Final Project Report: Emotion Recognition in Videos Through Deep Neural Network Models

Koushik Roy

May 7, 2025

Abstract

This project investigates visual-only emotion recognition in videos using deep neural network architectures. Leveraging the RAVDESS dataset, we implement several models—including early fusion, late fusion, 3D convolutional, and CNN–RNN hybrids—compare their performance, and analyze the impact of temporal modeling, data augmentation, and hyperparameter choices.

Contents

1 Introduction

Emotion recognition from human facial expressions is a fundamental capability for any robotic or intelligent system that aspires to interact naturally and effectively with people. Non–verbal cues, and in particular facial expressions, convey a majority of our emotional intent: Mehrabian and Ferris showed that approximately 55% of emotional information is communicated visually, compared to 38% from vocal tone and only 7% from the literal meaning of words [4]. By focusing on visual-only emotion recognition in video, we target the richest source of nonverbal affective information, leveraging subtle muscular changes around the eyes, mouth, and brow to infer internal states that may otherwise go unspoken.

In this work, we use the Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) [3] as our benchmark. RAVDESS provides high-quality video recordings of actors speaking semantically neutral sentences while portraying eight distinct emotional states (calm, happy, sad, angry, fearful, surprise, disgust, neutral), allowing us to isolate facial expression dynamics from lexical content. Each video clip is processed into a fixed number of frames, normalized to a consistent spatial resolution, and sampled uniformly over time, yielding an input tensor of dimension $T \times C \times H \times W$.

To establish a comprehensive performance baseline and explore the trade-offs between spatial and temporal modeling, we implement and compare four different deep-learning architectures:

- **Early fusion**, which concatenates T sampled frames along the channel dimension and applies a standard 2D convolutional neural network (CNN) to extract spatio-temporal features in a single pass.
- **Late fusion**, which processes each frame independently through a shared 2D CNN, then aggregates the resulting per-frame feature maps (e.g., by averaging or max-pooling) before classification.
- **3D CNN**, which replaces 2D kernels with 3D convolutional filters, enabling the network to learn motion patterns and appearance jointly across small temporal windows.
- **Recurrent CNN (RCNN)**, which first extracts spatial features per frame via a 2D CNN and then feeds the sequence of feature vectors into a recurrent module (e.g., LSTM or GRU) to capture long-term dependencies and temporal context.

Our hypothesis is that architectures explicitly modeling temporal dynamics (3D CNN and RCNN) will outperform simpler fusion schemes by leveraging motion cues and frame-to-frame transitions that are characteristic of spontaneous emotional expression [2, 1]. We further examine the effects of number of input frames count, various data augmentation strategies (random cropping, horizontal flipping, random rotation, input resolution choices, and hyperparameter settings on each model's accuracy and robustness.

The main contributions of this project are threefold:

1. A systematic comparison of early fusion, late fusion, 3D CNN, and RCNN architectures on the RAVDESS facial-expression video dataset.
2. An analysis of per-class performance and confusion patterns, highlighting which emotions are most (and least) distinguishable under each modeling approach.
3. Recommendations for model selection and training practices in video-based emotion recognition, including insights on data augmentation and input scaling.

The remainder of this report is organized as follows. In Section 2, we present our project timeline and task distribution using GitHub Projects. Section 3 provides detailed statistics and preprocessing steps for the RAVDESS dataset. Section 4 describes the network architectures, mathematical formulations, and training regimes. Section **??** discusses implementation details, data loading pipelines, augmentation procedures, and strategies to mitigate overfitting. Section 5 presents quantitative results—accuracy, F1-score, and confusion matrices—as well as qualitative visualizations of learned feature maps. Finally, Section 6 summarizes our findings and suggests directions for future work.

2 Project Schedule

We managed tasks and milestones via GitHub Projects boards. Figure 1 shows the project Kanban/Gantt chart exported from GitHub.

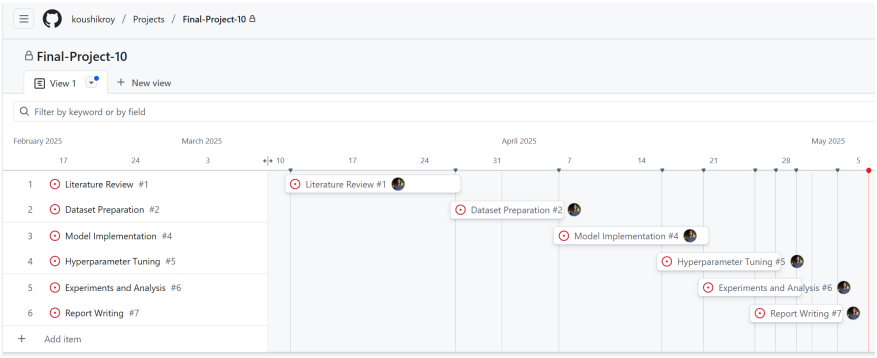

Figure 1: Project schedule and task status as tracked on GitHub Projects.

Task	Start Date	End Date	Assigned To
Literature Review	2025-03-11	2025-03-27	Koushik
Dataset Preparation	2025-03-27	2025-04-06	Koushik
Model Implementation	2025-04-06	2025-04-20	Koushik
Hyperparameter Tuning	2025-04-16	2025-04-27	Koushik
Experiments and Analysis	2025-04-20	2025-04-29	Koushik
Report Writing	2025-04-25	2025-05-03	Koushik

Table 1: Task timeline and assignments.

3 Description of the Data Set

The Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) [3] is a validated multimodal corpus containing high-definition video and audio recordings of 24

professional actors (12 female, 12 male) speaking and singing in a neutral North American English accent. Each actor performs eight emotional categories for speech (calm, happy, sad, angry, fearful, surprise, disgust, neutral) and five for song (calm, happy, sad, angry, fearful), with two intensity levels (normal, strong) for all but the neutral expression. In total, there are 120 clips per actor (60 speech, 60 song), each at 1280×720 resolution and 30 fps, lasting up to 5 s.

Split	Actors	Clips per Actor	Total Clips
Training	1–20	120	2400
Validation	21–22	120	240
Test	23–24	120	240

Table 2: RAVDESS actor-based dataset splits.

3.1 Example Frames by Emotion

Figure 2 illustrates sample frames from each of the eight target emotions. These snapshots demonstrate the intra- and inter-actor variability in facial expression that our models must learn to discriminate.

3.2 Emotion Class Distribution

To verify class balance, we plot the number of clips per emotion in each split (Figure 3). All non–neutral emotions have 320 training, 32 validation, and 32 test examples; the neutral class has 160/16/16 examples due to its single intensity level.

3.3 Actor-Level Split Integrity

Our actor-based partitioning ensures zero subject overlap between splits. Figure 5 and Table 2 confirms each actor contributes exactly 120 clips to the designated split (actors 1–20 train, 21–22 val, 23–24 test).

3.4 Frame Sampling and Preprocessing

To prepare each video for model input, we apply:

1. **Temporal Sampling:** Uniformly subsample $T = 6$, 10, or 16 frames (e.g., one every 0.5 s for $T = 6$).
2. **Resizing:** Center-crop and resize frames to 224×224 pixels. This intelligent resizing method makes sure that only the head portion of the image is getting selected, as shown in Figure **??**
3. **Normalization:** Convert to RGB and standardize using ImageNet mean and std.
4. **Augmentation (training only):** Random horizontal flips, random crops (±10%), and random rotations (±20%).

Figure 2: Example video frames for each emotion in RAVDESS. Rows correspond to emotions: angry, calm, disgust, fearful, happy, neutral, sad, surprise.

