Turbulence Prediction using GOES Satellite Data

Executive Summary

The pipeline and system which the Tufts University Team Sky Blue designed and developed utilized publicly available GOES 16 Satellite Data, and specifically the Cloud Moisture Imagery Product (CMIPC) data, in order to predict the turbulence over the Continental United States (CONUS) region up to 8 hours into the future. In particular, the prediction is facilitated by a machine learning model developed and architected by the team which utilizes current atmospheric conditions. Ground truth data for the labeling of verified turbulence to pair with the satellite data during model training came from PIREPs. In order to interface with this model, the team (in conjunction with Team Celestial Blue, who worked with NEXRAD data) developed a user interface that is able to filter on and use the model output data.

Data Specifications

As above mentioned, for the task of predicting turbulence within the CONUS region, the GOES CMIPC data was trained, served from the Amazon AWS S3 Bucket found here. In order to work with this dataset, we leveraged the Goes2Go Python library. When deciding what satellite bands to train the model with, the team focused on choosing bands that would be more likely to provide information by which the model could learn underlying patterns with turbulence, and specifically Clear Air Turbulence (CAT). The bands that we ultimately decide on were: bands 8 upper level water vapor (wavelength 6.2), band 9 mid level water vapor (wavelength 7.3), band 10 low level water vapor (wavelength 7.3), band 13 clean long wave infrared window (wavelength 10.3), band 14 infrared long wave (wavelength 11.2), and band 15 dirty window (wavelength 12.3). This decision was made using the Band Source Documentation. Note that for the scope of the current project, the team restricted the GOES data exclusively to GOES-16 EAST. Custom code logic would be needed to union with GOES 18 or any other GOES satellites.

Pipeline Usage

The pipeline developed and packaged by the team consists of two main modules: a satellite module that preprocesses, cleans, and works with the GOES Satellite data source and a PIREP module that preprocesses, filters on, and parses retrieved information from pilot reports fetched from the PIREP Dataset Source. These two main modules contain the logic for pulling both the input data (often called "X" data in machine learning) and the ground truth data (often called "Y" data in machine learning) in a form that is ready to be used to train the machine learning model.

To utilize these modules for model training, utility functions have been provided and documented in *model.py* under the *src* folder. These utility functions call the relevant and required components of the modules and pass on the data to a Tensorflow Generator class (*Generator.py*) for model training.

Model Explanation

Different machine learning model architectures are designed to perform better at different tasks, such as regression, classification, clustering, etc. In order to train a model that could be adept at predicting turbulence, the team needed to design a model architecture that would be adept at forecasting, which can be thought of as a subset of the regression problem. That is to say, forecasting involves predicting a future value based on historical data and relationships between feature variables. This was decided on instead of treating the problem as a binary classification of turbulence vs no turbulence, which would have been a simpler problem formulation but would likely have yielded less actionable insights and predictions.

When deciding on the kind of machine learning model to base our model development on, one of the key considerations was the temporal dynamics underlying weather data, which of course is at the heart of this problem and its accompanying dataset. The dynamical systems underlying weather data are highly temporal in nature, and so treating this problem as a typical (x,y) data pairing in traditional supervised techniques would lose a lot of that temporal information. Thus, the team decided that the problem should be framed as a time series forecasting problem, which meant that in the data given to the model during training, some sense of time had to be encoded so it could learn based on that information.

One of the most successful models in learning time patterns and information is the **Long Short Term Memory (LSTM)** architecture, which is able to learn based on recent and more long term information about the data. Furthermore, another aspect to consider is that GOES data comes in the form of satellite images. One of the most successful models for learning information in images by identifying patterns in those images is the **Convolutional Neural Network (CNN).** Thus, for this model, we implemented a Convolutional LSTM (ConvLSTM2D from Tensorflow) architecture to combine the benefits of both of these neural network architectures. Additional resources for these architectures are linked under Resources. In order to edit and build upon this model architecture, please reference the section on Model Training Instructions.

Model Training Instructions

The code required to train the model using the data pipeline described under Pipeline Usage lives within the *model.py* file under the *src* folder. It is important to note that the result of model training is a .keras model file from which a separate script can conduct inference, but making predictions is not part of this particular script. For more information on model predictions, see Frontend Usage and Model Predictions.

In order to make the process of model training flexible per the different cases, the entrypoint to our pipeline (*model.py*) has several command line arguments set up to customize how the user wants the model training to be conducted. In order to streamline the usage of this file, the command line arguments are outlined here and can also be found by running with the -h option:

--save_path is a required parameter whose value should be a string containing the file path at which the user would like to save the model and any associated artefacts of the code

--start_date should be a string in the format of "YYYY_MM_DD" determining the start date of the range on which the train-validation split will be conducted for model training

--end_date should be a string in the format of "YYYY_MM_DD" determining the end date of the range on which the train-validation split will be conducted for model training

--batch size is a parameter whose value dictates how the size of the batches on which the model will be trained at any given time. For maximal CUDA level optimization, this should be a power of 2. Furthermore, the major limitation on batch_size is the memory available from the hardware, discussed further in Hardware Constraints.

