Data Preparation

The foundation of successfully training the machine learning models all stems from the data preparation of the training set. The data set provided to us was already classified into two folders: damage and no damage. Consequently, loading the data into Python data structures was relatively simple. The data was simply transformed into multidimensional arrays and appended to the independent variable array. Simultaneously, depending on where the image came from, the dependent variable array or the "classes" was updated with the respective class of the image. Here to work with numeric values, I assigned damaged images the value of "1" and undamaged images the value of "0". From there, the data was free to be split into train and test sets using a 0.2 test size split.

What was noticeable was that the images were originally colored. For the first attempt to split the data, the server ran out of memory very quickly. Consequently, another change had to be made to the image before it could be split. Because the images were originally RGB, that would require each image to be described as a 3D matrix, requiring a lot of computational power to process. Consequently, as the images were loaded into Python data structures, they were also converted into grayscale images, which are much less computationally intense.

After splitting the data set, more preparation was done to ensure the optimal performance of our models. First, for the ANN, I flattened the input into a 1D array to be fed into the Dense architecture. Lastly, all of the values of the images were normalized by the maximum pixel value of 255. This way, all the pixel intensities were scaled between 0 and 1, ensuring that each feature contributed proportionally to the training process. By the end of the data preparation process, the dataset was optimized for model training.

Model design

For the Hurricane analysis classification model, three different models were explored to find the optimal model for overall applications. The three models explored were a fully Dense ANN with one hidden layer, a LeNet-5 CNN Architecture, and an alternative LeNet-5 CNN Architecture originating from a research paper.

Dense ANN

The first model we experimented with and explored was the dense ANN with one hidden layer. Although this three-layer ANN was coded to run 30 EPOCHS, it only ran 8 EPOCHS due to early stopping. The condition we decided to choose was a stagnating validation loss to help keep our model from overfitting.

LeNet-5 CNN

The next model we implemented was the base LeNet-5 CNN with the first layer being 6 filters with a size 5x5. From this, we then implemented another convolutional layer with 16 filters of size 5x5. By implementing these layers, we are able to understand specifically the spatial relationships and translational variance. From this, we then flatten our vectors and pass them into a normal dense ANN with 4 layers. We slowly reduce the number of perceptrons as we go through the layers to help the model understand patterns and, in the end, identify whether a building is damaged or not. For this model, the model was trained for 7 EPOCHS due to early stopping. The total EPOCHS this model was supposed to run was 15, though our validation loss was going up more than our allowed error, resulting in restoring our model to the previous iteration.

LeNet-5 CNN Alternate

The last model we implemented was the LeNet 5 CNN based on the research paper. The first convolutional layer consists of 32 filters, with each filter being 3x3. The next layers consisted of 64 filters, 128 filters, and lastly 128 filters again. Each of the filters in the layers was also 3x3. In between each convolutional layer, a max pooling layer was implemented, being the same number of filters as the convolutional layer, with each filter being 2x2. The reason this paper, along with our CNN, increases the number of filters as we go across the CNN is because of the basis of how CNNs understand patterns. Earlier in the model, simpler patterns are understood, which is why the number of filters is relatively low. When you go further in, though, more specific and abstract features are understood, which are captured by more filters, maximizing the use of the CNN. The vectors are then flattened and sent into a normal Dense ANN consisting of 4 layers, including a Flattening layer, a Dropout layer, a hidden layer, and an output layer. By implementing a Flattening layer, we can take our 2d matrix from the CNN filters and turn it into a single vector. The use of the Dropout layer helps our perceptrons not "memorize" training data and not rely heavily on one another. In this way, we are able to accurately make sure all perceptrons pick up the pattern. The next layers follow the same architecture as the normal ANN.

Model Evaluation

Dense ANN: The results show a 65% accuracy on our validation training set until it ended up stagnating, meaning the model found it hard to understand the patterns that differentiated damaged buildings from undamaged buildings. This could be because of the spatial relationships that ANNs inherently disregard when flattening the vectors.

LeNet-5 CNN: When training this model, the associated accuracy and loss were 91.44% and .2294. We feel that this is a good value, though it could be optimized slightly by a different architecture, as seen later on.

LeNet-5 CNN Alternate: When training this model, we set the model to run for a maximum of 15 epochs with an early stopping condition based on our validation loss. This model was trained for 8 full EPOCHS and stopped on the 9th EPOCH. When trained on our test set, the model produced an accuracy and loss value of 95.73% and .109.

Confidence and Best Model

The LeNet-5 CNN Alternate model proved to provide the best results for our test set. We think the use of more filters, combined with being a smaller size, helped the perceptrons later on learn better patterns. Also, we think the use of 4 convolutional layers provided the model with a better understanding of the spatial information. Though the number of trainable parameters for this model is greater than the other models, making the number of computations exponentially higher, we believe that the tradeoff for accuracy outweighs the computational workload. The model achieved a validation accuracy of 95.4% with a low and stable validation loss, demonstrating strong generalization and minimal overfitting. Early stopping at epoch nine prevented unnecessary training while preserving the model's optimal state. Furthermore, the close alignment between training and validation accuracies indicates consistent learning. Consequently, these results make us confident in the model's reliability.

Model Deployment and Inference

The trained convolutional neural network was deployed through a Flask-based inference server, containerized using Docker. The required dependencies are installed during the image build process through the Dockerfile. Once the container is built and executed, the Flask server launches, and users can curl the localhost to access endpoints and make requests.

To deploy the server, you must pull the image made for this model, "jyl2027/hurricane-api" from Docker Hub (docker pull jyl2027/hurricane-api:latest). Next, you must start the container using "docker compose up -d –build". Now you are able to access the main endpoints. The GET /summary endpoint returns metadata about the model, including its architecture, input and output shapes, number of parameters, and loss function. The POST /inference endpoint accepts an image path and returns a classification label indicating whether the image contains "damage" or "no damage." When finished with inference, users should use "docker compose down" to remove the container. Overall, deployment and inference are performed using HTTP requests. Users can query the model summary or obtain predictions.

Two commands can be used to curl to access each respective endpoint. "curl localhost:5000/summary" and "curl -X POST -F 'image=@damage/example.jpeg' localhost:5000/inference". For the POST, please replace "damage/example.jpeg" with the location of the image you want to identify.