# MLOps_Assignment2

November 30, 2025

### 0.1 Assignment 2: Containerization and Edge Computing

**DONG**

[ ]:

## 1 Part 1: Conceptual Questions

#### 1.0.1 1. Explain the key differences between virtual machines (VMs) and containers. Discuss how each technology manages system resources and isolation.

Virtual Machines (VMs) vs Containers Key Differences: - Aspect: Virtual Machines (VMs) Containers

```
- Architecture: Runs on hypervisor, includes full OS    Runs on host OS, shares kernel


- Isolation:    Complete hardware-level isolation   Process-level isolation


- Size: GBs (entire OS + app)   MBs (app + dependencies only)


- Boot: Time    Minutes Seconds
```

Resource Management:

- VMs: Each VM has dedicated resources (CPU, memory, storage) allocated by hypervisor
- Containers: Share host OS kernel, use cgroups and namespaces for resource limits

Isolation:

```
- VMs: Full isolation - different OS, kernel, hardware abstraction

- Containers: Process isolation - shared kernel but isolated processes, networks, filesystems
```

#### 1.0.2 2. Docker in Development and Deployment

- Standardization: Consistent environments across development, testing, production
- Dependency Management: Bundles application with all dependencies

- Isolation: Prevents conflicts between different applications
- Reproducibility: Same image runs identically anywhere

**Advantages for ML Projects:**

**Example benefits:**

- Reproducible experiments
- Consistent training environments
- Easy model deployment
- Version control for entire ML stack
- Simplified dependency management (CUDA, TensorFlow, etc.)

### 1.0.3 3. Edge Computing in Machine Learning

Edge computing processes data near the source rather than in centralized cloud servers. Importance in ML:

```
Low Latency: Real-time inference for applications like autonomous vehicles

Bandwidth Efficiency: Reduced data transmission to cloud

Privacy: Sensitive data stays locally

Offline Operation: Functions without internet connection
```

### 1.0.4 4. Dockerizing Python ML Application

Steps to Create Docker Container:

```
Prepare Application

    Organize code and dependencies

    Create requirements.txt

    Prepare model files

Create Dockerfile:
```

**Main components:** FROM python:3.9-slim # Base image

WORKDIR /app # Working directory

COPY requirements.txt . # Copy dependencies RUN pip install -r requirements.txt # Install packages

COPY . . # Copy application code

EXPOSE 5000 # Expose port

CMD [“python”, “app.py”] # Run command

### 1.0.5 5. Container Registries in MLOps

Role in MLOps Lifecycle:

```
Version Control: Track different versions of ML environments

Collaboration: Share reproducible environments across teams

Deployment: Consistent deployment from development to production

CI/CD Integration: Automated testing and deployment
```

**Example workflow:**

1. Development → Build Docker image with training code
2. Testing → Pull image in CI/CD pipeline
3. Training → Run container with GPU support
4. Deployment → Push inference image to registry
5. Production → Pull and run inference container

[ ]:

## 2 Part 2: Hands-On Tasks

[1]:
```
# Install all required packages
# !pip install flask tensorflow pillow numpy requests

# Verify installations
import flask
import tensorflow as tf
from PIL import Image
import numpy as np
print("  All packages installed successfully!")
print(f"TensorFlow version: {tf.__version__}")
```

```
  All packages installed successfully!
TensorFlow version: 2.20.0
```

### 2.1 Build the enhanced Flask app in Jupyter Notebook

[2]:
```
from flask import Flask, request, jsonify
import base64
import numpy as np
from PIL import Image
import io
import tensorflow as tf
import logging
import json
```

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

print(" Libraries imported successfully!")
```

```
 Libraries imported successfully!
```

Create Sample TensorFlow Lite Model

[3]:
```
# Create a simple model for testing
def create_demo_model():
    model = tf.keras.Sequential([
        tf.keras.layers.InputLayer(input_shape=(224, 224, 3)),
        tf.keras.layers.Conv2D(32, 3, activation='relu'),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    # Convert to TensorFlow Lite
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    tflite_model = converter.convert()

    # Save model
    with open('model.tflite', 'wb') as f:
        f.write(tflite_model)

    return tflite_model

# Create the model
tflite_model = create_demo_model()
print(" Demo TFLite model created: model.tflite")
```

```
C:\Users\LENOVO\anaconda3\Lib\site-
packages\keras\src\layers\core\input_layer.py:27: UserWarning: Argument
`input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
INFO:absl:Function `function` contains input name(s) resource with unsupported
characters which will be renamed to
sequential_1_dense_1_biasadd_readvariableop_resource in the SavedModel.
INFO:absl:Function `function` contains input name(s) resource with unsupported
characters which will be renamed to
sequential_1_dense_1_biasadd_readvariableop_resource in the SavedModel.

