COE379L *Joseph Suess*

Project #2 *Siddhant Singh*

I. Data Preparation

The first step in preparing the data was to perform basic exploratory analysis. This involved going into the data repository on GitHub to analyze what the images looked like. On initial inspection: The images appeared to be from Hurricane Harvey, with some showing hurricane-based damage while others did not. This yielded the initial observations that this was a *supervised, binary classification problem*.

Next, the images were downloaded into a folder on the VM. Approximately 14,000 damage images were observed compared to the 7,000 non-damaged images. This alone constituted a major preprocessing choice: To reduce the dataset to choose random damaged images, and eventually produce a random subset of the model such that the damaged images equaled the non-damaged. A shuffle of the dataset was performed afterwards.

Each image was observed to be 128x128. Because of project constraints, this served as the final input dimension, meaning this could not be modified later. This would later be a significant drawback in both the LeNet as well as the ANN, which will be discussed shortly. The output vector should also only be [0, 1] representing 0 → Damage and 1 → No Damage.

II. Model Design

The first model that was constructed was the fully-connected, or dense, Artificial Neural Network. Initially, the obvious choice was to flatten the image into a 1-D input for the Dense layers. This yielded 128x128x3 features, or ~49,000 features. I opted for 3 Dense layers, of gradually decreasing perceptrons with the ReLU non-linear activation function. The amount of neurons in each layer yielded the pattern [512→256→128]. One of the main drawbacks of not being able to change the input size, meant that it was difficult to add more layers and/or more neurons.

I briefly tested a 32x32x3 image set, which performed significantly better than the 128x128x3 original dataset to prove this theory. Memory was no longer a bottleneck, and the reduction of features enabled the ANN to achieve better validation accuracy.

However, not much else was tested with the ANN, primarily due to the architecture itself. The biggest issue with the ANN was neither memory, nor features, rather it was the lack of spatial awareness. Flattening the image destroyed any spatial relationships within the image itself, so the model, in essence, had no knowledge on each image itself except for a flattened 1-D representation.

The next architecture explored, and an improvement, nonetheless was the LeNet-5 Architecture. Basic philosophy about the model was to use small convolutional layers first to capture low-level features, texture and edges. I decided on using 2 pooling layers, of 2x2 dimension each, to reduce spatial relationships. The idea was once we applied a basic texture extraction, we could then use an ANN-like structure with fully-connected dense layers to perform classification on extracted features. I found that before the sigmoid classification that these layers, 120 and 84 were semi-ideal amounts of perceptrons, respectively for these dense layers.

The last, but most important architecture explored was the customized one from *Cao, Quoc Dung, and Youngjun Choe.* Their architecture featured a custom-CNN, with 4 original convolutional layers to capture not only low-level patterns, but also complex shapes and high-level shapes. 4 Pooling layers were also utilized, as well as a steep dropout of 0.5 towards the end before 1 dense layer of 512 perceptrons. Because this model was highly specialized for this dataset, having a dropout of 50% of neurons helped immensely with the issue of overfitting that plagues highly-custom models like this one.

Each model was trained with a batch size of 16 and a maximum of 50 epochs. When validation accuracy begins to slow-down, the learning rate was automatically adjusted, but more importantly, when the accuracy hit a plateau, the model stopped training. Meaning, all 3 models trained for 10-20 epochs on average, despite having that upper maximum of 50.

III. Model Performance

The model that performed the best was by-far the customizable one from the research paper as discussed previously. From the first epoch, both training and validation accuracy started reasonably high, as both were above 80%. As the training neared almost perfect, validation continued to climb, which is rare in models where the training data hits close to 100%. This would suggest that the architecture was continuing to learn, and the graph in the notebook Part II highlights a thoroughly well-trained model with a validation accuracy at almost 98%. I am extremely confident in the performance of that model, much more so than LeNet-5 and certainly more than the ANN, which was almost attributable to random guessing.

The dense ANN had serious trouble choosing the correct weights at the very start of Epoch 0, and I found that the ANN only performed somewhat well (around 60%-70% validation accuracy) when it had chosen a decent weight at the start of the iterations. More often than not, and included in the notebook, is an example where the model did not choose weights close to correct, and as a result, did not perform better than random guessing. This was expected considering the input dimension constraints as well as the memory constraints on the VM.

It should be noted that the LeNet-5 did learn much better. The curve shows a gradual increase in the training data performance, however, the validation accuracy did not improve much throughout the epochs it was trained on. Still, the end result was close to 80%, and the LeNet-5 did a proper job of establishing good baseline performance from the beginning.

IV. Model Inference and Deployment

The model deployed was the customized-CNN from the research paper. It was deployed through the terminal in the COE379-VM. Deployment was completed through setting up a Flask script that served a localhost endpoint. Endpoint /summary contains a brief overview of the model itself, with attributes such as input_shape as well as the various layers that went into creation. Deployment was fairly straightforward once I understood what to do, and after testing a local copy with the grader_script, I uploaded a docker image of the Flask server and handcreated the docker-compose file to streamline the deployment process.

To use the model for inference is very simple. I describe in the readme about having first to curl an image from the dataset, though it can be any image on a local computer. For example, a curl request on this endpoint yields an example of a non-damaged image. Once the server is running after performing **docker-compose up**, a curl request such as

```
curl -X POST http://localhost:5000/inference -F "img_path"
```

will return a JSON attribute containing a "prediction" field that is either "damage" or "no_damage" depending on what the model suggests from the CNN.

Server-side, there is not much validation taking place for this to happen. All that occurs is the image is converted into RGB and that array is the input to the model, which returns a sigmoid function whether there is damage[0] or no_damage[1].