Athulya Saravanakumar
COE Project 2

Project 2 Written Report

Data preparation

To prepare the dataset, I first loaded all satellite images from the provided directory structure, which contained two folders: damage and no_damage. Each image was resized to a consistent resolution of 128×128×3, converted to float32, and normalized to the range [0, 1] to stabilize neural network training. I also shuffled the entire dataset before splitting to avoid any ordering bias in how the images were stored. These preprocessing steps ensured that all images had a uniform shape and format suitable for input into deep learning models.

After preprocessing, I split the dataset into separate training (≈70%), validation (≈15%), and testing (≈15%) sets. This split allowed me to tune hyperparameters using the validation set and objectively measure performance on unseen test data. Altogether, these steps created a clean, machine-learning-ready dataset that supports fair evaluation across the different models I built.

Model design

For model design, I explored three architectures covered in class: a fully connected Artificial Neural Network (ANN), the original LeNet-5 convolutional neural network, and the Alternate LeNet-5 architecture described in the referenced research paper. The dense ANN served as a baseline model. Since ANNs do not exploit spatial structure, I flattened each 128×128×3 image into a long vector and used several dense layers with dropout to prevent overfitting. This model was intentionally simple so I could compare how much performance improved when adding convolutional inductive biases. Although the ANN did learn meaningful patterns, its accuracy plateaued around ~0.71, confirming that flat dense layers alone are not ideal for image classification tasks involving spatial features.

Next, I implemented the classical LeNet-5 CNN, choosing this architecture because it introduces convolutional filters and pooling layers that capture local spatial patterns much more effectively than a dense network. The model used stacked convolution → pooling blocks followed by fully connected layers, resulting in significantly higher performance (~0.93 test accuracy). Finally, I implemented the Alternate LeNet-5 architecture from the paper, which deepens the convolutional stack (32→64→128→128 filters), increases representational capacity, and includes dropout for regularization. This model achieved the best performance (~0.96 test accuracy). Based on these results, I selected the Alternate LeNet-5 as my final deployed model because it provided the strongest generalization while remaining computationally feasible for CPU-only inference.

Model evaluation

Among the three architectures I evaluated, the Alternate LeNet-5 model performed the best. The dense ANN reached approximately 71% accuracy, and the classical LeNet-5 improved substantially to around 93% accuracy. The Alternate LeNet-5 architecture—which included deeper convolutional layers and dropout regularization—achieved the highest performance with a test accuracy of roughly 96%. This model demonstrated strong generalization across both damage and no_damage classes and was therefore selected as the final deployed model for the inference server.

I am confident in the final model's performance. Not only did it achieve strong results on the held-out test set, but it also passed the provided automated grading suite with 100% accuracy, correctly classifying all unseen test images served by the grader! :)

Model deployment and Inference

My final model (the Alternate LeNet-5 CNN) is exported as damage.keras and served through a Flask API (api.py). When the server starts, it loads damage.keras once into memory and exposes two HTTP endpoints: GET /summary, which returns JSON metadata about the model (name, version, parameter count, and a short description), and POST /inference, which accepts an image file under the key "image" in a multipart/form-data request. The /inference endpoint decodes the raw image bytes, resizes the image to 128×128, converts it to float32, normalizes pixel values to [0, 1], runs the model, and returns a JSON object of the form {"prediction": "damage"} or {"prediction": "no_damage"} based on a 0.5 threshold.

I containerized this API using Docker and published the image to Docker Hub as

athulyas2004/ml-damage-api.

A user can pull and run the model with:

```
docker pull athulyas2004/ml-damage-api
docker run -d -p 5000:5000 --name inference_server athulyas2004/ml-damage-api
```

Once the container is running, a user can send a request such as:

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

or use Python:

```
import requests
url = "http://localhost:5000/inference"
with open("<path_to_image>.jpeg", "rb") as f:
    resp = requests.post(url, files={"image": f})
print(resp.json())
```

In both cases, the server returns a single JSON object with the predicted class label!

Usage of AI - ChatGPT 5.1

[1] Prompt: Syntax for loading data sets from a directory using python given two datasets/categories.
Response:

```
dataset = tf.keras.utils.image_dataset_from_directory(
    "Project2",
    labels="inferred",
    label_mode="int",
    color_mode="rgb",
    image_size=(128, 128),
    batch_size=2,
    shuffle=True,
    seed=1
)
```

[2] Prompt: What are ways I can figure out basic attributes of data set (keep it brief)
Response:

Length of images, shape, height, width, channels, data type, and pixel values!

