Hurricane Harvey Building Damage Classification — Project Report

Data Preparation

In this project, we were given a dataset of satellite images of buildings in Texas following Hurricane Harvey. The goal was to classify each image as showing a damaged or undamaged building. The dataset was structured in two folders: one for images labeled damaged, and another for undamaged. Our first task was to preprocess these images to prepare them for training and evaluation.

To begin, we loaded all images using the PIL library and resized them to a uniform size of 128x128 pixels to ensure consistent input dimensions for our neural networks. We chose this size as a balance between maintaining visual detail and reducing computational cost. Each image was converted to RGB mode (if not already in that format), and we normalized the pixel values by dividing by 255 to bring them into the [0, 1] range, which is standard for neural network inputs.

Once all images were loaded and normalized, we randomly shuffled the dataset and split it into three subsets: 70% for training, 15% for validation, and 15% for testing. This split was manually implemented using train_test_split from sklearn, rather than relying on TensorFlow's built-in generators. This manual approach gave us more explicit control and guaranteed that the test data would remain completely unseen during training and hyperparameter tuning. After preprocessing, we visualized random samples from the training set to verify that labels were correctly assigned and the resizing had preserved enough detail.

Model Design

We implemented and evaluated three different neural network architectures, each with increasing complexity. All models were trained and tested using TensorFlow/Keras in the Jupyter notebook coe_project3_part1n2.ipynb. Each model used binary cross-entropy loss and the Adam optimizer, and was trained for 10 epochs with validation-based early stopping.

1. **Dense Fully Connected Network (ANN)**
   Our baseline model consisted of a Flatten layer, followed by two dense layers with 256 and 128 units, each using ReLU activation. A Dropout layer (rate: 0.3) was used to mitigate overfitting. The output layer was a single sigmoid unit to predict damage probability. While this model was computationally efficient, it did not include any convolutional filters, and thus lacked the ability to capture local spatial patterns in the

images.

2. **LeNet-5 CNN**
   As a more spatially aware alternative, we implemented a version of the LeNet-5 convolutional neural network adapted for RGB input. This model started with a convolutional layer of 6 filters (5×5 kernel) and tanh activation, followed by average pooling. It was followed by a second convolutional layer with 16 filters and another pooling layer, and ended with two dense layers of 120 and 84 units, before the final binary output. LeNet-5 introduced convolutional feature extraction, allowing the model to better capture texture and structural anomalies associated with damage.

3. **Alternate LeNet-5 (Modern CNN)**
   Our final and best-performing model was a **combined architecture** that integrated ideas from both LeNet-5 and modern CNN design practices. This custom-built model featured three convolutional layers with increasing filter sizes (32, 64, and 128), each followed by ReLU activations and MaxPooling layers to downsample spatial dimensions. The feature maps were flattened and passed through a dense layer with 256 units and dropout regularization, then output through a sigmoid-activated single neuron. This architecture was purpose-built to capture both low- and high-level spatial features, and it proved to be significantly more robust and accurate than the earlier models.

This combined CNN model effectively drew on the strengths of traditional feature-extraction pipelines and modern best practices, resulting in superior generalization and performance.

Model Evaluation

After training all three models on the same data splits, we compared their performance on the unseen test set to evaluate generalization.

The fully connected ANN achieved test accuracy around 66.2%, which was relatively low. It struggled especially on borderline cases where structural damage was subtle. This reinforced our expectation that dense-only models are not suitable for image tasks without spatial feature extraction.

The classic LeNet-5 model performed better, achieving around 84.1% accuracy on the test set. It was noticeably more stable across epochs and produced fewer false positives than the dense model. However, it still exhibited occasional overfitting and struggled with lighting and scale variations.

The best results were obtained from the alternate LeNet-5 model, which consistently achieved 96.5% accuracy on the test set. It was more robust to image noise and color variation, likely due to the deeper convolutional pipeline and increased capacity.

Given the high test accuracy and stable validation performance of the Alternate LeNet-5 model, we are confident in its ability to serve as a reliable damage classifier. While further enhancements like data augmentation or hyperparameter tuning may improve performance marginally, the current model achieves state-of-the-art results for the task and data at hand.

Model Deployment and Inference

The goal was to enable robust, real-time predictions in a scalable and platform-independent manner. To achieve this, we developed a RESTful web API using Flask and containerized the entire application using Docker, ensuring consistency across different development and deployment environments.

The deployed API is built around a simple yet extensible architecture. At startup, the Flask application loads the trained Keras model from disk (specifically from the app/model/best_model/ directory). This model is expected to be in HDF5 format and includes both the architecture and the learned weights. Upon receiving a request, the server processes the input image and returns a classification result. The inference process closely mirrors the preprocessing steps used during training to ensure consistency in input data representation.

Two key HTTP endpoints are exposed:

1. **GET /summary** – This endpoint provides a lightweight summary of the model's structure and readiness. It returns a JSON object containing metadata such as the model name, input shape, output type, and number of parameters. This endpoint is used both as a health check and for grading validation, ensuring that the model is properly loaded and the server is responsive.

2. **POST /inference** – This is the primary endpoint for model inference. It expects a multipart/form-data request with an image file provided under the "image" key. Upon receiving the request, the server uses the Pillow library to load and process the image. It resizes the image to 128x128 pixels, converts it to RGB if necessary, normalizes the pixel values to the [0,1][0, 1][0,1] range, and reshapes the array to the expected input shape of the model. The preprocessed image is then passed to the model for inference. The output is a probability score, which is thresholded at 0.5 to return a binary classification—either "damage" or "no_damage"—in a structured JSON response.

The server is designed to run inside a Docker container to ensure reproducibility and portability. Two deployment options are provided for convenience:

- **Pre-built Docker Image**: Users can quickly deploy the API by pulling the latest image from Docker Hub using docker pull parpat/damage-detection:latest, then starting the server with docker-compose up. This method is the most convenient and minimizes setup time.

- **Build from Source**: For users who prefer or need to build the image locally, the repository provides a Dockerfile that installs all required dependencies (TensorFlow, Flask, NumPy, Pillow, etc.) and copies the model and application files into the container. Running docker build -t damage-detection . followed by docker run -p 5000:5000 damage-detection launches the server locally.

In addition to this robust deployment setup, the project provides detailed instructions in the README.md for testing the server using three different methods: Python scripts with requests, curl commands from the command line, and direct inference using example images from the test dataset. Each method demonstrates how to query the server, receive predictions, and interpret responses, making the system approachable for users with varying levels of experience.

Conclusion

This project demonstrated a full end-to-end machine learning pipeline for damage detection from satellite imagery — from data preprocessing, model selection, and training, to deployment using a containerized Flask API. Our best model, a deep convolutional neural network based on an improved LeNet-5 architecture, achieved a test accuracy of 96.5%, indicating strong generalization on unseen imagery.

The model was successfully deployed via Docker and can be queried with a simple HTTP request, making it practical for integration into disaster response pipelines. Future directions include incorporating geographic metadata, training on larger datasets, and expanding the model to support multi-class damage levels.