Lab 07: Deep Learning for Gender Classification from Facial Images

Lab 07

Deep Learning for Gender Classification from Facial Images

1. Introduction and Objectives

1.1 Introduction

Deep Learning is a subset of machine learning that uses artificial neural networks with multiple layers (hence "deep") to learn hierarchical representations of data. Unlike traditional machine learning approaches that require manually engineered features, deep neural networks automatically learn feature representations at multiple levels of abstraction. This is particularly powerful for image data, where raw pixel values can be transformed into increasingly complex features (edges → shapes → facial parts → gender characteristics).

In this lab, we will build a neural network to classify the gender of individuals from facial images. The task requires us to:

1. Load and preprocess image data from the UTKFace dataset (available on Kaggle)
2. Design and train a deep neural network using TensorFlow/Keras
3. Evaluate model performance using appropriate metrics
4. Save and persist the trained model for deployment
5. Create a prediction interface similar to Lab 06 but using deep learning

This lab bridges the gap between feature engineering (Labs 03-04) and production deployment (Lab 06) by demonstrating how neural networks eliminate manual feature engineering while learning powerful representations directly from pixel data.

1.2 Learning Objectives

Upon completing this lab, students will be able to:

1. Understand the fundamentals of neural networks: layers, neurons, activation functions, and backpropagation
2. Preprocess image data (normalization, resizing, augmentation) for deep learning
3. Build and train a Convolutional Neural Network (CNN) using TensorFlow/Keras
4. Implement data augmentation techniques to improve model generalization
5. Evaluate deep learning models using accuracy, precision, recall, and confusion matrices
6. Save and load trained neural network models for inference
7. Implement a REST API for gender prediction using the trained model (extending Lab 06 concepts)

2. Setup and Prerequisites

2.1 Required Libraries

Install the following libraries:

```
pip install tensorflow
pip install keras
pip install opencv-python
pip install numpy
pip install pandas
pip install scikit-learn
pip install matplotlib
pip install pillow
pip install requests
pip install fastapi
pip install uvicorn[standard]
pip install joblib
```

2.2 Dataset Preparation

This lab uses the **UTKFace dataset** from Kaggle, which contains over 20,000 facial images with labeled age, gender, and ethnicity. Download the dataset:

Dataset Link: https://www.kaggle.com/datasets/jangedoo/utkface-new

Steps to download:

1. Create a Kaggle account and API token
2. Place `kaggle.json` in `~/.kaggle/` directory
3. Run: `kaggle datasets download -d jangedoo/utkface-new`
4. Extract the dataset to `data/utkface/`

Dataset Structure:

```
data/
├── utkface/
│   ├── [age]_[gender]_[race]_[date&time].jpg
│   ├── [age]_[gender]_[race]_[date&time].jpg
│   └── ... (20,000+ images)
```

