Adam Hunter

Firenet Capstone Project Report

2.21.24

**Introduction**

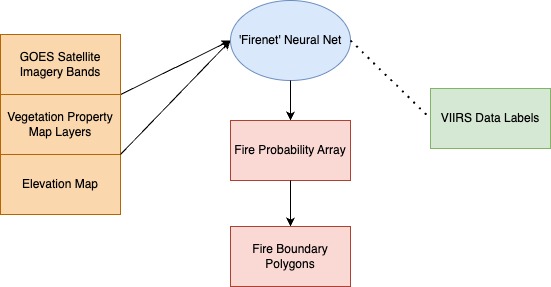
The climate of the Earth in the last century is marked by a rising frequency of extreme weather events. While long-term prevention strategies are needed to address climate change, the immediate and unpredictable nature of these extremes necessitates short-term, reactive measures. Consequently, there is a pressing need to enhance quick decision-making processes in the emergency response to any climate-related disaster.

In recent times, the occurrence and intensity of *large* wildfires in particular have escalated, making up the bulk of the area burned each year. Responding to these fast-moving and devastating fires demands accurate data regarding the location of the fire's edge, its speed, and its likely path of spread. Gathering this crucial information poses significant logistical challenges and risks, as current methods to obtain data on the fire's status involve either aerial surveys or missions on the ground. This results in a critical shortage of near-real-time data on specific wildfire incidents, increasing the difficulty of predicting fire behavior and adding to the hazardous nature of managing wildfires.

Satellite remote sensing offers critical opportunities to collect intelligence on fire behavior, yet it falls short in providing the necessary spatial or temporal resolution for precise and timely classifications of wildfire boundaries. For instance, geostationary satellites like the Geostationary Operational Environmental Satellite (GOES) offer high-frequency observations of the Earth (every 5 minutes) but suffer from limited spatial resolution due to their great distance from the Earth's surface. Conversely, low Earth orbit (LEO) satellites, such as the carriers of the Visible Infrared Imaging Radiometer Suite (VIIRS) or Moderate Resolution Imaging Spectroradiometer (MODIS), deliver high spatial resolution but cannot provide continuous coverage because they orbit the Earth causing their coverage to have large gaps in time for any given location.

An article titled “A Deep Learning Approach to Downscale Geostationary Satellite Imagery for Decision Support in High Impact Wildfires” published to the journal *Forests* in 2021 laid the foundation for a deep learning model that could combine the temporal coverage of GOES with the spatial resolution of VIIRS. Authors of this paper had developed a prototype model that would use the high-resolution, accurate fire classifications from VIIRS fires as training data, with GOES data as input.

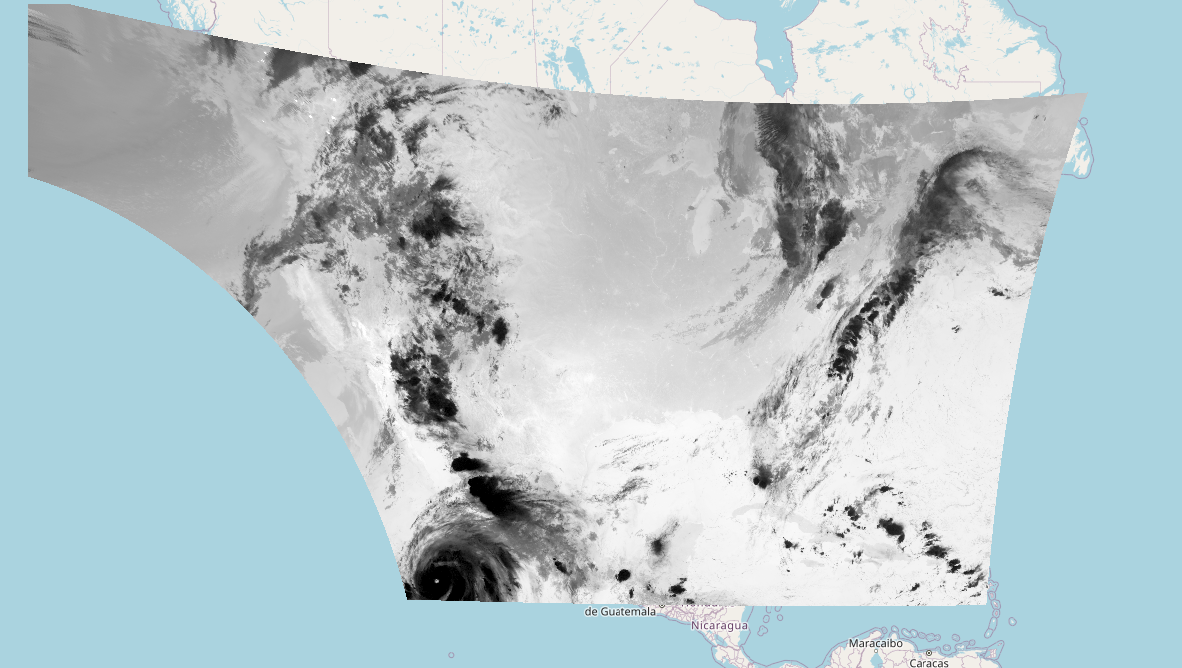
Essentially a neural net could be trained to recognize the LEO fire labels from GOES imagery. This model could potentially be as accurate as VIIRS and as timely as GOES. Figure 1 is a diagram displaying the inputs and outputs of the neural net.



**Figure 1:** Orange boxes are model input, green box is training data labels, and red boxes are output.

A friend of mine spent time working on a grant to recreate this model and his code formed the basis for my capstone. The bulk of my capstone work involved refactoring his code to make it more modular, streamlined, maintainable, and automated. My other task was to deploy it to the cloud so that some of the long-running processes and scheduled prediction outputs could be displayed automatically and consistently. I accomplished these two tasks, streamlining the process of model retraining, which will be the future direction necessary for Firenet to move in before it can produce real value. A fully optimized ‘Firenet’ could be useful for tracking large, fast-growing fires. It could trace their boundaries, calculate their speed of movement, and potentially even predict their path of spread.