INFO:tensorflow:Assets written to:
C:\Users\LENOVO\AppData\Local\Temp\tmpzfo5l3ha\assets

INFO:tensorflow:Assets written to:
C:\Users\LENOVO\AppData\Local\Temp\tmpzfo5l3ha\assets
```

```
Saved artifact at 'C:\Users\LENOVO\AppData\Local\Temp\tmpzfo5l3ha'. The
following endpoints are available:

* Endpoint 'serve'
  args_0 (POSITIONAL_ONLY): TensorSpec(shape=(None, 224, 224, 3),
dtype=tf.float32, name='keras_tensor')
Output Type:
  TensorSpec(shape=(None, 10), dtype=tf.float32, name=None)
Captures:
  1808787770256: TensorSpec(shape=(), dtype=tf.resource, name=None)
  1808787771600: TensorSpec(shape=(), dtype=tf.resource, name=None)
  1808787771216: TensorSpec(shape=(), dtype=tf.resource, name=None)
  1808787771984: TensorSpec(shape=(), dtype=tf.resource, name=None)
 Demo TFLite model created: model.tflite
```

Load TensorFlow Lite Model

[4]:
```
# Load the TFLite model
interpreter = tf.lite.Interpreter(model_path='model.tflite')
interpreter.allocate_tensors()

# Get model details
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

print(" TensorFlow Lite model loaded successfully!")
print(f"Input shape: {input_details[0]['shape']}")
print(f"Input type: {input_details[0]['dtype']}")
print(f"Output shape: {output_details[0]['shape']}")
```

```
 TensorFlow Lite model loaded successfully!
Input shape: [  1 224 224   3]
Input type: <class 'numpy.float32'>
Output shape: [ 1 10]
```

```
C:\Users\LENOVO\anaconda3\Lib\site-
packages\tensorflow\lite\python\interpreter.py:457: UserWarning:     Warning:
tf.lite.Interpreter is deprecated and is scheduled for deletion in
    TF 2.20. Please use the LiteRT interpreter from the ai_edge_litert package.
    See the [migration guide](https://ai.google.dev/edge/litert/migration)
    for details.

  warnings.warn(_INTERPRETER_DELETION_WARNING)
```

Image Processing Functions

[5]:
```
def decode_base64_image(base64_string):
    """Decode base64 encoded image string"""
    try:
```

```
        if ',' in base64_string:
            base64_string = base64_string.split(',')[1]

        image_data = base64.b64decode(base64_string)
        return image_data
    except Exception as e:
        logger.error(f"Error decoding base64 image: {e}")
        raise

def preprocess_image(image_data, target_size=(224, 224)):
    """Preprocess image for model inference"""
    try:
        # Open image from bytes
        image = Image.open(io.BytesIO(image_data))

        # Convert to RGB if necessary
        if image.mode != 'RGB':
            image = image.convert('RGB')

        # Resize image
        image = image.resize(target_size)

        # Convert to numpy array and normalize
        image_array = np.array(image, dtype=np.float32) / 255.0

        # Add batch dimension
        image_array = np.expand_dims(image_array, axis=0)

        return image_array

    except Exception as e:
        logger.error(f"Error preprocessing image: {e}")
        raise

print(" Image processing functions defined!")
```

```
 Image processing functions defined!
```

Image Processing Functions

[6]:
```
def decode_base64_image(base64_string):
    """Decode base64 encoded image string"""
    try:
        if ',' in base64_string:
            base64_string = base64_string.split(',')[1]

        image_data = base64.b64decode(base64_string)
        return image_data
```

```
    except Exception as e:
        logger.error(f"Error decoding base64 image: {e}")
        raise

def preprocess_image(image_data, target_size=(224, 224)):
    """Preprocess image for model inference"""
    try:
        # Open image from bytes
        image = Image.open(io.BytesIO(image_data))

