COE Project 3

Aidan Liu, Arabel Rachman

Professor: Joe Stubbs

Data preparation: what did you do? (1 pt)

For our data preparation, we prepared it in four steps. Firstly, We made sure that the code was clean and ready to be used by the models. This was done in the Jupyter Notebook where the first cell has shutil.rmtree("data/train") and shutil.rmtree("data/test") inside of a try/except function. The functionality of doing this makes sure that previous runs are cleaned and that executing a new run runs in a clean slate. This also means that there are no train/test leaks. Then we rebuilt the folder hierarchy by putting the folders with data within the damage and no_damage folders. Secondly, we decided to use splitting. This was done because the notebook takes in the data using os.listdir('data/damage/damage') and os.listdir('data/no_damage/no_damage'), collecting 14 170 damage images and 7 152 no-damage images. We used Python's random number generator to ensure that the generated numbers were fair and reproducible. Then we split the data into 80% training data and 20% testing data. We used 2 lists which were mutually exclusive and printed counts which showed that the split ratio was indeed correct. Thirdly, we copied each of the new images using shutil.copyfile(src, dst). We copy the image and afterward we visualize 5 random examples from data/test/damage and data/test/no_damage using matplotlib. Fourthly, we enter the data that was prepared into TensorFlow. We do this through tf.keras.utils.image_dataset_from_directory('data/train', validation_split=0.2, subset='both', image_size=(128,128), batch_size=32) which will shuffle all the data and come up with 2 Dataset objects. We then map these 2 objects through a Rescaling(1./255) layer. In these 4 steps, the Jupyter notebook can transform the raw data of satellite tiles into datasets that the models can process.

Model design: which architectures did you explore and what decisions did you make for each? (2 pts)

In this project we explored 3 architectures given to us in the project folder: A dense (i.e., fully connected) ANN, the Lenet-5 CNN architecture, and the alternate-Lenet-5 CNN architecture. We deployed each one in a Jupyter notebook and came up with the following conclusions. Starting off with the dense ANN baseline, we used the function *Sequential([Flatten(), Dense(256,'relu'), Dropout(0.3), Dense(128,'relu'), Dropout(0.3), Dense(1,'sigmoid')]);* in order to create a network which ignores spatial locality. It compiled in just 0.4 seconds but the test accuracy was very low at 73.7%. The result that we got for this architecture made us come to the decision that we need to confirm the pixel order information, which proved to be crucial for reliable detection in rooftop damage for this dataset. Nextly, for the classic Lenet-5 CNN architecture which has 2 3x3 convolutional layers. With our set parameters, we trained the model for 40 epochs using the function RMSprop(lr=1e-4) and achieved 92.1 % accuracy. This 17% jump validated our initial decision of using weight sharing for the pixels. For now this was the best accuracy that we had achieved. However, the last architecture that we tried out, the Alternate-Lenet-5 architecture surpassed the previous 2 architectures. According to the paper that was linked to this architecture in the project folder page, it detailed the architecture having 4 convolution pairs, Conv2D(32), Conv2D(64), Conv2D(128), and Conv2D(128). Each of the 4 pairs with 3 × 3 kernels and activations. This deeper variant of the classical Lenet-5 architecture would use a hyper parameter grid search. As for the results, over 40 epochs of testing this model had a 97.1% test accuracy, 0.97 precision/recall, and 0.99 ROC AUC. This model far outperformed the original Lenet-5 architecture by 5%. We used the function, model.summary(), across all the different models to keep track of the parameter counts and an epoch matrix to monitor the testing accuracies.

Model evaluation: what model performed the best? How confident are you in your model? (1 pt)

When we compared all 3 architectures using the test set we can clearly see that the alternate-Lenet-5 was performing the best with 97.1% overall accuracy, 0.97 precision, 0.97 recall, and a 0.99 ROC AUC. This performance beat the standard LeNet-5 with

92.1 % accuracy, 0.97 AUC. Both the Lenet-5 models outperformed the dense ANN baseline with 73.7 % accuracy, 0.79 AUC. The confusion matrix showed only 96 false negatives which could be contributed by damaged rooftops that were mislabeled as intact, and 25 false positives. This translates to a F1-score of 0.97. Through a manual look through the 200~ mis-classifications in the images we found that most of the false negatives occur with trees that obscure the rooftop while the false positives were from other factors that maybe could have taken place (according to me and my partner's hypothesis). Given that the dataset was pretty big, balanced, had minimal over-fit, and a tight cross-validation spread, we are mostly confident that the model that we have chosen will sustain the accuracy for the tester.py file for future hurricane imagery captured by the same satellite platform.

Model deployment and inference: a brief description of how to deploy/serve your model and how to use the model for inference (this material should also be in the README with examples) (1 pt)

We deployed our model using a Flask-based REST API served within a Docker container. This allowed the model to be hosted independently of the Jupyter notebook that it was trained on. We did this by creating a Dockerfile that hosts the best performing model. Pushing the Docker to Docker Hub allows it to be launched using the command docker-compose up. Once deployed, the model listens on port 5000 and it has two endpoints. The first endpoint, /summary, responds to GET requests and returns metadata about the model, including its architecture type, input and output tensor shapes, and the parameter specifications. The second endpoint, /inference, accepts POSTS requests containing the image file for the model to predict. Upon receiving an image, the model returns a JSON response, classifying the image as damage or no_damage. This design allows the model to be fully deployed. This allows the model to be easily accessible for inference using standard HTTP requests. The mode is able to be easily fed with images to predict, and it returns its predictions in a straightforward way.