**Neural Net Data Sources**

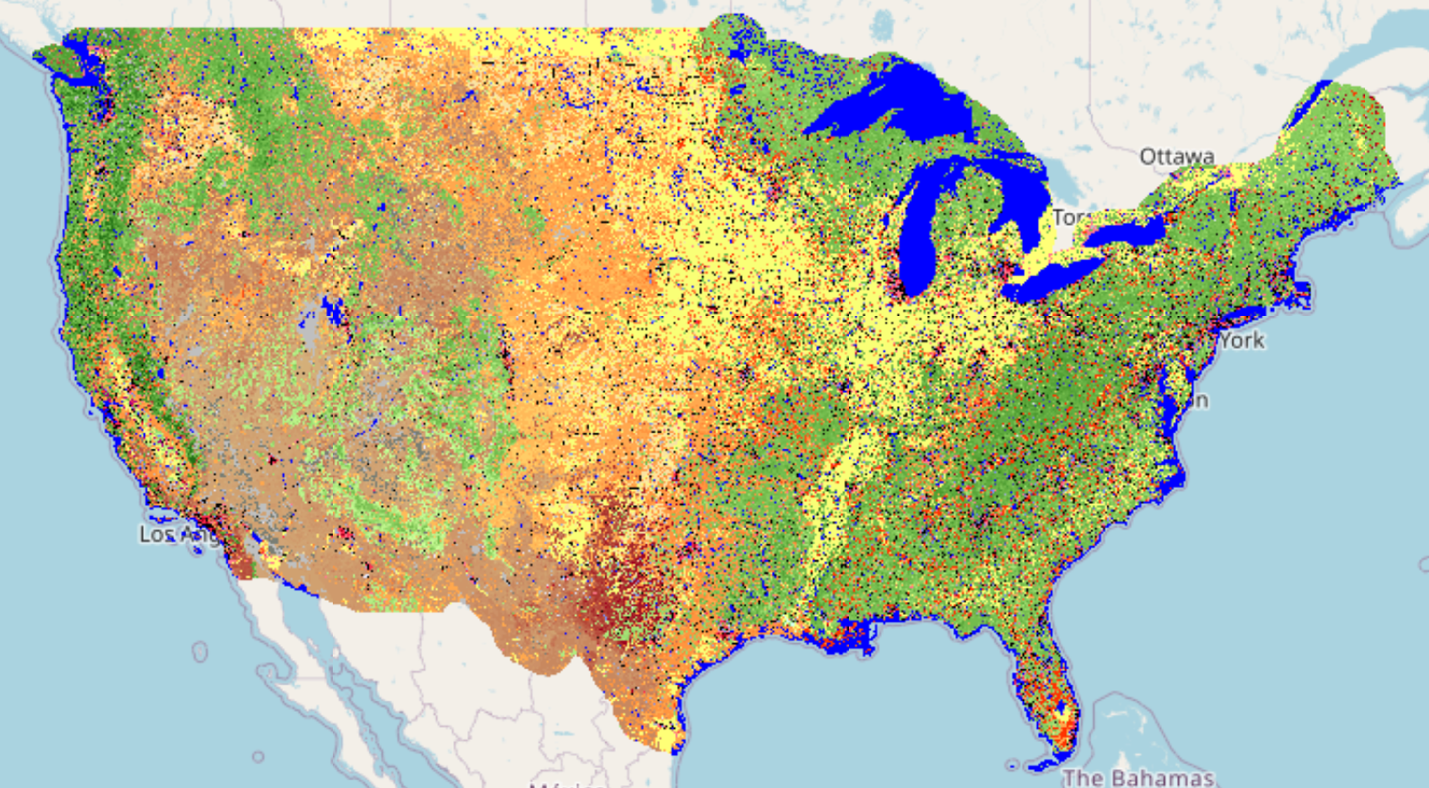
The data used in this project comes from three sources, these were introduced above but I will describe each in detail here. GOES imagery from the Advanced Baseline Imager Level 2 Cloud and Moisture Imagery (MCMIPC) product is the primary input. This is weather satellite imagery, and it is composed of 16 spectral bands. These bands correspond to light wavelength ranges visible and invisible. The most useful ones for fire detection are the infrared bands, especially band 7 shown below in figure 2. 

**Figure 2:** This is an image of the 7th band of the MCMIPC product from GOES-16 which captures the continental US. Band 7 with its short wavelength detection is highly sensitive to the temperature of a pixel [band7]. A few fires are just barely visible as white flecks in Western Canada. This is a scan from August 17, 2023.

The resolution of GOES imagery is 2km per pixel, in other words each pixel represents a square 2 km wide and 2 km high[[1]](#footnote-1). This is very low resolution which is what makes using this imagery for fire detection challenging. To help alleviate this, firenet input uses a median composite of the last hour’s worth of imagery – 12 separate images, one every 5 minutes. This has the benefit of reducing false positive fire classifications, as sunlight reflectance may produce incorrect classifications in single scans but is unlikely to have a consistent signature over the course of an hour. This median composite of the continental US across 16 bands of imagery is the first data source for the model input.