        # Convert to RGB if necessary
        if image.mode != 'RGB':
            image = image.convert('RGB')

        # Resize image
        image = image.resize(target_size)

        # Convert to numpy array and normalize
        image_array = np.array(image, dtype=np.float32) / 255.0

        # Add batch dimension
        image_array = np.expand_dims(image_array, axis=0)

        return image_array

    except Exception as e:
        logger.error(f"Error preprocessing image: {e}")
        raise

print(" Image processing functions defined!")
```

```
 Image processing functions defined!
```

Prediction Function

[7]:
```
def predict_with_tflite(preprocessed_image):
    """Perform inference using TensorFlow Lite model"""
    try:
        # Get input and output tensors
        input_details = interpreter.get_input_details()
        output_details = interpreter.get_output_details()

        # Set input tensor
        interpreter.set_tensor(input_details[0]['index'], preprocessed_image)

        # Run inference
        interpreter.invoke()
```

```
        # Get prediction results
        output_data = interpreter.get_tensor(output_details[0]['index'])

        # Process output
        predictions = {
            'predictions': output_data.tolist(),
            'predicted_class': int(np.argmax(output_data)),
            'confidence': float(np.max(output_data)),
            'all_confidences': [float(x) for x in output_data[0]]
        }

        return predictions

    except Exception as e:
        logger.error(f"Error during inference: {e}")
        raise

print(" Prediction function defined!")
```

```
Prediction function defined!
```

Create Enhanced Flask App

[8]:
```
# Create Flask app
app = Flask(__name__)

@app.route('/')
def home():
    return """
    <h1> Enhanced Flask App with TensorFlow Lite</h1>
    <p>Endpoints available:</p>
    <ul>
        <li><a href="/health">/health</a> - Health check</li>
        <li>/predict - POST endpoint for image prediction</li>
        <li>/process-image - POST endpoint for image processing</li>
    </ul>
    """

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({
        'status': 'healthy',
        'model_loaded': True,
        'endpoints': ['/health', '/predict', '/process-image']
    })

print(" Basic Flask app created!")
```

```
Basic Flask app created!
```

Add Image Processing Endpoint

[9]:
```python
@app.route('/process-image', methods=['POST'])
def process_image():
    """Endpoint for image processing without prediction"""
    try:
        data = request.get_json()

        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400

        # Decode base64 image
        image_data = decode_base64_image(data['image'])

        # Preprocess image
        processed_image = preprocess_image(image_data)

        return jsonify({
            'status': 'success',
            'message': 'Image processed successfully',
            'image_shape': processed_image.shape,
            'data_range': {
                'min': float(processed_image.min()),
                'max': float(processed_image.max())
            }
        })

    except Exception as e:
        logger.error(f"Image processing error: {e}")
        return jsonify({'error': str(e)}), 500

print("  Image processing endpoint added!")
```

```
 Image processing endpoint added!
```

Add Prediction Endpoint

[10]:
```python
@app.route('/predict', methods=['POST'])
def predict():
    """Endpoint for image prediction"""
    try:
        data = request.get_json()

        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400

        logger.info("Received prediction request")
```

```
        # Decode base64 image
        image_data = decode_base64_image(data['image'])

        # Preprocess image
        processed_image = preprocess_image(image_data)

        # Make prediction
        predictions = predict_with_tflite(processed_image)

        logger.info(f"Prediction completed: class␣
↪{predictions['predicted_class']}")

        return jsonify({
            'status': 'success',
            'predictions': predictions
        })

    except Exception as e:
        logger.error(f"Prediction error: {e}")
        return jsonify({'error': str(e)}), 500

print(" Prediction endpoint added!")
```

```
 Prediction endpoint added!
```

Test Image Creation Function

[11]:
```
def create_test_image_base64(width=224, height=224):
    """Create a test image and return as base64"""
    # Create a random image
    img_array = np.random.randint(0, 255, (height, width, 3), dtype=np.uint8)
    img = Image.fromarray(img_array)

    # Convert to base64
    buffered = io.BytesIO()
    img.save(buffered, format="JPEG")
    img_str = base64.b64encode(buffered.getvalue()).decode()

    return f"{img_str}"

# Create test image
test_image = create_test_image_base64()
print(" Test image created!")
print(f"Base64 length: {len(test_image)}")
```

```
 Test image created!
Base64 length: 40927
```

Run and Test the Complete App

[12]:
```
import threading
import time
import requests

def run_flask_app():
    app.run(host='0.0.0.0', port=5000, debug=False, use_reloader=False)

# Start Flask app in background
flask_thread = threading.Thread(target=run_flask_app)
flask_thread.daemon = True
flask_thread.start()

# Wait for app to start
time.sleep(3)

print(" Flask app started on http://localhost:5000")
print("Testing endpoints...")