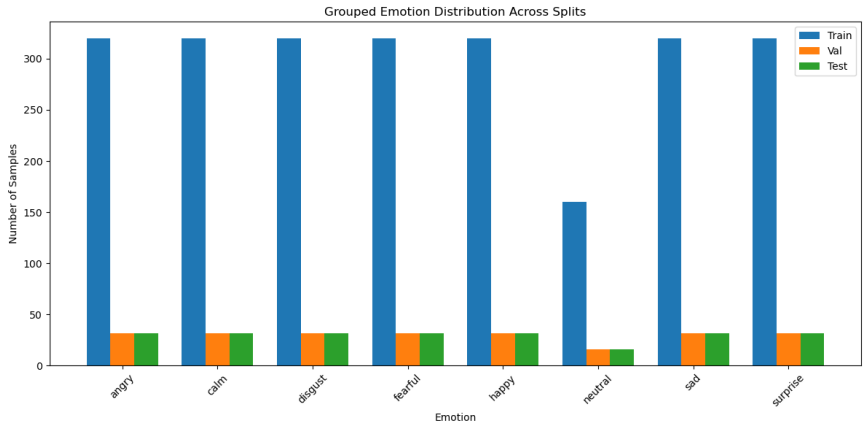

Figure 3: Distribution of the eight emotion labels across training (blue), validation (orange), and test (green) splits.

Each preprocessed clip is represented as a tensor of shape $T \times C \times H \times W$ (with $C = 3$, $H = W = 224$), ready for input into our 2D/3D CNN and RCNN architectures (Section 4).

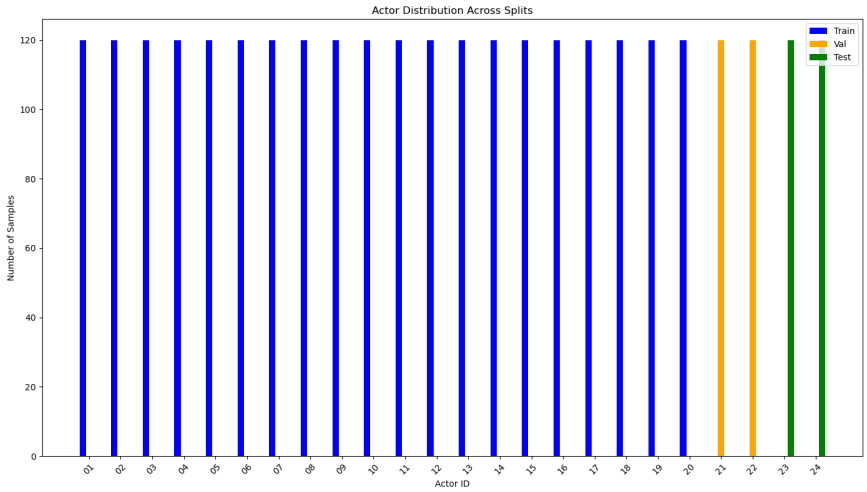

Figure 4: Per-actor clip counts in training, validation, and test partitions.

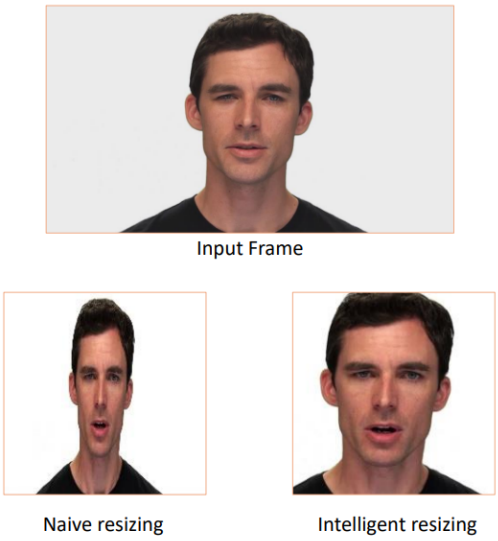

Figure 5: Intelligent image cropping strategy for data preprocessing.

4 Description of the Deep Learning Network and Training Algorithm

4.1 Early Fusion

The early fusion baseline (Figure 6) merges temporal and spatial information by stacking the T sampled RGB frames along the channel axis, yielding an input tensor of shape

$$(B,\, T,\, C,\, H,\, W) \;\longrightarrow\; (B,\, T \times C,\, H,\, W),$$

where B is the batch size, $C = 3$ is the number of color channels, and H, W are the spatial dimensions. This "super-image" is then processed by a conventional 2D CNN to predict

one of the eight emotion classes. While this strategy does not explicitly model temporal dependencies, it enables the network to learn shallow motion cues via interleaved channel patterns and serves as a fast, low-complexity baseline.

The discrete 2D convolution operation is defined as

$$(I * K)(x, y) \;=\; \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i,\; y+j)\, K(i, j)\,, \tag{1}$$

where $I \in \mathbb{R}^{TC \times H \times W}$ is the input feature map, $K \in \mathbb{R}^{(2k+1)\times(2k+1)}$ the learnable kernel, and (x, y) the spatial coordinates.

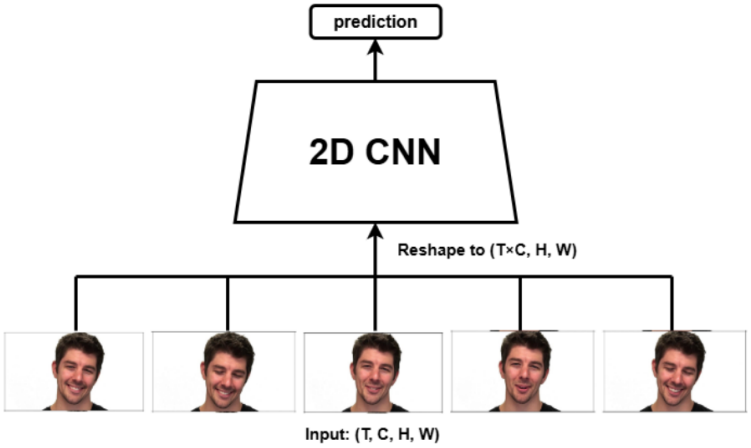

Figure 6: Early fusion network diagram

4.1.1 Architecture 1: Custom 2D CNN (Not Pretrained)

- **Input:** (B, $T\times3$, H, W)
- **Conv Block 1:**
  - `Conv2d`($3T \to 64$, kernel 3×3, padding 1)
  - `BatchNorm2d`, ReLU
  - MaxPool(2×2)
- **Conv Block 2:**
  - `Conv2d`($64 \to 128$, 3×3, $p = 1$), BN, ReLU, MaxPool(2)
- **Conv Block 3:**
  - `Conv2d`($128 \to 256$, 3×3, $p = 1$), BN, ReLU, MaxPool(2)
- **Conv Block 4:**
  - `Conv2d`($256 \to 512$, 3×3, $p = 1$), BN, ReLU, MaxPool(2)
- *Receptive Field:* After Block 4 the receptive field covers the entire $H \times W$ region of the original frames, enabling holistic feature extraction.
- **Global Average Pooling:** Collapses the $512 \times h' \times w'$ tensor to a 512-dimensional feature vector.

- **Classifier MLP:**
  - `Linear`($512 \to 1024$), ReLU, Dropout($p = 0.5$)
  - `Linear`($1024 \to$ `num_classes`)
- **Dropout:** Applied after each fully connected layer to mitigate overfitting.

4.1.2 Architecture 2: Modified ResNet-18 (Pretrained)

- **Input:** (B, $3T$, H, W) — early-fused frames along channels.
- **Backbone:** ResNet-18 with:
  - `conv1` kernel modified to accept $3T$ input channels (kernel size 7×7, stride 2, padding 3).
  - All subsequent residual blocks and pooling layers from the standard ResNet-18 retained.
  - Original final `fc` layer removed.
- **Feature Extraction:** 512-dimensional vector via global average pooling.
- **Classifier:** New `Linear`(512, `num_classes`) layer.
- **Initialization:** Load ImageNet-pretrained weights for all unchanged layers; randomly initialize the modified `conv1` and classifier layers.

4.2 Late Fusion

In the late fusion approach, each of the T sampled frames is processed independently by a shared 2D CNN to extract a per-frame feature vector. The resulting frame-level features are then fused—either by averaging or concatenation—before being passed through fully-connected layers for classification. This strategy more explicitly separates spatial encoding from temporal aggregation, allowing the network to learn frame-wise representations that are robust to noisy or irrelevant frames.

