



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	2
2 Data Cleaning & Preprocessing	3
3 Methods & Algorithms	3
3.1 Model Architecture	3
3.2 Training Setup	4
4 Experimental Results	4
5 Repository Link	5
A Appendix: Source Code	6
A.1 validation_notebook.ipynb	6
A.2 initialize_project.py	12
A.3 src/check_models.py	14
A.4 src/inspect_data.py	16
A.5 src/make_dataset_action.py	18
A.6 src/train_action_model.py	22



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.



```

specifications.md
todo.md
project.py
E.md

Training with optimizer: SGD
=====
2025-12-01 01:13:50.569277: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M1
2025-12-01 01:13:50.569569: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 8.00 GB
2025-12-01 01:13:50.569579: I metal_plugin/src/device/metal_device.cc:296] maxCacheSize: 2.07 GB
2025-12-01 01:13:50.569903: I tensorflow/core/common/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA support.
2025-12-01 01:13:50.569940: I tensorflow/core/common/pluggable_device_factory.cc:271] Created TensorFlow low device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) => physical PluggableDevice (device: 0, name: METAL , provider_name: <undefined>)
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_128_no_top.h5
9406464/9406464 [94%] 3s @us/step
Model: "action_model_sgd"
=====
Layer (type)           | Output Shape        | Param #
-----|-----|-----
frames (InputLayer)   | (None, 16, 128, 128, 3) | 0
scale_minus1_1 (Lambda)| (None, 16, 128, 128, 3) | 8
frame_cnn (TimeDistributed)| (None, 16, 1280) | 2,257,984
lstm (LSTM)           | (None, 64)          | 344,320
predictions (Dense)   | (None, 4)           | 256
=====
Total params: 2,692,564 (9.93 MB)
Trainable params: 344,568 (1.31 MB)
Non-trainable params: 2,257,984 (8.61 MB)
Epoch 1/15
2025-12-01 01:14:11.537372: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.
97/97 [=====] 486s 4s/step - accuracy: 0.8630 - loss: 0.4755 - val_accuracy: 1.0000 - val_loss: 0.0451
Epoch 2/15
16/97 [=====] 2:00 1s/step - accuracy: 1.0000 - loss: 0.0854
3s @us/step to generate a command.
Ln 1, Col 1  Spaces: 4  UTF-8  LF  Python

```

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.



```
(mecha_env) Elmala7@Elmala7s-MacBook-Air ~ % 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_gd.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/meta_device.cc:1154] Metal device set to: Apple M1
2025-12-02 17:05:07.466418: I metal_plugin/src/device/meta_device.cc:296] systemMemory: 8.00 GB
2025-12-02 17:05:07.466418: I metal_plugin/src/device/meta_device.cc:313] maxCacheSize: 2.67 GB
2025-12-02 17:05:07.466652: I tensorflow/core/common_runtime/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, defaulting 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_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, 128) | 2,257,984 |
| lstm (LSTM) | (None, 64) | 344,320 |
| predictions (Dense) | (None, 4) | 268 |
+-----+-----+-----+
Total params: 3,291,726 (12.56 MB)
Trainable params: 344,500 (1.31 MB)
Non-trainable params: 2,257,984 (8.61 MB)
Optimizer params: 889,162 (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 80s/step
Raw output probabilities:
Class 0 (Walking): 0.0007
Class 1 (Waving): 0.0003
Class 2 (Standing): 0.9996
Class 3 (Sitting): 0.0004

Predicted class index: 2
Predicted class name: Standing
(mecha_env) Elmala7@Elmala7s-MacBook-Air ~ %
```

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.

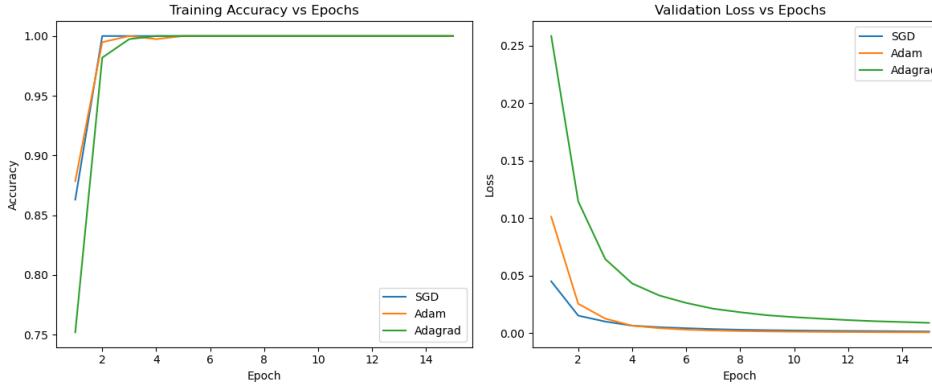


Figure 5: Validation Loss comparison showing the convergence speed of the different optimizers.

