

Project 2 | Use of AI

Sanika Nandpure, Melissa Huang

[1] I have a dataset folder with two subdirectories: one for each category of image. I want to load this dataset in as a single dataset in a Python data structure so that I can do ML with it later. How do I do this? [CHATGPT]

Got it 👍 — sounds like you have something like this:

```
dataset/  
├─ cats/  
│   ├── cat1.jpg  
│   ├── cat2.jpg  
│   └── ...  
└─ dogs/  
    ├── dog1.jpg  
    ├── dog2.jpg  
    └── ...
```

You want to load **all** of these images into one dataset (with their labels), so you can later use them for machine learning.

Here are a few common and clean ways to do this — depending on what ML framework you'll use later (PyTorch, TensorFlow, or general-purpose NumPy/scikit-learn).

Option 1: PyTorch

(**torchvision.datasets.ImageFolder**)

This is *the easiest* if you'll use PyTorch.

```
from torchvision import datasets, transforms  
  
# Define any transforms (resizing, normalization, etc.)  
transform = transforms.Compose([  
    transforms.Resize((128, 128)),  
    transforms.ToTensor()  
)
```

```
# Load dataset
dataset = datasets.ImageFolder(root="dataset", transform=transform)

# Check first item
img, label = dataset[0]
print(img.shape, label) # label is an integer index for the class

# You can see class-to-index mapping:
print(dataset.classes)      # ['cats', 'dogs']
print(dataset.class_to_idx) # {'cats': 0, 'dogs': 1}
```

If you want to split into train/test:

```
from torch.utils.data import random_split

train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size,
test_size])
```

Option 2: TensorFlow/Keras (**image_dataset_from_directory**)

If you're using TensorFlow or Keras, this one-liner works beautifully.

```
import tensorflow as tf

dataset = tf.keras.utils.image_dataset_from_directory(
    "dataset",
    image_size=(128, 128),
    batch_size=32
)

# Inspect
for images, labels in dataset.take(1):
```

```
print(images.shape, labels.shape)
```

If you want to split:

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset",
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=(128, 128),
    batch_size=32
)
val_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset",
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=(128, 128),
    batch_size=32
)
```

Option 3: Generic Python / NumPy (for manual ML pipelines)

If you're not using a deep learning framework yet:

```
import os
import cv2
import numpy as np

data = []
labels = []

root = "dataset"
classes = os.listdir(root)
```

```

for label, cls in enumerate(classes):
    folder = os.path.join(root, cls)
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        img = cv2.imread(img_path)
        img = cv2.resize(img, (128, 128))
        data.append(img)
        labels.append(label)

# Convert to arrays
X = np.array(data)
y = np.array(labels)

print(X.shape, y.shape)

```

Now you can pass `(X, y)` into scikit-learn or any custom ML model.

Would you like me to show you how to **wrap this into a custom PyTorch `Dataset` class** (useful if you plan to extend or augment it later)?

[2] How to show an image from a dataset of images in tensorflow [GOOGLE GEMINI (from search results)]

To display an image from a TensorFlow dataset, you typically need to extract the image data from the dataset and then use a plotting library like Matplotlib to visualize it.

Here's a common approach:

Iterate through the dataset to get a batch or individual image:

If your dataset is batched, you can iterate to get a batch of images and labels. If it's not batched, you can get individual image and label pairs.

Python

```

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

# Assuming 'dataset' is your tf.data.Dataset object
# For example, if you loaded images from a directory:
# dataset = tf.keras.utils.image_dataset_from_directory(
#     'path/to/your/images',

```

```
# labels='inferred',
# image_size=(180, 180),
# batch_size=32
# )
```

```
# Get a single batch of images and labels
for images, labels in dataset.take(1): # .take(1) gets one batch
    # 'images' will be a tensor of shape (batch_size, height, width, channels)
    # 'labels' will be a tensor of shape (batch_size,)
    break # Exit after getting the first batch
```

Convert the TensorFlow tensor to a NumPy array (if necessary) and display:

Matplotlib's `imshow` function typically works with NumPy arrays. If your image tensor is not already a NumPy array, you'll need to convert it using `.numpy()`. Also, ensure the data type is suitable for display (e.g., `uint8` for 0-255 pixel values).

Python

```
# Display the first image in the batch
plt.figure(figsize=(8, 8))
plt.imshow(images[0].numpy().astype("uint8"))
plt.title(f'Label: {labels[0].numpy()}') # Display the label if available
plt.axis("off")
plt.show()
```

Display multiple images (optional).

You can loop through the images in a batch to display several at once, often in a grid format.

Python