# Test health endpoint
try:
    health_response = requests.get('http://localhost:5000/health')
    print(f" Health check: {health_response.status_code} - {health_response.
 ↪json()}")
except Exception as e:
    print(f" Health check failed: {e}")

# Test prediction endpoint
try:
    test_data = {'image': test_image}
    predict_response = requests.post('http://localhost:5000/predict',␣
 ↪json=test_data)
    print(f" Prediction test: {predict_response.status_code}")
    if predict_response.status_code == 200:
        result = predict_response.json()
        print(f"   Predicted class: {result['predictions']['predicted_class']}")
        print(f"   Confidence: {result['predictions']['confidence']:.4f}")
except Exception as e:
    print(f" Prediction test failed: {e}")

print("\n Enhanced Flask App is running!")
print("You can now test it in your browser: http://localhost:5000")
print("Or send POST requests to /predict and /process-image")
```

```
 * Serving Flask app '__main__'
 * Debug mode: off
INFO:werkzeug:WARNING: This is a development server. Do not use it in a
production deployment. Use a production WSGI server instead.
```

```
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.2.17:5000
INFO:werkzeug:Press CTRL+C to quit

 Flask app started on http://localhost:5000
Testing endpoints…

INFO:werkzeug:127.0.0.1 - - [30/Nov/2025 00:15:36] "GET /health HTTP/1.1" 200 -

 Health check: 200 - {'endpoints': ['/health', '/predict', '/process-image'],
'model_loaded': True, 'status': 'healthy'}

INFO:__main__:Received prediction request
INFO:__main__:Prediction completed: class 3
INFO:werkzeug:127.0.0.1 - - [30/Nov/2025 00:15:38] "POST /predict HTTP/1.1" 200
-

 Prediction test: 200
   Predicted class: 3
   Confidence: 0.1113


 Enhanced Flask App is running!
You can now test it in your browser: http://localhost:5000
Or send POST requests to /predict and /process-image

INFO:werkzeug:127.0.0.1 - - [30/Nov/2025 00:16:25] "GET /health HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Nov/2025 00:16:27] "GET / HTTP/1.1" 200 -
```

### 2.2 Create Deployment Files

Create Requirements file

[13]:
```
requirements = """
flask==2.3.3
tensorflow==2.13.0
pillow==10.0.0
numpy==1.24.3
gunicorn==21.2.0
"""

with open('requirements.txt', 'w') as f:
    f.write(requirements)

print(" requirements.txt created")
```

```
 requirements.txt created
```

[14]:
```
!type requirements.txt
```

```
flask==2.3.3
```

```
tensorflow==2.13.0
pillow==10.0.0
numpy==1.24.3
gunicorn==21.2.0
```

Create Dockerfile

[15]:
```
# Create Dockerfile for deployment
dockerfile_content = """
# Use Python 3.9 slim image
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \\
    gcc \\
    && rm -rf /var/lib/apt/lists/*

# Copy requirements and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code and model
COPY app.py .
COPY model.tflite .

# Create non-root user for security
RUN useradd --create-home --shell /bin/bash app
USER app

# Expose port
EXPOSE 5000

# Run the application
CMD ["python", "app.py"]
"""

with open('Dockerfile', 'w') as f:
    f.write(dockerfile_content)

print(" Dockerfile created")
```

```
 Dockerfile created
```

Create Standalone App File

[16]:
```
# Let's create a clean version of app.py without invisible characters
clean_app_code = '''from flask import Flask, request, jsonify
import base64
import numpy as np
from PIL import Image
import io
import tensorflow as tf
import logging

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

app = Flask(__name__)

# Load TensorFlow Lite model
try:
    interpreter = tf.lite.Interpreter(model_path="model.tflite")
    interpreter.allocate_tensors()
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    logger.info("TensorFlow Lite model loaded successfully")
except Exception as e:
    logger.error(f"Error loading model: {e}")
    interpreter = None

def decode_base64_image(base64_string):
    """Decode base64 encoded image string"""
    try:
        if ',' in base64_string:
            base64_string = base64_string.split(',')[1]
        image_data = base64.b64decode(base64_string)
        return image_data
    except Exception as e:
        logger.error(f"Error decoding base64 image: {e}")
        raise

def preprocess_image(image_data, target_size=(224, 224)):
    """Preprocess image for model inference"""
    try:
        image = Image.open(io.BytesIO(image_data))
        if image.mode != 'RGB':
            image = image.convert('RGB')
        image = image.resize(target_size)
        image_array = np.array(image, dtype=np.float32) / 255.0
        image_array = np.expand_dims(image_array, axis=0)
        return image_array
```