The second data source is a set of map layers that are updated yearly and contribute information to the model about a given area’s potential to burn. These map layers are produced by LANDFIRE (LF) a shared program between the U.S. Department of Agriculture Forest Service and the U.S. Department of the Interior[landfire about]. The map layers are produced by a combination of field surveys and remote sensing. Map layers used in the Firenet neural net input are the following:

* DEM layer: This layer represents a Digital Elevation Model (DEM). Elevation is crucial for understanding the topography of an area, which can influence the distribution and behavior of vegetation and fires.
* CBD layer: This layer indicates Canopy Bulk Density (CBD), which is a measure of the mass of an area's vegetation per unit area. It is used to assess the potential for biomass accumulation in the canopy, which can affect the behavior of fires.
* EVC layer: This layer shows Existing Vegetation Cover (EVC), which is a measure of the percentage of the landscape that is covered by vegetation. It is an important factor in assessing the fuel load that can contribute to wildfires.
* EVH layer: This layer represents Existing Vegetation Height (EVH), which is the average height of vegetation in a pixel. It is used to estimate the potential fuel load and can influence fire behavior.
* FBFM40 layer: This layer represents a standardized method for predicting fire behavior. It uses various fuel characteristics to estimate the burn potential of a given pixel.
* FVH layer: This layer represents Fuel Vegetation Height (FVH), which is a measure of the height of vegetation that serves as fuel for fires. This information is critical for predicting and managing the behavior of wildfires. [landfire product descs]



**Figure 3:** This image shows the fuel vegetation height map layer for 2022, each pixel has a vegetation height associated with it. [landfire viewer]

The final data source is fire classifications from VIIRS. VIIRS is a LEO satellite that captures infrared data and uses a decision tree to classify active fires[viirs decision tree]. These fire classifications can be downloaded as tabular data where fire pixels are rows and confidence level, latitude, longitude and time of viewing are columns. The training data preprocessing function seeks clusters of these points that are near each other in both space and time to use as inputs, these clusters represent fires. An example row of data is shown below in table 1.

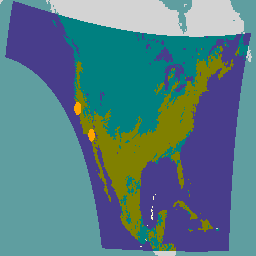
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Latitude | Longitude | Acq\_date | Acq\_time | Confidence |
| 36.354759 | 76.804565 | 1/1/20 | 724 | high |

**Table 1:** Example row of VIIRS data, columns not used in training data are removed.

**Prediction Comparison Data Sources**

The Firenet project’s [map](https://firenet-99.uw.r.appspot.com/) includes the fire classifications from the VIIRS satellite as described above. These classifications are useful to compare with Firenet outputs because they are based on higher spatial resolution imagery and should be more precise than even a perfectly trained Firenet model. It is important to recognize however that using VIIRS alone for comparison would be a mistake. This is because VIIRS does not have perfect temporal coverage, and fires that stop and start while it is orbiting a different region of the earth will be missed by it entirely. As such, a well-trained Firenet would classify some fires correctly that according to VIIRS would be false positives. For this reason, a fire classification map with higher temporal coverage would be helpful to also include for comparison. Fortunately, the GOES satellite itself provides such a map.

The GOES satellite provides a fire classification map in its suite of near-real-time outputs. An example visualization of this classification map is displayed in figure 4.



**Figure 4:** This is an example image from the FDCC imagery product. Each pixel is categorized according to a series of variables including the estimated temperature, and a variety of error measurements related to cloud conditions, sunlight reflection angle relative to the satellite etc.

The GOES FDCC product uses a decision tree with bands 2, 7, 14, and 15 as inputs. The accuracy of this product itself is hard to measure [hotspot reference], but with its temporal coverage combined with VIIRS spatial resolution, a good comparison for Firenet can be developed. If Firenet can successfully combine the spatial resolution of VIIRS with the temporal coverage of GOES, it should match the fire boundaries of VIIRS and it should bear at least some correspondence with the classifications of FDCC in areas where VIIRS was orbiting a different region at time of classification.

The final fire classification source that is included in the display map for comparison purposes is MODIS. The classifications from MODIS are very similar to VIIRS. They come from a decision tree operating on infrared bands from a LEO satellite, and are distributed from the NASA API as CSV data [modis users guide]. MODIS provides a second high-resolution comparison source, and could potentially highlight systematic flaws in using VIIRS as training data. Adding these MODIS classification boundaries was a relatively easy step as the functions for uploading VIIRS classifications could be reused with a few tweaks.

**Classification pipeline**

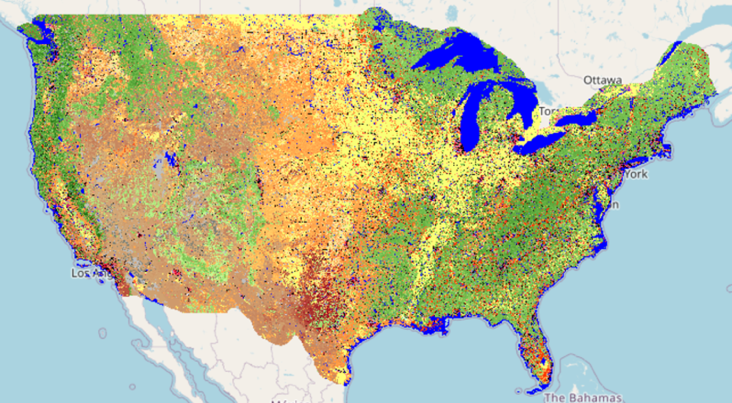
In this section of this document, the coding work that went into the data pipeline for this project will be explained piece by piece. I will describe these starting with the front-end display and moving towards the back-end classification generation.

**Flask Server for Map Display**

I created a flask server with a leaflet map that could display GeoJSON strings as points and polygons, which would allow a user to pan, zoom, and filter which layers to view. The flask server which is deployed with Google App Engine operates by fetching GeoJSON strings from a Google BigQuery (GBQ) database. The most recent Firenet predictions are one layer, while GOES and VIIRS decision tree fire classifications which serve for cross-validation comprise two other layers.