--mode is the last command line argument and it dictates whether model training conducts hyperparameter tuning, uses the team's knowledge and insight in which they picked their estimate at best hyperparameters for the problem, or resume training from a checkpoint. Note that depending on the mode chosen additional parameters are offered or required.

The current model architecture can be extended to be "deeper", which will allow the model to learn more complex patterns provided that more data is provided to the model. Note that in cases where data is sparse, such as in this case further explained in Vanishing Gradients, a deeper model is more likely to **overfit** or mimic the training data so well that it performs poorly on unseen data when making predictions.

To enhance the model's ability to detect fine-grained turbulence features, one approach is to stack multiple ConvLSTM2D layers and follow them with spatial refinement blocks. See Appendix A. Model Extension Example for Python code defining an example architecture.

Vanishing Gradients Problem

The vanishing gradient problem occurs when a model's gradients — the signals used to update the model during training — become too small to be useful. This stalls learning within the network. When data is sparse, such as when PIREP labels are infrequent over the CONUS grid, the model is often exposed to a majority of "inactive" or low-information inputs. As a result, the computed gradients during backpropagation are dominated by regions with little to no signal, causing them to shrink rapidly as they pass through layers.

In this context, sparsity effectively "drowns out" the rare but important features. The gradients corresponding to those features may never propagate back with enough strength to meaningfully adjust the weights in earlier layers. Over time, this leads to a model that fails to learn the subtle patterns associated with rare events despite having the capacity to do so.

Hardware Constraints

The size of the satellite data and the high dimensionality of our data (especially in the spaces of latitude, longitude, and altitude respectively in our grid over CONUS) made compute a bottleneck for the project. On both the memory and time axes, the team had to optimize pretty heavily, which made finding the balance point between these two considerations difficult to manage owing to the known trade-offs between them. As such, multiple avenues of optimization were considered and implemented, but further work would have to be conducted or more compute would have to be available to mitigate these more. The current model training pipeline has been built assuming it would run on a single GPU (H100 being the best available to the team).

Caching

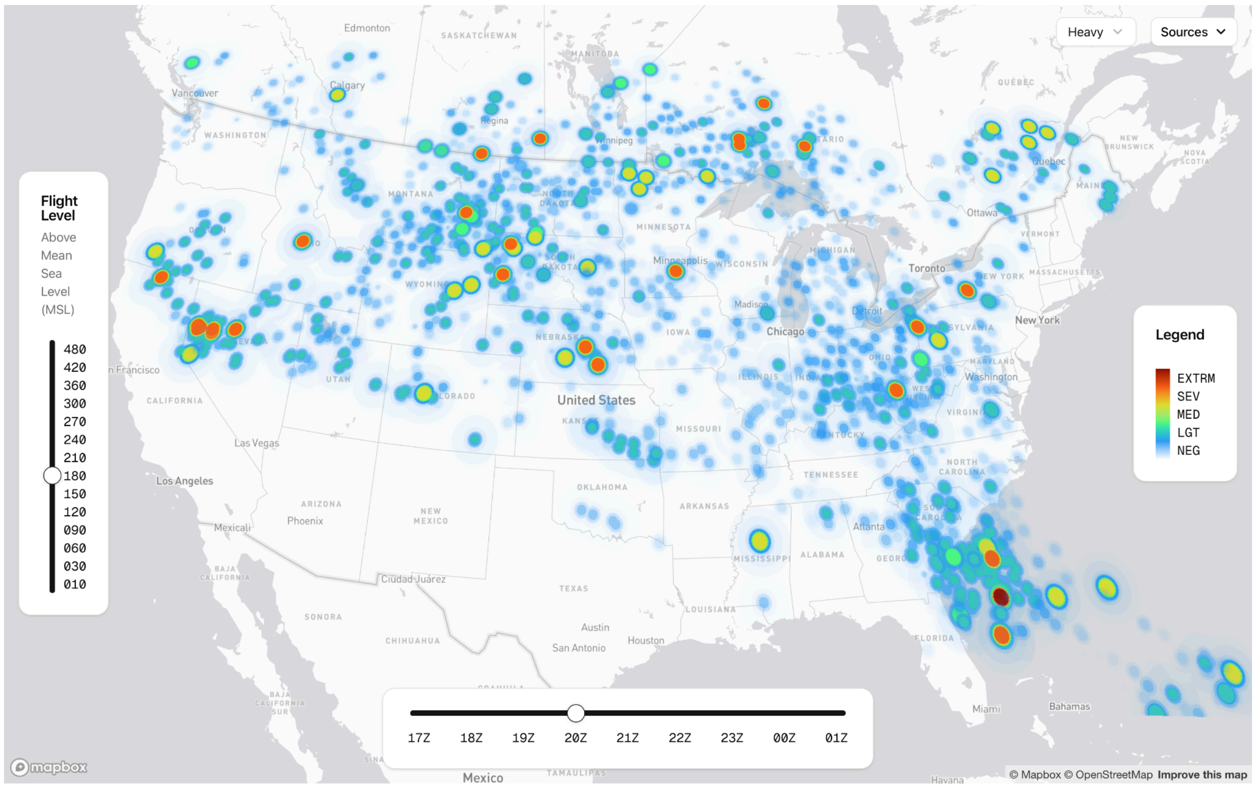
Retrieving satellite images from an AWS bucket took prohibitively long to do (3-7s per image at best; 3-10 minutes at worst). Given the large number of images needed to train a model, eliminating the need to fetch data during training would speed up training time significantly. This is especially true given that the "time series" framing of the data means that any given data image would be used 9 times throughout training. The team decided to prefetch images and save them to a local directory ahead of time, allowing for much quicker access during model training. PIERP data was localized in a similar manner. The team took all PIREPs in the time window that training was planned to occur and saved them so that they could be indexed and retrieved by timestamp. This also cut down data fetching time when training since the program doing the training did not have to reach across the internet for PIREP data.

