REDUCING THE COMPUTATIONAL COST OF URBAN FLOOD PREDICTION IN LOS ANGELES

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By

Laura Ani Bedoyan

December 2023
The thesis of Laura Ani Bedoyan is approved:

Mansoureh Lord, M.S.  

John Noga, Ph.D.  

Adam Kaplan, Ph.D., Chair

California State University, Northridge
Dedication

I dedicate this thesis project to my parents and sister.
Acknowledgements

Thank you to Ms. Mansoureh Lord for teaching me the fundamentals of Machine and Deep Learning. Thank you to Dr. Adam Kaplan and Dr. John Noga for your continuous support and guidance throughout this project. Thank you to the Los Angeles Department of Water and Power for providing preliminary data.
Table of Contents

Signature page ii
Dedication iii
Acknowledgements iv
List of Figures vii
Abstract ix
1 Introduction 1
  1.1 Background ....................................................... 1
  1.2 GIS Vocabulary .................................................... 1
  1.3 Stream gauges ..................................................... 2
  1.4 Objective ......................................................... 2
2 Related Work 3
3 Methodology 5
  3.1 Hardware ........................................................ 5
  3.2 Google Colab ....................................................... 5
  3.3 Libraries ........................................................ 6
    3.3.1 GDAL ......................................................... 6
    3.3.2 Google Earth Engine ........................................ 6
    3.3.3 PyTorch, Numpy, Scikit Learn, Pandas, Keras .............. 6
  3.4 Data Sources ...................................................... 6
  3.5 Area of Interest ................................................ 6
  3.6 Feature Engineering ............................................ 6
    3.6.1 Slope and Aspect ............................................ 7
    3.6.2 Sentinel-1 VV .............................................. 7
    3.6.3 Gravity Main Lines ......................................... 8
    3.6.4 Catch Basins ............................................... 9
    3.6.5 Lateral Lines ............................................... 9
    3.6.6 Social Sensitivity Index .................................. 10
    3.6.7 Levees ...................................................... 11
    3.6.8 Height ....................................................... 11
    3.6.9 Rainfall ..................................................... 12
    3.6.10 Vector to Raster .......................................... 13
    3.6.11 Raster to Numpy ........................................... 13
    3.6.12 Covariance Matrix ....................................... 13
    3.6.13 Patches ................................................... 14
  3.7 Architecture and Algorithms .................................. 14
    3.7.1 Overview .................................................... 14
    3.7.2 Autoencoder ............................................... 15
    3.7.3 Multi-Layer Perceptron ................................... 19
    3.7.4 K-Means Clustering ....................................... 22
4 Results 25
  4.0.1 Autoencoder .................................................. 25
  4.0.2 Multi-Layer Perceptron ..................................... 26
  4.0.3 K-Means Clustering ......................................... 28
  4.0.4 Time Complexity Analysis .................................. 29
List of Figures

1.1 Raster Image, from ESRI [2] ......................................................... 2
1.2 Stream gauge, from University of Texas [3] ................................. 2
3.1 Area of Interest .............................................................................. 7
3.2 Slope and Aspect Outputs of DEM of Bell Gardens and Downey .... 7
3.3 Sentinel-1 VV ............................................................................... 8
3.4 Average Aspect and Slope by Flooded Region .............................. 8
3.5 Gravity Main ................................................................................ 9
3.6 Catch Basin ................................................................................ 9
3.7 Lateral Line ................................................................................. 10
3.8 Social Sensitivity Index ................................................................. 10
3.9 Mean Social Sensitivity Index by Flooded Region ......................... 11
3.10 Levees ....................................................................................... 11
3.11 Height ....................................................................................... 12
3.12 Rainfall ..................................................................................... 12
3.13 Dataframe of Rainfall and Height Time Series for F45B and nearby AL383 ........................................................................ 13
3.14 Covariance Matrix ..................................................................... 14
3.15 Architecture ............................................................................... 14
3.16 PCA ......................................................................................... 15
3.17 Autoencoder Model .................................................................... 15
3.18 Autoencoder Specifics ................................................................. 16
3.19 Multi-Layer Perceptron Overview ............................................. 19
3.20 Embedded Vector Dataframe Flattened ...................................... 19
3.21 Multi-Layer Perceptron Data Frame Input ................................. 20
3.22 Embedded Vector Dataframe Statistics .................................... 20
3.23 Multi-Layer Perceptron Detailed Structure ............................... 21
3.24 Multi-Layer Perceptron Structure ............................................ 22
3.25 K Means Clustering Overview .................................................. 23
4.1 Autoencoder Learning Curve ...................................................... 25
4.2 Embedded Vector Heatmap ......................................................... 25
4.3 Correlation Matrix ..................................................................... 26
4.4 Multi-Layer Perceptron Learning Curve .................................... 27
4.5 Multi-Layer Perceptron Height Predictions of AOI ..................... 27
4.6 Multi-Layer Perceptron Height Predictions of AOI Heatmap ........ 28
4.7 Clustering Output ....................................................................... 28
4.8 Clustering Output Next to Sentinel-1 Output ............................... 29
ABSTRACT

REDUCING THE COMPUTATIONAL COST OF URBAN FLOOD PREDICTION IN LOS ANGELES

By

Laura Ani Bedoyan

Master of Science in Computer Science

Inundation maps have historically been computed using hydrologic models derived from the Navier-Stokes equations. For example, The Federal Emergency Management Agency (FEMA) uses such models to provide non-real time flood risk maps for the public. Real time flood inundation mapping using hydrologic techniques can be computationally expensive. With the rise of Machine Learning and Deep Learning models, faster approaches have been developed to optimize the real time flood mapping framework. Real time flood mapping has become increasingly complex to model using Data Science techniques due to the availability of large datasets that a model is able to learn from in a short amount of time. Hydrologic and Data Science flood mapping models can learn from a variety of variable inputs to predict if there is flooding in any resolution of inputs. In this project, the Autoencoder-Multi-Layer Perceptron-Clustering (AMC) model aimed to decrease the computational cost of predicting floods in the Downey/Bell Gardens by using unique features of urban areas as well as common features used in hydrologic models as input. An Autoencoder was trained (val MSE = .41, val Structural Similarity Index= 0.78) on the Downey/Bell Gardens area to create embedded vectors for each pixel of its 1m USGS DEM raster dataset. Input data into the model were static raster data (Slope, Aspect, Sentinel-1 SAR Vertical-Vertical (VV), Gravity Main Lines, Catch Basins, Social Sensitivity Index and Leveed Area). Four new stream gauge locations ran through the Autoencoder to generate embedded vector pixel values for each stream gauge location. Each embedded vector pixel value from each stream gauge was joined to its corresponding dynamic time series data – regional rainfall and stream gauge height. Each tuple formed from the join was combined into a dataframe to be used to train a Multi Layer Perceptron (MLP), with stream gauge height being the output vector. After training the MLP (val MSE= 0.37), the Downey/Bell Gardens area was sent through the MLP to return stream gauge height predictions for each pixel. Following this, each predicted height of the area was run through a K Means clustering algorithm to separate each height value into a flood or no flood group. The clustering algorithm was able to separate
height predictions into two groups with a Silhouette Index of 0.79 and the rate of an hourly flood inundation mapping for square meters per minute was 216,524,000 meters$^2$/minute. In conclusion, the Autoencoder-Multi-Layer Perceptron-Clustering (AMC) model successfully demonstrated its efficacy in optimizing real-time flood mapping for the Downey/Bell Gardens area. By leveraging unique urban features and conventional hydrologic inputs, the model reduced computational costs compared to other similar research. The integration of an Autoencoder for embedding, a Multi Layer Perceptron for prediction, and K Means clustering for classification yielded efficient hourly flood inundation maps, showcasing the potential of advanced Data Science techniques in addressing the challenges of real-time flood prediction in complex urban environments.
1.1 Background