Gender labels: 0 = Male, 1 = Female

3. Part A: Data Preprocessing and Exploration

3.1 Task A.1: Load and Explore Image Data

Goal: Load the UTKFace dataset, parse filenames for labels, and visualize sample images.

```
import os
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
import warnings
warnings.filterwarnings('ignore')

# Configuration
DATA_DIR = 'data/utkface'
IMG_SIZE = 64  # Resize all images to 64x64
BATCH_SIZE = 32
VALIDATION_SPLIT = 0.2
TEST_SPLIT = 0.1

# 1. Parse filenames and extract labels
def load_dataset(data_dir, img_size=64, max_samples=None):
    """
    Load images and extract labels from filenames.
    Filename format: [age]_[gender]_[race]_[date&time].jpg
    Gender: 0=Male, 1=Female
    """
    images = []
    labels = []
    skipped = 0

    image_files = [f for f in os.listdir(data_dir) if f.endswith('.jpg')]

    if max_samples:
        image_files = image_files[:max_samples]

    print(f"Total images found: {len(image_files)}")

    for idx, filename in enumerate(image_files):
        try:
            # Parse filename
            parts = filename.split('_')
            if len(parts) < 3:
                skipped += 1
                continue

            age = int(parts[0])
            gender = int(parts[1])  # 0=Male, 1=Female

            # Load and preprocess image
            img_path = os.path.join(data_dir, filename)
            img = cv2.imread(img_path)

            if img is None:
                skipped += 1
                continue

            # Convert BGR to RGB
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            # Resize to fixed size
```

```
            img = cv2.resize(img, (img_size, img_size))

            images.append(img)
            labels.append(gender)

            if (idx + 1) % 5000 == 0:
                print(f"Processed {idx + 1} images...")

        except Exception as e:
            skipped += 1
            continue

    print(f"\n--- Dataset Loading Complete ---")
    print(f"Successfully loaded: {len(images)} images")
    print(f"Skipped (corrupted/invalid): {skipped} images")

    return np.array(images), np.array(labels)

# 2. Load the dataset (using max_samples for faster iteration)
# For full training, remove max_samples or set to None
print("Loading UTKFace dataset...")
X, y = load_dataset(DATA_DIR, img_size=IMG_SIZE, max_samples=5000)

# 3. Analyze dataset
print(f"\n--- Dataset Statistics ---")
print(f"Images shape: {X.shape}")
print(f"Labels shape: {y.shape}")
print(f"Gender distribution: {np.bincount(y)}")
print(f"Male (0): {np.sum(y == 0)} images")
print(f"Female (1): {np.sum(y == 1)} images")
print(f"Image dtype: {X.dtype}")
print(f"Pixel value range: [{X.min()}, {X.max()}]")

# 4. Visualize sample images
fig, axes = plt.subplots(2, 5, figsize=(15, 6))
for idx in range(10):
    row = idx // 5
    col = idx % 5
    ax = axes[row, col]
    ax.imshow(X[idx])
    gender_label = "Female" if y[idx] == 1 else "Male"
    ax.set_title(f"Gender: {gender_label}")
    ax.axis('off')

plt.tight_layout()
plt.savefig('sample_images.png', dpi=100, bbox_inches='tight')
print("\nSample images visualization saved as 'sample_images.png'")
plt.show()
```

3.2 Task A.2: Normalize and Split Data

Goal: Normalize pixel values and split data into training, validation, and test sets.

```
from sklearn.model_selection import train_test_split

# 1. Normalize pixel values to [0, 1]
```

```
print("Normalizing pixel values...")
X_normalized = X.astype('float32') / 255.0
print(f"Normalized pixel range: [{X_normalized.min()},
{X_normalized.max()}]")

# 2. Split into train, validation, test
print("\nSplitting dataset...")
# First split: separate test set (10% of total)
X_temp, X_test, y_temp, y_test = train_test_split(
    X_normalized, y, test_size=TEST_SPLIT, random_state=42, stratify=y
)

# Second split: separate validation from training (20% of remaining = ~18% of
total)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=VALIDATION_SPLIT, random_state=42,
stratify=y_temp
)

print(f"Training set: {X_train.shape}")
print(f"Validation set: {X_val.shape}")
print(f"Test set: {X_test.shape}")

# 3. Verify class balance in splits
print(f"\nClass distribution in training set:")
print(f"  Male: {np.sum(y_train == 0)}, Female: {np.sum(y_train == 1)}")
print(f"Class distribution in validation set:")
print(f"  Male: {np.sum(y_val == 0)}, Female: {np.sum(y_val == 1)}")
print(f"Class distribution in test set:")
print(f"  Male: {np.sum(y_test == 0)}, Female: {np.sum(y_test == 1)}")
```

3.3 Discussion Questions (Part A)

1. Why is normalizing pixel values to the range [0, 1] important before feeding images to a neural network?
2. What is the purpose of splitting data into training, validation, and test sets in deep learning?
3. What could happen if the dataset had imbalanced classes (e.g., 90% male, 10% female)?
4. How does image resizing to 64×64 affect the information contained in the original images? What trade-offs are made?