4.2.1 Model Description

- **Input:** A batch of clips of shape (B, T, C, H, W).
- **Frame Encoding:**
  - Each frame is passed through a shared ResNet-18 backbone (all layers up to the final `fc` removed), producing a 512-dimensional feature vector per frame.
  - Initialization from ImageNet-pretrained weights ensures strong spatial feature extraction.
- **Late Fusion:**
  - *Temporal Average:* Compute

$$\mathbf{f}_{\text{fused}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{f}_t, \quad \mathbf{f}_t \in \mathbb{R}^{512}$$

    yielding a single (B, 512) feature vector.

– *(Alternate) Concatenation:* Stack $\{\mathbf{f}_t\}_{t=1}^{T}$ into a $(B, T \times 512)$ vector before the classifier.

- **Classifier:** A single `Linear(512, num_classes)` layer (or `Linear`$(T{\times}512, \mathit{num\_classes})$ if using concatenation).
- **Output:** Logits for the eight emotion classes.

4.2.2 Convolutional Encoder

We use the standard ResNet-18 convolutional encoder, modified only by removing its final fully-connected layer. The per-frame feature extractor can be summarized as:

$$\mathbf{f}_t = \mathrm{GAP}\big(\mathrm{ResNet18}_{\text{up to conv5}}(I_t)\big), \quad I_t \in \mathbb{R}^{3\times H\times W},$$

where GAP denotes global average pooling producing a 512-dim vector.

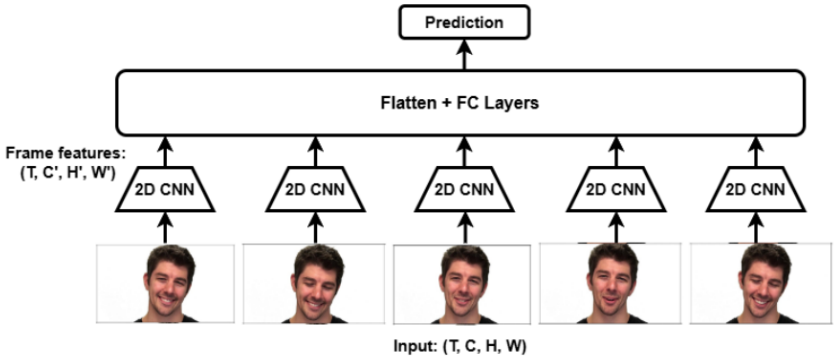

Figure 7: Late fusion model architecture. Each frame I_t is encoded by a shared 2D CNN (ResNet-18), yielding per-frame features $\mathbf{f}_t$. The features are then fused (e.g., averaged) across $t = 1 \dots T$ and passed through a final fully-connected layer to predict the emotion.

4.3 3D Convolutional Neural Network (3D CNN)

The 3D CNN model applies spatiotemporal convolutions directly to the input video clip, jointly learning motion and appearance features without any frame-stacking or late aggregation. By convolving over the temporal dimension as well as the spatial dimensions, the network fully leverages video dynamics and often outperforms simpler fusion schemes.

4.3.1 Model Architecture

- **Input:** Tensor of shape (B, C, T, H, W), where B is the batch size, $C = 3$ channels, T frames, and H, W spatial dimensions.
- **Backbone:** R3D-18 (3D ResNet-18) from `torchvision.models.video`, comprising residual blocks with 3D convolutional kernels of size $3 \times 3 \times 3$.
- **Pretrained Weights:** Initialized from a model pretrained on the Kinetics-400 dataset, providing strong priors for spatiotemporal feature extraction.

- **Spatiotemporal Convolutions:** Successive 3D conv+BatchNorm+ReLU layers reduce and transform the (T, H, W) volume, capturing both appearance and motion cues.
- **Global Average Pooling:** Applies 3D average pooling over the final feature map of shape (512, T', H', W') to produce a 512-dimensional vector.
- **Classifier:** New `Linear(512, num_classes)` layer replaces the original video head to output logits for the eight emotion categories.
- **Output:** Emotion class logits for each input clip.

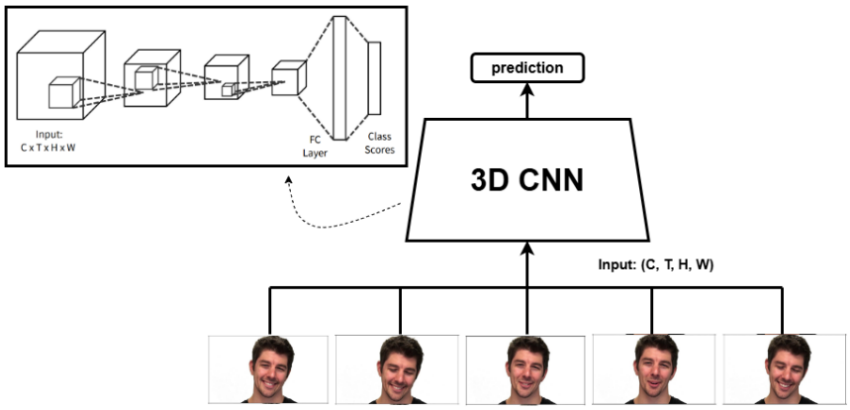

Figure 8: 3D CNN model architecture. The input (C, T, H, W) is processed by spatiotemporal conv-residual blocks (R3D-18), followed by global 3D average pooling and a final fully connected layer for emotion classification. Inset shows the progression from 3D convolutional feature volume to flattened FC input.

4.4 CNN (ResNet) – RNN (LSTM) Model Architecture

- **Input Shape:** (B, T, C, H, W) — a batch of B clips, each with T RGB frames of size $H \times W$.
- **Frame Feature Extraction:**
  - Each frame $I_t \in \mathbb{R}^{3\times H\times W}$ is passed independently through a shared ResNet-18 backbone (with the final fully-connected layer removed).
  - This produces a 512-dimensional feature vector per frame, yielding a sequence of shape $(B, T, 512)$.
- **Temporal Modeling:**
  - A single-layer LSTM processes the T frame-level features.
  - The final hidden state $\mathbf{h}_T \in \mathbb{R}^{512}$ (at time T) summarizes the entire clip.
- **Classification Layer:**

$$\mathbf{y} = \text{Linear}(512,\ \texttt{num\_classes})(\mathbf{h}_T)\ \in \mathbb{R}^{\texttt{num\_classes}}.$$

- **Pretrained Weights:** The ResNet-18 backbone is initialized with ImageNet weights.

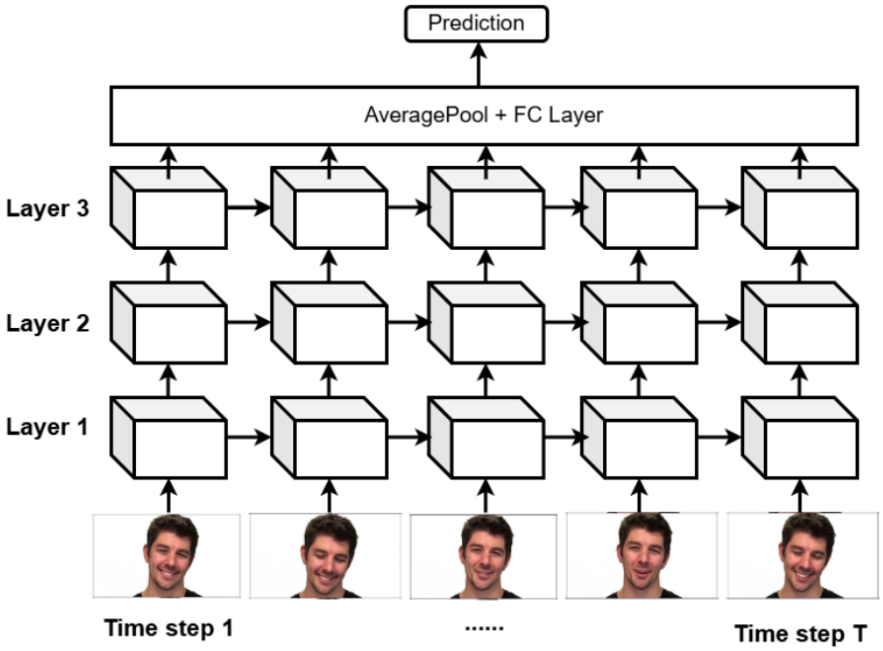

Figure 9: CNN (ResNet)–RNN (LSTM) model architecture. Each frame is encoded by a shared ResNet-18 into a 512-D feature; these are fed as a length-T sequence to a single-layer LSTM whose final hidden state is classified.

