🎭 Speech Emotion Recognition - Complete Guide

📋 TABLE OF CONTENTS

🚀 PART 1: PROJECT SETUP

Step 1.1: Create Project Folder Structure

```
CDAC_PROJECT/
├── emotion_recognition.py    # Main training script
├── app.py                    # Streamlit web interface
├── DATASET TO TRAIN/
│   └── CREMA-D/
│       └── AudioWAV/
│           ├── 1001_DFA_ANG_XX.wav
│           ├── 1001_DFA_DIS_XX.wav
│           └── ... (4,281 files)
├── emotion_model.h5          # Will be created after training
├── label_encoder_classes.npy # Will be created after training
└── requirements.txt          # Dependencies
```

Step 1.2: Install Python (if not installed)

1. Download Python 3.8-3.10 from https://www.python.org/downloads/
2. **IMPORTANT**: Check "Add Python to PATH" during installation
3. Verify installation:

```bash
python --version
# Should show: Python 3.x.x
```

Step 1.3: Install Required Libraries

Open Command Prompt (CMD) or PowerShell:

```bash
# Navigate to project folder
cd C:\Users\Dennismz\Desktop\CDAC_PROJECT

# Install all dependencies at once
pip install tensorflow==2.13.0
pip install librosa==0.10.1
pip install scikit-learn==1.3.0
pip install pyaudio
pip install noisereduce==2.0.1
pip install scipy==1.10.1
pip install matplotlib==3.7.2
pip install seaborn==0.12.2
pip install numpy==1.24.3
pip install pandas==2.0.3
pip install streamlit==1.28.0
pip install plotly==5.17.0
pip install sounddevice==0.4.6
pip install soundfile==0.12.1
```

OR create requirements.txt and install all at once:

```bash
# Create requirements.txt with this content:
tensorflow==2.13.0
librosa==0.10.1
scikit-learn==1.3.0
pyaudio
noisereduce==2.0.1
scipy==1.10.1
matplotlib==3.7.2
seaborn==0.12.2
numpy==1.24.3
pandas==2.0.3
streamlit==1.28.0
plotly==5.17.0
sounddevice==0.4.6
soundfile==0.12.1

# Then install:
pip install -r requirements.txt
```

Step 1.4: Verify Installation

```bash
python -c "import tensorflow; print('TensorFlow:', tensorflow.__version__)"
python -c "import librosa; print('Librosa:', librosa.__version__)"
python -c "import sklearn; print('Scikit-learn:', sklearn.__version__)"
```

📚 PART 2: UNDERSTANDING THE CODE

2.1: HIGH-LEVEL ARCHITECTURE

```
┌─────────────────────────────────────────┐
│           AUDIO INPUT                   │
│         (3 seconds of speech)           │
└────────────────┬────────────────────────┘
                 │
                 ▼
┌─────────────────────────────────────────┐
│         PREPROCESSING                   │
│  • Noise Reduction (removes background noise)  │
│  • Bandpass Filter (keeps 300-3400 Hz)  │
│  • Normalization (scales amplitude)     │
└────────────────┬────────────────────────┘
                 │
                 ▼
┌─────────────────────────────────────────┐
│      FEATURE EXTRACTION (194 features)  │
│  • MFCC (40): Voice characteristics     │
│  • Chroma (12): Pitch information       │
│  • Mel-Spec (128): Energy distribution  │
│  • Contrast (7): Texture                │
│  • Tonnetz (6): Harmony                 │
│  • ZCR (1): Signal changes              │
└────────────────┬────────────────────────┘
                 │
                 ▼
┌─────────────────────────────────────────┐
│         CNN-LSTM MODEL                  │
│                                         │
│  CNN Layers (Pattern Detection)         │
│    ├─ Conv1D + BatchNorm + MaxPool + Dropout  │
│    ├─ Conv1D + BatchNorm + MaxPool + Dropout  │
│    └─ Conv1D + BatchNorm + MaxPool + Dropout  │
│                                    │
│  LSTM Layers (Temporal Processing)               │
│    ├─ LSTM(128) + Dropout                  │
│    └─ LSTM(64) + Dropout                   │
│                                    │
│  Dense Layers (Classification)              │
│    ├─ Dense(128) + Dropout                 │
│    ├─ Dense(64) + Dropout                  │
│    └─ Dense(6, softmax) → Probabilities          │
└──────────────────────────────┬──────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────┐
│         EMOTION OUTPUT                     │
│  [angry, disgust, fearful, happy, neutral, sad]        │
│  [0.05,  0.02,    0.08,    0.78,  0.05,     0.02]      │
│                    ↑                       │
│              HAPPY (78% confident)               │
└─────────────────────────────────────────────────────────────┘
```

