Chris Deng (cmd4923), Jason Lee (jl78928)

Data Preparation

For the data preparation, we mostly did it as we imported the data. We actually loaded the data twice because the ANN and CNNs needed different data formats. We definitely could've only imported it once, but we felt it was easier for us if we did it twice.

For the ANN data preparation, we loaded each image as a 128x128 image, making sure that it was the size of the loaded image. We then turned the image into an array and divided it by 255 to make sure that all of the data was scaled correctly according to the RGB values, though, since there were really no other features, this might be unnecessary. We still felt it was good to do, though and a best practice. We then flattened the image array for input to the ANN. We also looked at what we stored just to be sure and checked the max and min values just to be sure everything was scaled correctly. We also looked at the number of arrays we had and made sure they matched the number of images we loaded. We also double checked the pixel size just to be sure the original images were actually 128x128.

Next, for the ANN data preparation, we say that the labels were actually objects for damaged and not damaged, so we turned those into strings and then did one hot encoding on them (dropping the first column) so that they turned into binary values (0 or 1). We made sure to note that damaged was 1 and not damaged was 0. We also made sure there were no missing values and looked at the first few rows of the data to make sure everything looked good. Lastly, we split the training data into training and testing sets, making the test size 0.3 and making sure to use stratify by Y and make sure the class distribution was preserved.

For the CNN data preparation, we did everything exactly the same as the ANN preparation, except we didn't flatten the array, but kept its 128x128x3 shape and made sure we double checked the shape.

Model Design:

We wanted to make sure we could see the base performance of each model before we decided to tune any of the models.

For the ANN, we just went with what we thought was a basic version with very arbitrary numbers for perceptrons. We honestly didn't think it would do that well compared to the CNNs, so we didn't put a ton of time into it.

For the regular Lenet-5 CNN, we just took the parameters from the already developed model. We then moved on to the alternate Lenet-5 model, and then never came back to try to tune the model because the alternate model did a lot better.

We did a lot more testing for the alternate Lenet-5 model because it did the best out of the box. The first thing we did was read the paper. It seemed like the data we were using to train our models was the exact same type of data as the paper, and the model was being used for the same purpose. One thing we did note was that the authors of the paper filtered out good and bad training data, such as those where clouds made up the whole image. We wanted to double-check our data and do something like this, but we ultimately ran out of time and never ended up doing this. Other than that, we felt the data was extremely similar to the one used in the paper, with the main difference being that the images in the paper were cropped to 150x150 images, while our images were 128x128.

To try to improve our model based on these known differences in the data we tested lowering the dropout, since our smaller image would mean less parameters, meaning dropout would theoretically be needed less. We tested dropout in increments of 0.05, ranging from 0.3 to 0.5. We found that a dropout value of 0.4 seemed to work marginally better, though a lot of it was due to random chance in how the model trained. We also tested training for additional epochs and changing the size of the filters in the CNN, but none of those chances seemed to help. Reading the paper, they trained it for 10 epochs because they found the model started to overfit after that, which confirmed what we found as well. Ultimately, as the original model did extremely well, we were only able to find one change (changing the dropout to 0.4) that actually improved the accuracy. That is the model we went with.

Model Evaluation:

We evaluated our models by using a confusion matrix mainly. Since there are only two classes and 4 different sections for the confusion matrix, we felt that it was a lot easier to interpret compared to the classification report, which had a variety of metrics that we felt were a little harder to interpret. But for the most part, we just evaluated using accuracy.

The reason we used accuracy is that once we started trying to tune the alternate Lenet-5 models, the number of misclassifications was already pretty low. Maybe it was chance, but both misclassification sections in our communication matrix were under 100 in the original alternate Lenet-5 architecture. Since that's already pretty good performance/, we thought accuracy captured enough of the data, to where we didn't need to worry about class biases too much. In addition, we looked at the confusion matrix mainly, so any biases in that regard would be caught very quickly. We are assuming that the final holdout test set used for evaluating our model will have the same class ratios as our given data, so accuracy would be a good metric in that case.

As stated above, the model that performed the best was the original alternate Lenet-5 model with just the dropout changed to 0.4 instead of 0.5. I would say that we are very confident in our model because through multiple iterations of this model, we have found it to be very robust and do very well on this dataset, as it was designed for essentially the same task.

Model Deployment and Inference:

For model deployment, we decided to create a Flask API and containerize it using Docker so that anyone could easily run our model without having to worry about dependencies or environment setup. We saved our best performing alternate Lenet-5 model which also had a dropout layer of 0.4, and built it into a service that could return predictions.

We created two endpoints for our API. The first endpoint is a GET request to /summary that returns information about the model, like what architecture we used and basic model details. We thought this would be useful for anyone trying to understand what model they were using. The second endpoint is a POST request to /inference that accepts an image file and returns a prediction. This is the main endpoint that actually does the damage classification.

For the inference server, you can send requests to it using curl, after the server is started. We can start the server by using docker run and clearing out any conflicting running dockers with docker ps and docker stop. In another terminal, we can then use the curl commands. This is how the GET request summary looks like:

```
curl localhost:5000/summary
```

This is an example of a POST request:

```
curl -X POST -F "image=@damage/-93.795_30.03779.jpeg" localhost:5000/inference
```

After the POST request, the API returns a prediction as a JSON response with either "damage" or "no_damage". We then used the test grader provided to make sure the format is correct.