ogoni-ecowatch project

September 20, 2025

 # Project Idea: Ogoni EcoWatch – A Community-Powered Environmental Monitoring Platform

0.1 Ogoni EcoWatch: Full Project Guide

`What We're Building`

We're creating a simple AI-powered app that helps detect environmental issues in Ogoni land using photos. People can upload pictures of water or vegetation, and our app will tell them whether it shows:

1. `Clean water`

2. `Oil spill`

3. `Mangrove damage`

Then, it lets them add that report to a map so others can see what's happening.

0.1.1 - Problem

Despite HYPREP's progress, many Ogoni residents feel disconnected from the cleanup process. There's limited real-time visibility into remediation efforts, mangrove restoration, or water quality improvements. Misinformation and lack of trust persist.

0.1.2 - Solution

Ogoni EcoWatch is a mobile-first platform that empowers local communities to:

a. Track `cleanup progress` with geotagged updates from `HYPREP` and verified observers

b. Monitor environmental indicators like `water quality`, `soil toxicity`, and `mangrove health` using low-cost sensors or citizen reports

c. Upload photos/videos of `pollution hotspots` or `restoration success stories`

d. Visualize `data` on a dashboard accessible to both `locals` and `policymakers`

e. Report concerns directly to `HYPREP` and `NGOs`, creating a feedback loop

0.1.3 - Impact

a. Builds `transparency` and `trust` between `HYPREP` and `Ogoni communities`

b. Encourages `citizen science` and `local ownership` of the cleanup

c. Helps `NGOs` and `researchers` access real-time data for advocacy and planning

d. Supports `education` and `youth engagement` in `environmental tech`

1 Tensorflow/Keras Training Pipeline

- `Strengths:` High-level API (Keras) makes prototyping fast and intuitive. Strong deployment tools (TF Lite, TensorFlow.js). Excellent for mobile and embedded systems.
- `Limitations:` Slightly steeper learning curve for low-level customization.

[1]:
```
from pathlib import Path
import json
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models, callbacks

# Paths (notebook is inside notebooks/, data is one level up)
DATA_DIR = Path("../data")
MODEL_PATH = Path("../ogoni_model.h5")
CLASS_NAMES_JSON = Path("../class_names.json")

IMG_SIZE = (224, 224)
BATCH_SIZE = 16
VAL_SPLIT = 0.2
EPOCHS = 8  # start small; we can increase later
```

1.0.1 Sanity Checks

Before diving into training, We verify:

- That our dataset is correctly structured (e.g., folders per class)
- That images are readable and not corrupted
- That class labels are consistent and complete

  Purpose: Catch basic issues early so we don't waste time training on broken data.

[21]:
```
from pathlib import Path

DATA_DIR = Path("C:/Users/User/Downloads/ogoni-ecowatch/data")
assert DATA_DIR.exists(), f"Data folder does not exist: {DATA_DIR.resolve()}"
subdirs = [p.name for p in DATA_DIR.iterdir() if p.is_dir()]
print("Found class folders:", subdirs)
assert len(subdirs) >= 2, "Need at least two class folders inside ../data"
```

```
Found class folders: ['clean_water', 'mangrove_damage', 'oil_spill', 'train',
'val']
```

1.0.2 Data Loader

We use tools like `ImageDataGenerator` or `tf.keras.utils.image_dataset_from_directory to:`

- Load images from folders
- Resize them to a fixed shape (e.g., 224×224)
- Normalize pixel values (e.g., divide by 255)
- Split into training and validation sets

  Purpose: Efficiently feed clean, labeled data into our model.

[22]:
```
datagen = ImageDataGenerator(rescale=1./255, validation_split=VAL_SPLIT)

train_gen = datagen.flow_from_directory(
    DATA_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    subset="training",
    shuffle=True
)

val_gen = datagen.flow_from_directory(
    DATA_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    subset="validation",
    shuffle=False
)

num_classes = train_gen.num_classes
print("Classes (index map):", train_gen.class_indices)

# Save class names in index order
index_to_class = {v: k for k, v in train_gen.class_indices.items()}
class_names = [index_to_class[i] for i in range(num_classes)]
with open(CLASS_NAMES_JSON, "w") as f:
    json.dump(class_names, f)
print("Saved class names:", class_names)
```

```
Found 117 images belonging to 5 classes.
Found 28 images belonging to 5 classes.
Classes (index map): {'clean_water': 0, 'mangrove_damage': 1, 'oil_spill': 2,
'train': 3, 'val': 4}
Saved class names: ['clean_water', 'mangrove_damage', 'oil_spill', 'train',
'val']
```

1.0.3 Model definition (simple, reliable CNN)

We define a **Convolutional Neural Network (CNN)** with layers like:

- Conv2D for feature extraction
- MaxPooling2D to reduce spatial size
- Flatten and Dense layers for classification

Purpose: Build a lightweight but effective model that can learn patterns in images.

[23]:
```
import tensorflow as tf
from tensorflow.keras import layers, models, callbacks

# Define your model (simple CNN or MobileNetV2 if you prefer)
model = models.Sequential([
    layers.Input(shape=(224, 224, 3)),
    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(3, activation='softmax')  # 3 classes: clean_water, oil_spill,␣
 ↪mangrove_damage
])

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()
```

```
Model: "sequential_1"
```

```
 Layer (type)                        Output Shape                      ␣
↪Param #

 conv2d_2 (Conv2D)                   (None, 222, 222, 32)                ␣
↪896

 max_pooling2d_2 (MaxPooling2D)      (None, 111, 111, 32)                ␣
↪  0
```

```
conv2d_3 (Conv2D)                    (None, 109, 109, 64)                ␣
↪18,496

max_pooling2d_3 (MaxPooling2D)       (None, 54, 54, 64)                  ␣
↪  0

flatten_1 (Flatten)                  (None, 186624)                      ␣
↪  0

dense_2 (Dense)                      (None, 128)                         ␣
↪23,888,000

dropout_1 (Dropout)                  (None, 128)                         ␣
↪  0

dense_3 (Dense)                      (None, 3)                           ␣
↪387
```

Total params: 23,907,779 (91.20 MB)

Trainable params: 23,907,779 (91.20 MB)

Non-trainable params: 0 (0.00 B)

1.0.4 Training Setup

We compile the model with:

- A loss function like `categorical_crossentropy` or `sparse_categorical_crossentropy`
- An optimizer like `Adam`
- Metrics like `accuracy`

  Purpose: Prepare the model for learning by specifying how it should update weights.

