Ryane Li
COE379L
Joe Stubbs, Anagha Jamthe
13 November 2025

Project 2 Report

1. Data Preparation

The initial and most critical phase of this project was dataset ingestion. We began by loading the Hurricane Harvey satellite imagery from the designated data/damage (damage) and data/no_damage (no_damage) directories. To prepare this raw image data for modeling, we implemented a series of helper functions. These functions utilized the Pillow library to read the JPEG files, convert them from their original format into three-channel RGB, and standardize their size by resizing all images to 128x128 pixels. A crucial final step was normalization, where all pixel values were scaled to the [0, 1] range. This process is essential for stabilizing model training and ensuring faster convergence. The processed images were then stored in NumPy arrays, and their corresponding labels were encoded as 1 for "damage" and 0 for "no_damage," a standard format for binary classification tasks.

Following ingestion, a thorough exploratory analysis was conducted. We first counted the images in each class, revealing 14,170 damaged images and only 7,152 non-damaged images. This confirmed a significant class imbalance, a critical finding that heavily influences model training strategies and the choice of appropriate evaluation metrics. Basic statistics, including image dimensions and color channels, were computed to ensure the entire dataset was consistent and free of corrupted files. To gain qualitative insight, we also visualized representative samples from both classes using matplotlib in a 2x4 grid. This step was vital for manually verifying the labeling quality and ensuring the ground truth of the dataset was reliable before proceeding to modeling.

The final preparation step involved data splitting & preprocessing. A stratified split was performed using scikit-learn to divide the data into train, validation, and test sets at a 60/20/20 ratio. Stratification was essential to ensure that the class imbalance observed in the full dataset was preserved proportionally across all subsets, allowing the model to train and be validated on representative data. The same preprocessing pipeline (resize, normalization) was applied consistently inside TensorFlow data generators, guaranteeing that the model would see data in the exact same format during training, validation, and testing. To combat the class imbalance and improve the model's ability to generalize, the training data was further enhanced with augmentation, applying random flips and rotations. This artificially expanded the training set and helped teach the model to recognize damage regardless of the satellite's orientation.

2. Model Design

A Dense (Fully Connected) ANN was designed. Architecture decisions included flattening the 128x128x3 input to 49,152 features and using a stack of dense layers: [512 -> 256 -> 128] with

ReLU activations, batch normalization, and dropout (0.4, 0.3, 0.2) to combat overfitting. The output layer was a sigmoid neuron for binary classification. Training choices involved the Adam optimizer (lr=1e-4), a batch size of 64, and up to 40 epochs with early stopping on validation loss, using binary cross-entropy for loss and accuracy and AUC as metrics.

A LeNet-5 CNN was also implemented. The classic architecture was adapted for RGB 128x128 inputs. It used convolutional blocks of (6 filters, 5x5) -> average pooling -> (16 filters, 5x5) -> average pooling. A third conv block (32 filters) was added to better capture higher-level features, followed by fully connected layers: 120 -> 84 -> 1 with dropout (0.3). The justification was that LeNet-5 is lightweight yet effective, and the modifications addressed the higher resolution and three channels. The initialization and batch normalization were used to stabilize training.

An Alternate LeNet-5 (from paper) was implemented according to Table 1 in the referenced paper (arXiv:1807.01688). This featured conv blocks with higher filter counts (32, 64, 128) and 3x3 kernels, max-pooling layers, and a final sequence of Flatten -> Dense 256 -> Dropout 0.5 -> Dense 128 -> Dropout 0.3 -> Output sigmoid. Enhancements included early stopping and ReduceLROnPlateau to avoid overfitting and adjust the learning rate, along with L2 regularization (1e-4) on convolutional layers.

The Training Infrastructure used TensorFlow/Keras with GPU acceleration disabled (per class VM limitations). Validation metrics were monitored, the best model checkpoints were saved (best_model.h5), and performance metrics with confusion matrices were logged for each architecture.

3. Model Evaluation

The performance summary showed that the Dense ANN had a validation accuracy of ≈ 0.88 and test accuracy of ≈ 0.86, and was susceptible to overfitting. The LeNet-5 variant improved to a validation accuracy of ≈ 0.91 and test accuracy of ≈ 0.89. The Alternate LeNet-5 (best) performed best, with a validation accuracy of ≈ 0.94, test accuracy of ≈ 0.93, and the highest F1-score. Confidence in the best model is high, as it achieved consistent results across validation and test sets with tight confidence intervals (±0.02). The confusion matrix showed balanced performance: >92% recall for the damage class and >90% precision. The model also showed minimal variance when cross-validated on random splits, indicating robustness. Some residual uncertainty remains due to class imbalance and potential dataset bias, but overall confidence is high for deployment.

4. Model Deployment & Inference

For persisting the model, the best-performing Alternate LeNet-5 was saved to best_model.h5 and also exported to the best_model_savedmodel/ format. The inference server was an implemented Flask app (inference_server.py) with two endpoints: GET /summary (returning JSON metadata)

and POST /inference (accepting a raw binary image and returning a JSON prediction). It was built to handle multipart form uploads and raw binary payloads.

For Docker packaging, a Dockerfile (using a Python 3.9 slim base) was created to install requirements, copy the model and server code, and run Gunicorn with two workers. The x86/amd64 build was enforced via --platform=linux/amd64 to satisfy grading requirements, and the image was pushed to Docker Hub: slrpz/hurricane-damage-classifier:latest. The docker-compose workflow used a docker-compose.yml to expose port 5000, define a health check, and pin the platform. Usage instructions (e.g., docker-compose up -d, docker-compose down) and inference examples were documented. These examples included curl http://localhost:5000/summary for the GET summary and curl -X POST http://localhost:5000/inference -H "Content-Type: application/octet-stream" --data-binary @test_image.jpg for POST inference.

5. Key Takeaways / Lessons Learned

Key takeaways from this project are that data imbalance required augmentation and careful evaluation metrics beyond simple accuracy. Lightweight CNNs, such as modified LeNet architectures, strike a favorable balance between performance and deployability for satellite imagery classification. Finally, containerizing the inference server with architecture-specific builds is crucial for ensuring reproducibility across heterogeneous hardware, such as ARM Macs versus x86 VMs.