

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

Koushik Roy

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	3
2	Project Schedule	4
3	Description of the Data Set	4
3.1	Example Frames by Emotion	5
3.2	Emotion Class Distribution	5
3.3	Actor-Level Split Integrity	5
3.4	Frame Sampling and Preprocessing	5
4	Description of the Deep Learning Network and Training Algorithm	7
4.1	Early Fusion	7
4.1.1	Architecture 1: Custom 2D CNN (Not Pretrained)	8
4.1.2	Architecture 2: Modified ResNet-18 (Pretrained)	9
4.2	Late Fusion	9
4.2.1	Model Description	9
4.2.2	Convolutional Encoder	10
4.3	3D Convolutional Neural Network (3D CNN)	10
4.3.1	Model Architecture	10
4.4	CNN (ResNet) – RNN (LSTM) Model Architecture	11

5	Implementation Details and Training Setup	12
5.1	Compute Environment and Dependencies	12
5.2	Dataset Organization	12
5.3	Data Loading Pipeline	13
5.4	Augmentation Pipeline	13
5.5	Training Hyperparameters	14
5.6	Overfitting Mitigation	14
6	Results and Analysis	14
6.1	Early Fusion	14
6.1.1	Architecture 1: Custom 2D CNN (Not Pretrained)	15
6.1.2	Architecture 2: Modified ResNet-18 (Pretrained)	15
6.2	Late Fusion	16
6.3	3D CNN	16
6.4	CNN (ResNet) – RNN (LSTM)	17
6.5	Misclassified Examples	18
7	Summary and Conclusions	20
8	Appendix	22
8.1	Training Code	22
8.2	Testing Code	29

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.
- **CNN-RNN**, 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 CNN-RNN) 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 5 discusses implementation details, data loading pipelines, augmentation procedures, and strategies to mitigate overfitting. Section 6 presents quantitative results—accuracy, and confusion matrices—as well as qualitative visualizations of learned feature maps. Finally, Section 7 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.

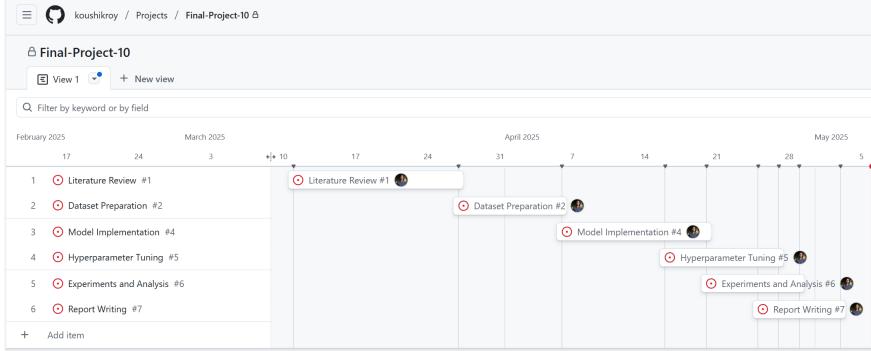


Figure 1: Project schedule and task status as tracked on GitHub Projects.

Table 1: Task timeline and assignments.

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

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.

Table 2: RAVDESS actor-based dataset splits.

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

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 4 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 5
3. **Normalization:** Convert to RGB and standardize using ImageNet mean and std.
4. **Augmentation (training only):** Random horizontal flips, random crops ($\pm 10\%$), and random rotations ($\pm 20\%$).



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

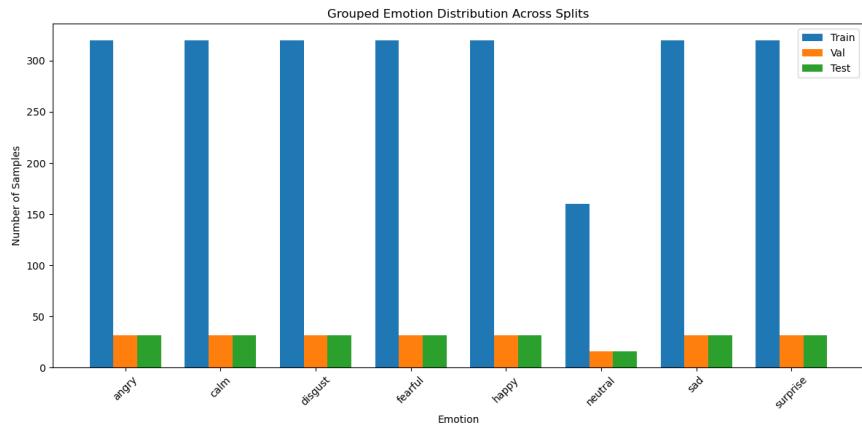


Figure 3: Distribution of the eight emotion labels across training (blue), validation (orange), and test (green) splits.

Each preprocessed clip is represented as a tensor of shape $T \times C \times H \times W$ (with $T = 6, 10$ or 16 frames, $C = 3$, $H = W = 224$), ready for input into our 2D/3D CNN and CNN-RNN architectures (Section 4).

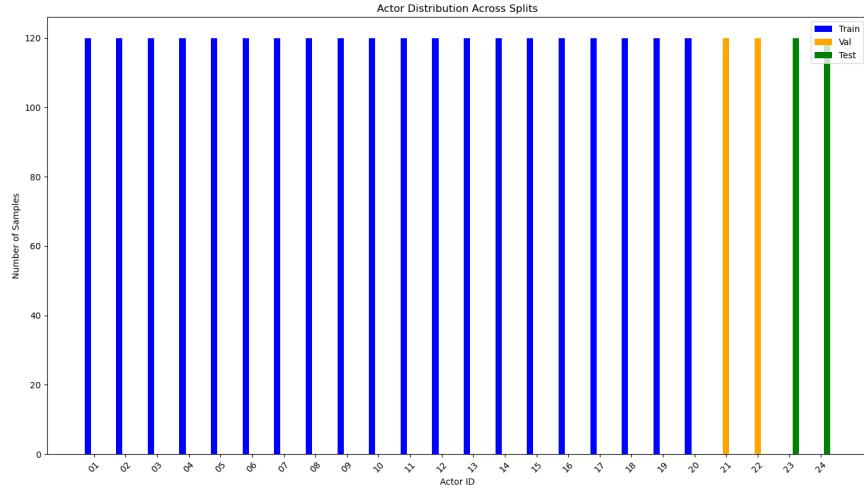


Figure 4: Per-actor clip counts in training, validation, and test partitions.