Urban flooding creates some of the most extensive and costly damages to cities around the world [48]. Most flooding in urban cities has been caused by levees and dams failing as opposed to over-topping of river ways [40, 48]. Urban flood risk management is more complex than rural flood management because of the engineering that confines the risk [40, 48]. Dam failures can result in harmful amounts of water being released into these urban areas and cause extensive damage [40]. The severity of urban flooding in Los Angeles is greater in parts of the city with lower socioeconomic status, which include Black and Hispanic communities [40, 48].

Hydrologic models have a larger computational cost on high resolution data than on coarser low resolution data [23, 41]. Currently, FEMA and the US Army Corps of Engineers uses hydraulic and statistical numerical models, like HEC-RAS, to create static flood risk maps [7, 1]. Hourly dynamic flood inundation maps are not available for public use through FEMA [1]. Nevo, S et al. used LSTM to produce a real time operational framework for real time flood inundation mapping with a case study in India and Bangladesh [33]. Machine and Deep Learning flood mapping models can reduce the dimensionality of features without losing their complexity with Autoencoder algorithms and embedded vectors [21, 14, 16, 51, 30]. More complex algorithms and unique features such as urban specific input can reduce the computational cost of creating a deployable model accessible to the public [29, 18]. Some Data Science flood mapping models use time series data like stream gauge height and discharge, or weather predictions like humidity or rainfall, while others focus on raster or vector data like satellite or DEM datasets [49, 42]. Few combine satellite data with such dynamic time series data [33]. In order for these models to be deployed for public use by organizations like FEMA, they must have a feasible computational cost and prove accuracy [33, 13, 1].

Typical Data Science and hydrologic urban flood models do not utilize demographics as features for their models; they include hydrologic data (rainfall, stream gauge height and discharge), satellite imagery or weather forecasting features as input [50, 18]. Hydrologic models are confined by the inputs that solve Shallow Water Equations [41, 7]. Data Science allows for features not commonly used in hydrologic real time flood inundation mapping models to be incorporated in the training of models [40, 18, 23]. Examples of these features are levees, demographics, and city engineering data [27]. By incorporating these other features, a stronger predictive model could be created that could not be developed otherwise with traditional input [23, 51].

1.2 GIS Vocabulary

Raster data contain pixel specific data for a location of interest—each pixel contains geographical information as well as feature attribute data (See Figure 1.1) [17, 20, 22]. A raster image can have different bands of different information [22]. For example, a raster data set for the same area can have one band for water presence and another for slope [22]. Each pixel would represent different values of water presence throughout the entire area. For example, Sentinel-1 satellite data contain different bands of vertical or horizontal wavelengths.

Vector data is not pixel specific but could be a point, line or polygon [17, 20, 22]. It is vertex-specific where each vertex has a latitude or longitude and its geographical location is not constrained in a pixel. Both raster and vector data have feature attributes. For example, one vector feature point could have time series data for rainfall or a raster dataset for levee area could have attributes where each pixel represents the square feet [17, 22].
1.3 Stream gauges

A stream gauge is a device used to measure the height and flow rate of a body of water, such as a river or stream [7, 46]. The primary purpose of a stream gauge is to monitor and record the water level and discharge over time. Fig 1.2 illustrates how the stream gauge measures height and discharge[46].

1.4 Objective

The purpose of this research is to create a low cost urban aware Deep Learning flood prediction model in real time for the area of interest, Bell Gardens/ Downey Cities of Los Angeles County. Urban-aware static and dynamic data were used to train two Deep Learning models and one Machine Learning model. Static raster data used were 1m USGS DEM (Slope, Aspect), Sentinel-1 VV, Gravity Main Lines, Catch Basins, Social Sensitivity Index, and Leveed Area. Dynamic time series data used were stream gauge heights from stations LADPW F45B, USGS 11098000, USGS 11097000 and USGS 1109245 and NWS rainfall for each city at each stream gauge. An Autoencoder was used to reduce the computational cost of the AMC by creating embedded vectors and to reduce the dimensionality of each feature at each pixel for the area of interest (AOI). This study focused on the DEM raster data since its high resolution might enable a reduction of computational cost. A supervised Multi-Layer Perceptron and unsupervised K Means Clustering algorithm were used to predict the height of the water in each pixel and to decide whether a pixel was flooded or not. This could provide more information about the maximum height required for individual pixels to be considered a non-flooded area.
Typical urban flood models are either created by hydrologic, statistical, or Deep Learning approaches [33]. They each aim to predict stream gauge height and flood inundation models by increasing resolution of predictions at smaller time increments and spatial area [42, 13, 25, 32]. Hydrologic models combine statistics and physical models to simulate hydrologic scenarios using Navier-Stokes equations on smaller amounts of data, while Deep Learning models use a larger amount of data to create simulations [41, 7]. Sanders et al created a parallel raster inundation model which had a computational time of 2181 square meters per 15 seconds with a grid coarseness value of 50 [41]. They used a 10ft DTM and used flow pathways, subsurface drainage, precipitation variability and surface/subsurface interactions as inputs into their model to output flux through modified Shallow Water Equations. These Physics models are consistent with conservation of mass and momentum and do not require as much data as Deep Learning models [25, 28]. Physics models perform as the equation expects, but Deep Learning models may find patterns that were not known about before as there is a black box of the unknown [25, 28].

Shallow water equations (derived from Navier-Stokes) are used to solve for flux in a given area and are used to predict flood inundation [41, 19, 5, 38, 26]. There are benefits to using Deep Learning models for flood inundation. Less parallel computing is needed to compute the shallow water equations and so they can be deployed to a wider audience [41, 33, 25]. Many papers have tried to solve shallow water equations using neural networks (some with physics loss functions) and PCA [19, 5, 38, 26]. They are computationally expensive and determining how to discretize the solution domain can be challenging [5, 38, 26]. These Deep Learning models are limited to the available data and to the algorithm you use [25].