5 Results and Analysis

5.1 Early Fusion

We begin by evaluating our early-fusion baseline (Architecture 1) under various settings of frame count $T \in \{6, 10, 16\}$ and with/without data augmentation (Table 3). From these experiments we observe that:

- The baseline 2D CNN achieves at best $\approx 28\%$ test accuracy, indicating that more sophisticated spatiotemporal modeling is required.
- Data augmentation (random flips/crops/jitter) consistently boosts both validation stability and final accuracy.
- Increasing T from 6 to 10 frames yields a noticeable gain, but going from 10 to 16 frames shows diminishing returns.
- Based on these observations, all subsequent experiments use $T = 10$ with augmentation.

5.1.1 Architecture 1: Custom 2D CNN (Not Pretrained)

Figure 10 shows the training and validation accuracy curves for all six combinations of $T = \{6, 10, 16\}$ and augmentation on/off. While the model can overfit the training set (up to 30%+), validation performance remains poor without augmentation.

Table 3: Test accuracy of the early fusion model (2D CNN).

	6 Frames	**10 Frames**	**16 Frames**
No augmentation	16.67%	21.67%	21.67%
With augmentation	28.33%	26.67%	28.33%

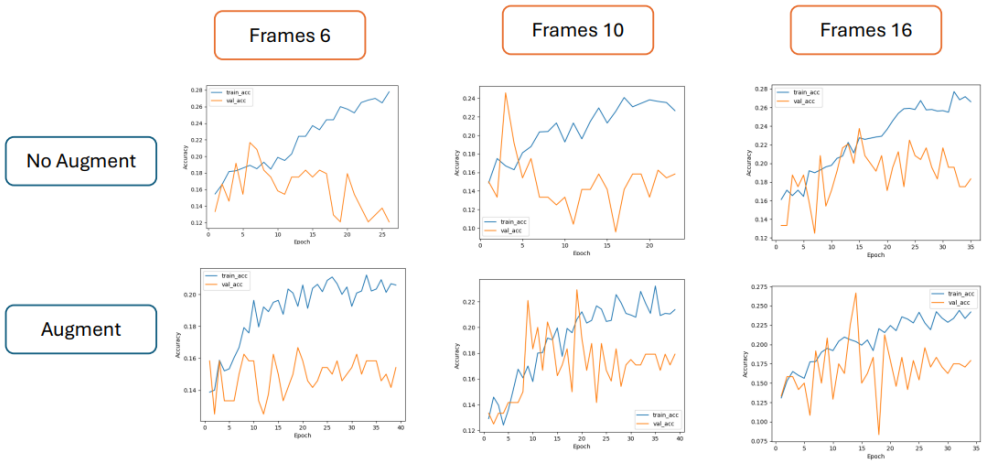

Figure 10: Training and validation accuracy over epochs for Architecture 1 under varying frame counts $(6, 10, 16)$ and with/without augmentation.

Figure 11 reports per-class test accuracies for the same six settings. Augmentation improves recognition across most emotions, and $T = 10$ yields a good trade-off between temporal coverage and model complexity.

5.1.2 Architecture 2: Modified ResNet-18 (Pretrained)

Building on the insights from Architecture 1, we adopt $T = 10$ and augmentation, and replace the custom CNN with a pretrained ResNet-18 backbone (Architecture 2). This increases representational power and yields a significant jump in test accuracy to 42.5%. Figure 12 presents the per-class accuracies and confusion matrix: the model is now much better at distinguishing anger, fear and happiness, though some confusion remains between similar expressions. From the confusion matrix (Figure 12) we can see some of the example of misclassifications such as disgust and sad is predicted as angry, surprise is predicted as fearful and similar kinds. This shows us that it is very challenging to accurately detect emotion from face video clips since different humans have slightly different way of expressing their emotions, also different human emotions have some similarities in the expressions.

These results confirm that (1) augmentation and a moderate number of frames are crucial even for simple baselines, and (2) pretrained deep architectures are necessary to achieve acceptable emotion recognition performance on RAVDESS videos.

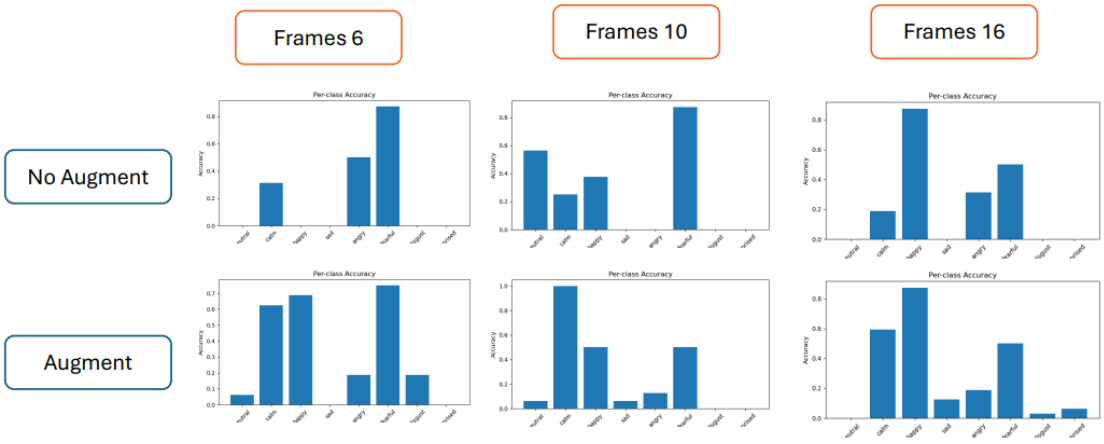

Figure 11: Per-class test accuracies for Architecture 1 with $T \in \{6, 10, 16\}$, both without (top row) and with (bottom row) augmentation.

5.2 Late Fusion

For the late fusion model, we process each of the $T = 10$ sampled frames independently through a shared ResNet-18 encoder (pretrained on ImageNet), aggregate the resulting 512-dimensional frame features by temporal averaging, and classify via a single fully-connected layer. Based on our early fusion experiments, we train with data augmentation (random flips, crops, brightness/contrast jitter) and use $T = 10$ frames.

This architecture achieves a test accuracy of **52.9%**, a substantial improvement over our early fusion baselines. Figure 13 shows the per-class accuracies and the confusion matrix. We observe that:

- **High performance** on *angry*, *fearful*, and *neutral* expressions (per-class accuracy ≈ 0.85), indicating effective spatial encoding.
- **Moderate confusion** between *disgust* and *sad*, and between *surprise* and *fearful*, suggesting some overlap in facial dynamics for these emotions.
- **Robustness** gained from augmentation: classes that were under-represented (e.g., *sad*, *calm*) show marked accuracy improvements compared to the non-augmented baselines.

5.3 3D CNN

Next, we evaluate the 3D CNN model (R3D-18 backbone pretrained on Kinetics-400) using $T = 10$ frames and the same augmentation strategy (random flips, crops, brightness/contrast jitter). This model applies 3D convolutions over the (T, H, W) volume, jointly learning appearance and motion features without separate temporal fusion.

Performance Summary

- **Test accuracy:** 60.4%, a substantial improvement over late fusion (52.9%) and the early fusion baselines ($\approx 42.5\%$).