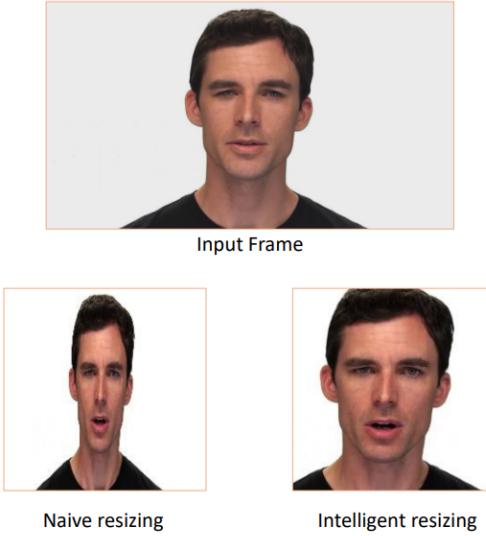


Figure 5: Intelligent image cropping strategy for data preprocessing.

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

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

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.

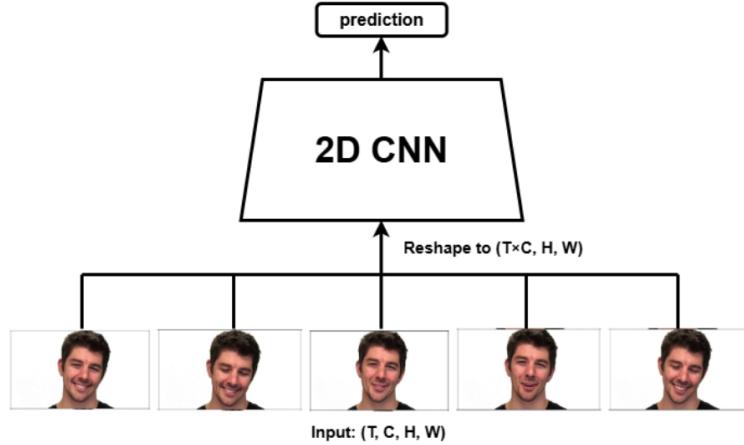


Figure 6: Early fusion network diagram

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

- **Input:** $(B, T \times 3, H, W)$
- **Conv Block 1:**
 - Conv2d($3T \rightarrow 64$, kernel 3×3 , padding 1)
 - BatchNorm2d, ReLU
 - MaxPool(2×2)
- **Conv Block 2:**
 - Conv2d($64 \rightarrow 128$, 3×3 , $p = 1$), BN, ReLU, MaxPool(2)
- **Conv Block 3:**
 - Conv2d($128 \rightarrow 256$, 3×3 , $p = 1$), BN, ReLU, MaxPool(2)
- **Conv Block 4:**
 - Conv2d($256 \rightarrow 512$, 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 → 1024)`, ReLU, Dropout($p = 0.5$)
 - `Linear(1024 → 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 (Figure 7) 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 by averaging 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×512, 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:

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

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

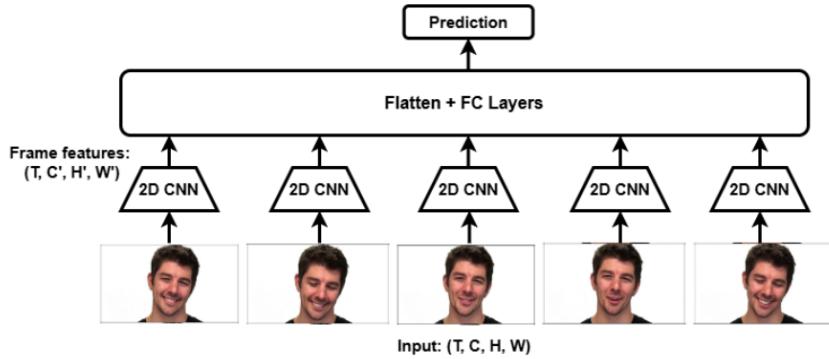


Figure 7: Late fusion model architecture. Each frame I_t is encoded by a shared 2D CNN (ResNet-18), yielding per-frame features \mathbf{f}_t . The features are then fused (e.g., averaged) across $t = 1 \dots T$ and passed through a final fully-connected layer to predict the emotion.

4.3 3D Convolutional Neural Network (3D CNN)

The 3D CNN model (Figure 8) 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.

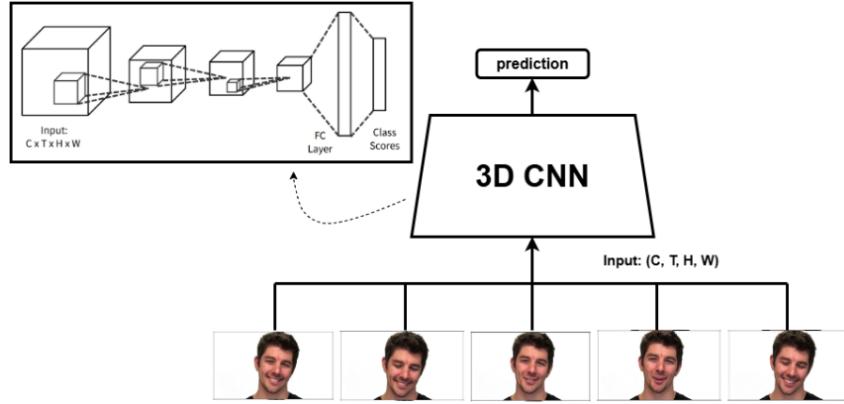


Figure 8: 3D CNN model architecture. The input (C, T, H, W) is processed by spatiotemporal conv-residual blocks (R3D-18), followed by global 3D average pooling and a final fully connected layer for emotion classification. Inset shows the progression from 3D convolutional feature volume to flattened FC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