4. Part B: Building and Training a Deep Neural Network

4.1 Task B.1: Design and Train a CNN Model

Goal: Flatten images into 1D vectors and build a fully connected neural network with multiple hidden layers.

```
import tensorflow as tf
from tensorflow import keras
```

```
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,
ModelCheckpoint

# 1. Flatten the training, validation, and test data
print("Flattening image data for fully connected network...")
# Reshape images from (N, 64, 64, 3) to (N, 12288)
# because 64 * 64 * 3 = 12,288 input features
X_train_flattened = X_train.reshape(X_train.shape[0], -1)
X_val_flattened = X_val.reshape(X_val.shape[0], -1)
X_test_flattened = X_test.reshape(X_test.shape[0], -1)

print(f"Original shape: {X_train.shape}")
print(f"Flattened shape: {X_train_flattened.shape}")
print(f"Total input features: {X_train_flattened.shape[1]}")

# 2. Create a Fully Connected Neural Network
print("\nBuilding Fully Connected Neural Network...")

input_dim = X_train_flattened.shape[1]  # 12,288 features

model = models.Sequential([
    # Input layer implicit in first Dense layer
    # Hidden Layer 1 - 1024 neurons
    layers.Dense(1024, activation='relu', input_dim=input_dim),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    # Hidden Layer 2 - 512 neurons
    layers.Dense(512, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    # Hidden Layer 3 - 256 neurons
    layers.Dense(256, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    # Hidden Layer 4 - 128 neurons
    layers.Dense(128, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.2),

    # Hidden Layer 5 - 64 neurons
    layers.Dense(64, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.2),

    # Output layer (binary classification)
    layers.Dense(1, activation='sigmoid')
])

# 3. Compile the model
print("\nCompiling model...")
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
```

```
    metrics=['accuracy',
             keras.metrics.Precision(name='precision'),
             keras.metrics.Recall(name='recall')]
)

# 4. Display model architecture
print("\n--- Model Architecture ---")
model.summary()

# Calculate number of parameters in first Dense layer
print(f"\n--- Parameter Analysis ---")
print(f"Input features: {input_dim}")
print(f"First hidden layer neurons: 1024")
print(f"Parameters in first layer (weights): {input_dim * 1024} + 1024 (bias)
= {input_dim * 1024 + 1024}")

# 5. Define callbacks for training
callbacks = [
    EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True,
        verbose=1
    ),
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=5,
        min_lr=1e-7,
        verbose=1
    ),
    ModelCheckpoint(
        'models/best_gender_model.h5',
        monitor='val_accuracy',
        save_best_only=True,
        verbose=1
    )
]

# 6. Train the model on flattened data
print("\n--- Starting Training ---")
history = model.fit(
    X_train_flattened, y_train,
    batch_size=BATCH_SIZE,
    epochs=2,
    validation_data=(X_val_flattened, y_val),
    callbacks=callbacks,
    verbose=1
)
print("\n--- Training Complete ---")
```