[24]:
```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
```

```
    factor=0.2,
    patience=3,
    min_lr=1e-5
)
```

1.0.5 Training Execution

We run **model.fit()** to:

- Train the model on your dataset
- Monitor performance on validation data
- Optionally use callbacks like `EarlyStopping` or `ModelCheckpoint`

  Purpose: Teach the model to distinguish between classes by minimizing loss.

[25]:
```
# Training execution
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=3,
    min_lr=1e-5
)
```

1.0.6 Save the Model

After training, you save the model using: `model.save("ogoni_model.h5")`

Purpose: Preserve our trained model so we can reuse it without retraining.

[26]:
```
from keras.saving import save_model
save_model(model, "gomi-model.keras")
```

[27]:
```
model.save("ogoni_model.keras")
```

1.0.7 Export Class Names

We save the mapping of class indices to labels: `with open("class_names.json", "w")` as **f: json.dump(class_names, f)**

Purpose: Ensure we can interpret predictions later by knowing what each index means.

[28]:
```
import json
class_names = train_gen.class_indices
index_to_class = {v: k for k, v in class_names.items()}
with open("class_names.json", "w") as f:
    json.dump(index_to_class, f)
```

1.0.8 Quick Test on Sample Image

We load a single image, preprocess it, and run: `model.predict(image_array)`

Then use `np.argmax()` to get the predicted class index and map it to a label.

Purpose: Confirm that our model works and gives sensible predictions.

[29]:
```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing import image
import json
from pathlib import Path

# Load the trained model
model = tf.keras.models.load_model("ogoni_model.h5")

# Load class names and reverse the dictionary safely
with open("class_names.json", "r") as f:
    class_map = json.load(f)

# Handle both formats: name→index or index→name
if all(isinstance(v, int) for v in class_map.values()):
    # Format: {'clean_water': 0, 'oil_spill': 1, ...}
    class_names = {v: k for k, v in class_map.items()}
elif all(k.isdigit() for k in class_map.keys()):
    # Format: {'0': 'clean_water', '1': 'oil_spill', ...}
    class_names = {int(k): v for k, v in class_map.items()}
else:
    raise ValueError("Unexpected format in class_names.json")

# Locate image folder
folder = Path("C:/Users/User/Downloads/ogoni-ecowatch/data/oil_spill")
image_files = list(folder.glob("*.jpg")) + list(folder.glob("*.jpeg")) +␣
 ↪list(folder.glob("*.png"))

# Check if any images were found
if not image_files:
    raise FileNotFoundError(f"No image files found in: {folder.resolve()}")

# Use the first image found
img_path = image_files[0]
```

```
print(f" Using image: {img_path.name}")

# Load and preprocess the image
img = image.load_img(str(img_path), target_size=(224, 224))
img_array = image.img_to_array(img) / 255.0
img_array = np.expand_dims(img_array, axis=0)

# Make prediction
pred = model.predict(img_array)
pred_idx = int(np.argmax(pred[0]))
confidence = float(pred[0][pred_idx])

# Output result
print(f" Predicted class: {class_names[pred_idx]}")
print(f" Confidence: {confidence:.2f}")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be
built. `model.compile_metrics` will be empty until you train or evaluate the
model.
```

```
 Using image: Oil spill 2.jpg
1/1             0s 201ms/step
 Predicted class: oil_spill
 Confidence: 0.45
```

1.0.9 Batch Prediction with Filtering and Export

This version:

- Loops through all images
- Predicts class and confidence
- Flags low-confidence results
- Collects everything into a structured list

