loss in accuracy.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	BACKGROUND	1
B.	LOS BACKGROUND	1
C.	PROBLEM STATEMENT	3
D.	LITERATURE REVIEW	4
1.	Prior Work	4
2.	Past LOS Research	5
3.	DEM Maps.....	7
4.	Machine Learning (ML).....	8
5.	NN Background.....	10
II.	SIMULATION SETUP	21
A.	METHODOLOGY	21
B.	COMPUTER PROGRAMMING SELECTION	22
C.	DEM MAPS AND DATA PREPARATION.....	24
D.	NN ARCHITECTURE	26
E.	HYPERPARAMETER OPTIMIZATION.....	28
F.	LOS CALCULATIONS	30
G.	ISSUES ENCOUNTERED.....	32
III.	SIMULATION RESULTS	35
IV.	CONCLUSIONS	41
A.	RESULT EXPLANATION.....	41
B.	RESULT APPLICATIONS	42
C.	FUTURE WORK.....	42
	APPENDIX A: BATTLE HISTORY	45
	APPENDIX B: NN HISTORY.....	47
	APPENDIX C: DEM MAP SCAN	49
	APPENDIX D: COMBINE CSV FILES.....	55
	APPENDIX E: ALTITUDE FILTER.....	57

APPENDIX F: CSV ROW COUNT.....	59
APPENDIX G: LOS PREDICTION.....	61
APPENDIX H: HYPERPARAMETER JMP14 SCRIPT.....	65
APPENDIX I: HYPERPARAMETER SCRIPT.....	77
APPENDIX J: NN TRAINING DATA SCRIPT	85
APPENDIX K: NN PREDICTION SCRIPT	95
APPENDIX L: ORIGINAL LOS CODE.....	99
APPENDIX M: ORIGINAL NN CODE.....	103
APPENDIX N: DEM MAP COVERAGE	105
LIST OF REFERENCES	107
INITIAL DISTRIBUTION LIST	113

LIST OF FIGURES

Figure 1.	An F-15 climbing out of Rainbow Canyon, Death Valley, CA (August 9, 2018). Source: [4].	3
Figure 2.	An observer over terrain. Source: [5].	5
Figure 3.	ML structures. Source: [16].	8
Figure 4.	Learning structure concepts. Source: [24].	10
Figure 5.	Deep learning representation of an image classification model. Adapted from [25], [26].	11
Figure 6.	Neural Network architecture. Source: [28].	12
Figure 7.	Multinode schematic. Adopted from [28].	14
Figure 8.	A two-layer neural network parameterized by its weights and biases. Adapted from [16], [25], [26].	15
Figure 9.	NN Forward Propagation signal (red). Source: [30].	15
Figure 10.	Solution space for loss function. Source: [23].	17
Figure 11.	Thesis simulation flowchart.	22
Figure 12.	LOS physical description.	32
Figure 13.	TPR vs. FPR define the ROC and AUC. Source: [40].	35
Figure 14.	Coastline training, validation and testing predicted accuracies (%) vs. distance (meters).	36
Figure 15.	Mountain training, validation and testing predicted accuracies (%) vs. distance (meters).	37
Figure 16.	Plain training, validation and testing predicted accuracies (%) vs. distance (meters).	38
Figure 17.	1/3 arc-second U.S. map coverage, highlighted in orange. Source: [20].	105
Figure 18.	1-meter DEM U.S. map coverage, highlighted in brown. Source: [20].	105

Figure 19.	1/9 arc-second U.S. coverage map, highlighted in orange. Source: [20].	106
Figure 20.	2 Arc-Second DEM Alaska map coverage, highlighted in orange. Source: [20].	106

LIST OF TABLES

Table 1.	The most common activation functions. Adapted from [25], [27].	19
Table 2.	PC hardware and software used.	23
Table 3.	Number of usable maps per state, divided up into coastlines, mountains and plains.	26
Table 4.	Coastline training, validation and testing, along with associated AUCs.	36
Table 5.	Mountain training, validation and testing, along with associated AUCs.	37
Table 6.	Plain training, validation and testing, along with associated AUCs.	38
Table 7.	NN vs. analytical LOS calculation times.	39
Table 8.	Hyperparameter test numbers.	65
Table 9.	Hyperparameter test values.	70

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AAA	Anti-Aircraft Artillery
API	Application Program Interface
CLFOS	Cloud Free Line of Sight
CSV	Comma Separated Values
DEM	Digital Elevation Map
EANN	Evolutionary Artificial Neural Network
ELU	Exponential Linear Unit
FFNN	Feed Forward Neural Network
FPLOS	Finite Time Predictor-based Line of Sight
GPU	Graphics Processing Unit
HL	Hidden Layer
LOS	Line of Sight
MCP	McCulloch-Pitts
ML	Machine Learning
NN	Neural Network
ReLU	Rectified Linear Unit
RF	Radio Frequency
TNM	The National Map
UAV	Unmanned Aerial Vehicle
SAM	Surface-to-Air Missile
SGD	Stochastic Gradient Descent
USGS	United States Geological Survey

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I'd like to acknowledge several folks who were important to this thesis. LTC Brian Wade provided the opportunity for this thesis to happen, and his advice and help were immense and indispensable. Dr. Joshua Gordis provided guidance and advice on not only the thesis, but on classwork as well.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Line of sight (LOS) is defined as an unobstructed straight-line view between an observer and the observed. The importance of maximizing one's LOS while minimizing enemy LOS is of critical importance in war. LOS between an observer and a target exists if a straight-line vector between the observer and target is not intersected by terrain for a given observer and target height. Many sensors, kinetic or non-kinetic weapons, and enablers require inter-visibility between the shooter and target for employment. A means to quickly analyze a terrain map and determine LOS would aid the warfighter in either maximizing his LOS (such as for an anti-aircraft missile placement) or minimize his LOS (such as for a vulnerable transport helicopter needing to hide from potential enemies as it moves). Further, most LOS programs are computationally expensive to run at scale, making any such analysis generally unavailable to analyze a large terrain set or to analyze many LOS vectors between formations of sensors/shooters and targets. A machine learning estimate of the LOS may solve this problem by reducing computational time by allowing a large number of LOS calculations to be performed with relatively small computation resources found on a common laptop. LOS maximization and minimization applies to several military and engineering topics. A tightly related subject is that of viewsheds, which will be discussed shortly. Concealment from enemies, maximizing LOS for reconnaissance, and radio frequency (RF) LOS problems all need a quick way to calculate the LOS of a geographical region. A manner to quickly calculate LOS for modern battlefields would have a myriad of applications, including such vehicles as helicopters, aircraft, UAVs, weapon placements and more.

B. LOS BACKGROUND

The tracking of ones LOS is of critical importance. Identifying and using concealment (minimizing LOS) has been a critical element in warfare throughout history. Such concealment and hiding depends greatly on the LOS between attackers and defenders,

on whether they can see each other or not. See Appendix A: Battle History for a brief historical look at how concealment and a minimized or maximized LOS matters in real life.

A viewshed is a collection of LOS vectors from a common origin. Putting it another way, it is the view of a geographical area from a specific vantage point [1]. Viewsheds are often used in terrain analysis, especially in regards to urban planning, military operations, and archaeology. For the purposes of this thesis, a viewshed will be represented by what is determined to be visible in each 2-dimensional LOS analysis.

Helicopters, which often fly close to the ground and within range of not just surface-to-air missiles (SAMs) but small arms fire and anti-aircraft artillery (AAA), would benefit with a means to quickly calculate a minimum LOS flightpath on demand. Helicopters such as the AH-64 Apache, OH-58 Kiowa, AH-1Z Viper, AH-1 SuperCobra, V-22 Osprey, and AH-60 Huey, among others, depend on concealment (via terrain features like valleys, forests, and mountains) to minimize LOS in order to approach an enemy target safely [2].

Beyond helicopters, vehicle like fighters and UAVs, would benefit with a quick way to calculate their LOS in order to keep up with changing battlefield conditions. Numerous attack aircraft, such as the F-15, F-16, F-18, F-22, F-35, A-10, RQ-2A, MQ-11L, RQ-3A, among others, use terrain to conceal their position from adversaries, minimizing LOS. Such low-flying tactics are known as “nap-of-the-Earth (NOE), and is effective against enemy detection, such as SAMs, AAA and high flying Airborne Warning and Control System (AWACS) aircraft [3]. Figure 1 shows an example of an F-15 fighter using a terrain feature (a canyon) and high speed maneuvering to approach a target. Alternatively, reconnaissance aircraft will want to maximize their LOS to gather as much intelligence as possible with their cameras and sensors. Lastly, RF applications would benefit a quick way to calculate LOS. Radio antennas, if placed in geographical locations to maximize their LOS, could increase their effective communication range. Radars would benefit for similar reasons.



Figure 1. An F-15 climbing out of Rainbow Canyon, Death Valley, CA (August 9, 2018). Source: [4].

C. PROBLEM STATEMENT

LOS calculations are critical in a variety of military and aircraft routing problems, but calculations of LOS at scale are infeasible onboard most aircraft. The goal of this work is to determine if a supervised machine learning model can reduce the computation time and requirements for LOS calculations as compared to traditional LOS calculation methods with minimal loss in accuracy. One of the most common supervised machine learning models that has seen much use and success is the feedforward neural network (FFNN). This thesis investigates if a FFNN can reduce the computational time for a large set of LOS calculations, with a minimal loss of accuracy, compared to more traditional LOS calculation methods.

In order to define LOS calculations in the context of neural networks (NNs), a history and groundwork must be set. What will follow includes, the importance of LOS calculations, what digital elevation maps (DEMs) are and how they were used, prior work, a LOS literature review, and what NNs are and why this method was used over that of others.

The paper organization is as follows. Chapter I introduces the background, problem statement, literature, prior work, review and NNs. Chapter II covers the simulation setup, addressing the methodology, DEM maps, NN architecture, hyperparameters, LOS calculations, and issues encountered. Chapter III gives simulation results. Chapter IV offers conclusions based on the results found. Finally, a list of appendices are given.

D. LITERATURE REVIEW

A literature review revealed insight into prior research. The aim is to determine what work has been done on viewsheds, LOS, and how these applied to DEM maps. Prior work conducted at NPS provides a backdrop as to the literature review.

1. Prior Work

A significant amount of work was conducted prior to this thesis. LTC Brian Wade, CPT Sean Clement, and MAJ James Jablonski were the initial project leads on an effort to determine if a NN could learn LOS calculations, and if such NN learning was faster than analytical LOS methods [5]. This proof-of-principle effort showed promise and that a NN could learn LOS calculations for a small map. To begin the process of generalizing this initial effort, they also explored synthetic terrain generation. The following scripts were written to test the LOS/NN concept:

- **LOS Calculations:** Using the script *1-syntheticLinearterrainandLOS.py*, synthetic terrain was generated, and a LOS calculation upon that terrain was performed (Appendix M: Original NN Code).
- **NN:** The script *2-linearNNETSyntheticTest.py*, would operate a simple NN upon the previous generated terrain, with the goal being to train the NN using this data (see Appendix M: Original NN Code).

Several insights were found. The most important insight was that it appeared feasible for a NN to learn LOS calculations. These two programs formed the basis for generating NN training data. Figure 2 shows an observer (blue dot) hovering over terrain.

This terrain was color-coded blue or red, depending if that observer can see that terrain or not (Appendix G). Figure 2 illustrates the concept behind the LOS calculations performed.

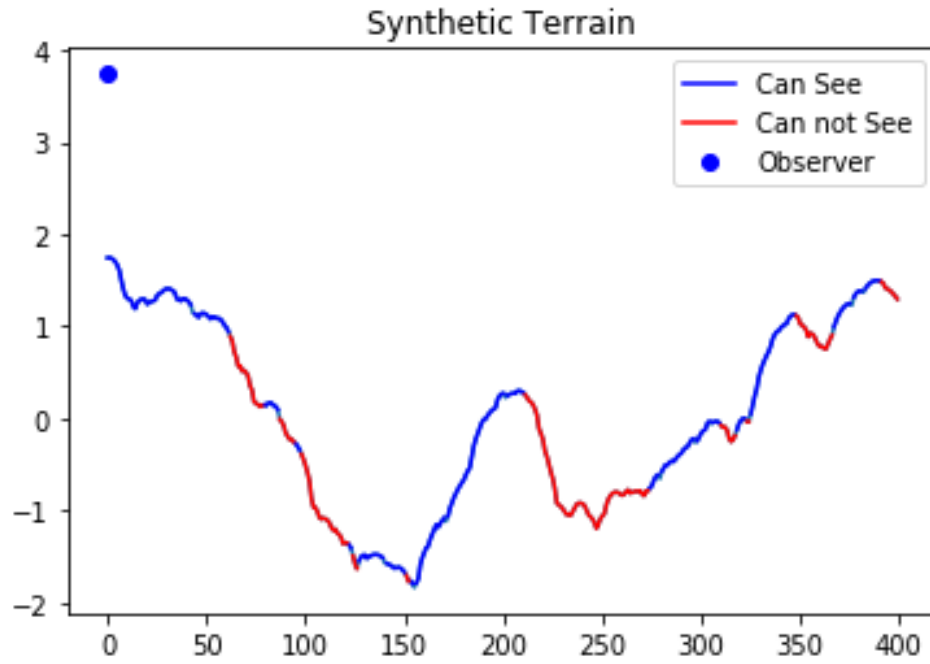


Figure 2. An observer over terrain. Source: [5].

2. Past LOS Research

Given inter-visibility's importance to military and non-military applications, there has been a significant amount of literature on the topics of LOS, DEMs, viewsheds and NN methods. The main goal of this literature search is to determine if this thesis problem has been investigated before, what happened in the past, and how that helps now.

Research on LOS has appeared in the literature, some of which found its way into this thesis computer scripts. As early as the 1950s, a means to determine LOS in regards to a computer terrain simulations was discussed. The means to determine LOS was by the "plane-tilt" test (grid-post algorithm), whereby a square grid would be created and the nearby elevations factored into whether or not other square grids were visible [6]. This grid-post method was implemented in this thesis's LOS calculations. LOS calculations have also been performed over real terrain data, in light of a simulated combat theater,

similar to what this current thesis is trying to solve [7]. In regards to combat, battlefield simulations upon 1-meter terrain maps have been used to help assess weapon effectiveness [8]. While not immediately useful, this research did make use of 1-meter resolution geographic data, just like this thesis. Beyond battlefields and into geography, algorithms for viewshed analysis have been made for more efficient geographic data analysis [9]. To fulfill combat applications, the Air Force has used CFLOS probability methods as tactical decision aids to assist combat aircraft in putting munitions on target [10]. Efforts to make LOS calculations more efficient have been undertaken, especially with an Implicit Min/Max KD-Tree method (though none seem to make use of NNs) [11].

Beyond practical LOS applications, a variety of NN methods and advancements were found that pertained to the problem statement, in regards to NNs. To conduct NNs in a timelier manner, faster CPUs and GPUs are needed. GPUs, having increased in processing power and come down in cost over the years, running LOS calculations on them has seen increasing use [12], [13]. These developments have led to efficient and scalable algorithms for calculating viewsheds via DEMs being developed [14]. As a result of this research, NN tools are available for anyone to use. Software tools such as Keras and TensorFlow have given rise to NN software that is accessible to the public [15], [16], [17].

The above research topics are but a small section of a much larger array of NN applications. As NNs are still a developing field, many more exciting discoveries and applications are yet to be found. The closest to an actual NN analysis using LOS was applications that assisted driverless car guidance in urban environments [18]. No clear cut examples emerged regarding solving LOS calculations with a NN.

LOS calculations, DEMs, and NNs have seen widespread use, and sometimes have been used together. However, despite searching, no instance of using an FFNN to quickly calculate an approximate LOS as compared to LOS analytical methods was found. As such, this thesis's goal will be to focus on generalizing the proof-of-principle efforts that showed that a FFNN could be used to calculate LOS to larger and more varied terrain sets.

3. DEM Maps

The LOS analysis typically uses a type of map data called Digital Elevation Models (DEMs) and such DEMs provided the basis of all the data studied in this thesis. DEMs are regularly spaced elevation arrays, referenced either to a Universal Transverse Mercator (UTM) projection or a geographic coordinate system [19]. In other words, DEMs are simply Earth terrain maps in a digital form, not unlike a topographic map one may find via Google Maps. There are a wide variety of methods to create DEM maps. These methods include LIDAR, radar, stereo photogrammetry (aerial surveys), interferometry, global positioning system (GPS), topographic maps, Doppler radar, and range imaging [19]. Instrumentation can be carried by satellites and aircraft.

Beyond DEM maps, there are other candidates for realistic terrain. Early on, there was a notion to create 3-dimensional random maps for use in a LOS program. It was found to be unwieldy and generally not a true real world representation, with an “uncanny valley” effect with even realistic, synthetic terrain. No synthetic example was convincingly mimicking real geography, and would have been time consuming to learn and implement such a program. It was decided to use DEM maps of real terrain as the primary data set.

The DEM map data used in this thesis is from the USGS “The National Map (TNM)” website [20]. The U.S. maps are geographical representations and associated territories. For this thesis, only the continental U.S. was considered; however, not all the data covers the U.S. equally. From Appendix N: DEM Map Coverage it is apparent that total geographical DEM coverage varies greatly, depending on resolution. Each DEM map (or height map) is composed of a number array (a raster file or raster data sets), composed of x-, y- and z-axis coordinates, and saved as .img files [21]. The map elevation is broken up into square areas, with each side being what defines the DEM map name (1-meter DEM means the square is 1-meter on a side, 1/3 arc-second DEM means each square is ~3.4-meters on a side, etc.). While a significant number of data types are available, several were of interest, including: 1 arc-second DEM, 1 meter DEM, 1/3 arc-second DEM, 1/9 arc-second DEM, 2 arc-second DEM – Alaska, and 5 meter DEM (Alaska Only) [19], [20].

This work will use the 1-meter resolution DEM maps. The reasons include that they are the highest resolution maps publicly available, they cover a fair portion of the United States, and they have an adequate variation in terrain types (for this thesis, the focus is on coastlines, mountains and plains). Military bases and other sensitive areas were avoided. Having defined what DEMs are, exploring machine learning (ML) methods to make use of the DEM data is in order. A background in ML will build a foundation to understand NNs, the basis for this thesis.

4. Machine Learning (ML)

ML methods provide a foundation for NNs. There are three different ML types, such as supervised, unsupervised and reinforcement learning (Figure) [16]. Each ML learning method provides a conceptual context the how and why of NNs.

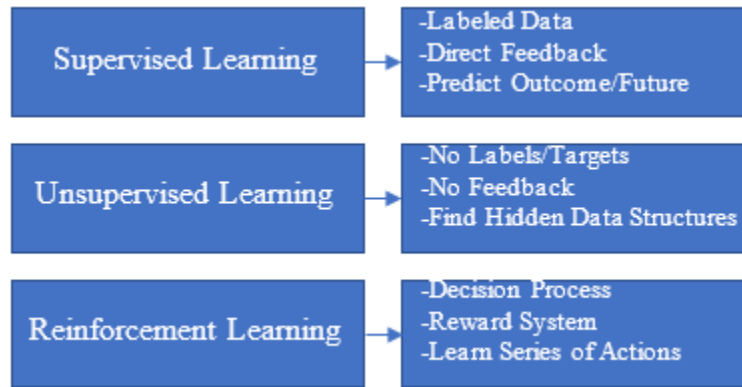


Figure 3. ML structures. Source: [16].

Supervised learning (upon which this thesis FFNN is based upon) involves developing a statistical model from labeled training data with known correct outputs, allowing one to make predictions on unseen or future data, a “validation set” [16], [22]. For each measured observation (x_i , $i=1\dots n$) there is an associated output measurement y_i , with this set being called labeled data [23]. Possible problems supervised learning can help solve includes regression and classification problems [16], [24].

Problems with a quantitative response are called regression problems, where the goal is to predict a given response value [24], [25]. In simplistic terms, think of x- and y-axis data (known inputs with outputs), where there is a need to overlay a function upon the data in order to make a prediction. Typical methods of solving regression problems include neural networks, random forests, and linear regression methods. Typical data is usually continuous and quantitative.

The goal of classification problems is to predict the categorical class labels for a new instance of data, learning and creating rules that distinguish between individual data points [16], [25]. Methods of solving classification problems include logistic regression, linear discriminate analysis, random forests, and neural networks. Typical data is usually discrete or qualitative.

Unsupervised learnings goal is to understand the relationships between variables or observations by grouping them into object classes, without labeled data, or with data of an unknown structure type [16], [24]. For every observation $i=1\dots n$, there is an associated vector of input measurements x_i , but no associated response output y_i [24]. Typical tasks may include identifying object groupings and reducing data dimensionality [24]. Dimensionality reduction aids in removing data noise, while maintaining most of the predictive performance [16]. Common techniques for conducting unsupervised learning may include association methods, factor analysis, and principal components [24]. In other words, unsupervised learning infers a structure [24]. See Figure 4 for a visual representation [23]. The goal of reinforcement learning is to develop a system (agent(s)) that improves model performance based on discrete environmental (state) interactions, using a reward system to help determine appropriate actions to take [16], [24]. The goal of reinforcement learning is to find a policy that maximizes the long-term expected reward. An example of reinforcement learning would be programs that play Chess, which get better with the passing of each game [16].

Reinforcement learning is composed of both reward and value functions [16], [24]. Reward functions, through each state, provide either positive or negative rewards, with each reward being defined as achieving some sort of goal. Value functions provide an estimation of each state and action pair, based upon the reward.

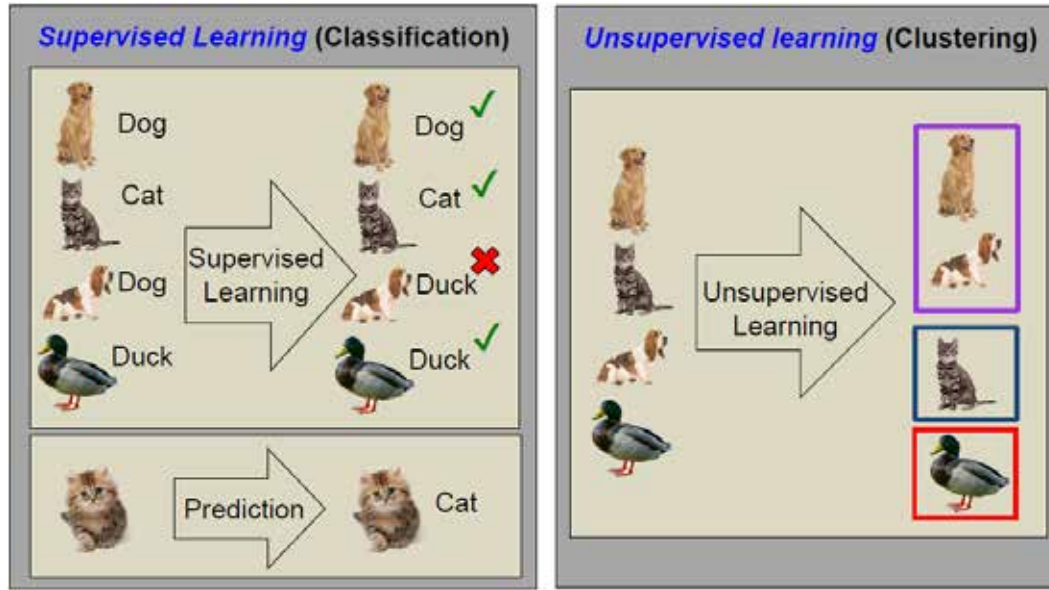


Figure 4. Learning structure concepts. Source: [24].

Reinforcement learning chooses an output based on a perceived world state and an associated reward through trial and error [24].