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

```
1 # Essential imports only
2 import os
3 from pathlib import Path
4 import warnings
5 warnings.filterwarnings('ignore')
6
7 import numpy as np
8 import pandas as pd
9 import matplotlib.pyplot as plt
10
11 import tensorflow as tf
12 from tensorflow.keras.models import load_model
13
14
15 # Suppress TensorFlow warnings
16 tf.get_logger().setLevel('ERROR')
17 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
18
19 print("    Imports successful")
20 # Project paths
21 base_dir = Path.cwd()
22 if base_dir.name == 'CSE480_MachineVision':
23     project_root = base_dir
24 else:
25     project_root = base_dir.parent
26
27 data_processed_dir = project_root / "data" / "processed"
28 models_dir = project_root / "models"
29 reports_dir = project_root / "reports"
30
31 # Create directories if needed
32 data_processed_dir.mkdir(parents=True, exist_ok=True)
33 models_dir.mkdir(parents=True, exist_ok=True)
34 reports_dir.mkdir(parents=True, exist_ok=True)
35
36 # Constants
37 CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
38 NUM_CLASSES = len(CLASSES)
39 SEQ_LENGTH = 16
40 IMG_SIZE = 128
41 OPTIMIZERS = ["SGD", "Adam", "Adagrad"]
42
43 print(f"Project root: {project_root}")
44
45 # =====
46 # SECTION: DATASET STATISTICS
47 # =====
48
49 # Basic statistics
50 print("=" * 60)
51 print("DATASET STATISTICS")
52 print("=" * 60)
```



```
53 print(f"Training Set: {X_train.shape[0]} samples, shape {X_train.shape}")
54 print(f"Test Set: {X_test.shape[0]} samples, shape {X_test.shape}")
55 print(f"Data type: {X_train.dtype}, Range: [{X_train.min():.3f}, {
      X_train.max():.3f}]")
56
57 # Class distribution
58 train_labels = np.argmax(y_train, axis=1)
59 test_labels = np.argmax(y_test, axis=1)
60
61 print("\nClass Distribution (Train/Test):")
62 for i, cls in enumerate(CLASSES):
63     train_count = (train_labels == i).sum()
64     test_count = (test_labels == i).sum()
65     print(f" {cls:12s}: {train_count:4d} / {test_count:4d}")
66
67 # =====
68 # SECTION: SAMPLE VISUALIZATION
69 # =====
70
71 # Show only 1 sample per class, 4 frames only (not all 16)
72 samples_per_class = 1
73 frames_to_show = 4 # Only show first 4 frames
74
75 fig, axes = plt.subplots(len(CLASSES), frames_to_show, figsize=(
    frames_to_show*2, len(CLASSES)*2))
76
77 for class_idx, class_name in enumerate(CLASSES):
78     class_mask = train_labels == class_idx
79     class_indices = np.where(class_mask)[0]
80
81     if len(class_indices) == 0:
82         continue
83
84     # Pick random sample
85     seq_idx = np.random.choice(class_indices)
86     sequence = X_train[seq_idx]
87     sequence_rgb = sequence[..., ::-1] # BGR to RGB
88
89     # Show only first 4 frames
90     for frame_idx in range(frames_to_show):
91         ax = axes[class_idx, frame_idx]
92         ax.imshow(sequence_rgb[frame_idx])
93         ax.axis('off')
94         if frame_idx == 0:
95             ax.set_title(f'{class_name}', fontsize=10, fontweight='bold')
96
97 plt.suptitle('Sample Sequences (4 frames shown)', fontsize=12,
    fontweight='bold')
98 plt.tight_layout()
99 plt.savefig(reports_dir / "sample_sequences_lightweight.png", dpi=100,
    bbox_inches='tight')
100 plt.show()
101 print(f"    Saved to {reports_dir / 'sample_sequences_lightweight.png'}")
102
103 # =====
```



```
104 # SECTION: DATA QUALITY CHECKS
105 # =====
106
107 # Essential checks only
108 print("=" * 60)
109 print("DATA QUALITY CHECKS")
110 print("=" * 60)
111
112 # Check shapes
113 expected_shape = (SEQ_LENGTH, IMG_SIZE, IMG_SIZE, 3)
114 train_shape_ok = X_train.shape[1:] == expected_shape
115 test_shape_ok = X_test.shape[1:] == expected_shape
116
117 print(f"Shape check: Train={train_shape_ok}, Test={test_shape_ok}")
118
119 # Check normalization
120 normalized = (X_train.min() >= 0 and X_train.max() <= 1)
121 print(f"Normalization: {normalized} (range: [{X_train.min():.3f}, {X_train.max():.3f}])")
122
123 # Check labels
124 train_onehot_ok = np.allclose(y_train.sum(axis=1), 1.0)
125 test_onehot_ok = np.allclose(y_test.sum(axis=1), 1.0)
126 print(f"One-hot encoding: Train={train_onehot_ok}, Test={test_onehot_ok}")
127
128 # Check for NaN/Inf