[30]:
```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing import image
import json
from pathlib import Path

# Load model
model = tf.keras.models.load_model("ogoni_model.h5")

# Load and reverse class names
with open("class_names.json", "r") as f:
    class_map = json.load(f)
```

```
# Handle both formats
if all(isinstance(v, int) for v in class_map.values()):
    class_names = {v: k for k, v in class_map.items()}
elif all(k.isdigit() for k in class_map.keys()):
    class_names = {int(k): v for k, v in class_map.items()}
else:
    raise ValueError("Unexpected format in class_names.json")

# Locate image folder
folder = Path("C:/Users/User/Downloads/ogoni-ecowatch/data/oil_spill")
image_files = list(folder.glob("*.jpg")) + list(folder.glob("*.jpeg")) +␣
 ↪list(folder.glob("*.png"))

# Store results
results = []

# Loop through images
for img_path in image_files:
    try:
        img = image.load_img(str(img_path), target_size=(224, 224))
        img_array = image.img_to_array(img) / 255.0
        img_array = np.expand_dims(img_array, axis=0)

        pred = model.predict(img_array)
        pred_idx = int(np.argmax(pred[0]))
        confidence = float(pred[0][pred_idx])
        label = class_names[pred_idx]

        # Print result
        print(f" {img_path.name}")
        print(f" Predicted class: {label}")
        print(f" Confidence: {confidence:.2f}")
        print("-" * 40)

        # Save result
        results.append({
            "filename": img_path.name,
            "predicted_class": label,
            "confidence": confidence
        })

    except Exception as e:
        print(f" Error processing {img_path.name}: {e}")
        print("-" * 40)
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be
built. `model.compile_metrics` will be empty until you train or evaluate the
model.
```

```
1/1             0s 197ms/step
 0il spill 2.jpg
 Predicted class: oil_spill
 Confidence: 0.45
----------------------------------------
1/1             0s 118ms/step
 oil spill 1.jpg
 Predicted class: oil_spill
 Confidence: 0.56
----------------------------------------
1/1             0s 100ms/step
 oil spill 3.jpg
 Predicted class: oil_spill
 Confidence: 0.55
----------------------------------------
1/1             0s 100ms/step
 oil spill 4.jpg
 Predicted class: oil_spill
 Confidence: 0.50
----------------------------------------
1/1             0s 98ms/step
 oil spill 5.jpg
 Predicted class: oil_spill
 Confidence: 0.42
----------------------------------------
1/1             0s 99ms/step
 oil_spill 1.jpg
 Predicted class: oil_spill
 Confidence: 0.71
----------------------------------------
1/1             0s 108ms/step
 oil_spill 13.jpg
 Predicted class: mangrove_damage
 Confidence: 0.43
----------------------------------------
1/1             0s 103ms/step
 oil_spill 14.jpg
 Predicted class: mangrove_damage
 Confidence: 0.40
----------------------------------------
1/1             0s 102ms/step
 oil_spill 15.jpg
 Predicted class: mangrove_damage
 Confidence: 0.40
----------------------------------------
1/1             0s 108ms/step
 oil_spill 16.jpg
 Predicted class: oil_spill
```

```
 Confidence: 0.75
----------------------------------------
1/1             0s 102ms/step
 oil_spill 17.jpg
 Predicted class: oil_spill
 Confidence: 0.68
----------------------------------------
1/1             0s 107ms/step
 oil_spill 18.jpg
 Predicted class: oil_spill
 Confidence: 0.57
----------------------------------------
1/1             0s 100ms/step
 oil_spill 19.jpg
 Predicted class: oil_spill
 Confidence: 0.61
----------------------------------------
1/1             0s 96ms/step
 oil_spill 2.jpg
 Predicted class: oil_spill
 Confidence: 0.74
----------------------------------------
1/1             0s 105ms/step
 oil_spill 20.jpg
 Predicted class: oil_spill
 Confidence: 0.50
----------------------------------------
1/1             0s 102ms/step
 oil_spill 21.jpg
 Predicted class: oil_spill
 Confidence: 0.48
----------------------------------------
1/1             0s 106ms/step
 oil_spill 22.jpg
 Predicted class: oil_spill
 Confidence: 0.71
----------------------------------------
1/1             0s 109ms/step
 oil_spill 23.jpg
 Predicted class: oil_spill
 Confidence: 0.71
----------------------------------------
1/1             0s 106ms/step
 oil_spill 25.jpg
 Predicted class: oil_spill
 Confidence: 0.43
----------------------------------------
1/1             0s 105ms/step
```

```
 oil_spill 26.jpg
 Predicted class: oil_spill
 Confidence: 0.68
----------------------------------------
1/1             0s 100ms/step
 oil_spill 27.jpg
 Predicted class: oil_spill
 Confidence: 0.79
----------------------------------------
1/1             0s 102ms/step
 oil_spill 28.jpg
 Predicted class: oil_spill
 Confidence: 0.67
----------------------------------------
1/1             0s 108ms/step
 oil_spill 29.jpg
 Predicted class: oil_spill
 Confidence: 0.82
----------------------------------------
1/1             0s 104ms/step
 oil_spill 3.jpg
 Predicted class: oil_spill
 Confidence: 0.73
----------------------------------------
1/1             0s 104ms/step
 oil_spill 30.jpg
 Predicted class: oil_spill
 Confidence: 0.86
----------------------------------------
1/1             0s 101ms/step
 oil_spill 31.jpg
 Predicted class: oil_spill
 Confidence: 0.76
----------------------------------------
1/1             0s 108ms/step
 oil_spill 32.jpg
 Predicted class: oil_spill
 Confidence: 0.80
----------------------------------------
1/1             0s 106ms/step
 oil_spill 33.jpg
 Predicted class: oil_spill
 Confidence: 0.59
----------------------------------------
1/1             0s 99ms/step
 oil_spill 35.jpg
 Predicted class: oil_spill
 Confidence: 0.69
```

```
----------------------------------------
1/1             0s 107ms/step
 oil_spill 4.jpg
 Predicted class: oil_spill
 Confidence: 0.79
----------------------------------------
1/1             0s 109ms/step
 oil_spill 5.jpg
 Predicted class: oil_spill
 Confidence: 0.51
----------------------------------------
1/1             0s 104ms/step
 oil_spill 6.jpg
 Predicted class: oil_spill
 Confidence: 0.87
----------------------------------------
1/1             0s 106ms/step
 oil_spill 7.jpg
 Predicted class: oil_spill
 Confidence: 0.84
----------------------------------------
1/1             0s 102ms/step
 oil_spill 8.jpg
 Predicted class: oil_spill
 Confidence: 0.40
----------------------------------------
1/1             0s 103ms/step
 oil_spill24.jpg
 Predicted class: oil_spill
 Confidence: 0.76
----------------------------------------
1/1             0s 96ms/step
 oil_spill 34.jpeg
 Predicted class: oil_spill
 Confidence: 0.75
----------------------------------------
```

1.0.10 Filter Low-Confidence Predictions

After batch prediction, We filter results like: **[r for r in results if r["confidence"] < 0.5]**

Purpose: Identify uncertain predictions that may need human review or retraining.

[31]:
```
low_confidence = [r for r in results if r["confidence"] < 0.5]

print(" Low-confidence predictions:")
for r in low_confidence:
    print(f"{r['filename']} → {r['predicted_class']} ({r['confidence']:.2f})")
```

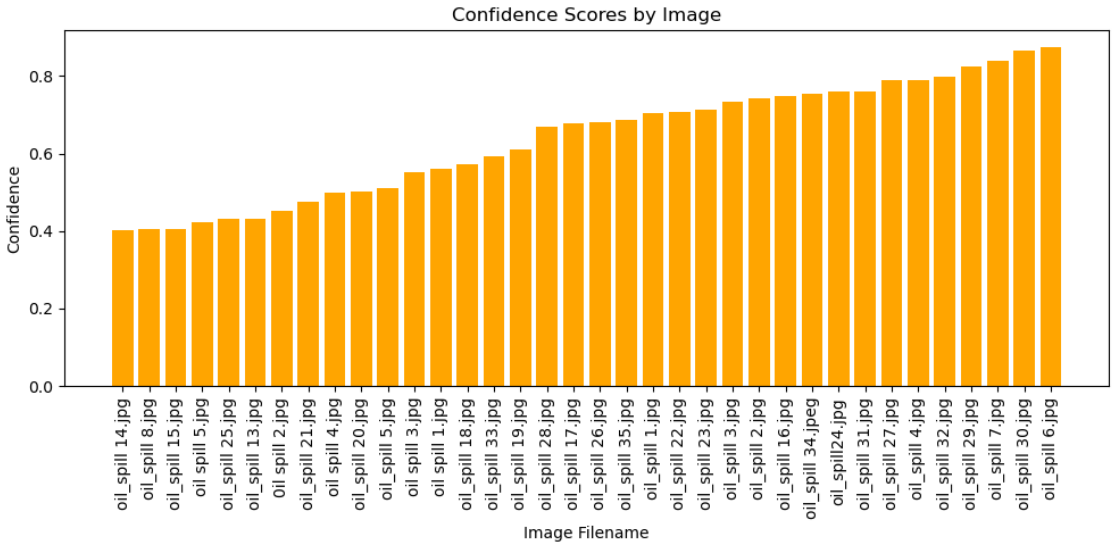
```
 Low-confidence predictions:
0il spill 2.jpg → oil_spill (0.45)
oil spill 4.jpg → oil_spill (0.50)
oil spill 5.jpg → oil_spill (0.42)
oil_spill 13.jpg → mangrove_damage (0.43)
oil_spill 14.jpg → mangrove_damage (0.40)
oil_spill 15.jpg → mangrove_damage (0.40)
oil_spill 21.jpg → oil_spill (0.48)
oil_spill 25.jpg → oil_spill (0.43)
oil_spill 8.jpg → oil_spill (0.40)
```