4.2 Task B.2: Evaluate Model Performance

Goal: Evaluate the trained model on the test set and visualize training history.

```
from sklearn.metrics import confusion_matrix, classification_report,
roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Evaluate on test set
print("Evaluating model on test set...")
test_loss, test_accuracy, test_precision, test_recall = model.evaluate(
    X_test, y_test, verbose=0
)

print(f"\n--- Test Set Performance ---")
print(f"Loss: {test_loss:.4f}")
print(f"Accuracy: {test_accuracy:.4f}")
print(f"Precision: {test_precision:.4f}")
print(f"Recall: {test_recall:.4f}")

# 2. Get predictions
print("\nGenerating predictions on test set...")
y_pred_proba = model.predict(X_test)
y_pred = (y_pred_proba > 0.5).astype(int).flatten()

# 3. Detailed classification metrics
print("\n--- Detailed Classification Report ---")
print(classification_report(y_test, y_pred, target_names=['Male', 'Female']))

# 4. Confusion matrix
cm = confusion_matrix(y_test, y_pred)
print(f"\nConfusion Matrix:")
print(cm)

# 5. Plot confusion matrix
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[0],
            xticklabels=['Male', 'Female'],
            yticklabels=['Male', 'Female'])
axes[0].set_title('Confusion Matrix')
axes[0].set_ylabel('True Label')
axes[0].set_xlabel('Predicted Label')

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

axes[1].plot(fpr, tpr, color='darkorange', lw=2,
             label=f'ROC curve (AUC = {roc_auc:.3f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlim([0.0, 1.0])
axes[1].set_ylim([0.0, 1.05])
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve')
axes[1].legend(loc="lower right")

plt.tight_layout()
```

```
plt.savefig('evaluation_metrics.png', dpi=100, bbox_inches='tight')
print("\nEvaluation plots saved as 'evaluation_metrics.png'")
plt.show()

# 6. Plot training history
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Accuracy
axes[0].plot(history.history['accuracy'], label='Training Accuracy')
axes[0].plot(history.history['val_accuracy'], label='Validation Accuracy')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Accuracy')
axes[0].set_title('Model Accuracy Over Epochs')
axes[0].legend()
axes[0].grid(True)

# Loss
axes[1].plot(history.history['loss'], label='Training Loss')
axes[1].plot(history.history['val_loss'], label='Validation Loss')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Loss')
axes[1].set_title('Model Loss Over Epochs')
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
plt.savefig('training_history.png', dpi=100, bbox_inches='tight')
print("Training history plots saved as 'training_history.png'")
plt.show()
```

4.3 Discussion Questions (Part B)

1. Explain the purpose of convolutional layers in a CNN. How do they differ from fully connected layers?
2. What is the role of batch normalization in the network? How does it improve training?
3. Why do we use dropout layers? How does it prevent overfitting?
4. Compare the learning curves (training vs. validation loss). What does it tell us about model generalization?
5. If the model shows signs of overfitting, what techniques could you apply to improve performance?