When designing the front-end display of fire classifications, it was necessary to choose between displaying fire classifications as TIFs vs. GeoJSON. TIFs are image files that can be mapped to a coordinate reference system (CRS) and GeoJSON is a way of describing points, lines, and shapes on a CRS. Either can be displayed against a base map viewable with a browser. The benefit of choosing to display TIF images would have been the capacity for a more fine-grained representation of model output. Each pixel’s shading could have added an extra dimension of model confidence. GeoJSON display is far more simplistic. For fire representation, it means model output will be simple points or polygons, indicating presence or absence of fire. Figures 5 and 6 show a comparison of TIF vs. GeoJSON format.

A map of the united states

Description automatically generated

**Figure 5:** This is an image from this project’s flask server, displaying GeoJSON points and polygons representing active fires.

**Figure 6:** This is the vegetation height layer introduced previously, pixel colors map to measures of height. A fire classification TIF would resemble this with each pixel’s value relating to fire probability.

One downside to the TIF option would be computational and storage costs. Each TIF encompassing the continental US would be roughly 100 MB large, which means storage costs would quickly skyrocket as a database of past fire prediction grows. A server constantly serving TIFS to multiple users would also get expensive (or slow) quickly. For reference the database at the time of writing contains 12,000 separate predictions, in their current form as GeoJSON, these take up about 20 MB of disk space. If these 12,000 predictions were all TIFS it would take up around 1,200 GB of disk space. Cloud computing and storage costs are only a few dollars per month for this project, using TIFS would likely increase these costs by at least 10x. Figure 7 shows a breakdown of cloud computing costs per month while using GeoJSON.

A graph with orange squares and white squares

Description automatically generated

**Figure 7:** Cloud computing costs in a typical month.

The other downside to the TIF option is that it puts more of the burden of interpretation on the viewer. A viewer examining a TIF of continuous values representing fire presence probabilities is likely doing a mental conversion into something like binary polygons anyway, they are trying to understand what is and isn’t fire. With these considerations in mind, GeoJSON seemed a good choice, especially given the lack of resources to fund this project currently. Future iterations of this project should store predictions as both TIFS and GeoJSON, as a database of accurate classification TIFs may offer value in a model that predicts paths of fire spread.

**Google BigQuery Database**

The second part of the pipeline is the Google BigQuery (GBQ) database. GBQ is a cloud relational database that can hold simple data like numbers and character strings in tables that relate to one another. This is an ideal format for storing the GeoJSON strings. My GBQ database includes four tables, one for the Firenet classifications, then one each for the comparison classifications, GOES FDCC, MODIS, and VIIRS. Each table is structured similarly with columns for GeoJSON strings and columns for the time they were added to the database. Table 2 shows an example row of data from the Firenet classification table.

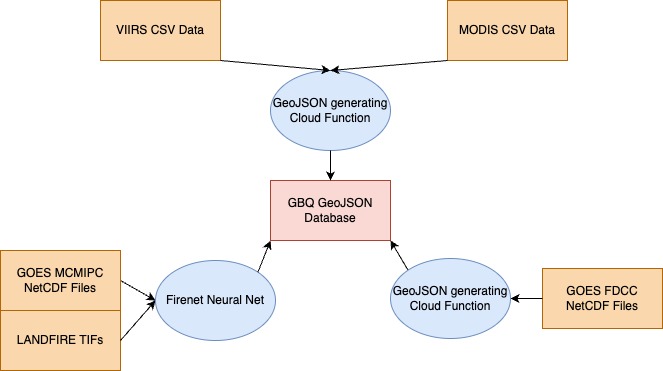
A screenshot of a computer

Description automatically generated

**Table 2:** The GeoJSON column in each table contains a series of GeoJSON ‘features’ which have a type and a set of latitude and longitude coordinates. The types in my database are all either “point” or “polygon.”

**Classification Functions**

The final overarching step in the classification pipeline is the ingestion of the most recent satellite data and the output of GeoJSON strings into the GBQ database. This step is the most complicated and involves several sub-steps. Figure 8 is a diagram for this step, breaking it down into four separate pipelines which terminate at the GBQ database. The function of each of these four pipelines is to ingest satellite product data, perform various transformations, and output GeoJSON to GBQ.



**Figure 8:** Orange boxes represent cloud data sources and blue circles represent cloud compute instances.

The simplest of these four pipelines is the VIIRS comparison data. VIIRS data is downloaded from a nasa.gov API, in the form of csv files. The classification pipeline downloads the most recent set of fire classifications every hour and converts them from a CSV to a GeoDataFrame (GDF). A GDF is a data structure provided by the GeoPandas library in Python, designed specifically for geographic data. Each row in a GeoDataFrame represents a geometric object (e.g., points, lines, polygons) with associated attributes. The key component of a GeoDataFrame that differentiates it from a regular DataFrame is the presence of a 'geometry' column that contains these geometric objects, which can be created from latitude and longitude values in the downloaded VIIRS CSV.

Once the VIIRS points have been converted to GDF format, they are clustered using the python library DBSCAN. DBSCAN looks for any geometries in the GDF that are 0.01 degrees away from one another, adding another column to the GDF indicating cluster number. The clusters are then filtered to only include instances where there are 5 or more points, and at least one of those points is labeled high-confidence. Once these high confidence clusters are identified, a ‘convex hull’ polygon is drawn around each cluster.

**Figure 9:** A convex hull is the smallest convex polygon that can enclose a set of points.

A black and green arrows

Description automatically generated

Each of these polygons is uploaded as a GeoJSON string, along with its datetime stamp to the GBQ database. The second pipeline, for MODIS classifications operates in virtually the same way. The only difference is that confidence levels for MODIS classifications are represented as numbers between 0 and 100, with the threshold for display currently placed at 80 or higher.