129 no_nan = not (np.isnan(X_train).any() or np.isnan(X_test).any())
130 no_inf = not (np.isinf(X_train).any() or np.isinf(X_test).any())
131 print(f"Data integrity: No NaN={no_nan}, No Inf={no_inf}")
132
133 print("\n    Data quality checks completed")
134
135 # =====
136 # SECTION: MODEL LOADING
137 # =====
138
139 # Load models
140 models_dict = {}
141
142 print("Loading trained models...\n")
143 for opt_name in OPTIMIZERS:
144     model_path = models_dir / f"action_model_{opt_name.lower()}.keras"
145
146     if not model_path.exists():
147         print(f"    {opt_name} model not found: {model_path}")
148         continue
149
150     try:
151         model = load_model(model_path, safe_mode=False)
152         models_dict[opt_name] = model
153         file_size_mb = model_path.stat().st_size / (1024 * 1024)
154         print(f"    {opt_name}: Loaded ({file_size_mb:.2f} MB)")
155     except Exception as e:
156         print(f"    Error loading {opt_name}: {e}")
157
158 if len(models_dict) == 0:
```



```
159     raise FileNotFoundError("No models loaded. Train models first:  
160     python src/train_action_model.py")  
161  
162 print(f"\n    Loaded {len(models_dict)} model(s)")  
163 # =====  
164 # SECTION: MODEL ARCHITECTURES  
165 # =====  
166  
167 # Full model summaries  
168 print("=" * 60)  
169 print("MODEL ARCHITECTURES - FULL DETAILS")  
170 print("=" * 60)  
171  
172 for opt_name, model in models_dict.items():  
173     print(f"\n{opt_name} Model - Complete Architecture")  
174     print(f"{'=' * 60}")  
175  
176     # Show full model summary  
177     model.summary()  
178  
179 # =====  
180 # SECTION: VERIFICATION  
181 # =====  
182  
183  
184 # Quick verification  
185 print("=" * 60)  
186 print("MODEL VERIFICATION")  
187 print("=" * 60)  
188  
189 # Select random sample from test set  
190 random_idx = np.random.randint(0, len(X_test))  
191 test_sample = X_test[random_idx:random_idx+1] # Keep batch dimension  
192 true_label_idx = np.argmax(y_test[random_idx])  
193 true_label = CLASSES[true_label_idx]  
194 print(f"Random sample index: {random_idx}, True label: {true_label}")  
195 print(f"Test sample shape: {test_sample.shape}")  
196  
197 for opt_name, model in models_dict.items():  
198     output = model.predict(test_sample, verbose=0)  
199     prob_sum = output.sum()  
200     pred_idx = np.argmax(output[0])  
201  
202     print(f"\n{opt_name}:")  
203     print(f"    Output shape: {output.shape}      ")  
204     print(f"    Probability sum: {prob_sum:.4f} {'' if np.isclose(  
205         prob_sum, 1.0) else ''}')  
206     print(f"    Predicted: {CLASSES[pred_idx]}")  
207  
208 print("\n    Model verification completed")  
209 # =====  
210 # SECTION: TRAINING RESULTS REFERENCE  
211 # =====  
212  
213 # Reference to training results  
214 print("=" * 60)
```



```
215 print("OPTIMIZER COMPARISON - TRAINING RESULTS")
216 print("=" * 60)
217 print("\nPerformance evaluation results are available from the training
      script.")
218 print("The optimizer comparison plot was generated during model
      training.\n")
219
220 # Check if training plot exists
221 training_plot_path = reports_dir / "milestone1_optimizer_comparison.png"
222
223 if training_plot_path.exists():
224     print(f"    Training results plot found: {training_plot_path.name}")
225
226     print("\nDisplaying optimizer comparison from training:")
227
228     # Display the plot
229     img = plt.imread(training_plot_path)
230     plt.figure(figsize=(12, 6))
231     plt.imshow(img)
232     plt.axis('off')
233     plt.title('Optimizer Comparison (from training)', fontsize=14,
234               fontweight='bold')
235     plt.tight_layout()
236     plt.show()
237     print(f"\n    Displayed training comparison plot")
238 else:
239     print(f"    Training plot not found: {training_plot_path}")
240     print("    Expected location: reports/
241           milestone1_optimizer_comparison.png")
242     print("    This plot is generated by: python src/train_action_model.
243         py")
244
245 print("\nNote: Detailed performance metrics (accuracy, loss) are
      available")
246 print("      from the training script outputs and saved model
      evaluations.")
