Deep Learning-Based Gender Classification System
Documentation

Computer Vision & Deep Learning Project

April 25, 2025

Contents

Chapter 1

Introduction

1.1 Project Overview

This document provides comprehensive documentation for a gender classification system developed using deep learning techniques. The system uses convolutional neural networks (CNNs) to classify human images as either men or women. The project consists of two main components: a training module for developing and evaluating the model, and a graphical user interface (GUI) application for real-time predictions.

1.2 Objectives

The primary objectives of this project are:

- To develop a robust CNN model for binary gender classification
- To handle potential dataset imbalances
- To implement effective data augmentation techniques
- To create a user-friendly interface for model predictions
- To collect and analyze user feedback on model performance

1.3 System Requirements

- Python 3.6 or higher
- PyTorch 1.7 or higher
- Torchvision
- Matplotlib
- Tkinter (for GUI)
- Pillow (for image processing)
- CUDA-compatible GPU (recommended but not required)

Chapter 2

Dataset

2.1 Dataset Structure

The dataset is organized in the following directory structure:

```
real_dataset/
|__ train/
|    |__ humans/
|         |__ men/
|         |__ women/
|__ test/
     |__ humans/
          |__ men/
          |__ women/
```

2.2 Data Preprocessing

The input images undergo several preprocessing steps:

- Resizing to 64×64 pixels
- Normalization with mean (0.5, 0.5, 0.5) and standard deviation (0.5, 0.5, 0.5)
- For training data: additional augmentation techniques

2.3 Data Augmentation

Data augmentation is implemented to improve model generalization and robustness. The augmentation pipeline includes:

```
train_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(20),
    transforms.RandomCrop(64, padding=8),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation
    =0.2, hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Listing 2.1: Data augmentation implementation

Chapter 3

Model Architecture

3.1 Convolutional Neural Network

The gender classification model uses a custom CNN architecture consisting of four convolutional blocks followed by three fully connected layers.

```python
class GenderCNN(nn.Module):
    def __init__(self, num_classes=2):
        super(GenderCNN, self).__init__()
        # First block
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)

        # Second block
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)

        # Third block
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)

        # Fourth block
        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(512)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.3)
        self.dropout2 = nn.Dropout(0.4)

        # Fully connected layers
        self.fc1 = nn.Linear(512 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = self.pool(torch.relu(self.bn4(self.conv4(x))))

        x = x.view(-1, 512 * 4 * 4)
        x = self.dropout1(torch.relu(self.fc1(x)))
        x = self.dropout2(torch.relu(self.fc2(x)))
```

```python
        x = self.fc3(x)
        return x
```

Listing 3.1: CNN architecture definition

3.2 Architecture Components

1. **Convolutional Blocks:** Four sequential blocks with increasing filter sizes (64, 128, 256, 512)
2. **Batch Normalization:** Applied after each convolutional layer
3. **Max Pooling:** Applied after each convolutional block to reduce spatial dimensions
4. **Dropout:** Applied after the first two fully connected layers (30% and 40% rates)
5. **Fully Connected Layers:** Three layers (512 $\rightarrow$ 256 $\rightarrow$ 2) for final classification

3.3 Model Parameters

Layer Type	Output Shape	Parameters
Input	(3, 64, 64)	0
Conv2d + BN + ReLU + MaxPool	(64, 32, 32)	1,856
Conv2d + BN + ReLU + MaxPool	(128, 16, 16)	73,984
Conv2d + BN + ReLU + MaxPool	(256, 8, 8)	295,424
Conv2d + BN + ReLU + MaxPool	(512, 4, 4)	1,180,160
Flatten	(8192)	0
Linear + ReLU + Dropout	(512)	4,195,328
Linear + ReLU + Dropout	(256)	131,328
Linear	(2)	514

Table 3.1: Model architecture summary

Chapter 4

Training Process

4.1 Training Configuration

Parameter	Value
Optimizer	Adam
Learning Rate	0.001
Weight Decay	1e-5
Loss Function	Cross Entropy Loss
Batch Size	32
Max Epochs	30
Early Stopping Patience	5 epochs

Table 4.1: Training configuration parameters

4.2 Learning Rate Schedule

A ReduceLROnPlateau scheduler is used to adaptively adjust the learning rate during training:

- Monitors validation loss
- Reduces learning rate by factor of 0.5 if no improvement for 3 epochs
- Minimum learning rate: 1e-6

4.3 Class Balancing

To address potential class imbalance in the dataset, the model uses a weighted random sampler:

```
# Calculate class weights to handle imbalanced data
class_counts = [0] * len(trainset.classes)
for _, label in trainset.samples:
    class_counts[label] += 1

# Create weighted sampler for imbalanced classes
```

```
class_weights = [1.0 / count for count in class_counts]
sample_weights = [class_weights[label] for _, label in trainset.samples
    ]
sampler = WeightedRandomSampler(
    weights=sample_weights,
    num_samples=len(sample_weights),
    replacement=True
)
```

Listing 4.1: Class balancing implementation

4.4 Early Stopping

Early stopping is implemented to prevent overfitting:

- Monitors validation accuracy
- Stops training if no improvement for 5 consecutive epochs
- Saves the best model based on validation accuracy

Chapter 5

Evaluation Metrics

5.1 Model Evaluation

The model is evaluated using the following metrics:

- Overall accuracy
- Class-wise accuracy
- Confusion matrix

```python
def evaluate_model(model, testloader, classes):
    model.eval()

    # Collect predictions and ground truth
    class_correct = [0] * len(classes)
    class_total = [0] * len(classes)

    confusion_matrix = np.zeros((len(classes), len(classes)), dtype=int
    )

    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)

            for i in range(len(labels)):
                label = labels[i]
                pred = predicted[i]
                confusion_matrix[label][pred] += 1
                if label == pred:
                    class_correct[label] += 1
                class_total[label] += 1
```

Listing 5.1: Model evaluation implementation

5.2 Visualization

Training progress and evaluation results are visualized using:

- Training and validation loss curves
- Validation accuracy curve
- Confusion matrix
- Sample image visualization

Chapter 6

Prediction Application

6.1 GUI Overview

The application provides a graphical user interface for real-time gender classification:

- Model selection and loading
- Image selection from file system
- Visual display of prediction results with confidence scores
- User feedback collection for prediction accuracy
- Statistics tracking for model performance

6.2 Prediction Process

```python
def predict_image(image_path, model, classes, device):
    try:
        # Load and preprocess the image
        image = Image.open(image_path).convert('RGB')
        img_tensor = pred_transform(image).unsqueeze(0).to(device)

        # Get model prediction
        with torch.no_grad():
            outputs = model(img_tensor)
            probabilities = F.softmax(outputs, dim=1)[0]

            # Get the top prediction and all class probabilities
            confidence_scores = {classes[i]: float(probabilities[i]) * 100
                                 for i in range(len(classes))}
            sorted_scores = sorted(confidence_scores.items(),
                                   key=lambda x: x[1], reverse=True)
            top_pred_class = sorted_scores[0][0]

            return image, top_pred_class, confidence_scores

    except Exception as e:
        print(f"Error during prediction: {e}")
```

```
        return None, None, None
```

Listing 6.1: Image prediction process

6.3 User Feedback Collection

The application collects and tracks user feedback on prediction accuracy:

- Records correct and incorrect predictions
- Calculates overall accuracy
- Tracks class-wise performance
- Provides summary statistics on application close

Chapter 7

Implementation Details

7.1 Project Structure

```
project_root/
|__ src/
|    |__ train.py              # Training script
|    |__ predict_gui.py        # GUI application
|    |__ models/               # Saved model directory
|         |__ best_cnn_gender.pth
|__ real_dataset/
     |__ train/
     |    |__ humans/
     |         |__ men/
     |         |__ women/
     |__ test/
          |__ humans/
               |__ men/
               |__ women/
```

7.2 Dependencies Management

Key dependencies include:

```
torch==1.9.0
torchvision==0.10.0
matplotlib==3.4.2
pillow==8.2.0
numpy==1.20.3
```

7.3 Error Handling

The implementation includes comprehensive error handling:

- Validation of dataset directory structure
- Model loading error recovery with alternative path checks

- Exception handling during image prediction
- User-friendly error messages in the GUI

Chapter 8

Conclusion

8.1 Summary

This project demonstrates a complete pipeline for gender classification using deep learning:

- Custom CNN architecture tailored for gender classification
- Robust training process with data augmentation and class balancing
- Comprehensive evaluation framework
- User-friendly GUI application for practical deployment

8.2 Future Improvements

Potential areas for future enhancement include:

- Transfer learning with pre-trained models (ResNet, VGG, etc.)
- Implementation of more advanced architectures
- Integration with real-time video for continuous classification
- Model quantization for mobile deployment
- Extension to multi-class classification for additional attributes

Appendix A

Code Listing

A.1 Training Module

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import WeightedRandomSampler
import matplotlib.pyplot as plt
import numpy as np
import os
import time

# Define the CNN Model for Gender Detection
class GenderCNN(nn.Module):
    def __init__(self, num_classes=2):
        super(GenderCNN, self).__init__()
        # First block
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)

        # Second block
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)

        # Third block
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)

        # Fourth block
        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(512)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.3)
        self.dropout2 = nn.Dropout(0.4)

        # Fully connected layers
        self.fc1 = nn.Linear(512 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)