Stream gauge predictive models predict water levels at a specific stream gauge intake and estimate the discharge [40, 33]. This type of model can be created with Deep Learning, hydraulic or statistical modeling [7, 41, 33]. While the stream gauge modeling provides much needed information to predict water height and discharge, there needs to be more in depth research on flood extent and inundation models [33]. Most of the models created for stream gauge predictive modeling were traditional feed forward Deep Learning models like LSTM, GRU, ANN, and CNN [15, 49, 4, 47]. Typical inputs into these models were National Water Model forecasts, topography/DEM data and stream gauge observations. These stream gauge models proved to work best in rural settings but not urban settings [15, 49]. Having other forecasting models as an input into the models and developing a customized loss function could improve the quality of the models [33, 15, 28]. Some stream gauge models did not use physics-based regularizers or loss functions and validated their models with hydraulic models afterwards [15, 49, 4, 47, 13, 12, 32, 6]. Nevo, S et al used the Nash Sutcliffe coefficient to verify and validate their Deep Learning models [33]. Filtering out noise was essential in these models, and some researchers used Gaussian noise reduction to refine their models [32, 50].

Daw, A and Karpate, A developed their loss function by using conservation of energy and momentum and created an equation to find the minimum loss of the inconsistency errors in their model [12]. Duan Y et al, Nevo, S et al, and Brenowitz, N et al found that they could use mean square error or log likelihood as their loss function because the physics constraints excluded some of the non-linear relationships that the Deep Learning model created [33, 15, 32].

Deep Learning and hydraulic models could both be used to create flood inundation models [40, 41, 33]. Xiang, Z et al and Berkahn, S et al focused solely on stream gauge height and discharge prediction which could then be used as an input into flood extent or inundation models [49, 4]. Nevo, S et al used stream gauge data, precipitation forecast and measurements as well as DEM and historic flood inundation maps [33]. They used linear regression, LSTM and their own thresholding algorithm to generate flood inundation maps. They implemented their model in India as a case study for a 470,000 km2 area. Flood extent models predict the flux and momentum of the stream gauge water height in a given area, but not at one particular point [41, 33]. Kahl et al enhanced their hydraulic flood extent model by identifying levees in a dual grid approach before
they simulated different flood events in Los Angeles [27].

Farahmad Hamed et al created an Attention-based Spatial Temporal Graph Convolutional Network for urban flood nowcasting incorporating urban aware features like geolocated twitter posts and 311 calls [18]. They were able to create a graph Convolutional Neural Network (CNN) to connect each census tract’s static data, such as elevation and land use, together by counting each tract as a node. They verified their model with traffic data; when there was slow traffic then they assumed the road was flooded. A general overview of the time complexity for their algorithms are as followed, assuming a basic microprocessor being used:

Graph Matrix: $O(N \cdot E)$
Spatial Temporal blocks: $O(F \cdot N^2 \cdot \tau)$
Fully Connected Layers: $O(M + N)$

Overall complexity: $O(N \cdot E + F \cdot N^2 \cdot \tau + M + N)$, where $N=$ number of nodes, $E=$ number of edges, $F=$ number of features, $\tau =$ time steps, $M =$ output units.

Another approach to flood prediction was to take satellite imagery and use data to predict where there would be flooding [50, 24, 34]. They are computationally expensive and also need high quality real time data. These models do not use any stream gauge data or hydraulic models.

Yuhang Duan et al. created a BiGRU Autoencoder in order to predict Remaining Useful Life (RUL) [16]. Their model builds off of a traditional RNN Autoencoder except that they assign weights and skip connections at critical points in the encoding and decoding process (attention mechanism). The BiGRU is used to generate embedded vectors in this unsupervised manner. Before input into the BiGRU model, the original time sequence data is divided into fixed length time windows for data quality purposes. The BiGRU model transforms raw sensor readings into one dimensional Health Index Vectors after the encoder. These embedded vectors are then used on the milling dataset which improves their predictive performance. Similarity matching was used in order to validate the BiGRU. Yuhang Duan et al. would like to explore the Transformer structure to replace the RNN unit in their BiGRU.

Zhou, Jingya and Liu et al. reviewed the Network Representation Learning (NRL) which involves training neural networks to represent an information network of graphs as a collection of node embeddings in a latent space [51]. Unlike Yuhang Duan et al., the embeddings are nodes not single vectors [16, 51]. These are used as a network of embeddings (multi-dimensional). Zhou et al. discuss how there are many different encoding schemes to create embedded nodes [51]. An Autoencoder called DeepWalk uses an Autoencoder and node2vec to create embedded nodes. Similar to the windowing process used by Yuhang Duan et al, Zhou et al explain how network preprocessing methods were essential to breaking down the original network before creating embedded nodes [16, 51]. Node embedding is created by the output vectors of the encoder which represent hidden features that were learned by the objective function. The researchers explain how essential interpretability is for these models. Further research is needed on the meaning of the latent features learned from the network.

Embedded vectors are able to capture meaningful representations of nodes in a network or data in a dataset [16, 51]. By embedding nodes into a lower dimensional space, they can present information that was not otherwise shown in the original data. By creating embedded vectors for different bands of raster data using an Autoencoder, new features could be created. The backbone for all embedded vectors is similar, whether it is for a one or multidimensional dataset. The purpose of embedded vectors are to reduce the dimensionality of the dataset while preserving the information in the multidimensional space. Autoencoders can take in many different shapes of data, which is helpful especially for raster datasets that have multiple bands of data. Using an Autoencoder makes it easy to decide what dimension you want to reduce to. To enhance the Autoencoder, there is an option to include attention mechanisms which should be considered in future research.
Chapter 3
Methodology

3.1 Hardware

A 128GB GV100 Orico Portable NVMe SSD with 10Gbps was used to store and transfer large raster datasets into Google Earth Engine. A 1.6 GHz Dual-Core Intel Core i5 MacBook Air with 8 GB 2133 MHz LPDDR3 was used to connect to the portable SD card.

3.2 Google Colab

Google Colab is a free cloud-based platform provided by Google that allows you to write and execute Python code in a collaborative environment [39]. Google Colab offers access to free CPU and GPU runtimes. Both were used in this study to compare computational time. Below are the specifications for each runtime on Google Colab.

Intel Xeon CPU

- **Processor**: Intel(R) Xeon(R) CPU @ 2.20GHz
- **Vendor ID**: GenuineIntel
- **CPU Family**: 6
- **Model**: 79
- **Model Name**: Intel(R) Xeon(R) CPU @ 2.20GHz
- **Stepping**: 0
- **Microcode**: 0xffffffff
- **CPU Frequency**: 2199.998 MHz
- **Cache Size**: 56320 KB

NVIDIA Tesla P100 GPU

- **GPU Model**: T4
- **GPU Type**: NVIDIA Tesla P100
- **Number of CUDA Cores**: 3584
- **Memory Size per Board**: 12GB HBM2
- **Memory Interface**: 3072-bit
- **Memory Bandwidth (ECC off)**: 540 Gbytes/sec
- **Thermal Solution**: Passive
- **Max Power Consumption**: 250W
- **Form Factor**: 4.376”H x 10.5” L
- **Number of GPUs**: 1x GP100
3.3 Libraries

3.3.1 GDAL

The GDAL (Geospatial Data Abstraction Library) is an open-source library and set of tools used for reading, writing, and manipulating geospatial data [20]. GDAL provides a common interface and set of functions for working with various geospatial data formats, including raster and vector formats. It is widely used in the field of geospatial and remote sensing applications for data conversion, transformation, and analysis. In the AMC model, GDAL was used in pre-processing the .TIFF files from OpenTopography.