247
248 # =====
249 # SECTION: EVALUATION
250 # =====
251
252 # Fast test set evaluation using model.evaluate() only
253 import time
254
255 print("=" * 60)
256 print("SECTION 4.1: TEST SET EVALUATION (FAST VERSION)")
257 print("=" * 60)
258
259 # Initialize results dictionary
260 evaluation_results = {}
261
262 print(f"\nTest set size: {X_test.shape[0]} samples")
263 print(f"Test set shape: {X_test.shape}\n")
264
265 # Evaluate each model
266 for opt_name in OPTIMIZERS:
267     if opt_name not in models_dict:
268         print(f"    Skipping {opt_name}: model not loaded")
```



```
264     continue
265
266     model = models_dict[opt_name]
267
268     print(f"Evaluating {opt_name}...")
269     start_time = time.time()
270
271     # FAST: Use model.evaluate() only (no predictions)
272     test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose
273     =0)
274
275     elapsed = time.time() - start_time
276
277     # Store results
278     evaluation_results[opt_name] = {
279         "test_loss": float(test_loss),
280         "test_accuracy": float(test_accuracy),
281     }
282
283     print(f"      {opt_name} - Accuracy: {test_accuracy:.4f}, Loss: {test_loss:.4f} ({elapsed:.2f}s)\n")
284
285 print("==" * 60)
286 print("TEST SET RESULTS - COMPARISON TABLE")
287 print("==" * 60 + "\n")
288
289 # Create comparison DataFrame
290 results_df = pd.DataFrame(evaluation_results).T
291 results_df.columns = ["Test Accuracy", "Test Loss"]
292 print(results_df.to_string())
293
294 print("\n      Fast evaluation completed")
```



A.2 initialize_project.py

```
1 #!/usr/bin/env python3
2 """
3 Project Initialization Script for CSE480 Machine Vision Project
4 Creates the required directory structure for the Action & Emotion
5 Recognition project.
6 """
7
8 import os
9 from pathlib import Path
10
11 def create_directory_structure(base_path="."):
12     """
13     Creates the required directory structure for the CSE480 project.
14
15     Args:
16         base_path (str): Base path where directories will be created (default: current directory)
17     """
18     directories = [
19         "data/raw",           # For original datasets (FER-2013, UCF
-101)
20         "data/processed",    # For resized images and processed data
21         "src",               # For source code
22         "models",             # To save trained .keras files
23         "notebooks",          # For experiments
24         "reports",            # For milestone reports
25     ]
26
27     base = Path(base_path)
28     created_dirs = []
29     existing_dirs = []
30
31     for directory in directories:
32         dir_path = base / directory
33         if dir_path.exists():
34             existing_dirs.append(str(dir_path))
35             print(f"    Directory already exists: {dir_path}")
36         else:
37             dir_path.mkdir(parents=True, exist_ok=True)
38             created_dirs.append(str(dir_path))
39             print(f"    Created directory: {dir_path}")
40
41     print("\n" + "="*60)
42     print("Project structure initialization complete!")
43     print("="*60)
44     if created_dirs:
45         print(f"\nCreated {len(created_dirs)} new directory(ies):")
46         for d in created_dirs:
47             print(f"    - {d}")
48     if existing_dirs:
49         print(f"\nFound {len(existing_dirs)} existing directory(ies):")
50         for d in existing_dirs:
51             print(f"    - {d}")
52     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

```
1 import numpy as np
2 from pathlib import Path
3
4 import tensorflow as tf
5
6 CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
7
8
9 def list_keras_files(models_dir: Path):
10     print(f"Searching for .keras files in: {models_dir}")
11     if not models_dir.exists():
12         print("models directory does not exist.")
13         return []
14
15     keras_files = sorted(models_dir.glob("*.keras"))
16     if not keras_files:
17         print("No .keras files found.")
18         return []
19
20     print("Found .keras model files:")
21     for path in keras_files:
22         size_bytes = path.stat().st_size
23         size_mb = size_bytes / (1024 * 1024)
24         print(f" - {path.name}: {size_mb:.2f} MB ({size_bytes} bytes)")
25
26     return keras_files
27
28
29 def load_any_model(models_dir: Path):
30     adam_path = models_dir / "action_model_adam.keras"
31     adagrad_path = models_dir / "action_model_adagrad.keras"
32
33     model_path = None
34     if adam_path.exists():
35         model_path = adam_path
36     elif adagrad_path.exists():
37         model_path = adagrad_path
38     else:
39         # Fallback: pick the first .keras file if available
40         keras_files = sorted(models_dir.glob("*.keras"))
41         if keras_files:
42             model_path = keras_files[0]
43
44     if model_path is None:
45         raise FileNotFoundError("No suitable .keras model file found in models/ directory.")
