

Project 2 Written Report

Sanika Nandpure, Melissa Huang

Data preparation:

Firstly, we visually inspected the data to understand how it was structured. We randomly sampled some images from the dataset and realized that a label of 0 means that the building in the picture is damaged, and 1 otherwise (we confirmed this upon realizing that the labels were based on the order of the folders in our dataset directory). We also printed the shapes of the image and found that the dimensions are 128*128*3. The image itself is 128*128 pixels, and we have 3 channels for RGB colors.

To pre-process the images, we normalized them to allow the neural network to learn faster.

We also split the dataset into a training and test set using the standard (70-30 split).

Model design:

We explored 3 main architectures: ANN, CNN, and the LeNet architecture. We explore the design decisions associated with each of these models below:

ANN:

We initially had the input layer to have the same number of neurons as the flattened image vector (128*128*3). However, we ran into an error in Jupyter where the memory allocation exceeded 10% of free system memory. We tried reducing the batch size to avoid using up memory; however, this did not fix the issue. Thus, we decided to reduce the size of the input image to be of dimensions 64*64*3.

- We acknowledge that there is a tradeoff here since resizing to a smaller image results in loss of information. However, we minimize this loss by using bilinear interpolation to resize the image (this is what `tf.resize()` uses by default as per the TensorFlow documentation). Bilinear interpolation replaces each pixel with the average of its four surrounding pixels; this preserves information since each pixel in the original image contributes something to the new pixel in the resized image.

We have three layers in the ANN, one input layer (of shape equal to the size of the flattened image vector), one dense layer with 64 neurons, and one output later (with 1 neuron since this is binary image classification).

CNN:

We implement CNNs with two different architectures.

- In the first architecture, we have 2 convolutional layers with 5x5 kernel size (each is followed by an `AveragePooling2D Layer`). We then have a `Flatten` layer followed by 2 Dense layers, followed by an output layer.
 - The values of the hyperparameters were arbitrary and mostly based on the values used during lecture.
- We then tried implementing an alternate CNN architecture with the same number of layers, but this time, different values for number of filters and kernel size. Firstly, we tried increasing the kernel size so that the network could capture a broader range of spatial features/context within the image. We also increased the number of filters in each of the convolutional layers to enable the

model to learn more complex patterns from the image. Finally, we trained this model on a higher number of epochs (10 as opposed to 5 for the previous models).

LeNet:

- We directly implemented the LeNet model from the paper.

Model evaluation:

The model that performed the best was the LeNet model, with an accuracy of 96.8% on the test set.

We are moderately confident that the model is correct because it has high accuracy on the test set; also, we tried running it on some pictures from the Internet of buildings after a hurricane (we cropped it to change the dimensions to be 128*128), and the results were reasonable. Here is an example of the image we used (the building is damaged, which the model predicted correctly).



Model deployment and inference:

Our docker image should be published under the name: spnandpure/ml-damage-api.

An external user can access this by pulling the docker image:

```
docker pull sanikanandpure/ml-damage-api
```

We decided not to include a docker-compose since the image simply contains one container, and a docker-compose is typically more useful if the image contains multiple containers or services.

Therefore, they can start the container, mapping the port properly and setting a name:

```
docker run -d -p 5000:5000 --name inference_server  
sanikanandpure/ml-damage-api
```

When the container is running, a user can make a POST request to the API endpoint at “<http://localhost:5000/inference>”. First, they must upload the image file to the appropriate directory. In their request, they must include the proper path to that image as the value to the key “image,” contained within a “files” object to send within the request body.

Here’s a code example using Python requests. A user would need to create a new Python file with the below code (suppose it’s named “test.py”). Then, in their terminal, they would simply execute the code via the command “python test.py”.

```
import requests  
url = "http://localhost:5000/inference"  
  
with open("<path_to_image_here>.jpg", "rb") as img:
```

```
files = {"image": img}
response = requests.post(url, files=files)

print(response.json())
```

Here's another code example using CURL. A user simply needs to run this command in their terminal.

```
curl -v -X POST http://localhost:5000/inference -F
"image=@<path_to_image_here>.jpg"
```

Both methods should return a json response that contains the model's prediction in the form of:

```
{"prediction": "damage"} or {"prediction": "no_damage"}
```

Finally, to stop the container, they can run the commands:

```
docker stop inference_server
```