Data Preparation:

The dataset used in this project consists of satellite images captured from Texas after Hurricane Harvey, categorized into two classes: “damage” and “no_damage.” An analysis of the dataset revealed a class imbalance, with 11,353 “no_damage” images and 5,705 “damage” images. To mitigate the potential bias toward the majority class, class weights were assigned during training so that both classes contributed equally to the loss function. This helped the model learn to identify damaged buildings more effectively despite the uneven data distribution.The dataset was organized into folders, with each folder corresponding to one of the two classes, allowing TensorFlow to automatically assign labels based on directory names. Images were loaded using the tf.keras.utils.image_dataset_from_directory() function, which efficiently creates TensorFlow datasets from image folders while automatically generating labels and batching the data. A 20% validation split was applied to the dataset, dividing it into training and validation subsets using the subset='training' and subset='validation' arguments.All images were resized to 224 × 224 × 3 to maintain consistent input dimensions for the neural networks, and a batch size of 32 was selected. Finally, all images were normalized by rescaling pixel values from the original range (0–255) to the range [0, 1] using TensorFlow’s Rescaling(1./255) layer. This normalization step improved model convergence and training stability across all architectures.

Model Design & Evaluation

Artificial Neural Network: For the first architecture, we implemented a Dense (fully connected) Artificial Neural Network(ANN) to establish a baseline for image classification. The input images were first flattened into one-dimensional vectors so they could be processed by dense layers. The model consisted of multiple hidden layers with 512, 256, 1228 and 64 neurons, each using the ReLU activation function to introduce nonlinearity. To prevent overfitting, Dropout layers were added after each dense layer, randomly deactivating a faction of neurons(30 -50%) during training. The output layer used a sigmoid activation, making it suitable for binary classification(damage, vs no damage). The model was compiled with the Adam optimizer and binary cross-entropy loss , and evaluated using accuracy, precision and recall as performance metrics

LeNet-5 Architecture:

For the second architecture, we implemented a LeNet-5–based Convolutional Neural Network (CNN), which was adapted from the original LeNet-5 model to handle colored (RGB) images of size. The network begins with two convolutional blocks. The first convolutional layer applies 6 filters of size 5 x 5 with a tanh activation function and padding = ‘same’, which preserves the spatial dimensions of the input. This allows the model to capture low-level features such as edges and color gradients. And it is followed by an average pooling layer with a pool size 2x2 and stride of 2, which reduces spatial resolution by half. The second convolutional layer increase the number of filter to 16, allowing the model to learn more complex, abstract features, and another average pooling layer downsamples the feature maps After the convolutional stages, the output feature maps are flattened and passed through two fully connected layers with 120 and 84 neurons, both using *tanh* activations. These layers learn non-linear combinations of extracted features. Finally, a single output neuron with a sigmoid activation is used for binary classification to classify the damage and no damage. For training, I used the Adam optimizer and binary cross-entropy loss, which are well-suited for binary classification tasks. The model was also evaluated

using accuracy, precision, and recall metrics to assess both general performance and class-wise reliability.

Alternate LeNet-5 Architecture: The third model explored was an Alternate LeNet-5 Convolutional Neural Network, based on the architecture described in *Table 1, Page 12* of the paper *"Handwritten Digit Recognition using Modified LeNet Architecture".* The model follows a deeper and more modernized LeNet-style design, consisting of three convolutional blocks followed by fully connected layers. Each convolutional block includes: A Conv2D layer that extracts progressively more abstract features (32, 64, and 128 filters respectively). Batch Normalization, which stabilizes training and accelerates convergence by normalizing activations. MaxPooling2D, which reduces spatial resolution by half, helping the model become translation-invariant and reducing computational cost.
The kernel size transitions from 5×5 in the first two blocks to 3x3 times in the third block. The smaller kernel in deeper layers helps capture finer details and local patterns. All convolutional layers use the ReLU activation function to introduce non-linearity and avoid vanishing gradient issues. After feature extraction, the network flattens the feature maps and passes them through two fully connected layers with 256 and 128 neurons, respectively, both followed by Dropout (rate = 0.5) to reduce overfitting by randomly disabling neurons during training. Finally, a sigmoid-activated output neuron is used for binary classification For optimization, the model still uses the Adam optimizer with binary cross-entropy loss, and its performance is monitored using accuracy, precision, and recall metrics. Compared to the original LeNet-5, this modified architecture is deeper, incorporates batch normalization and dropout for better generalization, and uses ReLU instead of tanh for faster and more stable learning. These adjustments make it more suitable for larger, color image datasets.