The third of the four classification pipelines is the GOES FDCC pipeline. This pipeline operates on NetCDF (NC) files downloaded from the GOES google cloud bucket (this bucket is essentially a file folder in the cloud). Every part of working with this file-type, from downloading, to unpacking their nested structure, to reprojecting them to a CRS that could be converted to GeoJSON was difficult. A major bottleneck in this project for me was the high learning curve in working with NC files.

NC datasets are designed to store and distribute scientific data. They are widely used in various scientific disciplines, especially in atmospheric and oceanic research. A NC dataset is structured around several key components [NetCDF Users Guide]:

* Dimensions: These define the shape of the data. For example, a global temperature dataset might have dimensions for latitude, longitude, and time. Dimensions are used to specify the size of variables.
* Variables: These contain the actual data stored in the dataset, such as light radiance values for satellite imagery. Each variable is associated with a set of dimensions that define its shape. Variables can represent scalar values (associated with zero dimensions) or arrays (associated with one or more dimensions). In the datasets I worked with, most variables were 2D arrays representing an imagery band. Each band was associated with a light wavelength range.
* Attributes: These provide metadata about the dataset, such as the units of measurement, data sources, or any other relevant information about the data or the dataset itself. Attributes can be global (applying to the entire dataset) or specific to individual variables.

NetCDF datasets are self-describing, meaning that they include information about the data they contain. This feature makes NetCDF files portable and easy to share between users and systems without losing the context of the data. NetCDF supports efficient access to large datasets and is capable of representing missing data, supporting data compression, and chunking for optimized access patterns. All of these features are excellent, but a few subtleties of how these files work tripped me up when attempting to make them interoperable with imagery saved in TIFs.

For the GOES FDCC NC files, the variable of interest was the data quality flag (DQF) variable. In the DQF array, points equal to 0 are good quality fire pixels. Transforming this NC array into GeoJSON necessitates moving from the fixed projection of a GOES satellite to the EPSG 4326 CRS which maps to latitude and longitude. Figures 10 and 11 demonstrate the difference between a fixed projection and a typical CRS.

A grid of squares in a grid

Description automatically generated

**Figure 10:** This figure demonstrates how a fixed grid projection system works in a simplified way. Note that grid squares along the edges represent more land-cover than grid squares in the middle. If they were drawn on the ground they might appear stretched. In some other CRS systems this is not the case.

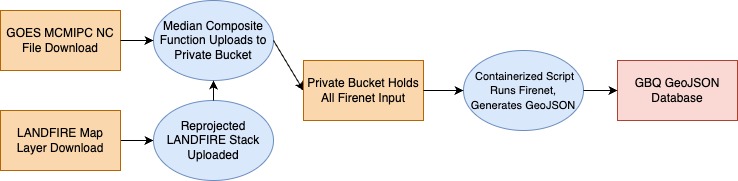
A map of the world with red dots

Description automatically generated

**Figure 11:** A typical CRS maps a grid to a model of the Earth’s surface.

There is a complicated equation using lots of trigonometry which I don’t understand that can be used to transform grid square indices from the coordinate reference system to latitudes and longitudes, and there are also GIS python libraries that can handle reprojection. I ended up using a third approach that seemed to work more consistently with NC files, using a reference file which simply mapped each fixed projection grid square index to a latitude and longitude center point. This is effectively a ‘lookup table’ provided to accompany GOES data. Using a python function with this lookup table to locate the latitude and longitude of each array index where DQF equals 0 allowed me to convert fire locations into a GDF, and then to GeoJSON which is uploaded to GBQ as in the last pipeline.

The fourth and final pipeline integrates multiple satellite data sources, matches them to a common CRS and resolution, and runs the neural net before generating GeoJSON classification polygons. This pipeline is the most complex by far, but it draws on many of the concepts explained in the previous two pipelines. To start with I will display a diagram of this pipeline in figure 12.



**Figure 12:** This pipeline is composed of three functions, the LANDFIRE uploading function runs once a year – whenever LANDFIRE layers are updates. The median composite function runs every 5 minutes – each time the GOES satellite scans the continental US. The containerized Firenet function currently runs every 12 hours to keep costs down, but it could run every 5 minutes with more funding.

This LANDFIRE function accesses the LANDFIRE TIFs which are natively 30 meters per pixel and resamples them to match the resolution of the training data labels – 375 meters per pixel. Resampling is the process of changing the resolution of an image file to a desired resolution. In this case, LANDFIRE is being down-sampled, the resolution is decreased to match the training data resolution. This process loses a lot of information, effectively turning the LANDFIRE layers into ‘pixelated’ versions of themselves, but it is a necessary step to match all input resolutions with each other before feeding them into the neural net. Rioxarray is a python package that has methods to change the CRS and resolution of satellite imagery, and this function and the Firenet function rely on this package.