```

```
    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = self.pool(torch.relu(self.bn4(self.conv4(x))))

        x = x.view(-1, 512 * 4 * 4)
        x = self.dropout1(torch.relu(self.fc1(x)))
        x = self.dropout2(torch.relu(self.fc2(x)))
        x = self.fc3(x)
        return x

# Define transformations for training and testing
train_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(20),
    transforms.RandomCrop(64, padding=8),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation
=0.2, hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

test_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Function to visualize a few sample images
def visualize_samples(dataset, classes, n=5):
    fig, axes = plt.subplots(len(classes), n, figsize=(15, 5*len(
classes)))
    for i, c in enumerate(classes):
        idx = dataset.class_to_idx[c]
        class_samples = [j for j, (_, label) in enumerate(dataset.
samples) if label == idx]

        for j in range(min(n, len(class_samples))):
            if j < len(class_samples):
                img, _ = dataset[class_samples[j]]
                img = img.numpy().transpose((1, 2, 0))
                img = img * 0.5 + 0.5  # Denormalize
                axes[i, j].imshow(img)
                axes[i, j].set_title(f"{c}")
                axes[i, j].axis("off")

    plt.tight_layout()
    plt.savefig(os.path.join(project_root, 'gender_samples.png'))
    plt.close()

# Function to train the model with validation
def train_model(model, criterion, optimizer, trainloader, testloader,
scheduler, epochs=30, early_stop_patience=5):
    best_acc = 0.0
    best_epoch = 0
```

```
    patience_counter = 0

    train_losses = []
    val_losses = []
    accuracies = []

    start_time = time.time()

    for epoch in range(epochs):
        # Training phase
        model.train()
        running_loss = 0.0

        for i, data in enumerate(trainloader, 0):
            inputs, labels = data[0].to(device), data[1].to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        epoch_loss = running_loss / len(trainloader)
        train_losses.append(epoch_loss)

        # Validation phase
        model.eval()
        correct = 0
        total = 0
        val_loss = 0.0

        with torch.no_grad():
            for data in testloader:
                images, labels = data[0].to(device), data[1].to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        epoch_val_loss = val_loss / len(testloader)
        val_losses.append(epoch_val_loss)

        accuracy = 100 * correct / total
        accuracies.append(accuracy)

        time_elapsed = time.time() - start_time
        print(f'Epoch {epoch+1}/{epochs} | Time: {time_elapsed:.1f}s | Train Loss: {epoch_loss:.3f} | Val Loss: {epoch_val_loss:.3f} | Accuracy: {accuracy:.2f}%')

        # Learning rate scheduler step
        scheduler.step(epoch_val_loss)

```

```python
        # Save best model
        if accuracy > best_acc:
            best_acc = accuracy
            best_epoch = epoch
            patience_counter = 0
            torch.save(model.state_dict(), os.path.join(models_dir, 'best_cnn_gender.pth'))
            print(f"    New best model saved (Accuracy: {best_acc:.2f}%)")
        else:
            patience_counter += 1
            if patience_counter >= early_stop_patience:
                print(f"Early stopping at epoch {epoch+1}. Best accuracy: {best_acc:.2f}% at epoch {best_epoch+1}")
                break

    # Final model save
    torch.save(model.state_dict(), os.path.join(models_dir, 'final_cnn_gender.pth'))
    print(f"    Final model saved")
    print(f"Best accuracy: {best_acc:.2f}% at epoch {best_epoch+1}")

    # Plot the training history
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')

    plt.subplot(1, 2, 2)
    plt.plot(accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Validation Accuracy')

    plt.tight_layout()
    plt.savefig(os.path.join(project_root, 'gender_training_history.png'))
    plt.close()

    return model, best_acc

# Function to evaluate and visualize model performance
def evaluate_model(model, testloader, classes):
    model.eval()

    # Collect predictions and ground truth
    class_correct = [0] * len(classes)
    class_total = [0] * len(classes)

    confusion_matrix = np.zeros((len(classes), len(classes)), dtype=int)

    with torch.no_grad():
```

```
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)

            for i in range(len(labels)):
                label = labels[i]
                pred = predicted[i]
                confusion_matrix[label][pred] += 1
                if label == pred:
                    class_correct[label] += 1
                class_total[label] += 1

    # Print class accuracies
    print("\nClass-wise Accuracy:")
    for i in range(len(classes)):
        accuracy = 100 * class_correct[i] / class_total[i] if class_total[i] > 0 else 0
        print(f'- {classes[i]}: {accuracy:.2f}% ({class_correct[i]}/{class_total[i]})')

    # Calculate overall accuracy
    overall_accuracy = 100 * sum(class_correct) / sum(class_total)
    print(f"\nOverall Accuracy: {overall_accuracy:.2f}%")

    # Visualize confusion matrix
    plt.figure(figsize=(8, 6))
    plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Confusion Matrix')
    plt.colorbar()

    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.tight_layout()

    plt.savefig(os.path.join(project_root, 'gender_confusion_matrix.png'))
    plt.close()

    return overall_accuracy, class_correct, class_total

# Main execution
if __name__ == "__main__":
    # Setup paths
    project_root = os.path.dirname(os.path.abspath(__file__))

    train_path = os.path.join(project_root, '..', 'real_dataset', 'train', 'humans')
    test_path = os.path.join(project_root, '..', 'real_dataset', 'test', 'humans')

    models_dir = os.path.join(project_root, 'models')
    os.makedirs(models_dir, exist_ok=True)
```

```python

    # Gender detection classes
    classes = ['men', 'women']

    # Check if dataset directories exist
    if not os.path.exists(train_path):
        print(f"    Training directory not found: {train_path}")
        print("Please create the following directory structure before training:")
        print(f"  {os.path.join(train_path, 'men')}")
        print(f"  {os.path.join(train_path, 'women')}")
        print(f"  {os.path.join(test_path, 'men')}")
        print(f"  {os.path.join(test_path, 'women')}")
        exit(1)

    # Setup device
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    # Load datasets
    print(f"Loading datasets from {train_path} and {test_path}...")
    try:
        trainset = datasets.ImageFolder(root=train_path, transform=train_transform)
        testset = datasets.ImageFolder(root=test_path, transform=test_transform)

        # Calculate class weights to handle imbalanced data
        class_counts = [0] * len(trainset.classes)
        for _, label in trainset.samples:
            class_counts[label] += 1

        print(f"Class distribution: {trainset.classes}")
        print(f"Class counts: {class_counts}")

        # Create weighted sampler for imbalanced classes
        class_weights = [1.0 / count for count in class_counts]
        sample_weights = [class_weights[label] for _, label in trainset.samples]
        sampler = WeightedRandomSampler(weights=sample_weights, num_samples=len(sample_weights), replacement=True)

        # Create data loaders
        trainloader = torch.utils.data.DataLoader(
            trainset, batch_size=32, sampler=sampler, num_workers=2
        )
        testloader = torch.utils.data.DataLoader(
            testset, batch_size=32, shuffle=False, num_workers=2
        )
    except Exception as e:
        print(f"    Error loading datasets: {e}")
        exit(1)

    # Visualize sample images
    visualize_samples(trainset, classes)

    # Create the model, loss function, optimizer, and scheduler
```

```
    model = GenderCNN(len(classes)).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.5, min_lr=1e-6)

    print(f"      Starting training for {len(classes)} classes: {classes}")

    # Train the model
    model, best_acc = train_model(
        model=model,
        criterion=criterion,
        optimizer=optimizer,
        trainloader=trainloader,
        testloader=testloader,
        scheduler=scheduler,
        epochs=30,
        early_stop_patience=5
    )

    # Load the best model for evaluation
    model.load_state_dict(torch.load(os.path.join(models_dir, 'best_cnn_gender.pth')))

    # Evaluate the model
    print("\n     Evaluating model performance...")
    accuracy, class_correct, class_total = evaluate_model(model, testloader, classes)

    print("\n   Training and evaluation complete.")
    print(f"      Best accuracy: {best_acc:.2f}%")
```

Listing A.1: Complete training module

A.2 Prediction Application

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
from torch.nn import functional as F
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import tkinter as tk
from tkinter import filedialog, messagebox, simpledialog, ttk
from PIL import Image
import os
import sys
import numpy as np
import argparse
import time

# Define the CNN model architecture (must match the training architecture)
class ImprovedCNN(nn.Module):
```

```
    def __init__(self, num_classes):
        super(ImprovedCNN, self).__init__()
        # First block
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)

        # Second block
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)

        # Third block
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)

        # Fourth block
        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(512)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.3)
        self.dropout2 = nn.Dropout(0.4)

        # Fully connected layers
        self.fc1 = nn.Linear(512 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = self.pool(torch.relu(self.bn4(self.conv4(x))))

        x = x.view(-1, 512 * 4 * 4)
        x = self.dropout1(torch.relu(self.fc1(x)))
        x = self.dropout2(torch.relu(self.fc2(x)))
        x = self.fc3(x)
        return x

# Define the transform for prediction (must match the test transform in training)
pred_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Global variables to track statistics
prediction_stats = {
    'total': 0,
    'correct': 0,
    'incorrect': 0,
    'class_predictions': {},
    'class_correct': {}
}

# Dictionary of available models and their classes
MODEL_CONFIGS = {
```

```
    'animals': {
        'model_path': 'models/best_cnn_animals.pth',
        'classes': None  # Will be determined dynamically from test directory
    },
    'gender': {
        'model_path': 'models/best_cnn_gender.pth',
        'classes': ['men', 'women']
    }
}

# Function to load the model
def load_model(model_type, model_path, test_dir=None):
    """Load a trained model from disk"""
    # Get project root directory
    project_root = os.path.dirname(os.path.abspath(__file__))
    parent_dir = os.path.dirname(project_root)  # Go up one level to the main project directory

    print(f"Project root: {project_root}")
    print(f"Parent directory: {parent_dir}")

    # Get model configuration
    if model_type not in MODEL_CONFIGS:
        print(f"Error: Unknown model type: {model_type}")
        print(f"Available model types: {list(MODEL_CONFIGS.keys())}")
        return None, None, None

    config = MODEL_CONFIGS[model_type]

    # Use provided model path or adjust default path
    if not model_path:
        model_path = os.path.join(project_root, config['model_path'])

    # Check if model file exists
    if not os.path.exists(model_path):
        print(f"Error: Model file not found: {model_path}")
        # Try to find model in the src/models directory instead
        alt_model_path = os.path.join(project_root, "models", os.path.basename(model_path))
        if os.path.exists(alt_model_path):
            print(f"    Found model at alternative location: {alt_model_path}")
            model_path = alt_model_path
        else:
            print("Error: Could not find model file in alternative locations")
            return None, None, None

    # Determine classes based on model type
    classes = config['classes']
    if classes is None:
        # For models like 'animals' where classes should be determined from the test directory
        try:
            # Look in the parent directory instead
            test_dir = os.path.join(parent_dir, "real_dataset", "test", "animals")
```

```python
            print(f"Looking for classes in: {test_dir}")

            if not os.path.exists(test_dir):
                print(f"Error: Directory not found: {test_dir}")
                # Try alternative path
                test_dir = os.path.join(parent_dir, "real_dataset", "train", "animals")
                print(f"Trying alternative path: {test_dir}")

                if not os.path.exists(test_dir):
                    print(f"Error: Alternative directory not found: {test_dir}")
                    return None, None, None

            classes = [d for d in os.listdir(test_dir) if os.path.isdir(os.path.join(test_dir, d))]
            classes.sort()  # Ensure consistent order
            print(f"Found classes: {classes}")
        except Exception as e:
            print(f"Error: Error determining classes from directory: {e}")
            return None, None, None

    # Ensure we have valid classes
    if not classes:
        print(f"Error: Could not determine classes for model type: {model_type}")
        return None, None, None

    print(f"Model type: {model_type}")
    print(f"Classes: {classes}")
    print(f"Using model file: {model_path}")

    # Create and load model
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    model = ImprovedCNN(len(classes)).to(device)

    try:
        model.load_state_dict(torch.load(model_path, map_location=device))
        model.eval()
        print("    Model loaded successfully")

        # Initialize statistics counters for each class
        for cls in classes:
            prediction_stats['class_predictions'][cls] = 0
            prediction_stats['class_correct'][cls] = 0

        return model, classes, device
    except Exception as e:
        print(f"Error: Error loading model: {e}")
        return None, None, None

# Function to make a prediction
def predict_image(image_path, model, classes, device):
```

```python
    try:
        # Load and preprocess the image
        image = Image.open(image_path).convert('RGB')
        img_tensor = pred_transform(image).unsqueeze(0).to(device)

        # Get model prediction
        with torch.no_grad():
            outputs = model(img_tensor)
            probabilities = F.softmax(outputs, dim=1)[0]

            # Get the top prediction and all class probabilities
            confidence_scores = {classes[i]: float(probabilities[i]) * 100 for i in range(len(classes))}
            sorted_scores = sorted(confidence_scores.items(), key=lambda x: x[1], reverse=True)
            top_pred_class = sorted_scores[0][0]

            return image, top_pred_class, confidence_scores

    except Exception as e:
        print(f"Error: Error during prediction: {e}")
        return None, None, None

# Class for the prediction application GUI
class PredictionApp:
    def __init__(self, root, available_models):
        self.root = root
        self.available_models = available_models
        self.model = None
        self.classes = None
        self.device = None
        self.current_model_type = None
        self.current_image_path = None

        # Set window properties
        self.root.title("Image Classification")
        self.root.geometry("1000x850")  # Increased height to accommodate model selector
        self.root.configure(bg="#f0f0f0")

        # Create main frame
        self.main_frame = tk.Frame(root, bg="#f0f0f0")
        self.main_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)

        # Create header
        self.header_label = tk.Label(
            self.main_frame,
            text="Image Classification",
            font=("Arial", 24, "bold"),
            bg="#f0f0f0"
        )
        self.header_label.pack(pady=(0, 10))

        # Create model selection frame
        self.model_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
        self.model_frame.pack(fill=tk.X, pady=10)

```

```
        # Add model selection label
        tk.Label(
            self.model_frame,
            text="Select Model:",
            font=("Arial", 14),
            bg="#f0f0f0"
        ).pack(side=tk.LEFT, padx=(0, 10))

        # Add model selection dropdown
        self.model_var = tk.StringVar(value=list(self.available_models.keys())[0])
        self.model_dropdown = ttk.Combobox(
            self.model_frame,
            textvariable=self.model_var,
            values=list(self.available_models.keys()),
            font=("Arial", 12),
            state="readonly",
            width=15
        )
        self.model_dropdown.pack(side=tk.LEFT, padx=(0, 10))

        # Add model load button
        self.load_model_button = tk.Button(
            self.model_frame,
            text="Load Model",
            font=("Arial", 12),
            command=self.load_selected_model,
            bg="#4285F4",
            fg="white",
            padx=10,
            pady=5
        )
        self.load_model_button.pack(side=tk.LEFT)

        # Create button frame
        self.button_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
        self.button_frame.pack(fill=tk.X, pady=10)

        # Add select image button (initially disabled)
        self.select_button = tk.Button(
            self.button_frame,
            text="Select Image",
            font=("Arial", 14),
            command=self.select_image,
            bg="#4CAF50",
            fg="white",
            padx=20,
            pady=10,
            relief=tk.RAISED,
            borderwidth=2,
            state=tk.DISABLED
        )
        self.select_button.pack(side=tk.LEFT, padx=(0, 10))

        # Add quit button
        self.quit_button = tk.Button(
            self.button_frame,
            text="Quit",
```

```
            font=("Arial", 14),
            command=self.quit_application,
            bg="#F44336",
            fg="white",
            padx=20,
            pady=10,
            relief=tk.RAISED,
            borderwidth=2
        )
        self.quit_button.pack(side=tk.RIGHT)

        # Create content frame
        self.content_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
        self.content_frame.pack(fill=tk.BOTH, expand=True, pady=10)

        # Create a frame for the figure
        self.figure_frame = tk.Frame(self.content_frame, bg="#f0f0f0")
        self.figure_frame.pack(fill=tk.BOTH, expand=True)

        # Create matplotlib figure for the image and predictions
        self.fig = plt.figure(figsize=(10, 6))
        self.canvas = FigureCanvasTkAgg(self.fig, self.figure_frame)
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        # Create feedback frame with a title
        self.feedback_title = tk.Label(
            self.main_frame,
            text="Was the prediction correct?",
            font=("Arial", 16, "bold"),
            bg="#f0f0f0"
        )
        self.feedback_title.pack(pady=(15, 5))

        # Create feedback frame
        self.feedback_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
        self.feedback_frame.pack(fill=tk.X, pady=10)

        # Add correct button
        self.correct_button = tk.Button(
            self.feedback_frame,
            text="Correct    ",
            font=("Arial", 14),
            command=lambda: self.record_feedback(True),
            bg="#4CAF50",
            fg="white",
            state=tk.DISABLED,
            padx=25,
            pady=10
        )
        self.correct_button.pack(side=tk.LEFT, padx=(0, 10))

        # Add incorrect button
        self.incorrect_button = tk.Button(
            self.feedback_frame,
            text="Incorrect    ",
            font=("Arial", 14),
            command=lambda: self.record_feedback(False),
            bg="#F44336",
```

```
            fg="white",
            state=tk.DISABLED,
            padx=25,
            pady=10
        )
        self.incorrect_button.pack(side=tk.LEFT)

        # Stats label
        self.stats_label = tk.Label(
            self.main_frame,
            text="Total: 0 | Correct: 0 | Incorrect: 0 | Accuracy: 0.00%",
            font=("Arial", 14, "bold"),
            bg="#f0f0f0"
        )
        self.stats_label.pack(pady=10)

        # Status message
        self.status_label = tk.Label(
            self.main_frame,
            text="Please load a model to start",
            font=("Arial", 12, "italic"),
            fg="#555555",
            bg="#f0f0f0"
        )
        self.status_label.pack(pady=(0, 10))

        # Set up class variables
        self.current_prediction = None

    def load_selected_model(self):
        """Load the model selected from the dropdown"""
        model_type = self.model_var.get()

        # Update status
        self.status_label.config(text=f"Loading {model_type} model...")
        self.root.update()

        # Reset prediction stats for new model
        global prediction_stats
        prediction_stats = {
            'total': 0,
            'correct': 0,
            'incorrect': 0,
            'class_predictions': {},
            'class_correct': {}
        }

        # Load model
        project_root = os.path.dirname(os.path.abspath(__file__))
        parent_dir = os.path.dirname(project_root)

        config = MODEL_CONFIGS[model_type]
        model_path = os.path.join(project_root, config['model_path'])

        # Determine test directory based on directory structure
        test_dir = None
        if model_type == 'animals':
```

```
            # Try to find the test directory in the parent directory (main project directory)
            test_dir = os.path.join(parent_dir, 'real_dataset', 'test', 'animals')
            if not os.path.exists(test_dir):
                test_dir = os.path.join(parent_dir, 'real_dataset', 'train', 'animals')

        self.model, self.classes, self.device = load_model(model_type, model_path, test_dir)

        if self.model:
            self.current_model_type = model_type
            self.status_label.config(text=f"{model_type.capitalize()} model loaded successfully. Please select an image.")
            self.select_button.config(state=tk.NORMAL)
            self.stats_label.config(text="Total: 0 | Correct: 0 | Incorrect: 0 | Accuracy: 0.00%")
        else:
            self.status_label.config(text=f"Error loading {model_type} model. Please check the model file or directory structure.")
            self.select_button.config(state=tk.DISABLED)

    def select_image(self):
        """Open a file dialog to select an image"""
        filetypes = [
            ("Image files", "*.jpg *.jpeg *.png *.bmp *.gif"),
            ("All files", "*.*")
        ]

        filepath = filedialog.askopenfilename(
            title="Select Image",
            filetypes=filetypes
        )

        if filepath:
            self.current_image_path = filepath
            self.predict_and_display(filepath)

    def predict_and_display(self, image_path):
        """Run prediction and display results"""
        # Update status
        self.status_label.config(text="Analyzing image...")
        self.root.update()

        # Run prediction
        image, prediction, confidence_scores = predict_image(
            image_path, self.model, self.classes, self.device
        )

        if image and prediction and confidence_scores:
            # Store current prediction
            self.current_prediction = prediction

            # Clear previous figure
            self.fig.clear()

            # Create two subplots - one for image, one for bar chart
```

```
            ax1 = self.fig.add_subplot(1, 2, 1)
            ax2 = self.fig.add_subplot(1, 2, 2)

            # Display image
            ax1.imshow(image)
            ax1.set_title(f"Prediction: {prediction}")
            ax1.axis('off')

            # Create bar chart of confidence scores
            sorted_scores = sorted(confidence_scores.items(), key=lambda x: x[1], reverse=True)
            classes = [item[0] for item in sorted_scores]
            scores = [item[1] for item in sorted_scores]

            bars = ax2.bar(classes, scores, color=['#4285F4' if cls == prediction else '#A0A0A0' for cls in classes])
            ax2.set_ylabel('Confidence (%)')
            ax2.set_title('Class Predictions')
            ax2.set_ylim([0, 100])

            # Add percentage labels above bars
            for bar in bars:
                height = bar.get_height()
                ax2.annotate(f'{height:.1f}%',
                            xy=(bar.get_x() + bar.get_width() / 2, height),
                            xytext=(0, 3),  # 3 points vertical offset
                            textcoords="offset points",
                            ha='center', va='bottom',
                            fontsize=9)

            # Rotate x-axis labels for better readability if needed
            if len(classes) > 3:
                plt.setp(ax2.get_xticklabels(), rotation=45, ha='right')

            # Update the canvas
            self.fig.tight_layout()
            self.canvas.draw()

            # Enable feedback buttons
            self.correct_button.config(state=tk.NORMAL)
            self.incorrect_button.config(state=tk.NORMAL)

            # Update status
            self.status_label.config(text=f"Prediction complete. Model says: {prediction}")

            # Update statistics display
            global prediction_stats
            prediction_stats['total'] += 1
            prediction_stats['class_predictions'][prediction] += 1
            self.update_stats_display()
        else:
            # Handle prediction failure
            self.status_label.config(text="Error analyzing image. Please try another image.")
            messagebox.showerror("Prediction Error", "Could not analyze
```

```
 the selected image.")

    def record_feedback(self, is_correct):
        """Record user feedback on prediction accuracy"""
        if self.current_prediction:
            global prediction_stats

            if is_correct:
                prediction_stats['correct'] += 1
                prediction_stats['class_correct'][self.current_prediction] += 1
                feedback_msg = "    Feedback recorded: Prediction was correct!"
            else:
                prediction_stats['incorrect'] += 1
                feedback_msg = "    Feedback recorded: Prediction was incorrect."

                # Optionally ask for correct class if prediction was wrong
                if len(self.classes) > 2:  # Only for multi-class problems
                    correct_class = simpledialog.askstring(
                        "Correct Class",
                        f"What was the correct class?\nOptions: {', '.join(self.classes)}",
                        parent=self.root
                    )

                    if correct_class and correct_class in self.classes:
                        feedback_msg += f" (Correct class: {correct_class})"

            # Update status and stats display
            self.status_label.config(text=feedback_msg)
            self.update_stats_display()

            # Reset buttons for next prediction
            self.correct_button.config(state=tk.DISABLED)
            self.incorrect_button.config(state=tk.DISABLED)

    def update_stats_display(self):
        """Update the statistics display label"""
        global prediction_stats

        total = prediction_stats['total']
        correct = prediction_stats['correct']
        incorrect = prediction_stats['incorrect']

        if total > 0:
            accuracy = (correct / total) * 100
            stats_text = f"Total: {total} | Correct: {correct} | Incorrect: {incorrect} | Accuracy: {accuracy:.2f}%"
        else:
            stats_text = "Total: 0 | Correct: 0 | Incorrect: 0 | Accuracy: 0.00%"

        self.stats_label.config(text=stats_text)
```

```python

    def quit_application(self):
        """Exit the application and show final statistics"""
        global prediction_stats

        # Create final statistics message
        total = prediction_stats['total']

        if total > 0:
            accuracy = (prediction_stats['correct'] / total) * 100
            message = f"Session Statistics:\n\n"
            message += f"Total predictions: {total}\n"
            message += f"Correct: {prediction_stats['correct']} ({(prediction_stats['correct']/total)*100:.2f}%)\n"
            message += f"Incorrect: {prediction_stats['incorrect']} ({(prediction_stats['incorrect']/total)*100:.2f}%)\n\n"

            # Add per-class statistics
            message += "Class Performance:\n"
            for cls in self.classes:
                predictions = prediction_stats['class_predictions'].get(cls, 0)
                correct = prediction_stats['class_correct'].get(cls, 0)

                if predictions > 0:
                    class_accuracy = (correct / predictions) * 100
                    message += f"{cls}: {correct}/{predictions} correct ({class_accuracy:.2f}%)\n"
                else:
                    message += f"{cls}: No predictions\n"

            messagebox.showinfo("Session Statistics", message)

        self.root.destroy()

# Run the application
if __name__ == "__main__":
    root = tk.Tk()
    app = PredictionApp(root, MODEL_CONFIGS)
    root.mainloop()
```

Listing A.2: Complete prediction application