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

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

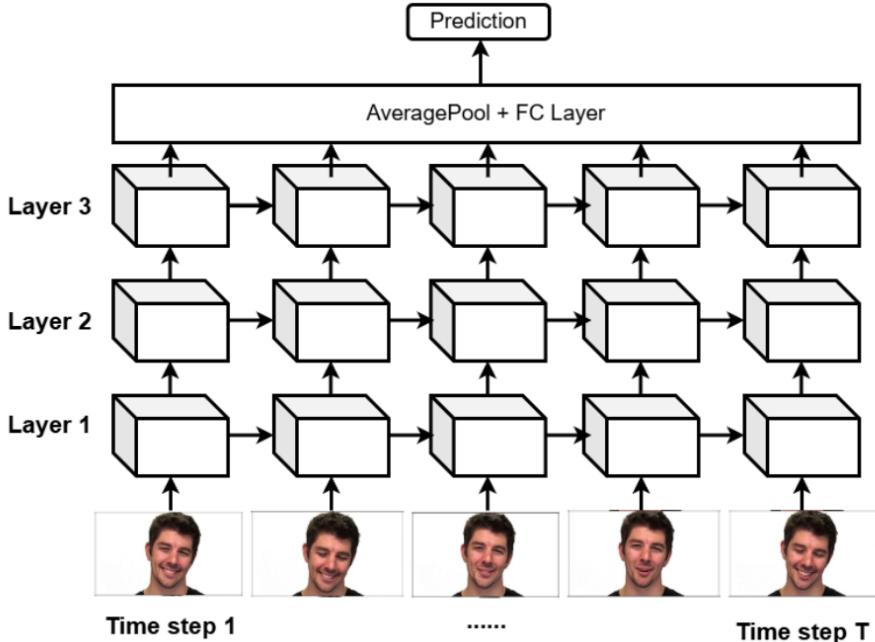


Figure 9: CNN (ResNet)-RNN (LSTM) model architecture. Each frame is encoded by a shared ResNet-18 into a 512-D feature; these are fed as a length- T sequence to a single-layer LSTM whose final hidden state is classified.

5 Implementation Details and Training Setup

This section details our compute environment, on-disk dataset organization, the PyTorch data-loading workflow (including frame sampling), the augmentation pipeline, all major hyperparameters, and the measures employed to counter overfitting.

5.1 Compute Environment and Dependencies

Experiments were conducted in Python 3.8 with PyTorch 1.9.0 and torchvision 0.10.0 on a single NVIDIA 4090 GPU (24 GB VRAM). Key libraries include NumPy, Pillow, scikit-learn for metrics, and Matplotlib for visualization. All random seeds (Python, NumPy, PyTorch) were fixed to ensure reproducibility, and deterministic CuDNN was enabled.

5.2 Dataset Organization

The RAVDESS clips were structured as:

```
\$DATA_ROOT/
  train/          % actor-based emotion folders
    angry/
      actor01_clip01/
        frame0001.jpg ... frame00N.jpg
    calm/
```

```

    ... (eight emotions total)
val/   (same)
test/  (same)

```

5.3 Data Loading Pipeline

A custom `Dataset` subclass implements:

1. Listing and sorting all frames in a clip folder.
2. Computing T uniformly spaced indices via

$$i_k = \lfloor k \frac{N-1}{T-1} \rfloor, \quad k = 0, \dots, T-1.$$

3. Loading each selected frame, applying per-frame transforms, and concatenating the resulting tensors along the channel axis to form a $(3T) \times H \times W$ input.

This is wrapped in a `DataLoader` with batch size = 8, 4 workers, pinning memory, and shuffling enabled during training only.

Algorithm 1 Frame Sampling Procedure

```

1: Input: clip directory; total frames  $N$ ; frames to sample  $T$ 
2: List and sort frame filenames  $\{f_0, \dots, f_{N-1}\}$ 
3: for  $k = 0$  to  $T - 1$  do
4:    $i \leftarrow \lfloor k(N-1)/(T-1) \rfloor$ 
5:   Load and transform frame  $f_i$ 
6:   Append to list
7: end for
8: Concatenate list along channel dimension
9: return concatenated tensor

```

5.4 Augmentation Pipeline

Augmentations applied *only* in training, in the following order:

- Resize shorter side to 128 px
- Random crop to 112×112
- Horizontal flip with $p = 0.5$
- Rotation up to $\pm 20^\circ$
- Conversion to tensor and normalization with ImageNet statistics

Validation and test use only resize, center-crop, tensor conversion, and normalization.

Parameter	Value	Notes
Batch size	8	per GPU
Optimizer	Adam	$\beta = (0.9, 0.999)$
Initial learning rate	1×10^{-4}	with weight decay
Weight decay (L2)	1×10^{-4}	on all weights
Scheduler	ReduceLROnPlateau	factor=0.5, patience=2
Epochs	100	early stop at pat.=5
Frame count T	10	best trade-off
Crop size	224×224	after resize

Table 3: Key training settings.

5.5 Training Hyperparameters

5.6 Overfitting Mitigation

We deployed multiple strategies:

- **Data augmentation** for spatial and photometric diversity.
- **Batch normalization** in all convolutional blocks.
- **Dropout** ($p = 0.5$) after fully-connected layers.
- **Weight decay** to regularize learned weights.
- **Early stopping** on validation accuracy (patience = 5).
- **LR scheduling** via ReduceLROnPlateau.
- **Gradient clipping** at 5.0 to stabilize training.

6 Results and Analysis

6.1 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 4). 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.

Table 4: 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%

6.1.1 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.

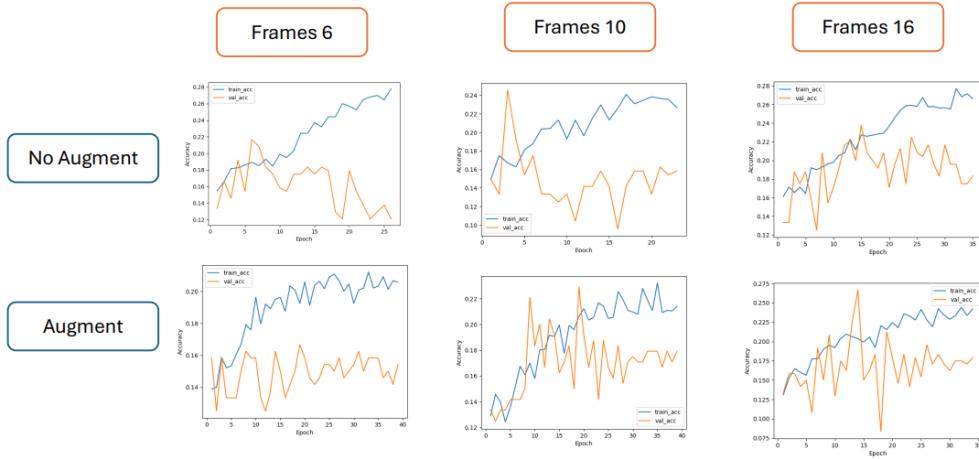


Figure 10: Training and validation accuracy over epochs for Architecture 1 under varying frame counts (6, 10, 16) and with/without augmentation.

