I. Data Preparation

Data preparation for Project 2 was relatively straightforward, as more emphasis could be placed on the neural networks to learn relevant patterns in the data as compared to the classical machine learning methods of Project 1. Still, I had to try out a few different methods in order to get my models to run correctly.

In my first attempt at data preparation, I simply converted all of the image files into numpy arrays to be fed through the network for training and inference. When my kernel crashed due to memory constraints, just 2 epochs into training my first ANN, I was forced to re-evaluate the data structure that I was using. From here I decided to try out a tensorflow based data structure because I figured this might be most compatible with my tensorflow based models. After making this adjustment my networks trained much more smoothly, and so I settled here.

At this point, I realized that I should print out some of my images in order to get an understanding of what the differences between damaged and undamaged houses looked like. The primary difference that stuck out to me was that the undamaged houses often shared a uniform roof color (the color of shingles) and appeared to have sharp edges at their boundaries where the dark colored houses or the shadows of the houses met the green grass. On the other hand, the leftover debris from the damaged houses was often much less separable from the natural terrain and was often discolored (in one image, white chunks of a house that had been torn up were scattered around in the yard). From these observations, I concluded that a house that has been torn up by a hurricane is going to be much more discolored than an intact house with a uniformly colored roof. Thus, I decided to encode the original colors of each image into the data as they seemed to be an important feature for classification.

The final steps of compiling my datasets were to create a tuple for each sample containing the image data and the label, normalize each image sample by dividing each pixel intensity by 255, shuffle the dataset, split the dataset into a trainSet, valSet, and testSet, and finally batch each of these sets. For my test, train, validation split, I wanted to send a large majority of my data into my train set in order to build a model that would perform well on unseen data. My next priority was to accurately represent to myself how each model that I trained might perform on unseen data, so that I could confidently choose the best model. That left the validation set size as my last priority; I needed my validation set to provide insight into the overall trend of model training, but didn't need it to tell me exactly how well my model would perform on a new dataset. I ultimately chose to direct 75% of my data into my trainSet, 10% into my valSet, and 15% into my testSet. Additionally, I chose to batch my data using a batch size of 32 because I felt that this gave me a good balance between finding an optimal convergence and not exceeding the memory limits of my VM.

II. Model Design and Evaluation

My model architectures were constrained by my input data size (128x128x3) and the number of classes that I was trying to predict (2). Considering that I was doing binary classification, I decided to use a sigmoid activation function in the last layer of each of my neural networks and I decided to train each of my models using the binary crossentropy loss function.

The first model that I experimented with was the very first ANN that we built in class which simply contained 4 perceptrons in the first layer and 128 in the second. This model performed horribly, but caught my attention from the fact that its validation accuracy would perfectly stabilize after 3 epochs. After consideration and research, I determined that this was due to the bottleneck of only using 4 perceptrons in the first layer. The 49,152 dimensional input signal was being killed when it was being condensed to just 4 dimensions. The adjustment that I made was to use 256 perceptrons in the first layer and 128 in the second. This model performed better but still only achieved 69% accuracy.

I noticed that the dataset was imbalanced (66% of the images were of damaged houses). In an effort to build a better model, I trained my next ANN using the same architecture of the previous one but optimizing for AUC instead of accuracy. Somewhat surprisingly, this model achieved a 74% accuracy, 5% better than the first ANN. I was curious whether I could improve this result by changing the decision threshold, so I experimented with moving it around and found that I could achieve 74% accuracy with a threshold at both 0.5 and 0.4. While I found this interesting, I felt that these results were still poor and I decided to move on to a more robust CNN.

The first CNN that I built used the LENET5 architecture presented in class. I trained this model using the adam optimizer with a learning rate of 1^{-3}. This model achieved 91.9% accuracy when the decision threshold was set to 0.5, and 91.2% accuracy when the decision threshold was set to 0.6. I decided not to do much else with this model, as I suspected that the next CNN architecture, which was designed for the exact same classification task, would perform much better.

The next CNN architecture that I tried was based off of the architecture found here, with the only difference being that I changed the input size from 150x150x3 to 128x128x3 in order to better suit my data. This model performed very well, achieving about a 98% accuracy when evaluated using a decision threshold of 0.5. And scoring about 0.2% higher when evaluated at a decision threshold of 0.53. Unfortunately, at this point my VM exceeded its memory limit and my kernel crashed. Thus, I lost these testing records, hence the reason why I'm referencing these scores using the word about. When I re-ran my notebook, this CNN, achieved a 98% accuracy when evaluated using a decision threshold of 0.5, and showed improvement when evaluated at a decision threshold of 0.4. Due to the difference between this result and my pre-crash result, I determined that manipulating the decision threshold led to noisy results, and I decided to simply use a decision threshold of 0.5.

The results that I got from this architecture were much better than any of my previous results, so I decided to start playing with the hyperparameters of this model in order to see whether or not I could do any better. I felt confident in my selection of binary cross entropy as my loss function due to the fact that I was solving a binary classification problem and I felt confident in optimizing for accuracy due to the fact that this would be the metric that my project would ultimately be evaluated on. Thus, I decided to simply try to find the best optimizer to use during model training. My search included adam optimizers with a learning rate of 1^{-3} and 1^{-4}, and SGD optimizers with learning rates of 1^{-3}, 1^{-2}, and 1^{-1}. The adam optimizer with an lr of 1^{-3} achieved 98% accuracy, the adam optimizer with a lr of 1^{-4} achieved 97.3% accuracy, the SGD optimizer with an lr of 1^{-3} achieved 95.8% accuracy, the SGD optimizer with an lr of 1^{-2} achieved 97.8% accuracy, and the SGD optimizer with an lr of 1^{-1} achieved 66% accuracy. I elected to use the CNN which used an adam optimizer with an lr of 1^{-3} in my final product as it had performed the best.

III. Model Deployment and Inference

My model can be deployed by cloning my repo and simply running “docker compose up” from the same level as my docker-compose.yml file. The model will be served at port 5000, and once deployed, a model summary can be retrieved using the GET /summary endpoint, and inference can be run using the POST /inference endpoint. The inference server will respond to POST requests with a JSON object that says either { “prediction”: “damage”} or { “prediction”: “no_damage”}. The GET endpoint can be tested using “curl http://127.0.0.1:5000/summary” and the POST endpoint can be tested using “curl -s -X POST http://localhost:5000/inference -F “image=@$(pwd)/path_to_the_image_from_pwd.jpeg”. Additionally, POSTs to the server can be made from python using a command along the lines of rsp = requests.post(“http://172.17.0.1:5000/inference”, files=data), where data = {"image": open(path, 'rb')}, and GET requests can be made using rsp = requests.post(“http://172.17.0.1:5000/summary”). Note: when I first tried to run start_grader.sh, I received an “invalid file reference format” error. I was able to fix this by changing the command from “docker run -it --rm -v $(pwd)/data:/data -v $(pwd)/grader.py:/grader.py -v $(pwd)/project3-results:/results --entrypoint=python kmw4568/ml-satellite-api /grader.py” to “docker run -it --rm -v "$(pwd)/data:/data" -v "$(pwd)/grader.py:/grader.py" -v "$(pwd)/project3-results:/results" --entrypoint=python kmw4568/ml-satellite-api /grader.py”. A similar adjustment may need to be made on the graders side in order to get my server to run.