5. NN Background

An exploration into NN (in particular, deep learning) would build upon the ML foundation laid earlier, eventually connecting NNs with LOS calculations. This section presents an overall view of deep learning, NN mathematics, and a description of weights, biases, optimizers, loss functions, activation functions, FFNNs, forward propagation, and back propagation. All of these functions will be described later in this chapter. See Appendix B for NN history.

The NN for this thesis relied on deep learning (Figures 5 and 6). A NN is composed of an assembly of neurons, possibly including multiple neuron layers. Deep learning refers

to successive hidden (representation) layers of neurons, each layer being an attempt to breakdown a complex problem into simpler, constituent parts [25]. As an example of how a complex data set can be broken down into simpler constituents, Figure 5 shows how a digital image (an arbitrary data set) can be broken down into successively simpler representations.

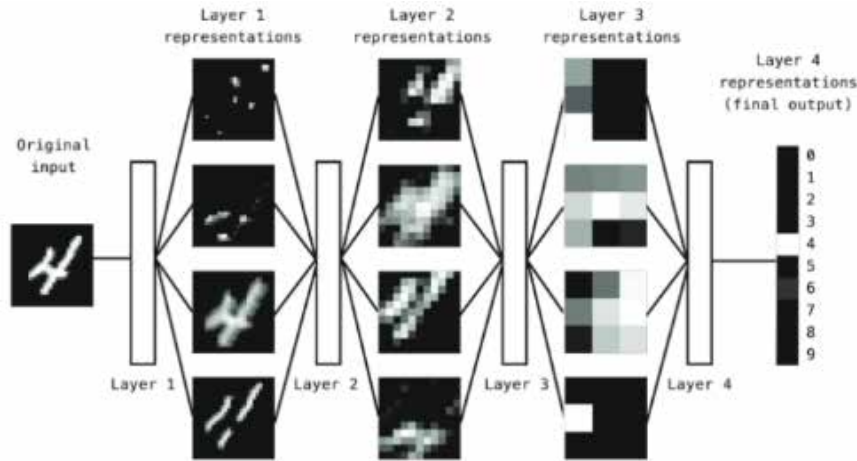


Figure 5. Deep learning representation of an image classification model. Adapted from [25], [26].

Think of it as an array of numbers, which can represent an image, a matrix, collected data, and so on. While this is a narrow example, it does illustrate how information gets distilled from data. Deep learning is like a multistage information distillation process, with data going through filters and being increasingly purified [25], [26]. Deep learning can be realized via a NN.

An illustration showing the connections between neurons and their associated layers would provide a visual representation to the mathematics that will soon follow. Figure 6 shows a basic understanding of each individual neuron, and their connections between layers [27]. Figure 6 gives an overall view of how NNs are assembled. The NN architecture includes an input layer, at least one hidden layer, and an output layer (Figure 6). Figure 6 illustrates how 3 input neuron (P_1 , P_2 and P_3) connect to two hidden layer

neurons, with one output neuron. A more detailed descriptions of each part is given later (activation functions, loss functions, optimizers, etc.).

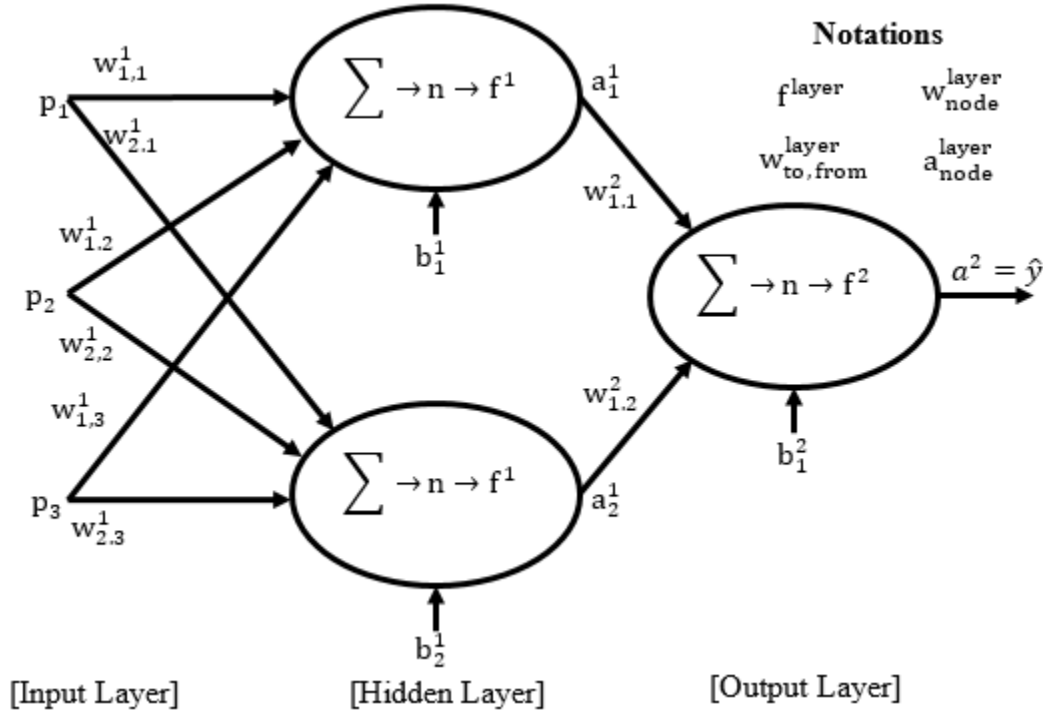


Figure 6. Neural Network architecture. Source: [28].

Equations 1 through 4 provide mathematical context to Figure 6 [29]. Biases (b) are applied, along with weights (w) via backpropagation (to be discussed later). Equation 5 is a combination of all fore-mentioned equations and gives the NN output value $\hat{y}$ [28].

$$a = ([\sum wp] + b) = f([w_{1,1}p_1 + w_{1,2}p_2 + w_{1,3}p_3 + \dots + w_{1,R}p_{1,R}] + b) \quad (\text{Equation 1})$$

$$\hat{y} = a^2 = f^2(w_{1,1}^2 a_1^1 + w_{1,2}^2 a_2^1 + b_1^2) \quad (\text{Equation 2})$$

$$a_1^1 = f^1(w_{1,1}^1 p_1 + w_{1,2}^1 p_2 + w_{1,3}^1 p_3 + b_1^1) \quad (\text{Equation 3})$$

$$a_2^1 = f^1(w_{2,1}^1 p_1 + w_{2,2}^1 p_2 + w_{2,3}^1 p_3 + b_2^1) \quad (\text{Equation 4})$$

$$\hat{y} = f^2 \left(\begin{array}{c} w_{1,1}^2 [f^1(w_{1,1}^1 a_1^1 + w_{1,2}^1 a_2^1 + b_1^1)] + \dots \\ \dots w_{2,1}^2 [f^1(w_{2,1}^1 p_1 + w_{2,2}^1 p_2 + w_{2,3}^1 p_3 + b_2^1)] + b_1^2 \end{array} \right) \quad (\text{Equation 5})$$

where [28]:

- p equals the input neuron.
- w equals the weight of the input edge to node.
- a equals the node layer activation function.
- b equals node bias.

As more neurons and hidden layers are added, matrix algebra comes into play [29]. As more neurons are added, the number of calculations required rapidly rises to astronomical levels. However, due to the power of modern computers, NN calculations are easily completed, especially if using a GPU for parallel processing. In order to understand the mathematical background, several actions are taken through the neuron, such as an input vector, bias vector, weight matrix and activation function (see Figure 7). The input vector is a single-column input layer (P) that inputs data into the input layer [28] [30]. Here, R equals the number of inputs variables, S equals the number of nodes. The bias vector b ($S \times 1$) inputs values (one per node per layer) into each activation function [28] [30]. The weight matrix w ($S \times R$) is composed of R weights (one per input) entering the Sth row per node [28] [30]. An activation function is applied to the output vector ($S \times 1$) [28].

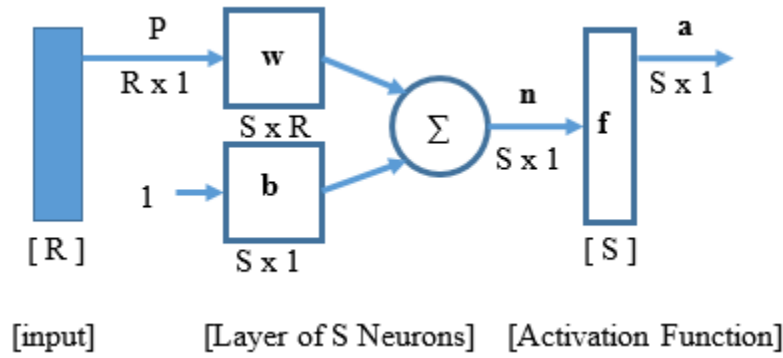


Figure 7. Multinode schematic. Adopted from [28].

Figure 7 shows the conceptual way a NN works via linear algebra [25]. Input vector P ($R \times 1$) is weighed, and enters the input layer, layer 1. After an activation function (a function that decides whether or not to pass an input signal), the processed data moves on to the second hidden layer, having a bias added to it. The new vector ($S \times 1$) goes to the output layer to give a prediction $\hat{y}$ [27]. This predictions accuracy is measured by the loss function, which gauges how wrong the prediction is compared to the true value given in the training data [25]. An optimizer estimates weights and biases that may improve the NN prediction, and the cycle iterates over and over again [25]. This supervised learning process can be compared to teaching a student mathematics by giving them a set of quizzes that contain questions with answers (training set). After the student has memorized each quiz, a new quiz with no answers is administered (validation set), and the student (based on his prior memorization of patterns) answers the quiz with a reasonable amount of accuracy. Figure 8 shows a more conceptualized form of Figure 7. This is the way that the NN will learn to conduct LOS calculations. A large set of LOS calculations is performed, the FFNN memorizes this training set, and then a new LOS calculation set is introduced to test this FFNN to see if it has learned to make predictions.

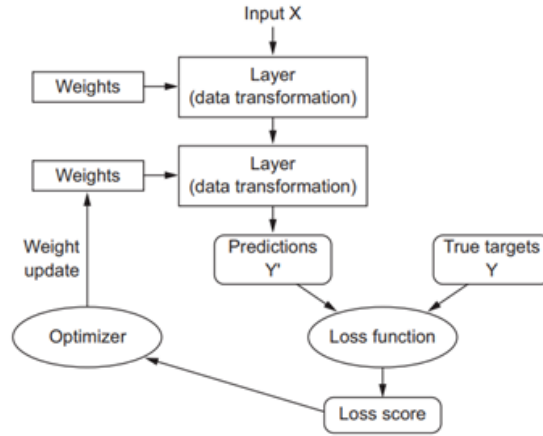


Figure 8. A two-layer neural network parameterized by its weights and biases. Adapted from [16], [25], [26].

The above processes is called forward propagation. Forward propagation is how the input signals propagate through the NN. Forward propagation, simply stated, is what the action is called as each neuron activates, and sends a signal to the next layer, as described in the above diagrams and equations. More concisely, forward propagation occurs when a nodes output in a given layer (using a weighted sum of a previous layer nodal output) is generated, with a bias added [28]. The final signal consists of an activation function applied to it and the signal propagated [28]. Figure 9 shows forward propagation in action, as a signal moves from the input layer all the way to the output layer, to be repeated in a new iteration, or epoch [30].

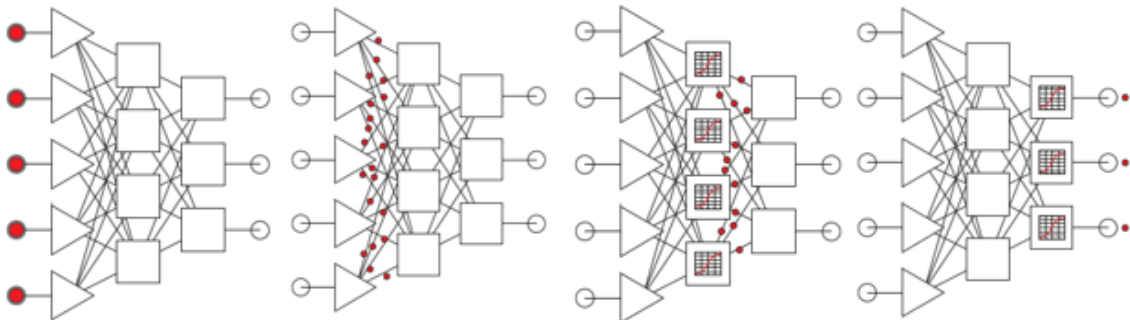


Figure 9. NN Forward Propagation signal (red). Source: [30].

Forward propagation is the culmination of the previous mentioned equations. Equation 6 and Equation 7 give the first hidden layer activation function, and all the n^{th} hidden layers after that. Equation 8 is the general form of forward propagation [28]. Representing a NN as a matrix simplifies the overall design and makes it easier for computers to process [28].

$$a^1 = f^1(w^1P + b^1) \quad (\text{Equation 6})$$

$$a^n = f^n(w^n a^{n-1} + b^n) \quad (\text{Equation 7})$$

$$a^n = \begin{bmatrix} w_{11}^n & w_{12}^n & \dots & w_{1R}^n \\ w_{21}^n & w_{22}^n & \dots & w_{2R}^n \\ \vdots & \vdots & \ddots & \vdots \\ w_{S1}^n & w_{S1}^n & \dots & w_{SR}^n \end{bmatrix} \begin{bmatrix} a_1^{n-1} \\ a_2^{n-1} \\ \vdots \\ a_r^{n-1} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{bmatrix} \quad (\text{Equation 8})$$

(S x R) (R x 1) + (S x 1)

where n is the current layer identification number [28].

The adjustments to the weights and bias occur during the training process using stochastic gradient descent optimization algorithms [31]. The loss function is how NNs measure their performance on the given training data, which will help guide the NN towards better model accuracy, based on the training data inputs and outputs [31]. This gradient descent can be thought of as a solution space, where the combinations of weights and biases give a particular model accuracy. The “descent” portion of gradient descent is the derivative (or slope) of any point on that solution space, which may have local maximums and minimums. In the simplest terms, a loss function measures what is predicted vs. the actual value, which can guide the optimization algorithm to the best solution (if one exists) [31], [32]. Figure 10 gives a visual demonstration of this [23], [32].

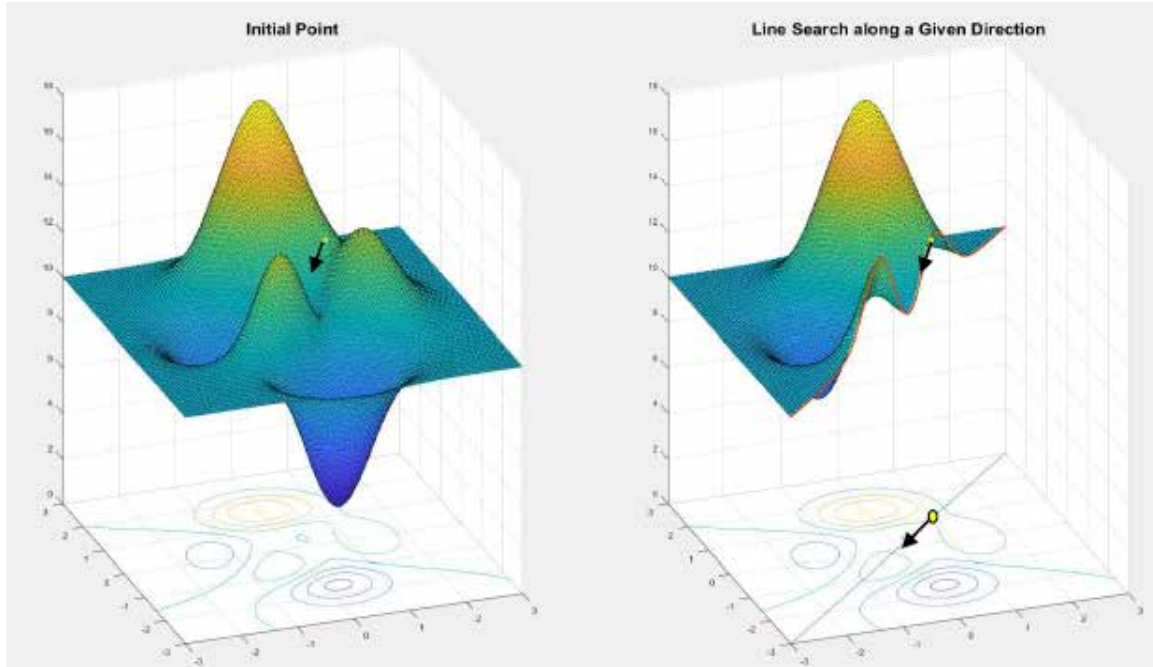


Figure 10. Solution space for loss function. Source: [23].

In order to make NN models achieve any appreciable accuracy, a loss function must be selected. It is up to the person writing a NN to select which loss function is most appropriate, as not all are applicable to any given problem. The loss functions include, but are not limited to, Mean Squared Error, Mean Absolute Error, Mean Absolute Percentage Error, Mean Squared Logarithmic Error, Hinge, Squared Hinge, Categorical Hinge, Logcosh, Huber Loss, Binary Cross-entropy, Sparse Categorical Cross-entropy, Kullback Leibler Divergence, Cosine Proximity and Categorical Cross-entropy [31], [33]. Experimentation and research on what has been tried before is key to determining what loss function works best for a given application.

While loss functions help guide the NN to better predictive accuracy, that predictive accuracy is dependent on how the action functions are used. To recap, an activation function is a function that decides whether a neuron is going to output a signal or not, by calculating the weighted sum, and adding a bias to it [34], [35]. NNs without non-linear activation functions are essentially just fancy linear regression models [34]. Since activation functions perform non-linear transformation to their inputs, this makes the overall NN capable of learning and doing more complex tasking [34].

Choosing an activation function is dependent on the application. Assuming binary output, a sigmoid activation function might be best, whereas if the programmer doesn't know the best option, ReLU is often the default activation function of choice, due to its versatility and ability to resist the vanishing gradient problem [34], [36]. Experimentation might be required, along with a more thorough investigation into the hyperparameters (discussed later in this thesis).

Activation functions can take the form of (but not limited to) a step, linear, tanh or sigmoid shapes. Depending on which activation function is used, the output can be binary (-1 to 1 or 0 to 1) or as a fractional number inside such boundaries [25]. One can think of activation functions as akin to the action potential of biological neurons, producing a binary output (1 for a fixed voltage output, 0 for no signal). If the activation function values are emerging from an output layer, these values are collected as the predicted $\hat{y}$ [16]. Table I gives a list of possible activation functions.

There are at least two concerns regarding activation functions, the vanishing gradient problem and the exploding gradient problem. The vanishing gradient problem happens when the neuron weights and biases don't update appreciably, limiting learning significantly [36]. Assuming Equation 9's partial derivative $\partial C / \partial w^{(L)}$ is vanishingly small, then the "vanishing gradient problem" becomes apparent [36]. Weights and biases, no matter what the learning rate is, won't update appreciably if the partial derivative is tiny. This lack of appreciable updating will prevent further NN accuracy improvements. However, some activation functions may overcome this limitation, such as ReLU, ELU or Leaky ReLU [36].

$$w^{(L)} = w^{(L)} - \text{learning rate} \times \frac{\partial C}{\partial w^{(L)}} \quad (\text{Equation 9})$$

Table 1. The most common activation functions. Adapted from [25], [27].

Name	Input/Output Relation	Range
Hard Limit	$(a = 0)$	$n < 0$
	$(a = 1)$	$n \geq 0$
Symmetrical Hard Limit	$(a = -1)$	$n < 0$
	$(a = +1)$	$n \geq 0$
Linear	$a = n$	$-\infty$ to ∞
Saturating Linear	$(a = 0)$	$n < 0$
	$(a = n)$	$0 \leq n \leq 1$
	$(a = 1)$	$n > 1$
Symmetric Saturating Linear	$a = -1$	$n < -1$
	$(a = n)$	$-1 \leq n \leq 1$
	$(a = 1)$	$n > 1$
Log-Sigmoid	$a(n) = \frac{1}{1 + e^{-n}}$	0 to 1
Hyperbolic Tangent Sigmoid	$a(n) = \frac{e^n - e^{-n}}{e^n + e^{-n}}$	-1 to 1
Positive Linear	$(a = 0)$	$n < 0$
	$(a = n)$	$0 \leq n$
Competitive	$(a = 1)$	Neuron with max n
	$(a = 0)$	All other neurons
Exponential Linear Unit (Elu)	(n)	$n > 0$
	$a(e^n - 1)$	$n \leq 0$
Rectified Linear Unit (ReLU)	n	$n > 0$
	0	$n \leq 0$
Leaky ReLU	n	$n > 0$
	an	$n \leq 0$

The exploding gradient problem is the opposite of the vanishing gradient problem, and is a significant issue [36]. In this situation, the weight and bias values explode in value rapidly [36]. With the correct selection of activation function and using a method called gradient clipping, this value explosion can be limited or prevented [36]. Gradient clipping is a method upon which a maximum value is specified (such as 0.5 or 1) [36]. However, this does not imply or guarantee that you avoid the vanishing gradient problem.

While activation function selection is critical, choosing the optimizer is also important in solving NN problems. The optimizer (optimization algorithm) is a function that uses the loss function gradient to adjust the weights and bias. The loss-space is very noisy with multiple problem areas for a standard gradient decent algorithm to include local

optimal, saddle points, and relatively flat areas. Each optimizer uses various methods to help the gradient descent overcome issues with these problem areas and adjust the weights and bias terms in a stepwise fashion towards a solution [37]. Looking at Keras, several optimizer algorithms exist, including Stochastic Gradient Descent (SGD), SGD with Nesterov Momentum, Adagrad, Adadelata, RMSprop, Adam, AdaMax, Nadam and AMSgrad [37].

Having listed each NN functions, this leads to an issue. How to select the best combination of values for activation functions, loss functions, optimizers, etc.? These parameters are known as hyperparameters, as these cannot be picked by the program, only by the programmer. Hyperparameters are variables that the programmer must choose; the program cannot pick them. Such hyperparameters may include the number of neurons, batch size, the number of hidden layers, optimizer selection, learning rate, loss function, and more [25], [31]. Hyperparameter selection will be expanded on in Chapter II.

Chapter I covered a lot of ground. The stage has been set for determining if NNs can calculate LOS faster than direct LOS methods. The problem background, problem statement, LOS, DEM maps, literature review, prior work, LOS applications, ML background, NN background, activation functions, loss functions, and optimizers were discussed. This background on how NNs work provided a glimpse at a whole world of knowledge available, and it lays the groundwork for understanding Chapter II. Chapter I set the foundation, Chapter II builds the solution upon that foundation.

II. SIMULATION SETUP

Chapter I laid the foundation and background to understand how an FFNN will solve a LOS calculation faster than direct LOS methods. Chapter II will introduce the solution methodology, selection of a computer programming language, DEM map selection and preparation, NN architecture, hyperparameters selection, analytical LOS calculations, and encountered issues. While Chapter I spelled-out the problem statement, Chapter II explains how the problem is solved.

A. METHODOLOGY

The simulation setup was written in Python, compiled in Anaconda-Navigator, Spyder 4.0.1 (to be elaborated on later). In order to determine if NNs can calculate LOS faster than any analytical method, several steps are required, including data collection, hyperparameter selection, training the NN, and producing a prediction file. How each step was done are given later in this thesis. For data collection, 2-dimensional map scans were conducted on the downloaded DEM maps, extending out to 1000-meters with 1-meter resolution (DEM maps can be either .tif or .img files.). A total of 200 scans per map were done, using random azimuth angle for each scan, and saved in a .csv file. Using a space-filling experimental design, a combination of hyperparameters that gave the highest NN model accuracy were found. A NN script was written that accepted the data training set (as a .csv file) and uses the selected hyperparameters. Once a data set was trained, an .h5 file was generated which provided the ability to transfer the trained NN to other programs. Using the trained NN (imported with a .h5 file), test a new NN with a new test data set. Additionally, time the script and compare that time to the traditional LOS calculation methods on the same data set. By comparing the time it took to complete a trained NN to finish calculating a LOS over a given range with that of an analytical LOS program, a means to gauge the NN can be created.