[3] Prompt: Give me different numbers of layers and perceptrons to experiment with for ANN model.
Response:

If you want a more "reasonable" ANN (and reduce memory), you can:

- Downsample to 64×64 for the ANN only
- Use a smaller single hidden layer

```
X_train_ann = tf.image.resize(X_train, (64, 64)).numpy()
X_val_ann   = tf.image.resize(X_val,   (64, 64)).numpy()
X_test_ann  = tf.image.resize(X_test,  (64, 64)).numpy()

def build_small_dense_ann(input_shape):
    model = models.Sequential([
        layers.Flatten(input_shape=input_shape),
        layers.Dense(64, activation="relu"),
        layers.Dense(1, activation="sigmoid")
    ])

    model.compile(
        optimizer="adam",
```

```
        loss="binary_crossentropy",
        metrics=["accuracy"]
    )
    return model

small_dense_model = build_small_dense_ann((64, 64, 3))
small_dense_model.summary()

history_small_dense = small_dense_model.fit(
    X_train_ann, y_train,
    validation_data=(X_val_ann, y_val),
    epochs=20,
    batch_size=64,
    verbose=1
)

small_dense_test_loss, small_dense_test_acc = small_dense_model.evaluate(
    X_test_ann, y_test, verbose=0
)
print("Small Dense ANN - Test Accuracy:", small_dense_test_acc)
```

Other prompts are me asking questions to validate whether I was going in the right path. Remainder was related to Docker file building and running

[4] Prompt: In ~/nb-data/Project_2, create the Dockerfile if you haven't yet (I can write it if you say your TF version). i havent done this yet

ChatGPT said:

Got you, we'll set up the Dockerfile from scratch 👇

We just need to make sure Docker uses the **same TF version** you used in your venv, so the model loads fine.

1️⃣ Find your TensorFlow version (in venv)

In `~/nb-data/Project_2`, with `(venv)` active:

```
python -c "import tensorflow as tf; print(tf.__version__)"
```

You'll see something like:

```
2.18.0
```

Remember that number; I'll call it TF_VERSION below.

---

2 Create Dockerfile in ~/nb-data/Project_2

Still in ~/nb-data/Project_2:

```
nano Dockerfile
```

Paste this **exactly**, but replace TF_VERSION_HERE with the version you just printed (e.g. 2.18.0):

```
FROM python:3.11

# Install the same TensorFlow + Flask versions as your venv
RUN pip install --no-cache-dir "tensorflow==TF_VERSION_HERE"
"Flask==3.1.2"

# Workdir inside the container
WORKDIR /app

# Copy your API code and trained model into the image
COPY api.py /app/api.py
COPY damage.keras /app/damage.keras

# Expose Flask port
EXPOSE 5000

# Run the server
```

```
CMD ["python", "api.py"]
```

Example if your TF is `2.18.0`:

```
RUN pip install --no-cache-dir "tensorflow==2.18.0"
"Flask==3.1.2"
```

Save & exit nano: `Ctrl+O` → ENTER → `Ctrl+X`.

---

3 Build the Docker image

From `~/nb-data/Project_2`:

```
docker build -t damage-api:latest .
```

This may take a bit (TF is big), but should finish without errors.

---

4 Run your Dockerized inference server

```
docker run --rm -p 5000:5000 damage-api:latest
```

You should see similar logs to `python api.py` (TF CPU info + Flask startup), but now it's running **inside Docker**.

Leave this terminal **running** (don't Ctrl+C yet).

---

5 Test from another terminal

Open a second SSH session to the VM:

```
cd ~/nb-data
```

```
# Quick check that the container is up and listening on 5000
docker ps
```

You should see something like:

```
CONTAINER ID   IMAGE        ...   PORTS
xxxxxx         damage-api   ...   0.0.0.0:5000->5000/tcp
```

Then:

```
curl http://127.0.0.1:5000/summary

curl -v -X POST \
  http://127.0.0.1:5000/inference \
  -F
"image=@Project_2/Project2/damage/-93.528502_30.987438.jpeg"
```

If you get the JSON prediction like before, your **Dockerized server is good** ✅

6️⃣ Run the grader

With the container still running:

```
cd ~/nb-data/coe379L-fa25/code/Project2
chmod +x start_grader.sh   # only needed once
./start_grader.sh
```

That will spin up the grader container and hit your `/summary` and `/inference`.

[5] Help me make a read me file

Quick README format (exactly what your prof asked for)

Here's a **short** `README.md` you can drop into `Project_2/`:

````
# Damage Classification Inference Server


This project provides a Dockerized inference server for a CNN
model (Alternate LeNet-5) that classifies post–Hurricane Harvey
satellite images as **damage** or **no_damage**.


---


## Starting the Server (docker-compose)


From the directory containing `docker-compose.yml`, run:


```bash
docker compose up -d
````

This command:

- Pulls the image `athulyas2004/ml-damage-api:latest` from Docker Hub (if not already present).
- Starts a container named `inference_server`.
- Exposes the API at `http://localhost:5000`.

````

Stopping the Server

To stop and remove the container created by docker-compose:

```
docker compose down
```

API Endpoints

- `GET /summary`
  Returns JSON metadata about the model (name, description, version, number of parameters).

`POST /inference`
Accepts an image file (key: `image`, sent as `multipart/form-data`) and returns:

```
{ "prediction": "damage" }
```

or

```
{ "prediction": "no_damage" }
```

-

Example Requests

Using curl

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

```
curl -X POST http://localhost:5000/inference \
  -F "image=@/full/path/to/image.jpeg"
```

Using Python

```python
import requests


# Summary

summary = requests.get("http://localhost:5000/summary")

print("Summary:", summary.json())


# Inference

url = "http://localhost:5000/inference"

with open("/full/path/to/image.jpeg", "rb") as f:

    resp = requests.post(url, files={"image": f})

print("Prediction:", resp.json())
```