```
plt.figure(figsize=(10, 10))
for i in range(min(9, len(images))): # Display up to 9 images
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(f'Label: {labels[i].numpy()}')
    plt.axis("off")
plt.show()
```

Key Points:

Data Type: Ensure your image data is in a suitable format for `imshow`, typically `uint8` for 0-255 pixel values or `float32` for 0.0-1.0 normalized values. If your images are float and not normalized, you may need to scale them before converting to `uint8`.

TensorFlow to NumPy: Use `.numpy()` to convert TensorFlow tensors to NumPy arrays for Matplotlib.

Batching: If your dataset is batched, remember to access individual images within the batch (e.g., `images[0]`).

[3] Write the code to implement the CNN architecture from the paper attached (table 1, page 12). Our input includes images of size (128, 128, 3) and we want our output to be a binary classification (just one output).

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Conv2D, MaxPooling2D, Flatten, Dense, Dropout
)
from tensorflow.keras.regularizers import l2

# Define model
model = Sequential([
    # Block 1
    Conv2D(32, (3, 3), activation='relu',
           input_shape=(128, 128, 3)),
    MaxPooling2D(pool_size=(2, 2)),

    # Block 2
    Conv2D(64, (3, 3), activation='relu',
           MaxPooling2D(pool_size=(2, 2)),

    # Block 3
    Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(1e-6)),
    MaxPooling2D(pool_size=(2, 2)),

    # Block 4
    Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(1e-6)),
    MaxPooling2D(pool_size=(2, 2)),

    # Classification head
    Flatten(),
    Dropout(0.5),
    Dense(512, activation='relu', kernel_regularizer=l2(1e-6)),
    Dense(1, activation='sigmoid') # binary output
])

# Compile model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Summary
model.summary()
```

[4] Rewrite your previous code without the kernel_regularizer and use a style that just uses the .add function to the model.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()

# Block 1
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Block 2
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Block 3
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Block 4
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Classification head
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu'))
model.add(Dense(1, activation='sigmoid')) # binary output

# Compile model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Summary
model.summary()
```

[Used in api.py] Using an uploaded raw image file via passing it in as a mutli-part/form into Flask's built-in request.files object, how do we need to preprocess the data obtained by request.files['image'] to be able to pass it into a model.predict() function?

Minimal correct version:

```
import tensorflow as tf

import numpy as np


def preprocess_input(im_file):

    # read uploaded bytes

    img_bytes = im_file.read()


    # decode into tensor, force 3 channels

    img = tf.io.decode_image(img_bytes, channels=3)


    # resize correctly

    img = tf.image.resize(img, [128, 128])


    # convert to float32 and normalize if needed

    img = tf.cast(img, tf.float32) / 255.0


    # convert to numpy

    arr = img.numpy()
```



```
# add batch dimension: (1, 128, 128, 3)

arr = np.expand_dims(arr, axis=0)

return arr
```

Notes:

- No `.reshape(128, 128, 3)` ever. Resize is the correct operation.
- The batch dimension is added after preprocessing.
- This matches input shape `(128, 128, 3)` with a batch size of 1.

[Used in a testing python file] Given previous code sent about the server, how should we go about testing it? We want to use requests and pass in an uploaded file to test the /inference route.

Correct way to test your inference server:

```
import requests

path = "/data/damage/example.jpg" # real image path on disk
data = {"image": open(path, "rb")}

rsp = requests.post("http://172.17.0.1:5000/inference", files=data)
print(rsp.json()) # should be {"prediction": "damage"} or
{"prediction": "no_damage"}
```

You **cannot** send preprocessed arrays from `X_test`; your server only accepts raw image files.

[Used for the README and user instructions] What are the instructions to give to an external user to pull a docker image and start the container?

Here's a clear set of instructions you could give to an external user so they can run your inference server:

1. Install Docker

Make sure Docker is installed on your machine:

- [Docker Desktop for Windows/macOS](#)
- `sudo apt install docker.io` on Linux

Optional but recommended: Docker Compose is included in Docker Desktop. On Linux, you can install it separately if needed.

2. Pull the Docker image

```
docker pull yourusername/inference-server:latest
```

- Replace `yourusername` with your Docker Hub username.
- If you tagged it differently, use that tag instead of `latest`.

3. Run the container

Option A: Using `docker run` (single container)

```
docker run -d -p 5000:5000 --name inference_server yourusername/inference-server:latest
```

- `-d` → run in the background (detached mode)
- `-p 5000:5000` → map container port 5000 to local machine port 5000
- `--name inference_server` → optional, gives the container a name

To stop the container:

```
docker stop inference_server
```

```
docker rm inference_server # removes the container if desired
```

Option B: Using **docker-compose** (optional)

Create a **docker-compose.yml** file with:

```
version: "3.9"
```

```
services:
```

```
  inference:
```

```
    image: yourusername/inference-server:latest
```

```
    ports:
```

```
      - "5000:5000"
```

```
    restart: unless-stopped
```

Then run:

```
# Start container
```

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

```
# Stop container
```

```
docker-compose down
```

```
# View logs
```

```
docker-compose logs -f
```

4. Test the inference server

Once the container is running, you can make a POST request with an image:

Python example:

```
import requests
```

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

```
with open("test_image.jpg", "rb") as f:
```

```
    files = {"image": f}
```

```
    response = requests.post(url, files=files)
```

```
print(response.json())
```

cURL example:

```
curl -X POST -F "image=@test_image.jpg" http://localhost:5000/inference
```

- The response should return the model's prediction, e.g., `{"prediction": "damage"}`.

If you want, I can write this as a **ready-to-copy “User Instructions” document** that looks professional and complete. It would cover everything from pulling the image to making a test request. Do you want me to do that?