The thesis methodology included several steps. Figure 11 shows the overall thesis flowchart, starting from downloading DEM maps, preparing the data, training the NN, and

calculating the final accuracy and computation time of the traditional LOS method and the NN.

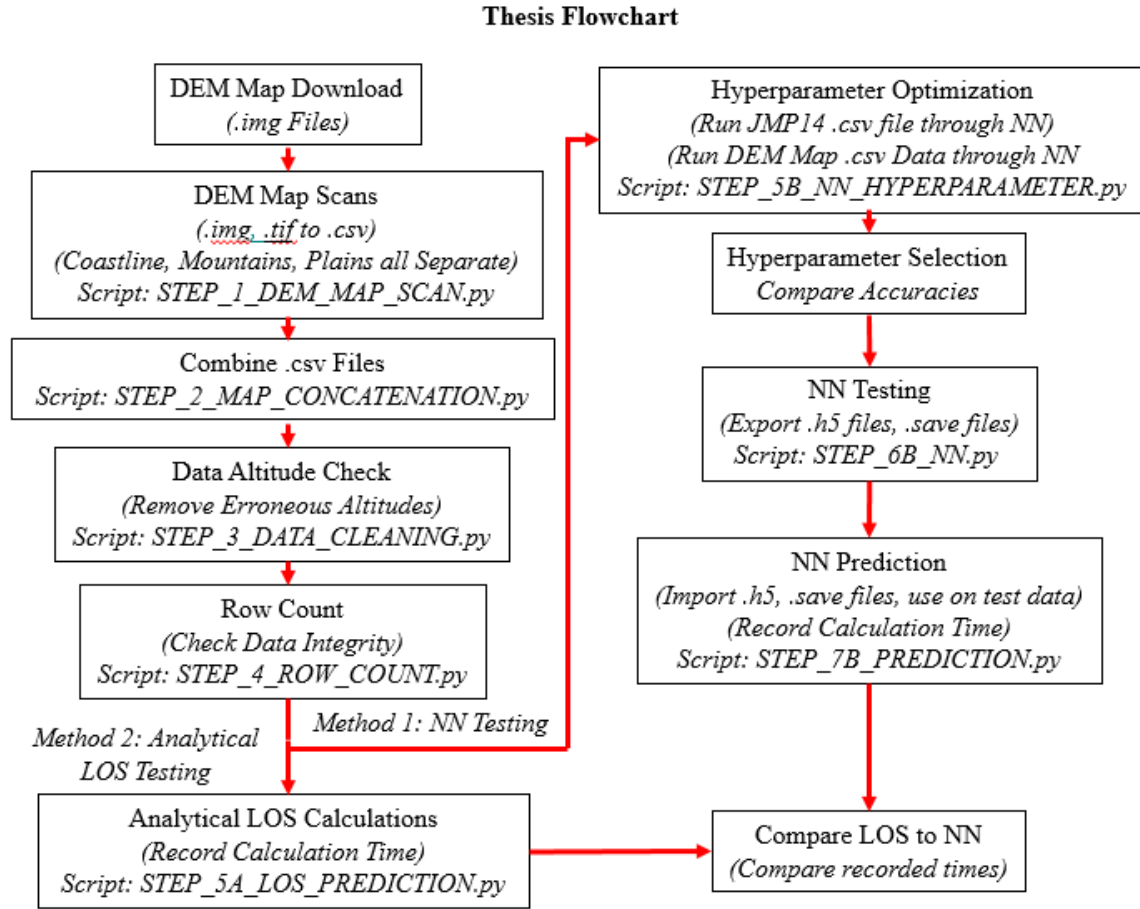


Figure 11. Thesis simulation flowchart.

B. COMPUTER PROGRAMMING SELECTION

Two programming languages were considered to write the FFNN, Matlab 2019B and Python 3.7. Both languages were considered as each had the requisite tools needed to create the NN architecture desired. Ultimately, the author chose Python for a variety of reasons. The first of these was licensing. Matlab required an internet connection to verify its license keys, without which it would not function. Python has no such issues. Second, Matlab has an extensive NN library, but Python makes use of the industry-standard Tensorflow and Keras libraries. These open source Python and libraries are used must more

throughout industry than MATLAB based on a review of the literature. Additionally, an active open source community addresses user questions and a variety of textbooks are available to help new users. Finally, Python is a new language to the thesis author. The author had extensive experience in MATLAB and wished to learn a new computer language.

These advantages led to Python being selected. However, Python is not without its own issues, those issues are listed below. The first being regular updates. Python, being an open-source computer language, presented a double-edged sword. While regular updates can provide new capabilities, Python functions can become defunct and older programs may no longer work correctly. Matlab is more stable in this regard. The second issue is GPU usage. The Tensorflow GPU implementation was not totally stable, and GPU usage was limited to NVIDIA GPU graphics cards as of the year 2020. Matlab has a much more integrated and stable GPU coding. Finally, Python is slower than some other languages, such as C and C++. It has comparable speed to MATLAB, and for this thesis this wasn't a problem. Python compiling times were still reasonable and C and C++ don't have extensive NN libraries like Python has [38].

As a future reference, Table 2 gives the hardware and software used to write and compile the coding used in this project. Table 2 provides a background so as hardware improves and software changes, a reference exists to ascertain any issues with the code and hardware listed here and in the appendices.

Table 2. PC hardware and software used.

Variable	Values
Operating System	Ubuntu 18.04.4 LTS (Linux-x86_64)
Memory Size	31.4 GiB
Processor	Intel® Core™ i5-4690K CPU @ 3.50GHz x 4
Scripting Language	Python 3.7
Environment	Anaconda-Navigator 1.9.7
Code Compiler	Spyder 4.0.1

C. DEM MAPS AND DATA PREPARATION

The DEM map preparation included several steps. The goal was to take the publically available DEM maps and determine the LOS of each 1000 data point scan. This data provided the basis to not only train any FFNN designed, but also provide a means to test the LOS analytical method too. The test in question was to make a comparison between the FFNN and analytical method completion times in Python.

The steps required in order to prepare the needed LOS calculations are given below in some detail [20]. Data preparation began by visiting The National Map (TNM) website (<https://viewer.nationalmap.gov/basic/>). On the website, under “Elevation Products (3DEP)” select “1-Meter” DEM. Select “Show Availability” for a color-coded overlay of the U.S., displaying what is available for that map type. Divide up the map downloads into coastline, mountains and plains, and save into their own separate folders. Once a given geographical area is selected for testing, zoom into the given area (this can vary from the size of a town, a U.S. State, or the entire U.S. (depending on how far you have zoomed in or out), and select “Find Products.” A list of DEM maps will be presented, based on the geographical map area chosen. Next, map integrity must be checked. Not all maps on the TNM website are complete, some lack geographical data (represented as blacked-out areas), and such data (or lack thereof) is visible via the “Thumbnail” button under “Actions.” Some maps are intersect on TNM, yielding overlapping terrain data. Once a map has been selected, click “Download” under the “Actions” column. Download every map deemed required. The downloaded map data come either as .zip files or .tif files. Unzip the .zip files and saved in a folder. The unzipped data folders each contain either a .tif or an .img file. Remove these files into their own folder which must contain only .img or .tif files (Caution: it is critical there are only .img or .tif files, otherwise the future map scans will not work correctly). For this thesis, only .img files were used.

After the data download and file selection, the data was prepared and made ready for use. The readability of each map file was determined, to uncover any unreadable maps. Using the program *STEP_1_DEM_MAP_SCAN.py*, and setting the variable step equal to 1, scan small batches of .img files (~10 files), until you have scanned every map that you

have saved. Setting step to 1 limits the number of runs on each map to 1, which was found to be enough to determine whether the map was readable or not.

Once a group of .img files have been proven to be readable, the full map scan may begin. From *STEP_1_DEM_MAP_SCAN.py* (Appendix C), set the variable step to 20 (20 randomly oriented scans per .img map). The output will be a .csv file. As the data is broken up into mountains, plains and coastlines, there will be three separate map file scans, one for each terrain type. Each .csv file contains six columns, with a full scan 1000 rows deep. Those six values are x coordinate, y coordinate z (altitude) coordinate, scan angle, result (zero or one) and filename. If scanning all the .img files is too much to run at once (often due to memory limitations), scanning smaller batches and concatenating them together using script *STEP_2_MAP_CONCATENATION.py* (Appendix D) is helpful. The script *STEP_3_DATA_CLEANING.py* (Appendix E) deletes erroneous altitude values. An altitude value error came up during some scans, with a value of -3.402823×10^{38} meters. The after mentioned program is written to remove data sets that contained this error (so if a 1000 row scan contained at least one erroneous altitude error, all rows were removed to prevent partial data sets from appearing). These partial data sets would otherwise train the NN incorrectly. A final check involved counting the available .csv rows (*STEP_4_ROW_COUNT.py*, Appendix F). If the row count is divisible by 1000, then that indicated there is no partial data sets. The data collection and processing is completed after checking for bad altitude data.

Three terrain types (coastline, mountains, and plains) were selected to give what is believed to be a good representation of U.S. geography. Table 3 gives which U.S. states 1-meter DEM maps came from, how many viable maps were used, and what terrain type they were. The goal was to ideally have 150 maps per given terrain type (Table 3). There was a total of 108 coastline maps, 150 mountain maps and 150 plains maps.

Table 3. Number of usable maps per state, divided up into coastlines, mountains and plains.

State	Terrain Types		
	Coastlines	Mountains	Plains
California	28	44	
Florida	18		
Louisiana	27		
Texas	35		
Colorado		56	
Tennessee		50	
Kansas			49
Nabraska			48
South Dakota			53

There were several issues that led to fewer coastline maps as compared to the other two terrain types. There was a limited selection of available 1-meter DEM coastline maps to pick from. Many coastline maps were not complete, and the available data didn't extend very far out to sea (just off most coastlines). Areas like Texas and Louisiana, with barrier islands and inlets, provided the best selection of coastline, while California maps often provided maps where one map edge comprised the coastline, making some maps mostly dry land (making California coastlines different from other states). Much more terrain was available inland from the oceans as compared to the coastline. Whenever a mountain or plains map proved to be unreadable, more DEM maps could be simply downloaded elsewhere, which was not the case for coastlines.

D. NN ARCHITECTURE

The terrain data provides the training data for a supervised learning model. The model for this thesis is a FFNN. The FFNN was written in Keras, a high-level interface that runs on top of Tensorflow. Keras provides functionality that allows for ease of use, including loading, manipulating, and normalizing data. Said data can be broken up into training, testing, and validation sets, for further use in preparing the FFNN for practical use.

A FFNN is a simple NN that feeds information in an input layer, contains a hidden layer or layers, and generates results from an output layer. Compare this to recurrent or convolutional NNs, and FFNN provides an advantage in that it is relatively simple to make.

The FFNN has several characteristics worth noting. The goal of FFNNs (also known as multilayer perceptrons) is to approximate a given function [27]. Equation 10 gives a basic example of a function to be modeled. Where y was the output data, x was the training data and θ are the training parameters that give the best function approximation upon a given data set [27].

$$y = f(x; \theta) \quad (\text{Equation 10})$$

To implement the FFNN, the application program interface (API) known as Keras is used. Keras is written in Python and is designed for easy programming of NNs. Keras is built over TensorFlow, the underlying programming that makes creating NNs possible in Python.

Keras allows all the functionality needed in order to process data and make sense of it [15]. Data types such as NumPy Arrays, Tensorflow Dataset Objects expand what one can do with NNs. Keras builds models in “layers”, allowing a programmer to write “layers” of a NN in a sequence, allowing for an easier coding experience. Keras allows for training a NN and evaluating the results.

The current FFNN was written in Python via Spyder 4.0.1. To better understand how the FFNN was written, the following components were implemented in sequence. First, specify any needed code libraries (Numpy, Pandas, Keras, etc.). Set a timer to determine how long the code ran for once executed. This timer will be the basis of comparison between the analytical LOS calculations and NN LOS calculations. Any variables (number of hidden nodes, training set size, etc.) was added next, clearly labeling all variables. The file directory, along with any .csv files came next. The data was rearranged, scaled and divided up into training and test data sets. The NN was specified via “model” and “model.add”, specifying an input layer, hidden layers, and output layer, and all associated activation functions. (an optimizer was also specified). Additionally, the

model was evaluated by executing it, and scores recorded. Lastly, as the data scaling must be maintained in all present and future data, a scaler file was created, to be imported into any future model.

FFNN construction was detailed in this section. Data preprocessing, Keras sequence construction, file directories, optimizers, model evaluations and program outputs were briefly touched upon. Having specified an overall NN architecture, a description on how that architecture was optimized must be given, in the form of hyperparameters.

E. HYPERPARAMETER OPTIMIZATION

The FFNN hyperparameters were optimized in a series of steps, detailed here. To recap, hyperparameters are program parameters that the programmer must select, the NN cannot pick them. The goal was to determine hyperparameters ideal for each terrain type, coastal, mountains and plains (so three separate NNs). The output would be two file types, .h5 files and a .save file. The .h5 file contained the FFNN weights and biases, while the .save file maintained a common data scaling factor. Maintaining the scaling factor was important, as without it, any new data that scaled beyond zero or one would give false positives.

The following steps were conducted, both to generate map data, and testing out the hyperparameters. A .csv data sets for mountains, plains and coastline was created (using 12 maps per terrain type, instead of the full data set). See section C: DEM Maps and Data Preparation for data set construction. Using the software program JMP14, a space-filling design that tested out numerous hyperparameter combinations, including batch size, hidden layers, number of neurons per hidden layer, and learning rate was created. The output would be the model accuracy to gauge performance.

The NN hyperparameter optimization selection was done via a space-filling design in the software package JMP14. This both tested out numerous possible parameter combinations in the solution space, and generated a useful table of values that can be iterated over by a Python program. See Appendix H and I for how the hyperparameters were tested and setup.

The following hyperparameters needed optimization [15], [25]:

- **Epoch:** The number of iterations the training set would be run through the FFNN to update the weights and biases appropriately.
- **Optimizer:** Several optimizers were available, including SGD, RMSprop, Adam, Adagrad, Adadelata, Adamax, and Nadam.
- **Dropout Value:** In each hidden layer, the possibility exists that some neurons will update often, while others languish. To fix this, a random selection of neurons are switched “off” during a given epoch to force other neurons to update.
- **Alpha:** This value is associated with a particular activation function, a Leaky ReLU (defined further below). Alpha determines the gradient slope coefficient. The default value (which was used) was 0.3.
- **Batch Size:** The number of data sets to input at one time during one epoch.
- **Hidden Layer 1 (HL1):** The number of neurons in layer 1.
- **Hidden Layer 2 (HL2):** The number of neurons in layer 2.
- **Hidden Layer 3 (HL3):** The number of neurons in layer 3.
- **Hidden Layer 4 (HL4):** The number of neurons in layer 4.

See Table I for a list of activation functions. The activation functions used in this project included ReLU, Elu, Leaky ReLU and Sigmoid. The Rectified Linear Unit (ReLU) is the most commonly used activation function. The ReLU generates either a zero (for negative values) or one (for positive values, Equation 11) [39]. This act of rounding to zero or one facilitates gradient descent (it mostly eliminates the vanishing gradient problem) [36].

$$\text{ReLU}(x) = \max(0, x) \quad (\text{Equation 11})$$

The Exponential Linear Unit (ELU): The ELU fixes several issues with ReLU while keeping some of its positives. For input values greater than zero, it activates just like a ReLU, but if a negative input value is inputted, the output value is slightly below zero as the outcome (Equation 12). ELU avoids the “dead relu” problem, but the downsides include longer computational time, and it can potentially experience the exploding gradient problem [36].

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases} \quad (\text{Equation 12})$$

Equation 13 shows that the input equals the output for values greater than x . However, negative value inputs are multiplied by a α value (usually between 0.1 and 0.3) and generate a negative. This solves both the “dead relu” problem, and avoids the vanishing gradient problem [36]. The downside is that there is no avoidance of the exploding gradient problem [36].

$$\text{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \quad (\text{Equation 13})$$

The Sigmoid activation function was used for the final output NN layer. This logistic function generates an output between zero and one (see Equation 14). While the sigmoid function struggles with the vanishing gradient problem, it generates outputs that are rounded to either zero or one, indicative of what we expect (zero for when we cannot visually see a data point, one when we can).

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \quad (\text{Equation 14})$$

F. LOS CALCULATIONS

At the heart of this thesis lies the analytical LOS calculations. The answer as to whether the NN predictions or direct LOS calculations is faster is partly covered in this section. Equations 15 and 16 show the initial LOS calculations that were implemented in this thesis.

$$OG = \frac{\text{Rise}}{\text{Run}} = \sum_{i=0} \frac{[(E(i)+h_{\text{obj}}(i))-(E(0)+h_{\text{oh}})]}{d(i)} \quad (\text{Equation 15})$$

$$MHL = OG(i) \sum_{i=0} d(i) + E(0) + h_0 \quad (\text{Equation 16})$$

Supervised learning relies on the error between the model and the true value (known as labeled data). In the context of this thesis, this means that the actual binary value that specified if LOS existed between the target and each point in each training vector has to be calculated using the traditional LOS method before the FFNN could be trained. The traditional method is outlined in Figure 12.

LOS is calculated in several steps. A line is calculated between the observer and the final data end point, forming part of a triangle. This consists of 1000 data points, 1-meter wide. Compare the terrain data points with the height of the previously calculated hypotenuse. If the terrain data point is less than the hypotenuse, then it is within view of the observer. Once a data point has been found that is above the initially calculated hypotenuse, a new hypotenuse is calculated (LOS line). Once this occurs, any terrain data points that are beneath this LOS line are considered unobservable. Once another point is found to be above the LOS line, another hypotenuse LOS line is found.

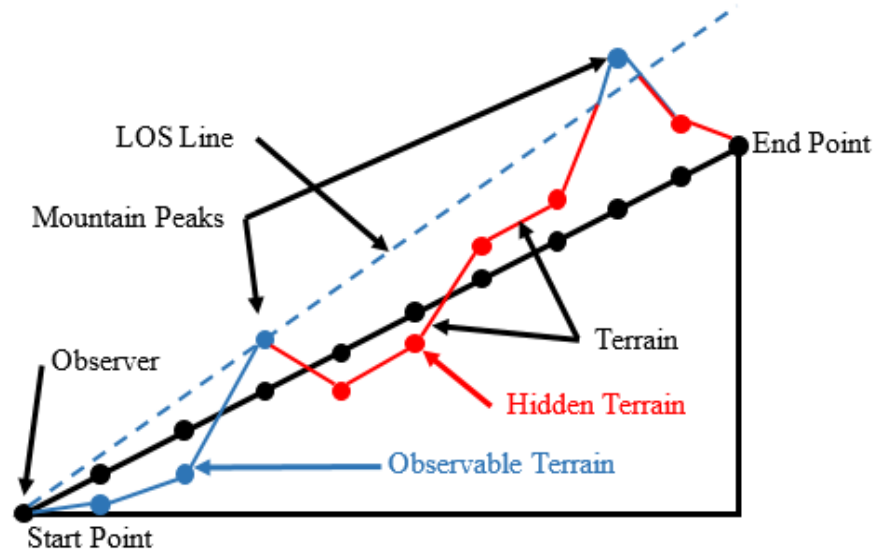


Figure 12. LOS physical description.

G. ISSUES ENCOUNTERED

While the LOS analysis, both NN and analytical, were carried out successfully, issues arose. Those issues included unrealistic altitude values, incomplete data, software issues, running out of memory, and more. Those issues are detailed below.

- GPU Driver Corruption:** In the process of setting up TensorFlow, the GPU drivers were corrupted, and had to be reinstalled. Getting the GPU to perform calculations was abandoned.
- Altitude Data Errors:** It was found that in some map scans, all the altitude values would read -3.402823×10^{38} meters, necessitating their removal, and checking to see if the resultant data set was still divisible by 1000 (a check for partial data sets, since all sets should contain exactly 1000 rows). See Appendix E for this error removal process.
- Memory Issues:** The map data completely filled up all 32 GB of RAM available if left unchecked. This was fixed by doing smaller, more numerous map scans to better manage the memory, and clearing arrays and variables after each for-loop run.

- **Partial DEM Maps:** Care had to be taken to avoid DEM map problems. Before downloading a map, it had to be checked for any blacked-out areas (no data), overlapping areas between DEM maps, and unreadable downloaded maps.
- **Unreadable DEM Maps:** A particularly frustrating issue was that, while most .img map files were successfully downloaded, a few could not be read correctly, preventing a map scan and halting the map scanning program. Carefully scanning small batches of .img map files would eventually reveal which .img was the one halting the program, and would be deleted. This scanning method proved slow and tedious.
- **Incomplete Data Sets:** Occasionally, and without reason, the scanned data in each .csv would be incomplete (so instead of 30,000,000 rows of data, we may get 29,999,992). Scans would have to redone, or the smaller .csv data files recombined again.
- **Overwritten Data Files:** In the event one wants to redo a map scan, one must ensure that the prior .csv files have been moved or deleted from the specified file directory. Any new data specified to those file names don't overwrite the old data, and the new data is appended to the old data file.

Chapter II built the proverbial house upon Chapter I's foundation. Chapter II set the stage for determining if a FFNN could calculate LOS faster than a direct LOS calculation method. This chapter defined the steps to complete the needed calculations, and discussed NN architecture, data preparation, hyperparameter selection, issues encountered and LOS calculations. Chapter III takes off from Chapter II, testing what has been made and generating results.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SIMULATION RESULTS

The FFNN simulation runs consisted of two parts, the training and testing results. Both training and testing data made use of three broad map types: coastline, mountains and plains. The training data consisted of all maps downloaded, while the testing data consisted of 12 maps of each terrain type. The training data trained three FFNNs, one for each terrain type (coastlines, plains and mountains) (Appendix J). Once each FFNN was trained, they were tested using the smaller test set. Though the test set maps originated from the training set, they were rescanned and analyzed again, essentially creating unique data (due to randomly selected azimuths) as compared to the training data originally made.

The data for both the training and testing results is given in terms of model accuracy (%), area under the curve (AUC), and receiver operating characteristics (ROC). The accuracy refers to how well the model was able to make a prediction, ROC is a visual representation of the classifier model performance, and AUC is the area under the ROC curve. Figure 13 illustrates this concept, where the true positive rate (TPR) refers to how one may predict a positive result, and observe a positive result, while the false positive rate (FPR) refers to how one may predict a positive result, but observe a negative result [40]. The higher the AUC value (ranging from 1 to 0) the more TPR values you have.

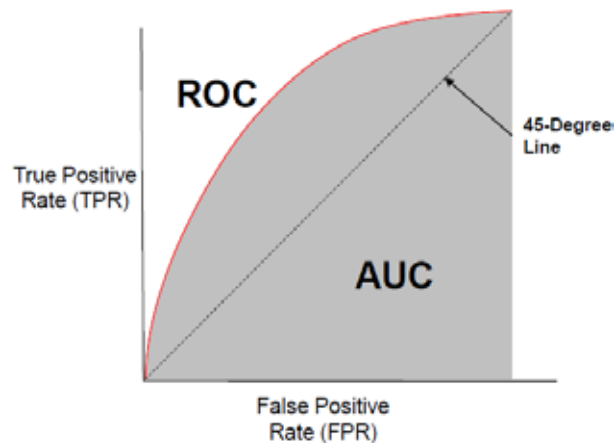


Figure 13. TPR vs. FPR define the ROC and AUC. Source: [40].

