Part 1: Data Preparation

To load the data, I created train and test directories for each label: damage and no damage. I then collected the paths for the images in each label and split the paths into train and test categories by using 80% of the images to train and 20% to test. I checked to see if there were overlaps between the splits and affirmed that there were none. I then copied the files into the train and test directories. After moving all the image files into train and test folders, I went ahead with pre-processing so that they could be used for the training models. I selected the target size for each image to be 150x150, since the images come in different sizes and this would make them uniform and make processing easier. I then created a TensorFlow Dataset for both the training datta and testing data from the image files in the directory. Each image had labels which corresponded to the directory that it was in. After this I rescaled each image by 1/255 since the pixel values are of range [0,255] to [0,1].

Part 2: Model Evaluation

The first model architecture that I explored was the Artificial Neural Networks (ANN). I first made a flatten layer to reshape the input data into an array. After that, I did three layers. Since the input layer can have any number of perceptrons, I did 120. However, I made sure that the input shape matched the shape of the input (150*150,). For the hidden layer, I used 128 neurons. Both of these layers used the Rectified Linear Unit activation function. For the last layer, I used 2 neurons since there are 2 target labels to predict. For this layer, I used the softmax activation function. I explored the use of tanh and sigmoid activation functions since those are used for binary classification and softmax is often used for multiclass classification. When I did this, I also had to change the loss function to be “binary_crossentropy”. In this case, I found that the tanh function had a much higher loss in comparison to the other activation functions. When I attempted to use the sigmoid function, I also had to change the number of neurons to 1 since the output value is a number between [0,1] that represents the probability of being in one class. The sigmoid function performed relatively well, with an accuracy of 66.4% on the test data when I ran it. However, I ultimately went with the softmax activation function because it had a lower loss and higher accuracy. Overall the accuracy of the ANN model on the test data was 71%.

The second model architecture I explored was the Lenet-5 model. The input shape was (150, 150, 3). Again, for this model I explored the use of tanh and sigmoid activation functions. The tanh function had the worst accuracy out of all the activation functions. The sigmoid activation function performed about the same as the softmax activation function when I ran it, but had a very slightly lower accuracy. Ultimately, I went with the softmax activation function which had an accuracy of about 91% on the test data.

The last model architecture I explored was the Alternate Lenet-5 Model. For this, I followed what the paper explained. I did 4 2D convolutional layers, with 32, 64, 128, and 128 filters of size 3x3, respectively. The first layer had an input shape of (150, 150, 3). After each of the convolutional layers, I did a 2D Max pooling layer with a 2x2 pooling size. After these layers, I did a flatten layer which flattened the multi-dimensional output from the convolutional layers so that it could feed into the fully connected layers which need one-dimensional data. I did a 50% dropout to try and prevent overfitting. I did one fully connected layer with 512 neurons. Finally, the output layer had 2 neurons since there are 2 labels, and I used a softmax activation function. To compile, I used the RMSprop optimizer with an initial learning rate of 10^-4. I also made the loss function “sparse_categorical_crossentropy”. When I fit this model, it had the highest accuracy of all the models I explored. It had an accuracy of 98.3% on the test data.

Because it had the highest accuracy, the alternate lenet-5 model is the model I chose to continue on with. Since it was able to accurately predict 98.3% of the test data, I am fairly confident in this model's ability to classify images as "damaged" or "not damaged".

Part 3: Model Deployment

To deploy this model, you can first build it using the following command: "docker build -t niccoleriera/ml-hurricane-api ." To run it, use the following command: "docker run -it --rm -p 5000:5000 niccoleriera/ml-hurricane-api". Where I put my user (niccoleriera), you would put yours. There are 2 routes: a GET route and a POST route. The GET route outputs metadata about the model. To use this route, you would need to open another window on the VM and use curl to access the GET route: "curl localhost:5000/models/hurricane/v1". The POST route takes an image in the form of a JSON list, applies the model, and returns the prediction. To access the POST route, you need to use the jupyter notebook. Examples can be found in the README.md that is in this github repo. To test the server, I grabbed the first entry of the rescaled test dataset. When I applied the POST route, I got a list of 1x2 vectores. For example, if the first entry of the results was [0.999, 0.0002]. This means that it predicted a 0 for that entry. If the next was [4.26e-06, 0.999]. This means that it predicted a 1 for that entry.