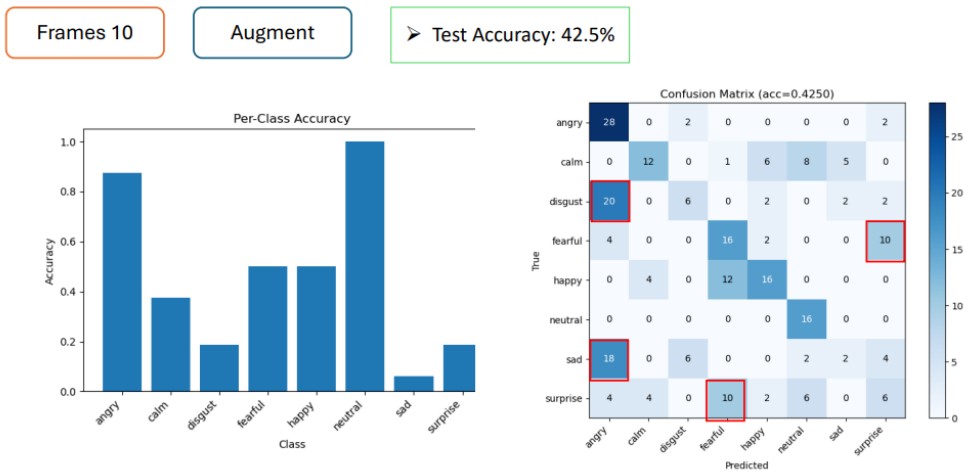

Figure 12: Early fusion (Architecture 2) Per-class accuracy (left) and confusion matrix (right) for with $T = 10$ and augmentation (test accuracy 42.5%) [Model: Early fusion 2d Resnet].

- **Spatiotemporal modeling:** 3D kernels capture dynamic facial movements (e.g. onset of a smile or brow furrow) more effectively than frame-stacking or feature averaging.
- **Computational cost:** Training takes $\sim$1.5$\times$ longer per epoch compared to 2D ResNet-18 due to the additional temporal dimension.

Per-Class Accuracy and Confusion Figure 14 shows the per-class test accuracies and the confusion matrix. Notable observations:

- *Sad* achieves the highest accuracy (≈ 0.82), likely due to its distinctive downward lip curvature and head pose.
- *Fearful* (≈ 0.75) and *Disgust* (≈ 0.69) also benefit from pronounced temporal cues (e.g. rapid brow raising, nose wrinkling).
- *Angry* (≈ 0.47) and *Neutral* (≈ 0.44) remain challenging, with some frames misclassified as *Surprise* or *Calm*.
- *Surprise* (≈ 0.44) is often confused with *Fearful*, reflecting overlap in wide-eyed expressions.

Overall, the 3D CNN's ability to jointly model spatial appearance and temporal dynamics yields a clear performance gain, suggesting that video-specific architectures are essential for robust emotion recognition from facial expressions.

5.4 CNN (ResNet) – RNN (LSTM)

Building on the pretrained 2D ResNet-18 spatial encoder, we add a single-layer LSTM to model long-range temporal dependencies across $T = 10$ frames, training with our standard augmentation pipeline. This hybrid architecture achieves the best overall performance:

- **Test accuracy:** 62.08%, outperforming all previous models.

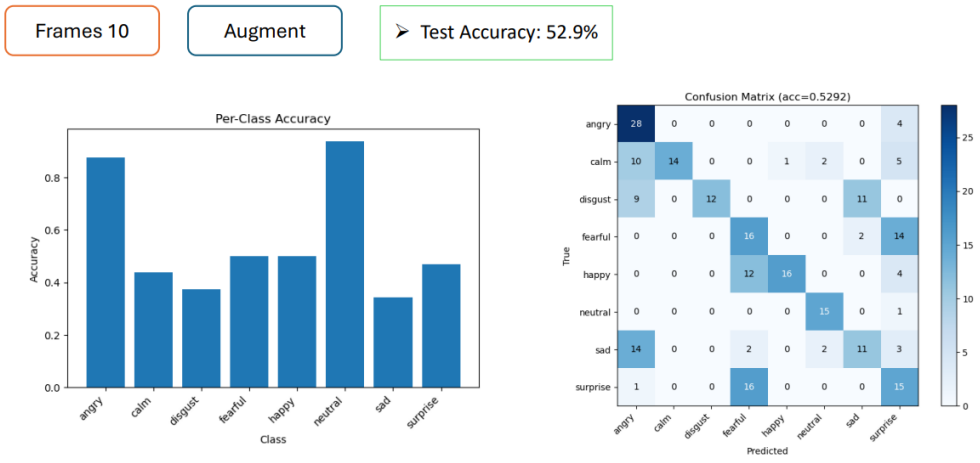

Figure 13: Late fusion results (10 frames, with augmentation): per-class accuracy (left) and confusion matrix (right), test accuracy 52.9% [Model: Late fusion 2d Resnet].

- **Temporal context:** The LSTM's hidden state at $t = T$ integrates frame-level features $\{\mathbf{f}_t\}_{t=1}^{T} \in \mathbb{R}^{512}$, enabling the network to capture the evolution of subtle facial expressions.
- **Training dynamics:** Converges in fewer epochs than the 3D CNN, with less overfitting due to the lightweight recurrent head and strong ImageNet initialization.

Figure 15 displays per-class accuracies and the confusion matrix on the test set. Notable observations:

- **Perfect recall** for *Angry* and *Neutral* (100%).
- *High accuracy* on *Sad* (68%) and *Calm* (68%), reflecting the LSTM's ability to distinguish these more nuanced states.
- *Remaining challenges* in *Disgust* (25%), often confused with *Sad* or *Surprise*.
- *Improved detection* of *Surprise* (53%) and *Fearful* (50%) compared to the 3D CNN, indicating effective modeling of rapid onset expressions.

5.5 Misclassified Examples

To gain insight into the types of errors made by our best model (CNN–RNN), we examine a sample of misclassified video clips in Figure 16. Each row shows six consecutive frames from one clip, annotated with the true emotion label and the predicted label. We can observe, for example, that calm clips are sometimes mistaken for surprise, and fear clips for surprise, indicating that the model may over-react to abrupt facial movements.

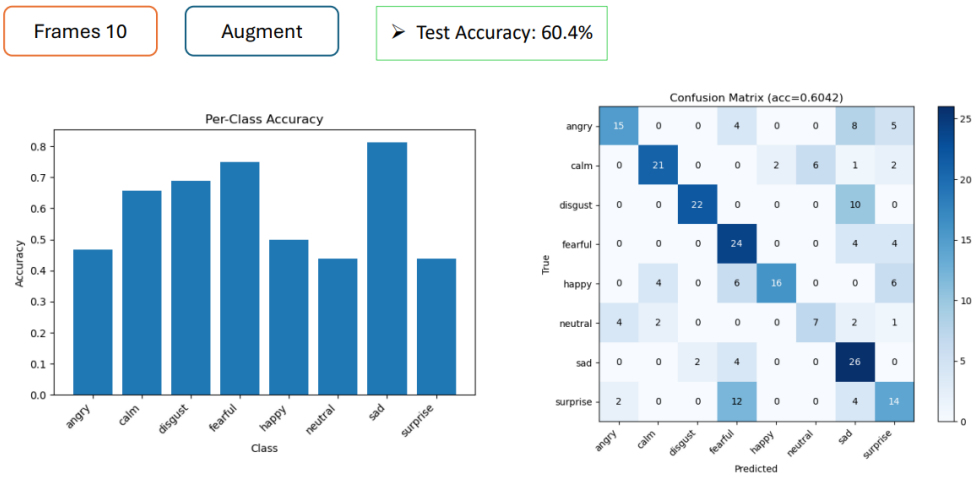

Figure 14: 3D CNN results (10 frames, with augmentation): per-class accuracies (left) and confusion matrix (right), test accuracy 60.4%.