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.

6.1.2 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.

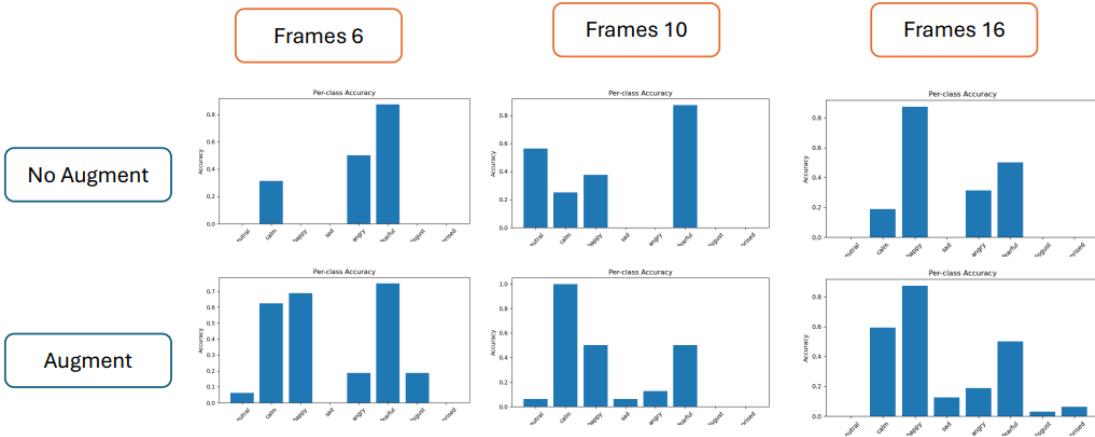


Figure 11: Per-class test accuracies for Architecture 1 with $T \in \{6, 10, 16\}$, both without (top row) and with (bottom row) augmentation.

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.

6.2 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, random rotation) 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*, 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.

6.3 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. This model applies 3D convolutions over the (T, H, W) volume, jointly learning appearance and motion features without separate temporal fusion.

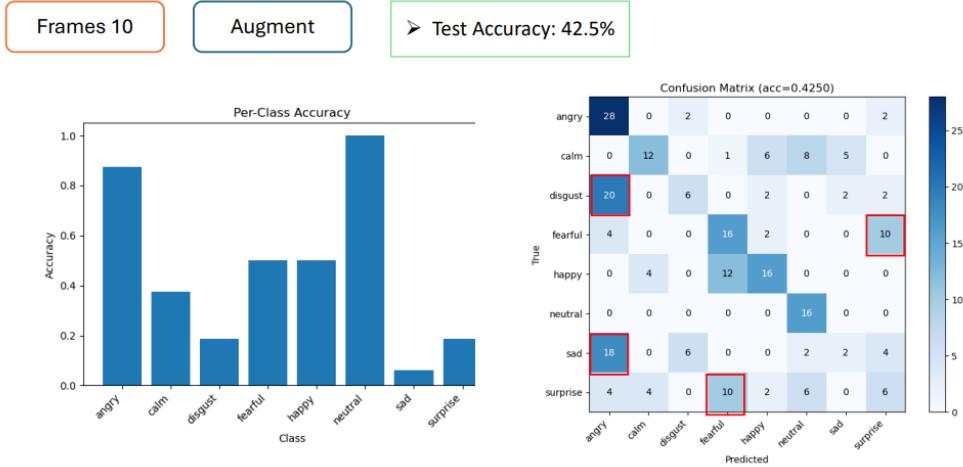


Figure 12: Early fusion (Architecture 2) Per-class accuracy (left) and confusion matrix (right) for $T = 10$ and augmentation (test accuracy 42.5%) [Model: Early fusion 2d Resnet].

Performance Summary

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

6.4 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:

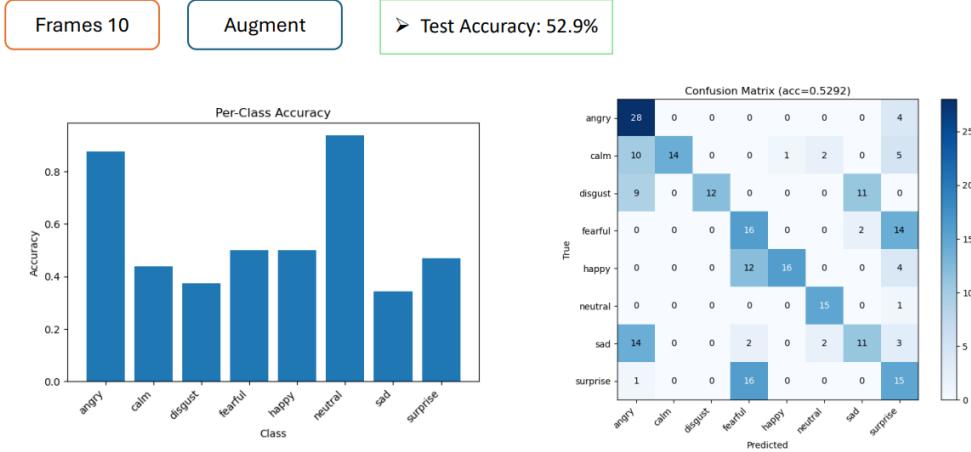


Figure 13: Late fusion results (10 frames, with augmentation): per-class accuracy (left) and confusion matrix (right), test accuracy 52.9% [Model: Late fusion 2d Resnet].

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

6.5 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.