Multiprocessing

Fetching data before training would take just as long as fetching during training unless the data to be fetched was split among many processes or machines. The team opted to run a Python script that ran several processes in parallel which all worked to download individual images and save them into a local directory. The efficiency of this approach scaled with the number of processes until the network bandwidth of the machine those processes were running on was reached. At that point, the only way to get any more image throughput would be to distribute work among multiple machines. The team was able to use the Tufts High Performance Computing Cluster to do this by running the script *satellite_cache.py* on several machines in parallel, saving those images to local directories so they could be accessed quickly during training.

Frontend Usage and Model Predictions

In order to generate model predictions for the frontend, the script *model_prediction.py* should be used. It contains functionality to create a correctly formatted input frame from a given timestamp. In the case of predicting for the frontend, this script should run on the hour every hour to predict the next 8 hours of data. This script requires that a model already be trained and saved in a .keras file (such as the one that *model.py* creates).



Our frontend, depicted in the image example above, is the interface by which a user can interact with the model outputs of the trained model when it is used to make predictions of turbulence for a designated time window. The frontend allows users to choose timeframes, altitudes, or data sources (satellite or radar) for predictions, and can choose an aircraft size classification to get precise turbulence risks.

The frontend was built using Next.js, TypeScript, and Tailwind CSS, and utilizes Mapbox for the maps and graphics. Running the app requires Node.js, npm, and a Mapbox API key; current functionality is sufficiently handled by Mapbox's free tier, but for extensive production use may exceed the free limits. Detailed instructions for acquiring a Mapbox API key are provided in the README documentation for the frontend repository.

Controls are organized in a single directory, with each UI element separated to allow for easy repositioning and reconfiguring within the main Map component. Predictions are currently sourced from a local cache of frames in .gif format, stored at `public/frames/{source}/frame{num}/alt{idx}.gif` where `{source}` is the source of the data (either "sat" for satellite or "rad" for radar), `{num}` is the frame number, which represents the index along the time slider, and `{idx}` is the index of the altitude.

This is designed to easily plug into a future backend which would automatically source predictions at regular intervals, so that the frontend can simply provide a source, timestamp, and altitude to fetch a relevant frame. Frames are displayed using Mapbox's raster layers, and cached locally on the user's end to reduce response times when filters are changed.

Appendix A: Model Extension Example

```
model = keras.Sequential()
model.add(keras.layers.Input((dim_frames, dim_lat, dim_lon, dim_bands)))
# First ConvLSTM2D block
model.add(keras.layers.ConvLSTM2D(
    filters=hp_filters,
    kernel_size=(3, 3),
    padding="same",
    return_sequences=True
))
model.add(keras.layers.BatchNormalization())

# Second ConvLSTM2D block
model.add(keras.layers.ConvLSTM2D(
    filters=hp_filters * 2,
    kernel_size=(3, 3),
    padding="same",
    return_sequences=True
))
model.add(keras.layers.BatchNormalization())

# Dropout for regularization
model.add(keras.layers.Dropout(rate=hp_dropout))

# 3D convolution to refine features over time and space
model.add(keras.layers.Conv3D(
    filters=dim_alt * 2,
    kernel_size=(3, 3, 3),
    activation="relu",
    padding="same"
))
model.add(keras.layers.BatchNormalization())

# 1x1x1 convolution to project to output dimension
model.add(keras.layers.Conv3D(
  filters=dim_alt,
  kernel_size=(1, 1, 1),
  activation="linear",
  padding="same"
))
```

Resources

GOES Satellite Imagery Viewer: this is the resource we utilized to visualize satellite images and to verify our projections and data processing were reasonable.
PIREP Dataset Source: this is the resource we utilized to pull PIREPs to use for our ground truth dataset.
Band Source Documentation: this is the resource we used to decide which bands might be most relevant for training our model on satellite imagery from GOES-16 for predicting CAT in CONUS.
CNN Explainer: interactive explanation of Convolutional Neural Networks with examples provided so you can visualize how these networks learn as well as providing some intuitive mathematical explanation
Technical LSTM Explainer: technical deep dive into how Long Short Term Memory neural networks work, building upon their predecessor, known as Recurrent Neural Networks.
LSTM Explainer: slightly shorter, less technical LSTM explainer that gives a good overview of this architecture's various components and how they allow the model to learn temporal information.