5. Part C: Model Persistence and Deployment

5.1 Task C.1: Save and Load the Trained Model

Goal: Persist the trained model and create utility functions for loading and inference.

```
import os

import json
```

```
import numpy as np

import cv2

import tensorflow as tf

from tensorflow import keras



# --- Assumptions / variables you must have defined earlier in your script --
-

# model               -> trained Keras model (expects flattened input if
trained that way)

# IMG_SIZE            -> integer, e.g. 128

# X_train, X_val, X_test, X_test_flattened, y_test -> datasets used earlier

# test_accuracy, test_precision, test_recall -> metrics computed earlier



# Example placeholders (uncomment / replace with real values)

# IMG_SIZE = 128

# model = ...          # your trained model

# X_train, X_val, X_test = ...

# X_test_flattened = ... # flattened test images if you trained on flattened
inputs

# y_test = ...



os.makedirs('models', exist_ok=True)



print("Saving trained model...")



# 1) Save as H5 (classic) — fine for later loading in Python

h5_path = 'models/gender_classifier_model.h5'
```

```
model.save(h5_path)

print(f"✓ Saved as H5 format: {h5_path}")


# 2) Save as SavedModel directory (for TF Serving / TFLite conversion)

# In Keras 3 use model.export(...) to create a TensorFlow SavedModel
directory

saved_model_dir = 'models/gender_classifier_savedmodel'

# remove existing folder if you want to overwrite (optional)

if os.path.exists(saved_model_dir):

    import shutil

    shutil.rmtree(saved_model_dir)

model.export(saved_model_dir)

print(f"✓ Exported SavedModel directory: {saved_model_dir}")


# (Alternative) Save in native Keras single-file format:

# model.save('models/gender_classifier.keras')

# print("✓ Saved as .keras single-file format")


# 3) Save metadata (ensure values are JSON-serializable)

input_shape = [IMG_SIZE, IMG_SIZE, 3]

flattened_features = int(np.prod(input_shape))  # compute dynamically


metadata = {

    'model_name': 'Gender Classifier Fully Connected Network',

    'input_shape': input_shape,

    'flattened_input_features': flattened_features,
```

```
    'output_classes': ['Male', 'Female'],

    'normalization': 'pixel values divided by 255.0',

    'test_accuracy': float(test_accuracy) if 'test_accuracy' in globals()
else None,

    'test_precision': float(test_precision) if 'test_precision' in globals()
else None,

    'test_recall': float(test_recall) if 'test_recall' in globals() else
None,

    'training_samples': int(len(X_train)) if 'X_train' in globals() else
None,

    'validation_samples': int(len(X_val)) if 'X_val' in globals() else None,

    'test_samples': int(len(X_test)) if 'X_test' in globals() else None

}



with open('models/model_metadata.json', 'w') as f:

    json.dump(metadata, f, indent=4)

print("✓ Saved metadata: models/model_metadata.json")



# 4) Load and verify the model (load from H5 for quick test)

print("\n--- Loading H5 model ---")

loaded_model = keras.models.load_model(h5_path)

print("✓ Model successfully loaded")

print(f"Model input shape: {loaded_model.input_shape}")

print(f"Model output shape: {loaded_model.output_shape}")



# 5) Verify predictions are identical (use same input preprocessing)

print("\nVerifying prediction consistency...")
```

```
# Prepare a sample. Use X_test_flattened if your model expects flattened
vectors;

# otherwise use X_test with shape (H,W,3) if model expects images.

if 'X_test_flattened' in globals():

    sample_idx = np.random.randint(0, len(X_test_flattened))

    sample_image = X_test_flattened[sample_idx:sample_idx+1]  # shape (1,
flattened_features)

else:

    # fallback if you only have image arrays

    sample_idx = np.random.randint(0, len(X_test))

    # make sure to normalize / flatten the same way used in training

    sample_image = X_test[sample_idx:sample_idx+1]

    sample_image = sample_image.reshape((1, -1)) if
loaded_model.input_shape[-1] == flattened_features else sample_image



original_pred = model.predict(sample_image, verbose=0)

loaded_pred = loaded_model.predict(sample_image, verbose=0)



print(f"Original model prediction (first element):
{original_pred.flatten()[0]:.6f}")

print(f"Loaded model prediction (first element):
{loaded_pred.flatten()[0]:.6f}")

print(f"Predictions match (allclose): {np.allclose(original_pred,
loaded_pred)}")



# 6) Inference helper (robustified)

def predict_gender(image_path, model=loaded_model, img_size=IMG_SIZE):

    """

    Predict gender from an image file.
```

```
    Returns:
        dict with 'gender', 'confidence', 'raw_prediction' or 'error'
    """
    try:
        img = cv2.imread(image_path)
        if img is None:
            return {'error': f'Could not load image: {image_path}'}
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (img_size, img_size))
        img = img.astype('float32') / 255.0

        # Determine what model expects (flattened vs image)
        expected_shape = model.input_shape  # (None, N) or (None, H, W, C)
        if len(expected_shape) == 2:
            # flattened vector expected (None, features)
            img_flat = img.flatten().astype('float32')
            if img_flat.size != expected_shape[1]:
                return {'error': f'Flattened size mismatch: expected
{expected_shape[1]}, got {img_flat.size}'}
            inp = np.expand_dims(img_flat, axis=0)
        else:
            # image tensor expected (None, H, W, C)
            inp = np.expand_dims(img, axis=0)

        pred = model.predict(inp, verbose=0)[0]
```

```
        # For two-class softmax output, choose index; if single-output
sigmoid, interpret differently

        if pred.size == 1:

            raw = float(pred[0])

            gender = 'Female' if raw > 0.5 else 'Male'

            confidence = raw if raw > 0.5 else (1 - raw)

        else:

            # multi-class softmax

            idx = int(np.argmax(pred))

            raw = float(pred[idx])

            gender = ['Male', 'Female'][idx] if len(pred) == 2 else
f'class_{idx}'

            confidence = raw



        return {'gender': gender, 'confidence': float(confidence),
'raw_prediction': float(raw)}



    except Exception as e:

        return {'error': str(e)}



# 7) Test the inference function using a sample from X_test

print("\n--- Testing Inference Function ---")

test_image_idx = 0

# If X_test contains normalized floats, convert back to uint8 for writing a
test file

test_image_array = (X_test[test_image_idx] * 255).astype(np.uint8) if
X_test.dtype != np.uint8 else X_test[test_image_idx]

test_image_path = 'test_image_sample.jpg'
```

```
cv2.imwrite(test_image_path, cv2.cvtColor(test_image_array,
cv2.COLOR_RGB2BGR))



result = predict_gender(test_image_path)

actual_gender = 'Female' if int(y_test[test_image_idx]) == 1 else 'Male'

if 'error' in result:

    print("Inference error:", result['error'])

else:

    print(f"Predicted: {result['gender']} (confidence:
{result['confidence']:.4f})")

    print(f"Actual: {actual_gender}")
```