```
    except Exception as e:
        logger.error(f"Error preprocessing image: {e}")
        raise

def predict_with_tflite(preprocessed_image):
    """Perform inference using TensorFlow Lite model"""
    try:
        input_details = interpreter.get_input_details()
        output_details = interpreter.get_output_details()
        interpreter.set_tensor(input_details[0]['index'], preprocessed_image)
        interpreter.invoke()
        output_data = interpreter.get_tensor(output_details[0]['index'])
        predictions = {
            'predictions': output_data.tolist(),
            'predicted_class': int(np.argmax(output_data)),
            'confidence': float(np.max(output_data)),
            'all_confidences': [float(x) for x in output_data[0]]
        }
        return predictions
    except Exception as e:
        logger.error(f"Error during inference: {e}")
        raise

@app.route('/')
def home():
    return """
    <h1>Enhanced Flask App with TensorFlow Lite</h1>
    <p>Endpoints available:</p>
    <ul>
        <li><a href="/health">/health</a> - Health check</li>
        <li>/predict - POST endpoint for image prediction</li>
        <li>/process-image - POST endpoint for image processing</li>
    </ul>
    """

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({
        'status': 'healthy',
        'model_loaded': interpreter is not None,
        'endpoints': ['/health', '/predict', '/process-image']
    })

@app.route('/process-image', methods=['POST'])
def process_image():
    """Endpoint for image processing without prediction"""
    try:
```

```
        data = request.get_json()
        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400
        image_data = decode_base64_image(data['image'])
        processed_image = preprocess_image(image_data)
        return jsonify({
            'status': 'success',
            'message': 'Image processed successfully',
            'image_shape': processed_image.shape,
            'data_range': {
                'min': float(processed_image.min()),
                'max': float(processed_image.max())
            }
        })
    except Exception as e:
        logger.error(f"Image processing error: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/predict', methods=['POST'])
def predict():
    """Endpoint for image prediction"""
    try:
        data = request.get_json()
        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400
        image_data = decode_base64_image(data['image'])
        processed_image = preprocess_image(image_data)
        predictions = predict_with_tflite(processed_image)
        return jsonify({
            'status': 'success',
            'predictions': predictions
        })
    except Exception as e:
        logger.error(f"Prediction error: {e}")
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=False)
'''

# Write the clean version
with open('app_clean.py', 'w') as f:
    f.write(clean_app_code)

print(" Clean app file created: app_clean.py")
```

```
 Clean app file created: app_clean.py
```

### 2.3 Test and Deploy

[ ]:
```
# Test the clean version
print("Testing clean app version...")
!python app_clean.py
```

[17]:
```
# Check what's in your app.py file
print("Current app.py content:")
print("=" * 50)
with open('app.py', 'r') as f:
    content = f.read()
    print(content)
print("=" * 50)
```

```
Current app.py content:
==================================================
from flask import Flask, request, jsonify
import base64
import numpy as np
from PIL import Image
import io
import tensorflow as tf
import logging

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

app = Flask(__name__)

# Load TensorFlow Lite model
try:
    interpreter = tf.lite.Interpreter(model_path="model.tflite")
    interpreter.allocate_tensors()
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    logger.info("TensorFlow Lite model loaded successfully")
except Exception as e:
    logger.error(f"Error loading model: {e}")
    interpreter = None

def decode_base64_image(base64_string):
    """Decode base64 encoded image string"""
    try:
        if ',' in base64_string:
            base64_string = base64_string.split(',')[1]
        image_data = base64.b64decode(base64_string)
        return image_data
```