46
47     print(f"\nLoading model from: {model_path}")
48     model = tf.keras.models.load_model(model_path, safe_mode=False)
49     return model, model_path
50
51
52 def load_random_test_sample(processed_dir: Path):
53     x_test_path = processed_dir / "action_X_test.npy"
54     if not x_test_path.exists():
55         raise FileNotFoundError(f"Test data file not found: {x_test_path}
```



```
    x_test_path})")
56
57     X_test = np.load(x_test_path)
58     if len(X_test) == 0:
59         raise ValueError("Test set is empty; cannot pick a random
sample.")
60
61     idx = np.random.randint(0, len(X_test))
62     sample = X_test[idx: idx + 1] # keep batch dimension for model.
63     predict
64     print(f"\nUsing random test sample index: {idx}")
65     print(f"Sample shape: {sample.shape}")
66     return sample, idx
67
68 def main():
69     base_dir = Path(__file__).resolve().parents[1]
70     models_dir = base_dir / "models"
71     processed_dir = base_dir / "data" / "processed"
72
73     # 1) List all .keras files and sizes
74     list_keras_files(models_dir)
75
76     # 2) Load preferred model (Adam, then Adagrad)
77     model, model_path = load_any_model(models_dir)
78
79     print("\nModel summary:")
80     model.summary()
81
82     # 3) Load one random test sample
83     sample, idx = load_random_test_sample(processed_dir)
84
85     # 4) Run inference
86     print("\nRunning model.predict on the sample...")
87     probs = model.predict(sample)
88
89     if probs.ndim != 2 or probs.shape[1] != len(CLASSES):
90         raise ValueError(
91             f"Unexpected prediction shape {probs.shape}; expected (1, {len(CLASSES)}).")
92         )
93
94     probs_row = probs[0]
95     print("Raw output probabilities:")
96     for i, (cls, p) in enumerate(zip(CLASSES, probs_row)):
97         print(f"  Class {i} ({cls}): {p:.4f}")
98
99     pred_idx = int(np.argmax(probs_row))
100    pred_class = CLASSES[pred_idx]
101
102    print(f"\nPredicted class index: {pred_idx}")
103    print(f"Predicted class name: {pred_class}")
104
105
106 if __name__ == "__main__":
107     # Reduce TensorFlow logging noise
108     tf.get_logger().setLevel("ERROR")
109     main()
```



A.4 src/inspect_data.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from pathlib import Path
4
5 # Class order must match the one used in make_dataset_action.py
6 CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
7
8
9 def load_processed_data():
10     base_dir = Path(__file__).resolve().parents[1]
11     processed_dir = base_dir / "data" / "processed"
12
13     X_train_path = processed_dir / "action_X_train.npy"
14     y_train_path = processed_dir / "action_y_train.npy"
15
16     if not X_train_path.exists() or not y_train_path.exists():
17         raise FileNotFoundError(
18             f"Processed dataset not found. Expected files: {X_train_path.name}, {y_train_path.name} in {processed_dir}"
19         )
20
21     X_train = np.load(X_train_path)
22     y_train = np.load(y_train_path)
23
24     return X_train, y_train
25
26
27
28 def pick_random_sample(X_train, y_train):
29     if len(X_train) == 0:
30         raise ValueError("Empty training set: no samples to inspect.")
31
32     idx = np.random.randint(0, len(X_train))
33     sample = X_train[idx] # shape: (16, 128, 128, 3)
34     label_one_hot = y_train[idx]
35
36     if label_one_hot.ndim == 0 or label_one_hot.shape[0] != len(CLASSES):
37         raise ValueError(
38             f"Unexpected label shape {label_one_hot.shape}; "
39             f"expected one-hot of length {len(CLASSES)}."
40         )
41
42     class_idx = int(np.argmax(label_one_hot))
43     label_name = CLASSES[class_idx]
44
45     print(f"Random sample index: {idx}")
46     print(f"Label index: {class_idx}")
47     print(f"Label name: {label_name}")
48
49     return sample, label_name
50
51
52 def plot_sequence(sequence, label_name):
53     if sequence.shape[0] != 16:
54         raise ValueError(f"Expected sequence length 16, got {sequence}.
