Aditya Pulipaka

COE379L – Stubbs

11/14/2025

Project 2 Report: Accurate Hurricane Damage Classification

Data Preparation:

We were provided with an extensive number of images, prelabeled as damaged or undamaged buildings. These images were stored in two separate folders, and we were not given any input dimension. The first thing I did, therefore, was a little bit of exploratory analysis on the input dimensions of our data. I printed the shape of a single jpeg image out of the entire dataset to get an initial idea, which returned 128x128. I then assumed regarding the dimension of the rest of the data and moved forward with creating a TensorFlow dataset that would optimize data loading for the models I would later train. This part was heavily aided by AI, since my first trial of data loading was using the python 'OS' module to list all files in each directory, then load them using pillow. After a quick session with Claude, I settled on using the built-in TensorFlow image_dataset_from_directory module that allowed for streamlined access to all images. I then used dataset.map() with a normalization layer that rescaled pixel values from 0-255 to 0-1. Finally, I divided the dataset into a 70% training, 15% evaluation, and 15% testing split, allowing for reasonable evaluation data while also maintaining a reliable testing sample.

Model Design:

1. ANN:

I first explored ANN architectures, working from the examples done in class, such as MNIST clothes classification. Building the ANN required first flattening the data, which was done using a Flatten layer from TensorFlow, rather than reshaping as done in the class examples. This was a necessary change given that I was using a built-in TensorFlow dataset rather than a NumPy array. In addition, considering this was a binary classification task, I needed to change the output dimension of the model, which was previously a categorical output with length 10 in all the class examples. A google search yielded that a single output node with a sigmoid activation function would give best results for binary classification. Additionally, compiling the model with the binary_crossentropy loss function would yield the best results for training. In all, the model consisted of ~407 million parameters with the following layers:

1. Flatten layer
2. Fully connected layer of dim (8192,), input dim (128*128*3,), activation ReLU
3. Fully connected layer of dim (512,), activation ReLU
4. Fully connected layer of dim (1,), activation Sigmoid

This model had a reported test accuracy of 71.37%, setting a pretty low bar for the following models. To elaborate on some of the decisions that were made, I set the output dimension for each layer to be some power of two that seemed like a decent stepdown from the previous dimension and limited the model to two internal fully connected layers, then optimized based on validation accuracy.

2. LeNet-5 Vanilla:

This model didn't have a flatten layer following the convolutional layers, as it follows a CNN architecture and was implemented exactly as detailed in our class notes (and online):

1. Conv2D layer 5x5, input dim (128,128,3) output dim (124,124,6), activation ReLU
2. AveragePooling layer 2x2, output dim (62,62,6)
3. Conv2D layer 5x5, input dim (128,128,3) output dim (58,58,16), activation ReLU
4. AveragePooling layer 2x2, output dim (29,29,16)
5. Flatten layer, output dim (,13456)
6. Fully connected layer, output dim (,120), activation ReLU
7. Fully connected layer, output dim (,84), activation ReLU
8. Fully connected layer, output dim (,1), activation Sigmoid

This model achieved 94% test accuracy, a significant improvement over the initial ANN.

3. LeNet-5 Variant

Some changes were implemented following the research paper by Cao and Choe linked in the project description. These included changing all convolution layers to 3x3 convolution, changing all pooling layers to MaxPooling (still 2x2), and adding two convolution-pooling pairs before flattening. After flattening, a dropout layer (0.5) was added followed by a single fully connected layer with output dim 512 and an output node, totaling 2.6 million parameters. The activation functions remained unchanged. This modified version of LeNet-5 performed significantly better than the vanilla version, achieving 97.23% test accuracy.

4. LeNet-5 Full Dropout

In the same research paper, the authors detail a version with full dropout, which further prevents overfitting. I implemented this model by adding a 0.25 dropout layer after each convolution-pooling pair. This model achieved 97.85% test accuracy, showing a further improvement (although slight) over the vanilla model.

5. VGG16 with only output training

I wanted to go ahead and try the VGG16 model, so I followed some documentation online to use the base VGG16 model (in a frozen state) with some augmentations:

1. vgg_base, input dim (128*128*3,), output dim (,4,4,512)
2. Flatten layer

3. Fully connected layer, output dim (256,), activation ReLU
4. Dropout layer
5. Output layer, activation Sigmoid

This model performed quite badly compared to our LeNet5 alternate implementation, with only 93.61% accuracy.

6. VGG16 full training

When the full VGG16 network was exposed to training, the test accuracy improved to 98.25%, leading to its selection as the best of the 6 models tested.

Model Evaluation:

As mentioned above, the VGG16 model performed the best, and with an accuracy of 98.25%, I have quite good confidence that if we could not directly use the model for identifying building damage with absolute certainty, we can at the very least implement a different training scheme or a decision function that would prioritize recall or precision, based on the desired sensitivity, and achieve near 100% confidence in the desired metric.

Model Deployment:

This model was deployed using Flask to host the inference server. Two endpoints, /summary and /inference are served, which accept GET and POST requests, respectively. The server, along with the VGG16 variant model that was settled upon, were packaged into a docker image with flask, TensorFlow, and pillow libraries installed, and pushed to docker hub as adipu24/project02:latest.

To run, simply run “docker-compose up” in the directory with the docker-compose.yml file, as detailed in the README in the part3 directory.

POST requests to the /inference endpoint must be formatted as binary streams in the files attribute of the POST JSON, as is also detailed in the same README.