Comprehensive Guide to CRNN Architecture for Music Emotion Recognition

This document provides a detailed explanation of the Convolutional Recurrent Neural Network (CRNN) architecture used in the Sentio project for music emotion recognition. The CRNN architecture combines the strengths of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to effectively process time-series audio data and capture both local patterns and temporal relationships.

Table of Contents

Introduction to CRNN

CRNN architectures represent a powerful fusion of two complementary deep learning approaches: CNNs excel at extracting local spatial features, while RNNs capture temporal dependencies across sequences. For music emotion recognition, this combination is particularly effective because:

1. **Music is inherently sequential** - Emotions evolve over time as the music progresses
2. **Both local and global patterns matter** - Individual patterns (chords, notes) and their progression over time contribute to emotional impact
3. **Feature hierarchy is important** - Low-level audio features combine to create higher-level musical concepts that evoke emotion

The CRNN architecture implements a "feature extraction → temporal modeling → prediction" pipeline specifically designed for time-series audio data.

Architecture Overview

Our CRNN architecture follows this general structure:

```
Input (Sequence of Audio Features)
    ↓
```

```
Convolutional Block 1 (Feature Extraction)
    ↓
Convolutional Block 2 (Higher-level Features)
    ↓
Recurrent Block (Temporal Relationships)
    ↓
Dense Layers (Dimensionality Reduction)
    ↓
Output (Valence & Arousal Predictions)
```

Each component is carefully designed to contribute to effective emotion recognition:

- **Input Layer:** Accepts time-series audio features (MFCCs, spectrograms, etc.)
- **Convolutional Blocks:** Extract meaningful patterns from audio features
- **Recurrent Blocks:** Model temporal relationships across the music sequence
- **Dense Layers:** Transform high-dimensional features into emotion predictions
- **Output Layer:** Provides valence and arousal values representing emotional content

Input Layer & Data Preprocessing

Input Shape & Format

The input to our CRNN model is a 3D tensor with shape:

`(batch_size, sequence_length, n_features)`

- **batch_size**: Number of samples processed at once (e.g., 32)
- **sequence_length**: Number of time steps per sample (e.g., 600)
- **n_features**: Number of features per time step (e.g., 40)

For our model, we typically use:

- Sequence length of 600 (representing time steps in our audio features)
- 40 features per time step (including MFCCs, spectral features, etc.)

```
# Input layer definition
from tensorflow.keras.layers import Input

inputs = Input(shape=(sequence_length, n_features), name='audio_features')
```

Data Preprocessing

Before feeding audio data into the model, we apply several preprocessing steps:

1. **Feature Extraction**

- Extract MFCCs, chroma features, spectral contrast, etc.
- Frame-level features extracted at regular intervals (typically 20-40ms)

2. **Feature Normalization**
   - Z-score normalization (mean=0, std=1) per feature
   - Ensures all features contribute equally to the model
3. **Segmentation**
   - Split longer audio files into fixed-length segments
   - Ensures consistent input dimensions for the model
4. **Label Processing**
   - Scale emotion annotations (valence and arousal) to desired range [-1, 1]
   - Handle missing values or inconsistent annotations

Convolutional Blocks

Convolutional blocks in our CRNN architecture serve to extract meaningful patterns from the raw audio features. They progressively transform low-level features into more abstract representations.

Conv1D Layers

We use 1D convolutions since our data represents a time sequence:

```
from tensorflow.keras.layers import Conv1D

# Example of a Conv1D layer
conv1 = Conv1D(
    filters=64,           # Number of output filters
    kernel_size=3,        # Size of the convolutional window
    activation='relu',    # Activation function
    padding='same',       # Padding strategy
    name='conv1'
)(inputs)
```

Key parameters:

- **filters**: Number of output channels (feature maps)
- **kernel_size**: Width of the convolutional window (in time steps)
- **activation**: Typically ReLU to introduce non-linearity
- **padding**: 'same' to maintain sequence length

Batch Normalization

After convolution operations, we apply batch normalization to:

- Stabilize and accelerate training
- Reduce internal covariate shift
- Act as a form of regularization

```
from tensorflow.keras.layers import BatchNormalization

bn1 = BatchNormalization()(conv1)
```

Max Pooling

To reduce dimensionality and focus on the most important features, we use max pooling:

```
from tensorflow.keras.layers import MaxPooling1D

pool1 = MaxPooling1D(pool_size=2)(bn1)  # Reduces sequence length by half
```

Max pooling:

- Reduces computational load
- Provides translation invariance
- Highlights dominant features

Dropout

To prevent overfitting, we apply dropout after pooling layers:

```
from tensorflow.keras.layers import Dropout

drop1 = Dropout(rate=0.2)(pool1)
```

Dropout randomly deactivates a fraction of neurons during training, forcing the network to learn redundant representations and improving generalization.