```



```
shape[0]}")  
55  
56     # sequence shape: (16, H, W, 3), in BGR order from OpenCV  
57     # Convert each frame from BGR -> RGB for correct display with  
58     # matplotlib  
59     frames_rgb = sequence[..., ::-1]  
60  
61     fig, axes = plt.subplots(4, 4, figsize=(8, 8))  
62     axes = axes.flatten()  
63  
64     for i in range(16):  
65         ax = axes[i]  
66         ax.imshow(frames_rgb[i])  
67         ax.axis("off")  
68         ax.set_title(str(i + 1))  
69  
70     fig.suptitle(f"Action: {label_name}")  
71     plt.tight_layout()  
72     plt.show()  
73  
74 if __name__ == "__main__":  
75     X_train, y_train = load_processed_data()  
76     sample, label_name = pick_random_sample(X_train, y_train)  
77     print(f"Sample shape: {sample.shape}")  
78     plot_sequence(sample, label_name)
```



A.5 src/make_dataset_action.py

```
1 import cv2
2 import numpy as np
3 from pathlib import Path
4 from tqdm import tqdm
5
6 IMG_SIZE = 128
7 SEQ_LENGTH = 16
8 CLASSES = ["Walking", "Waving", "Standing", "Sitting"]
9 MIN_CUSTOM_SAMPLES = 50
10 CUSTOM_STEP = 60
11
12
13 def load_video_frames(video_path):
14     cap = cv2.VideoCapture(str(video_path))
15     frames = []
16     while True:
17         ret, frame = cap.read()
18         if not ret:
19             break
20         frame = cv2.resize(frame, (IMG_SIZE, IMG_SIZE))
21         frame = frame.astype("float32") / 255.0
22         frames.append(frame)
23     cap.release()
24     return frames
25
26
27 def sample_sequence_from_video(frames):
28     n = len(frames)
29     if n == 0:
30         return None
31     if n >= SEQ_LENGTH:
32         indices = np.linspace(0, n - 1, SEQ_LENGTH).astype(int)
33     else:
34         indices = [i % n for i in range(SEQ_LENGTH)]
35     sequence = [frames[i] for i in indices]
36     return np.stack(sequence, axis=0)
37
38
39 def slice_long_video_into_sequences(frames):
40     sequences = []
41     n = len(frames)
42     if n == 0:
43         return sequences
44     if n < SEQ_LENGTH:
45         seq = sample_sequence_from_video(frames)
46         if seq is not None:
47             sequences.append(seq)
48     else:
49         for start in range(0, n - SEQ_LENGTH + 1, CUSTOM_STEP):
50             window = frames[start:start + SEQ_LENGTH]
51             if len(window) == SEQ_LENGTH:
52                 sequences.append(np.stack(window, axis=0))
53     if len(sequences) == 0:
54         return sequences
55     while len(sequences) < MIN_CUSTOM_SAMPLES:
56         for seq in list(sequences):
```



```
57         sequences.append(seq.copy())
58     if len(sequences) >= MIN_CUSTOM_SAMPLES:
59         break
60 return sequences
61
62
63 def augment_horizontal_flip(sequence):
64     return np.flip(sequence, axis=2)
65
66
67 def load_ucf101_sequences(ucf_root, class_to_idx):
68     folder_to_label = {
69         "WalkingWithDog": "Walking",
70     }
71     video_label_pairs = []
72     for folder_name, label in folder_to_label.items():
73         folder_path = ucf_root / folder_name
74         if not folder_path.is_dir():
75             continue
76         for pattern in ("*.avi", "*.mp4", "*.mov", "*.mkv"):
77             for video_path in folder_path.glob(pattern):
78                 video_label_pairs.append((video_path, label))
79     sequences = []
80     labels = []
81     for video_path, label in tqdm(video_label_pairs, desc="Processing
UCF101 videos"):
82         frames = load_video_frames(video_path)
83         seq = sample_sequence_from_video(frames)
84         if seq is None:
85             continue
86         sequences.append(seq)
87         labels.append(class_to_idx[label])
88         flipped = augment_horizontal_flip(seq)
89         sequences.append(flipped)
90         labels.append(class_to_idx[label])
91     return sequences, labels
92
93
94 def load_custom_sequences(custom_root, class_to_idx):
95     sequences = []
96     labels = []
97
98     # Handle Waving from custom (can be a folder of clips or a single
long video)
99     waving_dir_candidates = [
100         custom_root / "HandWaving",
101         custom_root / "Waving",
102     ]
103     waving_file_candidates = [
104         custom_root / "HandWaving.mov",
105         custom_root / "HandWaving.mp4",
106         custom_root / "Waving.mov",
107         custom_root / "Waving.mp4",
108     ]
109     waving_path = None
110     for candidate in waving_dir_candidates + waving_file_candidates:
111         if candidate.exists():
112             waving_path = candidate
```



```
113         break
114
115     if waving_path is not None:
116         if waving_path.is_dir():
117             video_paths = []
118             for pattern in ("*.avi", "*.mp4", "*.mov", "*.mkv"):