VGG 16: The fourth model developed was a deeper Convolutional Neural Network inspired by the VGG architecture, designed to further enhance feature extraction capability compared to the LeNet-based models. This architecture consists of three convolutional blocks followed by fully connected layers. Each block includes two 3×3 convolutional layers with ReLU activation and a MaxPooling2D layer (2×2, stride=2). The number of filters increases progressively across the blocks—64, 128, and 256—to allow the network to capture increasingly complex visual patterns as the spatial dimension decreases. Using padding='same' preserves feature map size before pooling, ensuring efficient feature retention. After the convolutional stages, the network flattens the feature maps and passes them through two dense layers with 512 and 256 neurons, each followed by a 0.5 dropout layer to reduce overfitting by randomly deactivating neurons during training. The final layer uses a single neuron with a sigmoid activation function for binary classification of "damage" versus "no_damage." The model is compiled with the Adam optimizer and binary cross-entropy loss, while accuracy, precision, and recall are used as performance metrics to better evaluate classification quality. Compared to the previous LeNet-style networks, this VGG-inspired design is significantly deeper, utilizes smaller kernels for finer feature extraction, and applies dropout regularization to improve generalization. These design choices make the model more robust and suitable for handling larger, high-resolution color image datasets.

Model	Accuracy	Precision	Recall	F1
Dense ANN Model	0.6516	0.3484	1.0000	0.5168
LeNet5 Model	0.6500	0.3503	1.000	0.5188
LeNet5 Model(Paper)	0.9883	0.9763	0.9880	0.9821
VGG 16	0.6728	0.000	0.000	0.000

In this project, both precision and recall are critical metrics that must be balanced carefully. Recall is particularly important because missing a damaged building (false negative) could have serious consequences in disaster response or structural assessment. At the same time, precision is not as critical as recall but also needs to pay attention since it ensures that resources are not wasted on incorrectly classified as damaged. Therefore, the ideal model should achieve high recall to capture all the damage building while achieving relatively high precision for efficient resource allocation. Based on the table we can see, in all four models, LeNet models demonstrated a strong balance between precision and recall, achieving an accuracy above 0.97, precision above 0.98 and recall above 0.96. This performance shows that the LeNet-5 model in paper can effectively identify most damaged buildings while minimizing false alarms, providing reliable and efficient predictions. Although VGG-16 showed non-zero precision and recall during training, the final evaluation produced 0 precision and 0 recall because the model was trained for only 5 epochs(we only do 5 epochs trained because the training time for each epoch takes 1 hour), which is far too short for a deep architecture like VGG-16 to converge. VGG-16 has over 138 million parameters, it requires significantly more training time, careful hyperparameter tuning, and often fine-tuning with a pre-trained ImageNet backbone. With so few epochs, the model likely failed to learn meaningful features for the “damage” class and instead defaulted to predicting only the majority class (“no_damage”). For the further investigation, we trained it for more epochs to see if it got good performance

Model Deployment:

Inference server: The best performing model(LeNet5 from paper) was deployed using a Flask-based inference server to allow external access for evaluation and testing. The server included two required endpoints: GET/SUMMARY and POST /inference. The /summary endpoints returned model metadata such as name, type, and input size in JSON format, ensuring the user could verify that the correct model was running. The /inference endpoint accepted aa binary image input and performed classification using the trained model, returning a JSON response in the required format — either {‘prediction”: “damge”} or {“predictiion” “no_damage”}. The server was designed to load the saved Kera model, preprocess the incoming image, and produce and output a prediction. To make the inference service portable and reproducible, the entire setup was containerized using Docker.

Docker: To make the inference service portable and reproducible, the entire setup was containerized using Docker. The Docker image included Python, Flask, Kerass and all dependencies listed in the requirements.txt file, along with the trained model file. The image was built on the class VM to ensure x86 architecture compatibility.. The container was configured to start the Flask server automatically, exposing port 5000 for HTTP requests. This deployment approach ensured the model could be easily executed and tested in a consistent and isolated environment

Use of AI:

[1] Tool: ChatGPT

Prompt: I have the labeled image and put into two folders, give me the train dataset code

Output::

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    labels="inferred",# assgin the label based on the folder names "damge, no damage"
    label_mode = "int",
    validation_split = 0.2,
    subset= "training",
    seed = SEED,
    image_size = IMG_SIZE,
    batch_size = BATCH_SIZE
```

[2] Tool: ChatGPT

Prompt: I want to count the amount of the images for each class

Output:

```
import glob
counts = {cls: len(glob.glob(os.path.join(data_dir, cls, "*"))) for cls in class_names}
print("Image counts per class:", counts)
```