2.2: CODE EXPLANATION - SECTION BY SECTION

SECTION 1: AUDIO PREPROCESSING

```python
def apply_noise_reduction(audio, sr):
    return nr.reduce_noise(y=audio, sr=sr, prop_decrease=0.8)
```

What it does:

- Removes 80% of background noise
- Uses spectral gating technique
- Makes speech clearer

Why it's needed:

- Real-world audio has background noise
- Noise can confuse the model
- Improves feature extraction quality

Example:

```
Before: [hiss...] "I'm happy" [hiss...]
After:  "I'm happy" (clear)
```

```python
def bandpass_filter(audio, lowcut=300, highcut=3400, sr=22050):
    b, a = butter_bandpass(lowcut, highcut, sr)
    return lfilter(b, a, audio)
```

What it does:

- Keeps frequencies between 300-3400 Hz
- Removes very low (rumble) and very high (hiss) frequencies
- Simulates telephone quality

Why 300-3400 Hz?

- Human speech fundamental: 85-255 Hz
- Most speech information: 300-3400 Hz
- Removes non-speech frequencies

Visual:

```
Frequency Spectrum:
0 Hz ──────────────────────────────────── 22050 Hz
     ↓        ↓         ↓
  Remove   KEEP (speech)  Remove
  (bass)   300-3400 Hz   (noise)
```

```python
def preprocess_audio(file_path, apply_noise_red=True, apply_bandpass=True):
    audio, sr = librosa.load(file_path, duration=3, offset=0.5, sr=22050)
    audio = audio / np.max(np.abs(audio))  # Normalize
```

What it does:

1. **Load audio**: Read 3 seconds starting at 0.5 seconds
    - Why 0.5s offset? Skips initial silence

   - Why 3s? Standard length for emotion detection

2. **Normalize**: Scale all audio to range [-1, 1]
   - Why? Makes loud and quiet recordings comparable
   - Formula: audio / max(|audio|)

Example:

```
Original audio: [-500, 0, 1000] (loud)
Normalized:     [-0.5, 0, 1.0]  (standard range)
```

SECTION 2: DATA AUGMENTATION

```python
def augment_audio(audio, sr):
    augmentations = []
    augmentations.append(audio)                    # 1. Original
    augmentations.append(add_noise(audio, 0.003))   # 2. Light noise
    augmentations.append(add_noise(audio, 0.007))   # 3. Heavy noise
    augmentations.append(pitch_shift(audio, sr, 2))  # 4. Higher pitch
    augmentations.append(pitch_shift(audio, sr, -2)) # 5. Lower pitch
    augmentations.append(time_stretch(audio, 0.9))   # 6. Slower
    augmentations.append(time_stretch(audio, 1.1))   # 7. Faster
    augmentations.append(change_speed(audio, 0.95))  # 8. Slight slow
    augmentations.append(change_speed(audio, 1.05))  # 9. Slight fast
    return augmentations
```

Why augmentation?