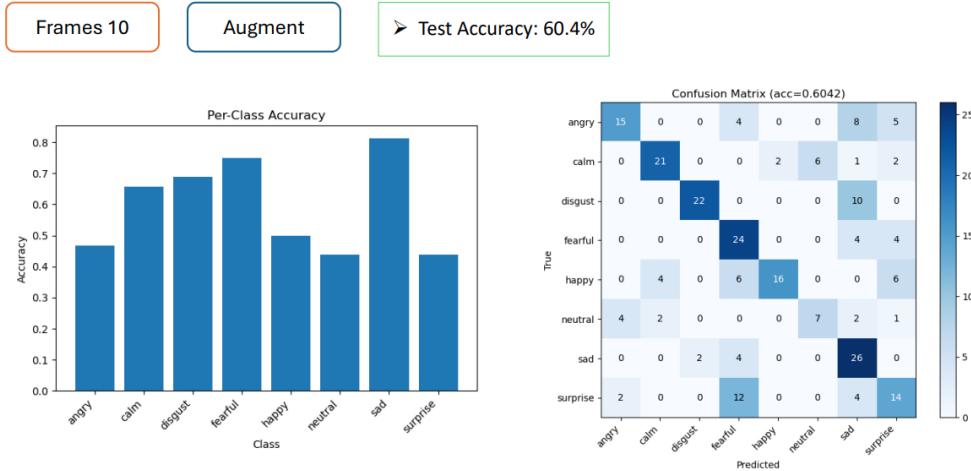


Figure 14: 3D CNN results (10 frames, with augmentation): per-class accuracies (left) and confusion matrix (right), test accuracy 60.4%.



Figure 16: Examples of misclassified clips on the test set.

From these examples, we note common confusion patterns:

- **Calm vs. Surprise:** subtle onset of mouth opening can trigger a “surprise” prediction.

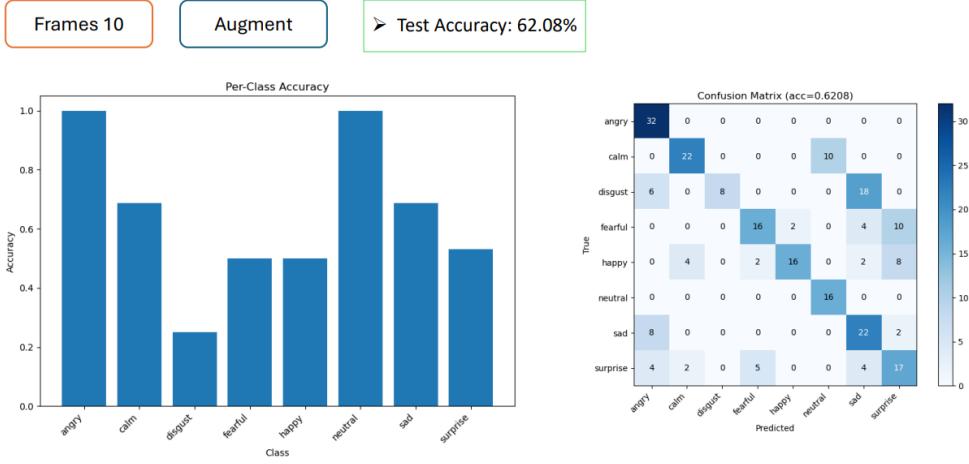


Figure 15: CNN–RNN (ResNet-18 + LSTM) results (10 frames, with augmentation): per-class accuracy (left) and confusion matrix (right), test accuracy 62.08%.

- **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.

7 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, Pretrained ResNet-18)**
2. **Late Fusion (Pretrained ResNet-18 + feature averaging)**
3. **3d CNN (ResNet 3D)**
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 random rotation) 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*, *sad*, and *neutral* are among the easiest emotions to recognize (per-class accuracy is greater than 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. Comparing 3d CNN with CNN-RNN, the overall accuracy on the CNN-RNN model is slightly better but the 3d CNN gives a better balance of the per class accuracy.

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 CNN-RNNs 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.

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

```

1 # early_fusion_arch_2_train.py
2 """
3 Script: early_fusion_arch_2_train.py
4 This script trains an early-fusion ResNet-based video classification
5 model.
6 It fuses multiple video frames at the input by concatenating them
7 along the channel dimension.
8 """
9
10 import os    # for filesystem operations
11 import time  # for logging timestamps
12 import argparse # for parsing command-line arguments
13 import logging # for logging training progress
14 import shutil # for cleaning or creating output directories
15 from collections import defaultdict # for tracking training history
16
17 import numpy as np   # numerical operations on arrays

```

```

16 from PIL import Image # image loading
17
18 import torch # main PyTorch library
19 import torch.nn as nn # neural network modules
20 import torch.optim as optim # optimization algorithms
21 from torch.optim.lr_scheduler import ReduceLROnPlateau # learning
22     rate scheduler
22 from torch.utils.data import Dataset, DataLoader # data handling
23 import torchvision.transforms as transforms # image preprocessing
24 from torchvision.models import resnet18 # ResNet-18 backbone
25
26
27 class EarlyFusionDataset(Dataset):
28     """
29     Custom Dataset for early-fusion of video frames.
30     Each sample directory contains multiple frames (.jpg). We
31     uniformly sample num_frames.
32     """
33
34     def __init__(self, root_dir, split, num_frames, transform=None):
35         # root_dir: base data directory
36         # split: 'train' or 'val'
37         # num_frames: number of frames to sample per video
38         # transform: torchvision transforms to apply per frame
39         self.samples = [] # list of sample folder paths
40         self.labels = [] # corresponding class labels
41         self.transform = transform
42         self.num_frames = num_frames
43
44         # discover classes and assign label indices
45         classes = sorted(os.listdir(os.path.join(root_dir, split)))
46         self.class_to_idx = {cls: i for i, cls in enumerate(classes)}
47
48         # gather sample paths and labels
49         for cls in classes:
50             cls_dir = os.path.join(root_dir, split, cls)
51             for sid in os.listdir(cls_dir):
52                 samp_dir = os.path.join(cls_dir, sid)
53                 if os.path.isdir(samp_dir):
54                     self.samples.append(samp_dir)
55                     self.labels.append(self.class_to_idx[cls])
56
57     def __len__(self):
58         # total number of samples
59         return len(self.samples)
60
61     def __getitem__(self, idx):