119                 for video_path in waving_path.glob(pattern):
120                     video_paths.append(video_path)
121             for video_path in tqdm(video_paths, desc="Processing custom
Waving (folder)"):
122                 frames = load_video_frames(video_path)
123                 seq = sample_sequence_from_video(frames)
124                 if seq is None:
125                     continue
126                 sequences.append(seq)
127                 labels.append(class_to_idx["Waving"])
128                 flipped = augment_horizontal_flip(seq)
129                 sequences.append(flipped)
130                 labels.append(class_to_idx["Waving"])
131             else:
132                 frames = load_video_frames(waving_path)
133                 seqs = slice_long_video_into_sequences(frames)
134                 for seq in tqdm(seqs, desc="Processing custom Waving (file)
", leave=False):
135                     sequences.append(seq)
136                     labels.append(class_to_idx["Waving"])
137                     flipped = augment_horizontal_flip(seq)
138                     sequences.append(flipped)
139                     labels.append(class_to_idx["Waving"])
140
141     label_to_files = {
142         "Standing": ["Standing.mov", "Standing.mp4"],
143         "Sitting": ["Sitting.mov", "Sitting.mp4"],
144     }
145     for label, filenames in label_to_files.items():
146         video_path = None
147         for name in filenames:
148             candidate = custom_root / name
149             if candidate.exists():
150                 video_path = candidate
151                 break
152         if video_path is None:
153             continue
154         frames = load_video_frames(video_path)
155         seqs = slice_long_video_into_sequences(frames)
156         for seq in tqdm(seqs, desc=f"Processing custom {label}", leave=
False):
157             sequences.append(seq)
158             labels.append(class_to_idx[label])
159             flipped = augment_horizontal_flip(seq)
160             sequences.append(flipped)
161             labels.append(class_to_idx[label])
162     return sequences, labels
163
164
165 def build_action_dataset():
166     base_dir = Path(__file__).resolve().parents[1]
167     raw_root = base_dir / "data" / "raw"
```



```
168 ucf_root = raw_root / "ucf101"
169 custom_root = raw_root / "custom"
170 class_to_idx = {name: idx for idx, name in enumerate(CLASSES)}
171 sequences = []
172 labels = []
173 ucf_sequences, ucf_labels = load_ucf101_sequences(ucf_root,
174 class_to_idx)
175 sequences.extend(ucf_sequences)
176 labels.extend(ucf_labels)
177 custom_sequences, custom_labels = load_custom_sequences(custom_root,
178 , class_to_idx)
179 sequences.extend(custom_sequences)
180 labels.extend(custom_labels)
181 if len(sequences) == 0:
182     raise RuntimeError("No sequences were generated. Check that
183 video files exist under data/raw.")
184 X = np.stack(sequences, axis=0)
185 y_idx = np.array(labels, dtype=np.int64)
186 num_classes = len(CLASSES)
187 y = np.eye(num_classes, dtype=np.float32)[y_idx]
188 indices = np.random.permutation(len(X))
189 X = X[indices]
190 y = y[indices]
191 split_idx = int(0.8 * len(X))
192 if split_idx == 0 or split_idx == len(X):
193     raise RuntimeError("Not enough samples to create a non-empty
194 train/test split.")
195 X_train = X[:split_idx]
196 y_train = y[:split_idx]
197 X_test = X[split_idx:]
198 y_test = y[split_idx:]
199 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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from pathlib import Path
4
5 import tensorflow as tf
6 from tensorflow.keras import layers, models
7 from tensorflow.keras.applications import MobileNetV2
8 from tensorflow.keras.optimizers import SGD, Adam, Adagrad
9
10 # Dataset / model configuration
11 IMG_SIZE = 128
12 SEQ_LENGTH = 16
13 NUM_CLASSES = 4
14 BATCH_SIZE = 4
15 EPOCHS = 15
16 OPTIMIZERS = ["SGD", "Adam", "Adagrad"]
17
18
19 def load_data():
20     base_dir = Path(__file__).resolve().parents[1]
21     processed_dir = base_dir / "data" / "processed"
22
23     X_train = np.load(processed_dir / "action_X_train.npy")
24     y_train = np.load(processed_dir / "action_y_train.npy")
25     X_test = np.load(processed_dir / "action_X_test.npy")
26     y_test = np.load(processed_dir / "action_y_test.npy")
27
28     return X_train, y_train, X_test, y_test
29
30
31 def build_model(optimizer_name: str) -> models.Model:
32     """Builds a CNN-LSTM action recognition model with MobileNetV2
33     backbone.
34
35     Input: sequence of 16 frames with shape (16, 128, 128, 3).
36     Spatial features: MobileNetV2 (ImageNet, include_top=False, pooling
37     ='avg').