3.3.2 Google Earth Engine

Google Earth Engine is a cloud-based geospatial platform and library developed by Google [22]. It provides a vast repository of publicly available earth observation data, algorithms, tools via their custom JavaScript and Python based API. They can also be used for conducting geospatial analysis and remote sensing on a massive scale. Google Earth Engine has a student tier max limit of 250 GB of cloud storage. The total amount of storage used for the AMC model for all features (or assets) was 282.84 MB. Google Earth Engine Code Editor utilizes the Google Earth Engine Javascript API to load in specific assets and create unique visualizations and charts. Features can also be accessed and processed from Google Earth Engine via their Python API.

3.3.3 PyTorch, Numpy, Scikit Learn, Pandas, Keras

Pytorch is a Deep Learning library used to create neural networks like Autoencoders and other Sequential models [37]. In order to train a Pytorch model, you must input data of a specific size. For example, it could be 2D or 3D. Numpy is a library that can easily create such data geometries so that when it comes time to train the model, the data can be easily manipulated [35]. Numpy was used for data manipulation in PyTorch. Pandas is a similar library to Numpy but used for tabular data [36]. Most Machine Learning models can be created with Scikit Learn [43]. All data in Numpy and Pandas were subject to normalization and standardization before input into the models for training. Keras is a high-level neural networks API written in Python, that facilitates the development and training of Deep Learning models [39].

3.4 Data Sources

Data was retrieved from several different sources. OpenTopography had USGS 1m DEM raster data for students [44]. Los Angeles Department of Public Works provided stream gauge data for F45B and hourly rainfall at rain gauge AL383 [11]. Los Angeles County GIS Hub had publicly available data for Social Sensitivity Index [8]. County of Los Angeles, Department of Public Works had public access to their storm drain data [10]. The National Levee Database had public access to levee shape files [45].

3.5 Area of Interest

The area of interest tested by the AMC model was the Bell Gardens/Downey cities in Los Angeles County where the Los Angeles River meets the Rio Hondo River. The total area is $5.413 \times 10^7$ meters$^2$ (See Figure 3.1). The Dynamic data used was during the February 2023 Winter Storm [46, 31]. Additional stream gauges and rain gauges were used from USGS 11098000, USGS 11097000 and USGS 1109245 in order to diversify the training and validation data for the Multi-Layer Perceptron. All data processing was performed in the same fashion for those stream gauges for a much smaller area of the gauge’s location and will be detailed further on in this paper. All areas used EPSG 4326.

3.6 Feature Engineering

Features were downloaded from each of the sources above (see next sections for specifics) and stored onto the Orico SD card. Then each feature was uploaded to Google Earth Engine storage as individual assets. Afterwards, using Google Earth Engine’s Python API, all static vector shape files were converted into raster data types (Google Earth Engine Image object) [22]. A new combined Image object was created by adding
each raster asset as a single band. This streamlined the conversion of a singular Image object into a Numpy array, where each band was an additional depth of the array and made the Numpy array 3D. This approach made it simple to utilize Scikit Learn, PyTorch and Keras for creating customized models. Subsequent sections will delve into the intricacies of this process, providing detailed insights into the dynamic and static features utilized.

3.6.1 Slope and Aspect

The United States Geological Survey 3D Elevation Program 1 meter Digital Elevation Model (DEM) was used to create slope and aspect features [44]. GDAL Library was initially used to split the DEM raster into smaller pieces. Google Earth Engine Python API was able to generate slope and aspect with internal methods that took in the original DEM raster as input [44, 22]. For the slope raster, each pixel represented the steepness of the ground and aspect is the direction that the slope is facing (See Figure 3.2). Both slope and aspect were calculated as degrees where the more yellow the color in the map the greater the degree. The darker the color the lower the degrees of slope and aspect.

3.6.2 Sentinel-1 VV

Google Earth Engine Python API was used to analyze Sentinel-1 synthetic aperture radar (SAR) data to identify flooded areas [22]. The method loaded a collection of publicly available SAR images, selected the VV co-polarization, clipped the images to specific area of interest and smoothed the data to reduce noise. It then calculated the difference in radar intensities before (2023-02-10, 2023-04-01) and after (2023-04-01, 2023-05-01) to identify flooded regions using a threshold. Finally, it removed non-flooded water areas (e.g.,
oceans, lakes) using a global surface water dataset and updated the flood masks for the area of interest. The result was a mask indicating areas affected by floods in the specified region for the intended time granularity (See Figure 3.3) [22]. This data was processed into vector data type.

Using Google Earth Engine JavaScript API, the average slope and aspect were calculated at each flooded region (See Figure 3.4). The mean slope for each flooded region ranged from 2.0 and 0.1 degrees. The mean aspect for each flooded region ranged from 180 and 225 degrees.

3.6.3 Gravity Main Lines

The Los Angeles Storm Drain System Dataset from Los Angeles Department of Public Works GIS Hub consists of many different attributes of a storm drain system. One attribute of the system are gravity main lines which are underground pipes and channels (See Figure 3.5) [11]. Lateral lines connect catch basins to gravity main lines and water in gravity main lines flow into the open channels which eventually lead to the ocean. The diameter of the main line is the value used to display the feature. Most gravity main lines in the area of interest below have a diameter of 20 inches. This data was processed as vector data type.
3.6.4 Catch Basins

Another attribute from Los Angeles Storm Drain System Dataset from Los Angeles Department of Public Works GIS Hub is catch basins, which catch runoff from gutters (See Figure 3.6). The water in the catch basins go through to lateral lines and eventually to gravity main lines to the ocean [11]. The rotation of most catch basins in this study was about 350 degrees. This data was processed as vector data type.

3.6.5 Lateral Lines

The Los Angeles Storm Drain System Dataset from Los Angeles Department of Public Works GIS Hub consisted of many different attributes of a storm drain system. One attribute, lateral lines, connects catch basins to gravity main lines (See Figure 3.7) [11]. The diameter of the lateral line was the value used to display the feature. Most lateral lines in the area of interest below have a diameter of 20-25 inches. This data was processed as vector data type.
3.6.6 Social Sensitivity Index

Social Sensitivity Index (SSI) is a metric created by LA County Climate Vulnerability Assessment (See Figure 3.8). Data was retrieved from Los Angeles County. A low SSI value refers to an area where individuals in LA County are at an increased risk of being affected by climate-related hazards [8]. This data was processed as vector data type. The SSI values range from -3 to 5, where the lowest indicates a high risk to climate related disaster and the highest indicated a lower risk to climate related disaster.