```

```

60     # load one sample: uniformly sample num_frames from
61     # available frames
62     samp_dir = self.samples[idx]
63     label = self.labels[idx]
64     # list and sort all jpg frames
65     frames = sorted([
66         os.path.join(samp_dir, f)
67         for f in os.listdir(samp_dir) if f.endswith('.jpg')
68     ])
69     # select frame indices evenly spaced
70     idxs = np.linspace(0, len(frames) - 1, self.num_frames,
71                         dtype=int)
72
73     imgs = []
74     # load, transform, and collect frames
75     for i in idxs:
76         img = Image.open(frames[i]).convert('RGB')
77         if self.transform:
78             img = self.transform(img)
79         imgs.append(img)
80
81     # concatenate along channel dimension: (C * num_frames, H, W
82     # )
83     clip = torch.cat(imgs, dim=0)
84     return clip, label
85
86
87
88
89
90     class EarlyFusionResNet(nn.Module):
91         """
92             Early-fusion ResNet model.
93             First conv layer accepts concatenated frames (3*num_frames
94             channels).
95             Uses pretrained ResNet-18 backbone (excluding original conv1 and
96             fc).
97         """
98
99         def __init__(self, num_classes, num_frames):
100             super().__init__()
101             # load pretrained ResNet-18
102             base_model = resnet18(pretrained=True)
103             # compute input channels: 3 channels per frame
104             in_channels = 3 * num_frames
105             # rebuild feature extractor: new conv1 + remaining layers (
106             # excluding fc)
107             self.features = nn.Sequential(
108                 nn.Conv2d(
109                     in_channels, 64,
110                     kernel_size=7, stride=2, padding=3,

```

```

101         bias=False
102     ),
103     *list(base_model.children())[1:-1] # remove original
104     # conv1 and final FC
105 )
106 # final classification layer
107 self.fc = nn.Linear(512, num_classes)
108
109 def forward(self, x):
110     # x shape: (B, C*num_frames, H, W)
111     x = self.features(x)
112     # flatten to (B, 512)
113     x = torch.flatten(x, 1)
114     # project to class logits
115     return self.fc(x)
116
117 def train_epoch(model, loader, criterion, optimizer, device):
118     """
119     Run one training epoch: forward, loss, backward, optimize.
120     Returns average loss and accuracy.
121     """
122     model.train()
123     running_loss = 0.0
124     correct = total = 0
125     for inputs, labels in loader:
126         inputs, labels = inputs.to(device), labels.to(device)
127         optimizer.zero_grad()
128         outputs = model(inputs)
129         loss = criterion(outputs, labels)
130         loss.backward()
131         optimizer.step()
132
133         running_loss += loss.item() * inputs.size(0)
134         preds = outputs.argmax(1)
135         correct += (preds == labels).sum().item()
136         total += labels.size(0)
137
138     avg_loss = running_loss / total
139     accuracy = correct / total
140     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
149     model.eval()
150     running_loss = 0.0
151     correct = total = 0
152     with torch.no_grad():
153         for inputs, labels in loader:
154             inputs, labels = inputs.to(device), labels.to(device)
155             outputs = model(inputs)
156             loss = criterion(outputs, labels)
157
158             running_loss += loss.item() * inputs.size(0)
159             preds = outputs.argmax(1)
160             correct += (preds == labels).sum().item()
161             total += labels.size(0)
162
163     avg_loss = running_loss / total
164     accuracy = correct / total
165     return avg_loss, accuracy
166
167 def main():
168     # parse command-line arguments
169     parser = argparse.ArgumentParser(description="Train early-fusion
170         ResNet video classifier")
171     parser.add_argument('--data_root', required=True,
172                         help='Root directory of dataset (with train/
173                             and val/ subfolders)')
174     parser.add_argument('--num_frames', type=int, default=16,
175                         help='Number of frames to sample per video')
176     parser.add_argument('--epochs', type=int, default=30,
177                         help='Maximum number of training epochs')
178     parser.add_argument('--batch_size', type=int, default=8,
179                         help='Batch size for DataLoader')
180     parser.add_argument('--lr', type=float, default=1e-4,
181                         help='Initial learning rate')
182     parser.add_argument('--wd', type=float, default=1e-4,
183                         help='Weight decay (L2 regularization)')
184     parser.add_argument('--patience', type=int, default=5,
185                         help='Patience for early stopping')
186     parser.add_argument('--num_workers', type=int, default=4,
187                         help='Number of DataLoader worker processes')
188     parser.add_argument('--out_dir', default='output',
189                         help='Directory to save logs, models, and
190                             plots')
191     parser.add_argument('--augment', action='store_true',

```

```

189         help='Enable random crop augmentation during
190             training')
191
192     args = parser.parse_args()
193
194     # prepare output directory
195     if os.path.exists(args.out_dir):
196         shutil.rmtree(args.out_dir)
197     os.makedirs(args.out_dir, exist_ok=True)
198
199     # setup logging to file and console
200     logging.basicConfig(
201         level=logging.INFO,
202         format='%(asctime)s %(message)s',
203         handlers=[
204             logging.FileHandler(os.path.join(args.out_dir, 'log_train.txt')),
205             logging.StreamHandler()
206         ]
207     )
208
209     # select device: GPU if available
210     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
211     logging.info(f"Using device: {device}")
212
213     # define image normalization (ImageNet stats)
214     normalize = transforms.Normalize(
215         mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
216     )
217
218     # compose transforms: resize -> crop -> tensor -> normalize
219     transform = transforms.Compose([
220         transforms.Resize((128, 128)),
221         transforms.RandomCrop((112, 112)) if args.augment else
222             transforms.CenterCrop((112, 112)),
223         transforms.ToTensor(),
224         normalize
225     ])
226
227
228     # load datasets and dataloaders
229     train_ds = EarlyFusionDataset(args.data_root, 'train', args.
230         num_frames, transform)
231     val_ds = EarlyFusionDataset(args.data_root, 'val', args.
232         num_frames, transform)
233     train_loader = DataLoader(train_ds, batch_size=args.batch_size,
234         shuffle=True, num_workers=args.
235             num_workers)
236     val_loader = DataLoader(val_ds, batch_size=args.batch_size,