- **Problem**: Limited data (4,281 samples)
- **Solution**: Create variations (×9) = 38,529 samples!
- **Benefit**: Model learns to handle real-world variations

What each augmentation simulates:

1. **Noise addition**
   - Simulates: Noisy environments (traffic, room noise)
   - Effect: Adds random values to signal
2. **Pitch shift**
   - Simulates: Different voice pitches (men/women/children)
   - Effect: Shifts frequency up or down

- Example: Makes deep voice sound higher

3. **Time stretch**
   - Simulates: Different speaking speeds
   - Effect: Makes audio longer/shorter without changing pitch
   - Example: Fast talker vs. slow talker
4. **Speed change**
   - Simulates: Natural variation in speech rate
   - Effect: Changes both duration and pitch slightly

Visual Example:

```
Original:    "I'm happy" (1.0s, normal pitch)
Noise:       "I'm happy" (1.0s, with static)
Pitch +2:    "I'm happy" (1.0s, higher voice)
Time 0.9x:   "I'm  happy" (1.1s, slower)
```

SECTION 3: FEATURE EXTRACTION

```python
def extract_features(audio, sr):
    result = np.array([])

    # 1. MFCC (40 features)
    mfccs = np.mean(librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=40).T, axis=0)
    result = np.hstack((result, mfccs))
```

MFCC - Mel-Frequency Cepstral Coefficients (Most Important!)

What it captures:

- Vocal tract shape
- Timbre (voice quality)
- Speaking style

How it works:

1. Convert audio to frequency spectrum
2. Apply Mel scale (mimics human hearing)
3. Take logarithm

4. Apply DCT (Discrete Cosine Transform)
5. Extract 40 coefficients

Why 40 coefficients?

- First 13: Most important voice features
- Next 27: Capture subtle variations
- More = better emotion distinction

Analogy: Like a "fingerprint" of voice characteristics

---

python
```python
# 2. Chroma (12 features)
chroma = np.mean(librosa.feature.chroma_stft(y=audio, sr=sr).T, axis=0)
```

Chroma - Pitch Class Profile

What it captures:

- Musical pitch content
- Tone of voice
- Intonation patterns

How it works:

- Maps all frequencies to 12 pitch classes (C, C#, D, ..., B)
- Shows which pitches are present
- Ignores octaves (C2 and C4 treated same)

Why for emotion?

- Happy: Higher pitches, varied intonation
- Sad: Lower pitches, monotone
- Angry: Harsh, intense pitches

---

python
```python
# 3. Mel-Spectrogram (128 features)
mel = np.mean(librosa.feature.melspectrogram(y=audio, sr=sr).T, axis=0)
```

Mel-Spectrogram - Energy Distribution

What it captures:

- Energy at different frequencies over time
- Voice intensity patterns
- Loudness variations

How it works:

- Splits audio into frequency bands (128 bins)
- Measures energy in each band
- Uses Mel scale for better human perception

Why for emotion?

- Angry: High energy, loud
- Sad: Low energy, quiet
- Fearful: Variable energy

---

python

```python
# 4. Spectral Contrast (7 features)
contrast = np.mean(librosa.feature.spectral_contrast(y=audio, sr=sr).T, axis=0)
```

Spectral Contrast - Texture

What it captures:

- Difference between peaks and valleys in spectrum
- Voice texture and roughness
- Clarity vs. noisiness

Why for emotion?

- Angry: High contrast (harsh, rough)
- Calm: Low contrast (smooth)

---

python

```
# 5. Tonnetz (6 features)
tonnetz = np.mean(librosa.feature.tonnetz(y=librosa.effects.harmonic(audio), sr=sr).T, axis=0)
```

Tonnetz - Harmonic Content

What it captures:

- Harmonic relationships in voice
- Musical intervals
- Voice "richness"

---

python

```
# 6. Zero-Crossing Rate (1 feature)
zcr = np.mean(librosa.feature.zero_crossing_rate(audio).T, axis=0)
```

ZCR - Signal Changes

What it captures:

- How often signal crosses zero
- Indicates noisiness vs. tonality

Why for emotion?