Simulation training results were gathered, as shown in Tables 4, 5 and 6. These results are based on the maps of Table 3. Each of the following tables are divided up into training, testing, and validation accuracies, along with the associated AUC values. Figures 14, 15, and 16 graphically illustrate Tables 4 through 6.

Table 4. Coastline training, validation and testing, along with associated AUCs

Distance (Meters)	Training Accuracy (%)	Training AUC	Validation Accuracy (%)	Validation AUC	Testing Accuracy (%)	Testing AUC
100	98.07%	0.92	98.27%	0.92	98.14%	0.92
200	96.48%	0.94	96.60%	0.93	96.38%	0.93
300	94.50%	0.92	94.93%	0.92	94.39%	0.92
400	93.20%	0.91	93.35%	0.9	93.00%	0.9
500	90.94%	0.89	90.94%	0.88	90.94 %	0.88
600	90.03%	0.89	90.00%	0.88	89.76 %	0.89
700	87.82%	0.87	87.42%	0.87	87.76%	0.87
800	87.24%	0.88	86.92%	0.87	87.13%	0.87
900	86.19%	0.88	85.68%	0.87	85.89%	0.87
1000	84.92%	0.88	84.25%	0.86	84.72%	0.87

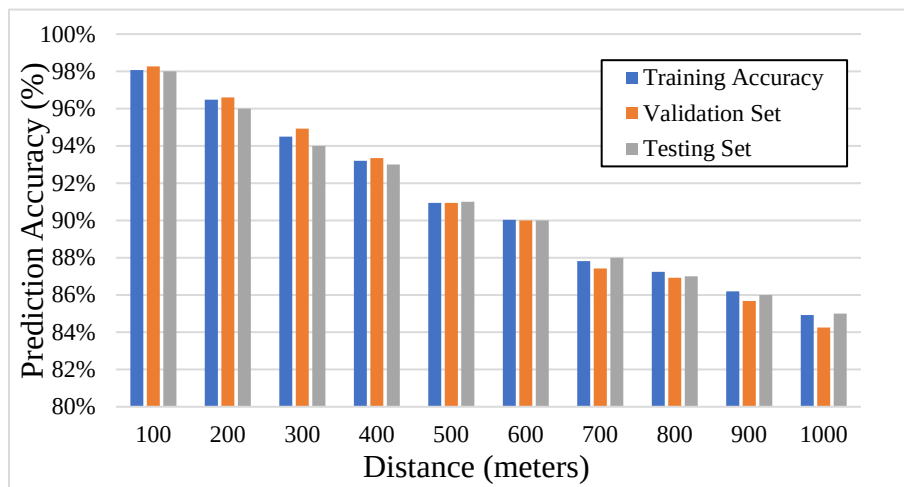


Figure 14. Coastline training, validation and testing predicted accuracies (%) vs. distance (meters).

Table 5. Mountain training, validation and testing, along with associated AUCs.

Distance (Meters)	Training Accuracy (%)	Training AUC	Validation Accuracy (%)	Validation AUC	Testing Accuracy (%)	Testing AUC
100	89.37%	0.78	89.20%	0.78	89.02%	0.79
200	77.65%	0.79	77.35%	0.79	76.84%	0.79
300	72.80%	0.76	72.33%	0.76	72.26%	0.76
400	72.12%	0.76	71.65%	0.75	71.93%	0.76
500	70.11%	0.74	69.73%	0.74	69.68%	0.74
600	68.38%	0.74	68.42%	0.74	68.55%	0.74
700	68.84%	0.74	68.65%	0.74	68.73%	0.73
800	69.71%	0.77	69.60%	0.77	69.67%	0.77
900	70.62%	0.75	70.52%	0.75	70.67%	0.75
1000	71.70%	0.71	71.46%	0.71	71.67%	0.71

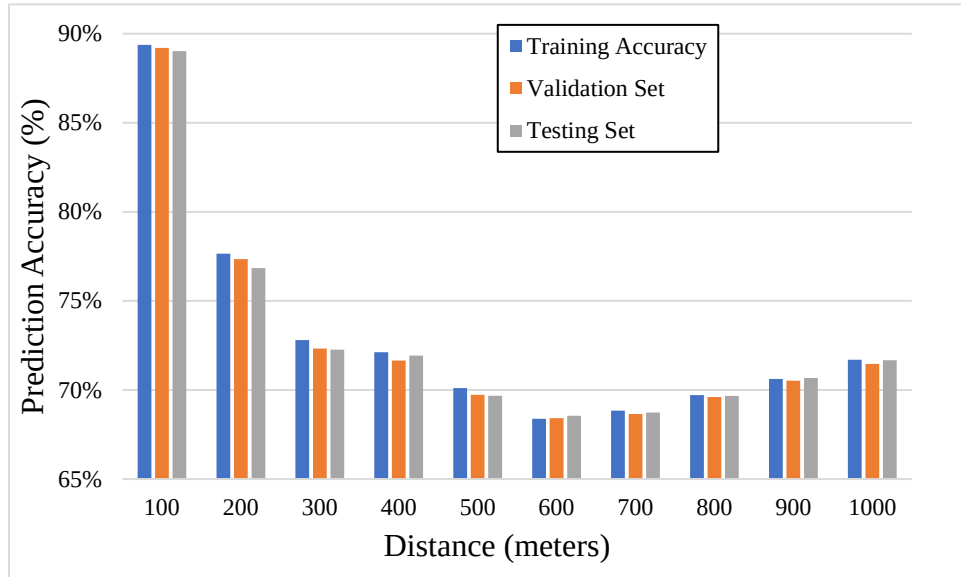


Figure 15. Mountain training, validation and testing predicted accuracies (%) vs. distance (meters).

Table 6. Plain training, validation and testing, along with associated AUCs.

Distance (Meters)	Training Accuracy (%)	Training AUC	Validation Accuracy (%)	Validation AUC	Testing Accuracy (%)	Testing AUC
100	98.83%	0.80	98.87%	0.80	98.79%	0.80
200	95.19%	0.77	95.31%	0.77	94.92%	0.77
300	90.99%	0.76	91.06%	0.76	90.67%	0.76
400	87.02%	0.77	87.13%	0.77	86.73%	0.77
500	83.10%	0.75	83.27%	0.74	82.58%	0.75
600	79.72%	0.74	79.94%	0.74	79.21%	0.74
700	76.73%	0.74	76.89%	0.74	76.27%	0.74
800	74.61%	0.76	74.89%	0.76	74.28%	0.76
900	71.65%	0.74	71.82%	0.74	71.27%	0.74
1000	70.84%	0.74	71.11%	0.75	70.59%	0.74

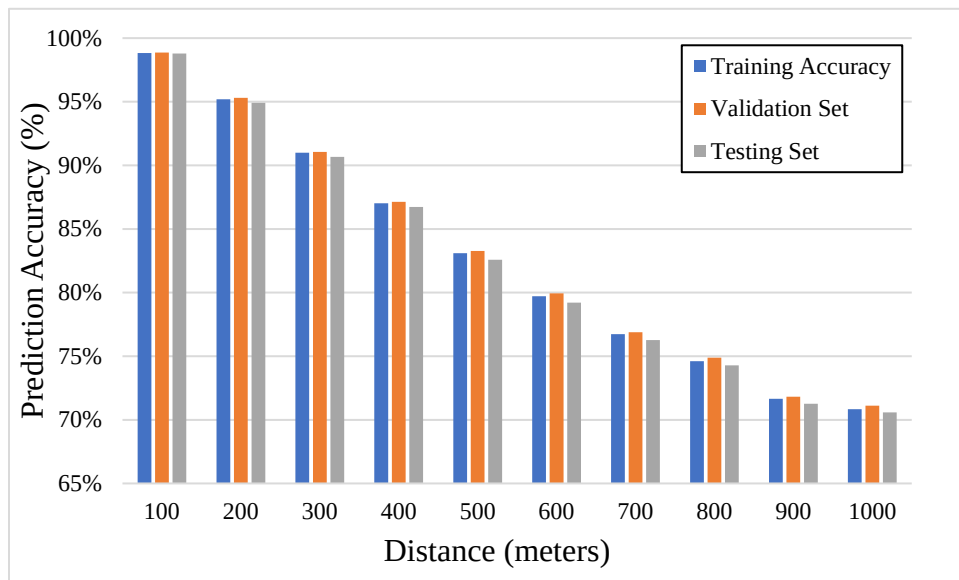


Figure 16. Plain training, validation and testing predicted accuracies (%) vs. distance (meters).

Tables 4 through 6 (and Figures 14 through 16) assume that 80% of the data is training, 10% is validation and 10% is testing. All data assumes the observer is 20 meters above the terrain.

Table 7. NN vs. analytical LOS calculation times.

	FFNN Calculations			LOS Calculations	
Terrain Type	Prediction Accuracy (%)	AUC	Run Time (Seconds)	Run Time (Seconds)	Range (Meters)
Coastline	89.43	0.84	3.02	22.904	600
Mountains	89.96	0.82	3.72	28.863	100
Plains	92.35	0.73	5.09	28.878	300

For Table 7, the results compared the FFNN calculation times and prediction accuracy, and LOS calculation times. While Tables 4 through 6 made use of the entire map data sets downloaded, a smaller data set was created to test the FFNN accuracy.

A data test set was created to test the trained FFNN (Appendix K). Several initial geography conditions were set. The coastline data consisted of four California, Texas and Florida maps. The mountain data consisted of three California, Tennessee, Colorado, and Nevada maps. The plains maps consisted of four Kansas, Nebraska, and South Dakota maps. The ranges with accuracies close too or over 90% are shown in Table 7.

A note must be made on how the data was weighed. The data set is a binary classification problem, with one indicating a distant point from an observer is visible, zero if not. However, there were many more ones then zeros, skewing the data. This would bias the models and made predictions difficult. Thus, the model was weighed with the zeros multiplied by 100, and the ones multiplied by one. By using this multiplicative value, any data point that is a one has its loss function calculated, but then multiply that loss by the weight. This increases the “importance” of a minority class’s loss in back propagation calculations.

Table 5 used the same DEM map data set to test both the NN model, and analytical LOS calculation script.

Chapter III generated results comparing FFNN calculation runtimes with LOS calculation runtimes. It was found that the FFNN, using the same initial data set as the analytical LOS calculations, was able to complete its calculations faster.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCLUSIONS

This thesis presented an extensive background on NNs, and how they can provide faster LOS calculation times (within a given accuracy). Having given a chapter background (Chapter I), simulation setup (Chapter II), simulation data and results (Chapter III). Chapter IV ties it all together, explaining the results. The conclusion is relevant to a variety of aircraft, weapon emplacements and RF communications in that, should there be a means to quickly calculate the LOS in the field, this could enable numerous military applications, as mentioned in Chapter I. Such applications would be minimizing enemy contact, maximizing RF range, maximizing weapon range and more.

A. RESULT EXPLANATION

From Table 4 and 5, several conclusions can be drawn. The FFNNs performed (at the cost of accuracy) LOS calculations faster than analytical LOS calculations. Both plains and coastline accuracies drop off almost linearly as a function of distance from the observer. However, FFNN mountain accuracies level off around ~70%. This might be due to the fact that, since the observer cannot see far in mountain terrain, the majority of terrain is non-observable, thus the data sets consist mostly of zeros. However, mountain peaks might show up (represented by 1's for observable) in that sea of zeros in a fairly consistent manner, keeping the accuracy at ~70%. The accuracies between the training and test sets were very nearly equal, indicating any data features present were uniform across both sets.

For the FFNNs, there was a trade-off between computational speed and model accuracy. From Table 7, a set of ranges were set for what minimum accuracy was desired, and in this case, was set to 90%. This 90% criteria shows that each FFNN (mountains, plains, coastlines) each were accurate out to 100-, 300-, and 600-meters, respectively. Compare this to the analytical LOS calculations, which were 100% accurate at all ranges. However, From Table 7, it is shown that the analytical LOS calculations runs for approximately 28 seconds, vs. 3 to 5 seconds with the FFNNs. It's a tradeoff between accuracy and calculation time.

B. RESULT APPLICATIONS

The results have several potential applications. This thesis concludes that NNs can calculate LOS faster than direct LOS calculations (with an accuracy penalty). The following applications could potentially be realized. In regards to anti-aircraft artillery (AAA) a trained NN can calculate LOS much faster, a method can be setup to scan surrounding terrain, count which scans had the most observable terrain points, and find a location with the most visibility, giving transporter erector sets (TELs) a good location at which to place missiles and radars (find a location that has the most observable locations in range of it). This maximized LOS viewpoint may help better defend a given area, such as a military base, radar, bunker, or other important position. If a flight path can be quickly found that minimized LOS every step of the way, then aircraft survivability may increase. On the flipside, by looking for a maximum LOS flightpath, the best path for reconnaissance aircraft may be found, minimizing the amount of time spent flying might be found (which can save fuel and time). Finding a location with a maximum LOS in a given geographical area would help minimize the required equipment to maintain communications due to the extended range.

C. FUTURE WORK

Possible future work may include several projects. A Matlab program called Flypath3D provides a means to simulate realistic aircraft and missile flight (and strikes) in Matlab. One idea might be to introduce a FFNN that calculates LOS, generate a 3D color-coded map of distance vs. accuracy, and create a path algorithm to minimize LOS (using the Unity game engine is another possible means of displaying these simulations). Battle simulations with aircraft models using realistic physics might provide a more realistic simulation with fewer computational resources, where various battle scenarios may be played out. LOS calculating need not be limited to aircraft. Determining where to put radio antennas and radars can be important too, as it could maximize range and minimize the amount of gear needed for communication. In another FFNN/LOS application, a missile could, in real time, redirect its flight path to dodge a previously unseen AAA or point

defense system along its flightpath. This missile, upon detecting the threat, calculate a new path that minimizes LOS, and avoid destruction.

The possibilities are endless for NNs, as they grow in importance each year. The above applications list is not exhaustive, and there is no doubt many more applications of NNs and LOS calculating can be realized. As NNs have only really taken off in the last decade and are not quite a mature technology as of yet, new possibilities still exist. Such possibilities are applicable to the warfighter, and giving that warfighter one more tool to better prepare for the battlefield.

Having found that an FFNN can calculate LOS faster than traditional LOS methods, the initial premise of this thesis was achieved.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: BATTLE HISTORY

In numerous historical battles the effective use of LOS over terrain and through weather was critical to victory, including:

- **Battle of Teutoburg Forest (9 A.D.):** Occurring within the dense Teutoburg forest inside Germania (modern-day Germany), Germanic leader Arminius launched several surprise ambushes upon 20,000 Roman soldiers, killing most of them over a period of 4 days, concealing his movements via rough terrain and dense foliage [41]. Rome never successfully conquered Germania, though it did launch raids into it from time to time [42].
- **Battle of Trenton (1776):** General George Washington, leading 2,400 troops at night in a snow storm, arrived in Trenton, New Jersey, defeating the surprised 1,800 German Hessians stationed there [43]. This action most likely saved the American war effort to defeat the British.
- **Battle of Isandlwana (1879):** Concealed by terrain, 20,000 Zulu warriors came within 7 miles of a British military encampment without being detected, before collectively breaking into a run and massacring 2,000 surprised troops [44], [45].
- **Battle of Midway (1942):** A force of 16 U.S.M.C. Douglas SBD Dauntless dive-bombers, using cloud cover and diving out of the sun's direction, surprised the Japanese Imperial Navy and sank the carriers Akagi, Kaga, Soryu, and later the Hiryu, all of which had participated in the bombing of Pearl Harbor, Hawaii on December 7th, 1941 [46]. This action crippled Japan's naval air power, leading to that nation's defeat by 1945.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: NN HISTORY

In order to associate military LOS applications to NNs a history and description of ML must be given. ML is a subset of AI [16]. AI was born in the 1950s, at the behest of the early computer science pioneers who wondered if machines could think on their own one day [25], [47]. ML involves self-learning algorithms that derive knowledge from data in order to make predictions, or put in another way, a way to automate intellectual jobs normally performed by humans [16], [25]. Instead of a human sifting through data to derive rules and create models, ML provides a means to more efficiently analyze large amounts of data for knowledge, to gradually improve models [16].

ML methods predate computers. Probabilistic modeling (Naïve Bayes algorithm), logistic regression, and the McCulloch-Pitts (MCP) neuron were some of the earliest methods and research into ML by the 1940s [16], [25].

Programming a computer to act intelligently is not a new concept. When Charles Babbage invented his analytical computer during the 1830s and 1840s, Ada Lovelace (the first computer programmer) commented on this new computer: “The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform...its province is to assist us in making available what we’re already acquainted with” [25]. In contrast to Lovelace, the famed computer scientist, mathematician, logician, cryptanalyst, and philosopher Dr. Alan Turing, in his landmark 1950 paper “Computing Machinery and Intelligence,” pondered if a general purpose computer might be capable of learning, in which he concluded they eventually could [25]. The progression of ML over time is given below:

- **1950s:** Turing test, the first neural networks created, programs that play checkers, Perceptron’s defined further [24], [48] , [49] .
- **1960s:** Quicksort algorithm, computer assisted manufacturing, DARPA AI program, first expert systems, programs that play Tic Tac Toe, nearest neighbor algorithm [24], [50], [51], [52], [53].

- **1970s:** Robotic Arms, backpropagation, Stanford cart, speak and spell systems [24], [54].
- **1980s:** Expert systems, Deep Thought defeats a chess master, neural network research continues, explanation-based learning, recurrent neural networks, NetTalk, backpropagation experiments, reinforcement learning continue [50], [55].
- **1990s:** Deep Blue defeats master chess player Kasparov, Predator drones, Long Short-Term Memory (LSTM), programs that play Backgammon, Random Forests [24], [50], [56], [57].
- **2000s:** Torch ML library, NetFlix Prize, Roomba, DARPA Grand Challenge, Google's deep learning research, ImageNet, Kaggle.com, Microsoft Kinect [24], [50] .
- **2010s:** TensorFlow, Keras, Google Brain, Watson beats humans at Jeopardy!, Siri, AlphaGo beats human Go master, Facebook's DeepFace, Sibyl [24], [58], [50].

The last decade has seen tremendous advances in NNs and ML in general, mainly due to the rise in both inexpensive computing power, and practical methods to conduct NN analysis.

APPENDIX C: DEM MAP SCAN

The first step in creating the data step, the script STEP_1_DEM_SCAN.py reads the .img or .tif files, does a series of randomly oriented scans, collects elevation points, and creates a .csv file. The code is below:

```
#Create DEM Map (.tif, .bil)
#https://www.earthdatascience.org/tutorials/visualize-digital-
elevation-model-contours-matplotlib/
# https://www.geodose.com/2018/03/create-elevation-profile-generator-
python.html
#https://earthexplorer.usgs.gov/

#AUTHORS: LTC BRIAN WADE, MAJ JAMES JABLONSKI, JOHN GRANT

# IMPORT LIBRARIES
import rasterio
import numpy as np
import matplotlib.pyplot as plt
import math
import os, random
import multiprocessing as mp
from timeit import default_timer as timer
import pandas as pd
import csv
import gc

# BEGIN CALCULATION TIMER
start = timer()

# FOR LOOP THROUGH SAVED MAPS

z_list2=[] #For-Loop through all the saved .img files "1-meter resolution
DEM Maps"

#directory = r'/media/gman/Grant/Thesis_Maps/Test_runs/'
#directory = r'/media/gman/Grant/Thesis_Maps/Test_Data/'
#directory = r'/media/gman/Grant/Thesis_Maps/divided_data/data_1'
#directory = r'/media/gman/Grant/Thesis_Maps/divided_data/data_10'
#directory = r'/media/gman/Grant/Thesis_Maps/hyperparameter_Maps/'
#directory = r'/media/gman/Grant/Thesis_Maps/hyperparameter_Maps/Coast_hyperparamete
r_maps/'
#directory = r'/media/gman/Grant/Thesis_Maps/hyperparameter_Maps/Mountain_hyperparam
eter_maps/'
#directory = r'/media/gman/Grant/Thesis_Maps/hyperparameter_Maps/Mountain_hyperparam
eter_maps'
```

```

#directory =
r'/media/gman/Grant/Thesis_Maps/hyperparameter_Maps/Plains_hyperparameter_maps/'

#directory =
r'/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/plains/'
directory = r'/media/gman/Grant/thesis_complete/mountains/'

#random.choice(os.listdir("directory")) #Randomly select a map.
#z_list1=[]
for file in os.listdir(directory):
#for file in random.choice(os.listdir(directory)):
    filename=os.fsdecode(file)
    if filename.endswith('.img'): #also .tif, or .img
        fpath=os.path.join(directory,filename)

        dataset=rasterio.open(fpath).read()[0,:,:]

        # VARIABLES
        pt_spacing = 1 #grid size in meters (m/grid)

        observerHeight=20
        targetHeight=0
        want_plot=1
        want_terrain_set=0

        num_pts = 1000; #300
        step=200; #How many runs to iterate terrain generation.

# CREATE FILENAME ARRAY FOR LATER APPENDING TO OUTPUT LIST
    filename_array=np.full((1,num_pts),filename)

# MAP SCAN VIA FOR LOOP
    z_list1=[] #For-Loop through all 1 map, collecting data at random
points.
    for k in range(0,step):
        # GRID DESIGN
        ray_size = num_pts * pt_spacing #length of ray in meters

        #_, size1, size2 = dataset.shape
        size1, size2 = dataset.shape #number of grids in x and y
directions

        #grids that are the boundry
        left = ray_size -1
        right = size1 - ray_size -1
        bottom = ray_size -1
        top = size2 - ray_size -1

        #Select random start grid within boundry
        x0=round(np.random.uniform(left,right));
        y0=round(np.random.uniform(bottom,top));

        #shoot an azimuth at random direction

```

```

        angle = np.random.uniform(0,2*math.pi) #Randomly select an
azimuth
        angle_array=np.full((1,num_pts),angle) # Rad, Make an array
to be appended to the output file.
        #Find heights of points along azimuth
        x_list = [] #array of x grid points
        y_list = [] #array of y grid points
        z_list = [] #height at grid points

        #array of evenly spaces steps
        step_size = np.arange(0, ray_size, pt_spacing).tolist()

        # GRID LOOP
        for n in step_size:

            #(decimal) grid of step
            x_step = x0 + n*math.cos(angle)
            y_step = y0 + n*math.sin(angle)

            #add the grid to a list
            x_list.append(x_step)
            y_list.append(y_step)

            if n==0:
                ht=dataset.item((x0,y0))
            else:
                #find grid points around the step
                x1 = math.floor(x_step)
                x2 = math.ceil(x_step)
                y1 = math.floor(y_step)
                y2 = math.ceil(y_step)

                #See how far the step is from each grid point
                dis1 = math.sqrt((x_step - x1)**2 + (y_step - y1)**2)
                dis2 = math.sqrt((x_step - x2)**2 + (y_step - y1)**2)
                dis3 = math.sqrt((x_step - x1)**2 + (y_step - y2)**2)
                dis4 = math.sqrt((x_step - x2)**2 + (y_step - y2)**2)

                #Get the heights of each grid point
                ht1 = dataset.item((x1,y1))
                ht2 = dataset.item((x2,y1))
                ht3 = dataset.item((x1,y2))
                ht4 = dataset.item((x2,y2))

                #Height of step is the wighted average of grid points
around it
                ht
                =
                ((dis1*ht1)+(dis2*ht2)+(dis3*ht3)+(dis4*ht4))/(dis1+dis2+dis3+dis4)

                #add height to the list
                z_list.append(ht)

            #df = pd.DataFrame(z_list, columns = ["columns"])
            #df.to_csv('thesis_run.csv')

