Project 3 Report

Aastha Agrawal

1. Data Preparation

For this project, I began by gathering satellite images from a provided dataset in which images are organized into two folders: "damage" and "no_damage." My data preparation steps included the following:

- **Data Loading & Splitting:**
  I used Python's TensorFlow Keras utilities (specifically, ImageDataGenerator) to load images and perform basic preprocessing. Given that the dataset originally contained only two folders, I implemented custom Python scripts to split the data into training, validation, and testing subsets. This allowed me to use separate subsets for training the neural networks and evaluating their performance.

- **Rescaling & Normalization:**
  All images were rescaled by 1/255 to normalize pixel intensity values to the [0, 1] range. This is a common practice when training neural networks as it helps in achieving stable gradients and faster convergence.

- **Data Augmentation:**
  For the training set, I applied various augmentations (such as rotations, width/height shifts, and horizontal flips) to combat overfitting and improve the model's generalization ability. These augmentations helped the neural networks learn robust features under varying conditions, which is crucial for satellite imagery that may exhibit substantial variability.

Overall, my data preparation ensured that I had a balanced, normalized, and augmented dataset ready for training multiple neural network architectures.

2. Model Design and Training

I explored several architectures to solve the binary classification problem of detecting damaged buildings from satellite images:

- **Dense ANN:**
  I implemented a fully connected neural network where the images were first flattened. This network was relatively simple, with two hidden layers (256 and 128 neurons) interleaved with dropout layers to reduce overfitting, and a final sigmoid output layer for binary classification. Although fast to train, the Dense ANN struggled to capture spatial features effectively and generally underperformed compared to CNN-based architectures.

- **LeNet-5 CNN:**
  Inspired by the classical LeNet-5 architecture, I adapted this convolutional neural network (CNN) for RGB images by altering the input shape and output layer to suit my

64×64 image size and binary output. LeNet-5 comprised two blocks of convolution, activation (using tanh), and average pooling, followed by fully connected layers. This network captured image features much more effectively than the Dense ANN.

- **Alternate-LeNet-5 CNN:**
  I also experimented with an alternative version of LeNet-5 that uses a slightly different convolutional configuration (with more filters and the ReLU activation function) to potentially capture more detailed features. This architecture comprised two convolutional blocks (each with an increased number of filters and average pooling) followed by a larger fully connected layer before the final output.

Decisions Made:

- I used dropout in the Dense ANN to combat overfitting.

- For the CNN architectures, I adapted filter sizes and activation functions (tanh in the original LeNet-5 vs. ReLU in the alternate version) to assess differences in performance.

- I maintained a binary output layer using sigmoid activation in all models to align with the binary classification problem.

3. Model Evaluation

After training, I evaluated each architecture on the test dataset using standard metrics (accuracy, confusion matrix, and a classification report). My findings were as follows:

- **Performance:**
  The alternate LeNet-5 CNN outperformed both the Dense ANN and the original LeNet-5 in my experiments, consistently achieving the highest test accuracy. This indicates that a deeper CNN with ReLU activation was better at capturing the necessary spatial features required for this task.

- **Confidence:**
  While the final accuracy was encouraging, certain classes (e.g., damaged images) required special attention due to imbalance. To address this, I incorporated data augmentation and considered class weights for further improvement if necessary. Overall, the best model (alternate LeNet-5) demonstrated robust performance on the test data, which gives me a good level of confidence in its ability to generalize, though I remain open to further tuning if additional data or challenges are encountered.

4. Model Deployment and Inference

Deployment Approach:
For model deployment, I developed a simple inference server using Flask. This server loads the pre-trained best model from disk (saved as saved_model.h5) and exposes two endpoints:

- **GET /summary:** Returns JSON metadata about the model (name and description).

- **POST /inference:** Accepts a binary image file (either via multipart/form-data or raw binary) and returns a JSON response with a key "prediction" set to either "damage" or "no_damage".

Packaging & Dockerization:
I wrote a Dockerfile that packages the Flask application along with the required dependencies (as outlined in requirements.txt) and the saved model. The Docker image exposes port 5000 and is built for x86 architecture (recommended to build on my class VM). I pushed the image to Docker Hub under the repository aasthaagrawal10/projectthree:v1.

Docker Compose:
A docker-compose.yml file has been provided so that the inference server can be easily started and stopped. Running docker-compose up launches the container on port 5000 and makes the server accessible via http://localhost:5000, while docker-compose down stops the server.

Usage Examples (also included in the README):

To query the model summary: curl http://localhost:5000/summary

To make a prediction with a test image: curl --form "image=@test_image.jpg" http://localhost:5000/inference

Overall, my inference server is now ready for deployment. The provided Docker image can be pulled from Docker Hub, and using Docker Compose simplifies managing the container lifecycle. This completes the model deployment and inference portion of the project.

ChatGBT Help: During this project, I encountered significant challenges in understanding and using Docker for containerizing my inference server. I have never used docker in my ECE Degree Plan yet so I was struggling. I initially struggled with building Docker images, configuring docker-compose for proper port mapping, and troubleshooting various Docker-related errors, which slowed my progress considerably. To overcome these obstacles, I leveraged ChatGPT for detailed guidance and code examples. ChatGPT provided me with step-by-step instructions on writing a Dockerfile, setting up a docker-compose.yml file, and resolving issues related to image builds and port conflicts. This assistance enabled me to successfully build, deploy, and push my Docker image to Docker Hub, ultimately ensuring that my model inference server met all deployment requirements for the project.