The mean SSI (See Figure 3.9) varied significantly for each flooded region. The mean SSI value was as large as 3.5 in a some flooded region while was as low as -1.5 in other flooded regions.
3.6.7 Levees

Data was retrieved from the National Levee Database. Levees are elevated structures to decrease flood risk (See Figure 3.10) [45]. Square area was used to classify each polygon. The dark purple region below has no levee present because the square area of levee is zero. This data was processed as vector data type.

3.6.8 Height

Water height (inches) was measured over time in a channel at stream gauge F45B which is located in the area of interest Downey/Bell Gardens area (See Figure 3.11 for example data at Stream gauge F45B). Preliminary data was received from LADPW [10]. Stream gauge height data was also collected at USGS 11098000, USGS 11097000 and USGS 1109245 stations to diversify the data set from 2023-02-10 to 2023-05-01 [46]. Late February had the largest height value.
3.6.9 Rainfall

Rainfall amount (inches) was collected at nearby Stream gauge F45B, Rain gauge AL383 [10]. Rainfall at this gauge was generalized to be the same rainfall amount at Stream gauge F45B (See Figure 3.12). Hourly rainfall data was also collected from the National Weather Service for each of the corresponding USGS stations (USGS 11098000, USGS 11097000 and USGS 1109245) for their respective nearby Rain gauges (Tujunga, Pasadena, Sepulveda Basin) [31]. The greatest rainfall amount was in late March because it actually represents the amount of accumulated rain in the stream gauge at this point in time. The total accumulated rain fall for each area was used as a static vector feature.
3.6.10 Vector to Raster

Levee area, Social Sensitivity Index, Catch Basin, Gravity Main, and total accumulated rain for the region were all vector data types. They were all converted to Image or raster data types using the Google Earth Engine Python API. In this way, they could all be unique bands for an Image object so that all data could be converted into Numpy arrays. See next section.

3.6.11 Raster to Numpy

The sample rectangle function from Google Earth Engine Python API was used to create 9 2D Numpy arrays from Slope, Aspect, Levee, SSI, Lateral, Catch Basin, Gravity, Sentinel-1 VV and total accumulated rain for the region. Each 2D Numpy array had a size of 480 x 480 pixels after pre-processing increasing the scale of the input rasters to complete a successful conversion [22, 35, 37]. After stacking all 7 2D Numpy arrays the shape of the new Numpy array was (480, 480, 9).

3.6.12 Covariance Matrix

Covariance is a statistical measure that quantifies the degree to which two variables change together. The covariance between two variables, \(X\) and \(Y\), is calculated using the formula seen below:

\[
cov_{X,Y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N-1},
\]

where \(x\) and \(y\) are individual data points, \(\bar{x}\) and \(\bar{y}\) are the means of \(x\) and \(y\), and \(N\) is the number of data points.

In our case, \(N\) is the size of the raster feature image 480 x 480. A covariance matrix was created to understand how each feature was different from one another. A covariance value of zero means that the two variables do not vary together. This is optimal for feature relationships in a Deep Learning model. As seen in Fig 3.14, all selected features have low covariance values except with itself when it has a covariance value of one.

This covariance matrix was created for all features 0-8 respectively, for Slope, Aspect, Levee, SSI, Lateral Line, Catch Basin, Gravity Main Lines, Sentinel-1 VV and Total Rain.
3.6.13 Patches

After the Numpy conversion, 4x4 patches were created out of the (480, 480, 9) raster input so that batches could be created to train the Autoencoder [37]. The shape of the final input after patches were created was (14400, 9, 4, 4). This represents 14,400 patches with dimension 9x4x4 with 9 bands and 4x4 pixels for each band.

3.7 Architecture and Algorithms

3.7.1 Overview

Fig 3.15 shows the overall architecture of the modeling framework. Each part of the architecture will be described in each section below.
3.7.1.1 Principal Component Analysis

A Principal component analysis (PCA) model for dimensionality reduction of the raster data was initially used to create embeddings for the static raster data [43]. PCA works by projecting the original Numpy data onto the eigenvectors of the covariance matrix, defining new axes in the feature space [43]. To determine the appropriate number of components, plotting the Cumulative Explained Variance ratio will help identify the point where adding more components does not improve the new feature space. Since the Cumulative Explained Variance (CEV) started to bend at 7 components with a CEV of 1.0 we see that this algorithm did not do well at dimensionality reduction for the raster data (See Figure 3.16).

![Cumulative Explained Variance vs. Number of Components](image)

Figure 3.16: PCA

Since the PCA algorithm did not work as expected, an Autoencoder was created to perform dimensionality reduction and to create embedded vectors for the data set.

3.7.2 Autoencoder

Autoencoders can also serve as effective tools for dimensionality reduction by employing convolutional layers for image data [21, 30, 16, 51]. In this architecture, there exists both an encoder and a decoder, both structured as Neural Networks with Conv2D layers and Conv2DTranspose layers, respectively [37, 16]. The Conv2D layers in the encoder must ensure that the input dimensions match the number of data features, while the output shape of the Conv2DTranspose layers in the decoder must align with the input shape of the Conv2D layers. Likewise, the output shape of the Conv2D layers in the encoder should match the input shape of the Conv2DTranspose layers in the decoder. The output of the encoder is the new feature called Embedded Vector (See Figure 3.17). This model is an unsupervised model because it generates new feature from the encoder. The model learns from the training data and see how well the decoder piece reconstructs the input image.

![Autoencoder Model](image)

Figure 3.17: Autoencoder Model
After the Autoencoder was trained, the output of only the encoder was used to reduce the dimensionality of the features down to one. In this way, the output of the encoder created a new feature from all of the 9 input static features into one to be used later in the time series model, and the data set will not be as large. Fig 3.18 shows the design of the completed Autoencoder and how the output of the encoder is used for the time series portion of the modeling framework.

The encoder consists of input of (9, 4, 4) with a 3x3 kernel with a stride =1. The next layer of encoder convolutions takes an input of (4, 4, 4) also with a 3 x 3 kernel and stride =1. The final layer of the encoder outputs a size of (1, 4, 4) before it goes into the decoder. The decoder takes the same shape as the encoder. All layers ensured that the data was normalized with Batch2D and also used the ReLU activation function in hidden layers after each layer. The basic time complexity of the algorithm is as follows:

\[
O(N \cdot K \cdot K \cdot D) + O(N \cdot K \cdot K \cdot D)
\]

where:

- \( N \) is the number of pixels,
- \( K \) is the size of the convolutional kernel, and
- \( D \) is the depth (number of channels) of the input.

The next sections focus on the specifics of the components of the Autoencoder. Batch size of 200, 1000 epochs and learning rate of .00001 was used. All data was shuffled and a random state of 42 was used.