```

```

def losCal(elevationsVector, observer_Height, objectHeight):

    lengthOfVector=len(elevationsVector)
    distances=np.array(list(range(0,lengthOfVector)))
    distances[0]=.01 # just to avoid a divide by 0 error -
this pixel will be visible anyway.

#gradients=(elevationsVector[1:lengthOfVector])/distances
    visibility = np.zeros(lengthOfVector)
    #surfgradients = np.zeros(lengthOfVector)
    observerGradients = np.zeros(lengthOfVector)

    visibility[0]=1
    visibility[1]=1
    for i in range(1,lengthOfVector):
        #surfgradients[i] = (elevationsVector[i]-
(elevationsVector[0] + observer_Height)) / distances[i]
        observerGradients[i] = ((elevationsVector[i] +
objectHeight)-(elevationsVector[0] + observer_Height)) / distances[i]

maxheightLine=observerGradients[i]*distances[0:i]+(elevationsVector[0]
+observer_Height)

        canSee=sum((elevationsVector[0:i]>maxheightLine)*1)

        if canSee==0:
            visibility[i]=1

        #if (i>1):
        #if (observerGradients[i] >
max(surfgradients[1:i])):
            #visibility[i] = 1

    return (visibility)

#CALL-IN THE TERRAIN WE FOUND EARLIER

if want_terrain_set==1:
    visibility = np.apply_along_axis(losCal, 1,z_list,
observer_Height=observerHeight, objectHeight=targetHeight )
    if want_plot==1:

        result=losCal(z_list,observerHeight,targetHeight)
        xpts=np.array(list(range(0, len(z_list))))
        canSeeTerrain=np.ma.masked_where(result==0,z_list)
        noSeeTerrain=np.ma.masked_where(result==1,z_list)
        observer_plt=z_list[0] + observerHeight

        # fig, ax = plt.subplots()
        # ax.plot(xpts,z_list)
        # ax.plot(xpts, canSeeTerrain, color='blue', label='Can
See')

```

```

See')
    # ax.plot(xpts, noSeeTerrain, color='red', label='Can not
    # ax.plot(0, observer_plt, 'bo', label='Observer')
    # plt.title('Synthetic Terrain')
    # ax.legend()

    # plt.show()
    # z_list = z_list.astype({'column1':'float'})
    gc.collect()
# Start/End X (m), Start/End Y (m), Elevations (m), angle (Rad), Observer
Sighting (yes=1, no=0)
    array1=np.vstack((x_list, y_list,z_list, angle_array,
result, filename_array))
    array1_transpose=array1.transpose()
    z_list1.append(array1_transpose)
    z_list2.extend(z_list1) #The indention is critical, don't change

#Compile all, single map runs into one .csv file
flatlist = [z_list2]
flatlist = []

for sublist in z_list2:
    for item in sublist:
        flatlist.append(item)
    gc.collect()
gc.collect()
with
open("/media/gman/Grant/thesis_complete/scanned_map_csv/mountain_scan.c
sv","w") as output:
    writer=csv.writer(output)
    writer.writerows(flatlist)
flatlist=None
dataset=None
z_list=None
y_list=None
x_list=None
xpts=None
sublist=None
result=None
noSeeTerrain=None
item=None
angle_array=None
array1=None
array1_transpose=None
canSeeTerrain=None
writer=None
filename_array=[]
size1=[]
size2=[]
step_size=[]
# z_list1=[]
# z_list2=[]
file=None
filename=None
fpath=None

```

```
gc.collect()
gc.collect()
print("without GPU:", timer()-start)
```

APPENDIX D: COMBINE CSV FILES

This script (STEP_2_MAP_CONCATENATION.py) combined CSV files together. This was extremely helpful as sometimes the .csv data files were simply too large to be written as one file (without running out of RAM), thus smaller files were created, and combined together.

```
# CSV Processing: Combine CSV files together.
# Author: John Grant, jmgrantunr@gmail.com, john.m.grant@navy.mil
# Goal: Combine together all the processes maps into one large file.
#https://stackoverflow.com/questions/1810743/how-to-set-the-current-
working-directory/1810760ArithmeticError

import pandas as pd
import os
import glob

CHUNK_SIZE = 1000000
# #os.chdir("/media/gman/Grant/Thesis_Maps/test_runs_3/Plain_maps")
os.chdir("/media/gman/Grant/Thesis_Maps/test_runs_3/Coast_maps/coast
s_maps")
csv_file_list = ["coasts_run1.csv", "coasts_run2.csv",
"coasts_run3.csv", "coasts_run4.csv", "coasts_run5.csv",
"coasts_run6.csv"]
#output_file = "/media/gman/Grant/Thesis_Maps/processed_maps/"

for csv_file_name in csv_file_list:
    chunk_container = pd.read_csv(csv_file_name,
chunksize=CHUNK_SIZE, encoding = "latin1")
    for chunk in chunk_container:
        chunk.to_csv("coasts_combined.csv", mode='a', index=False,
header=False, encoding='utf-8-sig') #Sometimes 'latin1' works best
also
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: ALTITUDE FILTER

This code was written to delete rows that contained an extremely high (-3.402823E038 m) erroneous altitude value. The data was broken up into 1000 row chunks, scanned for bad values, and the whole chunk deleted if any value displayed the incorrect value. This was to prevent having partial data sets where some values were good, but the rest deleted.

```
# PROGRAM: Remove Bad Data Sets
# GOAL: Break the data into groups of 1000 rows, if any row has -
3.4*10^38,
# delete the entire group.
# Author: John Grant, see also LTC Brian Wade notes and coding.
# https://stackoverflow.com/questions/44729727/pandas-slice-large-
dataframe-in-chunks

# CALL-IN LIBRARIES
import pandas as pd
import numpy as np
import os
import logging
#logging.getLogger('tensorflow').setLevel(logging.ERROR)

os.chdir('/media/gman/Grant/Thesis_Maps/test_runs_3/Coast_maps/coast
s_maps') #Set file directory.
coasts_data='coasts_combined.csv'

coasts = pd.read_csv(coasts_data, header=None, encoding='latin1')

#Combine all data into a single frame

frames = [coasts] #[plains,mountains,coast]
df = pd.concat(frames)

#### Arrange data so that each LoS vector of 1000 pts is in a single
row #####

n = 1000 #chunk row size
list_df = [df[i:i+n] for i in range(0,df.shape[0],n)]

l=[]
for index, frame in enumerate(list_df):
    vals = frame.iloc[:,2].values
    # if any values in this part of the frame are wrong, store index
for deletion
    if np.any(vals < -100000):
```

```
        l.append(index)
for x in sorted(l, reverse=True):
    del list_df[x]

df1 = pd.concat(list_df)
df1.to_csv(r'/media/gman/Grant/Thesis_Maps/test_runs_3/Coast_maps/co
asts_maps/coasts_combined_cleaned.csv', index = False, header=False)
```

APPENDIX F: CSV ROW COUNT

This script was written to count the rows in a given .csv file. This was helpful in determining whether or not the altitude filter program (Appendix E) was deleting individual rows out of each data set (if the row count was not divisible by 1000, then there were map scans with only partial data sets). Keep in mind most .csv files had millions of rows, making manual checking impossible.

```
# Row Count: Check the number of rows in a CSV file
# Author: Author: John Grant, jmgrantunr@gmail.com,
john.m.grant@navy.mil
# Goal: How many rows does a given csv file have?
# IMPORT LIBRARY
import csv
import os
import pandas as pd
import numpy as np

# DIRECTORY

#directory =
"/media/gman/John/Aerospace_Classes/Thesis_Programs/mountains_total_
rev1.csv"
directory =
"/media/gman/Grant/Thesis_Maps/test_runs_3/Coast_maps/coasts_maps/co
asts_combined.csv"
# with open(directory) as f:
#     reader = csv.reader(f,delimiter = ",")
#     data = list(reader)
#     row_count = len(data)

f = open(directory)
numlines = len(f.readlines())

os.chdir("/media/gman/Grant/Thesis_Maps/test_runs_3/Coast_maps/coast
s_maps/")
data=pd.read_csv("coasts_combined.csv")
t=data.tail(3000000)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G: LOS PREDICTION

A direct LOS calculation was performed by STEP5A_LOS_PREDICTION.py. The results of this script are compared to the output of one of the NNs (plains, mountain, or coastline).

```
# LOS CALCULATION SCRIPT
# AUTHORS: LTC BRIAN WADE, MAJ JAMES JABLONSKI, CPT SEAN CLEMENT, JOHN GRANT
#
import pandas as pd
import numpy as np
import os
import logging
logging.getLogger('tensorflow').setLevel(logging.ERROR)
from timeit import default_timer as timer
from datetime import datetime
from warnings import simplefilter

# SET TIMER
startTime = datetime.now()
start = timer()

# SET WARNINGS
simplefilter(action='ignore', category=FutureWarning)

# SET DIRECTORY
#os.chdir('/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/plains_complete_maps/')
os.chdir('/media/gman/Grant/Thesis_Maps/test_run_5/coasts/')
#os.chdir('/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/')
#directory =
r'/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/plains_complete_maps/h5_files/'

# LOAD DATA MAP
plains_data='coastline_combined_cleaned.csv'
#plains_data = 'plains_hyperparameter_test.csv'
#plains_data = 'plains_hyperparameter_test.csv'
plains = pd.read_csv(plains_data, header=None, encoding='latin1')
frames = [plains]
df = pd.concat(frames)

# TERRAIN COORDINATES
#x_list = df.iloc[:,0] #array of x grid points
#y_list = df.iloc[:,1]
#z_list = df.iloc[:,2]
#z_list = df.iloc[1000:2000,2].tolist()
```

```

# DATA PROCESSING
n = 1000 #chunk row size
list_df = [z_list[i:i+n].tolist() for i in range(0,z_list.shape[0],n)]

# SUBSECTION DATA
k = 500 #subsecting list_df
shortened_sublists = [sublist[:k] for sublist in list_df]

# CREATE EMPTY LISTS
z_list1=[]
z_list2=[]

# VARIABLES
observerHeight=20
targetHeight=0

# LOS CALCULATION FUNCTION
for sub_list in shortened_sublists:
    def losCal(elevationsVector, observer_Height, objectHeight):

        lengthOfVector=len(elevationsVector)
        distances=np.array(list(range(0,lengthOfVector)))
        distances[0]=.01 # just to avoid a divide by 0 error - this
        pixel will be visible anyway.
        #gradients=(elevationsVector[1:lengthOfVector])/distances
        visibility = np.zeros(lengthOfVector)
        #surfgradients = np.zeros(lengthOfVector)
        observerGradients = np.zeros(lengthOfVector)

        visibility[0]=1
        visibility[1]=1
        for i in range(1,lengthOfVector):
            #surfgradients[i] = (elevationsVector[i]-
            (elevationsVector[0] + observer_Height)) / distances[i]
            observerGradients[i] = ((elevationsVector[i] +
            objectHeight)-(elevationsVector[0] + observer_Height)) / distances[i]

        maxheightLine=observerGradients[i]*distances[0:i]+(elevationsVector[
        0] +observer_Height)

        canSee=sum((elevationsVector[0:i]>maxheightLine)*1)

        if canSee==0:
            visibility[i]=1

        return (visibility)

    #if want_plot==1:
    result = losCal(sub_list,observerHeight,targetHeight)
    array1=np.vstack(result)
    array1_transpose=array1.transpose()
    z_list1.append(array1_transpose)

