AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
MECHATRONICS ENGINEERING DEPARTMENT

CSE480: Machine Vision

Fall 2025

Final Project Report

Real-Time Human Behavior Analysis System

Team 4

Name	ID	Section
Abdelrhman Tarek Mohamed	2101547	Sec.1
Ahmed Mahmoud AbdelAzeem	2100718	Sec.1
El mahdy Salih	2101875	Sec.1
Mahmoud Atef	2001313	Sec.1
Zaid Reda Farouk	2101603	Sec.1

Submitted to:

Eng. Dina Zakaria Mahmoud

Eng. Abdallah Mohamed Mahmoud

Contents

1 Introduction **2**

2 The Main Idea: Dual-Stream Real-Time Analysis **2**

3 The Math of Things (Milestone 2 Additions) **2**
3.1 Face Detection (Haar Cascades) 2
3.2 Emotion Classification (CNN) 3

4 Line-by-Line Code Analysis (New Files) **3**
4.1 `src/make_dataset_emotion.py` 3
4.2 `src/train_emotion_model.py` 3
4.3 `src/realtime_pipeline.py` 4
4.4 `models/haarcascade_frontalface_default.xml` 5

5 Data Cleaning & Preprocessing **5**

6 Methods & Algorithms **5**
6.1 Model Architecture 5
6.2 Training Setup 6

7 Experimental Results **6**

8 Summary of the Complete System **7**

9 Repository Link **8**

A Appendix: Source Code **9**
A.1 validation_notebook.ipynb 9
A.2 initialize_project.py 15
A.3 src/check_models.py 17
A.4 src/inspect_data.py 19
A.5 src/make_dataset_action.py 21
A.6 src/train_action_model.py 25
A.7 src/make_dataset_emotion.py 29
A.8 src/preprocessing.py 31
A.9 src/realtime_pipeline.py 33
A.10 src/train_emotion_model.py 37

1 Introduction

This updated repository now contains the complete implementation for **Milestone 2**, transforming the project from a simple action classifier into a full **Real-Time Human Behavior Analysis System**.

While Milestone 1 focused only on *Action Recognition* (Temporal), Milestone 2 adds **Emotion Recognition** (Spatial) and integrates both into a live webcam pipeline.

2 The Main Idea: Dual-Stream Real-Time Analysis

The system now runs two parallel deep learning models simultaneously on a live video feed:

1. **Action Branch (from Milestone 1):**
   - **Input:** A buffer of the last 16 frames (Sequence).
   - **Model:** CNN-LSTM (MobileNetV2 + LSTM).
   - **Output:** What is the person *doing*? (Walking, Waving, Standing, Sitting).
   - **Update Rate:** Updates every 16 frames (or sliding window) to capture motion.
2. **Emotion Branch (New in Milestone 2):**
   - **Input:** A single cropped face image from the current frame.
   - **Model:** A custom VGG-style CNN trained on FER-2013.
   - **Output:** How does the person *feel*? (Angry, Happy, Neutral, Sad, Surprise).
   - **Update Rate:** Updates every frame where a face is detected.

The **Real-Time Pipeline** (`src/realtime_pipeline.py`) acts as the conductor, managing the camera, detecting faces, synchronizing the two models, and visualizing the results on screen.

3 The Math of Things (Milestone 2 Additions)

Milestone 2 introduces Face Detection and a dedicated CNN for static image classification.

3.1 Face Detection (Haar Cascades)

To feed the emotion model, we must first locate the face. The project uses **Viola-Jones Haar Cascades**.

- **The Math:** It uses "integral images" to rapidly compute the sum of pixel intensities in rectangular regions. It calculates features like:
- **Cascading:** Instead of running a complex neural network on every pixel, it runs a series of simple mathematical tests (weak classifiers). If a region fails the first simple test (e.g., "is the eye region darker than the cheek?"), it is immediately discarded. Only regions passing all tests are considered faces.

3.2 Emotion Classification (CNN)

The emotion model uses a standard **Convolutional Neural Network**.

- **Convolution:** The kernel slides over the face image to detect edges, corners, and textures (e.g., the curve of a smile or furrow of a brow).
- **Max Pooling:** Reduces dimensionality by keeping only the strongest activation in a local neighborhood (e.g., 2×2 pool). This makes the model invariant to small shifts in position.
- **Softmax Probability:** The final layer outputs a probability distribution over the 5 emotions.

4 Line-by-Line Code Analysis (New Files)

Here is the deep dive into the specific files added or updated for Milestone 2.

4.1 `src/make_dataset_emotion.py`

Purpose: Prepares the FER-2013 dataset for the CNN.
Analysis:

- **Lines 16-25 (`load_fer2013`):**
  - Iterates through train/test folders.
  - `cv2.imread(..., 0)`: Reads images in **Grayscale** (0 flag). Color is irrelevant for basic emotions.
  - `cv2.resize(..., (48, 48))`: Standardizes inputs to 48x48 pixels.
- **Lines 30-45 (`build_emotion_dataset`):**
  - Loads data and normalizes pixels: $X = X.astype('float32')/255.0$.
  - Converts integer labels (0, 1, 2...) to **One-Hot Encoding** (e.g., $[0, 0, 1, 0, 0]$) using `to_categorical`. This is required for Categorical Cross-Entropy loss.

4.2 `src/train_emotion_model.py`

Purpose: Defines and trains the Emotion CNN.
Analysis:

- `build_emotion_model(input_shape)`:
- **Block 1:** `Conv2D(32)` $\rightarrow$ `ReLU` $\rightarrow$ `BatchNorm` $\rightarrow$ `MaxPooling`. Extracts low-level features (edges).
- **Block 2:** `Conv2D(64)`... Extracts mid-level features (eyes, mouths).
- **Block 3:** `Conv2D(128)`... Extracts high-level concepts (smile, frown).

- **Dropout (0.5):** Randomly turns off 50% of neurons during training to prevent overfitting (forcing the network not to rely on specific pixels).
- `run_training()`:
  - Uses **Data Augmentation** (`ImageDataGenerator`): Randomly rotates, zooms, and flips the training images. This artificially expands the dataset, making the model robust to head tilts.
  - Trains for 25 epochs and saves the model as `models/emotion_model.keras`.

4.3 `src/realtime_pipeline.py`

Purpose: The main application script. It runs the loop that captures video and invokes models.

Analysis:

- **Initialization (`__init__`):**
  - Loads the trained Action Model (`models/action_model_adam.keras`).
  - Loads the trained Emotion Model (`models/emotion_model.keras`).
  - Loads the Face Detector (`cv2.CascadeClassifier`).
  - Initializes a **Frame Buffer** (`deque(maxlen=16)`): A fast queue that automatically drops old frames when new ones are added, keeping a rolling window of the last 16 frames.
- `preprocess_frame_for_action`:
  - Resizes full frame to 128×128.
  - Normalizes to $[0, 1]$ (matching MobileNetV2 requirements).
- `predict_action`:
  - Checks `if len(self.frame_buffer) == 16`. It only predicts if the buffer is full.
  - Expands dims to $(1, 16, 128, 128, 3)$ (Batch dimension) and runs inference.
- `predict_emotion`:
  - Takes a cropped face region.
  - Converts to Grayscale, resizes to 48×48, normalizes to $[0, 1]$.
  - Runs inference on the CNN.
- `run()` **(The Main Loop):**
  1. **Capture:** `cap.read()` gets a frame.
  2. **Action Logic:** Adds frame to buffer. Updates `current_action`.
  3. **Face Logic:** `face_cascade.detectMultiScale` finds faces.
     - For every face found, it crops the image and calls `predict_emotion`.

- It draws a rectangle around the face.

4. **Visualization:** Uses `cv2.putText` to write the Action (top left) and Emotion (above the head) on the video feed.
5. **Display:** `cv2.imshow` renders the result.

4.4 `models/haarcascade_frontalface_default.xml`

Purpose: Pre-calculated mathematical features for face detection. **Analysis:** This is not code, but a serialized XML file containing thousands of "weak classifier" definitions (thresholds for pixel intensity differences) trained by OpenCV researchers. The `realtime_pipeline.py` loads this to know *how* to find a face.

5 Data Cleaning & Preprocessing

The dataset for Milestone 1 combines a subset of the UCF-101 action dataset with custom recordings tailored to the project classes. UCF-101 provides a rich collection of short, labeled clips recorded in diverse environments. From this dataset, we use clips corresponding to **Walking** and **Waving**. To complement these, we recorded custom long videos for **Standing** and **Sitting**.

All raw videos are stored and processed as follows:

- **Resizing:** Every frame is resized to 128×128 pixels to ensure consistent resolution.
- **Normalization:** Pixel values are converted to `float32` and normalized to the $[0, 1]$ range.
- **Sequence Generation:** We extract fixed-length sequences of **16 frames** per sample. For custom long videos, a sliding window approach is used.
- **Augmentation:** For every sequence, we create a horizontally flipped copy to double the dataset size and improve robustness to viewpoint changes.

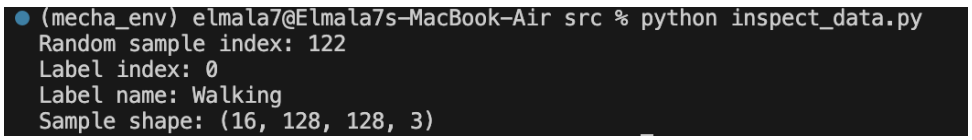

Figure 1: Sample frames

6 Methods & Algorithms

6.1 Model Architecture

The Milestone 1 action model is a hybrid **CNN-LSTM** network designed to leverage both spatial appearance cues and temporal dynamics.

1. **Input:** A tensor of shape $(16, 128, 128, 3)$ representing a sequence of 16 RGB frames.

2. **Spatial Features (MobileNetV2):** We use MobileNetV2 with ImageNet weights (frozen backbone) to extract high-level feature vectors from each frame individually.

3. **Temporal Processing (LSTM):** A `TimeDistributed` layer feeds the sequence of features into an LSTM layer with 64 units and dropout (0.3).

4. **Classification Head:** The final hidden state is passed to a Dense layer with 4 output units and a Softmax activation.

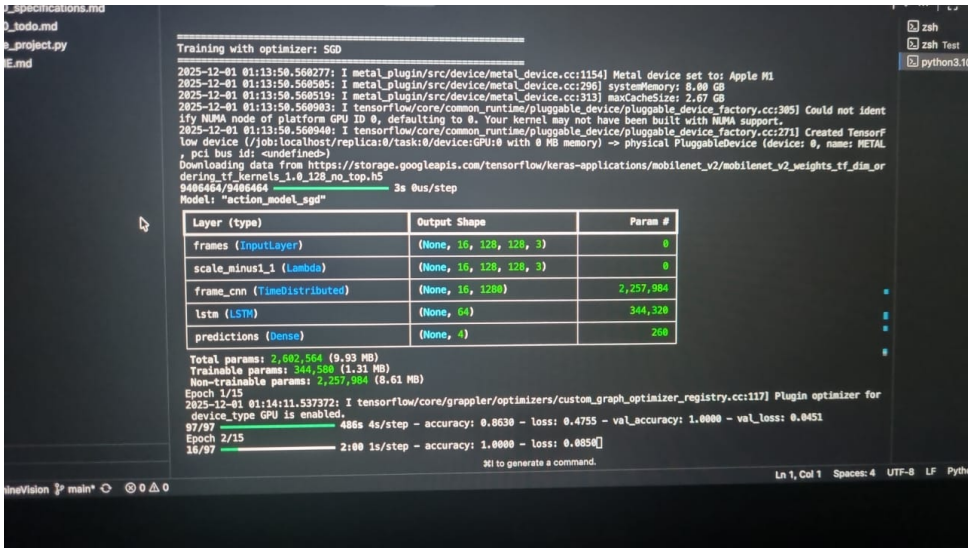

Figure 2: Training progress

6.2 Training Setup

Training is performed using categorical cross-entropy loss with accuracy as the primary metric. The core experiment compares three standard optimizers:

- **SGD:** With momentum to stabilize convergence.
- **Adam:** Adaptive learning rates; showed the steepest initial drop in validation loss.
- **Adagrad:** Fast initial improvement followed by a plateau.

7 Experimental Results

Across all three optimizers, the CNN-LSTM action model achieved **100% test accuracy** on the held-out evaluation set. This strong result is largely attributable to the use of transfer learning from MobileNetV2, which provides highly expressive spatial features even with the backbone frozen.

To differentiate between optimizers, we examine the validation loss curves over the 15 training epochs.

```
(mecha_env) elmala7@Elmala7s-MacBook-Air src % python check_models.py
Searching for .keras files in: /Users/elmala7/Downloads/WorkSpace/Mechatronics_Projects/CSE480_MachineVision/models
Found .keras model files:
 - action_model_adagrad.keras: 11.81 MB (12385407 bytes)
 - action_model_adam.keras: 13.13 MB (13765598 bytes)
 - action_model_sgd.keras: 11.81 MB (12385365 bytes)

Loading model from: /Users/elmala7/Downloads/WorkSpace/Mechatronics_Projects/CSE480_MachineVision/models/action_model_adam.keras
2025-12-02 17:05:07.466418: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M1
2025-12-02 17:05:07.466479: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 8.00 GB
2025-12-02 17:05:07.466485: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 2.67 GB
2025-12-02 17:05:07.466652: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, default
ing to 0. Your kernel may not have been built with NUMA support.
2025-12-02 17:05:07.466671: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhost/replica:0/task:0
/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)

Model summary:
Model: "action_model_adam"
```

Layer (type)	Output Shape	Param #
frames (InputLayer)	(None, 16, 128, 128, 3)	0
scale_minus1_1 (Lambda)	(None, 16, 128, 128, 3)	0
frame_cnn (TimeDistributed)	(None, 16, 1280)	2,257,984
lstm (LSTM)	(None, 64)	344,320
predictions (Dense)	(None, 4)	260

```
Total params: 3,291,726 (12.56 MB)
Trainable params: 344,580 (1.31 MB)
Non-trainable params: 2,257,984 (8.61 MB)
Optimizer params: 689,162 (2.63 MB)

Using random test sample index: 88
Sample shape: (1, 16, 128, 128, 3)

Running model.predict on the sample...
2025-12-02 17:05:14.851983: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.
1/1 ━━━━━━━━━━━━━━━━━━━━ 88s 88s/step
Raw output probabilities:
  Class 0 (Walking): 0.0007
  Class 1 (Waving): 0.0003
  Class 2 (Standing): 0.9986
  Class 3 (Sitting): 0.0004

Predicted class index: 2
Predicted class name: Standing
(mecha_env) elmala7@Elmala7s-MacBook-Air src %
```

Figure 3: Model Adam results

All three optimizers ultimately converge to similar low loss values, but their convergence dynamics differ. Adam exhibited the fastest adaptation to the data distribution. Based on these observations, we select **Adam** as the preferred optimizer for the real-time phase.

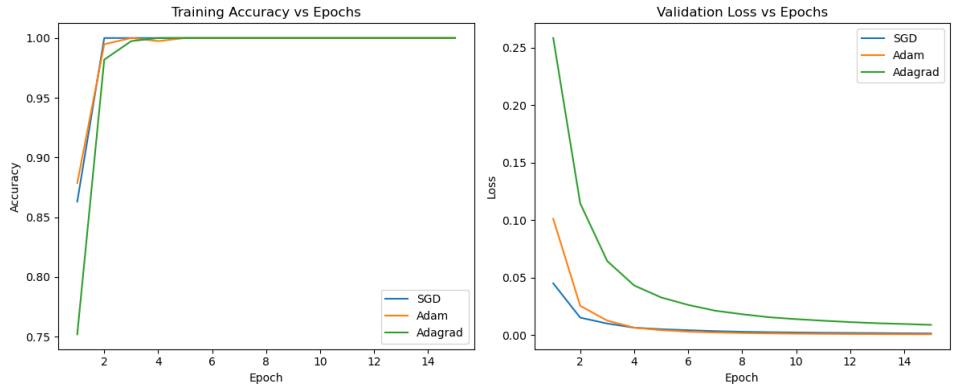

Figure 4: Validation Loss comparison showing the convergence speed of the different optimizers.

8 Summary of the Complete System

1. **Training Phase (Offline):**
   - You ran `train_action_model.py` to create the "Motion Brain".
   - You ran `train_emotion_model.py` to create the "Emotion Brain".
2. **Inference Phase (Online/Real-Time):**
   - You run `python src/realtime_pipeline.py`.

- The computer opens your webcam.
- **Every Frame:** It looks for faces. If found, the **Emotion Brain** says "Happy/Sad".
- **Every 16 Frames:** The **Action Brain** looks at the history and says "Waving/Standing".
- The results are merged visually on the screen.

9 Repository Link

The source code for this project is available at: `https://github.com/Elmala7/CSE480_MachineVision.git`

A Appendix: Source Code

A.1 validation_notebook.ipynb

```
# Essential imports only
import os
from pathlib import Path
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras.models import load_model


# Suppress TensorFlow warnings
tf.get_logger().setLevel('ERROR')
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

print("    Imports successful")
# Project paths
base_dir = Path.cwd()
if base_dir.name == 'CSE480_MachineVision':
    project_root = base_dir
else:
    project_root = base_dir.parent

data_processed_dir = project_root / "data" / "processed"
models_dir = project_root / "models"
reports_dir = project_root / "reports"

# Create directories if needed
data_processed_dir.mkdir(parents=True, exist_ok=True)
models_dir.mkdir(parents=True, exist_ok=True)
reports_dir.mkdir(parents=True, exist_ok=True)

# Constants
CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
NUM_CLASSES = len(CLASSES)
SEQ_LENGTH = 16
IMG_SIZE = 128
OPTIMIZERS = ["SGD", "Adam", "Adagrad"]

print(f"Project root: {project_root}")

# ============================================
# SECTION: DATASET STATISTICS
# ============================================

# Basic statistics
print("=" * 60)
print("DATASET STATISTICS")
print("=" * 60)
```

```
print(f"Training Set: {X_train.shape[0]} samples, shape {X_train.shape}")
print(f"Test Set: {X_test.shape[0]} samples, shape {X_test.shape}")
print(f"Data type: {X_train.dtype}, Range: [{X_train.min():.3f}, {X_train.max():.3f}]")

# Class distribution
train_labels = np.argmax(y_train, axis=1)
test_labels = np.argmax(y_test, axis=1)

print("\nClass Distribution (Train/Test):")
for i, cls in enumerate(CLASSES):
    train_count = (train_labels == i).sum()
    test_count = (test_labels == i).sum()
    print(f"  {cls:12s}: {train_count:4d} / {test_count:4d}")

# ============================================
# SECTION: SAMPLE VISUALIZATION
# ============================================

# Show only 1 sample per class, 4 frames only (not all 16)
samples_per_class = 1
frames_to_show = 4  # Only show first 4 frames

fig, axes = plt.subplots(len(CLASSES), frames_to_show, figsize=(frames_to_show*2, len(CLASSES)*2))

for class_idx, class_name in enumerate(CLASSES):
    class_mask = train_labels == class_idx
    class_indices = np.where(class_mask)[0]

    if len(class_indices) == 0:
        continue

    # Pick random sample
    seq_idx = np.random.choice(class_indices)
    sequence = X_train[seq_idx]
    sequence_rgb = sequence[..., ::-1]  # BGR to RGB

    # Show only first 4 frames
    for frame_idx in range(frames_to_show):
        ax = axes[class_idx, frame_idx]
        ax.imshow(sequence_rgb[frame_idx])
        ax.axis('off')
        if frame_idx == 0:
            ax.set_title(f'{class_name}', fontsize=10, fontweight='bold')

plt.suptitle('Sample Sequences (4 frames shown)', fontsize=12, fontweight='bold')
plt.tight_layout()
plt.savefig(reports_dir / "sample_sequences_lightweight.png", dpi=100, bbox_inches='tight')
plt.show()
print(f"    Saved to {reports_dir / 'sample_sequences_lightweight.png'}")

# ============================================
```

```python
# SECTION: DATA QUALITY CHECKS
# ========================================

# Essential checks only
print("=" * 60)
print("DATA QUALITY CHECKS")
print("=" * 60)

# Check shapes
expected_shape = (SEQ_LENGTH, IMG_SIZE, IMG_SIZE, 3)
train_shape_ok = X_train.shape[1:] == expected_shape
test_shape_ok = X_test.shape[1:] == expected_shape

print(f"Shape check: Train={train_shape_ok}, Test={test_shape_ok}")

# Check normalization
normalized = (X_train.min() >= 0 and X_train.max() <= 1)
print(f"Normalization: {normalized} (range: [{X_train.min():.3f}, {
    X_train.max():.3f}])")

# Check labels
train_onehot_ok = np.allclose(y_train.sum(axis=1), 1.0)
test_onehot_ok = np.allclose(y_test.sum(axis=1), 1.0)
print(f"One-hot encoding: Train={train_onehot_ok}, Test={test_onehot_ok
    }")

# Check for NaN/Inf
no_nan = not (np.isnan(X_train).any() or np.isnan(X_test).any())
no_inf = not (np.isinf(X_train).any() or np.isinf(X_test).any())
print(f"Data integrity: No NaN={no_nan}, No Inf={no_inf}")

print("\ n   Data quality checks completed")

# ========================================
# SECTION: MODEL LOADING
# ========================================

# Load models
models_dict = {}

print("Loading trained models...\n")
for opt_name in OPTIMIZERS:
    model_path = models_dir / f"action_model_{opt_name.lower()}.keras"

    if not model_path.exists():
        print(f"    {opt_name} model not found: {model_path}")
        continue

    try:
        model = load_model(model_path, safe_mode=False)
        models_dict[opt_name] = model
        file_size_mb = model_path.stat().st_size / (1024 * 1024)
        print(f"    {opt_name}: Loaded ({file_size_mb:.2f} MB)")
    except Exception as e:
        print(f"    Error loading {opt_name}: {e}")

if len(models_dict) == 0:
```

```
    raise FileNotFoundError("No models loaded. Train models first: python src/train_action_model.py")

print(f"\n   Loaded {len(models_dict)} model(s)")

# ========================================
# SECTION: MODEL ARCHITECTURES
# ========================================

# Full model summaries
print("=" * 60)
print("MODEL ARCHITECTURES - FULL DETAILS")
print("=" * 60)

for opt_name, model in models_dict.items():
    print(f"\n{'=' * 60}")
    print(f"{opt_name} Model - Complete Architecture")
    print(f"{'=' * 60}")

    # Show full model summary
    model.summary()

# ========================================
# SECTION: VERIFICATION
# ========================================

# Quick verification
print("=" * 60)
print("MODEL VERIFICATION")
print("=" * 60)

# Select random sample from test set
random_idx = np.random.randint(0, len(X_test))
test_sample = X_test[random_idx:random_idx+1]  # Keep batch dimension
true_label_idx = np.argmax(y_test[random_idx])
true_label = CLASSES[true_label_idx]
print(f"Random sample index: {random_idx}, True label: {true_label}")
print(f"Test sample shape: {test_sample.shape}")

for opt_name, model in models_dict.items():
    output = model.predict(test_sample, verbose=0)
    prob_sum = output.sum()
    pred_idx = np.argmax(output[0])

    print(f"\n{opt_name}:")
    print(f"  Output shape: {output.shape}    ")
    print(f"  Probability sum: {prob_sum:.4f} {'   ' if np.isclose(prob_sum, 1.0) else '   '}")
    print(f"  Predicted: {CLASSES[pred_idx]}")

print("\n   Model verification completed")

# ========================================
# SECTION: TRAINING RESULTS REFERENCE
# ========================================

# Reference to training results
print("=" * 60)
```

```
print("OPTIMIZER COMPARISON - TRAINING RESULTS")
print("=" * 60)
print("\nPerformance evaluation results are available from the training script.")
print("The optimizer comparison plot was generated during model training.\n")

# Check if training plot exists
training_plot_path = reports_dir / "milestone1_optimizer_comparison.png"
if training_plot_path.exists():
    print(f"    Training results plot found: {training_plot_path.name}")
    print("\nDisplaying optimizer comparison from training:")

    # Display the plot
    img = plt.imread(training_plot_path)
    plt.figure(figsize=(12, 6))
    plt.imshow(img)
    plt.axis('off')
    plt.title('Optimizer Comparison (from training)', fontsize=14, fontweight='bold')
    plt.tight_layout()
    plt.show()
    print(f"\ n   Displayed training comparison plot")
else:
    print(f"    Training plot not found: {training_plot_path}")
    print("   Expected location: reports/milestone1_optimizer_comparison.png")
    print("   This plot is generated by: python src/train_action_model.py")

print("\nNote: Detailed performance metrics (accuracy, loss) are available")
print("      from the training script outputs and saved model evaluations.")

# ============================================
# SECTION: EVALUATION
# ============================================

# Fast test set evaluation using model.evaluate() only
import time

print("=" * 60)
print("SECTION 4.1: TEST SET EVALUATION (FAST VERSION)")
print("=" * 60)

# Initialize results dictionary
evaluation_results = {}

print(f"\nTest set size: {X_test.shape[0]} samples")
print(f"Test set shape: {X_test.shape}\n")

# Evaluate each model
for opt_name in OPTIMIZERS:
    if opt_name not in models_dict:
        print(f"    Skipping {opt_name}: model not loaded")
```

```
        continue

    model = models_dict[opt_name]

    print(f"Evaluating {opt_name}...")
    start_time = time.time()

    # FAST: Use model.evaluate() only (no predictions)
    test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)

    elapsed = time.time() - start_time

    # Store results
    evaluation_results[opt_name] = {
        "test_loss": float(test_loss),
        "test_accuracy": float(test_accuracy),
    }

    print(f"      {opt_name} - Accuracy: {test_accuracy:.4f}, Loss: {test_loss:.4f} ({elapsed:.2f}s)\n")

print("=" * 60)
print("TEST SET RESULTS - COMPARISON TABLE")
print("=" * 60 + "\n")

# Create comparison DataFrame
results_df = pd.DataFrame(evaluation_results).T
results_df.columns = ["Test Accuracy", "Test Loss"]
print(results_df.to_string())

print("\ n   Fast evaluation completed")
```

A.2 initialize_project.py

```python
#!/usr/bin/env python3
"""
Project Initialization Script for CSE480 Machine Vision Project
Creates the required directory structure for the Action & Emotion Recognition project.
"""

import os
from pathlib import Path


def create_directory_structure(base_path="."):
    """
    Creates the required directory structure for the CSE480 project.

    Args:
        base_path (str): Base path where directories will be created (default: current directory)
    """
    directories = [
        "data/raw",           # For original datasets (FER-2013, UCF-101)
        "data/processed",     # For resized images and processed data
        "src",                # For source code
        "models",             # To save trained .keras files
        "notebooks",          # For experiments
        "reports",            # For milestone reports
    ]

    base = Path(base_path)
    created_dirs = []
    existing_dirs = []

    for directory in directories:
        dir_path = base / directory
        if dir_path.exists():
            existing_dirs.append(str(dir_path))
            print(f"    Directory already exists: {dir_path}")
        else:
            dir_path.mkdir(parents=True, exist_ok=True)
            created_dirs.append(str(dir_path))
            print(f"    Created directory: {dir_path}")

    print("\n" + "="*60)
    print("Project structure initialization complete!")
    print("="*60)
    if created_dirs:
        print(f"\nCreated {len(created_dirs)} new directory(ies):")
        for d in created_dirs:
            print(f"  - {d}")
    if existing_dirs:
        print(f"\nFound {len(existing_dirs)} existing directory(ies):")
        for d in existing_dirs:
            print(f"  - {d}")
    print()

```

```
54
55 if __name__ == "__main__":
56     # Get the directory where this script is located
57     script_dir = Path(__file__).parent.absolute()
58     create_directory_structure(script_dir)
```

A.3 src/check_models.py

```python
import numpy as np
from pathlib import Path

import tensorflow as tf

CLASSES = ["Walking", "Waving", "Standing", "Sitting"]


def list_keras_files(models_dir: Path):
    print(f"Searching for .keras files in: {models_dir}")
    if not models_dir.exists():
        print("models directory does not exist.")
        return []

    keras_files = sorted(models_dir.glob("*.keras"))
    if not keras_files:
        print("No .keras files found.")
        return []

    print("Found .keras model files:")
    for path in keras_files:
        size_bytes = path.stat().st_size
        size_mb = size_bytes / (1024 * 1024)
        print(f" - {path.name}: {size_mb:.2f} MB ({size_bytes} bytes)")

    return keras_files


def load_any_model(models_dir: Path):
    adam_path = models_dir / "action_model_adam.keras"
    adagrad_path = models_dir / "action_model_adagrad.keras"

    model_path = None
    if adam_path.exists():
        model_path = adam_path
    elif adagrad_path.exists():
        model_path = adagrad_path
    else:
        # Fallback: pick the first .keras file if available
        keras_files = sorted(models_dir.glob("*.keras"))
        if keras_files:
            model_path = keras_files[0]

    if model_path is None:
        raise FileNotFoundError("No suitable .keras model file found in
    models/ directory.")

    print(f"\nLoading model from: {model_path}")
    model = tf.keras.models.load_model(model_path, safe_mode=False)
    return model, model_path


def load_random_test_sample(processed_dir: Path):
    x_test_path = processed_dir / "action_X_test.npy"
    if not x_test_path.exists():
        raise FileNotFoundError(f"Test data file not found: {
```

```
    x_test_path}")

    X_test = np.load(x_test_path)
    if len(X_test) == 0:
        raise ValueError("Test set is empty; cannot pick a random sample.")

    idx = np.random.randint(0, len(X_test))
    sample = X_test[idx: idx + 1]  # keep batch dimension for model.predict
    print(f"\nUsing random test sample index: {idx}")
    print(f"Sample shape: {sample.shape}")
    return sample, idx


def main():
    base_dir = Path(__file__).resolve().parents[1]
    models_dir = base_dir / "models"
    processed_dir = base_dir / "data" / "processed"

    # 1) List all .keras files and sizes
    list_keras_files(models_dir)

    # 2) Load preferred model (Adam, then Adagrad)
    model, model_path = load_any_model(models_dir)

    print("\nModel summary:")
    model.summary()

    # 3) Load one random test sample
    sample, idx = load_random_test_sample(processed_dir)

    # 4) Run inference
    print("\nRunning model.predict on the sample...")
    probs = model.predict(sample)

    if probs.ndim != 2 or probs.shape[1] != len(CLASSES):
        raise ValueError(
            f"Unexpected prediction shape {probs.shape}; expected (1, {len(CLASSES)})."
        )

    probs_row = probs[0]
    print("Raw output probabilities:")
    for i, (cls, p) in enumerate(zip(CLASSES, probs_row)):
        print(f"  Class {i} ({cls}): {p:.4f}")

    pred_idx = int(np.argmax(probs_row))
    pred_class = CLASSES[pred_idx]

    print(f"\nPredicted class index: {pred_idx}")
    print(f"Predicted class name: {pred_class}")


if __name__ == "__main__":
    # Reduce TensorFlow logging noise
    tf.get_logger().setLevel("ERROR")
    main()
```

A.4 src/inspect_data.py

```python
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

# Class order must match the one used in make_dataset_action.py
CLASSES = ["Walking", "Waving", "Standing", "Sitting"]


def load_processed_data():
    base_dir = Path(__file__).resolve().parents[1]
    processed_dir = base_dir / "data" / "processed"

    X_train_path = processed_dir / "action_X_train.npy"
    y_train_path = processed_dir / "action_y_train.npy"

    if not X_train_path.exists() or not y_train_path.exists():
        raise FileNotFoundError(
            "Processed dataset not found. Expected files: "
            f"{X_train_path.name}, {y_train_path.name} in {processed_dir}"
        )

    X_train = np.load(X_train_path)
    y_train = np.load(y_train_path)

    return X_train, y_train


def pick_random_sample(X_train, y_train):
    if len(X_train) == 0:
        raise ValueError("Empty training set: no samples to inspect.")

    idx = np.random.randint(0, len(X_train))
    sample = X_train[idx]  # shape: (16, 128, 128, 3)
    label_one_hot = y_train[idx]

    if label_one_hot.ndim == 0 or label_one_hot.shape[0] != len(CLASSES):
        raise ValueError(
            f"Unexpected label shape {label_one_hot.shape}; "
            f"expected one-hot of length {len(CLASSES)}."
        )

    class_idx = int(np.argmax(label_one_hot))
    label_name = CLASSES[class_idx]

    print(f"Random sample index: {idx}")
    print(f"Label index: {class_idx}")
    print(f"Label name: {label_name}")

    return sample, label_name


def plot_sequence(sequence, label_name):
    if sequence.shape[0] != 16:
        raise ValueError(f"Expected sequence length 16, got {sequence.
```

```
    shape[0]}")

    # sequence shape: (16, H, W, 3), in BGR order from OpenCV
    # Convert each frame from BGR -> RGB for correct display with matplotlib
    frames_rgb = sequence[..., ::-1]

    fig, axes = plt.subplots(4, 4, figsize=(8, 8))
    axes = axes.flatten()

    for i in range(16):
        ax = axes[i]
        ax.imshow(frames_rgb[i])
        ax.axis("off")
        ax.set_title(str(i + 1))

    fig.suptitle(f"Action: {label_name}")
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    X_train, y_train = load_processed_data()
    sample, label_name = pick_random_sample(X_train, y_train)
    print(f"Sample shape: {sample.shape}")
    plot_sequence(sample, label_name)
```

A.5 src/make_dataset_action.py

```python
import cv2
import numpy as np
from pathlib import Path
from tqdm import tqdm

IMG_SIZE = 128
SEQ_LENGTH = 16
CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
MIN_CUSTOM_SAMPLES = 50
CUSTOM_STEP = 60


def load_video_frames(video_path):
    cap = cv2.VideoCapture(str(video_path))
    frames = []
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        frame = cv2.resize(frame, (IMG_SIZE, IMG_SIZE))
        frame = frame.astype("float32") / 255.0
        frames.append(frame)
    cap.release()
    return frames


def sample_sequence_from_video(frames):
    n = len(frames)
    if n == 0:
        return None
    if n >= SEQ_LENGTH:
        indices = np.linspace(0, n - 1, SEQ_LENGTH).astype(int)
    else:
        indices = [i % n for i in range(SEQ_LENGTH)]
    sequence = [frames[i] for i in indices]
    return np.stack(sequence, axis=0)


def slice_long_video_into_sequences(frames):
    sequences = []
    n = len(frames)
    if n == 0:
        return sequences
    if n < SEQ_LENGTH:
        seq = sample_sequence_from_video(frames)
        if seq is not None:
            sequences.append(seq)
    else:
        for start in range(0, n - SEQ_LENGTH + 1, CUSTOM_STEP):
            window = frames[start:start + SEQ_LENGTH]
            if len(window) == SEQ_LENGTH:
                sequences.append(np.stack(window, axis=0))
    if len(sequences) == 0:
        return sequences
    while len(sequences) < MIN_CUSTOM_SAMPLES:
        for seq in list(sequences):
```

```
            sequences.append(seq.copy())
            if len(sequences) >= MIN_CUSTOM_SAMPLES:
                break
    return sequences


def augment_horizontal_flip(sequence):
    return np.flip(sequence, axis=2)


def load_ucf101_sequences(ucf_root, class_to_idx):
    folder_to_label = {
        "WalkingWithDog": "Walking",
    }
    video_label_pairs = []
    for folder_name, label in folder_to_label.items():
        folder_path = ucf_root / folder_name
        if not folder_path.is_dir():
            continue
        for pattern in ("*.avi", "*.mp4", "*.mov", "*.mkv"):
            for video_path in folder_path.glob(pattern):
                video_label_pairs.append((video_path, label))
    sequences = []
    labels = []
    for video_path, label in tqdm(video_label_pairs, desc="Processing UCF101 videos"):
        frames = load_video_frames(video_path)
        seq = sample_sequence_from_video(frames)
        if seq is None:
            continue
        sequences.append(seq)
        labels.append(class_to_idx[label])
        flipped = augment_horizontal_flip(seq)
        sequences.append(flipped)
        labels.append(class_to_idx[label])
    return sequences, labels


def load_custom_sequences(custom_root, class_to_idx):
    sequences = []
    labels = []

    # Handle Waving from custom (can be a folder of clips or a single long video)
    waving_dir_candidates = [
        custom_root / "HandWaving",
        custom_root / "Waving",
    ]
    waving_file_candidates = [
        custom_root / "HandWaving.mov",
        custom_root / "HandWaving.mp4",
        custom_root / "Waving.mov",
        custom_root / "Waving.mp4",
    ]
    waving_path = None
    for candidate in waving_dir_candidates + waving_file_candidates:
        if candidate.exists():
            waving_path = candidate
```

```
            break

    if waving_path is not None:
        if waving_path.is_dir():
            video_paths = []
            for pattern in ("*.avi", "*.mp4", "*.mov", "*.mkv"):
                for video_path in waving_path.glob(pattern):
                    video_paths.append(video_path)
            for video_path in tqdm(video_paths, desc="Processing custom Waving (folder)"):
                frames = load_video_frames(video_path)
                seq = sample_sequence_from_video(frames)
                if seq is None:
                    continue
                sequences.append(seq)
                labels.append(class_to_idx["Waving"])
                flipped = augment_horizontal_flip(seq)
                sequences.append(flipped)
                labels.append(class_to_idx["Waving"])
        else:
            frames = load_video_frames(waving_path)
            seqs = slice_long_video_into_sequences(frames)
            for seq in tqdm(seqs, desc="Processing custom Waving (file)", leave=False):
                sequences.append(seq)
                labels.append(class_to_idx["Waving"])
                flipped = augment_horizontal_flip(seq)
                sequences.append(flipped)
                labels.append(class_to_idx["Waving"])

    label_to_files = {
        "Standing": ["Standing.mov", "Standing.mp4"],
        "Sitting": ["Sitting.mov", "Sitting.mp4"],
    }
    for label, filenames in label_to_files.items():
        video_path = None
        for name in filenames:
            candidate = custom_root / name
            if candidate.exists():
                video_path = candidate
                break
        if video_path is None:
            continue
        frames = load_video_frames(video_path)
        seqs = slice_long_video_into_sequences(frames)
        for seq in tqdm(seqs, desc=f"Processing custom {label}", leave=False):
            sequences.append(seq)
            labels.append(class_to_idx[label])
            flipped = augment_horizontal_flip(seq)
            sequences.append(flipped)
            labels.append(class_to_idx[label])
    return sequences, labels


def build_action_dataset():
    base_dir = Path(__file__).resolve().parents[1]
    raw_root = base_dir / "data" / "raw"
```

```
    ucf_root = raw_root / "ucf101"
    custom_root = raw_root / "custom"
    class_to_idx = {name: idx for idx, name in enumerate(CLASSES)}
    sequences = []
    labels = []
    ucf_sequences, ucf_labels = load_ucf101_sequences(ucf_root,
   class_to_idx)
    sequences.extend(ucf_sequences)
    labels.extend(ucf_labels)
    custom_sequences, custom_labels = load_custom_sequences(custom_root
   , class_to_idx)
    sequences.extend(custom_sequences)
    labels.extend(custom_labels)
    if len(sequences) == 0:
        raise RuntimeError("No sequences were generated. Check that
   video files exist under data/raw.")
    X = np.stack(sequences, axis=0)
    y_idx = np.array(labels, dtype=np.int64)
    num_classes = len(CLASSES)
    y = np.eye(num_classes, dtype=np.float32)[y_idx]
    indices = np.random.permutation(len(X))
    X = X[indices]
    y = y[indices]
    split_idx = int(0.8 * len(X))
    if split_idx == 0 or split_idx == len(X):
        raise RuntimeError("Not enough samples to create a non-empty
   train/test split.")
    X_train = X[:split_idx]
    y_train = y[:split_idx]
    X_test = X[split_idx:]
    y_test = y[split_idx:]
    return X_train, y_train, X_test, y_test


if __name__ == "__main__":
    X_train, y_train, X_test, y_test = build_action_dataset()
    base_dir = Path(__file__).resolve().parents[1]
    processed_dir = base_dir / "data" / "processed"
    processed_dir.mkdir(parents=True, exist_ok=True)
    np.save(processed_dir / "action_X_train.npy", X_train)
    np.save(processed_dir / "action_y_train.npy", y_train)
    np.save(processed_dir / "action_X_test.npy", X_test)
    np.save(processed_dir / "action_y_test.npy", y_test)
    print("Saved processed datasets to", processed_dir)
    print("action_X_train.npy shape:", X_train.shape)
    print("action_y_train.npy shape:", y_train.shape)
    print("action_X_test.npy shape:", X_test.shape)
    print("action_y_test.npy shape:", y_test.shape)
```

A.6 src/train_action_model.py

```python
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.optimizers import SGD, Adam, Adagrad

# Dataset / model configuration
IMG_SIZE = 128
SEQ_LENGTH = 16
NUM_CLASSES = 4
BATCH_SIZE = 4
EPOCHS = 15
OPTIMIZERS = ["SGD", "Adam", "Adagrad"]


def load_data():
    base_dir = Path(__file__).resolve().parents[1]
    processed_dir = base_dir / "data" / "processed"

    X_train = np.load(processed_dir / "action_X_train.npy")
    y_train = np.load(processed_dir / "action_y_train.npy")
    X_test = np.load(processed_dir / "action_X_test.npy")
    y_test = np.load(processed_dir / "action_y_test.npy")

    return X_train, y_train, X_test, y_test


def build_model(optimizer_name: str) -> models.Model:
    """Builds a CNN-LSTM action recognition model with MobileNetV2 backbone.

    Input: sequence of 16 frames with shape (16, 128, 128, 3).
    Spatial features: MobileNetV2 (ImageNet, include_top=False, pooling='avg').
    Temporal features: LSTM(64, dropout=0.3).
    Classifier: Dense(4, softmax).
    """

    inputs = layers.Input(shape=(SEQ_LENGTH, IMG_SIZE, IMG_SIZE, 3), name="frames")

    # MobileNetV2 expects inputs in [-1, 1]; our dataset is in [0, 1]
    x = layers.Lambda(lambda z: z * 2.0 - 1.0, name="scale_minus1_1")(inputs)

    base_cnn = MobileNetV2(
        include_top=False,
        weights="imagenet",
        pooling="avg",
        input_shape=(IMG_SIZE, IMG_SIZE, 3),
    )
    # Freeze backbone for faster training on M1
    base_cnn.trainable = False
```

```
    x = layers.TimeDistributed(base_cnn, name="frame_cnn")(x)
    x = layers.LSTM(64, dropout=0.3, name="lstm")(x)
    outputs = layers.Dense(NUM_CLASSES, activation="softmax", name="predictions")(x)

    model = models.Model(inputs=inputs, outputs=outputs, name=f"action_model_{optimizer_name.lower()}")

    opt_name = optimizer_name.lower()
    if opt_name == "sgd":
        optimizer = SGD(learning_rate=1e-3, momentum=0.9)
    elif opt_name == "adam":
        optimizer = Adam(learning_rate=1e-4)
    elif opt_name == "adagrad":
        optimizer = Adagrad(learning_rate=1e-3)
    else:
        raise ValueError(f"Unsupported optimizer name: {optimizer_name}")

    model.compile(
        optimizer=optimizer,
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )

    return model


def run_experiments():
    base_dir = Path(__file__).resolve().parents[1]
    models_dir = base_dir / "models"
    reports_dir = base_dir / "reports"
    models_dir.mkdir(parents=True, exist_ok=True)
    reports_dir.mkdir(parents=True, exist_ok=True)

    X_train, y_train, X_test, y_test = load_data()

    history_dict = {}
    results = []

    for opt_name in OPTIMIZERS:
        print("\n" + "=" * 60)
        print(f"Training with optimizer: {opt_name}")
        print("=" * 60)

        model = build_model(opt_name)
        model.summary()

        history = model.fit(
            X_train,
            y_train,
            validation_data=(X_test, y_test),
            epochs=EPOCHS,
            batch_size=BATCH_SIZE,
            verbose=1,
        )

```

```
        history_dict[opt_name] = history.history

        model_path = models_dir / f"action_model_{opt_name.lower()}.keras"
        model.save(model_path)
        print(f"Saved model to {model_path}")

        test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
        print(f"{opt_name} Test Accuracy: {test_acc:.4f}")

        results.append({
            "optimizer": opt_name,
            "test_accuracy": float(test_acc),
            "test_loss": float(test_loss),
        })

    # Visualization: accuracy & validation loss curves
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    for opt_name in OPTIMIZERS:
        hist = history_dict[opt_name]
        # Handle possible key naming differences
        if "accuracy" in hist:
            train_acc = hist["accuracy"]
        elif "categorical_accuracy" in hist:
            train_acc = hist["categorical_accuracy"]
        else:
            raise KeyError(f"No accuracy key found in history for {opt_name}: {hist.keys()}")

        val_loss = hist.get("val_loss")
        epochs_range = range(1, len(train_acc) + 1)

        ax1.plot(epochs_range, train_acc, label=opt_name)
        if val_loss is not None:
            ax2.plot(epochs_range, val_loss, label=opt_name)

    ax1.set_title("Training Accuracy vs Epochs")
    ax1.set_xlabel("Epoch")
    ax1.set_ylabel("Accuracy")
    ax1.legend()

    ax2.set_title("Validation Loss vs Epochs")
    ax2.set_xlabel("Epoch")
    ax2.set_ylabel("Loss")
    ax2.legend()

    fig.tight_layout()
    plot_path = reports_dir / "milestone1_optimizer_comparison.png"
    fig.savefig(plot_path)
    plt.close(fig)
    print(f"Saved optimizer comparison plot to {plot_path}")

    # Final summary table
    print("\nFinal Test Accuracy by Optimizer:")
    print("-" * 40)
    print(f"{'Optimizer':<12}{'Test Accuracy':>15}")
    print("-" * 40)
```

```
    for r in results:
        print(f"{r['optimizer']:<12}{r['test_accuracy']:>15.4f}")
    print("-" * 40)


if __name__ == "__main__":
    # Limit TensorFlow logging noise
    tf.get_logger().setLevel("ERROR")
    run_experiments()
```

A.7 src/make_dataset_emotion.py

```python
import sys
from pathlib import Path

import cv2
import numpy as np
from tqdm import tqdm

IMG_SIZE = 48
CLASSES = ["angry", "disgust", "fear", "happy", "sad", "surprise", "neutral"]
NUM_CLASSES = len(CLASSES)


def _one_hot(labels: np.ndarray) -> np.ndarray:
    if labels.min() < 0 or labels.max() >= NUM_CLASSES:
        raise ValueError(f"Labels out of range [0, {NUM_CLASSES - 1}]: min={labels.min()}, max={labels.max}")
    return np.eye(NUM_CLASSES, dtype=np.float32)[labels]


def _collect_image_label_pairs(split_dir: Path, class_to_idx: dict):
    pairs = []
    patterns = ("*.png", "*.jpg", "*.jpeg", "*.bmp", "*.tif", "*.tiff")

    for class_name in CLASSES:
        class_dir = split_dir / class_name
        if not class_dir.is_dir():
            raise FileNotFoundError(f"Missing class folder: {class_dir}")

        class_files = []
        for pattern in patterns:
            class_files.extend(class_dir.glob(pattern))
        class_files = sorted(class_files)

        label_idx = class_to_idx[class_name]
        for img_path in class_files:
            pairs.append((img_path, label_idx))

    return pairs


def _load_split(split_dir: Path, split_name: str, class_to_idx: dict):
    pairs = _collect_image_label_pairs(split_dir, class_to_idx)
    if len(pairs) == 0:
        raise RuntimeError(f"No images found in {split_dir}")

    X_list = []
    y_list = []
    skipped = 0

    for img_path, label_idx in tqdm(pairs, desc=f"Loading {split_name}"):
        img = cv2.imread(str(img_path), cv2.IMREAD_GRAYSCALE)
        if img is None:
            skipped += 1
```

```
            continue

        img = cv2.resize(img, (IMG_SIZE, IMG_SIZE), interpolation=cv2.INTER_AREA)
        img = img.astype(np.float32) / 255.0
        img = img.reshape(IMG_SIZE, IMG_SIZE, 1)

        X_list.append(img)
        y_list.append(label_idx)

    if len(X_list) == 0:
        raise RuntimeError(f"No valid images were loaded for split '{split_name}'.")

    if skipped > 0:
        print(f"    Skipped {skipped} unreadable image(s) in split '{split_name}'.")

    X = np.stack(X_list, axis=0)
    y = _one_hot(np.array(y_list, dtype=np.int64))
    return X, y


if __name__ == "__main__":
    base_dir = Path(__file__).resolve().parents[1]
    raw_root = base_dir / "data" / "raw" / "fer2013"
    train_dir = raw_root / "train"
    test_dir = raw_root / "test"

    if not train_dir.is_dir() or not test_dir.is_dir():
        print("    FER-2013 folder dataset not found.")
        print("Expected structure:")
        print(f"  {train_dir}/<class_name>/*.png")
        print(f"  {test_dir}/<class_name>/*.png")
        print("\nExpected class folders:")
        print("  " + ", ".join(CLASSES))
        sys.exit(1)

    class_to_idx = {name: idx for idx, name in enumerate(CLASSES)}

    X_train, y_train = _load_split(train_dir, "train", class_to_idx)
    X_test, y_test = _load_split(test_dir, "test", class_to_idx)

    processed_dir = base_dir / "data" / "processed"
    processed_dir.mkdir(parents=True, exist_ok=True)

    np.save(processed_dir / "emotion_X_train.npy", X_train)
    np.save(processed_dir / "emotion_y_train.npy", y_train)
    np.save(processed_dir / "emotion_X_test.npy", X_test)
    np.save(processed_dir / "emotion_y_test.npy", y_test)

    print("Saved processed emotion dataset to", processed_dir)
    print("emotion_X_train.npy shape:", X_train.shape)
    print("emotion_y_train.npy shape:", y_train.shape)
    print("emotion_X_test.npy shape:", X_test.shape)
    print("emotion_y_test.npy shape:", y_test.shape)
```

A.8 src/preprocessing.py

```python
"""
Preprocessing utilities for CSE480 Machine Vision Project
Handles image preprocessing for emotion recognition model.
"""

import cv2
import numpy as np


def process_face(image):
    """
    Processes a face image for emotion recognition model.

    Converts the image to grayscale and resizes it to 48x48 pixels
    as required by the Emotion Model specifications.

    Args:
        image: Input image (numpy array) in BGR, RGB, or grayscale format.
               Can be a file path (str) or numpy array.

    Returns:
        numpy.ndarray: Processed grayscale image of shape (48, 48)
    """
    # Load image if it's a file path
    if isinstance(image, str):
        img = cv2.imread(image)
        if img is None:
            raise ValueError(f"Could not load image from path: {image}")
    else:
        img = image.copy()

    # Convert to grayscale if needed
    if len(img.shape) == 3:
        # Image is BGR or RGB
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    else:
        # Image is already grayscale
        gray = img

    # Resize to 48x48
    resized = cv2.resize(gray, (48, 48), interpolation=cv2.INTER_AREA)

    return resized


if __name__ == "__main__":
    # Create a dummy blank image (e.g., 100x100 grayscale)
    dummy_image = np.zeros((100, 100), dtype=np.uint8)

    # Process the image
    processed = process_face(dummy_image)

    # Print the final shape
    print(f"Input shape: {dummy_image.shape}")
```

```
    print(f"Output shape: {processed.shape}")

    # Verify it's (48, 48)
    assert processed.shape == (48, 48), f"Expected shape (48, 48), got {processed.shape}"
    print("\n  Test passed! Output shape is (48, 48)")
```

A.9 src/realtime_pipeline.py

```python
import os
import time
from collections import deque
from pathlib import Path

import cv2
import numpy as np
import tensorflow as tf

os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

ACTION_CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
EMOTION_CLASSES = ["angry", "disgust", "fear", "happy", "sad", "surprise", "neutral"]

SEQ_LENGTH = 16
ACTION_IMG_SIZE = 128
EMOTION_IMG_SIZE = 48

ACTION_PRED_INTERVAL = 5


def _load_model(model_path: Path):
    if not model_path.is_file():
        raise FileNotFoundError(f"Model not found: {model_path}")

    try:
        return tf.keras.models.load_model(model_path, safe_mode=False, compile=False)
    except TypeError:
        return tf.keras.models.load_model(model_path, compile=False)


def _draw_text(img, text: str, org: tuple[int, int], font_scale: float = 0.7, color=(255, 255, 255)):
    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(img, text, org, font, font_scale, (0, 0, 0), 4, cv2.LINE_AA)
    cv2.putText(img, text, org, font, font_scale, color, 2, cv2.LINE_AA)


def _preprocess_emotion_face(gray_frame: np.ndarray, face_bbox: tuple[int, int, int, int]) -> np.ndarray:
    x, y, w, h = face_bbox
    roi = gray_frame[y:y + h, x:x + w]
    roi = cv2.resize(roi, (EMOTION_IMG_SIZE, EMOTION_IMG_SIZE), interpolation=cv2.INTER_AREA)
    roi = roi.astype(np.float32) / 255.0
    roi = roi.reshape(1, EMOTION_IMG_SIZE, EMOTION_IMG_SIZE, 1)
    return roi


def _preprocess_action_frame(bgr_frame: np.ndarray) -> np.ndarray:
    frame = cv2.resize(bgr_frame, (ACTION_IMG_SIZE, ACTION_IMG_SIZE), interpolation=cv2.INTER_AREA)
```

```
    frame = frame.astype(np.float32) / 255.0
    return frame


def _select_largest_face(faces: np.ndarray):
    if faces is None or len(faces) == 0:
        return None
    x, y, w, h = max(faces, key=lambda b: int(b[2]) * int(b[3]))
    return int(x), int(y), int(w), int(h)


def main():
    tf.get_logger().setLevel("ERROR")

    base_dir = Path(__file__).resolve().parents[1]
    models_dir = base_dir / "models"

    action_model_path = models_dir / "action_model_adam.keras"
    emotion_model_path = models_dir / "emotion_model_best.keras"

    action_model = _load_model(action_model_path)
    emotion_model = _load_model(emotion_model_path)

    face_cascade_path = base_dir / "models" / "
   haarcascade_frontalface_default.xml"
    if not face_cascade_path.exists():
        raise FileNotFoundError(f"HaarCascade not found: {
   face_cascade_path}")

    face_cascade = cv2.CascadeClassifier(str(face_cascade_path))
    if face_cascade.empty():
        raise RuntimeError(f"Failed to load HaarCascade: {
   face_cascade_path}")

    cap = cv2.VideoCapture(0)
    if not cap.isOpened() and hasattr(cv2, "CAP_AVFOUNDATION"):
        cap = cv2.VideoCapture(0, cv2.CAP_AVFOUNDATION)

    if not cap.isOpened():
        raise RuntimeError("Could not open webcam (VideoCapture(0)).")

    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    buffer = deque(maxlen=SEQ_LENGTH)
    frame_count = 0
    last_action_pred_frame = -ACTION_PRED_INTERVAL

    action_label = "Collecting..."
    action_conf = 0.0

    emotion_label = "No face"
    emotion_conf = 0.0

    last_time = time.time()
    fps_ema = 0.0

    window_name = "CSE480 Real-Time Pipeline"
```

```
104
105     while True:
106         ok, frame = cap.read()
107         if not ok or frame is None:
108             break
109
110         frame_count += 1
111
112         now = time.time()
113         dt = now - last_time
114         last_time = now
115         fps = (1.0 / dt) if dt > 0 else 0.0
116         fps_ema = fps if fps_ema == 0.0 else (0.9 * fps_ema + 0.1 * fps
    )
117
118         gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
119         faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
    minNeighbors=5, minSize=(40, 40))
120         best_face = _select_largest_face(faces)
121
122         if best_face is not None:
123             x, y, w, h = best_face
124             try:
125                 face_input = _preprocess_emotion_face(gray, best_face)
126                 e_probs = emotion_model.predict(face_input, verbose=0)
    [0]
127                 e_idx = int(np.argmax(e_probs))
128                 emotion_label = EMOTION_CLASSES[e_idx]
129                 emotion_conf = float(e_probs[e_idx])
130             except Exception:
131                 emotion_label = "Face err"
132                 emotion_conf = 0.0
133
134             cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0),
    2)
135             label_y = y - 10 if y - 10 > 10 else y + h + 20
136             _draw_text(frame, f"{emotion_label} {emotion_conf * 100:.1f
    }%", (x, label_y), font_scale=0.6, color=(0, 255, 0))
137         else:
138             emotion_label = "No face"
139             emotion_conf = 0.0
140
141         buffer.append(_preprocess_action_frame(frame))
142
143         if len(buffer) == SEQ_LENGTH and (frame_count -
    last_action_pred_frame) >= ACTION_PRED_INTERVAL:
144             seq = np.stack(buffer, axis=0).astype(np.float32)
145             seq = np.expand_dims(seq, axis=0)
146             a_probs = action_model.predict(seq, verbose=0)[0]
147             a_idx = int(np.argmax(a_probs))
148             action_label = ACTION_CLASSES[a_idx]
149             action_conf = float(a_probs[a_idx])
150             last_action_pred_frame = frame_count
151
152         _draw_text(frame, f"Status: {action_label} ({action_conf *
    100:.0f}%)", (10, 30), font_scale=0.8, color=(255, 255, 255))
153
154         fps_text = f"FPS: {fps_ema:.1f}"
```

```
155         x_fps = max(10, frame.shape[1] - 170)
156         _draw_text(frame, fps_text, (x_fps, 30), font_scale=0.7, color
    =(255, 255, 0))
157 
158         cv2.imshow(window_name, frame)
159         key = cv2.waitKey(1) & 0xFF
160         if key == ord("q") or key == 27:
161             break
162 
163     cap.release()
164     cv2.destroyAllWindows()
165 
166 
167 if __name__ == "__main__":
168     main()
```

A.10 src/train_emotion_model.py

```python
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

import tensorflow as tf
from tensorflow.keras import layers, models

IMG_SIZE = 48
NUM_CLASSES = 7
BATCH_SIZE = 64
EPOCHS = 15


def load_data():
    base_dir = Path(__file__).resolve().parents[1]
    processed_dir = base_dir / "data" / "processed"

    X_train = np.load(processed_dir / "emotion_X_train.npy")
    y_train = np.load(processed_dir / "emotion_y_train.npy")
    X_test = np.load(processed_dir / "emotion_X_test.npy")
    y_test = np.load(processed_dir / "emotion_y_test.npy")

    return X_train, y_train, X_test, y_test


def build_vgg_model() -> models.Model:
    inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 1), name="image")

    x = layers.Conv2D(32, (3, 3), padding="same", activation="relu")(
    inputs)
    x = layers.Conv2D(32, (3, 3), padding="same", activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=(2, 2))(x)
    x = layers.Dropout(0.25)(x)

    x = layers.Conv2D(64, (3, 3), padding="same", activation="relu")(x)
    x = layers.Conv2D(64, (3, 3), padding="same", activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=(2, 2))(x)
    x = layers.Dropout(0.25)(x)

    x = layers.Conv2D(128, (3, 3), padding="same", activation="relu")(x
    )
    x = layers.MaxPooling2D(pool_size=(2, 2))(x)
    x = layers.Dropout(0.25)(x)

    x = layers.Flatten()(x)
    x = layers.Dense(256, activation="relu")(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(NUM_CLASSES, activation="softmax", name="
    predictions")(x)

    model = models.Model(inputs=inputs, outputs=outputs, name="
    emotion_simple_vgg")
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
```

```
    )
    return model


def _residual_block(x, filters: int, stride: int = 1):
    shortcut = x

    x = layers.Conv2D(filters, (3, 3), strides=stride, padding="same", use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    x = layers.Conv2D(filters, (3, 3), strides=1, padding="same", use_bias=False)(x)
    x = layers.BatchNormalization()(x)

    if shortcut.shape[-1] != filters or stride != 1:
        shortcut = layers.Conv2D(filters, (1, 1), strides=stride, padding="same", use_bias=False)(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)

    x = layers.Add()([x, shortcut])
    x = layers.ReLU()(x)
    return x


def build_resnet_model() -> models.Model:
    inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 1), name="image")

    x = layers.Conv2D(32, (3, 3), padding="same", use_bias=False)(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    x = _residual_block(x, 32, stride=1)
    x = _residual_block(x, 32, stride=1)

    x = _residual_block(x, 64, stride=2)
    x = _residual_block(x, 64, stride=1)

    x = _residual_block(x, 128, stride=2)
    x = _residual_block(x, 128, stride=1)

    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(NUM_CLASSES, activation="softmax", name="predictions")(x)

    model = models.Model(inputs=inputs, outputs=outputs, name="emotion_mini_resnet")
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model


```

```
def _get_metric(history: dict, keys):
    for k in keys:
        if k in history:
            return history[k]
    raise KeyError(f"None of the keys {keys} found in history. Available keys: {list(history.keys())}")


def _train_one(name: str, build_fn, X_train, y_train, X_val, y_val, models_dir: Path):
    tf.keras.backend.clear_session()

    model = build_fn()
    print("\n" + "=" * 60)
    print(f"Training: {name}")
    print("=" * 60)
    model.summary()

    weights_path = models_dir / f"emotion_{name}_best.weights.h5"
    checkpoint = tf.keras.callbacks.ModelCheckpoint(
        filepath=weights_path,
        monitor="val_accuracy",
        mode="max",
        save_best_only=True,
        save_weights_only=True,
        verbose=1,
    )

    history = model.fit(
        X_train,
        y_train,
        validation_data=(X_val, y_val),
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        verbose=1,
        callbacks=[checkpoint],
    )

    model.load_weights(weights_path)
    val_acc = _get_metric(history.history, ["val_accuracy", "val_categorical_accuracy"])
    best_val_acc = float(np.max(val_acc))

    val_loss, val_acc_eval = model.evaluate(X_val, y_val, verbose=0)
    print(f"{name} best val_accuracy (history): {best_val_acc:.4f}")
    print(f"{name} val_accuracy (evaluate best weights): {val_acc_eval:.4f}")

    return model, history.history, best_val_acc


def run_experiments():
    base_dir = Path(__file__).resolve().parents[1]
    models_dir = base_dir / "models"
    reports_dir = base_dir / "reports"
    models_dir.mkdir(parents=True, exist_ok=True)
    reports_dir.mkdir(parents=True, exist_ok=True)

```

```
    X_train, y_train, X_test, y_test = load_data()
    print("Loaded emotion dataset:")
    print("X_train:", X_train.shape, "y_train:", y_train.shape)
    print("X_test:", X_test.shape, "y_test:", y_test.shape)

    vgg_model, vgg_hist, vgg_best = _train_one(
        name="vgg",
        build_fn=build_vgg_model,
        X_train=X_train,
        y_train=y_train,
        X_val=X_test,
        y_val=y_test,
        models_dir=models_dir,
    )

    resnet_model, resnet_hist, resnet_best = _train_one(
        name="mini_resnet",
        build_fn=build_resnet_model,
        X_train=X_train,
        y_train=y_train,
        X_val=X_test,
        y_val=y_test,
        models_dir=models_dir,
    )

    vgg_val_acc = _get_metric(vgg_hist, ["val_accuracy", "
val_categorical_accuracy"])
    resnet_val_acc = _get_metric(resnet_hist, ["val_accuracy", "
val_categorical_accuracy"])

    epochs_range = range(1, len(vgg_val_acc) + 1)
    fig, ax = plt.subplots(figsize=(9, 5))
    ax.plot(epochs_range, vgg_val_acc, label="Model A - Simple VGG")
    ax.plot(epochs_range, resnet_val_acc, label="Model B - Mini-ResNet"
)
    ax.set_title("FER-2013 Architecture Experiment: Validation Accuracy
")
    ax.set_xlabel("Epoch")
    ax.set_ylabel("Validation Accuracy")
    ax.legend()
    fig.tight_layout()

    plot_path = reports_dir / "milestone2_architecture_comparison.png"
    fig.savefig(plot_path)
    plt.close(fig)
    print(f"Saved architecture comparison plot to {plot_path}")

    if resnet_best > vgg_best:
        best_name = "mini_resnet"
        best_model = resnet_model
        best_score = resnet_best
    else:
        best_name = "vgg"
        best_model = vgg_model
        best_score = vgg_best

    best_path = models_dir / "emotion_model_best.keras"
    best_model.save(best_path)
```

```
213     print(f"Saved best model ({best_name}, best val_accuracy={
        best_score:.4f}) to {best_path}")
214
215
216 if __name__ == "__main__":
217     tf.get_logger().setLevel("ERROR")
218     run_experiments()
```