```
    except Exception as e:
        logger.error(f"Error decoding base64 image: {e}")
        raise

def preprocess_image(image_data, target_size=(224, 224)):
    """Preprocess image for model inference"""
    try:
        image = Image.open(io.BytesIO(image_data))
        if image.mode != 'RGB':
            image = image.convert('RGB')
        image = image.resize(target_size)
        image_array = np.array(image, dtype=np.float32) / 255.0
        image_array = np.expand_dims(image_array, axis=0)
        return image_array
    except Exception as e:
        logger.error(f"Error preprocessing image: {e}")
        raise

def predict_with_tflite(preprocessed_image):
    """Perform inference using TensorFlow Lite model"""
    try:
        input_details = interpreter.get_input_details()
        output_details = interpreter.get_output_details()
        interpreter.set_tensor(input_details[0]['index'], preprocessed_image)
        interpreter.invoke()
        output_data = interpreter.get_tensor(output_details[0]['index'])
        predictions = {
            'predictions': output_data.tolist(),
            'predicted_class': int(np.argmax(output_data)),
            'confidence': float(np.max(output_data)),
            'all_confidences': [float(x) for x in output_data[0]]
        }
        return predictions
    except Exception as e:
        logger.error(f"Error during inference: {e}")
        raise

@app.route('/')
def home():
    return "Flask App is Running!"

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({
        'status': 'healthy',
        'model_loaded': interpreter is not None,
        'endpoints': ['/health', '/predict', '/process-image']
    })
```

```
@app.route('/process-image', methods=['POST'])
def process_image():
    """Endpoint for image processing without prediction"""
    try:
        data = request.get_json()
        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400

        # Decode base64 image
        image_data = decode_base64_image(data['image'])

        # Preprocess image
        processed_image = preprocess_image(image_data)

        return jsonify({
            'status': 'success',
            'message': 'Image processed successfully',
            'image_shape': processed_image.shape,
            'data_range': {
                'min': float(processed_image.min()),
                'max': float(processed_image.max())
            }
        })

    except Exception as e:
        logger.error(f"Image processing error: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/predict', methods=['POST'])
def predict():
    """Endpoint for image prediction"""
    try:
        data = request.get_json()
        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400

        logger.info("Received prediction request")

        # Decode base64 image
        image_data = decode_base64_image(data['image'])

        # Preprocess image
        processed_image = preprocess_image(image_data)

        # Make prediction
        predictions = predict_with_tflite(processed_image)
```

```
        logger.info(f"Prediction completed: class
{predictions['predicted_class']}")

        return jsonify({
            'status': 'success',
            'predictions': predictions
        })

    except Exception as e:
        logger.error(f"Prediction error: {e}")
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=False)

==================================================
```

[18]:
```
# Create the correct enhanced Flask app
enhanced_app_code = '''from flask import Flask, request, jsonify
import base64
import numpy as np
from PIL import Image
import io
import tensorflow as tf
import logging

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

app = Flask(__name__)

# Load TensorFlow Lite model
try:
    interpreter = tf.lite.Interpreter(model_path="model.tflite")
    interpreter.allocate_tensors()
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    logger.info("TensorFlow Lite model loaded successfully")
except Exception as e:
    logger.error(f"Error loading model: {e}")
    interpreter = None

def decode_base64_image(base64_string):
    """Decode base64 encoded image string"""
    try:
        if ',' in base64_string:
```

```
            base64_string = base64_string.split(',')[1]
        image_data = base64.b64decode(base64_string)
        return image_data
    except Exception as e:
        logger.error(f"Error decoding base64 image: {e}")
        raise

def preprocess_image(image_data, target_size=(224, 224)):
    """Preprocess image for model inference"""
    try:
        image = Image.open(io.BytesIO(image_data))
        if image.mode != 'RGB':
            image = image.convert('RGB')
        image = image.resize(target_size)
        image_array = np.array(image, dtype=np.float32) / 255.0
        image_array = np.expand_dims(image_array, axis=0)
        return image_array
    except Exception as e:
        logger.error(f"Error preprocessing image: {e}")
        raise

def predict_with_tflite(preprocessed_image):
    """Perform inference using TensorFlow Lite model"""
    try:
        input_details = interpreter.get_input_details()
        output_details = interpreter.get_output_details()
        interpreter.set_tensor(input_details[0]['index'], preprocessed_image)
        interpreter.invoke()
        output_data = interpreter.get_tensor(output_details[0]['index'])
        predictions = {
            'predictions': output_data.tolist(),
            'predicted_class': int(np.argmax(output_data)),
            'confidence': float(np.max(output_data)),
            'all_confidences': [float(x) for x in output_data[0]]
        }
        return predictions
    except Exception as e:
        logger.error(f"Error during inference: {e}")
        raise

@app.route('/')
def home():
    return "Flask App is Running!"