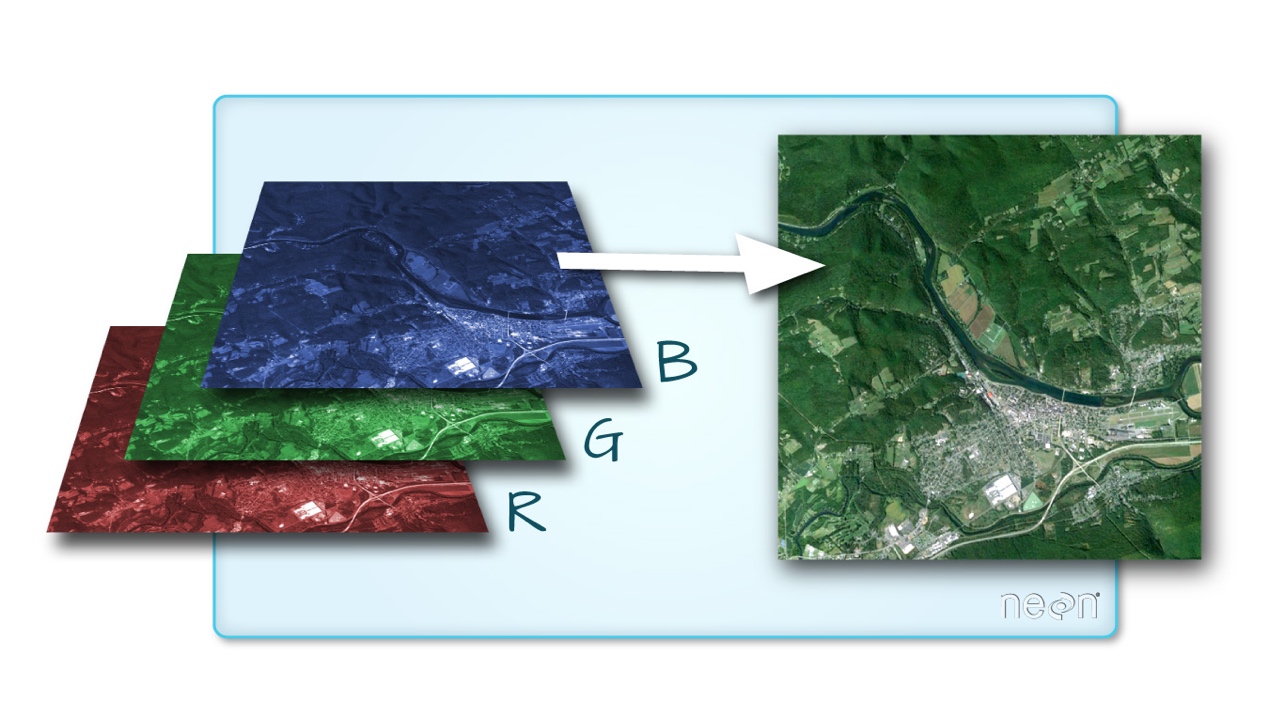
In the LANDFIRE function, the resampled reprojected LANDFIRE layers are also cut down to a particular bounding box in the Western US. The reason for cutting down the layers this way is that it reduces the computing costs to run Firenet. By working with a small but active region to process fire classification for, the project can demonstrate the capacity of Firenet without costing too much. After being clipped to a bounding box, the LANDFIRE layers are stacked in an NC file, and uploaded to the private cloud bucket. They serve as a template that has been matched to the correct resolution, CRS, and spatial extent. Changing this LANDFIRE reference NC file in the private bucket can redirect the ‘attention’ of Firenet to a different region, which will be as easy as changing the coordinates in the bounding box argument of the LANDFIRE function.

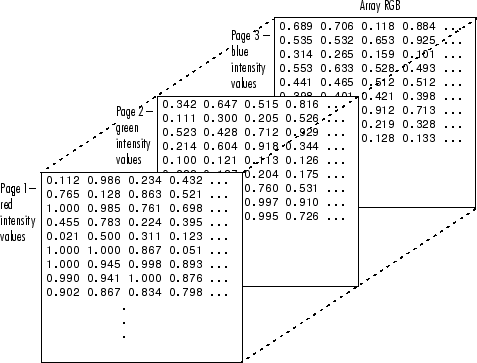
The median composite function downloads the 12 most recent MCMIPC NC files (1 hours worth) and finds the median value of each band at each pixel to reduce noise in the input. This function then downloads the LANDFIRE NC, matches the MCMIPC files to its resolution, CRS, and extent, and stacks them into a single NC file. A feature engineering function then creates four new bands by dividing bands by each other. These are essentially map layers that represent the ratio of one band to another. Each of these engineered features provides useful information to the model for classifying fires [one concern paper]. There are four engineered features in the model taken from the specifications in the One Concern paper, channels 6/5, 7/5, 7/6, and 7/14.

* Channels 6/5: Enhances contrast between fire-affected areas and snow/ice or clouds, aiding in the neural net's ability to distinguish fires from other bright surfaces.
* Channels 7/5: Improves the neural net's capability to detect fires under varying lighting conditions by leveraging differences in reflectance between fire and non-fire surfaces.
* Channels 7/6: Assists the neural net in differentiating between fire and cloud particles by exploiting the distinct thermal properties and sizes of cloud and fire particles.
* Channels 7/14: Enhances the neural net's ability to identify fires by comparing longwave infrared emissions with shortwave emissions to detect heat sources.

After feature engineering, all unused bands are removed from the dataset, leaving a 21-band image, which is uploaded to my private cloud bucket. This image contains within it all the information necessary for Firenet to identify fire boundaries, at least in theory!

The final step in the pipeline is the function that runs Firenet. This function downloads the output from the previous function as input, and outputs the fire boundary classifications as GeoJSON. After downloading the previously described 21 band raster image as input, this function needs to transform the raster imagery bands into a three-dimensional array of normalized numbers.

The best way to conceptualize this three-dimensional array is by understanding its components, which are two-dimensional arrays of numbers that correspond to different imagery bands. Figure 13 shows a simplified version of this. 



**Figure 13:** Each imagery band can be represented as a two-dimensional array of numbers between 0 and 1, indicating the value at each pixel for the band in question. The neural net input includes 21 bands instead of three.

Once the data has been turned into this 3-D array, it is almost ready to be classified by the neural net. Two things need to happen before this can occur though. First, the array needs to be broken into small chunks. The model input begins as a 1748x2266x21 array, and is converted into 945 21x64x64 chunks. Each of these chunks represents an area of roughly 140,000 acres. Chunking in this way allows the neural net to recognize fires more easily since they are taking up a larger proportion of the chunk.