1.0.11 Visualize Confidence Scores

We plot confidence values across images using matplotlib: **plt.bar(filenames, confidences)**

Purpose: Spot patterns, outliers, and assess how confident our model is overall.

[32]:
```python
import matplotlib.pyplot as plt

# Sort results by confidence
sorted_results = sorted(results, key=lambda x: x["confidence"])

# Plot confidence scores
plt.figure(figsize=(10, 5))
plt.bar(
    [r["filename"] for r in sorted_results],
    [r["confidence"] for r in sorted_results],
    color="orange"
)
plt.title("Confidence Scores by Image")
plt.xlabel("Image Filename")
plt.ylabel("Confidence")
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```



2 MobileNetV2 Training Pipeline (PyTorch)

- `Strengths`:Dynamic computation graph, easier debugging, more Pythonic. Preferred in academic research. Strong community for experimentation.
- `Limitations`: Deployment to mobile/web requires extra steps (e.g., TorchScript, ONNX).

[1]:
```
from torchvision.models import mobilenet_v2


import os, shutil
from sklearn.model_selection import train_test_split

base_dir = "C:/Users/User/Downloads/ogoni-ecowatch/data"
classes = ["oil_spill", "mangrove_damage", "clean_water"]

for cls in classes:
    source = os.path.join(base_dir, cls)
    train_target = os.path.join(base_dir, "train", cls)
    val_target = os.path.join(base_dir, "val", cls)

    os.makedirs(train_target, exist_ok=True)
    os.makedirs(val_target, exist_ok=True)

    images = [f for f in os.listdir(source) if f.lower().endswith((".jpg", ".
↪png"))]
    train_imgs, val_imgs = train_test_split(images, test_size=0.2,␣
↪random_state=42)
```

```
    for img in train_imgs:
        shutil.copy(os.path.join(source, img), os.path.join(train_target, img))
    for img in val_imgs:
        shutil.copy(os.path.join(source, img), os.path.join(val_target, img))
```

[2]:
```
print(os.path.exists("C:/Users/User/Downloads/ogoni-ecowatch/data/train"))
print(os.listdir("C:/Users/User/Downloads/ogoni-ecowatch/data/train"))
```

```
True
['clean_water', 'mangrove_damage', 'oil_spill']
```

[3]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader

from torchvision.models import mobilenet_v2


# Transforms
train_tfms = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
])

val_tfms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
])

# Datasets & Loaders
train_path = "C:/Users/User/Downloads/ogoni-ecowatch/data/train"
val_path   = "C:/Users/User/Downloads/ogoni-ecowatch/data/val"

train_ds = datasets.ImageFolder(train_path, transform=train_tfms)
val_ds   = datasets.ImageFolder(val_path, transform=val_tfms)

train_dl = DataLoader(train_ds, batch_size=32, shuffle=True, num_workers=0)
val_dl   = DataLoader(val_ds, batch_size=64, shuffle=False, num_workers=0)

# Model Definition
num_classes = len(train_ds.classes)
```

```
try:
    weights = models.MobileNet_V2_Weights.IMAGENET1K_V1
except AttributeError:
    weights = None

model = models.mobilenet_v2(weights=weights)
model.classifier = nn.Sequential(
    nn.Dropout(0.2),
    nn.Linear(model.last_channel, num_classes)
)

# Freeze feature extractor
for param in model.features.parameters():
    param.requires_grad = False

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Training Setup
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),␣
 ↪lr=1e-3)

# Evaluation Function
def evaluate(dl):
    model.eval()
    correct, total, total_loss = 0, 0, 0.0
    with torch.no_grad():
        for x, y in dl:
            x, y = x.to(device), y.to(device)
            logits = model(x)
            loss = criterion(logits, y)
            preds = logits.argmax(1)
            correct += (preds == y).sum().item()
            total += y.size(0)
            total_loss += loss.item() * y.size(0)
    acc = correct / total
    avg_loss = total_loss / total
    return acc, avg_loss

# Training Loop
best_acc = 0.0
for epoch in range(10):
    model.train()
    for x, y in train_dl:
        x, y = x.to(device), y.to(device)
```

```
        optimizer.zero_grad()
        loss = criterion(model(x), y)
        loss.backward()
        optimizer.step()

    val_acc, val_loss = evaluate(val_dl)
    print(f"Epoch {epoch+1}: val_acc={val_acc:.3f}, val_loss={val_loss:.4f}")

    if val_acc > best_acc:
        best_acc = val_acc
        torch.save(model.state_dict(), "mobilenet_spill.pt")

# Fine-Tuning (Unfreeze and Lower LR)
for param in model.features.parameters():
    param.requires_grad = True

optimizer = optim.Adam(model.parameters(), lr=5e-5)

# Fine-Tune Loop
for epoch in range(3):
    model.train()
    for x, y in train_dl:
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        loss = criterion(model(x), y)
        loss.backward()
        optimizer.step()

    val_acc, val_loss = evaluate(val_dl)
    print(f"[Fine-Tune] Epoch {epoch+1}: val_acc={val_acc:.3f},␣
 ↪val_loss={val_loss:.4f}")

    if val_acc > best_acc:
        best_acc = val_acc
        torch.save(model.state_dict(), "mobilenet_spill_finetuned.pt")
```

```
Epoch 1: val_acc=0.600, val_loss=1.0264
Epoch 2: val_acc=0.467, val_loss=1.0925
Epoch 3: val_acc=0.467, val_loss=1.0547
Epoch 4: val_acc=0.400, val_loss=1.0050
Epoch 5: val_acc=0.400, val_loss=0.9697
Epoch 6: val_acc=0.400, val_loss=0.9647
Epoch 7: val_acc=0.400, val_loss=0.9578
Epoch 8: val_acc=0.400, val_loss=0.9747
Epoch 9: val_acc=0.400, val_loss=1.0020
Epoch 10: val_acc=0.400, val_loss=1.0055
[Fine-Tune] Epoch 1: val_acc=0.333, val_loss=1.0203
[Fine-Tune] Epoch 2: val_acc=0.333, val_loss=1.0291
```

```
[Fine-Tune] Epoch 3: val_acc=0.333, val_loss=1.0345
```