```

```
# z_list2.extend(z_list1)

# TIMER
print("The run time in seconds is:", timer()-start)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX H: HYPERPARAMETER JMP14 SCRIPT

There are two parts to this script, the input file and the hyperparameter test script STEP_5B_NN_HYPERPARAMETERS.py.

Two sets of hyperparameters were used. One with hidden layer (HL) values between 0 and 100, and the other from 100 to 500. In Table 8, the HL neurons ranged from 0 to 100, while Table 9 varied from 100 to 1000 neurons. For the Activation Functions (ACT), the numbers 0, 1, and 2 correspond to Relu, Elu and LeakReLU, respectively.

The script STEP_5B_NN_HYPERPARAMETERS.py uses the hyperparameters that generated the best accuracy, initially using smaller data sets due to time constraints.

Table 8. Hyperparameter test numbers.

HL1	HL2	HL3	HL4	ACT1	ACT2	ACT3	ACT4	Batch Size	Learning Rate
30	0	90	50	0	0	0	0	10000	0.999
40	60	60	60	2	1	0	0	1000	0.0011
1	0	80	60	1	0	1	2	100	0.9268
70	80	1	100	2	2	2	2	10000	0.0744
70	50	40	40	1	1	2	0	1	0.8624
60	10	30	80	2	1	0	2	10	0.1389
50	60	0	20	2	2	0	0	10	0.8073
90	0	30	10	1	0	2	1	10000	0.1941
70	60	30	10	2	1	2	0	10000	0.0367
1	20	0	70	2	1	0	1	10000	0.9642
20	30	50	70	0	2	0	1	1000	0.76
80	20	70	100	1	0	0	2	10	0.2413
60	30	10	60	1	1	0	1	1	0.1064
10	10	60	20	1	0	0	0	10	0.8948
60	60	80	30	0	2	1	0	10	0.7196
100	10	80	90	1	1	2	2	10	0.2816
30	10	30	90	1	0	0	0	100	0.1665
100	10	90	1	2	0	0	2	1	0.835
10	50	20	70	1	1	2	1	10000	0.6855
80	1	30	80	2	2	1	2	1000	0.3158
30	80	40	100	1	0	1	0	10000	0.7838

10	10	0	50	0	2	0	0	100	0.2176
90	100	70	10	0	1	0	2	1	0.0554
30	60	1	100	0	0	0	1	10000	0.9456
10	80	90	20	2	0	0	2	1	0.9826
80	100	10	100	0	2	2	0	1000	0.0185
70	80	70	50	0	2	1	2	1	0.6562
100	0	40	40	1	0	0	0	10	0.3451
50	30	80	0	0	1	1	0	1000	0.7399
70	50	20	70	2	1	0	1	100	0.2614
100	70	70	90	2	0	1	2	10000	0.1226
40	50	50	1	2	0	0	1	1	0.8787
80	0	0	50	1	0	0	0	1000	0.0902
10	90	100	100	0	2	1	2	100	0.9109
50	60	60	60	2	2	1	0	100	0.6313
90	30	0	90	0	1	0	0	10	0.3701
30	40	70	1	2	2	2	2	10000	0.2986
100	50	60	70	1	1	2	2	1000	0.7026
100	50	70	10	2	0	0	1	100	0.8212
50	100	80	20	1	1	1	0	1	0.1803
100	80	90	1	1	1	2	2	1000	0.1525
70	100	30	100	0	2	0	0	1	0.61
100	80	10	30	0	0	1	1	10	0.3912
50	90	20	20	0	1	2	0	1000	0.8487
70	90	30	1	1	2	2	2	10000	0.3304
90	100	100	1	2	1	0	1	1	0.6711
70	1	1	0	1	0	2	0	100	0.2295
50	100	10	50	1	1	2	1	10	0.7719
20	70	0	0	0	1	0	0	100	0.7956
50	90	30	10	0	2	0	1	1000	0.5917
30	80	0	20	0	2	0	0	10	0.4093
30	20	10	90	0	1	1	1	1	0.2058
70	90	90	60	2	1	2	1	100	0.6438
1	1	70	0	0	0	2	1	10000	0.3575
60	70	90	60	0	2	1	2	100	0.7297
1	1	90	1	0	2	0	1	10000	0.2713
100	60	50	80	0	2	1	2	1000	0.0648
90	1	60	70	0	2	1	1	10000	0.9549
50	70	30	60	0	0	2	1	10	0.0277
50	10	30	1	1	1	2	2	10	0.9914

1	70	20	100	0	2	1	0	1	0.5764
50	60	90	70	1	2	2	2	10	0.4248
80	100	10	1	1	2	2	1	1	0.75
10	60	100	10	1	2	2	0	1000	0.2513
70	0	30	90	2	0	2	2	10	0.9359
100	40	90	30	1	1	1	2	10000	0.6207
1	20	30	0	1	0	1	0	100	0.0094
100	50	80	70	2	1	0	1	1	0.3807
30	20	40	1	0	2	0	0	100	0.0461
80	50	50	60	0	2	1	1	10	0.9734
10	70	0	50	2	1	0	2	1	0.3071
30	0	30	90	1	0	0	1	10	0.1145
100	100	10	50	2	0	2	2	10000	0.694
70	1	40	100	1	1	1	1	10000	0.8705
20	0	20	10	2	0	0	0	1000	0.9029
80	40	0	20	1	1	0	2	10000	0.563
40	30	40	60	1	2	1	1	10	0.4379
80	80	30	10	0	2	0	0	1000	0.131
30	1	100	70	2	1	0	0	10	0.29
100	70	90	90	2	0	0	0	10000	0.7111
70	20	90	50	2	2	0	1	1	0.6009
50	60	30	30	2	2	2	1	1000	0.0982
80	50	10	40	2	2	2	2	1000	0.4003
40	30	0	0	0	1	0	0	1	0.8867
100	40	1	60	2	1	0	1	100	0.0822
80	80	0	20	2	0	0	1	1	0.9189
90	100	70	70	1	1	2	1	1000	0.6637
10	70	1	0	2	0	1	0	10	0.3377
70	40	70	70	1	2	2	2	10000	0.1595
90	40	70	70	0	1	2	2	1000	0.8281
50	10	90	100	0	1	1	0	100	0.4491
80	90	20	30	2	1	0	2	10000	0.5518
100	30	40	90	2	1	0	2	100	0.1872
30	60	0	100	1	2	0	0	1000	0.323
1	10	1	50	1	1	1	0	10000	0.8141
60	20	0	90	1	0	1	1	10	0.6783
1	0	40	40	0	0	2	1	10	0.417
1	60	10	20	2	1	1	0	1	0.5841
40	40	100	20	1	2	1	2	1	0.8555

10	100	100	1	2	0	0	2	10	0.1734
90	20	0	1	0	1	0	1	1000	0.1456
30	70	20	70	0	2	1	0	1000	0.8419
40	100	80	40	0	1	0	2	10	0.6377
30	50	60	60	0	1	2	1	1000	0.3637
1	70	70	80	1	2	1	2	1000	0.4587
70	100	70	1	0	2	0	0	1	0.5422
10	10	60	90	0	2	2	1	1	0.2235
60	40	50	50	2	1	2	2	10000	0.7778
30	50	70	1	2	0	2	2	10	0.5698
70	100	50	40	1	0	1	1	10000	0.3513
80	80	10	50	2	2	0	2	1	0.6501
70	20	10	50	2	0	0	0	100	0.4313
50	50	60	100	0	2	2	2	100	0.7659
70	90	0	50	1	2	0	2	100	0.2355
100	50	60	40	0	2	0	1	1000	0.8014
70	10	30	60	2	0	0	0	1000	0.1999
20	80	20	40	1	0	1	2	1000	0.7895
30	70	30	40	2	2	2	1	1	0.2117
30	10	80	1	2	2	2	1	10	0.467
90	1	100	1	1	2	1	1	10	0.5341
70	50	0	80	2	1	0	2	1000	0.6154
1	40	60	0	2	2	1	0	1	0.3859
60	80	1	100	1	0	2	2	1000	0.2765
20	100	1	0	1	2	0	0	10000	0.7348
70	20	30	90	1	1	1	2	10000	0.2563
50	1	80	20	1	2	1	1	100	0.9407
100	50	50	50	0	2	1	2	10	0.5574
20	1	100	10	1	0	1	0	100	0.4435
50	0	40	50	1	0	0	0	100	0.06
60	1	90	0	1	2	0	0	1000	0.6261
1	20	100	30	0	0	0	0	10000	0.3754
40	30	100	60	0	2	2	0	1000	0.7549
1	0	20	0	0	0	0	0	10	0.7245
10	10	40	10	2	0	0	2	10	0.978
20	20	30	80	1	1	0	2	1	0.0231
30	30	30	20	0	2	0	1	10000	0.0415
60	1	0	60	1	1	0	2	1	0.9596
10	30	100	0	2	2	1	0	10000	0.527

50	1	50	30	0	1	0	2	100	0.474
1	1	100	0	2	1	0	0	100	0.2463
70	70	1	70	1	1	0	1	1	0.0696
80	30	60	10	2	1	0	2	1000	0.9956
20	50	60	60	0	0	0	2	10	0.7449
100	1	10	10	1	1	0	0	100	0.2664
90	60	80	70	2	0	0	0	1000	0.0138
60	60	1	0	2	0	1	2	1	0.5963
30	80	1	10	1	1	1	1	1000	0.9312
50	80	100	1	2	1	1	2	10	0.4048
80	0	0	80	0	0	0	2	1000	0.3114
1	0	20	50	1	0	2	1	100	0.9503
40	100	0	30	1	1	0	1	10	0.4539
10	0	60	1	0	0	0	1	100	0.5469
70	60	1	0	0	0	0	0	1	0.0507
1	10	30	20	1	1	0	0	1000	0.6983
80	50	30	70	2	1	1	0	1000	0.2944
90	70	30	90	2	2	1	1	10	0.0051
60	20	100	100	2	1	0	2	1	0.1269
100	90	0	60	2	0	0	1	1000	0.4801
70	90	40	50	0	0	2	2	1000	0.5209
40	1	40	10	0	2	1	1	1	0.9689
1	90	90	90	0	1	1	0	1000	0.6055
30	40	80	90	0	1	0	2	100	0.3957
40	90	30	0	2	0	1	0	100	0.8746
20	10	50	80	0	1	0	0	10000	0.1024
30	100	50	90	1	2	0	2	10	0.0322
40	30	40	90	0	2	2	0	1000	0.987
1	100	60	100	1	0	0	1	10000	0.8989
70	70	30	50	2	0	1	0	10000	0.707
80	50	90	30	2	0	2	1	1000	0.6896
50	60	50	90	2	0	2	2	10	0.0863
20	80	10	90	0	0	1	1	10000	0.9149
60	90	80	90	2	2	0	2	1000	0.2858
60	100	80	20	1	2	2	0	1	0.8664
70	40	30	0	0	2	1	0	10	0.1105
30	40	100	30	1	2	1	0	1	0.5156
90	80	80	10	0	2	1	0	10000	0.4856
40	100	90	60	1	0	2	0	10	0.7153

20	50	20	1	1	0	2	1	1	0.8907
10	60	0	30	2	1	0	2	100	0.3029
1	10	50	100	2	2	0	2	100	0.4628
100	1	1	20	0	0	0	2	100	0.135
90	30	1	80	2	2	1	1	10000	0.5804
60	100	60	60	0	2	1	0	1	0.5381
20	30	100	90	0	0	1	0	10000	0.4209
40	0	80	100	0	0	1	0	1000	0.6674
50	60	0	10	0	0	0	0	1	0.334
10	60	0	1	2	2	0	0	100	0.1185
1	1	30	50	0	1	2	1	10000	0.5108
80	30	0	50	1	0	0	2	100	0.4904
90	30	40	40	0	1	2	1	1000	0.9068
1	70	10	50	0	1	1	0	1000	0.0782
20	50	20	80	1	1	2	0	1	0.8827
40	0	50	50	0	0	1	0	10	0.9229
50	30	30	50	2	2	0	2	1	0.0942
70	30	10	80	1	2	0	1	100	0.4131
100	90	80	10	1	2	2	2	1	0.8176
50	1	1	10	1	0	0	2	10000	0.5066
100	70	40	60	0	0	0	0	10000	0.4949
10	50	70	1	0	2	2	2	100	0.1561
10	40	70	1	1	1	2	0	1000	0.5007

Table 9. Hyperparameter test values.

HL1	HL2	HL3	HL4	Act1	Act2	Act3	Act4	Batch Size	Learning Rate
250	250	500	100	2	1	0	0	1000	0.009999689
250	250	250	1000	0	2	0	0	10000	0.001001815
1000	100	250	100	1	2	2	1	10000	0.009339842
250	250	250	250	2	2	2	1	10000	0.001662489
250	100	100	250	1	2	0	0	1000	0.008759617
500	250	250	100	2	1	1	1	10000	0.002242846
250	250	1000	500	0	2	2	1	10000	0.008263246
1000	1000	1000	1000	1	0	0	0	10000	0.002739112
100	250	500	500	0	1	1	2	10000	0.001323616
100	100	250	500	0	1	0	0	1000	0.009678498

250	1000	1000	1000	0	1	2	0	10000	0.007837506
250	1000	250	100	1	1	1	0	10000	0.003165474
1000	500	100	1000	2	0	1	1	10000	0.009050262
250	500	250	250	0	1	1	1	10000	0.001951465
500	1000	1000	1000	2	0	2	2	1000	0.007475022
100	250	500	100	2	2	1	1	10000	0.003528403
500	500	1000	100	0	0	2	2	1000	0.008512399
100	100	1000	100	2	0	1	0	1000	0.002489814
250	1000	500	250	2	1	0	2	1000	0.007165518
250	250	500	250	1	1	2	1	10000	0.00383733
100	250	500	500	1	2	0	2	1000	0.002951355
100	500	1000	250	1	0	2	0	1000	0.008051071
500	500	500	250	2	2	2	1	1000	0.001491992
1000	500	100	100	2	1	1	1	10000	0.009510702
100	500	500	1000	2	1	2	1	1000	0.001158282
250	250	100	250	1	2	2	1	10000	0.009843617
500	100	1000	1000	2	0	2	0	1000	0.006902218
250	100	100	100	0	2	2	0	10000	0.004100504
500	250	1000	500	1	2	0	2	1000	0.003346726
1000	1000	250	500	1	0	1	1	10000	0.007656601
100	500	500	100	1	1	1	1	10000	0.002096239
100	100	250	250	1	0	0	1	10000	0.0089053
1000	250	1000	100	0	2	1	1	10000	0.009195387
100	250	500	500	2	2	2	1	1000	0.001805497
100	1000	250	500	2	1	2	2	10000	0.00667729
250	250	100	100	2	1	2	1	10000	0.004324199
500	100	250	1000	0	2	1	2	10000	0.007320927
1000	1000	1000	500	0	1	0	1	10000	0.003682485
250	1000	1000	100	1	2	2	2	1000	0.002614191
100	500	500	1000	2	0	1	2	10000	0.008389187
250	1000	100	100	0	0	1	0	10000	0.006486592
250	100	100	500	0	2	2	2	10000	0.00451491
500	250	1000	250	0	0	1	2	1000	0.008637059
250	100	1000	500	2	1	2	2	1000	0.002365542
1000	1000	100	100	1	0	0	0	10000	0.007033882
250	250	1000	1000	2	1	0	0	10000	0.003969241
1000	500	1000	1000	0	0	1	1	1000	0.003058285
500	500	100	250	2	2	0	1	10000	0.007945019
250	250	100	1000	2	1	1	2	10000	0.006324396

500	250	1000	500	2	0	1	1	10000	0.004676794
250	250	250	250	2	2	1	1	10000	0.002844591
1000	250	100	1000	1	2	0	0	10000	0.008158144
250	1000	100	1000	0	0	0	1	10000	0.006790099
1000	250	1000	250	2	0	2	2	1000	0.004212329
250	1000	500	100	2	1	0	1	10000	0.003437414
1000	100	250	250	2	2	0	1	10000	0.007565238
500	250	1000	100	2	1	2	0	1000	0.001407905
1000	500	250	500	2	0	1	2	1000	0.009425642
100	100	100	100	2	0	1	0	1000	0.009761243
1000	1000	250	100	0	1	2	2	1000	0.001079983
100	100	500	100	1	0	0	0	10000	0.006185694
250	250	500	100	2	1	0	1	1000	0.004815544
100	1000	250	500	1	0	2	0	10000	0.00325456
250	250	250	100	2	1	0	0	1000	0.007745688
100	1000	500	1000	2	1	1	0	10000	0.001579531
500	500	250	100	1	0	2	2	10000	0.006581405
250	100	250	250	2	0	0	0	1000	0.004420097
250	1000	100	1000	2	1	2	1	10000	0.00992581
100	1000	500	1000	1	0	1	2	1000	0.009594948
1000	1000	1000	100	0	1	1	1	10000	0.001241772
250	1000	1000	100	0	2	1	0	10000	0.007243719
500	500	1000	250	1	2	0	2	1000	0.003758336
250	250	1000	500	2	1	2	0	1000	0.002169555
250	500	250	250	0	2	1	1	1000	0.008978013
1000	500	250	500	1	2	0	2	1000	0.001878203
100	250	250	100	1	1	0	1	1000	0.006066801
100	1000	1000	500	2	0	2	0	1000	0.004934962
500	100	500	500	1	2	1	2	1000	0.009265852
1000	500	500	250	0	1	2	1	10000	0.007397781
500	1000	250	100	0	1	2	1	10000	0.003605012
100	1000	250	1000	2	1	1	0	10000	0.008831532
1000	500	500	500	1	2	1	2	1000	0.00640527
100	1000	250	1000	2	2	2	2	1000	0.004595869
1000	250	100	250	1	0	1	0	10000	0.00202395
100	1000	1000	250	0	2	0	0	10000	0.00173325
500	100	100	500	2	2	0	1	1000	0.009122685
500	1000	100	250	1	2	2	0	1000	0.006967433
500	1000	100	100	2	2	0	0	1000	0.004035597

500	250	1000	250	2	1	2	0	10000	0.008325936
1000	100	100	1000	1	0	2	2	1000	0.002552444
500	500	1000	100	2	0	1	0	10000	0.005965074
500	1000	100	100	1	0	1	0	1000	0.00503744
1000	500	1000	1000	0	0	1	0	10000	0.008574376
250	1000	500	500	1	0	2	1	1000	0.003904205
1000	1000	1000	250	0	1	0	2	1000	0.007100339
250	1000	250	500	0	2	1	2	1000	0.002677676
100	250	500	250	1	0	2	2	10000	0.006255122
500	1000	500	100	0	1	1	1	1000	0.004746367
250	500	500	100	2	0	0	2	10000	0.002304901
250	500	1000	250	0	0	1	2	1000	0.008699069
500	1000	100	100	2	2	1	2	1000	0.002427591
500	100	100	250	2	2	1	1	10000	0.008447813
500	250	1000	250	0	2	0	0	1000	0.006733514
1000	500	1000	100	0	1	0	2	10000	0.005878875
500	1000	250	250	2	1	1	1	10000	0.005123454
1000	1000	500	100	0	1	1	1	10000	0.004268209
500	250	1000	500	2	2	0	2	1000	0.003005371
500	500	250	500	0	1	0	1	1000	0.007891896
250	100	1000	250	0	2	1	2	10000	0.008104967
500	100	100	250	2	2	0	0	1000	0.004875054
500	100	100	1000	0	0	2	1	1000	0.006128283
1000	250	250	1000	1	0	0	2	1000	0.003113636
250	1000	500	1000	2	1	1	0	10000	0.006846529
250	250	500	250	0	1	2	0	10000	0.004157149
100	250	1000	1000	0	2	0	1	10000	0.002793292
250	500	500	500	1	1	2	0	10000	0.007998892
250	100	500	250	0	2	0	2	10000	0.008211195
1000	1000	250	250	1	2	1	0	10000	0.002900052
1000	500	100	250	1	1	0	0	10000	0.005804656
250	500	250	1000	0	2	1	2	10000	0.005197398
1000	250	250	100	0	1	0	1	10000	0.006533827
250	500	100	100	1	0	0	0	1000	0.004467853
500	250	100	500	0	1	0	1	10000	0.003300761
500	100	1000	500	0	2	0	2	10000	0.007609759
250	250	500	250	1	1	0	1	10000	0.003482666
250	250	100	500	2	1	1	1	1000	0.001535222
500	500	250	1000	2	1	2	0	1000	0.004985023

1000	100	100	500	1	2	2	1	1000	0.006016268
1000	1000	100	500	0	1	1	0	1000	0.007790867
500	500	500	500	2	2	2	1	1000	0.006629296
250	500	1000	100	0	1	2	2	10000	0.004371096
1000	250	500	250	2	2	1	1	10000	0.009468011
250	250	250	100	2	1	0	0	10000	0.001199563
1000	500	100	1000	0	1	1	1	10000	0.009884681
100	250	1000	500	2	2	1	0	10000	0.007518548
100	1000	100	500	1	1	1	1	1000	0.001365597
250	500	1000	500	2	0	2	1	10000	0.009636949
1000	500	100	100	1	1	0	2	1000	0.005741299
100	250	250	500	1	2	1	2	10000	0.00526213
250	500	100	500	0	0	1	2	10000	0.003391866
500	100	100	1000	2	2	1	2	1000	0.001040664
250	1000	250	250	1	0	2	1	1000	0.009381444
100	1000	1000	250	0	2	0	0	10000	0.003207101
100	1000	100	250	0	0	0	2	10000	0.00770172
100	1000	1000	100	2	1	2	0	1000	0.001622126
250	100	500	1000	0	0	1	1	10000	0.009800882
500	500	1000	500	0	2	2	1	10000	0.004635872
250	100	100	1000	1	1	1	1	1000	0.006364887
100	1000	100	250	1	0	1	0	10000	0.003798096
250	100	250	1000	1	1	1	2	1000	0.007205245
1000	1000	500	1000	0	0	2	0	1000	0.001451127
500	100	100	250	0	0	2	0	1000	0.009551979
100	500	100	1000	0	2	2	2	1000	0.005922121
1000	1000	250	100	1	0	2	2	1000	0.005080339
250	250	100	500	2	0	0	1	10000	0.00364389
250	500	250	500	0	1	0	2	1000	0.007358498
250	500	500	100	2	0	2	1	10000	0.005685281
500	500	500	100	1	0	0	0	10000	0.001282465
500	100	250	500	2	2	0	1	10000	0.005317225
250	500	1000	1000	0	1	1	2	10000	0.009719731
100	500	1000	100	2	0	2	0	10000	0.00886797
250	1000	100	100	2	0	0	0	10000	0.009963944
250	250	500	100	0	1	0	0	1000	0.006446152
1000	250	1000	1000	2	0	2	2	10000	0.002133097
1000	250	100	100	0	2	1	2	1000	0.004555932
1000	100	100	250	0	0	0	0	10000	0.001915714

500	250	500	100	2	2	1	2	10000	0.001118366
500	500	1000	1000	0	1	2	1	1000	0.009158841
500	1000	100	250	1	1	1	2	1000	0.00901366
250	250	100	250	1	1	0	2	1000	0.001769144
100	100	250	500	1	0	0	0	1000	0.007436945
1000	1000	100	1000	2	1	2	2	10000	0.003567066
500	500	250	1000	1	1	2	1	1000	0.009303439
100	500	500	100	2	2	1	0	1000	0.005637746
250	100	250	500	1	2	1	1	10000	0.005365356
1000	1000	250	500	1	2	2	2	1000	0.002207848
1000	250	100	250	0	0	0	1	10000	0.007281948
100	500	500	250	2	0	1	0	1000	0.003721089
100	500	250	250	1	0	0	0	10000	0.001988253
100	250	500	250	0	2	1	2	1000	0.005840521
1000	1000	100	500	0	0	0	0	10000	0.005159845
100	500	500	250	2	1	1	2	1000	0.008794602
500	250	500	250	0	2	2	1	1000	0.006220825
100	100	100	100	1	2	0	1	10000	0.00478066
250	250	250	1000	2	1	0	0	1000	0.006999766
250	250	100	100	2	1	0	2	10000	0.004002411
250	100	250	500	0	2	0	0	10000	0.008941808
1000	250	100	100	2	2	2	1	10000	0.005594565
500	1000	500	100	1	1	1	0	10000	0.005409052
500	100	500	1000	1	2	0	0	10000	0.001841856
1000	100	100	100	1	1	0	0	10000	0.009086827
100	100	1000	250	2	2	0	0	10000	0.002060642
250	100	1000	500	1	0	2	1	1000	0.008480125
250	1000	100	1000	1	1	0	2	10000	0.001697474
250	500	500	100	1	2	1	2	1000	0.004711979
500	1000	500	250	0	2	0	1	10000	0.005556852
500	100	1000	100	1	1	0	1	10000	0.005446795
1000	1000	250	500	2	2	0	1	1000	0.00264605
100	250	500	100	1	1	0	0	10000	0.009231533
500	1000	500	500	2	1	1	1	1000	0.005501565

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX I: HYPERPARAMETER SCRIPT

The hyperparameter selection was done with both an input file (see Appendix H) and the following script file (STEP_5B_NN_HYPERPARAMETER.py):

```
#PROJECT: GRANT THESIS
#PROJECT GOAL: FEED-FORWARD NEURAL NETWORK (FFNN)

#AUTHOR: LTC BRIAN WADE, JOHN GRANT, brian.wade@nps.edu,
john.m.grant@navy.mil, jmgrantunr@gmail.com
#Citations:
# https://stackoverflow.com/questions/26414913/normalize-columns-of-
pandas-data-frame
# https://datascience.stackexchange.com/questions/15135/train-test-
validation-set-splitting-in-sklearn
# https://keras.io/optimizers/
# https://machinelearningmastery.com/dropout-regularization-deep-
learning-models-keras/
# https://keras.io/models/model/
# https://towardsdatascience.com/sigmoid-activation-and-binary-
crossentropy-a-less-than-perfect-match-b801e130e31
# CALL-IN LIBRARIES
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import sklearn.model_selection as model_selection
from keras.layers import Dense, Input, LocallyConnected1D, Dropout,
BatchNormalization, Reshape, Flatten, LeakyReLU
from keras.models import Model, Sequential
from keras import optimizers
#from numba import jit, cuda
#from tensorflow.python.client import device_lib
#from keras import backend as K
#from sklearn.preprocessing import OneHotEncoder
import keras
import os
import tensorflow as tf
from keras.layers import MaxPooling2D

# VARIABLES
training_size=.8 #the training, testing and validation sizes must equal
1 (Example: .8+.1+.1 =1.0)
testing_size=.2
validation_size=.5 #The validation set is a fraction of the test set.
np.random.seed(5)
epoch=500; #How many times to iterate over the training data.
#losses='sparse_categorical_crossentropy'
losses = 'binary_crossentropy'

metrics=['acc']
#metrics=['binary_accuracy']
```

```

dropout1=0.2;
alpha1=0.3;

# CALL-IN JMP-14 HYPERPARAMETER INPUTS
#directory_DOE=pd.read_csv('/media/gman/Grant/Thesis_Maps/processed_maps/JMP_14_DOE_cleaned_rev2.csv',header=None, encoding='UTF-8')
directory_DOE =
pd.read_csv('/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/map_runs/FFFD_test4.csv', header=None, skiprows=[-1], encoding='utf-8')
#directory_DOE =
pd.read_csv('/media/gman/Grant/Thesis_Maps/processed_maps/testing/JMP_14_DOE_cleaned_rev5.csv', header=None, skiprows=[-1], encoding='utf-8')
# FFNN FOR LOOP
# Purpose: loop through the FFNN, using JMP-14 file as DOE input.
results = np.zeros((len(directory_DOE), 3) ) #note the double (( ))

# CALL-IN TERRAIN DATA FILE
os.chdir('/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/') #Set file directory.

#directory='/media/gman/Grant/Thesis_Maps/processed_maps/hyperparamater_training_corrected.csv'
directory='plains_hyperparameter_test_1.csv'

df2=directory_DOE;

# CREATE USABLE DATA FILE FROM CSV FILENAME IMPORT

readCSV = pd.read_csv(directory, header=None) #make the input data a file in memory.
df=readCSV.iloc[:,[0,1,2,3,4]] #Grab the first 5 rows of data, skip the last row, its not needed here.
nrows = len(readCSV) #Number of rows in the input file
nsets = int(nrows/1000) #Divide the number of input file rows by 1000.
input_set = np.empty((nsets, 1000))
output_set = np.zeros((nsets, 1000))

ind = 0

for k in range(nsets):
    end_ind = ind+1000
    input_chunk = readCSV.iloc[ind:end_ind,2] #Use only the z-axis (altitude) data.
    output_chunk = readCSV.iloc[ind:end_ind,4]
    input_set[k,:] = input_chunk #use input_set for data, it is the correct shape.
    output_set[k,:] = output_chunk #use output_set for output data.
    ind += 1000

# CREATE USABLE DATA FILE FROM CSV FILENAME IMPORT
# readCSV = pd.read_csv(directory, header=None)
# df=readCSV.iloc[:,[0,1,2,3,4]]

# SCALE DATA FROM 0 TO 1
#x1 = input_set.values #returns a numpy array

```

```

min_max_scaler = preprocessing.MinMaxScaler()
data_scaled = min_max_scaler.fit_transform(input_set)
df1 = pd.DataFrame(data_scaled) #Fully-scaled data, range: 0 to 1
# Note, the output_set has already essentially been scaled, so only scale
the input_set.

#data_scaled1 = min_max_scaler.fit_transform(output_set)
df3 = pd.DataFrame(output_set)

# # SCALE DATA FROM 0 TO 1
# x1 = df.values #returns a numpy array
# min_max_scaler = preprocessing.MinMaxScaler()
# data_scaled = min_max_scaler.fit_transform(x1)
# df1 = pd.DataFrame(data_scaled) #Fully-scaled data, range: 0 to 1

# # CREATE TRAINING, TESTING AND VALIDATION SETS
# X=df1.iloc[:, [0,1,2,3]]
# y=df1.iloc[:, [4]]

# CREATE TRAINING, TESTING AND VALIDATION SETS
X=df1
y=df3

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
train_size=training_size, test_size=testing_size, random_state=100)
X_test, X_val, y_test, y_val = model_selection.train_test_split(X_test,
y_test, test_size=validation_size, random_state=1)

# Note: The data is ready, now to setup the FFNN such that hidden layers
and activation functions turn on and off, depending on the input variable
table.

for i in range(len(directory_DOE)): #iterate over JMP-14 file
#for i in directory_DOE:

    # FFNN AND VARIABLE "SWITCHBOARD"

    model = Sequential()
    input_dim = X_train.shape[1]
    # Input Layer
    #optimizer1='adam' # Options: sgd, RMSprop, adagrad, adadelata, adam,
adamax, nadam,
# optimizer1 = keras.optimizers.Adam(learning_rate=df2.iloc[i,9],
beta_1=0.9, beta_2=0.999, amsgrad=False)
# optimizer1 = keras.optimizers.Adamax(learning_rate=df2.iloc[i,9],
# optimizer1 = keras.optimizers.RMSprop(learning_rate=df2.iloc[i,9],
rho=0.9)
# optimizer1 = keras.optimizers.Adagrad(learning_rate=df2.iloc[i,9])
# optimizer1 = keras.optimizers.Adadelata(learning_rate=df2.iloc[i,9],
rho=0.95)
optimizer1 = keras.optimizers.Adamax(learning_rate=df2.iloc[i,9],
beta_1=0.9, beta_2=0.999)

```

```

# optimizer1 = keras.optimizers.Nadam(learning_rate=df2.iloc[i,9],
beta_1=0.9, beta_2=0.999)
batch1=df2.iloc[i,8];

# INPUT LAYER
model.add(Dense(1000,input_dim=input_dim,activation='relu'))

model.add(Dropout(0.2))
# 1st Hidden Layer
if df2.iloc[i,0] > 0: #In reality, this will always be true since
it'll never be zero in
    if df2.iloc[i,4] == 0: # 0 correlates to relu
        model.add(Dense(df2.iloc[i,0],input_dim=input_dim,
activation='relu'))
#        model.add(Dropout(dropout1))
    else:
        pass
    if df2.iloc[i,4] == 1: # 1 correlates to elu
        model.add(Dense(df2.iloc[i,0], activation='elu'))
        model.add(Dropout(dropout1))
#        model.add(BatchNormalization(input_shape=(1000,)))
#        model.add(MaxPooling2D(4,))
    else:
        pass
    if df2.iloc[i,4] == 2: # 2 correlates to leakyrelu
        model.add(Dense(df2.iloc[i,0]))
        model.add(LeakyReLU(alpha=alpha1))
        model.add(Dropout(dropout1))
#        model.add(BatchNormalization(input_shape=(1000,)))
#        model.add(MaxPooling2D(4,))
    else:
        pass
else:
    pass

# 2nd Hidden Layer
if df2.iloc[i,1] > 0: #If equal to 0, this layer drops out.
    if df2.iloc[i,5] == 0: # 0 correlates to relu
        model.add(Dense(df2.iloc[i,1],input_dim=input_dim,
activation='relu')) #, input_dim=input_dim
        model.add(Dropout(dropout1))
#        model.add(BatchNormalization(input_shape=(1000,)))
#        model.add(MaxPooling2D(4,))
    else:
        pass
    if df2.iloc[i,5] == 1: # 1 correlates to elu
        model.add(Dense(df2.iloc[i,1], activation='elu'))
        model.add(Dropout(dropout1))
#        model.add(BatchNormalization(input_shape=(1000,)))
#        model.add(MaxPooling2D(4,))
    else:
        pass
    if df2.iloc[i,5] == 2: # 2 correlates to leakyrelu
        model.add(Dense(df2.iloc[i,1]))
        model.add(LeakyReLU(alpha=alpha1))

```

```

        model.add(Dropout(dropout1))
#        model.add(BatchNormalization(input_shape=(1000,)))
#        model.add(MaxPooling2D(4,))
    else:
        pass
else:
    pass

    # 3rd Hidden Layer
    if df2.iloc[i,2] > 0: #In reality, this will always be true since
it'll never be zero in
        if df2.iloc[i,6] == 0: # 0 correlates to relu
            model.add(Dense(df2.iloc[i,2],input_dim=input_dim,
activation='relu')) #, input_dim=input_dim
            model.add(Dropout(dropout1))
#            model.add(BatchNormalization(input_shape=(1000,)))
#            model.add(MaxPooling2D(4,))
        else:
            pass
        if df2.iloc[i,6] == 1: # 1 correlates to elu
            model.add(Dense(df2.iloc[i,2], activation='elu'))
            model.add(Dropout(dropout1))
#            model.add(BatchNormalization(input_shape=(1000,)))
#            model.add(MaxPooling2D(4,))
        else:
            pass
        if df2.iloc[i,6] == 2: # 2 correlates to leakyrelu
            model.add(Dense(df2.iloc[i,2]))
            model.add(LeakyReLU(alpha=alpha1))
            model.add(Dropout(dropout1))
#            model.add(BatchNormalization(input_shape=(1000,)))
#            model.add(MaxPooling2D(4,))
        else:
            pass
    else:
        pass

    # 4rth Hidden Layer
    if df2.iloc[i,3] > 0: #If equal to 0, this layer drops out.
        if df2.iloc[i,7] == 0: # 0 correlates to relu
            model.add(Dense(df2.iloc[i,3], input_dim=input_dim,
activation='relu')) #, input_dim=input_dim
            model.add(Dropout(dropout1))
#            model.add(BatchNormalization(input_shape=(1000,)))
#            model.add(MaxPooling2D(4,))
        else:
            pass
        if df2.iloc[i,7] == 1: # 1 correlates to elu
            model.add(Dense(df2.iloc[i,3], activation='elu'))
            model.add(Dropout(dropout1))
#            model.add(BatchNormalization(input_shape=(1000,)))
#            model.add(MaxPooling2D(4,))
        else:
            pass
        if df2.iloc[i,7] == 2: # 2 correlates to leakyrelu

```