5.2 Task C.2: Create FastAPI Deployment Service

Goal: Build a REST API for the gender classification model (extending Lab 06 concepts).

Create `app/schemas.py`:

```
from pydantic import BaseModel
from typing import Literal

class GenderPredictRequest(BaseModel):
    image_base64: str

class GenderPredictResponse(BaseModel):
    gender: Literal["Male", "Female"]
    confidence: float
    raw_prediction: float

class ModelMetadataResponse(BaseModel):
    model_name: str
    input_shape: list
    output_classes: list
    test_accuracy: float
    test_precision: float
    test_recall: float
```

Create `app/main.py`:

```
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
import numpy as np
import cv2
import base64
import json
```

```
import logging
import tensorflow as tf
from .schemas import GenderPredictRequest, GenderPredictResponse,
ModelMetadataResponse

app = FastAPI(title="Gender Classification API")

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

# CORS configuration
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Model paths
MODEL_PATH = "models/gender_classifier_model.h5"
METADATA_PATH = "models/model_metadata.json"
IMG_SIZE = 64

# Global variables for loaded artifacts
model = None
metadata = None

def load_artifacts():
    global model, metadata
    try:
        model = tf.keras.models.load_model(MODEL_PATH)
        logger.info("✓ Model loaded successfully")

        with open(METADATA_PATH, 'r') as f:
            metadata = json.load(f)
        logger.info("✓ Metadata loaded successfully")
    except Exception as e:
        logger.error(f"Error loading artifacts: {e}")
        raise

@app.on_event("startup")
def startup_event():
    load_artifacts()
    logger.info("Application startup complete")

@app.get('/health')
def health():
    return {'status': 'healthy', 'service': 'Gender Classification API'}

@app.get('/metadata')
def get_metadata():
    return metadata

@app.post('/predict', response_model=GenderPredictResponse)
def predict(req: GenderPredictRequest):
```

```
    """
    Predict gender from base64-encoded image.
    """
    try:
        # Decode base64 image
        image_data = base64.b64decode(req.image_base64)
        nparr = np.frombuffer(image_data, np.uint8)
        img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)

        if img is None:
            raise ValueError("Could not decode image")

        # Preprocess
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
        img = img.astype('float32') / 255.0

        # Predict
        prediction = model.predict(np.array([img]), verbose=0)[0][0]

        gender = 'Female' if prediction > 0.5 else 'Male'
        confidence = prediction if prediction > 0.5 else (1 - prediction)

        return GenderPredictResponse(
            gender=gender,
            confidence=float(confidence),
            raw_prediction=float(prediction)
        )

    except Exception as e:
        logger.exception("Prediction error")
        raise HTTPException(status_code=400, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Create `requirements.txt`:

```
tensorflow
keras
opencv-python
numpy
pandas
scikit-learn
matplotlib
pillow
fastapi
uvicorn[standard]
joblib
```

5.3 Running the API

```
# From project root
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

```
# API documentation will be available at http://localhost:8000/docs
```

Test with curl:

```
# Example: Convert an image to base64 and send
python -c "
import base64
with open('test_image_sample.jpg', 'rb') as f:
    img_b64 = base64.b64encode(f.read()).decode()
print(img_b64)
" > image_b64.txt

curl -X POST "http://localhost:8000/predict" \
  -H "Content-Type: application/json" \
  -d '{"image_base64":"'$(cat image_b64.txt)'"}' | python -m json.tool
```

5.4 Discussion Questions (Part C)

1. What are the advantages of saving a model in SavedModel format versus H5 format for production deployment?
2. How would you implement model versioning in the FastAPI service to support A/B testing?
3. What additional preprocessing steps could you add to the API to make it more robust to different image formats and sizes?
4. How would you monitor inference latency and model performance in production?

6. Part D: Advanced Concepts and Extensions

6.1 Task D.1: Transfer Learning (Optional Advanced Exercise)

```
# Load pre-trained model
print("Building transfer learning model...")

base_model = tf.keras.applications.MobileNetV2(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,
    weights='imagenet'
)

# Freeze base model weights
base_model.trainable = False

# Add custom top layers
transfer_model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])

transfer_model.compile(
```

```
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

print(transfer_model.summary())

# Train with fewer epochs since base is pre-trained
history_transfer = transfer_model.fit(
    train_generator,
    steps_per_epoch=len(X_train) // BATCH_SIZE,
    epochs=20,
    validation_data=(X_val, y_val),
    callbacks=callbacks,
    verbose=1
)

print("\n✓ Transfer learning model trained")
```

6.2 Task D.2: Model Interpretability with Visualization

```
# Visualize learned features from first convolutional layer
def visualize_filters(model, layer_idx=0, num_filters=16):
    layer = model.layers[layer_idx]
    filters, biases = layer.get_weights()

    fig, axes = plt.subplots(4, 4, figsize=(12, 12))
    for i in range(num_filters):
        ax = axes[i // 4, i % 4]
        # Normalize filters for visualization
        f = filters[:, :, :, i]
        f = (f - f.min()) / (f.max() - f.min() + 1e-8)
        # Take mean across color channels
        ax.imshow(np.mean(f, axis=2), cmap='viridis')
        ax.set_title(f'Filter {i+1}')
        ax.axis('off')

    plt.tight_layout()
    plt.savefig('learned_filters.png', dpi=100, bbox_inches='tight')
    print("Learned filters visualization saved")
    plt.show()

visualize_filters(model, layer_idx=0)
```

7. Lab Submission Requirements

Students should submit:

1. **Training script** (`train_gender_classifier.py`) containing:
    - Data loading and preprocessing (Part A)
    - Model architecture and training (Part B)
    - Model evaluation with metrics and visualizations
2. **Model files**:

    - `models/gender_classifier_model.h5` (trained model)
    - `models/model_metadata.json` (model metadata)
3. **Deployment code**:
    - `app/main.py` (FastAPI application)
    - `app/schemas.py` (Pydantic models)
    - `requirements.txt`
    - `Dockerfile` (optional but recommended)
4. **Report** (2-3 pages) including:
    - Dataset description and preprocessing steps
    - Model architecture rationale
    - Training curves and performance metrics
    - Discussion of results and limitations
    - Answers to all discussion questions