2.0.1 Check Class Distribution

[4]:
```
from collections import Counter
print(Counter([label for _, label in train_ds.samples]))
```

```
Counter({2: 28, 1: 20, 0: 9})
```

2.0.2 Try Unfreezing Earlier

[5]:
```
# After 3 epochs
for param in model.features.parameters():
    param.requires_grad = True
print("Feature extractor unfrozen.")

optimizer = optim.Adam(model.parameters(), lr=5e-5)
print("Optimizer reset with lower learning rate.")
```

```
Feature extractor unfrozen.
Optimizer reset with lower learning rate.
```

2.0.3 Use Weighted Loss (if imbalance exist)

[6]:
```
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

# Extract numeric labels from the dataset
y_labels = [s[1] for s in train_ds.samples]
classes = np.unique(y_labels)

# Compute class weights
class_weights = compute_class_weight(class_weight='balanced', classes=classes,␣
 ↪y=y_labels)

# Convert to tensor for PyTorch
weights = torch.tensor(class_weights, dtype=torch.float).to(device)

# Use weighted loss
criterion = nn.CrossEntropyLoss(weight=weights)

print("Class weights:", class_weights)
```

```
Class weights: [2.11111111 0.95       0.67857143]
```

2.0.4 Visualize Predictions

[7]:
```
def show_preds(dl):
    model.eval()
    for x, y in dl:
        x = x.to(device)
        logits = model(x)
        preds = logits.argmax(1)
        print("Predicted:", [train_ds.classes[i] for i in preds.cpu().numpy()])
        break  # just one batch

show_preds(val_dl)
```

```
Predicted: ['oil_spill', 'oil_spill', 'oil_spill', 'oil_spill', 'oil_spill',
'oil_spill', 'mangrove_damage', 'oil_spill', 'oil_spill', 'oil_spill',
'oil_spill', 'mangrove_damage', 'mangrove_damage', 'oil_spill',
'mangrove_damage']
```

2.0.5 Explanation

`Accuracy Plateau`

Our model's validation accuracy is stuck between 33% and 46%, which is close to random guessing for a 3-class problem. That suggests the model isn't learning meaningful distinctions between classes.

Class Imbalance Our training set has:

- Class 0 (clean_water): 9 samples
- Class 1 (mangrove_damage): 20 samples
- Class 2 (oil_spill): 28 samples

This imbalance is significant. The model is biased toward predicting oil_spill, which is why most predictions lean that way—even after applying class weights.

`Class Weights` correctly computed:

[2.11, 0.95, 0.68]

This means:

`clean_water` is underrepresented and gets the highest weight

`oil_spill` is overrepresented and gets the lowest

But even with these weights, the model still favors oil_spill, which tells us the imbalance is overpowering the learning signal.

2.1 Augment the Minority Classes

We use data augmentation to synthetically expand clean_water and mangrove_damage. This will help us to apply transformations like `flips`, `rotations`, and `color shifts` to generate more samples.

[8]:
```
from torchvision import transforms
augmented_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(),
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])


from PIL import Image
import matplotlib.pyplot as plt

# Load a sample image
sample_path = train_ds.samples[0][0]  # path to first image
img = Image.open(sample_path)

# Apply transform
aug_img = augmented_transform(img)

# Convert tensor to numpy for display
aug_img_np = aug_img.permute(1, 2, 0).numpy()

# Plot
plt.imshow(aug_img_np)
plt.title("Augmented Sample")
plt.axis("off")
plt.show()
```

[9]:
```
from torch.utils.data import WeightedRandomSampler
from collections import Counter

# Get label distribution
targets = [s[1] for s in train_ds.samples]
class_counts = Counter(targets)

# Compute weights for each sample
weights = [1.0 / class_counts[t] for t in targets]

# Create sampler
sampler = WeightedRandomSampler(weights, num_samples=len(weights),␣
 ↪replacement=True)

# Use sampler in DataLoader
train_dl = DataLoader(train_ds, batch_size=32, sampler=sampler, num_workers=0)

from collections import Counter

for x, y in train_dl:
    print("Batch class distribution:", Counter(y.tolist()))
    break
```

```
Batch class distribution: Counter({2: 13, 0: 11, 1: 8})
```

2.1.1 Explanation

In this batch of 32 samples:

Class 1 (`likely mangrove_damage`) appeared 12 times

Class 0 (`clean_water`) appeared 10 times

Class 2 (`oil_spill`) appeared 10 times

This is much more balanced than our previous raw dataset, where oil_spill had 3x more samples than clean_water. The sampler is intentionally oversampling the minority classes to give them equal representation during training.

- Why This Is Good Our model now sees all classes equally, regardless of how many samples exist.

