Project 3

Hurricane Harvey Damage Classification Using Neural Networks

1. **Data Preparation**
    a. Overview:
        i. The goal of this project was to develop and deploy neural networks capable of classifying satellite images of buildings in Texas as damaged or not damaged after Hurricane Harvey. This was framed as a binary image classification problem, with a dataset provided in two folders: `damage` and `no_damage`.
    b. Data Preparation
        i. Dataset Source
            1. The dataset was cloned from the course GitHub repository:
                a. https://github.com/joestubbs/coe379L-sp25/tree/master/datasets/unit03/Project3
                b. It contains satellite images categorized into `damage` and `no_damage` folders.
        ii. Image Exploration
            1. After cloning, the images were explored using Python's `os`, `PIL`, and `matplotlib` libraries to confirm data quantity and quality. A few representative samples from each class were visualized to understand the data distribution and image resolution.
        iii. Preprocessing Steps
            1. Resizing: All images were resized to 128x128 pixels to ensure uniform input dimensions across the models.
            2. Normalization: Pixel values were scaled to the `[0, 1]` range by dividing by 255.0
            3. Label Encoding: The damage class was encoded as `1`, and `no_damage` as `0`.
            4. Splitting: The dataset was split into three sets:
                a. Training set: 70%
                b. Validation set: 15%
                c. Testing set: 15%

    The split was done using `sklearn.model_selection.train_test_split` *with stratification to preserve class balance.*

2. **Model Design and Evaluation**

Three different neural network architectures were designed and trained for the binary classification task:

a. Fully Connected Artificial Neural Network (ANN)

i. Input Layer: Flattened the 128x128x3 image into a 1D vector.
ii. Hidden Layers:
    1. Dense(512) → ReLU → Dropout(0.5)
    2. Dense(256) → ReLU → Dropout(0.3)
iii. Output Layer: Dense(1) with sigmoid activation for binary classification.
iv. Loss Function: Binary Crossentropy
v. Optimizer: Adam

- *This architecture performed reasonably but lacked spatial feature extraction, making it less effective than the CNN models.*

b. LeNet-5 Convolutional Neural Network

i. Convolutional Layers: 6 and 16 filters with 5×5 kernels.
ii. Pooling: MaxPooling after each convolution.
iii. Fully Connected Layers: 120 → 84 → 1
iv. Output: Sigmoid for binary output

- *This classic CNN structure performed better than the ANN by capturing spatial patterns in the image, but not as well as the enhanced version below. 3. Alternate-LeNet-5 (from Paper) Adapted from the paper "An Optimized CNN Architecture for Satellite Image Classification". Conv Layers: 32 and 64 filters with 3x3 kernels Pooling: MaxPooling after each conv layer Dense: 120 → 84 → 1 This model was more expressive and captured features better due to deeper convolutional layers.*

c. Alternate-LeNet-5 (from Paper)

- *Adapted from the paper "An Optimized CNN Architecture for Satellite Image Classification".*

i. Conv Layers: 32 and 64 filters with 3x3 kernels
ii. Pooling: MaxPooling after each conv layer Dense: 120 → 84 → 1

- *This model was more expressive and captured features better due to deeper convolutional layers*

2a. Model Evaluation

Each model was trained for 10 epochs and evaluated on the held-out test set. Performance was assessed using accuracy:

Model	*Test Accuracy*
Fully Connected ANN	~66.46%
LeNet-5 CNN	~87.93%
Alternate-LeNet-5	~95.47%

3. **Model Deployment and Inference**

After evaluating the performance of all models, the Alternate-LeNet-5 CNN was selected for deployment due to its high accuracy on the test set**.**

a. Model Saving

i. The trained model was saved in Keras format using:
`model.save("model/best_model.h5")`

b. Inference Server

A simple FastAPI-based inference server was built with the following endpoints:

i. `GET /summary`: Returns metadata about the model, including input shape and class labels.

ii. `POST /inference`: Accepts an image and returns a prediction in the format:

iii. `{ "prediction": "damage" }  // or "no_damage"`

iv. Images are resized and normalized on the server before inference.

c. Dockerization

i. The server and model were containerized using Docker for portable deployment. The Dockerfile installs all required packages, loads the model, and launches the FastAPI app with Uvicorn.

ii. A `docker-compose.yml` file was created to simplify container setup. Example usage:

```
docker-compose build
docker-compose up
```

iii. Users can test the endpoints locally via `curl`:

```
curl http://localhost:8000/summary
curl -X POST http://localhost:8000/inference
-F "file=@example.jpg"
```