--- File: model.py ---

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
from PIL import Image


class StrokeCNN(nn.Module):
def __init__(self):
super(StrokeCNN, self).__init__()
self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)  # 3 channels, 64 filters
self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
self.fc1 = nn.Linear(128 * 26 * 37, 128)  # Adjust based on the feature map size
self.fc2 = nn.Linear(128, 1)  # 1 output instead of 2


def forward(self, x):
x = F.relu(self.conv1(x))
x = F.max_pool2d(x, 2)
x = F.relu(self.conv2(x))
x = F.max_pool2d(x, 2)
x = F.relu(self.conv3(x))
x = F.max_pool2d(x, 2)
x = x.view(-1, 128 * 26 * 37)
x = F.relu(self.fc1(x))
x = self.fc2(x)
return x


def load_model():
model_path = "app/stroke_cnn (1).pth"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = StrokeCNN()  # Ensure this matches the trained architecture
model.load_state_dict(torch.load(model_path, map_location=device), strict=False)  # Allow part
model.to(device)
model.eval()

return model


def predict(image_path, model):
"""Predict stroke classification for a single image."""
transform = transforms.Compose([
transforms.Resize((215, 300)),  # Resize images to 225x225
transforms.RandomRotation(30),
transforms.RandomHorizontalFlip(),
transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),
transforms.ToTensor(),
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # RGB normalizati
])

image = Image.open(image_path).convert("RGB")  # Ensure image is RGB
image = transform(image).unsqueeze(0)  # Add batch dimension
```

--- File: model.py ---

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

with torch.no_grad():
output = model(image)
predicted_class = torch.sigmoid(output).item() # Use sigmoid for binary classificatio

return predicted_class *100

def analyze_images(image_paths):
"""Process multiple images and return predictions."""
model = load_model()
results = {}
for image_path in image_paths:
results[image_path] = predict(image_path, model)
return results

if __name__ == "__main__":
model = load_model()
test_images = ["image1.jpg", "image2.jpg", "image3.jpg", "image4.jpg"] # Replace w
results = analyze_images(test_images)
print(results)

--- File: server.py ---

```
import os
import logging
from flask import Flask, request, jsonify
from flask_cors import CORS
from werkzeug.utils import secure_filename
from model import load_model, predict  # Import your model functions


# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


app = Flask(__name__)
CORS(app)  # Enable CORS for frontend communication


# Configuration
UPLOAD_FOLDER = "uploads"
ALLOWED_EXTENSIONS = {"png", "jpg", "jpeg"}
MAX_FILE_SIZE = 10 * 1024 * 1024  # 10 MB
app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER
```

--- File: server.py ---

```
# Ensure upload folder exists
os.makedirs(UPLOAD_FOLDER, exist_ok=True)


# Load the CNN model once when the server starts
try:
model = load_model()
logger.info("■ Model loaded successfully")
except Exception as e:
logger.error(f"■ Error loading model: {e}")
model = None  # Prevents crashes if the model fails to load


def allowed_file(filename):
"""Check if file extension is allowed."""
return "." in filename and filename.rsplit(".", 1)[1].lower() in ALLOWED_EXTENSIONS


@app.route("/upload", methods=["POST"])
def upload_images():
"""Handles image uploads, runs predictions, and returns results."""
try:
logger.info("■ Request received")


# Check if face image is uploaded
if "face" not in request.files:
logger.error("■ Missing face image!")
return jsonify({"error": "Missing required face image"}), 400


# Save the uploaded face image
file = request.files["face"]
if file and allowed_file(file.filename):
# Validate file size
if file.content_length > MAX_FILE_SIZE:
logger.error(f"■ File size exceeds {MAX_FILE_SIZE} bytes")
return jsonify({"error": f"File size exceeds {MAX_FILE_SIZE // (1024 * 1024)} MB"}), 4


filename = secure_filename(file.filename)
filepath = os.path.join(app.config["UPLOAD_FOLDER"], filename)
file.save(filepath)
logger.info(f"■ Face image saved at {filepath}")
else:
logger.error(f"■ Invalid file format for face image: {file.filename}")
return jsonify({"error": "Invalid file format for face image. Only PNG, JPG, JPEG allow


