PROJECT 3 REPORT

My partner for this project is Mukund Raman, a student auditing COE379L.

Data Preparation

After importing the data into my Python environment, we began exploring the data. There are a total of 1000 damage images and a total of 608 no damage images. We used the random library in python to create an 80/20 split across train and test datasets, and then shutil to move the data. We ensured that the data conformed to the size expectation to be 150x150 and RGB, which was represented as (150, 150, 3). A tensorflow utility helped us create an image dataset from the directories. The dataset was then flattened to be between 0 and 1, to avoid large values. The label was binary in this case: 0 or 1. We didn't perform any statistical tests on the image data.

Model Design and Evaluation

The second step was to train the models. We utilized a total of four models, their architectures are described in the below tables.

Model Architecture and Performance Comparison Table

Model Name	Layers	Accuracy Score
Dense ANN	Flatten Dense Dense Dropout Dense Dense Dense	Test accuracy: 0.814 Test loss: 0.402
Lenet-5 CNN	Conv2D AveragePooling2D AveragePooling2D Flatten Dense Dense Dense	Test accuracy: 0.841 Test loss: 0.372

Alternate-Lenet-5 CNN	Conv2D MaxPooling2D Conv2D MaxPooling2D Conv2D MaxPooling2D Conv2D Flatten Dropout Dense Dense		Test accuracy: 0.907 Test loss: 0.242
VGG-16	InputLayer Conv2D Conv2D MaxPooling2D Conv2D Conv2D MaxPooling2D Conv2D Conv2D Conv2D MaxPooling2D Conv2D→	→Conv2D Conv2D MaxPooling2D Conv2D Conv2D Conv2D MaxPooling2D Flatten Dense Dropout Dense Dense	Test accuracy: 0.978 Test loss: 0.047

Initially, we began working on a dense, or fully connected, ANN. Initially, the input data was flattened because the dense layers require a 1D vector as an input, where the original raw data has 3 dimensions. Flatten will reshape the (150, 150, 3) into (67500,). Then we added some dense layers, followed by a dropout in order to prevent overfitting. This was followed by three dropout layers to bring it out to a binary output. The sigmoid activation function along with binary cross entropy was used for binary classification. The dense ANN had the lowest accuracy.

We moved onto a known algorithm named Lenet-5 CNN. This architecture was fixed and only required implementation in code. Its accuracy improved, but we could do better. The alternate-Lenet-5 algorithm also had a fixed architecture, and thus also only required implementation in code. For both Lenet-5 algorithms, we had to adjust the code to accept the input size of (150, 150, 3), and output a binary classification. The alternate Lenet-5 algorithm had a good accuracy of 90%, however we knew we could do better.

Finally, we wanted to try out the VGG-16 algorithm. From a brief internet search,

we saw that this algorithm is a popular algorithm that is commonly considered to be a strong algorithm for image classification problems. To implement this, we pulled the weights and biases from the tensorflow keras library and froze the initial layers so that the VGG-16 weights and biases wouldn't change during our training. Then we flattened the output to be outputted into a dense ANN, featuring dense layers and dropout layers to reduce overfitting. The output was again set to have only one neuron because it is a binary classification problem. This last neuron had an activation function of sigmoid and was compiled with a loss function of binary cross entropy, again to match the binary classification nature of the dataset. This model performed the best compared to the rest of the models.

The model we chose to use is VGG-16, because it performed the best. It had an accuracy of 98%, and its precision, recall, and f1-scores were all above 97%. None of these markers point to overfitting. We were very confident in our model to predict the test images with a high accuracy and low error rate.

Model Deployment and Inference

In order to deploy our model, we utilized an inference server created with the Flask python module. Flask allows us to easily create an inference server over HTTP with the GET and POST PHP methods. Within our server, we have two function calls that enable us to request a summary of the model and to classify an image of a building given an image file. The request method from Flask allows us to easily load up files from the server endpoint. Then, this server was packaged into a Docker container and built into a Docker image, then pushed to Docker hub. A user with Docker and Docker Compose installed on their machine can retrieve the Docker image and run the container through Docker compose functions.

The API can be used with two function calls. The first endpoint is the /summary endpoint accessed through GET, which returns a JSON of the model summary. The second endpoint is the image classification endpoint, accessed through /inference and the POST method, which takes in a binary image file under the key image, and returns the predicted label of damage or no damage. The user can stop the container through Docker compose once again and our README.md also includes troubleshooting steps. We tested our inference server on the given test grader and got 6/6 classification images correct. Looking forward to seeing how our model performs on the hidden dataset!