- High ZCR: Unvoiced sounds (whispers, harsh speech)
- Low ZCR: Voiced sounds (smooth speech)

TOTAL FEATURES: 194

```
40 (MFCC) + 12 (Chroma) + 128 (Mel) + 7 (Contrast) + 6 (Tonnetz) + 1 (ZCR) = 194
```

SECTION 4: DATASET LOADING - CREMA-D

python

```
def load_crema_data(data_path, augment=False):
    emotions_map = {
        'ANG': 'angry',
        'DIS': 'disgust',
        'FEA': 'fearful',
        'HAP': 'happy',
        'NEU': 'neutral',
        'SAD': 'sad'
    }
```

CREMA-D Filename Structure:

```
1001_DFA_ANG_XX.wav
│    │   │   │
│    │   │   └─ Intensity: XX, LO, MD, HI
│    │   └───────── Emotion: ANG, DIS, FEA, HAP, NEU, SAD
│    └───────────────── Sentence Type: DFA, IEO, etc.
└─────────────────────── Actor ID: 1001-1091
```

Loading Process:

```python
for file in glob.glob(os.path.join(data_path, "AudioWAV", "*.wav")):
    filename = os.path.basename(file)  # Get "1001_DFA_ANG_XX.wav"
    parts = filename.split("_")        # Split into ['1001', 'DFA', 'ANG', 'XX.wav']
    emotion_code = parts[2]            # Get 'ANG'
    emotion = emotions_map[emotion_code] # Convert to 'angry'
```

With Augmentation:

```
1 file → 9 augmented versions → 9 training samples
4,281 files × 9 = 38,529 total training samples
```

SECTION 5: CNN-LSTM MODEL ARCHITECTURE

```python
def create_cnn_lstm_model(input_shape, num_classes, config):
    model = models.Sequential([
        layers.Reshape((input_shape[0], 1), input_shape=input_shape),
```

Step 1: Reshape

```
Input:  [194]        → 1D array of features
Output: [194, 1]     → 2D for CNN (features, channels)
```

```python
    # CNN Block 1
    layers.Conv1D(64, 5, padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling1D(2),
    layers.Dropout(0.3),
```

CNN Block Explained:

Conv1D(64, 5)

- **64**: Number of filters (learnable patterns)
- **5**: Filter size (looks at 5 features at once)
- **ReLU**: Activation function (adds non-linearity)

What it learns:

- Local patterns in features
- Example: "High MFCC + Low Chroma = Angry?"

Visual:

```
Input features:  [f1, f2, f3, f4, f5, f6, f7, ...]
Filter (size 5): [w1, w2, w3, w4, w5]

Slide filter:
[f1, f2, f3, f4, f5] × [w1, w2, w3, w4, w5] = output1
    [f2, f3, f4, f5, f6] × [w1, w2, w3, w4, w5] = output2
        [f3, f4, f5, f6, f7] × [w1, w2, w3, w4, w5] = output3
...
```

BatchNormalization

- Normalizes layer outputs
- Speeds up training
- Reduces overfitting

MaxPooling1D(2)

- Takes maximum of every 2 values
- Reduces dimensions by half
- Keeps most important information

Dropout(0.3)

- Randomly drops 30% of neurons during training
- Prevents overfitting
- Forces model to learn robust features

```python
    # LSTM layers
    layers.LSTM(128, return_sequences=True),
    layers.Dropout(0.3),
    layers.LSTM(64),
    layers.Dropout(0.3),
```

LSTM - Long Short-Term Memory

Why LSTM after CNN?

- CNN: Extracts spatial patterns
- LSTM: Captures temporal dependencies

What it learns:

- How features change over time
- Sequences and patterns
- Context from earlier parts of speech

How LSTM works:

```
Input sequence:  [f1, f2, f3, f4, f5]
                  ↓  ↓  ↓  ↓  ↓
LSTM memory:    [h1]→[h2]→[h3]→[h4]→[h5]
                  ↓              ↓
             Hidden states    Final output
```

128 units: First LSTM captures detailed patterns **64 units**: Second LSTM summarizes information

---

python
```python
    # Dense layers
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='softmax')
```

Dense Layers - Classification

Dense(128, relu)

- Fully connected layer
- Combines all learned features
- 128 neurons for complex decision making

Final Dense(6, softmax)

- 6 outputs (one per emotion)
- Softmax: Converts to probabilities

Example Output:

```
[0.05, 0.02, 0.08, 0.78, 0.05, 0.02]
  ↓    ↓    ↓    ↓    ↓    ↓
angry disgust fear happy neutral sad

Total = 1.0 (100%)
Prediction: HAPPY (78% confident)
```