```

        model.add(Dense(df2.iloc[i,3]))
        model.add(LeakyReLU(alpha=alpha1))
        model.add(Dropout(dropout1))
#         model.add(BatchNormalization(input_shape=(1000,)))
#         model.add(MaxPooling2D(4,))
    else:
        pass
    else:
        pass
# Output Layer
model.add(Dense(1000,activation='sigmoid'))

# COMPILE MODELING METHOD
model.compile(optimizer=optimizer1, loss = losses, metrics = metrics)

# Fit Training
#         model.fit(X_train.to_numpy(),          y_train.to_numpy(),
batch_size=batch1, epochs=epoch)
# model.fit(X_test.to_numpy(), y_test.to_numpy(), batch_size=batch1,
epochs=epoch)
# model.fit(X_val.to_numpy(), y_val.to_numpy(), batch_size=batch1,
epochs=epoch)
# model.summary()
es=tf.keras.callbacks.EarlyStopping(monitor='val_loss', mode='min',
verbose=1)
model.fit(X_train,          y_train,          batch_size=batch1,
epochs=epoch,validation_data=(X_val, y_val), verbose=0, callbacks=[es])

#model.fit(X_val, y_val, batch_size=batch1, epochs=epoch)

#pred_train = model.predict(X_train)
#pred_val = model.predict(X_val)
#pred_test = model.predict(X_test)

#model.summary()

# Accuracy Testing
# scores1 = model.evaluate(X_train.to_numpy(), y_train.to_numpy())
# scores2 = model.evaluate(X_test.to_numpy(), y_test.to_numpy())
# scores3 = model.evaluate(X_val.to_numpy(), y_val.to_numpy())

scores1 = model.evaluate(X_train, y_train,verbose=0)
scores2 = model.evaluate(X_test, y_test,verbose=0)
scores3 = model.evaluate(X_val, y_val,verbose=0)

# with open('results.csv', mode='w') as data_file:

#     data_write=csv.writer(data_file,delimiter=',',)

results[i,0] = scores1[1]
results[i,1] = scores2[1]
results[i,2] = scores3[1]

print("Training Accuracy: %.2f%%\n" % (scores1[1]*100))
print("Testing Accuracy: %.2f%%\n" % (scores2[1]*100))

```

```
print("Validation Accuracy: %.2f%%\n" % (scores3[1]*100))

# SAVE DATA
#when complete with everything, write to csv:
np.savetxt('plain_4.csv', results, delimiter=',')
#    np.save('results_plains_JMP1.csv', results)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX J: NN TRAINING DATA SCRIPT

Using the JMP14 scripts, a set of suitable hyperparameters were selected. Below is the resultant script to train the FFNNs, STEP_6B_NN.py.

```
# FEED-FORWARD NEURAL NETWORK (FFNN) for Line-of-sight calculations
in different terrain.
#AUTHOR: BRIAN WADE, JOHN GRANT

# CALL-IN LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv

from sklearn import preprocessing
import sklearn.model_selection as model_selection
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
from keras.layers import Dense, Input, LocallyConnected1D, Dropout,
BatchNormalization, Reshape, Flatten, LeakyReLU
from keras.models import Model, Sequential, load_model
from keras.constraints import maxnorm
from keras import optimizers
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.utils.vis_utils import plot_model

import keras
import os
import tensorflow as tf

import joblib
import pickle
# !!! Saving models requires the h5py lib as well. !!!

import logging
logging.getLogger('tensorflow').setLevel(logging.ERROR)

# import warnings filter and ignore all future warnings
from warnings import simplefilter
simplefilter(action='ignore', category=FutureWarning)

from datetime import datetime
startTime = datetime.now()

#####
# Data Setup variables
training_size = .8 #the training, testing and validation sizes must
equal 1 (Example: .8+.1+.1 =1.0)
```

```

validation_size = .5 #The validation set is a fraction of the test
set.
np.random.seed(5)

# Training Hyperparameters
epoch = 200 #How many times to iterate over the training data.
batch_size = 1000
learning_rate = 0.004 # .005 for plains and coastline

#loss = 'sparse_categorical_crossentropy'

loss = 'binary_crossentropy'
metrics=['accuracy']
weights = {0:100, 1:1}

# Architecture Hyperparameters:
HL1 = 250      # plains 500, coastline 500
HL2 = 100      # plains 250, coastline 250
HL3 = 100      # plains 250, coastline 250
HL4 = 100      # plains 100, coastline 100
dropout1=0.2
alpha1=0.3

#####
#### Read and combine data #####
#Data files
current_dir                                     =
"/media/gman/Grant/thesis_complete/NN_Files/mountains_NN_files"
data_folder                                     =
'/media/gman/Grant/thesis_complete/NN_Files/mountains_NN_files'
mountain_data='mountains_total_cleaned.csv'
#mountains_data='mountains.csv'
#coast_data='coast.csv'

#Go to data folder and read in data into individual dataframes
#plains      = pd.read_csv(os.path.join(current_dir,    data_folder,
plain_data), header=None)
#mountains   = pd.read_csv(os.path.join(current_dir,    data_folder,
mountains_data), header=None)
mountains    = pd.read_csv(os.path.join(current_dir,    data_folder,
mountain_data), header=None)

# combine data into a single dataframe
terrain_set = ['mountains']#, 'mountains', 'coast']

for terrain in terrain_set:

    time_delta = datetime.now() - startTime
    delta_hour = time_delta.seconds//3600
    delta_min = ((time_delta.seconds - (delta_hour * 3600))/60)
    delta_sec = (time_delta.seconds - delta_hour*3600 - delta_min *
60)%60

```

```

print('#####')
print(f'Time elapsed since start of analysis: {delta_hour} hours,
{delta_min} minutes, {delta_sec} seconds')
print('starting terrain type: ' + terrain)
print('#####')
print(' ')

# load current terrain set into a dataframe
df = eval(terrain)

#### Arrange data so that each LoS vector of 1000 pts is in a
single row ####
nrows = len(df) #Number of rows in the input file
nsets = int(nrows/1000) #Divide the number of input file rows by
1000.
input_set = np.empty((nsets, 1000))
output_set = np.zeros((nsets, 1000))

# Separate data into 1000 meter vectors
ind = 0
for k in range(nsets):
    end_ind = ind+1000
    input_chunk = df.iloc[ind:end_ind,2] #Use only the z-axis
(altitude) data.
    output_chunk = df.iloc[ind:end_ind,4] #visibility - can see
(0 or 1)
    input_set[k,:] = input_chunk #use input_set for data, it is
the correct shape.
    output_set[k,:] = output_chunk #use output_set for output
data.
    ind += 1000

#### SCALE DATA FROM 0 TO 1 #####
min_max_scaler = preprocessing.MinMaxScaler()
data_scaled = min_max_scaler.fit_transform(input_set)
X = pd.DataFrame(data_scaled) #Fully-scaled data, range: 0 to 1

# Save scaler info for later deployment
scaler_filename = os.path.join(current_dir, 'models',
'FFNN_LOS_scaler.save')
joblib.dump(min_max_scaler, scaler_filename)

# Note, the output_set has already essentially been scaled, so
only scale the input_set.
y = pd.DataFrame(output_set)

# Prevent Keras from thinking onehot encoded
(https://stackoverflow.com/questions/48254832/keras-class-weight-in-
multi-label-binary-classification)

first_entry_zero = np.where(y.iloc[:,0] == 0)
for entry in first_entry_zero:
    y.iloc[entry,0] =1

```

```

##### Split data for training #####
X_train, X_test, y_train, y_test =
model_selection.train_test_split(X, y, train_size=training_size,
random_state=100, shuffle = True)
X_test, X_val, y_test, y_val =
model_selection.train_test_split(X_test, y_test,
test_size=validation_size, random_state=1)

# initialize
end_pt_set = list(range(100,1100,100))
score_set = np.zeros((len(end_pt_set), 7)) # This will hold the
neural net fit R^2 scores [end_pt, train, val, test]
index = 0
auc_val_best = 0.5

# Loop through data with increasing end points
for end_pt in end_pt_set:

    print('#####')
    print('starting end point: ' + str(end_pt) + ' for terrain
type: ' + terrain)
    print('#####')

    X_train_now = X_train.iloc[:,0:end_pt]
    y_train_now = y_train.iloc[:,0:end_pt]

    X_val_now = X_val.iloc[:,0:end_pt]
    y_val_now = y_val.iloc[:,0:end_pt]

    X_test_now = X_test.iloc[:,0:end_pt]
    y_test_now = y_test.iloc[:,0:end_pt]

    ## NEURAL NETWORK TRAINING ##
    ##### Prepare the network for training #####
    model = Sequential()
    input_dim = X_train_now.shape[1]

    # Input and hidden layers
    model.add(Dense(1000, input_dim=input_dim, activation='relu'))
    #model.add(Dropout(rate = 0.2))

    model.add(Dense(HL1))
    model.add(Dense(HL2, activation='relu'))

    #model.add(Dropout(rate = dropout1))
    model.add(LeakyReLU(alpha=alpha1))
    #model.add(Dropout(rate = dropout1))

    model.add(LeakyReLU(alpha=alpha1))
    #model.add(Dropout(rate = dropout1))

```

```

model.add(Dense(HL4, activation='relu'))
# model.add(Dropout(rate = dropout1))
model.add(Dense(input_dim,activation='sigmoid'))

#### Compile and train the network ####
optimizer1 = keras.optimizers.Adam(lr = learning_rate, beta_1
= 0.9, beta_2 = 0.999, amsgrad = False)
model.compile(optimizer = optimizer1, loss = loss, metrics =
metrics)

#keras.optimizers.SGD(lr = 0.99, momentum = 0.99, nesterov =
True)
#model.compile(loss = loss, optimizer = 'SGD', metrics =
metrics)

es = EarlyStopping(monitor = 'val_loss', mode = 'min', verbose
= 1, patience = 10)
model_name = os.path.join(current_dir, 'models',
f'best_model_{terrain}_endpt_{end_pt}.h5')
mc = ModelCheckpoint(model_name, monitor='val_loss',
mode='min', verbose=1, save_best_only=True)
model.fit(X_train_now, y_train_now, batch_size = batch_size,
epochs = epoch, class_weight = weights, validation_data=(X_val_now,
y_val_now), callbacks=[es, mc])

# load the saved model
saved_model = load_model(model_name)

#### Evaluate the model fit ####
scores1 = saved_model.evaluate(X_train_now, y_train_now,
verbose=0)
scores2 = saved_model.evaluate(X_val_now, y_val_now,
verbose=0)
scores3 = saved_model.evaluate(X_test_now, y_test_now,
verbose=0)

# Predict train, val, and test sets
yhat_train = model.predict(X_train_now)
yhat_val = model.predict(X_val_now)
yhat_test = model.predict(X_test_now)

# Flatten data for ROC and AUC calcs
yhat_train_flat = yhat_train.flatten()
yhat_val_flat = yhat_val.flatten()
yhat_test_flat = yhat_test.flatten()

y_train_now_flat = y_train_now.values.flatten()
y_val_now_flat = y_val_now.values.flatten()
y_test_now_flat = y_test_now.values.flatten()

# Area under ROC curve metric
auc_train = roc_auc_score(y_train_now_flat, yhat_train_flat)

```

```

auc_val = roc_auc_score(y_val_now_flat, yhat_val_flat)
auc_test = roc_auc_score(y_test_now_flat, yhat_test_flat)

# Records accuracy and AUC
score_set[index,:] = [end_pt, scores1[1], auc_train,
scores2[1], auc_val, scores3[1], auc_test]
index += 1

np.savetxt(os.path.join(current_dir,
'Results',f'results_{terrain}_increasing_dis.csv'),
score_set,
delimiter=',')

# See if the AUC for the validation set is the best so far
for the terrain
# If it is better than the best so far, compute metric for a
ROC curve
if auc_val > auc_val_best:
    auc_val_best = auc_val
    #auc_train_best = auc_train
    #auc_test_best = auc_test

# Rind ROC curve
fpr_train, tpr_train, threshold_train =
roc_curve(y_train_now_flat, yhat_train_flat)
fpr_val, tpr_val, threshold_val = roc_curve(y_val_now_flat,
yhat_val_flat)
fpr_test, tpr_test, threshold_test =
roc_curve(y_test_now_flat, yhat_test_flat)

ROC_values = [end_pt, auc_train, auc_val, auc_test,
fpr_train, tpr_train, threshold_train, fpr_val, tpr_val,
threshold_val, fpr_test, tpr_test, threshold_test]
ROC_value_file_name = os.path.join(current_dir,
'Results',f'ROC_values_{terrain}.csv')
with open(ROC_value_file_name,'wb') as f:
    for row in ROC_values:
        np.savetxt(f, [row], fmt = '%5f', delimiter = ',')

## Plot results
#coast_results = np.array(pd.read_csv(os.path.join(current_dir,
'Results', 'results_coast_increasing_dis.csv'), header = None))
#mountains_results = np.array(pd.read_csv(os.path.join(current_dir,
'Results', 'results_mountains_increasing_dis.csv'), header = None))
#plains_results = np.array(pd.read_csv(os.path.join(current_dir,
'Results', 'results_plains_increasing_dis.csv'), header = None))

fig = plt.figure(figsize=(15, 5))
width = 0.2
ind = np.arange(len(mountains_results))

ax = fig.add_subplot(131)

```

```

#ax.bar(ind, plains_results[:,1], width, color = 'green', label =
'Plains')
#ax.bar(ind + width, coast_results[:,1], width, color = 'blue', label
= 'Coast')
ax.bar(ind + 2*width, mountains_results[:,1], width, color = 'grey',
label = 'Mountains')
ax.set_xticks(ind + width)
#ax.set_xticklabels((plains_results[:,0]).astype(int))
ax.set_xlabel('Predicted Distance (meters)')
ax.set_ylabel('Prediction Accuracy')
ax.set_title('Training Data')
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15),
fancybox=True, shadow=True, ncol=3)

ax = fig.add_subplot(132)
#ax.bar(ind, plains_results[:,2], width, color = 'green', label =
'Plains')
#ax.bar(ind + width, coast_results[:,2], width, color = 'blue', label
= 'Coast')
ax.bar(ind + 2*width, mountains_results[:,2], width, color = 'grey',
label = 'Mountains')
ax.set_xticks(ind + width)
#ax.set_xticklabels((plains_results[:,0]).astype(int))
ax.set_xlabel('Predicted Distance (meters)')
#ax.set_ylabel('Prediction Accuracy')
ax.set_title('Validation Data')
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15),
fancybox=True, shadow=True, ncol=3)

ax = fig.add_subplot(133)
#ax.bar(ind, plains_results[:,3], width, color = 'green', label =
'Plains')
#ax.bar(ind + width, coast_results[:,3], width, color = 'blue', label
= 'Coast')
ax.bar(ind + 2*width, mountains_results[:,3], width, color = 'grey',
label = 'Mountains')
ax.set_xticks(ind + width)
#ax.set_xticklabels((plains_results[:,0]).astype(int))
ax.set_xlabel('Predicted Distance (meters)')
#ax.set_ylabel('Prediction Accuracy')
ax.set_title('Test Data')
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15),
fancybox=True, shadow=True, ncol=3)

fig.tight_layout()
#plt.show()
plt.savefig(os.path.join(current_dir, 'Images',
'results_all_increasing_dis.png'))

# Save model summary
plot_model(model, to_file = os.path.join(current_dir, 'Images',
'NN_model.png'), show_shapes = True, show_layer_names = True)

```

```

# Produce a ROC curve for each terrain set
# fig, axs = plt.subplots(1,3, figsize=(15, 5))
# lw = 2
# i = 0

# for terrain in terrain_set:
#     ROC_value_file_name = os.path.join(current_dir, 'Results',
# f'ROC_values_{terrain}.csv')
#     with open(ROC_value_file_name) as file:
#         data_all = csv.reader(file, delimiter = ',')
#         data = list(data_all)
#         end_pt = data[0]
#         auc_train = data[1]
#         auc_val = data[2]
#         auc_test = data[3]
#         fpr_train = np.array(data[4])
#         tpr_train = np.array(data[5])
#         threshold_train = np.array(data[6])
#         fpr_val = np.array(data[7])
#         tpr_val = np.array(data[8])
#         threshold_val = np.array(data[9])
#         fpr_test = np.array(data[10])
#         tpr_test = np.array(data[11])
#         threshold_test = np.array(data[12])

#         fpr_train_unique, ROC_train_unique_indices =
np.unique(fpr_train, return_index=True)
#         fpr_val_unique, ROC_val_unique_indices = np.unique(fpr_val,
return_index=True)
#         fpr_test_unique, ROC_test_unique_indices = np.unique(fpr_test,
return_index=True)

#         tpr_train_unique = tpr_train[ROC_train_unique_indices]
#         tpr_val_unique = tpr_val[ROC_val_unique_indices]
#         tpr_test_unique = tpr_test[ROC_test_unique_indices]

#         axs[i].plot(fpr_train_unique, tpr_train_unique, color =
'darkblue', lw = lw, label = f'Train (area = {auc_train[0]})')
#         axs[i].plot(fpr_val_unique, tpr_val_unique, color='darkred', lw
= lw, label = f'Validation (area = {auc_val[0]})')
#         axs[i].plot(fpr_test_unique, tpr_test_unique, color =
'darkorange', lw = lw, label = f'Test (area = {auc_test[0]})')
#         axs[i].plot([0, 1], [0, 1], color='navy', lw = lw, linestyle =
'--')
#         axs[i].set_xlim([0.0, 1.0])
#         axs[i].set_ylim([0.0, 1.05])
#         axs[i].set_xlabel('False Positive Rate')
#         axs[i].set_ylabel('True Positive Rate')
#         axs[i].set_title(f'ROC for {terrain} Terrain and {end_pt} Data
Points')
#         axs[i].legend(loc = "lower right")

#     i += 1

```

```

# fig.tight_layout()
# plt.show()

# plt.savefig(os.path.join(current_dir, 'Images', 'ROC_Curves.png'))

## Calculate final run time and show complete
time_delta = datetime.now() - startTime
delta_hour = time_delta.seconds//3600
delta_min = ((time_delta.seconds - (delta_hour * 3600))//60)
delta_sec = (time_delta.seconds - delta_hour*3600 - delta_min * 60)%60

print(' ')
print(' ')
print('#####')
print('Computations Complete')
print(f'Time to complete analysis: {delta_hour} hours, {delta_min}
minutes, {delta_sec} seconds')
print('#####')

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX K: NN PREDICTION SCRIPT

This script, STEP_7B_PREDICTION.py, was used to make a prediction upon testing data.

```
# NN PLAINS TRAINING MODEL (COMPILE h.5 files, make prediction)

# CITATIONS
#https://machinelearningmastery.com/how-to-save-and-load-models-and-
#data-preparation-in-scikit-learn-for-later-use/
# https://stackoverflow.com/questions/35074549/how-to-load-a-model-
#from-an-hdf5-file-in-keras
#
#https://pyts.readthedocs.io/en/stable/generated/pyts.preprocessing.M
#inMaxScaler.html
# https://stackoverflow.com/questions/53152627/saving-
#standardscaler-model-for-use-on-new-datasets

# IMPORT LIBRARIES
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
from keras.models import Model, Sequential, load_model
import os
# Save some models with pickle
#Saving models requires the h5py lib as well.
import logging
logging.getLogger('tensorflow').setLevel(logging.ERROR)
from timeit import default_timer as timer
from datetime import datetime
from warnings import simplefilter
from sklearn.externals import joblib
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
import gc

# SET TIMER
startTime = datetime.now()
start = timer()

# SET WARNINGS
simplefilter(action='ignore', category=FutureWarning)

# SET DIRECTORY
os.chdir('/media/gman/Grant/thesis_complete/NN_Files/test_coastlines
/')

```

```

#directory
r'/media/gman/Grant/Thesis_Maps/test_runs_2/test_maps/plains_complet
e_maps/h5_files/'

# LOAD SCALAR FILE
scaler_filename = 'FFNN_LOS_scaler.save'
min_max_scaler = joblib.load(scaler_filename)

# LOAD DATA MAP
plains_data='coastline_combined_cleaned_small.csv'
plains = pd.read_csv(plains_data, header=None, encoding='latin1')
frames = [plains]
df = pd.concat(frames)

# DATA PREP
n = 1000
nrows = len(df) #Number of rows in the input file
nsets = int(nrows/n) #Divide the number of input file rows by 1000.
input_set = np.empty((nsets, n))
output_set = np.zeros((nsets, n))

ind = 0
for k in range(nsets):
    end_ind = ind+n
    input_chunk = df.iloc[ind:end_ind,2] #Use only the z-axis
(altitude) data.
    output_chunk = df.iloc[ind:end_ind,4]
    input_set[k,:] = input_chunk #use input_set for data, it is the
correct shape.
    output_set[k,:] = output_chunk #use output_set for output data.
    ind += n

# BREAKUP DATA INTO X AND Y
data_scaled = min_max_scaler.transform(input_set)
X = pd.DataFrame(data_scaled)
y = pd.DataFrame(output_set)

data_len=list(range(600,700,600))
score_set = np.zeros((len(data_len),3))
ind = 0
for end_pt in data_len:
    h5_file = 'best_model_coast_endpt_600.h5'
    model = load_model(h5_file)
    model.summary()
    X_run = X.iloc[:,0:end_pt]
    y_run = y.iloc[:,0:end_pt]

    # ITERATE OVER .H5 FILES

    # MODEL EVALUATION
    # Try two different ways to predict, and see what happens.
    score = model.evaluate(X_run, y_run, verbose=0)
    print("%s: %.2f%%" % (model.metrics_names[1], score[1]*100))

```

```

yhat_train = model.predict(X_run)
yhat_train_flat = yhat_train.flatten()
y_train_now_flat = y_run.values.flatten()
auc_train = roc_auc_score(y_train_now_flat, yhat_train_flat)

score_set[ind,:] = [end_pt, score[1], auc_train]
ind += 1
# ynew = model.predict(X_run)
# y1 = np.round(ynew, 0)
# acc = accuracy_score(y_run, y1)
# print('Test Accuracy:', acc*100)

# scores1 = model.evaluate(X_run, y1, verbose=0)
# score_set[ind,:] = [end_pt, score[1]]
# ind += 1
np.savetxt('coastline_test_file.csv', score_set, delimiter=',')
gc.collect()

# TIMER
print("The run time in seconds is:", timer()-start)

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX L: ORIGINAL LOS CODE

The original synthetic terrain generator and LOS calculation is given here.