# Ensure the model is loaded
if model is None:
logger.error("■ Model failed to load!")
return jsonify({"error": "Model failed to load. Try restarting the server."}), 500
```

--- File: server.py ---

```
# Run prediction for face image
logger.info("■ Running prediction for face...")
face_result = predict(filepath, model)
logger.info(f"■ Face prediction: {face_result}")


# Clean up uploaded face image
if os.path.exists(filepath):
os.remove(filepath)
logger.info(f"■■ Deleted {filepath}")


return jsonify({"results": {"face": face_result}})


except Exception as e:
logger.error(f"■ Server Error: {str(e)}")
return jsonify({"error": f"Internal server error: {str(e)}"}), 500


@app.route("/health", methods=["GET"])
def health_check():
"""Health check endpoint to verify server and model status."""
if model is None:
return jsonify({"status": "unhealthy", "error": "Model failed to load"}), 500
return jsonify({"status": "healthy"}), 200


@app.errorhandler(404)
def not_found(error):
return jsonify({"error": "Route not found"}), 404


@app.errorhandler(500)
def internal_error(error):
return jsonify({"error": "Internal server error"}), 500


if __name__ == "__main__":
app.run(host="0.0.0.0", port=5000, debug=True)
```

--- File: index.tsx ---

```
import React, { useState, useEffect } from 'react';
import { View, Button, Text, Image, Alert } from 'react-native';
import * as ImagePicker from 'expo-image-picker';

export default function App() {
const [image, setImage] = useState<string | null>(null);
const [prediction, setPrediction] = useState<string | null>(null);

// Request permissions for camera and gallery
const requestPermissions = async () => {
const { status: cameraStatus } = await ImagePicker.requestCameraPermissionsAsync();
const { status: mediaLibraryStatus } = await ImagePicker.requestMediaLibraryPermissionsAsync()
```

--- File: index.tsx ---

```
if (cameraStatus !== 'granted' || mediaLibraryStatus !== 'granted') {
Alert.alert('Permission to access camera or media library is required!');
}
};

// Function to take a photo using the camera
const takePhoto = async () => {
let result = await ImagePicker.launchCameraAsync({
allowsEditing: true,
aspect: [4, 3],
quality: 1,
});

console.log("Camera result:", result); // Debug log to check the result

// Check if the photo was not canceled and assets exist
if (!result.canceled && result.assets && result.assets.length > 0) {
const uri = result.assets[0].uri; // Get URI from the first asset
setImage(uri);
uploadImage(uri); // Upload the image immediately
} else {
Alert.alert('Photo capture cancelled!');
}
};

// Send the image to the Flask server for prediction
const uploadImage = async (uri: string) => {
if (!uri) {
Alert.alert("Please take a photo first!");
return;
}

let localUri = uri;
let filename = localUri.split("/").pop();
let type = "image/jpeg"; // Adjust based on the actual image format if necessary

const formData = new FormData();
formData.append("face", { uri: localUri, name: filename, type });

try {
const response = await fetch("http://192.168.68.69:5000/upload", {

method: "POST",
body: formData,
});

if (!response.ok) throw new Error("Server error");

const responseJson = await response.json();
setPrediction(responseJson.results.face);
Alert.alert("Prediction: " + responseJson.results.face);
} catch (error) {
console.error("Upload Error:", error);
Alert.alert("Upload Error: " + error.message);
}
};
```

--- File: index.tsx ---

```
// Request permissions on component mount
useEffect(() => {
requestPermissions();
}, []);

return (
<View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
<Text>Take a Photo for Prediction:</Text>
<Button title="Take a Photo" onPress={takePhoto} />

{image && (
<View style={{ marginTop: 20 }}>
<Text>Captured Image:</Text>
<Image source={{ uri: image }} style={{ width: 200, height: 200 }} />
</View>
)}

{prediction && (
<View style={{ marginTop: 20 }}>
<Text>Prediction Result:</Text>
<Text>{prediction}</Text>
</View>
)}
</View>
);
}
```