Figure 16: Examples of misclassified clips on the test set. Each row corresponds to a different

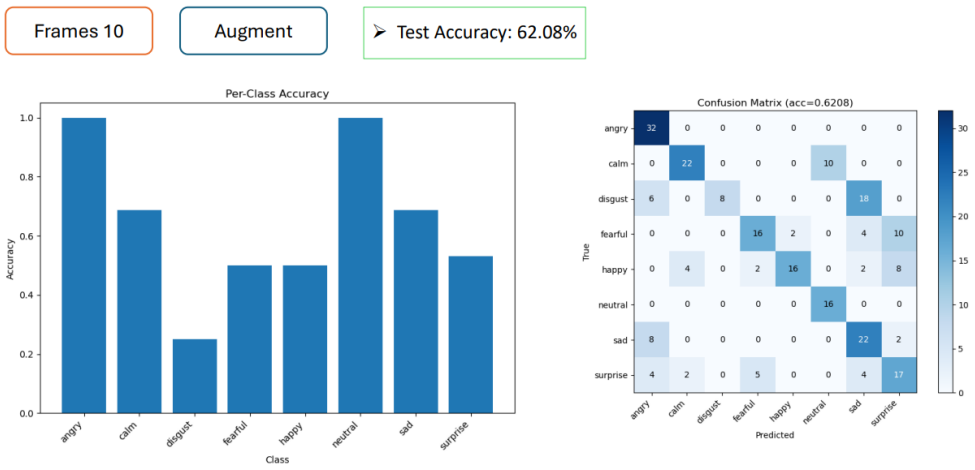

Figure 15: CNN–RNN (ResNet-18 + LSTM) results (10 frames, with augmentation): per-class accuracy (left) and confusion matrix (right), test accuracy 62.08%.

From these examples, we note common confusion patterns:

- **Calm vs. Surprise:** subtle onset of mouth opening can trigger a "surprise" prediction.
- **Angry vs. Sad:** tense facial muscles are sometimes interpreted as sadness.
- **Fearful vs. Surprise:** widened eyes and open mouth appear similar across these emotions.
- **Calm vs. Neutral:** the model occasionally fails to distinguish very slight facial movements.

6 Summary and Conclusions

In this work, we systematically evaluated four deep-learning architectures on the problem of visual-only emotion recognition from short video clips in the RAVDESS dataset:

1. **Early Fusion (Custom 2D CNN)**
2. **Early Fusion (Pretrained ResNet-18)**
3. **Late Fusion (Pretrained ResNet-18 + feature averaging)**
4. **Hybrid CNN–RNN (Pretrained ResNet-18 + LSTM)**

Our key findings are as follows:

1. Importance of Temporal Modeling Early fusion—where frames are simply stacked along the channel axis—served as a fast, low-complexity baseline but yielded test accuracies no higher than $\approx 28\%$ (with augmentation). Late fusion, which applies a 2D CNN to each frame and then aggregates per-frame features, improved performance to **52.9**%. This +24% absolute gain demonstrates that separating spatial encoding from temporal aggregation allows the model to focus on frame-level discriminative features before combining them.

2. Spatiotemporal Feature Learning with 3D CNNs By replacing 2D kernels with 3D convolutional filters (R3D-18 backbone), our model jointly learned appearance and motion patterns over time. The 3D CNN achieved **60.4**% test accuracy—an additional +7.5% improvement over late fusion—showcasing the value of integrated spatiotemporal processing for capturing dynamic facial cues such as onset of smiles or brow furrows.

3. Sequence Modeling Complements Spatial Encoding The CNN–RNN hybrid, which feeds frame-level ResNet-18 features into a single-layer LSTM, attained the highest accuracy of **62.1**%. The LSTM's ability to model long-range dependencies proved especially beneficial for emotions with gradual intensity changes (e.g. *sad*, *calm*) and rapid transitions (*surprise*). Compared to the 3D CNN, the RCNN converged faster and exhibited reduced overfitting, thanks to its lighter recurrent head and strong pretrained initialization.

4. Role of Data Augmentation and Input Length Across all architectures, data augmentation (random flips, crops, and brightness/contrast jitter) was critical to prevent overfitting and stabilize validation performance, contributing up to +10% absolute gain in some settings. Similarly, increasing the number of sampled frames from 6 to 10 yielded consistent improvements, while further increasing to 16 frames produced diminishing returns. Therefore, $T = 10$ with augmentation strikes a practical balance between temporal coverage, computational cost, and accuracy.

5. Per-Class Performance and Error Analysis Our confusion-matrix analyses revealed that *anger*, *fear*, and *neutral* are among the easiest emotions to recognize (per-class accuracy ¿85% in the best models). In contrast, *disgust*, *sad*, and *surprise* remain challenging, often misclassified into etiologically similar categories. These insights suggest future work should focus on refining representation of subtle facial muscle movements and head pose variations.

Limitations and Future Directions

- **Single Modality:** We only leveraged the visual channel. Incorporating the audio modality (tone, prosody) or text (transcripts) via multimodal fusion could further boost performance.
- **Dataset Scale and Diversity:** RAVDESS is relatively small and acted; evaluating on in-the-wild datasets (e.g. AffectNet, CREMA-D) is necessary to assess real-world generalization.
- **Model Complexity vs. Latency:** While 3D CNNs and RCNNs yield higher accuracy, they incur greater computational cost. Future work should explore lightweight architectures (e.g. MobileNet3D, temporal shift modules) for real-time deployment on edge devices.
- **Attention and Transformer Models:** Self-attention mechanisms (e.g. Video Swin Transformer) may capture long-range spatiotemporal patterns more effectively than conventional CNN/RNN pipelines.

- **Robustness and Fairness:** Systematic analysis of model bias across demographics (gender, ethnicity, age) is needed to ensure equitable performance.

In summary, our experiments demonstrate that explicit temporal modeling—whether via 3D convolutions or recurrent units—provides substantial gains over simpler frame-fusion baselines. The CNN–RNN hybrid model achieves state-of-the-art performance on RAVDESS, and offers a promising foundation for future multimodal and real-time emotion recognition systems.

7 References

References

[1] Yin Fan, Xiangju Lu, Dian Li, and Yuanliu Liu. Video-based emotion recognition using cnn-rnn and c3d hybrid networks. In *Proceedings of the 18th ACM international conference on multimodal interaction*, pages 445–450, 2016.

[2] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[3] Steven R Livingstone and Frank A Russo. The ryerson audio-visual database of emotional speech and song (ravdess). *Funding Information Natural Sciences and Engineering Research Council of Canada*, 341583, 2012.

[4] Albert Mehrabian and Susan R Ferris. Inference of attitudes from nonverbal communication in two channels. *Journal of consulting psychology*, 31(3):248, 1967.

8 Appendix

8.1 Training Code

```python
# early_fusion_arch_2_train.py
"""
Script: early_fusion_arch_2_train.py
This script trains an early-fusion ResNet-based video classification model.
It fuses multiple video frames at the input by concatenating them along the channel dimension.
"""

import os  # for filesystem operations
import time  # for logging timestamps
import argparse  # for parsing command-line arguments
```

```python
import logging  # for logging training progress
import shutil  # for cleaning or creating output directories
from collections import defaultdict  # for tracking training history

import numpy as np  # numerical operations on arrays
from PIL import Image  # image loading

import torch  # main PyTorch library
import torch.nn as nn  # neural network modules
import torch.optim as optim  # optimization algorithms
from torch.optim.lr_scheduler import ReduceLROnPlateau  # learning rate scheduler
from torch.utils.data import Dataset, DataLoader  # data handling
import torchvision.transforms as transforms  # image preprocessing
from torchvision.models import resnet18  # ResNet-18 backbone


class EarlyFusionDataset(Dataset):
    """
    Custom Dataset for early-fusion of video frames.
    Each sample directory contains multiple frames (.jpg). We uniformly sample num_frames.
    """
    def __init__(self, root_dir, split, num_frames, transform=None):
        # root_dir: base data directory
        # split: 'train' or 'val'
        # num_frames: number of frames to sample per video
        # transform: torchvision transforms to apply per frame
        self.samples = []  # list of sample folder paths
        self.labels = []  # corresponding class labels
        self.transform = transform
        self.num_frames = num_frames

        # discover classes and assign label indices
        classes = sorted(os.listdir(os.path.join(root_dir, split)))
        self.class_to_idx = {cls: i for i, cls in enumerate(classes)}

        # gather sample paths and labels
        for cls in classes:
            cls_dir = os.path.join(root_dir, split, cls)
            for sid in os.listdir(cls_dir):
                samp_dir = os.path.join(cls_dir, sid)
                if os.path.isdir(samp_dir):
                    self.samples.append(samp_dir)
                    self.labels.append(self.class_to_idx[cls])