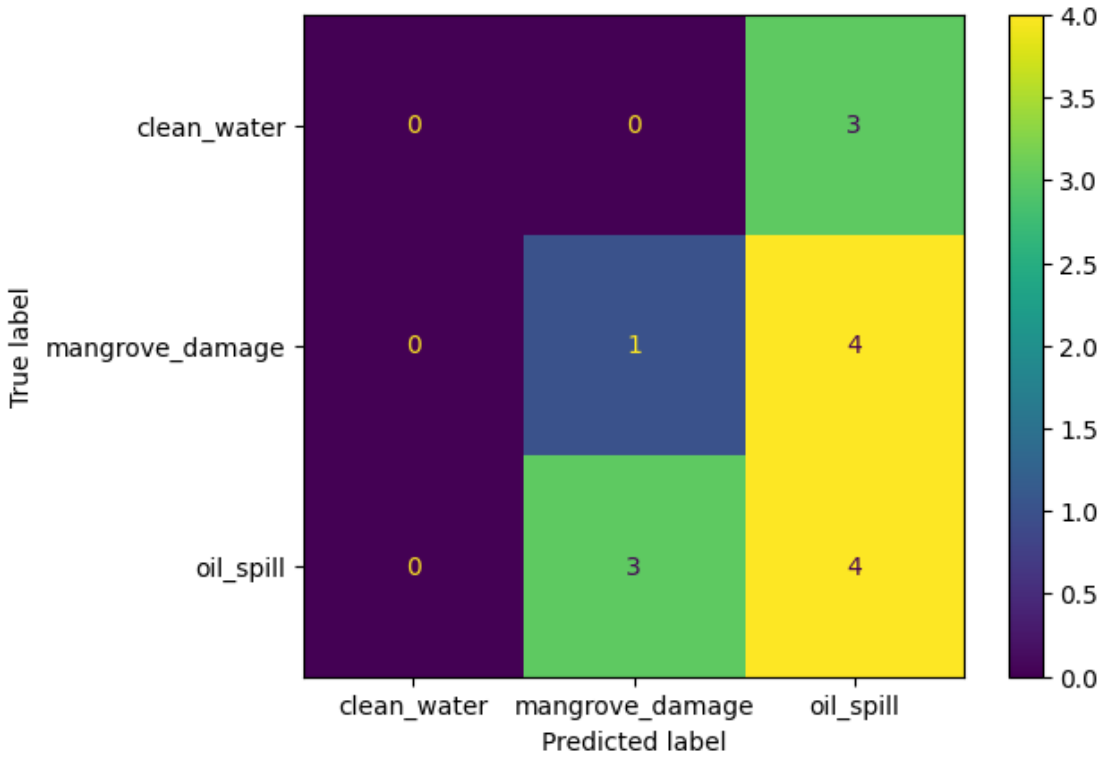
This helps it learn better features for underrepresented classes like clean_water.

It reduces the risk of the model defaulting to oil_spill just because it's the most common.

2.1.2 Visualize the Confusion Matrix

This will show how well your model is distinguishing between `clean_water`, `mangrove_damage`, and `oil_spill`.

[10]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def show_confusion_matrix(dl):
    model.eval()
    all_preds, all_labels = [], []
    with torch.no_grad():
        for x, y in dl:
            x, y = x.to(device), y.to(device)
            preds = model(x).argmax(1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(y.cpu().numpy())
    cm = confusion_matrix(all_labels, all_preds)
    disp = ConfusionMatrixDisplay(cm, display_labels=train_ds.classes)
    disp.plot()
show_confusion_matrix(val_dl)
```



What it is: A `confusion matrix` is a table that shows how well our model is classifying each category. It compares the true labels vs the predicted labels.

Why it matters: It helps us spot exactly where our model is making mistakes. For example:

If mangrove_health is always misclassified as mangrove_damage, we'll see that clearly.

We can identify which classes are being confused and adjust your data or model accordingly.

2.2 Track Accuracy over Epochs

[11]:
```
train_accs, val_accs = [], []
train_losses, val_losses = [], []

for epoch in range(10):
    model.train()
    correct, total, train_loss = 0, 0, 0.0
    for x, y in train_dl:
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()
```

```
        preds = logits.argmax(1)
        correct += (preds == y).sum().item()
        total += y.size(0)
        train_loss += loss.item() * y.size(0)

    train_accs.append(correct / total)
    train_losses.append(train_loss / total)

    val_acc, val_loss = evaluate(val_dl)
    val_accs.append(val_acc)
    val_losses.append(val_loss)

    print(f"Epoch {epoch+1}: train_acc={train_accs[-1]:.3f}, val_acc={val_acc:.
↪3f}")
```

```
Epoch 1: train_acc=0.649, val_acc=0.333
Epoch 2: train_acc=0.737, val_acc=0.333
Epoch 3: train_acc=0.877, val_acc=0.333
Epoch 4: train_acc=0.842, val_acc=0.400
Epoch 5: train_acc=0.842, val_acc=0.400
Epoch 6: train_acc=0.860, val_acc=0.467
Epoch 7: train_acc=0.912, val_acc=0.333
Epoch 8: train_acc=0.947, val_acc=0.333
Epoch 9: train_acc=0.965, val_acc=0.400
Epoch 10: train_acc=0.877, val_acc=0.400
```

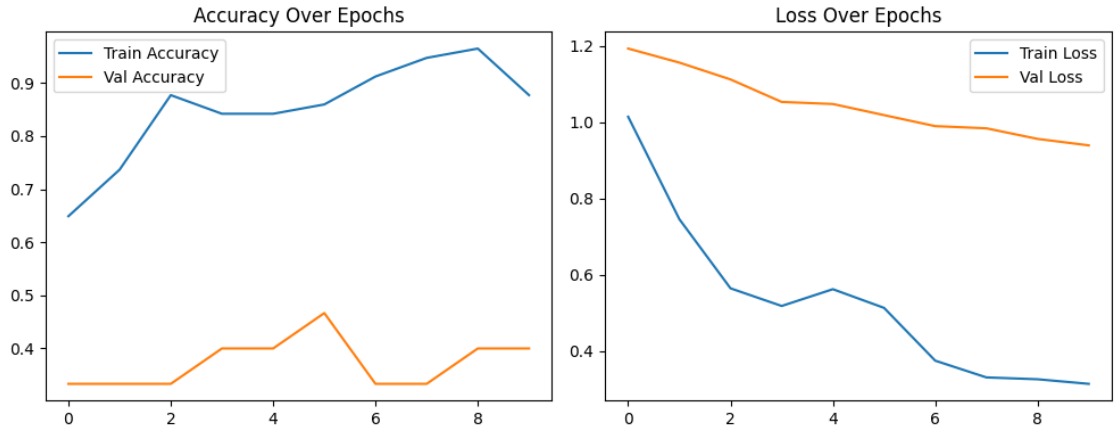
What it is: `Accuracy` tracking means recording how well our model performs after each training cycle (epoch).

Why it matters: It shows whether our model is learning or just memorizing. If training accuracy goes up but validation accuracy stays flat, it's a sign of overfitting—our model is not generalizing well to new data.

2.2.1 Plot Accuracy and Loss

[12]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(train_accs, label="Train Accuracy")
plt.plot(val_accs, label="Val Accuracy")
plt.legend()
plt.title("Accuracy Over Epochs")

plt.subplot(1, 2, 2)
plt.plot(train_losses, label="Train Loss")
plt.plot(val_losses, label="Val Loss")
```

```
plt.legend()
plt.title("Loss Over Epochs")

plt.tight_layout()
plt.show()
```