```

```

229         shuffle=False, num_workers=args.
230             num_workers)
231
232     # build model, loss, optimizer, scheduler
233     model = EarlyFusionResNet(len(train_ds.class_to_idx), args.
234         num_frames).to(device)
235     criterion = nn.CrossEntropyLoss()
236     optimizer = optim.Adam(model.parameters(), lr=args.lr,
237         weight_decay=args.wd)
238     scheduler = ReduceLROnPlateau(
239         optimizer, mode='max', factor=0.5, patience=2,
240         verbose=True
241     )
242
243     # training loop with early stopping
244     best_val_acc = 0.0
245     epochs_no_improve = 0
246     history = defaultdict(list)
247
248     for epoch in range(1, args.epochs + 1):
249         logging.info(f"Epoch {epoch}/{args.epochs}")
250         train_loss, train_acc = train_epoch(model, train_loader,
251             criterion, optimizer, device)
252         val_loss, val_acc = validate_epoch(model, val_loader,
253             criterion, device)
254         scheduler.step(val_acc)
255
256         # record metrics
257         history['train_loss'].append(train_loss)
258         history['val_loss'].append(val_loss)
259         history['train_acc'].append(train_acc)
260         history['val_acc'].append(val_acc)
261
262         # log results
263         logging.info(f"Train Loss: {train_loss:.4f} Acc: {train_acc
264             :.4f}")
265         logging.info(f"Val    Loss: {val_loss:.4f} Acc: {val_acc:.4f}
266             ")
267
268         # check for improvement
269         if val_acc > best_val_acc:
270             best_val_acc = val_acc
271             epochs_no_improve = 0
272             # save best model weights
273             torch.save(
274                 model.state_dict(),
275                 os.path.join(args.out_dir, 'best_model.pth'))

```

```

269     )
270     logging.info("Saved new best model")
271 else:
272     epochs_no_improve += 1
273     if epochs_no_improve >= args.patience:
274         logging.info("Early stopping triggered.")
275         break
276
277 # after training, save loss and accuracy curves
278 import matplotlib
279 matplotlib.use('Agg') # use non-interactive backend
280 import matplotlib.pyplot as plt
281
282 epochs = range(1, len(history['train_loss']) + 1)
283
284 # plot loss curve
285 plt.figure()
286 plt.plot(epochs, history['train_loss'], label='Train')
287 plt.plot(epochs, history['val_loss'], label='Val')
288 plt.xlabel('Epoch')
289 plt.ylabel('Loss')
290 plt.legend()
291 plt.savefig(os.path.join(args.out_dir, 'loss_curve.png'))
292
293 # plot accuracy curve
294 plt.figure()
295 plt.plot(epochs, history['train_acc'], label='Train')
296 plt.plot(epochs, history['val_acc'], label='Val')
297 plt.xlabel('Epoch')
298 plt.ylabel('Accuracy')
299 plt.legend()
300 plt.savefig(os.path.join(args.out_dir, 'acc_curve.png'))
301
302
303 if __name__ == '__main__':
304     main()

```

8.2 Testing Code

```

1      # early_fusion_arch_2_test.py
2      """
3      Script: text.py
4      This script evaluates a trained early-fusion ResNet video
5      classification model on a test set.
6      It computes overall accuracy, confusion matrix, and per-class
7      accuracy, saving visualizations for analysis.

```

```

6      """
7
8  import os    # for filesystem operations
9  import argparse # for parsing command-line arguments
10 import logging # for logging evaluation progress
11 import numpy as np   # for numerical operations
12 from PIL import Image # for image loading
13
14 import torch # main PyTorch library
15 import torch.nn as nn # neural network modules
16 from torch.utils.data import Dataset, DataLoader # data handling
17 import torchvision.transforms as transforms # image preprocessing
18     transforms
19 from torchvision.models import resnet18 # ResNet-18 backbone
20 from sklearn.metrics import confusion_matrix, accuracy_score #
21     evaluation metrics
22 import matplotlib
23 # use non-interactive backend for plotting to files
24 matplotlib.use('Agg')
25 import matplotlib.pyplot as plt # for creating plots
26
27
28 class EarlyFusionDataset(Dataset):
29     """
30         Custom Dataset for early-fusion testing.
31         Loads a directory of video frame folders, uniformly samples
32             num_frames.
33         Returns the concatenated clip, label, and file path for analysis
34             of misclassifications.
35     """
36
37     def __init__(self, root_dir, split, num_frames, transform=None):
38         # root_dir: path to dataset
39         # split: 'test' directory name
40         # num_frames: number of frames to sample per sample
41         # transform: torchvision transforms to apply on each frame
42         self.samples = [] # list of sample directories
43         self.labels = [] # corresponding labels
44         self.transform = transform
45         self.num_frames = num_frames
46
47         # identify class subdirectories and map to indices
48         classes = sorted(os.listdir(os.path.join(root_dir, split)))
49         self.class_to_idx = {cls: i for i, cls in enumerate(classes)
50 }
51
52         # gather sample paths and labels
53         for cls in classes:

```

```

48         cls_dir = os.path.join(root_dir, split, cls)
49         for sid in os.listdir(cls_dir):
50             samp_dir = os.path.join(cls_dir, sid)
51             if os.path.isdir(samp_dir):
52                 self.samples.append(samp_dir)
53                 self.labels.append(self.class_to_idx[cls])
54
55     def __len__(self):
56         # return total number of test samples
57         return len(self.samples)
58
59     def __getitem__(self, idx):
60         # load and preprocess frames for a single sample
61         samp_dir = self.samples[idx]
62         label = self.labels[idx]
63         # list all .jpg frames in sorted order
64         frames = sorted([
65             os.path.join(samp_dir, f)
66             for f in os.listdir(samp_dir) if f.endswith('.jpg')
67         ])
68         # choose indices evenly spaced across the video
69         idxs = np.linspace(0, len(frames) - 1, self.num_frames,
70                           dtype=int)
71
72         imgs = []
73         for i in idxs:
74             # open image and convert to RGB
75             img = Image.open(frames[i]).convert('RGB')
76             # apply transforms if provided
77             if self.transform:
78                 img = self.transform(img)
79             imgs.append(img)
80
81         # concatenate sampled frames along channel dimension: (C*
82             # num_frames, H, W)
83         clip = torch.cat(imgs, dim=0)
84         # return clip tensor, label index, and sample directory path
85         return clip, label, samp_dir
86
87     class EarlyFusionResNet(nn.Module):
88         """
89         Model: Early-fusion ResNet-18.
90         First conv layer takes concatenated frames as input channels.
91         """
92         def __init__(self, num_classes, num_frames):
93             super().__init__()