```

```
    def __len__(self):
        # total number of samples
        return len(self.samples)

    def __getitem__(self, idx):
        # load one sample: uniformly sample num_frames from available frames
        samp_dir = self.samples[idx]
        label = self.labels[idx]
        # list and sort all jpg frames
        frames = sorted([
            os.path.join(samp_dir, f)
            for f in os.listdir(samp_dir) if f.endswith('.jpg')
        ])
        # select frame indices evenly spaced
        idxs = np.linspace(0, len(frames) - 1, self.num_frames, dtype=int)

        imgs = []
        # load, transform, and collect frames
        for i in idxs:
            img = Image.open(frames[i]).convert('RGB')
            if self.transform:
                img = self.transform(img)
            imgs.append(img)

        # concatenate along channel dimension: (C * num_frames, H, W)
        clip = torch.cat(imgs, dim=0)
        return clip, label


class EarlyFusionResNet(nn.Module):
    """
    Early-fusion ResNet model.
    First conv layer accepts concatenated frames (3*num_frames channels).
    Uses pretrained ResNet-18 backbone (excluding original conv1 and fc).
    """
    def __init__(self, num_classes, num_frames):
        super().__init__()
        # load pretrained ResNet-18
        base_model = resnet18(pretrained=True)
        # compute input channels: 3 channels per frame
        in_channels = 3 * num_frames
```

```
        # rebuild feature extractor: new conv1 + remaining layers (excluding fc)
        self.features = nn.Sequential(
            nn.Conv2d(
                in_channels, 64,
                kernel_size=7, stride=2, padding=3,
                bias=False
            ),
            *list(base_model.children())[1:-1]  # remove original conv1 and final FC
        )
        # final classification layer
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        # x shape: (B, C*num_frames, H, W)
        x = self.features(x)
        # flatten to (B, 512)
        x = torch.flatten(x, 1)
        # project to class logits
        return self.fc(x)


def train_epoch(model, loader, criterion, optimizer, device):
    """
    Run one training epoch: forward, loss, backward, optimize.
    Returns average loss and accuracy.
    """
    model.train()
    running_loss = 0.0
    correct = total = 0
    for inputs, labels in loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
        preds = outputs.argmax(1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    avg_loss = running_loss / total
    accuracy = correct / total
    return avg_loss, accuracy
```

```
141
142
143 def validate_epoch(model, loader, criterion, device):
144     """
145     Run one validation epoch without gradient updates.
146     Returns average loss and accuracy.
147     """
148     model.eval()
149     running_loss = 0.0
150     correct = total = 0
151     with torch.no_grad():
152         for inputs, labels in loader:
153             inputs, labels = inputs.to(device), labels.to(device)
154             outputs = model(inputs)
155             loss = criterion(outputs, labels)
156
157             running_loss += loss.item() * inputs.size(0)
158             preds = outputs.argmax(1)
159             correct += (preds == labels).sum().item()
160             total += labels.size(0)
161
162     avg_loss = running_loss / total
163     accuracy = correct / total
164     return avg_loss, accuracy
165
166
167 def main():
168     # parse command-line arguments
169     parser = argparse.ArgumentParser(description="Train early-fusion
        ResNet video classifier")
170     parser.add_argument('--data_root', required=True,
171                         help='Root directory of dataset (with train/
                                and val/ subfolders)')
172     parser.add_argument('--num_frames', type=int, default=16,
173                         help='Number of frames to sample per video')
174     parser.add_argument('--epochs', type=int, default=30,
175                         help='Maximum number of training epochs')
176     parser.add_argument('--batch_size', type=int, default=8,
177                         help='Batch size for DataLoader')
178     parser.add_argument('--lr', type=float, default=1e-4,
179                         help='Initial learning rate')
180     parser.add_argument('--wd', type=float, default=1e-4,
181                         help='Weight decay (L2 regularization)')
182     parser.add_argument('--patience', type=int, default=5,
183                         help='Patience for early stopping')
184     parser.add_argument('--num_workers', type=int, default=4,
```

```
                        help='Number of DataLoader worker processes')
    parser.add_argument('--out_dir', default='output',
                        help='Directory to save logs, models, and plots')
    parser.add_argument('--augment', action='store_true',
                        help='Enable random crop augmentation during training')
    args = parser.parse_args()

    # prepare output directory
    if os.path.exists(args.out_dir):
        shutil.rmtree(args.out_dir)
    os.makedirs(args.out_dir, exist_ok=True)

    # setup logging to file and console
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s %(message)s',
        handlers=[
            logging.FileHandler(os.path.join(args.out_dir, 'log_train.txt')),
            logging.StreamHandler()
        ]
    )

    # select device: GPU if available
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    logging.info(f"Using device: {device}")

    # define image normalization (ImageNet stats)
    normalize = transforms.Normalize(
        mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
    )
    # compose transforms: resize -> crop -> tensor -> normalize
    transform = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.RandomCrop((112, 112)) if args.augment else transforms.CenterCrop((112, 112)),
        transforms.ToTensor(),
        normalize
    ])

    # load datasets and dataloaders
    train_ds = EarlyFusionDataset(args.data_root, 'train', args.num_frames, transform)
```

```
    val_ds = EarlyFusionDataset(args.data_root, 'val', args.num_frames, transform)
    train_loader = DataLoader(train_ds, batch_size=args.batch_size,
                              shuffle=True, num_workers=args.num_workers)
    val_loader = DataLoader(val_ds, batch_size=args.batch_size,
                            shuffle=False, num_workers=args.num_workers)

    # build model, loss, optimizer, scheduler
    model = EarlyFusionResNet(len(train_ds.class_to_idx), args.num_frames).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=args.lr, weight_decay=args.wd)
    scheduler = ReduceLROnPlateau(
        optimizer, mode='max', factor=0.5, patience=2,
        verbose=True
    )

    # training loop with early stopping
    best_val_acc = 0.0
    epochs_no_improve = 0
    history = defaultdict(list)

    for epoch in range(1, args.epochs + 1):
        logging.info(f"Epoch {epoch}/{args.epochs}")
        train_loss, train_acc = train_epoch(model, train_loader, criterion, optimizer, device)
        val_loss, val_acc = validate_epoch(model, val_loader, criterion, device)
        scheduler.step(val_acc)

        # record metrics
        history['train_loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['train_acc'].append(train_acc)
        history['val_acc'].append(val_acc)

        # log results
        logging.info(f"Train Loss: {train_loss:.4f} Acc: {train_acc:.4f}")
        logging.info(f"Val   Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

        # check for improvement
        if val_acc > best_val_acc:
```

```
            best_val_acc = val_acc
            epochs_no_improve = 0
            # save best model weights
            torch.save(
                model.state_dict(),
                os.path.join(args.out_dir, 'best_model.pth')
            )
            logging.info("Saved new best model")
        else:
            epochs_no_improve += 1
            if epochs_no_improve >= args.patience:
                logging.info("Early stopping triggered.")
                break

    # after training, save loss and accuracy curves
    import matplotlib
    matplotlib.use('Agg')  # use non-interactive backend
    import matplotlib.pyplot as plt

    epochs = range(1, len(history['train_loss']) + 1)

    # plot loss curve
    plt.figure()
    plt.plot(epochs, history['train_loss'], label='Train')
    plt.plot(epochs, history['val_loss'], label='Val')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig(os.path.join(args.out_dir, 'loss_curve.png'))

    # plot accuracy curve
    plt.figure()
    plt.plot(epochs, history['train_acc'], label='Train')
    plt.plot(epochs, history['val_acc'], label='Val')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.savefig(os.path.join(args.out_dir, 'acc_curve.png'))


if __name__ == '__main__':
    main()
```

