

Deep Learning-Based Gender Classification System Documentation

Computer Vision & Deep Learning Project

April 25, 2025

Contents

1	Introduction	3
1.1	Project Overview	3
1.2	Objectives	3
1.3	System Requirements	3
2	Dataset	4
2.1	Dataset Structure	4
2.2	Data Preprocessing	4
2.3	Data Augmentation	4
3	Model Architecture	6
3.1	Convolutional Neural Network	6
3.2	Architecture Components	7
3.3	Model Parameters	7
4	Training Process	8
4.1	Training Configuration	8
4.2	Learning Rate Schedule	8
4.3	Class Balancing	8
4.4	Early Stopping	9
5	Evaluation Metrics	10
5.1	Model Evaluation	10
5.2	Visualization	10
6	Prediction Application	12
6.1	GUI Overview	12
6.2	Prediction Process	12
6.3	User Feedback Collection	13
7	Implementation Details	14
7.1	Project Structure	14
7.2	Dependencies Management	14
7.3	Error Handling	14
8	Conclusion	16
8.1	Summary	16
8.2	Future Improvements	16

A	Code Listing	17
A.1	Training Module	17
A.2	Prediction Application	23

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:

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

```
8     transforms.ToTensor(),  
9     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
10 ])
```

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.

```
1 class GenderCNN(nn.Module):
2     def __init__(self, num_classes=2):
3         super(GenderCNN, self).__init__()
4         # First block
5         self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
6         self.bn1 = nn.BatchNorm2d(64)
7
8         # Second block
9         self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
10        self.bn2 = nn.BatchNorm2d(128)
11
12        # Third block
13        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
14        self.bn3 = nn.BatchNorm2d(256)
15
16        # Fourth block
17        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
18        self.bn4 = nn.BatchNorm2d(512)
19
20        self.pool = nn.MaxPool2d(2, 2)
21        self.dropout1 = nn.Dropout(0.3)
22        self.dropout2 = nn.Dropout(0.4)
23
24        # Fully connected layers
25        self.fc1 = nn.Linear(512 * 4 * 4, 512)
26        self.fc2 = nn.Linear(512, 256)
27        self.fc3 = nn.Linear(256, num_classes)
28
29    def forward(self, x):
30        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
31        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
32        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
33        x = self.pool(torch.relu(self.bn4(self.conv4(x))))
34
35        x = x.view(-1, 512 * 4 * 4)
36        x = self.dropout1(torch.relu(self.fc1(x)))
37        x = self.dropout2(torch.relu(self.fc2(x)))
```

```

38     x = self.fc3(x)
39     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:

```
1 # Calculate class weights to handle imbalanced data
2 class_counts = [0] * len(trainset.classes)
3 for _, label in trainset.samples:
4     class_counts[label] += 1
5
6 # Create weighted sampler for imbalanced classes
```

```

7 class_weights = [1.0 / count for count in class_counts]
8 sample_weights = [class_weights[label] for _, label in trainset.samples
9 ]
9 sampler = WeightedRandomSampler(
10     weights=sample_weights,
11     num_samples=len(sample_weights),
12     replacement=True
13 )

```

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

```
1 def evaluate_model(model, testloader, classes):
2     model.eval()
3
4     # Collect predictions and ground truth
5     class_correct = [0] * len(classes)
6     class_total = [0] * len(classes)
7
8     confusion_matrix = np.zeros((len(classes), len(classes)), dtype=int
9 )
10
11     with torch.no_grad():
12         for data in testloader:
13             images, labels = data[0].to(device), data[1].to(device)
14             outputs = model(images)
15             _, predicted = torch.max(outputs, 1)
16
17             for i in range(len(labels)):
18                 label = labels[i]
19                 pred = predicted[i]
20                 confusion_matrix[label][pred] += 1
21                 if label == pred:
22                     class_correct[label] += 1
23                     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

```
1 def predict_image(image_path, model, classes, device):
2     try:
3         # Load and preprocess the image
4         image = Image.open(image_path).convert('RGB')
5         img_tensor = pred_transform(image).unsqueeze(0).to(device)
6
7         # Get model prediction
8         with torch.no_grad():
9             outputs = model(img_tensor)
10            probabilities = F.softmax(outputs, dim=1)[0]
11
12            # Get the top prediction and all class probabilities
13            confidence_scores = {classes[i]: float(probabilities[i]) *
100
14                                for i in range(len(classes))}
15            sorted_scores = sorted(confidence_scores.items(),
16                                  key=lambda x: x[1], reverse=True)
17            top_pred_class = sorted_scores[0][0]
18
19            return image, top_pred_class, confidence_scores
20
21     except Exception as e:
22         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

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import torchvision.transforms as transforms
5 import torchvision.datasets as datasets
6 from torch.utils.data import WeightedRandomSampler
7 import matplotlib.pyplot as plt
8 import numpy as np
9 import os
10 import time
11
12 # Define the CNN Model for Gender Detection
13 class GenderCNN(nn.Module):
14     def __init__(self, num_classes=2):
15         super(GenderCNN, self).__init__()
16         # First block
17         self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
18         self.bn1 = nn.BatchNorm2d(64)
19
20         # Second block
21         self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
22         self.bn2 = nn.BatchNorm2d(128)
23
24         # Third block
25         self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
26         self.bn3 = nn.BatchNorm2d(256)
27
28         # Fourth block
29         self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
30         self.bn4 = nn.BatchNorm2d(512)
31
32         self.pool = nn.MaxPool2d(2, 2)
33         self.dropout1 = nn.Dropout(0.3)
34         self.dropout2 = nn.Dropout(0.4)
35
36         # Fully connected layers
37         self.fc1 = nn.Linear(512 * 4 * 4, 512)
38         self.fc2 = nn.Linear(512, 256)
39         self.fc3 = nn.Linear(256, num_classes)
40
```

```

41     def forward(self, x):
42         x = self.pool(torch.relu(self.bn1(self.conv1(x))))
43         x = self.pool(torch.relu(self.bn2(self.conv2(x))))
44         x = self.pool(torch.relu(self.bn3(self.conv3(x))))
45         x = self.pool(torch.relu(self.bn4(self.conv4(x))))
46
47         x = x.view(-1, 512 * 4 * 4)
48         x = self.dropout1(torch.relu(self.fc1(x)))
49         x = self.dropout2(torch.relu(self.fc2(x)))
50         x = self.fc3(x)
51         return x
52
53 # Define transformations for training and testing
54 train_transform = transforms.Compose([
55     transforms.Resize((64, 64)),
56     transforms.RandomHorizontalFlip(p=0.5),
57     transforms.RandomRotation(20),
58     transforms.RandomCrop(64, padding=8),
59     transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation
60                             =0.2, hue=0.1),
61     transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
62     transforms.ToTensor(),
63     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
64 ])
65
66 test_transform = transforms.Compose([
67     transforms.Resize((64, 64)),
68     transforms.ToTensor(),
69     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
70 ])
71
72 # Function to visualize a few sample images
73 def visualize_samples(dataset, classes, n=5):
74     fig, axes = plt.subplots(len(classes), n, figsize=(15, 5*len(
75         classes)))
76     for i, c in enumerate(classes):
77         idx = dataset.class_to_idx[c]
78         class_samples = [j for j, (_, label) in enumerate(dataset.
79             samples) if label == idx]
80
81         for j in range(min(n, len(class_samples))):
82             if j < len(class_samples):
83                 img, _ = dataset[class_samples[j]]
84                 img = img.numpy().transpose((1, 2, 0))
85                 img = img * 0.5 + 0.5 # Denormalize
86                 axes[i, j].imshow(img)
87                 axes[i, j].set_title(f"{c}")
88                 axes[i, j].axis("off")
89
90     plt.tight_layout()
91     plt.savefig(os.path.join(project_root, 'gender_samples.png'))
92     plt.close()
93
94 # Function to train the model with validation
95 def train_model(model, criterion, optimizer, trainloader, testloader,
96                 scheduler, epochs=30, early_stop_patience=5):
97     best_acc = 0.0
98     best_epoch = 0

```

```

95     patience_counter = 0
96
97     train_losses = []
98     val_losses = []
99     accuracies = []
100
101     start_time = time.time()
102
103     for epoch in range(epochs):
104         # Training phase
105         model.train()
106         running_loss = 0.0
107
108         for i, data in enumerate(trainloader, 0):
109             inputs, labels = data[0].to(device), data[1].to(device)
110
111             optimizer.zero_grad()
112             outputs = model(inputs)
113             loss = criterion(outputs, labels)
114             loss.backward()
115             optimizer.step()
116
117             running_loss += loss.item()
118
119         epoch_loss = running_loss / len(trainloader)
120         train_losses.append(epoch_loss)
121
122         # Validation phase
123         model.eval()
124         correct = 0
125         total = 0
126         val_loss = 0.0
127
128         with torch.no_grad():
129             for data in testloader:
130                 images, labels = data[0].to(device), data[1].to(device)
131                 outputs = model(images)
132                 loss = criterion(outputs, labels)
133                 val_loss += loss.item()
134
135                 _, predicted = torch.max(outputs.data, 1)
136                 total += labels.size(0)
137                 correct += (predicted == labels).sum().item()
138
139         epoch_val_loss = val_loss / len(testloader)
140         val_losses.append(epoch_val_loss)
141
142         accuracy = 100 * correct / total
143         accuracies.append(accuracy)
144
145         time_elapsed = time.time() - start_time
146         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}%')
147
148         # Learning rate scheduler step
149         scheduler.step(epoch_val_loss)
150

```

```

151     # Save best model
152     if accuracy > best_acc:
153         best_acc = accuracy
154         best_epoch = epoch
155         patience_counter = 0
156         torch.save(model.state_dict(), os.path.join(models_dir, '
best_cnn_gender.pth'))
157         print(f"    New best model saved (Accuracy: {best_acc:.2f
}%)"")
158     else:
159         patience_counter += 1
160         if patience_counter >= early_stop_patience:
161             print(f"Early stopping at epoch {epoch+1}. Best
accuracy: {best_acc:.2f}% at epoch {best_epoch+1}")
162             break
163
164     # Final model save
165     torch.save(model.state_dict(), os.path.join(models_dir, '
final_cnn_gender.pth'))
166     print(f"    Final model saved")
167     print(f"Best accuracy: {best_acc:.2f}% at epoch {best_epoch+1}")
168
169     # Plot the training history
170     plt.figure(figsize=(12, 4))
171
172     plt.subplot(1, 2, 1)
173     plt.plot(train_losses, label='Training Loss')
174     plt.plot(val_losses, label='Validation Loss')
175     plt.xlabel('Epochs')
176     plt.ylabel('Loss')
177     plt.legend()
178     plt.title('Training and Validation Loss')
179
180     plt.subplot(1, 2, 2)
181     plt.plot(accuracies, label='Validation Accuracy')
182     plt.xlabel('Epochs')
183     plt.ylabel('Accuracy (%)')
184     plt.title('Validation Accuracy')
185
186     plt.tight_layout()
187     plt.savefig(os.path.join(project_root, 'gender_training_history.png
'))
188     plt.close()
189
190     return model, best_acc
191
192 # Function to evaluate and visualize model performance
193 def evaluate_model(model, testloader, classes):
194     model.eval()
195
196     # Collect predictions and ground truth
197     class_correct = [0] * len(classes)
198     class_total = [0] * len(classes)
199
200     confusion_matrix = np.zeros((len(classes), len(classes)), dtype=int
)
201
202     with torch.no_grad():

```

```

203     for data in testloader:
204         images, labels = data[0].to(device), data[1].to(device)
205         outputs = model(images)
206         _, predicted = torch.max(outputs, 1)
207
208         for i in range(len(labels)):
209             label = labels[i]
210             pred = predicted[i]
211             confusion_matrix[label][pred] += 1
212             if label == pred:
213                 class_correct[label] += 1
214             class_total[label] += 1
215
216     # Print class accuracies
217     print("\nClass-wise Accuracy:")
218     for i in range(len(classes)):
219         accuracy = 100 * class_correct[i] / class_total[i] if
class_total[i] > 0 else 0
220         print(f'- {classes[i]}: {accuracy:.2f}% ({class_correct[i]}/{
class_total[i]})')
221
222     # Calculate overall accuracy
223     overall_accuracy = 100 * sum(class_correct) / sum(class_total)
224     print(f"\nOverall Accuracy: {overall_accuracy:.2f}%")
225
226     # Visualize confusion matrix
227     plt.figure(figsize=(8, 6))
228     plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.
Blues)
229     plt.title('Confusion Matrix')
230     plt.colorbar()
231
232     tick_marks = np.arange(len(classes))
233     plt.xticks(tick_marks, classes, rotation=45)
234     plt.yticks(tick_marks, classes)
235
236     plt.xlabel('Predicted Label')
237     plt.ylabel('True Label')
238     plt.tight_layout()
239
240     plt.savefig(os.path.join(project_root, 'gender_confusion_matrix.png
'))
241     plt.close()
242
243     return overall_accuracy, class_correct, class_total
244
245 # Main execution
246 if __name__ == "__main__":
247     # Setup paths
248     project_root = os.path.dirname(os.path.abspath(__file__))
249
250     train_path = os.path.join(project_root, '..', 'real_dataset', '
train', 'humans')
251     test_path = os.path.join(project_root, '..', 'real_dataset', 'test'
, 'humans')
252
253     models_dir = os.path.join(project_root, 'models')
254     os.makedirs(models_dir, exist_ok=True)

```

```

255
256 # Gender detection classes
257 classes = ['men', 'women']
258
259 # Check if dataset directories exist
260 if not os.path.exists(train_path):
261     print(f"    Training directory not found: {train_path}")
262     print("Please create the following directory structure before
training:")
263     print(f"    {os.path.join(train_path, 'men')}")
264     print(f"    {os.path.join(train_path, 'women')}")
265     print(f"    {os.path.join(test_path, 'men')}")
266     print(f"    {os.path.join(test_path, 'women')}")
267     exit(1)
268
269 # Setup device
270 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
')
271 print(f"Using device: {device}")
272
273 # Load datasets
274 print(f"Loading datasets from {train_path} and {test_path}...")
275 try:
276     trainset = datasets.ImageFolder(root=train_path, transform=
train_transform)
277     testset = datasets.ImageFolder(root=test_path, transform=
test_transform)
278
279     # Calculate class weights to handle imbalanced data
280     class_counts = [0] * len(trainset.classes)
281     for _, label in trainset.samples:
282         class_counts[label] += 1
283
284     print(f"Class distribution: {trainset.classes}")
285     print(f"Class counts: {class_counts}")
286
287     # Create weighted sampler for imbalanced classes
288     class_weights = [1.0 / count for count in class_counts]
289     sample_weights = [class_weights[label] for _, label in trainset
.samples]
290     sampler = WeightedRandomSampler(weights=sample_weights,
num_samples=len(sample_weights), replacement=True)
291
292     # Create data loaders
293     trainloader = torch.utils.data.DataLoader(
294         trainset, batch_size=32, sampler=sampler, num_workers=2
295     )
296     testloader = torch.utils.data.DataLoader(
297         testset, batch_size=32, shuffle=False, num_workers=2
298     )
299 except Exception as e:
300     print(f"    Error loading datasets: {e}")
301     exit(1)
302
303 # Visualize sample images
304 visualize_samples(trainset, classes)
305
306 # Create the model, loss function, optimizer, and scheduler

```

```

307     model = GenderCNN(len(classes)).to(device)
308     criterion = nn.CrossEntropyLoss()
309     optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1
e-5)
310     scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
patience=3, factor=0.5, min_lr=1e-6)
311
312     print(f"          Starting training for {len(classes)} classes: {
classes}")
313
314     # Train the model
315     model, best_acc = train_model(
316         model=model,
317         criterion=criterion,
318         optimizer=optimizer,
319         trainloader=trainloader,
320         testloader=testloader,
321         scheduler=scheduler,
322         epochs=30,
323         early_stop_patience=5
324     )
325
326     # Load the best model for evaluation
327     model.load_state_dict(torch.load(os.path.join(models_dir, '
best_cnn_gender.pth')))
328
329     # Evaluate the model
330     print("\n      Evaluating model performance...")
331     accuracy, class_correct, class_total = evaluate_model(model,
testloader, classes)
332
333     print("\n      Training and evaluation complete.")
334     print(f"          Best accuracy: {best_acc:.2f}%")

```

Listing A.1: Complete training module

A.2 Prediction Application

```

1  import torch
2  import torch.nn as nn
3  import torchvision.transforms as transforms
4  from torch.nn import functional as F
5  import matplotlib.pyplot as plt
6  from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
7  import tkinter as tk
8  from tkinter import filedialog, messagebox, simpledialog, ttk
9  from PIL import Image
10 import os
11 import sys
12 import numpy as np
13 import argparse
14 import time
15
16 # Define the CNN model architecture (must match the training
architecture)
17 class ImprovedCNN(nn.Module):

```



```

18     def __init__(self, num_classes):
19         super(ImprovedCNN, self).__init__()
20         # First block
21         self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
22         self.bn1 = nn.BatchNorm2d(64)
23
24         # Second block
25         self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
26         self.bn2 = nn.BatchNorm2d(128)
27
28         # Third block
29         self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
30         self.bn3 = nn.BatchNorm2d(256)
31
32         # Fourth block
33         self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
34         self.bn4 = nn.BatchNorm2d(512)
35
36         self.pool = nn.MaxPool2d(2, 2)
37         self.dropout1 = nn.Dropout(0.3)
38         self.dropout2 = nn.Dropout(0.4)
39
40         # Fully connected layers
41         self.fc1 = nn.Linear(512 * 4 * 4, 512)
42         self.fc2 = nn.Linear(512, 256)
43         self.fc3 = nn.Linear(256, num_classes)
44
45     def forward(self, x):
46         x = self.pool(torch.relu(self.bn1(self.conv1(x))))
47         x = self.pool(torch.relu(self.bn2(self.conv2(x))))
48         x = self.pool(torch.relu(self.bn3(self.conv3(x))))
49         x = self.pool(torch.relu(self.bn4(self.conv4(x))))
50
51         x = x.view(-1, 512 * 4 * 4)
52         x = self.dropout1(torch.relu(self.fc1(x)))
53         x = self.dropout2(torch.relu(self.fc2(x)))
54         x = self.fc3(x)
55         return x
56
57     # Define the transform for prediction (must match the test transform in
58     # training)
59     pred_transform = transforms.Compose([
60         transforms.Resize((64, 64)),
61         transforms.ToTensor(),
62         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
63     ])
64
65     # Global variables to track statistics
66     prediction_stats = {
67         'total': 0,
68         'correct': 0,
69         'incorrect': 0,
70         'class_predictions': {},
71         'class_correct': {}
72     }
73
74     # Dictionary of available models and their classes
75     MODEL_CONFIGS = {

```

```

75     'animals': {
76         'model_path': 'models/best_cnn_animals.pth',
77         'classes': None # Will be determined dynamically from test
directory
78     },
79     'gender': {
80         'model_path': 'models/best_cnn_gender.pth',
81         'classes': ['men', 'women']
82     }
83 }
84
85 # Function to load the model
86 def load_model(model_type, model_path, test_dir=None):
87     """Load a trained model from disk"""
88     # Get project root directory
89     project_root = os.path.dirname(os.path.abspath(__file__))
90     parent_dir = os.path.dirname(project_root) # Go up one level to
the main project directory
91
92     print(f"Project root: {project_root}")
93     print(f"Parent directory: {parent_dir}")
94
95     # Get model configuration
96     if model_type not in MODEL_CONFIGS:
97         print(f"Error: Unknown model type: {model_type}")
98         print(f"Available model types: {list(MODEL_CONFIGS.keys())}")
99         return None, None, None
100
101     config = MODEL_CONFIGS[model_type]
102
103     # Use provided model path or adjust default path
104     if not model_path:
105         model_path = os.path.join(project_root, config['model_path'])
106
107     # Check if model file exists
108     if not os.path.exists(model_path):
109         print(f"Error: Model file not found: {model_path}")
110         # Try to find model in the src/models directory instead
111         alt_model_path = os.path.join(project_root, "models", os.path.
basename(model_path))
112         if os.path.exists(alt_model_path):
113             print(f"    Found model at alternative location: {
alt_model_path}")
114             model_path = alt_model_path
115         else:
116             print("Error: Could not find model file in alternative
locations")
117             return None, None, None
118
119     # Determine classes based on model type
120     classes = config['classes']
121     if classes is None:
122         # For models like 'animals' where classes should be determined
from the test directory
123         try:
124             # Look in the parent directory instead
125             test_dir = os.path.join(parent_dir, "real_dataset", "test",
"animals")

```

```

126         print(f"Looking for classes in: {test_dir}")
127
128         if not os.path.exists(test_dir):
129             print(f"Error: Directory not found: {test_dir}")
130             # Try alternative path
131             test_dir = os.path.join(parent_dir, "real_dataset", "
train", "animals")
132             print(f"Trying alternative path: {test_dir}")
133
134             if not os.path.exists(test_dir):
135                 print(f"Error: Alternative directory not found: {
test_dir}")
136                 return None, None, None
137
138             classes = [d for d in os.listdir(test_dir) if os.path.isdir
(os.path.join(test_dir, d))]
139             classes.sort() # Ensure consistent order
140             print(f"Found classes: {classes}")
141             except Exception as e:
142                 print(f"Error: Error determining classes from directory: {e
}")
143                 return None, None, None
144
145             # Ensure we have valid classes
146             if not classes:
147                 print(f"Error: Could not determine classes for model type: {
model_type}")
148                 return None, None, None
149
150             print(f"Model type: {model_type}")
151             print(f"Classes: {classes}")
152             print(f"Using model file: {model_path}")
153
154             # Create and load model
155             device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
')
156             print(f"Using device: {device}")
157
158             model = ImprovedCNN(len(classes)).to(device)
159
160             try:
161                 model.load_state_dict(torch.load(model_path, map_location=
device))
162                 model.eval()
163                 print("    Model loaded successfully")
164
165                 # Initialize statistics counters for each class
166                 for cls in classes:
167                     prediction_stats['class_predictions'][cls] = 0
168                     prediction_stats['class_correct'][cls] = 0
169
170                 return model, classes, device
171             except Exception as e:
172                 print(f"Error: Error loading model: {e}")
173                 return None, None, None
174
175 # Function to make a prediction
176 def predict_image(image_path, model, classes, device):

```

```

177     try:
178         # Load and preprocess the image
179         image = Image.open(image_path).convert('RGB')
180         img_tensor = pred_transform(image).unsqueeze(0).to(device)
181
182         # Get model prediction
183         with torch.no_grad():
184             outputs = model(img_tensor)
185             probabilities = F.softmax(outputs, dim=1)[0]
186
187             # Get the top prediction and all class probabilities
188             confidence_scores = {classes[i]: float(probabilities[i]) *
100 for i in range(len(classes))}
189             sorted_scores = sorted(confidence_scores.items(), key=
lambda x: x[1], reverse=True)
190             top_pred_class = sorted_scores[0][0]
191
192             return image, top_pred_class, confidence_scores
193
194     except Exception as e:
195         print(f"Error: Error during prediction: {e}")
196         return None, None, None
197
198 # Class for the prediction application GUI
199 class PredictionApp:
200     def __init__(self, root, available_models):
201         self.root = root
202         self.available_models = available_models
203         self.model = None
204         self.classes = None
205         self.device = None
206         self.current_model_type = None
207         self.current_image_path = None
208
209         # Set window properties
210         self.root.title("Image Classification")
211         self.root.geometry("1000x850") # Increased height to
accommodate model selector
212         self.root.configure(bg="#f0f0f0")
213
214         # Create main frame
215         self.main_frame = tk.Frame(root, bg="#f0f0f0")
216         self.main_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady
=20)
217
218         # Create header
219         self.header_label = tk.Label(
220             self.main_frame,
221             text="Image Classification",
222             font=("Arial", 24, "bold"),
223             bg="#f0f0f0"
224         )
225         self.header_label.pack(pady=(0, 10))
226
227         # Create model selection frame
228         self.model_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
229         self.model_frame.pack(fill=tk.X, pady=10)
230

```

```

231     # Add model selection label
232     tk.Label(
233         self.model_frame,
234         text="Select Model:",
235         font=("Arial", 14),
236         bg="#f0f0f0"
237     ).pack(side=tk.LEFT, padx=(0, 10))
238
239     # Add model selection dropdown
240     self.model_var = tk.StringVar(value=list(self.available_models.
keys())[0])
241     self.model_dropdown = ttk.Combobox(
242         self.model_frame,
243         textvariable=self.model_var,
244         values=list(self.available_models.keys()),
245         font=("Arial", 12),
246         state="readonly",
247         width=15
248     )
249     self.model_dropdown.pack(side=tk.LEFT, padx=(0, 10))
250
251     # Add model load button
252     self.load_model_button = tk.Button(
253         self.model_frame,
254         text="Load Model",
255         font=("Arial", 12),
256         command=self.load_selected_model,
257         bg="#4285F4",
258         fg="white",
259         padx=10,
260         pady=5
261     )
262     self.load_model_button.pack(side=tk.LEFT)
263
264     # Create button frame
265     self.button_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
266     self.button_frame.pack(fill=tk.X, pady=10)
267
268     # Add select image button (initially disabled)
269     self.select_button = tk.Button(
270         self.button_frame,
271         text="Select Image",
272         font=("Arial", 14),
273         command=self.select_image,
274         bg="#4CAF50",
275         fg="white",
276         padx=20,
277         pady=10,
278         relief=tk.RAISED,
279         borderwidth=2,
280         state=tk.DISABLED
281     )
282     self.select_button.pack(side=tk.LEFT, padx=(0, 10))
283
284     # Add quit button
285     self.quit_button = tk.Button(
286         self.button_frame,
287         text="Quit",

```

```

288         font=("Arial", 14),
289         command=self.quit_application,
290         bg="#F44336",
291         fg="white",
292         padx=20,
293         pady=10,
294         relief=tk.RAISED,
295         borderwidth=2
296     )
297     self.quit_button.pack(side=tk.RIGHT)
298
299     # Create content frame
300     self.content_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
301     self.content_frame.pack(fill=tk.BOTH, expand=True, pady=10)
302
303     # Create a frame for the figure
304     self.figure_frame = tk.Frame(self.content_frame, bg="#f0f0f0")
305     self.figure_frame.pack(fill=tk.BOTH, expand=True)
306
307     # Create matplotlib figure for the image and predictions
308     self.fig = plt.figure(figsize=(10, 6))
309     self.canvas = FigureCanvasTkAgg(self.fig, self.figure_frame)
310     self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
311
312     # Create feedback frame with a title
313     self.feedback_title = tk.Label(
314         self.main_frame,
315         text="Was the prediction correct?",
316         font=("Arial", 16, "bold"),
317         bg="#f0f0f0"
318     )
319     self.feedback_title.pack(pady=(15, 5))
320
321     # Create feedback frame
322     self.feedback_frame = tk.Frame(self.main_frame, bg="#f0f0f0")
323     self.feedback_frame.pack(fill=tk.X, pady=10)
324
325     # Add correct button
326     self.correct_button = tk.Button(
327         self.feedback_frame,
328         text="Correct",
329         font=("Arial", 14),
330         command=lambda: self.record_feedback(True),
331         bg="#4CAF50",
332         fg="white",
333         state=tk.DISABLED,
334         padx=25,
335         pady=10
336     )
337     self.correct_button.pack(side=tk.LEFT, padx=(0, 10))
338
339     # Add incorrect button
340     self.incorrect_button = tk.Button(
341         self.feedback_frame,
342         text="Incorrect",
343         font=("Arial", 14),
344         command=lambda: self.record_feedback(False),
345         bg="#F44336",

```

```

346         fg="white",
347         state=tk.DISABLED,
348         padx=25,
349         pady=10
350     )
351     self.incorrect_button.pack(side=tk.LEFT)
352
353     # Stats label
354     self.stats_label = tk.Label(
355         self.main_frame,
356         text="Total: 0 | Correct: 0 | Incorrect: 0 | Accuracy:
0.00%",
357         font=("Arial", 14, "bold"),
358         bg="#f0f0f0"
359     )
360     self.stats_label.pack(pady=10)
361
362     # Status message
363     self.status_label = tk.Label(
364         self.main_frame,
365         text="Please load a model to start",
366         font=("Arial", 12, "italic"),
367         fg="#555555",
368         bg="#f0f0f0"
369     )
370     self.status_label.pack(pady=(0, 10))
371
372     # Set up class variables
373     self.current_prediction = None
374
375     def load_selected_model(self):
376         """Load the model selected from the dropdown"""
377         model_type = self.model_var.get()
378
379         # Update status
380         self.status_label.config(text=f>Loading {model_type} model...")
381         self.root.update()
382
383         # Reset prediction stats for new model
384         global prediction_stats
385         prediction_stats = {
386             'total': 0,
387             'correct': 0,
388             'incorrect': 0,
389             'class_predictions': {},
390             'class_correct': {}
391         }
392
393         # Load model
394         project_root = os.path.dirname(os.path.abspath(__file__))
395         parent_dir = os.path.dirname(project_root)
396
397         config = MODEL_CONFIGS[model_type]
398         model_path = os.path.join(project_root, config['model_path'])
399
400         # Determine test directory based on directory structure
401         test_dir = None
402         if model_type == 'animals':

```

```

403         # Try to find the test directory in the parent directory (
main project directory)
404         test_dir = os.path.join(parent_dir, 'real_dataset', 'test',
'animals')
405         if not os.path.exists(test_dir):
406             test_dir = os.path.join(parent_dir, 'real_dataset', '
train', 'animals')
407
408         self.model, self.classes, self.device = load_model(model_type,
model_path, test_dir)
409
410         if self.model:
411             self.current_model_type = model_type
412             self.status_label.config(text=f"{model_type.capitalize()}
model loaded successfully. Please select an image.")
413             self.select_button.config(state=tk.NORMAL)
414             self.stats_label.config(text="Total: 0 | Correct: 0 |
Incorrect: 0 | Accuracy: 0.00%")
415         else:
416             self.status_label.config(text=f"Error loading {model_type}
model. Please check the model file or directory structure.")
417             self.select_button.config(state=tk.DISABLED)
418
419     def select_image(self):
420         """Open a file dialog to select an image"""
421         filetypes = [
422             ("Image files", "*.jpg *.jpeg *.png *.bmp *.gif"),
423             ("All files", "*.*")
424         ]
425
426         filepath = filedialog.askopenfilename(
427             title="Select Image",
428             filetypes=filetypes
429         )
430
431         if filepath:
432             self.current_image_path = filepath
433             self.predict_and_display(filepath)
434
435     def predict_and_display(self, image_path):
436         """Run prediction and display results"""
437         # Update status
438         self.status_label.config(text="Analyzing image...")
439         self.root.update()
440
441         # Run prediction
442         image, prediction, confidence_scores = predict_image(
443             image_path, self.model, self.classes, self.device
444         )
445
446         if image and prediction and confidence_scores:
447             # Store current prediction
448             self.current_prediction = prediction
449
450             # Clear previous figure
451             self.fig.clear()
452
453             # Create two subplots - one for image, one for bar chart

```



```

454         ax1 = self.fig.add_subplot(1, 2, 1)
455         ax2 = self.fig.add_subplot(1, 2, 2)
456
457         # Display image
458         ax1.imshow(image)
459         ax1.set_title(f"Prediction: {prediction}")
460         ax1.axis('off')
461
462         # Create bar chart of confidence scores
463         sorted_scores = sorted(confidence_scores.items(), key=
lambda x: x[1], reverse=True)
464         classes = [item[0] for item in sorted_scores]
465         scores = [item[1] for item in sorted_scores]
466
467         bars = ax2.bar(classes, scores, color=['#4285F4' if cls ==
prediction else '#A0A0A0' for cls in classes])
468         ax2.set_ylabel('Confidence (%)')
469         ax2.set_title('Class Predictions')
470         ax2.set_ylim([0, 100])
471
472         # Add percentage labels above bars
473         for bar in bars:
474             height = bar.get_height()
475             ax2.annotate(f'{height:.1f}%',
476                         xy=(bar.get_x() + bar.get_width() / 2,
height),
477                         xytext=(0, 3), # 3 points vertical offset
478                         textcoords="offset points",
479                         ha='center', va='bottom',
480                         fontsize=9)
481
482         # Rotate x-axis labels for better readability if needed
483         if len(classes) > 3:
484             plt.setp(ax2.get_xticklabels(), rotation=45, ha='right'
)
485
486         # Update the canvas
487         self.fig.tight_layout()
488         self.canvas.draw()
489
490         # Enable feedback buttons
491         self.correct_button.config(state=tk.NORMAL)
492         self.incorrect_button.config(state=tk.NORMAL)
493
494         # Update status
495         self.status_label.config(text=f"Prediction complete. Model
says: {prediction}")
496
497         # Update statistics display
498         global prediction_stats
499         prediction_stats['total'] += 1
500         prediction_stats['class_predictions'][prediction] += 1
501         self.update_stats_display()
502     else:
503         # Handle prediction failure
504         self.status_label.config(text="Error analyzing image.
Please try another image.")
505         messagebox.showerror("Prediction Error", "Could not analyze

```

```

the selected image.")
506
507 def record_feedback(self, is_correct):
508     """Record user feedback on prediction accuracy"""
509     if self.current_prediction:
510         global prediction_stats
511
512         if is_correct:
513             prediction_stats['correct'] += 1
514             prediction_stats['class_correct'][self.
current_prediction] += 1
515             feedback_msg = "    Feedback recorded: Prediction was
correct!"
516         else:
517             prediction_stats['incorrect'] += 1
518             feedback_msg = "    Feedback recorded: Prediction was
incorrect."
519
520             # Optionally ask for correct class if prediction was
wrong
521             if len(self.classes) > 2: # Only for multi-class
problems
522                 correct_class = simplifiedialog.askstring(
523                     "Correct Class",
524                     f"What was the correct class?\nOptions: {'', '
'.join(self.classes)}",
525                     parent=self.root
526                 )
527
528                 if correct_class and correct_class in self.classes:
529                     feedback_msg += f" (Correct class: {
correct_class})"
530
531             # Update status and stats display
532             self.status_label.config(text=feedback_msg)
533             self.update_stats_display()
534
535             # Reset buttons for next prediction
536             self.correct_button.config(state=tk.DISABLED)
537             self.incorrect_button.config(state=tk.DISABLED)
538
539 def update_stats_display(self):
540     """Update the statistics display label"""
541     global prediction_stats
542
543     total = prediction_stats['total']
544     correct = prediction_stats['correct']
545     incorrect = prediction_stats['incorrect']
546
547     if total > 0:
548         accuracy = (correct / total) * 100
549         stats_text = f"Total: {total} | Correct: {correct} |
Incorrect: {incorrect} | Accuracy: {accuracy:.2f}%"
550     else:
551         stats_text = "Total: 0 | Correct: 0 | Incorrect: 0 |
Accuracy: 0.00%"
552
553     self.stats_label.config(text=stats_text)

```

```

554
555     def quit_application(self):
556         """Exit the application and show final statistics"""
557         global prediction_stats
558
559         # Create final statistics message
560         total = prediction_stats['total']
561
562         if total > 0:
563             accuracy = (prediction_stats['correct'] / total) * 100
564             message = f"Session Statistics:\n\n"
565             message += f"Total predictions: {total}\n"
566             message += f"Correct: {prediction_stats['correct']} ({(
prediction_stats['correct']/total)*100:.2f}%) \n"
567             message += f"Incorrect: {prediction_stats['incorrect']} ({(
prediction_stats['incorrect']/total)*100:.2f}%) \n\n"
568
569             # Add per-class statistics
570             message += "Class Performance:\n"
571             for cls in self.classes:
572                 predictions = prediction_stats['class_predictions'].get
(cls, 0)
573                 correct = prediction_stats['class_correct'].get(cls, 0)
574
575                 if predictions > 0:
576                     class_accuracy = (correct / predictions) * 100
577                     message += f"{cls}: {correct}/{predictions} correct
({class_accuracy:.2f}%) \n"
578                 else:
579                     message += f"{cls}: No predictions\n"
580
581             messagebox.showinfo("Session Statistics", message)
582
583             self.root.destroy()
584
585 # Run the application
586 if __name__ == "__main__":
587     root = tk.Tk()
588     app = PredictionApp(root, MODEL_CONFIGS)
589     root.mainloop()

```

Listing A.2: Complete prediction application