38     Temporal features: LSTM(64, dropout=0.3).
39     Classifier: Dense(4, softmax).
40     """
41
42     inputs = layers.Input(shape=(SEQ_LENGTH, IMG_SIZE, IMG_SIZE, 3),
43                           name="frames")
44
45     # MobileNetV2 expects inputs in [-1, 1]; our dataset is in [0, 1]
46     x = layers.Lambda(lambda z: z * 2.0 - 1.0, name="scale_minus1_1")(inputs)
47
48     base_cnn = MobileNetV2(
49         include_top=False,
50         weights="imagenet",
51         pooling="avg",
52         input_shape=(IMG_SIZE, IMG_SIZE, 3),
53     )
54     # Freeze backbone for faster training on M1
55     base_cnn.trainable = False
```



```
53
54     x = layers.TimeDistributed(base_cnn, name="frame_cnn")(x)
55     x = layers.LSTM(64, dropout=0.3, name="lstm")(x)
56     outputs = layers.Dense(NUM_CLASSES, activation="softmax", name="predictions")(x)
57
58     model = models.Model(inputs=inputs, outputs=outputs, name=f"action_model_{optimizer_name.lower()}")
59
60     opt_name = optimizer_name.lower()
61     if opt_name == "sgd":
62         optimizer = SGD(learning_rate=1e-3, momentum=0.9)
63     elif opt_name == "adam":
64         optimizer = Adam(learning_rate=1e-4)
65     elif opt_name == "adagrad":
66         optimizer = Adagrad(learning_rate=1e-3)
67     else:
68         raise ValueError(f"Unsupported optimizer name: {optimizer_name}")
69
70     model.compile(
71         optimizer=optimizer,
72         loss="categorical_crossentropy",
73         metrics=["accuracy"],
74     )
75
76     return model
77
78
79 def run_experiments():
80     base_dir = Path(__file__).resolve().parents[1]
81     models_dir = base_dir / "models"
82     reports_dir = base_dir / "reports"
83     models_dir.mkdir(parents=True, exist_ok=True)
84     reports_dir.mkdir(parents=True, exist_ok=True)
85
86     X_train, y_train, X_test, y_test = load_data()
87
88     history_dict = {}
89     results = []
90
91     for opt_name in OPTIMIZERS:
92         print("\n" + "=" * 60)
93         print(f"Training with optimizer: {opt_name}")
94         print("=" * 60)
95
96         model = build_model(opt_name)
97         model.summary()
98
99         history = model.fit(
100             X_train,
101             y_train,
102             validation_data=(X_test, y_test),
103             epochs=EPOCHS,
104             batch_size=BATCH_SIZE,
105             verbose=1,
106         )
107
```



```
108     history_dict[opt_name] = history.history
109
110     model_path = models_dir / f"action_model_{opt_name.lower()}.keras"
111     model.save(model_path)
112     print(f"Saved model to {model_path}")
113
114     test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
115     print(f"{opt_name} Test Accuracy: {test_acc:.4f}")
116
117     results.append({
118         "optimizer": opt_name,
119         "test_accuracy": float(test_acc),
120         "test_loss": float(test_loss),
121     })
122
123 # Visualization: accuracy & validation loss curves
124 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
125
126 for opt_name in OPTIMIZERS:
127     hist = history_dict[opt_name]
128     # Handle possible key naming differences
129     if "accuracy" in hist:
130         train_acc = hist["accuracy"]
131     elif "categorical_accuracy" in hist:
132         train_acc = hist["categorical_accuracy"]
133     else:
134         raise KeyError(f"No accuracy key found in history for {opt_name}: {hist.keys()}")
135
136     val_loss = hist.get("val_loss")
137     epochs_range = range(1, len(train_acc) + 1)
138
139     ax1.plot(epochs_range, train_acc, label=opt_name)
140     if val_loss is not None:
141         ax2.plot(epochs_range, val_loss, label=opt_name)
142
143     ax1.set_title("Training Accuracy vs Epochs")
144     ax1.set_xlabel("Epoch")
145     ax1.set_ylabel("Accuracy")
146     ax1.legend()
147
148     ax2.set_title("Validation Loss vs Epochs")
149     ax2.set_xlabel("Epoch")
150     ax2.set_ylabel("Loss")
151     ax2.legend()
152
153 fig.tight_layout()
154 plot_path = reports_dir / "milestone1_optimizer_comparison.png"
155 fig.savefig(plot_path)
156 plt.close(fig)
157 print(f"Saved optimizer comparison plot to {plot_path}")
158
159 # Final summary table
160 print("\nFinal Test Accuracy by Optimizer:")
161 print("-" * 40)
162 print(f"{'Optimizer':<12}{'Test Accuracy':>15}")
163 print("-" * 40)
```



```
164     for r in results:  
165         print(f"{r['optimizer']:<12}{r['test_accuracy']:>15.4f}")  
166     print("-" * 40)  
167  
168  
169 if __name__ == "__main__":  
170     # Limit TensorFlow logging noise  
171     tf.get_logger().setLevel("ERROR")  
172     run_experiments()
```