3.7.2.1 Conv2d

Torch.nn.Conv2d applied a 2D convolution over a input signal [37]. The output value at position \((i, j)\) can be precisely described as:

\[
\text{out}(i, j) = \text{bias}(j) + \sum_{k=0}^{C_m-1} \text{weight}(j, k) \ast \text{input}(i, k)
\]

Where:

- \( \text{out}(i, j) \) is the output at position \((i, j)\).
- \( \text{bias}(j) \) is the bias for the \(j\)-th output channel.
- \( \text{weight}(j, k) \) is the filter weight for the \(j\)-th output channel and \(k\)-th input channel.
- \( \text{input}(i, k) \) is the input value at position \((i, k)\).
- \( C_m \) is the number of input channels.

The parameters kernel_size, stride, padding, dilation and groups control the behavior of the convolution operation. In the AMC model, the kernel was 3 and stride, padding and dilation were 1.
The input and output shapes can be expressed as follows:

\[
\text{Input: } (N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}}) \text{ or } (C_{\text{in}}, H_{\text{in}}, W_{\text{in}}) \\
\text{Output: } (N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}}) \text{ or } (C_{\text{out}}, H_{\text{out}}, W_{\text{out}})
\]

Where:

\[
H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor
\]

\[
W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor
\]

These equations describe how the input and output shapes are determined based on the specified parameters. The specifics of the encoder are as follows:

```python
encoder = nn.Sequential(
    (0) : Conv2d(9, 4, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1)),
    (1) : ReLU(),
    (2) : BatchNorm2d(4, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True),
    (3) : Conv2d(4, 1, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1)),
    (4) : ReLU(),
    (5) : BatchNorm2d(1, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)
)
```

### 3.7.2.2 ConvTranspose2d

Torch.nn.ConvTranspose2d applies a 2D transposed convolution, often seen as the gradient of Conv2d with respect to its input [37]. The output at position \((i, j)\) is calculated as:

\[
\text{out}(i, j) = \text{bias}(j) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(k, j) \ast \text{input}(i, k)
\]

Where:

- \text{out}(i, j) is the output at position \((i, j)\).
- \text{bias}(j) is the bias for the \(j\)-th output channel.
- \text{weight}(k, j) is the filter weight for the \(k\)-th input channel and \(j\)-th output channel.
- \text{input}(i, k) is the input value at position \((i, k)\).
- \(C_{\text{in}}\) is the number of input channels.

The parameters kernel\_size, stride, padding, output\_padding, dilation, and groups control the behavior of the transposed convolution operation.

The input and output shapes can be expressed as follows (See Equation 3.1):

\[
\text{Input: } (N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}}) \text{ or } (C_{\text{in}}, H_{\text{in}}, W_{\text{in}}) \\
\text{Output: } (N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}}) \text{ or } (C_{\text{out}}, H_{\text{out}}, W_{\text{out}})
\]
\[ H_{out} = (H_{in} - 1) \cdot \text{stride}[0] - 2 \cdot \text{padding}[0] + \text{dilation}[0] \cdot (\text{kernel\_size}[0] - 1) \]
\[ + \text{output\_padding}[0] + 1 \]
\[ W_{out} = (W_{in} - 1) \cdot \text{stride}[1] - 2 \cdot \text{padding}[1] + \text{dilation}[1] \cdot (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1 \] (3.1)

The parameters \text{kernel\_size}, \text{stride}, \text{padding}, \text{output\_padding}, \text{dilation}, and \text{groups} the behavior of the convolution operation. In the AMC model, the kernel was 3 and stride, padding and dilation were 1. The specifics of the decoder are as follows.

decoder = nn.Sequential(
    (0) : nn.ConvTranspose2d(1, 4, \text{kernel\_size} = (3, 3), \text{stride} = (1, 1), \text{padding} = (1, 1)),
    (1) : nn.ReLU(),
    (2) : nn.BatchNorm2d(4, \text{eps} = 1e-05, \text{momentum} = 0.1, \text{affine} = \text{True}, \text{track\_running\_stats} = \text{True}),
    (3) : nn.ConvTranspose2d(4, 9, \text{kernel\_size} = (3, 3), \text{stride} = (1, 1), \text{padding} = (1, 1)),
    (4) : nn.ReLU(),
    (5) : nn.BatchNorm2d(9, \text{eps} = 1e-05, \text{momentum} = 0.1, \text{affine} = \text{True}, \text{track\_running\_stats} = \text{True})
)

3.7.2.3 ReLU

The Rectified Linear Unit (ReLU) is an activation function commonly used in neural networks [37, 30]. Its function is simple; it outputs the input value if it is positive, and zero otherwise.

\[ f(x) = \max(0, x) \]

When a positive input is passed through ReLU, it remains unchanged, but if the input is negative, it is replaced with zero. The key advantage of ReLU is its simplicity and efficiency in training deep neural networks. It helps introduce non-linearity to the model, enabling the network to learn complex patterns and representations.

3.7.2.4 BatchNorm2d

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with an additional channel dimension):

\[ y = \frac{\gamma \cdot (x - E[x])}{\sqrt{\text{Var}[x] + \epsilon}} + \beta \]

The mean and standard deviation are calculated per-dimension over the mini-batches, and \( \gamma \) and \( \beta \) are parameter vectors the size of the input [37]. By default, the elements of \( \gamma \) are set to 1, and the elements of \( \beta \) are set to 0. The input and output shape are the same.

3.7.2.5 Validation Metric- Mean Square Error

Mean Square Error computes the average of the squared differences between the actual and predicted values [16, 30]. In the Autoencoder, the input and output image pixel values were compared to one another to output total MSE per batch and epoch. With regression algorithms, using a metric for loss function like Mean Square Error is necessary as opposed to accuracy which is for classification algorithms.

\[ \text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \]
3.7.2.6 Validation Metric- Structural Similarity Index

The Structural Similarity Index (SSI) is a metric used to quantify the similarity between two images [43]. It assesses both luminance and contrast information as well as structural patterns, providing a more comprehensive measure of similarity than traditional pixel-based methods. SSI ranges from -1 to 1, with 1 indicating identical images. Higher values indicate greater similarity, making it a valuable tool in image quality assessment and comparison.

3.7.3 Multi-Layer Perceptron

3.7.3.1 Data Input

Small polygons approximately 5 square meters each, were generated at the positions corresponding to the four stream gauges: LADPW-F45B, USGS 11098000, USGS 11097000 and USGS 1109245 [44, 46]. Each of the static features (Slope, Aspect, Levee Area etc.) were intersected with each polygon (See Figure 3.19).

![Figure 3.19: Multi-Layer Perceptron Overview](image)

All of these four areas were run through the Autoencoder to create embedded vector values for each pixel in the stream gauge polygon areas. Data was flattened after running through the Autoencoder so that it could run in the Multi Layer Perceptron (See Figure 3.20).

<table>
<thead>
<tr>
<th>Embedded Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

![Figure 3.20: Embedded Vector Dataframe Flattened](image)
Each of the embedded vector values at one area were joined to its corresponding stream gauge and rain
gauge data. So, after each of the four locations got their embedded vector outputs, each value was joined
to their corresponding dynamic data. Afterwards, all data was joined together to create a large dataset (See
Figure 3.21) and data was flattened.