1-syntheticLinearterrainandLOS_mod.py

```
## Generating some synthetic terrain and corresponding visibility masks
with viewpoint at far left.

# Some other options for generating random terrain - I'm looking for my
source on this 17 OCT 19.
#https://gis.stackexchange.com/questions/20955/generating-random-but-
believable-digital-elevation-model
#http://vterrain.org/Elevation/Artificial/
# This can readily be adapted to build 2d terrain as well.

hillmin=10
hillmax=150
numhills=150
flattening=2
terrainLength=400
observerHeight=2
targetHeight=0

seed_num=42

want_plot=1
want_terrain_set=0

numSamples=100
terrain_file_name='newTerrains17Oct.npy'
visibility_file_name='newVisibility17Oct.npy'

import numpy as np
import matplotlib.pyplot as plt

# fix random seed for reproducibility
np.random.seed(seed_num)

def genLinearTerrain(terrainLength, hillmin, hillmax, numhills,
flattening):
    #Terrain gen works by 'adding' circular (would be spherical in 2d)
objects to the surface of the terrain.
    #Adding many of these of varying sizes...then squaring, results in
fairly realistic looking terrain.
    #'flattening' is simply taking the terrain values to a power - this
makes the peaks steeper and the bottoms flatter
    # similar to real terrain.
    terrain=np.zeros(terrainLength)
```

```

    for i in range(0,numhills):
        # add hill
        radius=np.random.uniform(hillmin, hillmax, 1)

        centerpoint=(np.random.uniform(0-
hillmax,terrainLength+hillmax))

        #based on centerpoint find which points along the line are
        effected by adding a circle.
        minPointAffected=np.round(centerpoint-radius).astype(int)[0]
        maxPointAffected=np.round(centerpoint+radius).astype(int)[0]

        if (minPointAffected < 0):
            minPointAffected=0

        if (maxPointAffected > terrainLength):
            maxPointAffected = terrainLength

        for j in range(minPointAffected, maxPointAffected):

            val=np.sqrt((radius**2-(centerpoint-j)**2))
            #if the value is 0
            if(np.isnan(val)):
                val=0
            terrain[j]=terrain[j]+val

    terrain=terrain**flattening

    terrain=(terrain-np.mean(terrain))/np.std(terrain)

    return(terrain)

def losCal(elevationsVector, observer_Height, objectHeight):

    lengthOfVector=len(elevationsVector)
    distances=np.array(list(range(0,lengthOfVector)))
    distances[0]=.01 # just to avoid a divide by 0 error - this pixel
will be visible anyway.
    #gradients=(elevationsVector[1:lengthOfVector])/distances
    visibility = np.zeros(lengthOfVector)
    #surfgradients = np.zeros(lengthOfVector)
    observerGradients = np.zeros(lengthOfVector)

    visibility[0]=1
    visibility[1]=1
    for i in range(1,lengthOfVector):
        #surfgradients[i] = (elevationsVector[i]-(elevationsVector[0] +
observer_Height)) / distances[i]
        observerGradients[i] = ((elevationsVector[i] + objectHeight)-
(elevationsVector[0] + observer_Height)) / distances[i]

```

```

maxheightLine=observerGradients[i]*distances[0:i]+(elevationsVector[0]
+observer_Height)

    canSee=sum((elevationsVector[0:i]>maxheightLine)*1)

    if canSee==0:
        visibility[i]=1

    #if (i>1):
        #if (observerGradients[i] > max(surfgradients[1:i])):
            #visibility[i] = 1

    return (visibility)

if want_terrain_set==1:

    terrains=np.zeros((numSamples, terrainLength))

    for i in range(0, numSamples):
        terrains[i]=genLinearTerrain(terrainLength, hillmin, hillmax,
numhills, flattening)
        if (i % 1000) == 0:
            print(i)

    #np.save('newTerrains.npy', terrains)
    #terrains=np.load('newTerrains.npy')
    #losCal(terrains[1:], ObserverHeight, targetHeight)

    visibility = np.apply_along_axis(losCal, 1, terrains,
observer_Height=observerHeight, objectHeight=targetHeight )

    np.save(terrain_file_name, terrains)
    np.save(visibility_file_name, visibility)

#This section demonstrates the functionality of both functions with a
crude display.
if want_plot==1:
    terrain=genLinearTerrain(terrainLength, hillmin, hillmax, numhills,
flattening)
    result=losCal(terrain, observerHeight, targetHeight)

    xpts=np.array(list(range(0, len(terrain))))

    canSeeTerrain=np.ma.masked_where(result==0, terrain)
    noSeeTerrain=np.ma.masked_where(result==1, terrain)
    observer_plt=terrain[0] + observerHeight

    fig, ax = plt.subplots()
    ax.plot(xpts, terrain)
    ax.plot(xpts, canSeeTerrain, color='blue', label='Can See')
    ax.plot(xpts, noSeeTerrain, color='red', label='Can not See')
    #plt.scatter(xpts, canSeeTerrain, color='blue')

```

```
#plt.scatter(xpts,noSeeTerrain,color='red')

ax.plot(0, observer_plt, 'bo', label='Observer')
plt.title('Synthetic Terrain')
ax.legend()

plt.show()

#plt.plot(terrain)
#plt.plot(result)
#plt.show()
```

APPENDIX M: ORIGINAL NN CODE

The original NN program that would make use of the results from 1-*syntheticLinearterrainandLOS_mod.py*. The authors were LTC Brian Wade, MAJ James Jablonski CPT Sean Clement [5].

2-linearNNETSyntheticTest.py

```
from keras.layers import Dense, Input, LocallyConnected1D, Dropout,
BatchNormalization, Reshape, Flatten, LeakyReLU
from keras.models import Model, Sequential
from keras import optimizers
import numpy as np

# fix random seed for reproducibility
np.random.seed(7)

import os, sys

# change to the directory with your data in it!!
#os.chdir('c://Users//jabal//Desktop//Army TRAC MTRY Projects//LoS
Calculation Code and Document Sharing Project Folder')
os.chdir('/media/gman/John/Aerospace_Classes/Thesis_Programs')

terrainS=np.load('newTerrains.npy')
vis=np.load('newVisibility.npy')

##Standardize data by 'row' so each grid presented to the nnet is the
same magnitude.
#Normal standardization by column would change the ratio of features on
the exemplar map areas.
#Not standardizing the data resulted in far lower accuracy - maxing out
at about 77% acc.
# just reduce memory requirements, no need to store as a float, may work
as logical.
terrainS=terrainS.astype('float32')
for i in range(len(terrainS)):
    terrainS[i,:]=(terrainS[i,:]-
np.mean(terrainS[i,:])/np.std(terrainS[i,:]))

vis=vis.astype(int)

msk = np.random.rand(len(terrainS)) < 0.9

terrainStrain = terrainS[msk]
terrainStest = terrainS[~msk]

vistrain=vis[msk]
vistest=vis[~msk]
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX N: DEM MAP COVERAGE

Several U.S. maps showing DEM coverage based on resolution are given below.

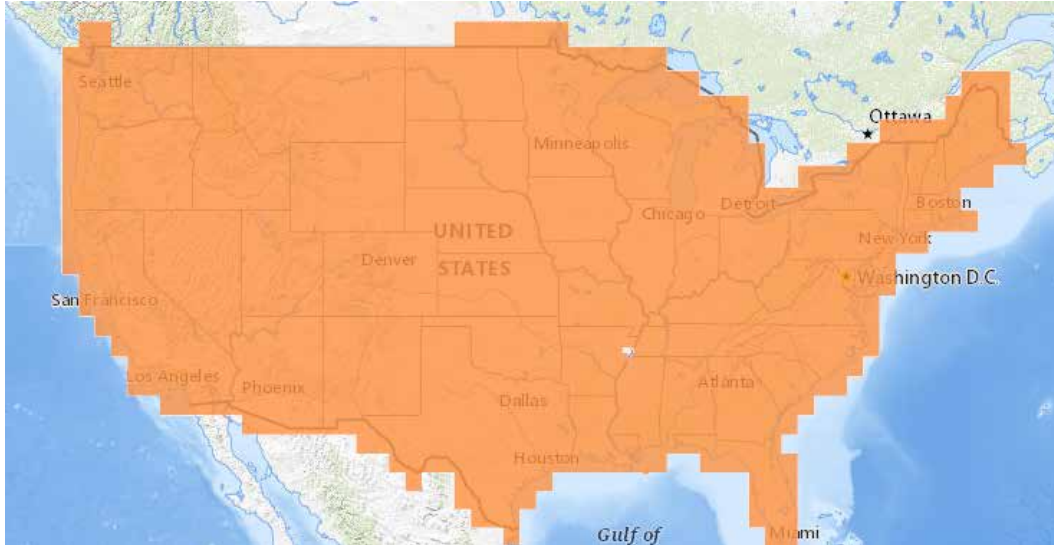


Figure 17. 1/3 arc-second U.S. map coverage, highlighted in orange.
Source: [20].

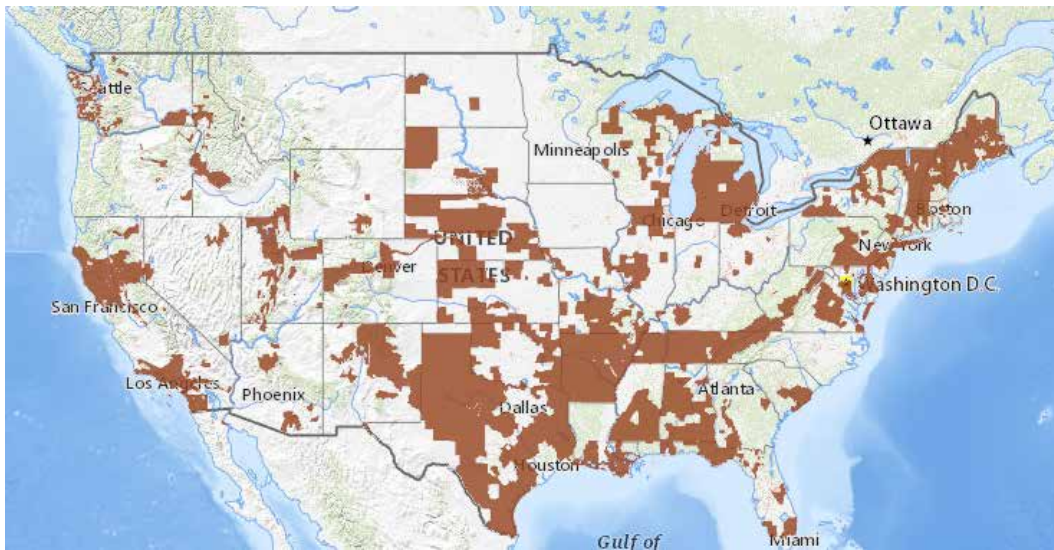


Figure 18. 1-meter DEM U.S. map coverage, highlighted in brown.
Source: [20].

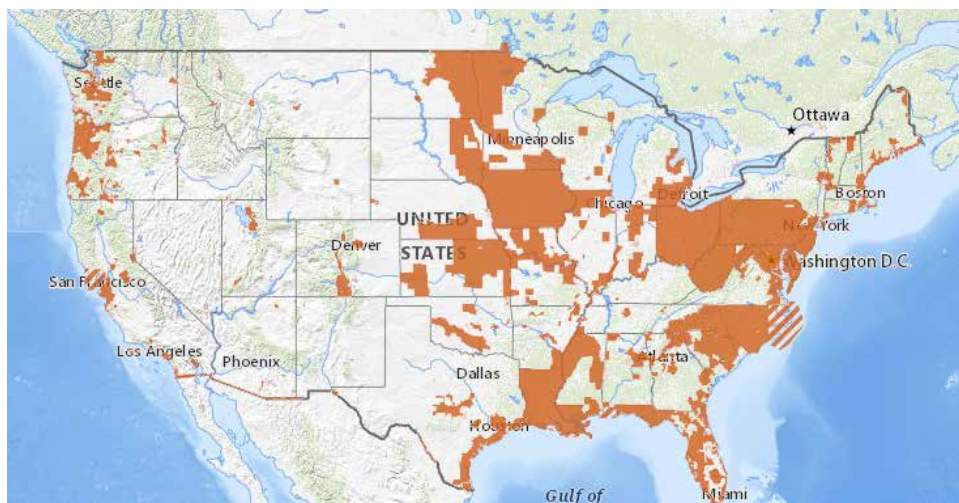


Figure 19. 1/9 arc-second U.S. coverage map, highlighted in orange. Source: [20].



Figure 20. 2 Arc-Second DEM Alaska map coverage, highlighted in orange. Source: [20].

LIST OF REFERENCES

- [1] “Viewshed,” Oxford. [Online]. Available: <https://www.merriam-webster.com/dictionary/viewshed>. [Accessed 22 August 2020].
- [2] “Attack helicopters,” Military Factory. [Online]. Available: <https://www.militaryfactory.com/aircraft/military-attack-helicopters.asp>. [Accessed 17 April 2020].
- [3] “Nap-of-the-Earth,” wikia, Inc. [Online]. Available: <https://military.wikia.org/wiki/Nap-of-the-earth>. [Accessed 21 May 2020].
- [4] J. Grant, *F15 flying in Rainbow Canyon* [photograph]. Death Valley, CA: John Grant, 2018.
- [5] B. Wade, S. Clement, and J. Jablonski, interview, November 2019.
- [6] L. Ford, H.S. Isaacson, and F.C. Pethel, “Computer terrain simulation for line-of-sight calculations,” *Operations Research*, vol. 7, no. 4, p. 6, 1959.
- [7] R.B. Seixas, M.R. Mediano, M. Gattass, and R. De, “Efficient line-of-sight algorithms for real terrain data,” *Instituto Nacional de Matematica Pura e Aplicada & TECGRAF/PUC-Rio*, Rio De Janeiro, 1999.
- [8] W. Baer, B. Mansager, W.G. Kemple, and J. Illingsworth, “Weapons design analysis using 1-meter terrain resolution battlefield simulators,” Naval Postgraduate School, Monterey, CA. 2000.
- [9] W. Feng, W. Gang, P. Deji, L. Yuan, Y. Liuzhong and W. Hongbo, “A parallel algorithm for viewshed analysis in three-dimensional digital earth,” *Computers & Geosciences*, vol. 75, pp. 57–65, 2015.
- [10] J. Golemboski, *Analysis of cloud-free line-of-sight probability calculations*, Wright-Patterson Air Force Base, Ohio: Air Force Institute of Technology, 2012.
- [11] B. Duvenhage, “Using an implicit min/max KD-Tree for doing efficient terrain line of sight calculations,” *AFRIGRAPH '09: Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, Feb. 2009, pp. 81–90, doi: 10.1145/1503454.1503469.
- [12] M.V. Andrade and S. Magalhaes, “Efficient viewshed computation on terrain in external memory,” *GeoInformatica*, vol. 15, no. 2, pp. 381–397, 2011.

- [13] G. A. Schiavone, J. Tracy, E. Woodruff, M. Gerber, T. Dere, and J. De La Cruz, "Toward accelerated line-of-sight intervisibility calculations using clusters of GPU's," *Proceedings of SPIE*, vol. 5423, no. 1, pp. 252–263, 2004.
- [14] S. Tabik, A. R. Cervilla, E. Zapata, and L. F. Romero, "Efficient data structure and highly scalable algorithm for total-viewshed computation," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, no. 1, pp. 304–310, 2015.
- [15] F. Chollet, "Introduction to Keras for Engineers," 1 04 2020. [Online]. Available: https://keras.io/getting_started/intro_to_keras_for_engineers/. [Accessed 10 June 2020].
- [16] S. Raschka, V. Mirjalili, *Python Machine Learning*, 2nd ed. Birmingham-Mumbai: Packt Publishing Ltd., 2017.
- [17] "TensorFlow." [Online]. Available: <https://www.tensorflow.org>. [Accessed 10 February 2020].
- [18] C. Stadler, X. Flamm, T. Gruber, A. Djanatljev, R. German, and D. Eckhoff, "A stochastic V2V LOS/NLOS model using neural networks for hardware-in-the-loop testing," *2017 IEEE Vehicular Networking Conference (NVC)*, vol. 2017, no. 11.
- [19] "What are digital elevation models (DEMs)?," USGS. [Online]. Available: https://www.usgs.gov/faqs/what-are-digital-elevation-models-dems?qt-news_science_products=0#qt-news_science_products. [Accessed 27 April 2020].
- [20] "TNM Download (V1.0)," USGS, 1 January 2020. [Online]. Available: <https://viewer.nationalmap.gov/basic/>. [Accessed 1 May 2020].
- [21] "Digital elevation models," ESRI, 2019. [Online]. Available: <https://learn.arcgis.com/en/related-concepts/digital-elevation-models.htm>. [Accessed 17 April 2020].
- [22] B. Wade, "Lesson 9: Controlling for overfitting," class notes, Naval Postgraduate School, Monterey, CA, ME4901, Summer 2019.
- [23] B. Wade, "Lesson 19: Artificial Neural Networks Training Methods," class notes, Naval Postgraduate School, Monterey, CA, ME4901, Summer 2019.
- [24] B. Wade, "Lesson 2: Artificial intelligence and machine learning," class notes, Naval Postgraduate School, Monterey, CA, ME4901, Summer 2019.

- [25] F. Chollet, *Deep Learning with Python*, Shelter Island, NY: Manning Publication Co., 2018.
- [26] H. Asemi, “What is deep learning?,” Analytics Vidhya, 15 February 2020. [Online]. Available: <https://medium.com/analytics-vidhya/what-is-deep-learning-7baea9610511>. [Accessed 7 April 2020].
- [27] T. Gupta, “Deep Learning: Feedforward neural network,” Medium, 4 January 2017. [Online]. Available: <https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7>. [Accessed 22 August 2020].
- [28] B. Wade, “Lesson 18: Artificial neural networks – Forward propagation,” class notes, Naval Postgraduate School, Monterey, CA, ME4901, Summer 2019.
- [29] B. Wade, “Lesson 16: Artificial Neural Networks/Case Study III Intro,” class notes, Naval Postgraduate School, Monterey, CA, ME4901, Summer 2019.
- [30] R. Tadeusiewicz, R. Chaki, N. Chaki, T. Gaciarz, B. Borowik, and B. Leper, *Exploring neural networks with C#*. Boca Raton: CRC Press, 2016.
- [31] J. Brownlee, “How to choose loss functions when training deep learning neural networks,” Machine Learning Mastery, 30 January 2019. [Online]. Available: <https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>. [Accessed 15 April 2020].
- [32] “Introduction to loss functions,” Algorithmia, 30 April 2018. [Online]. Available: <https://algorithmia.com/blog/introduction-to-loss-functions>. [Accessed 15 April 2020].
- [33] “Usage of loss functions,” Keras SIG. [Online]. Available: <https://keras.io/losses/>. [Accessed 15 April 2020].
- [34] S. Tiwari, “Activation functions in Neural Networks,” GeeksforGeeks. [Online]. Available: <https://www.geeksforgeeks.org/activation-functions-neural-networks/>. [Accessed 15 July 2020].
- [35] A. Sharma V, “Understanding activation functions in neural networks,” 30 March 2017. [Online]. Available: <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>. [Accessed 15 May 2020].

- [36] C. Hansen, "Activation functions explained," 10 July 2020. [Online]. Available: <https://mlfromscratch.com/activation-functions-explained/#/>. [Accessed 10 July 2020].
- [37] A. Bilogur, "Keras optimizers," Kaggle.com, 2019. [Online]. Available: <https://www.kaggle.com/residentmario/keras-optimizers>. [Accessed 17 May 2020].
- [38] "What are the drawbacks of Python?," Raps Mk, 2020. [Online]. Available: <http://net-informations.com/python/iq/disadvantages.htm>. [Accessed 1 April 2020].
- [39] "Rectified linear units (ReLU) In deep learning," Kaggle, 30 June 2020. [Online]. Available: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning>. [Accessed 10 July 2020].
- [40] B. Wade, "Classification model evaluation," Naval Postgraduate School, Monterey, 2020.
- [41] M. Hudson, "Battle of the Teutoburg Forest," Encyclopedia Britannica, 1 November 2019. [Online]. Available: <https://www.britannica.com/event/Battle-of-the-Teutoburg-Forest>. [Accessed 2 December 2019].
- [42] K. Schousboe, "Battle of Teutoburg Forest," 6 February 2017. [Online]. Available: <https://www.ancient.eu/article/1010/battle-of-teutoburg-forest/>. [Accessed 11 1 2020].
- [43] M. Ray, "Battle of Trenton and Princeton," Encyclopedia Britannica, 14 March 2020. [Online]. Available: <https://www.britannica.com/event/Battles-of-Trenton-and-Princeton>. [Accessed 22 August 2020].
- [44] "Battles of Isandlwana and Rorke's Drift," Encyclopedia Britannica, 15 January 2019. [Online]. Available: <https://www.britannica.com/event/Battles-of-Isandlwana-and-Rorkes-Drift>. [Accessed 2 December 2019].
- [45] T. Hughes, "12 Facts about the Battle of Isandlwana," HistoryHit, 5 December 2018. [Online]. Available: <https://www.historyhit.com/facts-about-the-battle-of-isandlwana/>. [Accessed 2 December 2019].
- [46] M. Ray, "Battle of Midway," *Encyclopedia Britannica*, 6 November 2019. [Online]. Available: https://www.britannica.com/event/Battle-of_Midway. [Accessed 1 December 2019].
- [47] "History of the Perceptron," CSULB. [Online]. Available: <http://web.csulb.edu/~cwallis/artificialn/History.htm>. [Accessed 5 April 2020].

- [48] D. A. Turing, “The imitation game,” *Computing Machinery and Intelligence*, 1 October 1950. [Online]. Available: <https://academic.oup.com/mind/article/LIX/236/433/986238>. [Accessed 4 April 2020].
- [49] J. Akst, “Machine learning, 1951,” LabX Media Group, 30 April 2019. [Online]. Available: <https://www.the-scientist.com/foundations/machine--learning--1951-65792>. [Accessed 10 April 2020].
- [50] B. Marr, “A short history of machine learning — Every manager should read,” *Forbes*, 19 February 2016. [Online]. Available: <https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#3b3d977d15e7>. [Accessed 15 April 2020].
- [51] J. Lee, “Donald Michie,” IEEE Computer Society. [Online]. Available: <https://history.computer.org/pioneers/michie.html>. [Accessed 15 April 2020].
- [52] M. Minsky, S. Papert, and L. Bottou, “Perceptrons: An introduction to computational geometry,” January 1969. [Online]. Available: <https://mitpress.mit.edu/books/perceptrons>. [Accessed 13 April 2020].
- [53] J. McCarthy, “Arthur Samuel: Pioneer in machine learning,” Stanford University. [Online]. Available: <http://infolab.stanford.edu/pub/voy/museum/samuel.html>. [Accessed 2 April 2020].
- [54] S. T. Linnainmaa, “Expansion of the accumulated rounding error.” *BIT* 16, pp. 146–160 (1976). <https://dio-org.libproxy.nps.edu/10.1007/BF01931367>.
- [55] D. Rumelhart, H. Geoffrey, and W. Ronald, “Learning representations by back-propagating errors,” Institute of Cognitive Science, University of California, 1986.
- [56] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *PubMed*, December 1997. [Online]. Available: https://www.researchgate.net/publication/13853244_Long_Short-term_Memory. [Accessed 5 April 2020].
- [57] T. K. Ho, “Random decision forests,” *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, pp. 278–282, 1995.
- [58] A. Chowdhry, “Facebook’s DeepFace software can match faces with 97.25% accuracy,” *Forbes*, 18 March 2014. [Online]. Available: <https://www.forbes.com/sites/amitchowdhry/2014/03/18/facebooks-deepface-software-can-match-faces-with-97-25-accuracy/#6fb9682f54fc>. [Accessed 11 April 2020].

- [59] J. Kacprzyk, L. Jain, V. Palade, J. Kacprzyk, V. Palade, L. Lakhmi, and D. Srinivasan, *Advances in evolutionary computing for system design*, Berlin: Springer, 2007.
- [60] Y. Yu, C. Guo, and H. Yu, “Finite-time predictor line-of-sight-based adaptive neural network path following for unmanned surface vessels with unknown dynamics and input saturation,” *International journal of Advanced Robotic Systems*, vol. 15, no. 6, 2018.
- [61] M. Danese, G. Nole, B. Murgante, G. Borruoso, A. Lapuci, and B. Murgante, “Identifying viewshed: New approaches to visual impact assessment,” *Studies in Computational Intelligence*, vol. 348, pp. 73–89, 2011.
- [62] S. Hagstrom and D. Messinger, “Line of sight analysis using voxelized discrete lidar,” *Proceedings of SPIE – The International Society for Optical Engineering*, vol. 8037, 2011.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California