```

```

93     # load pretrained ResNet-18 backbone
94     base = resnet18(pretrained=True)
95     # input channels: 3 (RGB) * number of frames
96     in_channels = 3 * num_frames
97     # build feature extractor: custom conv1 + remaining layers (exclude final FC)
98     self.features = nn.Sequential(
99         nn.Conv2d(
100             in_channels, 64,
101             kernel_size=7, stride=2, padding=3,
102             bias=False
103         ),
104         *list(base.children())[1:-1]    # skip original conv1 and fc layers
105     )
106     # final classification layer mapping 512 features to num_classes
107     self.fc = nn.Linear(512, num_classes)
108
109     def forward(self, x):
110         # x: tensor of shape (batch_size, C*num_frames, H, W)
111         x = self.features(x)
112         # flatten feature map to (batch_size, 512)
113         x = torch.flatten(x, 1)
114         # output logits per class
115         return self.fc(x)
116
117
118     def main():
119         # parse CLI arguments
120         parser = argparse.ArgumentParser(
121             description='Evaluate early-fusion ResNet on test set')
122
123         parser.add_argument('--data_root', required=True,
124                             help='Root directory of dataset containing train/val/test splits')
125         parser.add_argument('--model_path', required=True,
126                             help='Path to the saved model .pth file')
127         parser.add_argument('--num_frames', type=int, default=16,
128                             help='Number of frames to sample per video')
129         parser.add_argument('--batch_size', type=int, default=8,
130                             help='Batch size for DataLoader')
131         parser.add_argument('--out_dir', default='output_test',
132                             help='Directory to save logs and plots')
133
134         args = parser.parse_args()
135
# create output directory if it doesn't exist

```

```

136     os.makedirs(args.out_dir, exist_ok=True)
137     # configure logging to file and console
138     logging.basicConfig(
139         level=logging.INFO,
140         format='%(asctime)s %(message)s',
141         handlers=[
142             logging.FileHandler(os.path.join(args.out_dir, 'log_test
143                                         .txt')),
144             logging.StreamHandler()
145         ]
146     )
147
148     # select device: GPU if available, else CPU
149     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
150     logging.info(f'Using device: {device}')
151
152     # define standard ImageNet normalization
153     normalize = transforms.Normalize(
154         mean=[0.485, 0.456, 0.406],
155         std=[0.229, 0.224, 0.225]
156     )
157     # compose test transforms: resize, center crop, tensor
158     # conversion, normalization
159     transform = transforms.Compose([
160         transforms.Resize((128, 128)),
161         transforms.CenterCrop((112, 112)),
162         transforms.ToTensor(),
163         normalize
164     ])
165
166     # prepare test dataset and loader
167     test_ds = EarlyFusionDataset(
168         args.data_root, 'test', args.num_frames, transform
169     )
170     test_loader = DataLoader(
171         test_ds, batch_size=args.batch_size, shuffle=False
172     )
173
174     # initialize model and load saved weights
175     model = EarlyFusionResNet(
176         num_classes=len(test_ds.class_to_idx),
177         num_frames=args.num_frames
178     ).to(device)
179     model.load_state_dict(
180         torch.load(args.model_path, map_location=device)
181     )

```

```

180     model.eval() # set model to evaluation mode
181
182     # collect predictions, labels, and a few misclassified examples
183     all_preds, all_labels = [], []
184     misclassified = [] # store up to 5 misclassified sample paths
185         with true/pred
186
187     with torch.no_grad():
188         for clips, labels, paths in test_loader:
189             clips = clips.to(device)
190             outputs = model(clips)
191             # take predicted class (highest logit)
192             preds = outputs.argmax(dim=1).cpu().numpy()
193             labels_np = labels.numpy()
194
195             # accumulate for metrics
196             all_preds.extend(preds.tolist())
197             all_labels.extend(labels_np.tolist())
198
199             # record first few misclassifications for inspection
200             for p, t, d in zip(preds, labels_np, paths):
201                 if p != t and len(misclassified) < 5:
202                     misclassified.append((d, t, p))
203
204             # compute overall accuracy and confusion matrix
205             acc = accuracy_score(all_labels, all_preds)
206             cm = confusion_matrix(all_labels, all_preds)
207             logging.info(f'Test Accuracy: {acc:.4f}')
208             logging.info(f'Confusion Matrix:\n{cm}')
209
210             # map class indices back to names in sorted order
211             class_names = sorted(test_ds.class_to_idx, key=lambda k: test_ds
212                 .class_to_idx[k])
213
214             # plot and save confusion matrix heatmap
215             fig, ax = plt.subplots(figsize=(8, 6))
216             im = ax.imshow(cm, interpolation='nearest', cmap='Blues')
217             fig.colorbar(im, ax=ax)
218             ax.set_xticks(np.arange(len(class_names)))
219             ax.set_yticks(np.arange(len(class_names)))
220             ax.set_xticklabels(class_names, rotation=45, ha='right')
221             ax.set_yticklabels(class_names)
222             thresh = cm.max() / 2.0
223             for i in range(cm.shape[0]):
224                 for j in range(cm.shape[1]):
225                     color = 'white' if cm[i, j] > thresh else 'black'

```

```

224         ax.text(j, i, format(cm[i, j], 'd'), ha='center', va='center',
225                     color=color)
226     ax.set_ylabel('True Label')
227     ax.set_xlabel('Predicted Label')
228     ax.set_title(f'Confusion Matrix (acc={acc:.4f})')
229     plt.tight_layout()
230     plt.savefig(os.path.join(args.out_dir, 'confusion_matrix.png'))
231
232     # calculate and plot per-class accuracy
233     class_acc = cm.diagonal() / cm.sum(axis=1)
234     plt.figure()
235     plt.bar(range(len(class_acc)), class_acc)
236     plt.xticks(range(len(class_acc)), class_names, rotation=45, ha='right')
237     plt.ylabel('Accuracy')
238     plt.xlabel('Class')
239     plt.title('Per-Class Accuracy')
240     plt.tight_layout()
241     plt.savefig(os.path.join(args.out_dir, 'class_accuracy.png'))
242
243 if __name__ == '__main__':
244     main()

```