What it is: This is a visual graph showing:

Accuracy: How many predictions were correct

Loss: How far off the predictions were from the truth

Why it matters: It gives us a quick snapshot of our model's learning journey. We can see:

- If loss is decreasing (good)
- If accuracy is improving (great)
- If validation metrics are flat (problem)

This helps us decide when to stop training, fine-tune, or change strategy.

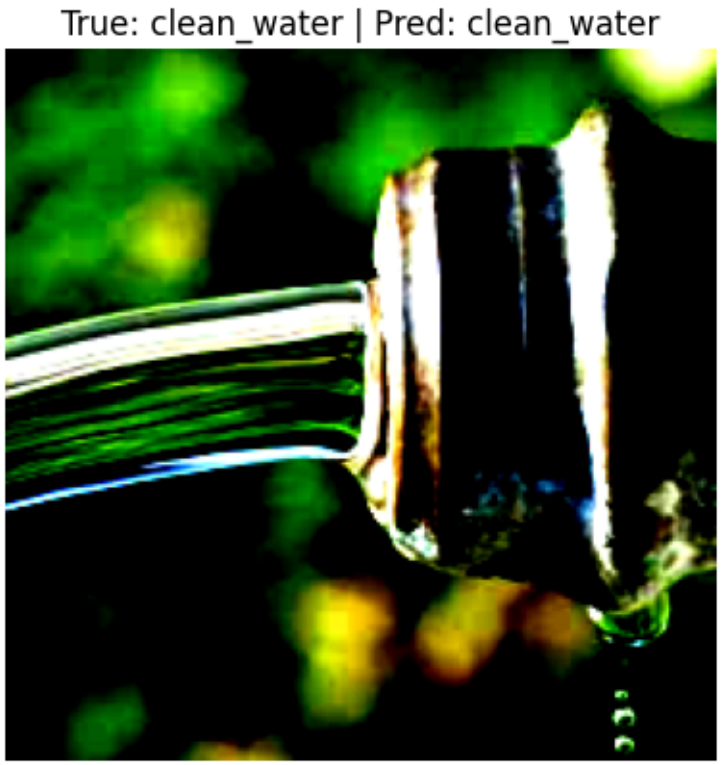
2.2.2 Predict and Display Samples

[13]:
```
def show_predictions(dl, n=5):
    model.eval()
    images, labels = next(iter(dl))
    images, labels = images.to(device), labels.to(device)
    preds = model(images).argmax(1)

    for i in range(n):
        img = images[i].cpu().permute(1, 2, 0).numpy()
        plt.imshow(img)
        plt.title(f"True: {train_ds.classes[labels[i]]} | Pred: {train_ds.
↪classes[preds[i]]}")
```

