Abhi Reddy

# Project Report

The goal of this project was to build, evaluate, and deploy a neural-network-based classifier that determines whether a satellite image of a building taken after a Hurricane shows damage or no damage. The data consisted of labeled images stored in two folders, one per class. The dataset was obtained from the course GitHub repository under datasets/unit03/Project2. The images are organized into two subdirectories, damage and no damage. After cloning the repository I pointed the data root to that directory and performed a quick check of the file system, confirming there were around 14,170 images in damage and 7,152 in no damage. This means the dataset is somewhat imbalanced, roughly 2:1 in favor of damaged buildings. I used the tf.keras function to perform label inference from the directory structure and return batched tf.data.Dataset objects. I used an 80/20 split at this stage, then further split the 20% validation subset into validation and test sets. The splitting is stratified by class because images from the dataset of the directory use the directory labels consistently, which helps preserve the class ratio in each subset. To prepare the data for training, I applied the following preprocessing steps. Normalization, where I rescaled the layer to scale pixel values from [0,255] to [0,1]. I then cached and prefetched each dataset and transformed it with cache().prefetch(tf.data.AUTOTUNE) to speed up training by overlapping input pipeline work with model execution. Finally I shuffled the training dataset with a buffer size of 1,000 to reduce correlation between consecutive batches. I also inspected one batch of images and labels to verify shapes and confirm that the labels aligned with visual expectations for each class. Overall, after preprocessing, the input shape for all models was 150, 150, 3 and labels were scalar binary values.

Next, I implemented and trained three main architectures. A dense artificial neural network (ANN), the LeNet-5 convolutional neural network, and an alternate LeNet-5 CNN with higher capacity inspired by the architecture in the provided paper. All three models take the same input shape and output a single sigmoid-activated logit, so they can all be trained with the same binary cross-entropy loss and accuracy metric. The dense ANN was the simplest architecture and served as a baseline. Since fully connected networks do not exploit structure, I first flattened the image, then stacked a few dense layers. The relatively large first dense layer compensates for the flattening of the 150 x 150 x 3 input, and dropout is used to reduce overfitting. The model was compiled with the Adam optimizer and binary crossentropy loss, and trained for a small number of epochs to get a reasonable baseline without excessive training time. Next, I implemented a convolutional architecture based on LeNet-5, which has been historically used for digit recognition. Adapting it to 150 x 150 RGB images, this model introduces translation-invariant feature extraction using small convolutional kernels and pooling. The fully connected part at the end is relatively small compared to the dense baseline, but the convolutional layers capture much richer spatial features. Finally, I implemented an alternate LeNet-5-style model with more filters and deeper convolutional blocks.. The goal was to increase representational capacity while staying in a fairly compact architecture.

Compared to the original LeNet-5, this alternate model uses more filters, increased depth, and dropout in the dense layer to control overfitting. All models were compiled with identical settings, optimizer, binary cross-entropy loss, accuracy metric, and trained for different epoch counts to strike a balance between convergence and training time.

To compare the architectures, I followed a consistent evaluation protocol, trained on the training split, monitoring validation loss and accuracy on the validation split. After training, I computed final metrics on the held-out test set, selecting the best model based on validation accuracy. The results were as follows. Dense ANN: validation accuracy = 0.65 validation loss relatively high. This model struggled to capture spatial patterns, as expected from a fully connected network of images. LeNet-5: validation accuracy = 0.92 with a reasonably low validation loss. This was a large improvement over the dense model, showing the benefit of convolutional structure. Alternate LeNet-5: validation accuracy = 0.96 with a test accuracy =0.95. So, the alternate LeNet-5 had the highest validation accuracy and also performed very well on the test set, indicating good generalization. The validation and test accuracies are close at about 96 to 95 percent, which suggests that while there is some mild overfitting, it is not crazy.

Several aspects of this model make me confident in the selection. First of all, it is consistent in its performance. The alternate LeNet-5 performed best on validation and remained strong on the tests set. It also has  a reasonable gap between training and validation. Training accuracy was high, but the validation curves did not collapse, which means the model did not simply memorize the training data. And although accuracy is influenced by the 2:1 class imbalance, the baseline accuracy of always predicting "damage" would be only about 66 percent. The best model's test accuracy around 95 percent is significantly higher than that.

After choosing the alternate LeNet-5 as the best model, I focused on deployment. The overall approach was to first save the model and metadata, then build a Flask inference server. On startup, the server should load the metadata.json and read image height, image width, and other fields. It should then load the trained model and expose two HTTP endpoints. First endpoint being GET /summary which returns a JSON object with model metadata, and POST /inference which accepts raw image bytes in the request body. The server reads the raw request data, decodes it into an RGB image using Pillow, resizes it to 150 x 150, and rescales pixel values to [0,1]. It also expands to a batch of size 1 and passes it through the model interpreting the output probability as "damage" if  >= 0.5, otherwise "no_damage".It finally returns a JSON{ "prediction": "damage" } Next, the inference server is encapsulated in a Docker image built from a Dockerfile that installs the required system libraries and Python packages, copies server.py and the models directory into /app and exposes port 8000 and runs python server.py. The image is built for the x86 architecture and pushed to Docker Hub as abhireddy23/hurricane-damage-api:v4. The docker-compose.yml file simplifies running the server. To deploy the server first run docker compose up -d which starts the container and maps localhost: 5000 on the host to port 8000 inside the container. Example requests include curl

http://localhost:5000/summary for summary and curl -X POST http://localhost:5000/inference \
--data-binary "@path/to/test_image.jpg" for inference. These commands are also provided in the readme. All in all, in this project I built a complete pipeline for post-disaster building damage recognition from satellite imagery. The project helped me understand and show the advantage of convolutional architectures over a simple dense ANN, with the alternate LeNet-5 achieving the best performance at around 95 percent test accuracy. The final system shows a simple API that can be integrated into larger applications.