A vital step at this stage is to find a way to save georeferencing metadata associated with each chunk. When the neural net sees a 3D chunk of numbers, it doesn’t know where on the planet they are taken from, it only knows how to predict an array of numbers that mimics the patterns of the labels it was trained on. The information referencing the location the input that the was taken from can be used to apply the neural net predictions to an actual location. Without saving this georeferencing data in an organized way, the fire classification for each chunk is effectively a meaningless array of numbers.

To save this metadata, I used the approach of creating a ‘template’ raster grid. I took the original 21 band raster, deleted 20 of the bands, and overwrote the remaining band with null values. This is essentially an emptied out map where every grid square corresponds to a real location, but the value associated with that location is a null. These empty grid squares will eventually be filled with a probability of fire at each location.

While chunking up the actual data, I assigned each chunk with a ‘slice object,’ which is essentially just a puzzle piece label that can be used later to indicate where the chunk belongs in the original grid. Each time the neural net created an output from the array input, I associated this puzzle piece label with the output. Once all outputs were created I used these labels to ‘mosaic’ all the small chunks into the original 1748x2266 shape. Lastly, I overwrote the null raster grid values with the values in this array.

At this stage there is a raster file mapped to the region of interest in the Northwest US. Each pixel has a value between 0 and 1 that indicates fire-probability. With this as input, I used an approach very similar to what was previously described for clustering and thresholding the VIIRS cross-evaluation data. I create a cutoff point at 0.95[[2]](#footnote-2), and then look for clusters of 5 or more pixels close to each other that have values higher than that cutoff. After these clusters are located, convex hull polygons are drawn around each, converted into GeoJSON, and uploaded to Google BigQuery.

**Model Accuracy Assessment**

To give away the conclusion of this section, our model predictions in their current form do not correspond well to reality, but there are a few clear opportunities to improve its performance. During the training of this model, it had been cross-evaluated with a training and test split as is standard practice. This involved keeping a set of the VIIRS training labels reserved, only training with 80% of the available labels. The model then attempted to classify fire boundaries on the other 20% of inputs he had labels for. This standard practice is used to monitor for “overfitting,” which is where a model learns the patterns of its training data in a way that can’t be extrapolated to unseen data. The cross-validation indicated a dice coefficient of 0.21.

A dice coefficient can range from 0 to 1 where 0 indicates no overlap between predictions and reality and 1 indicates perfect overlap. It is a good metric for model evaluation because it considers both false positives and false negatives. A dice coefficient of 0.21 generally would indicate a poorly performing model, but because of the difficulty of cross-evaluation previously discussed in the cross-evaluation data section, we thought our model might be better than initially apparent. Viewing model outputs during the initial model training process before I came onto the project, there were many instances of promising correspondences between model predictions and training data. Figure 14 shows two of these correspondences, with model predictions on the left and validation data on the right.

A blue and green image

Description automatically generatedA purple and green background

Description automatically generated

A green and yellow square

Description automatically generatedA purple and green background

Description automatically generated

**Figure 14:** These and other examples of correspondence seemed promising enough to move forward with the model.

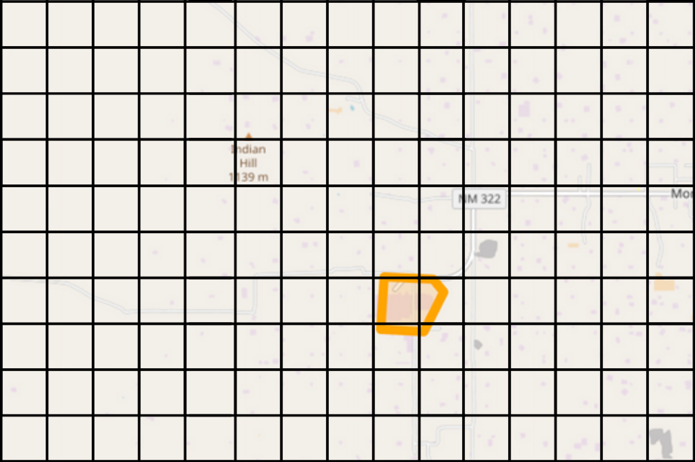
At the time of initial model cross-evaluation, it was quite reasonable to expect that through tuning model parameters and inputs, and increasing the size of the training data, a useful model could be created eventually. Once I created the full model-classification pipeline which allowed georeferencing of predictions on the entire continental US however, a more significant hurdle to creating a model with real-world value came to light.

What I found is that the model is seeing a *lot* more fires than it should even considering the low dice score. We think that this was not obvious during the initial training and cross-evaluation because both the training and test data always included a fire somewhere at the center of each image. Using this type of input seemed the most straightforward way to create training data rasters but it resulted in a very biased sample.

In hindsight, this was a major mistake in model training. What seems to have happened is that the model learned the most obvious pattern in its training data, which was that somewhere near the center of each input chunk there is a fire. This glaring issue was not entirely obvious until we started giving it large areas without fires to evaluate, and we realized it was seeing fires in the middle of every input chunk. The model now needs to be retrained entirely which I will explain next. Fortunately, the work that went into my capstone of creating these pipelines that made the various satellite imagery products interoperable will streamline the retraining process significantly.