![Figure 3.21: Multi-Layer Perceptron Data Frame Input](image)

The Multi-Layer Perceptron algorithm is a supervised regressor algorithm where the input vector are
Month, Day, Hour and Rainfall, Embedded Vector (EV) Value \([30]\). The output vector is Height. The
statistics for the input is as follows (See Figure 3.22).

![Figure 3.22: Embedded Vector Dataframe Statistics](image)

Ideally the output of this algorithm could predict the height of the water in each pixel.
The overall basic time complexity for this algorithm is as follows:

\[ O(L \cdot N \cdot (\sum_{i=1}^{n} M_i)) \]

where,

- \( N \) is the number of input features,
- \( L \) is the number of layers,
- \( M_i \) is the number of neurons in layer \( i \) for \( i = 1, \ldots, n \).

In this case, \( N=5 \), \( L=2 \), \( M_1=11 \), \( M_2=7 \) (See Figure 3.23 and 3.24).

---

**Model: "sequential_6"**

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_18 (Dense)</td>
<td>(None, 11)</td>
<td>66</td>
</tr>
<tr>
<td>activation_12 (Activation)</td>
<td>(None, 11)</td>
<td>0</td>
</tr>
<tr>
<td>dropout_6 (Dropout)</td>
<td>(None, 11)</td>
<td>0</td>
</tr>
<tr>
<td>dense_19 (Dense)</td>
<td>(None, 7)</td>
<td>84</td>
</tr>
<tr>
<td>activation_13 (Activation)</td>
<td>(None, 7)</td>
<td>0</td>
</tr>
<tr>
<td>dense_20 (Dense)</td>
<td>(None, 1)</td>
<td>8</td>
</tr>
</tbody>
</table>

Total params: 158 (632.00 Byte)
Trainable params: 158 (632.00 Byte)
Non-trainable params: 0 (0.00 Byte)

---

Figure 3.23: Multi-Layer Perceptron Detailed Structure

Batch size was 1350, there were 70 epochs and the learning curve was 0.001, Adam optimizer was used. In general, the output of a hidden layer can be expressed as:

\[ h = \text{ReLU}(W_1x + b_1) \]

where \( W_1 \) is the weight matrix connecting the input to the hidden layer, and \( b_1 \) is the bias vector for the hidden layer [9].

The output layer is a linear combination of the hidden layer activations:

\[ \hat{y} = W_2h + b_2 \]

where \( W_2 \) is the weight matrix connecting the hidden layer to the output layer, and \( b_2 \) is the bias vector for the output layer.

The network parameters, including the weights and biases, are updated during training using optimization algorithms like Adam optimizer to minimize the MSE loss.
Since this is a regressor model, metrics like MSE were used to better understand how well the model was performing.

### 3.7.3.2 Validation Metric- Mean Square Error

The loss function for regression can be defined as the mean squared error (MSE):

\[
\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2
\]

where \( n \) is the number of training samples, \( \hat{y}_i \) is the predicted output for sample \( i \), and \( y_i \) is the true output for sample \( i \).

### 3.7.4 K-Means Clustering

After training the Multi Layer Perceptron on the 4 small areas around the stream gauges, the original Area of Interest (Downey/Bell Gardens) which was output from the Autoencoder was sent through the trained Multi Layer Perceptron. Then its output while still flattened was sent through the clustering algorithm an then reshaped back into the 2D structure.

The goal of using K-Means Clustering was to use the output of the Multi Layer Perceptron to group pixel height predictions into either a flood or no flooding group (See Figure 3.25) [43]. The algorithm works as follows:

A dataset \( X \) is divided into \( k \) clusters by iteratively updating cluster centroids and assigning data points to the nearest centroid.

1. **Initialization**: Randomly select \( k \) initial cluster centroids. In our case 2 centroids were selected.
2. **Assignment**: Assign each data point to the nearest centroid using the Euclidean distance:

\[
\epsilon^{(i)} = \arg \min_j \| x_i - c_j \|^2
\]
where \( c^{(i)} \) is the centroid index assigned to data point \( x_i \).

3. **Update Centroids:** Update each centroid \( c_j \) by taking the mean of the data points assigned to it:

\[
c_j = \frac{1}{|S_j|} \sum_{i \in S_j} x_i
\]

where \( S_j \) is the set of data points assigned to cluster \( j \).

4. **Repeat:** Repeat steps 2 and 3 until convergence.

The basic time complexity for this algorithm is:

\[ O(N \cdot K \cdot D) \]

where \( N \) is the number of tuples, \( K \) is the number of clusters, and \( D \) is the number of features. In our case the number of tuples are 230,400, \( K \) is 2 and \( D \) is 1.

![Figure 3.25: K Means Clustering Overview](image)

### 3.7.4.1 Validation Metric

The silhouette index is a metric used to assess the quality of clustering in unsupervised learning [43]. It measures the similarity of an object to its own cluster compared to other clusters, providing a value between -1 and 1.

The silhouette index for an individual data point \( i \) is calculated as follows:

1. Compute the average distance \( (a_i) \) from the data point \( i \) to all other data points in the same cluster.
2. For each cluster other than the one to which \( i \) belongs, calculate the average distance \( (b_i) \) from \( i \) to all data points in that cluster. Choose the cluster with the smallest \( b_i \).
3. The silhouette index \( (s_i) \) for data point \( i \) is given by:

\[
s_i = \frac{b_i - a_i}{\max\{a_i, b_i\}}
\]

The overall silhouette index for the entire clustering is the average of the silhouette values for all data points. For a clustering with \( n \) data points, it is often denoted as \( S \):

\[
S = \frac{1}{n} \sum_{i=1}^{n} s_i
\]

A higher silhouette index indicates better-defined clusters. This metric was used to validate how well the clustering algorithm worked to separate the height dataset into flood or no flood groups. In this way a
threshold value for each pixel could be determined. If a pixel water height surpassed the threshold defined by the cluster, then there would be flooding.
Chapter 4

Results

4.0.1 Autoencoder

The Autoencoder performed with a relatively low loss. After training the Autoencoder on a small subset of the original area of interest, the whole area of interest was sent through the trained encoder piece to generate a new feature of embedded vectors for each pixel.

The model did not overfit because the validation curve as seen in (See Figure 4.1) is greater than the training curve. The final training loss was 0.35 and the final validation loss was 0.41. The final training Structural Similarity Index was 0.82 and the the final validation structural similarity index was 0.78.

![Figure 4.1: Autoencoder Learning Curve](image1)

The output of the encoder (See Figure 4.2) created a new feature that represented all input features combined into one.

![Figure 4.2: Embedded Vector Heatmap](image2)
The correlation is similar to the covariance matrix except it shows how similar two features are to each other instead of measuring the extent to which two features vary together. As seen in Figure 4.3 all features have a low correlation value except for rainfall and height and for the features and itself. Data was flattened before correlation was calculated.

