Project 3 Report

Data Preparation:

The dataset used for this project consisted of aerial property images labeled as either damage or no_damage. We first organized the data into a structure compatible with image classification tasks. The images were split into training and testing sets using an 80/20 ratio with no overlap between the two subsets. These were stored under data/property/train/ and data/property/test/, each with separate folders for both classes. This structure allowed for clean separation of data and easy loading using TensorFlow's image_dataset_from_directory.

All images were resized to 128×128 pixels during dataset creation using the image_size parameter. This resizing preserved enough spatial detail for effective feature extraction while maintaining manageable computational requirements. We also applied pixel normalization by scaling values to the range [0, 1] using a Rescaling layer (1.0/255), which helped improve training stability and model convergence. Each model worked with the 3-channel RGB format, so grayscale conversion was not required.

In total, 17,057 images were used, with 13,646 in the training set and 3,411 for validation. An additional 4,265 images were set aside for final testing. This preprocessing ensured consistency across experiments, enabling accurate model comparisons without confounding variation from input inconsistencies.

Model Design:

We explored and implemented three different architectures in this project:

1. Dense ANN (Artificial Neural Network)
   This model served as a baseline and consisted solely of fully connected layers. The input images were flattened and passed through multiple dense layers with ReLU activation functions. Dropout layers were introduced to mitigate overfitting. The model was compiled using the Adam optimizer and binary crossentropy loss, as the classification task was binary. While this model was simple, it provided a useful benchmark for comparison with convolutional networks.
2. LeNet-5 Convolutional Neural Network (CNN)
   The second model followed the classic LeNet-5 architecture, designed for digit recognition but adapted here for binary classification of property damage. It used two convolutional layers followed by average pooling, with the output flattened and passed through dense layers. The final layer used sigmoid activation for binary prediction. This model retained the original design elements such as average pooling and simple filters, offering a structured baseline with basic spatial feature extraction.
3. Alternate-LeNet-5 CNN
   The third model extended the LeNet-5 structure by introducing modern enhancements. It

used more convolutional layers with increasing depth and replaced average pooling with max pooling. ReLU activations were used throughout, and dropout was added before the final dense layers to reduce overfitting. Batch normalization was also considered in early experimentation. This version proved significantly more powerful and better suited to the complexity of the aerial imagery classification task.

Model Evaluation:

Each model was trained and evaluated on the same preprocessed dataset. The ANN model achieved moderate performance, with a final test accuracy of approximately 78.7%. It struggled to capture spatial structure in the images, which likely limited its generalization ability. The LeNet-5 CNN improved significantly upon the baseline, achieving a test accuracy of around 91.1%, benefiting from its ability to extract visual patterns through convolution and pooling.

The best performance came from the Alternate-LeNet-5 CNN, which achieved a test accuracy of 97.2%. Its deeper architecture and use of regularization helped it converge effectively while maintaining generalization. It consistently outperformed the other models across validation and test sets. Based on its accuracy, stability during training, and resistance to overfitting, we are confident that the Alternate-LeNet-5 model is the most reliable for deployment and inference in real-world use cases, however when tested against the test script it was only 50% accurate meaning in real world scenarios it would not be a great idea to use it which shows overfitting for our model.

Model Deployment and Inference:

The final trained model was saved in .keras format using Keras's model saving functionality, allowing for easy deployment across various platforms. It can be integrated into a web application or REST API using frameworks such as Flask, FastAPI, or TensorFlow Serving. To use the model for inference, it is first loaded using load_model() from Keras. Input images must be preprocessed in the same way as during training: resized to 128×128 pixels, converted to a NumPy array, normalized to a [0, 1] scale, and reshaped to match the model's expected input dimensions of (1, 128, 128, 3). Once processed, the image can be passed through the model to generate a prediction. The model outputs a probability between 0 and 1, which is thresholded at 0.5 to determine whether the image is classified as "damage" or "no_damage." This setup enables fast and reliable inference and can be extended to batch processing or integrated into mobile or web applications. The two ways to use docker to deploy our model is through a regular docker image or a docker-compose.yml file.

COE 379L - Project 3
Arsh Dauwa
Jeet Patel

Docker Image:

Clone the git repo and then run the command docker pull jetp104/api:1. Get into the folder with the docker file. Use the command docker build to build the image you pulled (Again make sure its in the same location as the Dockerfile). Then run docker run making sure it is using port 5000. This will start up the server through the docker image.

docker-compose.yml:

Make sure you have the entire repo cloned with every location being as its found on this git repo. Run the command docker-compose up and this will also start up the server.

How to use model for inference:

There are two routes in the api: /summary and /inference. The /summary route gives the metadata of the model we are using to make the inferences on the images and can be called by one of two curl commands which can be found in the README. The /inference route takes in a binary message and preprocesses it before being passed to the model which is described in depth above. To interact with the inference server you need to use a curl post command to give it a binary image once passed in the server will output a json that says prediction: result_of_model, the curl command is also seen in the README.

Example code and deployment steps are also included in the project's README file to support reproducibility, practical implementation, and give a more indepth way to interact with the server.

ChatGPT Uses:

ChatGPT was used to help debug the inference server where the image given by the test script wasn't getting predicted by the model. The fix was to preprocess the images in the api itself to make it the correct size and normalize it to be suitable for the model to predict.