SECTION 6: TRAINING PROCESS

python
```python
def train_model(ravdess_path, tess_path, crema_path, config, augment):
    # 1. Load data
    X, y = load_all_datasets(ravdess_path, tess_path, crema_path, augment)
```

What happens:

```
4,281 audio files
   ↓ (load & augment ×9)
38,529 samples
   ↓ (extract 194 features each)
X: [38529, 194] array
y: [38529] labels ('angry', 'happy', ...)
```

---

python

```python
# 2. Encode labels
le = LabelEncoder()
y_encoded = le.fit_transform(y)
y_categorical = keras.utils.to_categorical(y_encoded)
```

Label Encoding:

```
Text labels:  ['angry', 'happy', 'sad', 'angry', ...]
              ↓ LabelEncoder
Integer:      [0, 3, 5, 0, ...]
              ↓ to_categorical
One-hot:      [[1,0,0,0,0,0],
               [0,0,0,1,0,0],
               [0,0,0,0,0,1],
               [1,0,0,0,0,0], ...]
```

Why one-hot?

- Neural networks need numerical input
- One-hot prevents ordinal relationships
- Example: 'angry'=0, 'happy'=1 doesn't mean happy > angry

---

python

```python
# 3. Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y_categorical, test_size=0.2, random_state=42
)
```

Data Split:

```
Total: 38,529 samples
  ↓
Training: 30,823 (80%) - Model learns from these
Testing:  7,706 (20%) - Model evaluated on these

Why separate test set?
→ To measure real-world performance
→ Prevents memorization
```

---

python

```python
# 4. Compile model
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Compilation Settings:

Optimizer: Adam

- Adaptive learning rate
- Fast convergence
- Good for most problems

Loss: Categorical Crossentropy

- Measures prediction error
- Formula: -Σ(true × log(predicted))
- Lower = better

Learning Rate: 0.0001

- How much to adjust weights per step
- Too high: Unstable training
- Too low: Slow training

---

python

```
# 5. Callbacks
callbacks = [
    keras.callbacks.EarlyStopping(patience=20),
    keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=7),
    keras.callbacks.ModelCheckpoint(model_save_path)
]
```

EarlyStopping

- Stops if no improvement for 20 epochs
- Prevents wasting time
- Restores best weights

ReduceLROnPlateau

- Reduces learning rate if stuck
- New LR = Current LR × 0.5
- Helps escape local minima

ModelCheckpoint

- Saves best model automatically
- Based on validation accuracy

---

python

```
# 6. Train
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32
)
```

Training Loop:

```
For each epoch (1 to 100):
    For each batch of 32 samples:
        1. Forward pass (predict)
        2. Calculate loss
        3. Backward pass (update weights)
        4. Repeat for all batches

    Validate on test set
    Print accuracy & loss
    Check callbacks
```

Batch Size = 32:

- Process 32 samples at once
- Balances speed vs. accuracy
- Total batches per epoch: 30,823 / 32 ≈ 963

SECTION 7: REAL-TIME PREDICTION

```python
class RealTimeEmotionRecognizer:
    def record_audio(self):
        stream = self.audio.open(format=self.FORMAT, channels=1, rate=22050)
        frames = []
        for _ in range(0, int(self.RATE / self.CHUNK * self.RECORD_SECONDS)):
            data = stream.read(self.CHUNK)
            frames.append(data)
        return b''.join(frames)
```

Recording Process:

```
CHUNK = 1024 samples
RATE = 22050 Hz
SECONDS = 3

Total samples needed = 22050 × 3 = 66,150
Chunks needed = 66,150 / 1024 ≈ 65 chunks

For 65 times:
  Read 1024 samples from microphone
  Store in frames
Combine all frames into audio data
```

---

python

```python
def predict_emotion(self, audio_data):
    # Save to temp file
    wf = wave.open(temp_file, 'wb')
    wf.writeframes(audio_data)

    # Preprocess
    audio, sr = preprocess_audio(temp_file)

    # Extract features
    features = extract_features(audio, sr)

    # Predict
    prediction = self.model.predict(features)
    emotion_idx = np.argmax(prediction)
    emotion = self.label_classes[emotion_idx]
```

Prediction Flow:

```
Audio bytes
   ↓ (save to .wav)
Temp file
   ↓ (preprocess)
Clean audio [66,150 samples]
   ↓ (extract features)
Features [194 values]
   ↓ (model.predict)
Probabilities [6 values]
   ↓ (argmax)
Highest emotion index
   ↓ (lookup)
Emotion name
```

🏃 PART 3: RUNNING THE TRAINING

Step 3.1: Open Command Prompt

```bash
# Press Win + R, type 'cmd', press Enter
# OR search for "Command Prompt" in Windows

# Navigate to project folder
cd C:\Users\Dennismz\Desktop\CDAC_PROJECT
```

Step 3.2: Verify Files

```bash
# Check if dataset exists
dir "DATASET TO TRAIN\CREMA-D\AudioWAV"

# Should show 4,281 .wav files
```

Step 3.3: Start Training (Basic)

```bash
python emotion_recognition.py --mode train --crema_path "DATASET TO TRAIN\CREMA-D" --config default
```

What you'll see:

```
Loading CREMA-D dataset...
Processing: 1001_DFA_ANG_XX.wav
Processing: 1001_DFA_DIS_XX.wav
...
CREMA-D: 38529 samples loaded (with augmentation)

Training samples: 30823, Test samples: 7706
Feature shape: 194
Emotion classes: ['angry' 'disgust' 'fearful' 'happy' 'neutral' 'sad']

Model Summary:
_________________________________________________________________
Layer (type)              Output Shape              Param #
=================================================================
reshape (Reshape)         (None, 194, 1)            0
conv1d (Conv1D)           (None, 194, 64)           384
...
=================================================================
Total params: 1,234,567
Trainable params: 1,234,567

Training model...
Epoch 1/100
963/963 [==============================] - 45s 47ms/step - loss: 1.7234 - accuracy: 0.3456 - val_loss: 1.5123 - val_accuracy: 0.4123
Epoch 2/100
963/963 [==============================] - 42s 44ms/step - loss: 1.4567 - accuracy: 0.4567 - val_loss: 1.3456 - val_accuracy: 0.4890
...
```

Training time estimate:

- **Default config**: 30-60 minutes
- **Optimized config**: 60-120 minutes
- Depends on CPU/GPU speed

Step 3.4: Monitor Training

Watch for:

1. **Accuracy increasing**: Should go from ~35% to 75%+
2. **Loss decreasing**: Should go from ~1.7 to ~0.5
3. **Val_accuracy**: Should be close to training accuracy

Good training:

```
Epoch 50/100
loss: 0.5234 - accuracy: 0.8234 ✓
val_loss: 0.5678 - val_accuracy: 0.7890 ✓
(Val accuracy close to train accuracy)
```

Overfitting (bad):

```
Epoch 50/100
loss: 0.2123 - accuracy: 0.9500 ✓
val_loss: 1.2345 - val_accuracy: 0.5000 ✗
(Huge gap between train and val)
```

Step 3.5: Training Complete

Output files created:

```
CDAC_PROJECT/
├── emotion_model.h5            ← Trained model
├── label_encoder_classes.npy   ← Emotion labels
├── model_config.json           ← Hyperparameters
├── training_history.png        ← Accuracy/Loss plots
└── confusion_matrix.png        ← Performance visualization
```

Final output:

```
Test Accuracy: 0.7890
Test Loss: 0.5678


Classification Report:
           precision    recall  f1-score   support


      angry       0.82      0.79      0.80      1285
    disgust       0.76      0.81      0.78      1287
    fearful       0.78      0.76      0.77      1284
      happy       0.85      0.83      0.84      1283
    neutral       0.73      0.78      0.75      1284
        sad       0.79      0.76      0.77      1283


   accuracy                           0.79      7706

Training complete! Model saved to emotion_model.h5
```

🧪 PART 4: TESTING THE MODEL

Step 4.1: Test with Command Line

```bash
python emotion_recognition.py --mode predict --model_path emotion_model.h5
```

What you'll see:

```
============================================================
🎭
```