Correlation is a statistical measure that quantifies the strength and direction of a linear relationship between two numeric variables [30]. The correlation coefficient, often denoted by \( r \), ranges from -1 to 1 where -1 is a strong negative correlation and 1 is a strong positive correlation:

\[
 r = \frac{\sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n} (X_i - \bar{X})^2 \sum_{i=1}^{n} (Y_i - \bar{Y})^2}}
\]

Where:

- \( X_i \) : Individual data points for variable \( X \)
- \( Y_i \) : Individual data points for variable \( Y \)
- \( \bar{X} \) : Mean of variable \( X \)
- \( \bar{Y} \) : Mean of variable \( Y \)

It is often desirable to use features (variables) that are not highly correlated with each other [43, 30]. This is because high correlation between features can lead to multicollinearity, which can cause issues in certain statistical models. Even though Height and Rainfall had higher correlation values than that of the other features, the two features were retained in the model training.

\[\text{Figure 4.3: Correlation Matrix}\]

### 4.0.2 Multi-Layer Perceptron

The learning curve for the Multi Layer Perceptron showed training loss of: 0.3404 and a validation loss of 0.3422 (See Figure)

When the Area of Interest was run through the model with each embedded vector value joined with a rainfall value of 20in, Month= February, Day=25 and Hour 7, the height predictions for that model had a
Mean of 5.29 inches, Median of 5.30 inches, Standard Deviation of 0.0848, Min of -0.329 inches and Max of 5.392 inches (See Figure 4.5 where value is height of water in each pixel in inches).

Figure 4.4: Multi-Layer Perceptron Learning Curve

Figure 4.5: Multi-Layer Perceptron Height Predictions of AOI
The flattened data of height predictions were reshaped back into the 480x480 shape. Figure 4.6 is a heatmap of height predictions for the area of interest in the reshaped form. Areas that the Sentinel-1 data picked up as flooded tended to have lower height predictions than other areas for the AOI. The area where there is no levee had the highest height prediction values.

![Heatmap](image)

**Figure 4.6: Multi-Layer Perceptron Height Predictions of AOI Heatmap**

### 4.0.3 K-Means Clustering

The K-Means clustering algorithm had a silhouette score of 0.79. The clusters had a distinct separation at a height value of about 5.2 inches (See Figure 4.7).

![Clustering Output](image)

**Figure 4.7: Clustering Output**
The final output of the KMeans cluster on a heat map (See Figure 4.8) adds new features to the flooded areas. Cluster 0, non flooded, tended to have larger heights than cluster 1 which was flooded.

Figure 4.8: Clustering Output Next to Sentinel-1 Output

4.0.4 Time Complexity Analysis

For the AMC model, the basic time complexity is as follows. It only takes into account test data and not training (no epochs or batch sizes were taken into account):

\[ O(N \cdot K \cdot K \cdot D) + O(N \cdot K \cdot K \cdot D) + O(L \cdot N \cdot \sum_{i=1}^{n} M_i) + O(N \cdot K \cdot D) \]

\[ = O(230,400 \cdot 3 \cdot 3 \cdot 9) + O(230,400 \cdot 3 \cdot 3 \cdot 9) + O(5 \cdot 2 \cdot 11 \cdot 7) + O(230,400 \cdot 2 \cdot 1) \]

\[ = O(18,662,400) \]

Recall that Farahmand, H. et al created a spatial-temporal graph deep learning model. It’s overall time complexity is below, if run on AMC’s data. This is assuming that each pixel is connected to each and every other pixel. [18]:

\[ O(N \cdot E) + O(F \cdot N^2 \cdot \tau) + O(M + N) \]

\[ = O(230,400 \cdot \frac{(230,400 \cdot 230,399)}{2}) + O(9 \cdot 480^2 \cdot C) + O(11 + 7) \]

\[ = O(230,400 \cdot 26,574,336,000) \]
4.0.5 Computational Cost Analysis

Sanders et al. developed a cutting-edge flood inundation model, referred to as the PRIMo model [41]. This model showcases the rate of producing flood inundation models for meters$^2$/min [41]. The PRIMo model demonstrated a flood inundation model production rate of 8,724 meters$^2$/min for a consistent square area. In comparison, the AMC model exhibited a computational rate of 216,524,000 meters$^2$/min [41].

When run on the Google Colab CPU the time to test the same AOI on the AMC model was 15 seconds and was 13 seconds on the GPU.
Chapter 5

Conclusion and Future Work

The Autoencoder was able to reduce the dimensionality of the static raster data and create embedded vectors that represented each feature in each pixel. The Autoencoder which produced this data did not overfit, however, the validation training loss was unstable. Adding regularizers to help solve that made the model overfit. In order to mitigate overfitting overall the number of neurons and hidden layers were adjusted as well as the batch size and number of epochs. The Autoencoder took less than 1 second to output the embedded vector values for each pixel of the test data. This could be a good way to reduce high dimensional raster data to be less computationally expensive in other models. If the Autoencoder was trained on a wider dataset, perhaps in different cities and incorporated historic flood maps it could improve.

The Multi-Layer Perceptron result when compared with the Sentinel-1 total flooded area for the whole rain event, showed that predicted heights were lower in flooded areas but greater in regions without levees. It would be essential to validate the AMC model with a traditional Physics based model like HEC-RAS to compare accuracy. It was difficult to prove accuracy of this model because there were no historic flood inundation maps to compare them to. This is why physics loss functions are used for model validation and perhaps these could also be utilized in the AMC model in the future. Stream gauge discharge was not used in this model because the Multi-Layer Perceptron could only predict one dimension. It would be interesting to try and predict a different metric in the future. The Multi-Layer Perceptron took the longest to compute with a time of 13 seconds for the AOI. The dataset was the largest for this since it joined all embedded vector values for the AOI with corresponding time series data together.

A high silhouette index indicates the success of the K-Means Clustering algorithm, leading to the formation of two distinct groups representing flood and no flood. Notably, the flooded clusters exhibited lower heights than the non-flooded ones, and the elevated silhouette index revealed a threshold value effectively separating height values. This suggests a potential method for establishing a threshold value for each pixel, allowing instant classification of flooded or non-flooded status without relying on the clustering algorithm. However, validation against Physics models or the use of a loss function is crucial to assess the model’s reliability.

The computational efficiency of the AMC model surpassed that of the PRiMo model, highlighting a significant improvement in processing speed [41]. While the enhanced computational speed is a notable advantage, the feasibility of deployment depends on several factors. These factors may include the scalability of the model to handle real-world datasets, the availability of computational resources, and the adaptability of the model to diverse and dynamic environments.

In future work it would be interesting to verify this model with a similar algorithm like Farahmand, H. et al provided or more simply with a classic network flow algorithm [18]. Connecting every pixel and comprehending the existing condition of storm drains within the city to generate a precise graph poses a significant challenge. It is imperative, once more, to validate this model either through a high-resolution model or by referencing historical flood maps.
References