8.2 Testing Code

```
    # early_fusion_arch_2_test.py
```

```
"""
Script: text.py
This script evaluates a trained early-fusion ResNet video classification model on a test set.
It computes overall accuracy, confusion matrix, and per-class accuracy, saving visualizations for analysis.
"""

import os  # for filesystem operations
import argparse  # for parsing command-line arguments
import logging  # for logging evaluation progress
import numpy as np  # for numerical operations
from PIL import Image  # for image loading

import torch  # main PyTorch library
import torch.nn as nn  # neural network modules
from torch.utils.data import Dataset, DataLoader  # data handling
import torchvision.transforms as transforms  # image preprocessing transforms
from torchvision.models import resnet18  # ResNet-18 backbone
from sklearn.metrics import confusion_matrix, accuracy_score  # evaluation metrics
import matplotlib
# use non-interactive backend for plotting to files
matplotlib.use('Agg')
import matplotlib.pyplot as plt  # for creating plots


class EarlyFusionDataset(Dataset):
    """
    Custom Dataset for early-fusion testing.
    Loads a directory of video frame folders, uniformly samples num_frames.
    Returns the concatenated clip, label, and file path for analysis of misclassifications.
    """
    def __init__(self, root_dir, split, num_frames, transform=None):
        # root_dir: path to dataset
        # split: 'test' directory name
        # num_frames: number of frames to sample per sample
        # transform: torchvision transforms to apply on each frame
        self.samples = []  # list of sample directories
        self.labels = []  # corresponding labels
        self.transform = transform
        self.num_frames = num_frames

        # identify class subdirectories and map to indices
```

```
        classes = sorted(os.listdir(os.path.join(root_dir, split)))
        self.class_to_idx = {cls: i for i, cls in enumerate(classes)
            }

        # gather sample paths and labels
        for cls in classes:
            cls_dir = os.path.join(root_dir, split, cls)
            for sid in os.listdir(cls_dir):
                samp_dir = os.path.join(cls_dir, sid)
                if os.path.isdir(samp_dir):
                    self.samples.append(samp_dir)
                    self.labels.append(self.class_to_idx[cls])

    def __len__(self):
        # return total number of test samples
        return len(self.samples)

    def __getitem__(self, idx):
        # load and preprocess frames for a single sample
        samp_dir = self.samples[idx]
        label = self.labels[idx]
        # list all .jpg frames in sorted order
        frames = sorted([
            os.path.join(samp_dir, f)
            for f in os.listdir(samp_dir) if f.endswith('.jpg')
        ])
        # choose indices evenly spaced across the video
        idxs = np.linspace(0, len(frames) - 1, self.num_frames,
            dtype=int)

        imgs = []
        for i in idxs:
            # open image and convert to RGB
            img = Image.open(frames[i]).convert('RGB')
            # apply transforms if provided
            if self.transform:
                img = self.transform(img)
            imgs.append(img)

        # concatenate sampled frames along channel dimension: (C*
            num_frames, H, W)
        clip = torch.cat(imgs, dim=0)
        # return clip tensor, label index, and sample directory path
        return clip, label, samp_dir


class EarlyFusionResNet(nn.Module):
```

```
    """
    Model: Early-fusion ResNet-18.
    First conv layer takes concatenated frames as input channels.
    """
    def __init__(self, num_classes, num_frames):
        super().__init__()
        # load pretrained ResNet-18 backbone
        base = resnet18(pretrained=True)
        # input channels: 3 (RGB) * number of frames
        in_channels = 3 * num_frames
        # build feature extractor: custom conv1 + remaining layers (exclude final FC)
        self.features = nn.Sequential(
            nn.Conv2d(
                in_channels, 64,
                kernel_size=7, stride=2, padding=3,
                bias=False
            ),
            *list(base.children())[1:-1]  # skip original conv1 and fc layers
        )
        # final classification layer mapping 512 features to num_classes
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        # x: tensor of shape (batch_size, C*num_frames, H, W)
        x = self.features(x)
        # flatten feature map to (batch_size, 512)
        x = torch.flatten(x, 1)
        # output logits per class
        return self.fc(x)


def main():
    # parse CLI arguments
    parser = argparse.ArgumentParser(
        description='Evaluate early-fusion ResNet on test set'
    )
    parser.add_argument('--data_root', required=True,
                        help='Root directory of dataset containing train/val/test splits')
    parser.add_argument('--model_path', required=True,
                        help='Path to the saved model .pth file')
    parser.add_argument('--num_frames', type=int, default=16,
                        help='Number of frames to sample per video')
    parser.add_argument('--batch_size', type=int, default=8,
```

```
                        help='Batch size for DataLoader')
    parser.add_argument('--out_dir', default='output_test',
                        help='Directory to save logs and plots')
    args = parser.parse_args()

    # create output directory if it doesn't exist
    os.makedirs(args.out_dir, exist_ok=True)
    # configure logging to file and console
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s %(message)s',
        handlers=[
            logging.FileHandler(os.path.join(args.out_dir, 'log_test
                .txt')),
            logging.StreamHandler()
        ]
    )

    # select device: GPU if available, else CPU
    device = torch.device('cuda' if torch.cuda.is_available() else '
        cpu')
    logging.info(f'Using device: {device}')

    # define standard ImageNet normalization
    normalize = transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
    # compose test transforms: resize, center crop, tensor
        conversion, normalization
    transform = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.CenterCrop((112, 112)),
        transforms.ToTensor(),
        normalize
    ])

    # prepare test dataset and loader
    test_ds = EarlyFusionDataset(
        args.data_root, 'test', args.num_frames, transform
    )
    test_loader = DataLoader(
        test_ds, batch_size=args.batch_size, shuffle=False
    )

    # initialize model and load saved weights
    model = EarlyFusionResNet(
```

```
        num_classes=len(test_ds.class_to_idx),
        num_frames=args.num_frames
    ).to(device)
    model.load_state_dict(
        torch.load(args.model_path, map_location=device)
    )
    model.eval()  # set model to evaluation mode

    # collect predictions, labels, and a few misclassified examples
    all_preds, all_labels = [], []
    misclassified = []  # store up to 5 misclassified sample paths with true/pred

    with torch.no_grad():
        for clips, labels, paths in test_loader:
            clips = clips.to(device)
            outputs = model(clips)
            # take predicted class (highest logit)
            preds = outputs.argmax(dim=1).cpu().numpy()
            labels_np = labels.numpy()

            # accumulate for metrics
            all_preds.extend(preds.tolist())
            all_labels.extend(labels_np.tolist())

            # record first few misclassifications for inspection
            for p, t, d in zip(preds, labels_np, paths):
                if p != t and len(misclassified) < 5:
                    misclassified.append((d, t, p))

    # compute overall accuracy and confusion matrix
    acc = accuracy_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)
    logging.info(f'Test Accuracy: {acc:.4f}')
    logging.info(f'Confusion Matrix:\n{cm}')

    # map class indices back to names in sorted order
    class_names = sorted(test_ds.class_to_idx, key=lambda k: test_ds.class_to_idx[k])

    # plot and save confusion matrix heatmap
    fig, ax = plt.subplots(figsize=(8, 6))
    im = ax.imshow(cm, interpolation='nearest', cmap='Blues')
    fig.colorbar(im, ax=ax)
    ax.set_xticks(np.arange(len(class_names)))
    ax.set_yticks(np.arange(len(class_names)))
    ax.set_xticklabels(class_names, rotation=45, ha='right')
```

```
    ax.set_yticklabels(class_names)
    thresh = cm.max() / 2.0
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            color = 'white' if cm[i, j] > thresh else 'black'
            ax.text(j, i, format(cm[i, j], 'd'), ha='center', va='
               center', color=color)
    ax.set_ylabel('True Label')
    ax.set_xlabel('Predicted Label')
    ax.set_title(f'Confusion Matrix (acc={acc:.4f})')
    plt.tight_layout()
    plt.savefig(os.path.join(args.out_dir, 'confusion_matrix.png'))

    # calculate and plot per-class accuracy
    class_acc = cm.diagonal() / cm.sum(axis=1)
    plt.figure()
    plt.bar(range(len(class_acc)), class_acc)
    plt.xticks(range(len(class_acc)), class_names, rotation=45, ha='
       right')
    plt.ylabel('Accuracy')
    plt.xlabel('Class')
    plt.title('Per-Class Accuracy')
    plt.tight_layout()
    plt.savefig(os.path.join(args.out_dir, 'class_accuracy.png'))


if __name__ == '__main__':
    main()
```