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({
```

```
        'status': 'healthy',
        'model_loaded': interpreter is not None,
        'endpoints': ['/health', '/predict', '/process-image']
    })

@app.route('/process-image', methods=['POST'])
def process_image():
    """Endpoint for image processing without prediction"""
    try:
        data = request.get_json()
        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400

        # Decode base64 image
        image_data = decode_base64_image(data['image'])

        # Preprocess image
        processed_image = preprocess_image(image_data)

        return jsonify({
            'status': 'success',
            'message': 'Image processed successfully',
            'image_shape': processed_image.shape,
            'data_range': {
                'min': float(processed_image.min()),
                'max': float(processed_image.max())
            }
        })

    except Exception as e:
        logger.error(f"Image processing error: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/predict', methods=['POST'])
def predict():
    """Endpoint for image prediction"""
    try:
        data = request.get_json()
        if not data or 'image' not in data:
            return jsonify({'error': 'No image data provided'}), 400

        logger.info("Received prediction request")

        # Decode base64 image
        image_data = decode_base64_image(data['image'])

        # Preprocess image
```

```
        processed_image = preprocess_image(image_data)

        # Make prediction
        predictions = predict_with_tflite(processed_image)

        logger.info(f"Prediction completed: class␣
 ↪{predictions['predicted_class']}")

        return jsonify({
            'status': 'success',
            'predictions': predictions
        })

    except Exception as e:
        logger.error(f"Prediction error: {e}")
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=False)
'''

# Write the enhanced app
with open('app.py', 'w') as f:
    f.write(enhanced_app_code)

print(" Enhanced Flask app created successfully!")
print(" File: app.py")
print(" Ready to run with all endpoints!")
```

```
Enhanced Flask app created successfully!
File: app.py
Ready to run with all endpoints!
```

[19]:
```
import os

# Check if model file exists
if os.path.exists('model.tflite'):
    file_size = os.path.getsize('model.tflite')
    print(f" Model file found: model.tflite ({file_size} bytes)")
else:
    print(" Model file not found! Let's create it...")

    # Create a simple model if missing
    import tensorflow as tf

    model = tf.keras.Sequential([
        tf.keras.layers.InputLayer(input_shape=(224, 224, 3)),
```

```
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    tflite_model = converter.convert()

    with open('model.tflite', 'wb') as f:
        f.write(tflite_model)

    print(" Demo model created: model.tflite")
```

```
Model file found: model.tflite (6832 bytes)
```

[ ]:
```
# Run the enhanced Flask app
print("Starting Enhanced Flask App...")
print("All endpoints available:")
print("• http://localhost:5000/ - Home")
print("• http://localhost:5000/health - Health check")
print("• POST http://localhost:5000/predict - Image prediction")
print("• POST http://localhost:5000/process-image - Image processing")
print("\nApp is running... (Click Stop button when done)")

!python app.py
```

```
Starting Enhanced Flask App…
All endpoints available:
• http://localhost:5000/ - Home
• http://localhost:5000/health - Health check
• POST http://localhost:5000/predict - Image prediction
• POST http://localhost:5000/process-image - Image processing

App is running… (Click Stop button when done)
```

[ ]:

[ ]:

# Test All Endpoints

November 30, 2025

## 0.1 Test All Endpoints

**DONG**

[1]:
```
import requests

print("Testing Enhanced App Endpoints...")

# Test health endpoint
print("1. Health endpoint:")
response = requests.get('http://localhost:5000/health')
print(f"   Status: {response.status_code}")
print(f"   Response: {response.json()}")

print("2. Home endpoint:")
response = requests.get('http://localhost:5000/')
print(f"   Status: {response.status_code}")
print(f"   Response: {response.text}")

print("  If you see model_loaded: True, everything is working!")
```

```
Testing Enhanced App Endpoints…
1. Health endpoint:
   Status: 200
   Response: {'endpoints': ['/health', '/predict', '/process-image'],
'model_loaded': True, 'status': 'healthy'}
2. Home endpoint:
   Status: 200
   Response:
    <h1>  Enhanced Flask App with TensorFlow Lite</h1>
    <p>Endpoints available:</p>
    <ul>
        <li><a href="/health">/health</a> - Health check</li>
        <li>/predict - POST endpoint for image prediction</li>
        <li>/process-image - POST endpoint for image processing</li>
    </ul>

  If you see model_loaded: True, everything is working!
```

[ ]: