AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
MECHATRONICS ENGINEERING DEPARTMENT

CSE480: Machine Vision

Fall 2025

Milestone 1 Report

Action Recognition with CNN-LSTM

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 Problem Definition

Human Activity Recognition (HAR) is a core problem in modern computer vision with direct impact on surveillance, healthcare, human-computer interaction, and assistive technologies. In video surveillance, automatic understanding of activities such as walking, waving for attention, or remaining seated allows systems to filter large video streams for anomalous or safety-critical events. In healthcare and ambient assisted living, continuous monitoring of patients or elderly users can detect falls, prolonged inactivity, or unusual motion patterns without requiring manual observation.

Recognizing actions from raw video is challenging because an action is not defined by a single frame but by how the body configuration evolves over time. Static postures such as *Sitting* and *Standing* can often be inferred from a single image, but actions like *Walking* or *Waving* are inherently temporal.

Figure 1: Overview of the Human Activity Recognition (HAR) challenge, distinguishing static vs. dynamic actions.

A hybrid CNN-LSTM approach directly addresses this limitation by combining strong spatial feature extraction with temporal sequence modeling. The CNN branch focuses on *what* is present in each image, while the LSTM branch focuses on *how* these features

change over time.

2 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.

```
(mecha_env) elmala7@Elmala7s-MacBook-Air src % python inspect_data.py
Random sample index: 122
Label index: 0
Label name: Walking
Sample shape: (16, 128, 128, 3)
```

Figure 2: Sample frames

3 Methods & Algorithms

3.1 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.

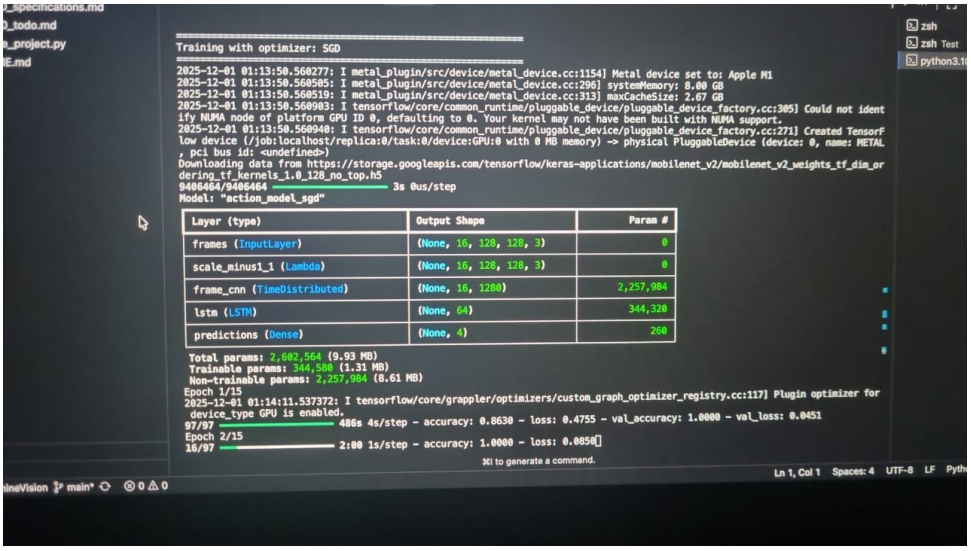

Figure 3: Training progress

3.2 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.

4 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.

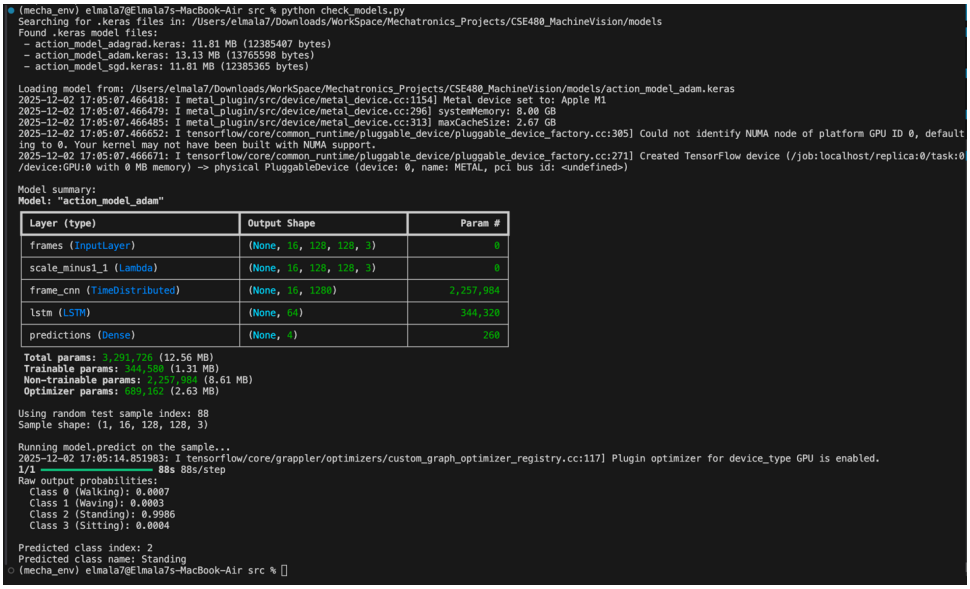

Figure 4: Model Adam results

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.

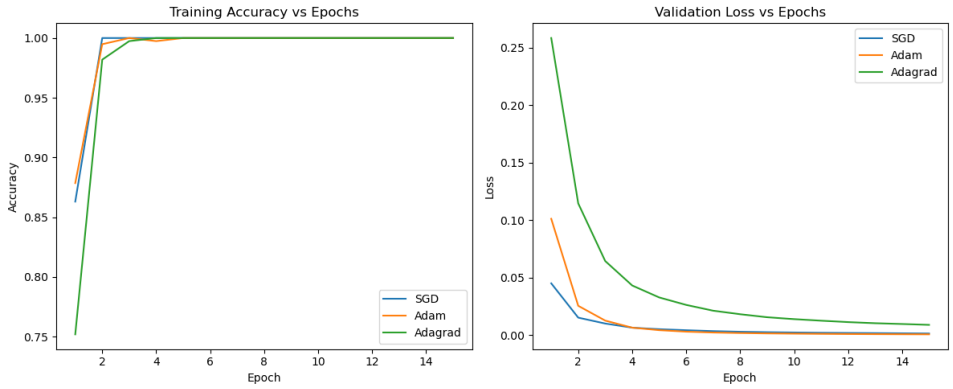

Figure 5: Validation Loss comparison showing the convergence speed of the different optimizers.

5 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()
```