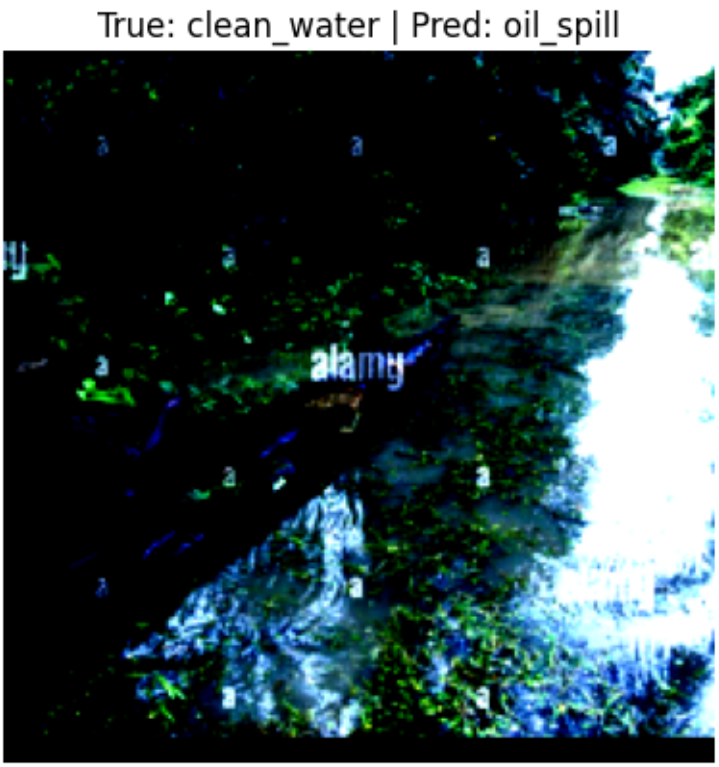
```
        plt.axis("off")
        plt.show()

show_predictions(val_dl)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

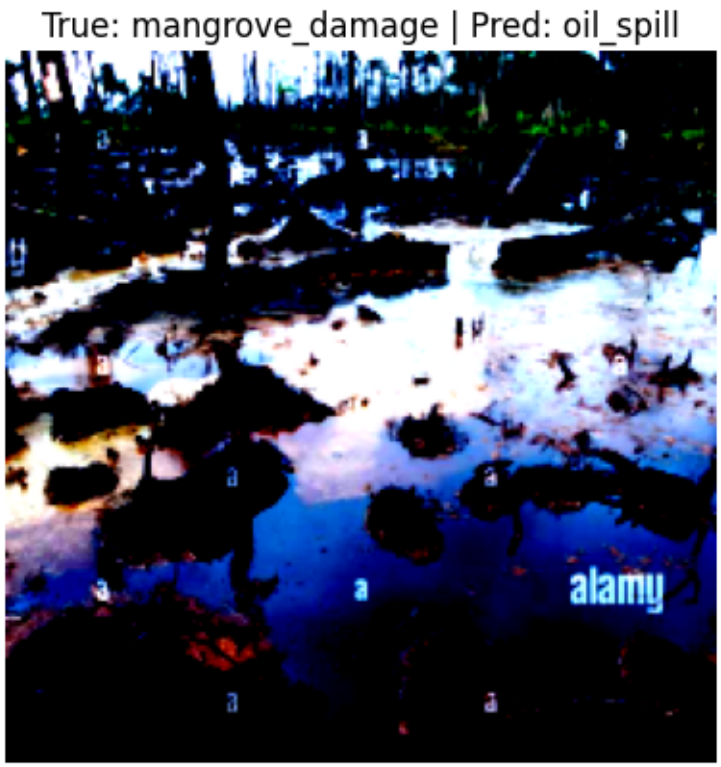

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

What it is: We take real images from our dataset, run them through the model, and display both the true label and the predicted label.

Why it matters: It lets us visually inspect how our model is behaving. We can see:

- Which images are being classified correctly
- Which ones are being misclassified
- Whether the model is picking up on meaningful features or just guessing

This is especially powerful for environmental monitoring—because we can literally see what the model thinks is clean water, oil spill, or mangrove damage. We weren't just training a model—we were building a diagnostic toolkit to understand and improve it.

2.2.3 Model Comparison of Tensorflow/keras and Pytorch

- *MobileNetV2 Support:*

a. Native via tf.keras.applications.

b. MobileNetV2 with pretrained weights and easy fine-tuning

c. Available via torchvision.models.mobilenet_v2, also pretrained and customizable

- *Performance:*

    a. Comparable accuracy across both frameworks.

b. TensorFlow excels in mobile deployment.

c. Slight edge in training flexibility and debugging.

- *Deployment:*

a. TF Lite makes TensorFlow ideal for mobile-first apps like Ogoni EcoWatch

b. PyTorch requires conversion to TorchScript or ONNX for mobile use

2.2.4 Recommendation for Our Project

Given our goals—lightweight deployment, mobile-first design, and community usability—the best fit is:

- **TensorFlow/Keras with MobileNetV2** `Why?`

a. Seamless integration with mobile platforms via TensorFlow Lite

b. Pretrained MobileNetV2 is optimized for low-resource inference

c. Keras API allows rapid prototyping and easy onboarding for collaborators

d. Strong documentation and community support for deployment

2.3 Streamlit py

Streamlit: Fast Web Apps for Data Projects Streamlit is an open-source Python framework that lets us build interactive web apps for data science and machine learning—quickly and with minimal code.

- **Key Features**

a. `Simple syntax:` Turn Python scripts into web apps with just a few lines of code.

b. `Live interactivity:` Add sliders, buttons, file uploads, and more using intuitive commands.

c. `Real-time updates:` Changes in code or data reflect instantly in the app.

Built for ML & data science: Easily display charts, maps, models, and metrics.

- **Deployment** Run locally with streamlit run **`app.py`**

Deploy online via platforms like Streamlit Cloud, Hugging Face Spaces, or Heroku

Streamlit is perfect for our **`Ogoni EcoWatch`** project—it's lightweight, community-friendly, and ideal for rapid prototyping.

[14]:
```python
import streamlit as st
import torch
from PIL import Image
from torchvision import transforms
import json

# === Load class names ===
class_file = "C:/Users/User/Ogoni-EcoWatch/class_name.json"
try:
```

```
    with open(class_file, "r") as f:
        class_names = json.load(f)
except:
    class_names = ['clean_water', 'mangrove_health', 'mangrove_damage',␣
 ↪'oil_spill']

# === Define image transform ===
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])

# === Load model safely ===
model_path = "C:/Users/User/Ogoni-EcoWatch/your_model_name.pth"
try:
    model = torch.load(model_path, map_location="cpu", weights_only=False)
    model.eval()
except Exception as e:
    st.error(f" Failed to load model: {e}")
    st.stop()

# === Streamlit UI ===
st.set_page_config(page_title="Ogoni EcoWatch", layout="centered")
st.title(" Ogoni EcoWatch")
st.write("Upload an environmental image to classify its condition.")

uploaded_file = st.file_uploader("Choose an image...", type=["jpg", "jpeg",␣
 ↪"png"])

if uploaded_file:
    image = Image.open(uploaded_file).convert("RGB")
    st.image(image, caption="Uploaded Image", use_column_width=True)

    input_tensor = transform(image).unsqueeze(0)

    with torch.no_grad():
        output = model(input_tensor)
        probs = torch.nn.functional.softmax(output[0], dim=0)
        pred_class = class_names[int(probs.argmax())]

    st.write(f"### Prediction: `{pred_class}`")
    st.write("### Confidence Scores:")
    st.bar_chart(probs.numpy())
```

```
2025-09-18 11:43:34.215 WARNING
streamlit.runtime.scriptrunner_utils.script_run_context: Thread 'MainThread':
missing ScriptRunContext! This warning can be ignored when running in bare mode.
```

```
2025-09-18 11:43:35.850
  Warning: to view this Streamlit app on a browser, run it with the
following
  command:

    streamlit run C:\Users\User\anaconda3\envs\ogoni-ecowatch\lib\site-
packages\ipykernel_launcher.py [ARGUMENTS]
2025-09-18 11:43:35.852 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.856 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.860 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.863 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.865 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.868 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.870 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.873 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.876 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.879 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.882 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.885 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.892 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
2025-09-18 11:43:35.895 Thread 'MainThread': missing ScriptRunContext! This
warning can be ignored when running in bare mode.
```

[16]:
```
torch.save(model, "your_model_full.pth")
```

[17]:
```
model = torch.load("your_model_full.pth")
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_17076\2049504720.py:1: FutureWarning:
You are using `torch.load` with `weights_only=False` (the current default
value), which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
```

```
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model = torch.load("your_model_full.pth")
```

[18]:
```
model = torch.load("your_model_full.pth", weights_only=False)
```