**Model Retraining**

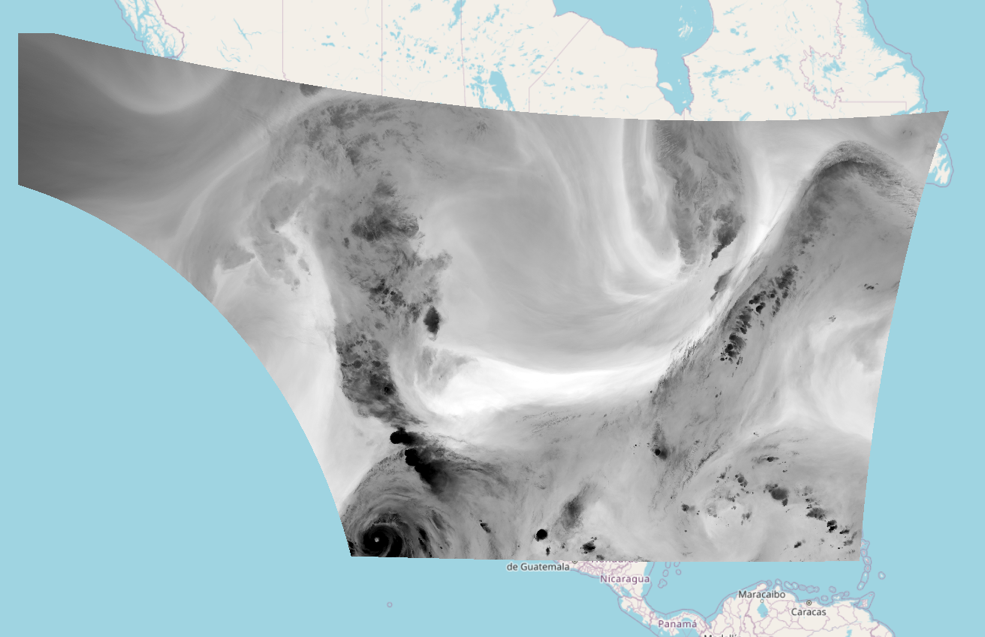
Where the previous training attempt created small raster chips surrounding fires, an effective training function will instead create larger rasters that cover the entire region of interest. Most of the raster area will be entirely absent of fires. These large rasters can be stacked with the input from the previously engineered pipelines and chunked into 64x64 chunks, many of which will include no fires. Figure 15 shows how training data chunks will be created for model retraining.



**Figure 15**: With grid squares representing model input chunks, we will have many cases of chunks without fires for the model to learn from. Additionally, the chunks that include fire presence will not all have fires positioned in the center.

Another benefit of creating the training chunks this way is that we can experiment with overlapping sets of chunks, which means the training data could be increased arbitrarily as we slice up the active fires in different ways. Even without this technique of synthetically increasing the training data volume, there are 2 additional years of overlap between VIIRS labels and GOES input that were. A modular and maintainable data pipeline is vital for allowing this sort of experimentation.

Experimenting with the input bands also has significant potential for improving model accuracy. Many of the bands included in the original model primarily capture cloud cover or atmospheric phenomena, and these bands are likely just noise for a model trying to classify fires. Band 8 for instance, shown below in figure 16 is referred to as the upper-level water vapor band, it is primarily used for identifying jet streams and turbulence signatures. Removing bands like this one when retraining should produce a more accurate model.



**Figure 16:** A band-8 scan is shown here from August 17 2023, large fires that are visible in other bands are completely invisible here.

Another area where experimentation may yield more accuracy is with the PyTorch UNET model’s hyperparameters. Hyperparameters are basically settings that can be tuned for how neural nets learn. Testing different hyperparameters is likely less important than being more selective about model input and fixing the issue with fires always being centered, but it may still have a significant effect on model accuracy. Some of the hyperparameters for initial model training are listed below with explanations and recommendations.

* **Number of Epochs** The number of epochs was set to 50. This parameter indicates how many times the training data is passed through the neural net while it is learning the data’s patterns. More epochs are essentially giving the model more chances to fine tune its pattern recognition, usually a point of diminishing returns is reached, but it likely will be a point higher than 50.
* **Learning rate:** The learning rate for initial model training was set to 0.002. A lower learning rate may be more effective for adapting to a complex “loss landscape,” especially with more epochs. The learning rate essentially adjusts how large the corrections the model makes to its mistakes with each epoch, more epochs with smaller corrections are good for a complex pattern recognition task. The low initial number of epochs and high learning rate choices were partly due to computation time limitations, which is not as relevant now that the project is being run in the cloud.
* **Optimizer:** The model optimizer used for training was the Adam optimizer, it determines how learning rates may be adapted per parameter across epochs. Testing other optimizers may yield better results.

Other model hyperparameters are a bit more complex like “kernel size” and “pooling size.” Through conversations with my collaborator who is experienced with using neural nets to detect land cover however, these are already likely to be near-optimal. These more complex ‘architectural’ hyperparameters are less frequently tuned when experimenting with a UNET. Tuning the architectural hyperparameters like these is akin to changing the ingredients in a cake recipe while changing the more simple hyperparameters is more like changing the oven temperature and bake time. To carry this metaphor further, none of this process-changing will amount to much while our inputs are still the equivalent of rotten eggs.

**Conclusion**

The process of working on this project had significant educational value for me. It served as an introduction to using neural nets and cloud computing, while also allowing me to develop my skills in working with satellite imagery. There is a lot of work to be done before this model is viable. However, the potential it has to assist in tracking the spread of large fast-moving fires in real-time is an area where small improvements may have tremendous value. The impact of this capstone project was to highlight a major flaw in the original model’s training, while making the retraining process significantly more streamlined. Additionally an automated classification pipeline is fully developed, and a retrained model can replace the old model by changing a single file of saved model parameters in a google cloud storage bucket.

Chuvieco, E., Aguado, I., Salas, J. et al. Satellite Remote Sensing Contributions to Wildland Fire Science and Management. Curr Forestry Rep 6, 81–96 (2020). https://doi.org/10.1007/s40725-020-00116-5

1. Pixel resolutions actually differ depending on their position. Pixels near the edges of the field of view are ‘stretched’ as the curvature of the earth means more land cover for that grid square. However GOES imagery is typically referred to as having 2km per pixel resolution. [↑](#footnote-ref-1)
2. This chosen cutoff is a placeholder, after model-retraining a new threshold should be found through experimenting with what leads to the best convergence with comparison classifications. [↑](#footnote-ref-2)