Complete Convolutional Block

A typical convolutional block in our architecture:

```
# CNN Block 1: Local pattern extraction
conv1 = Conv1D(filters=64, kernel_size=3, activation='relu', padding='same')(inputs)
bn1 = BatchNormalization()(conv1)
conv2 = Conv1D(filters=64, kernel_size=3, activation='relu', padding='same')(bn1)
pool1 = MaxPooling1D(pool_size=2)(conv2)  # 600 → 300 timesteps
drop1 = Dropout(0.2)(pool1)

# CNN Block 2: Higher-level patterns
conv3 = Conv1D(filters=128, kernel_size=3, activation='relu', padding='same')(drop1)
bn2 = BatchNormalization()(conv3)
conv4 = Conv1D(filters=128, kernel_size=3, activation='relu', padding='same')(bn2)
pool2 = MaxPooling1D(pool_size=2)(conv4)  # 300 → 150 timesteps
drop2 = Dropout(0.3)(pool2)
```

This progressive structure allows the network to learn increasingly complex patterns in the audio features.

Recurrent Blocks

After convolutional layers extract spatial features, recurrent layers model temporal relationships across the audio sequence.

LSTM Layers

We use Long Short-Term Memory (LSTM) cells because they:

- Handle long-term dependencies in the music
- Avoid the vanishing gradient problem
- Effectively model the temporal evolution of emotions

```
from tensorflow.keras.layers import LSTM

# LSTM layer that returns sequences
lstm1 = LSTM(
    units=128,                # Number of LSTM units (dimensionality of output)
    return_sequences=True,    # Return full sequence for stacking multiple LSTMs
    dropout=0.3,              # Input dropout rate
    recurrent_dropout=0.2,    # Recurrent state dropout rate
    name='lstm1'
)(drop2)

# Final LSTM layer that returns only the last output
lstm2 = LSTM(
    units=64,
    return_sequences=False,  # Return only the last output
    dropout=0.3,
    recurrent_dropout=0.2,
    name='lstm2'
)(lstm1)
```

Key parameters:

- **units**: Size of the LSTM cell's output
- **return_sequences**: Whether to return the full sequence or just the last output
- **dropout**: Regularization applied to inputs
- **recurrent_dropout**: Regularization applied to recurrent connections

Bidirectional LSTMs (Optional)

For more complex models, bidirectional LSTMs can be used to capture information from both past and future contexts:

```
from tensorflow.keras.layers import Bidirectional

bi_lstm = Bidirectional(LSTM(units=128, return_sequences=True, dropout=0.3))(drop2)
```

Bidirectional processing is particularly useful for understanding the full context of a musical piece, as emotions can be influenced by both preceding and following musical elements.

Attention Mechanism (Advanced)

For advanced models, attention mechanisms can help the model focus on the most emotionally relevant parts of the sequence:

```
from tensorflow.keras.layers import MultiHeadAttention

# Self-attention mechanism
attention_output = MultiHeadAttention(
    num_heads=8,
    key_dim=64,
    dropout=0.1
)(x, x)  # Self-attention
```

Dense Layers & Output

After extracting and processing features through convolutional and recurrent layers, dense layers map these high-dimensional features to emotion predictions.

Dense Layers

Dense layers progressively reduce dimensionality and transform features into emotion-related representations:

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.regularizers import l2

# First dense layer with L2 regularization
dense1 = Dense(
    units=64,                  # Number of neurons
    activation='relu',         # Activation function
    kernel_regularizer=l2(0.01),  # L2 regularization to prevent overfitting
    name='dense1'
)(lstm2)

# Apply dropout for regularization
drop3 = Dropout(0.4)(dense1)

# Second dense layer, further reducing dimensionality
dense2 = Dense(
    units=32,
    activation='relu',
    kernel_regularizer=l2(0.01),
```

```
    name='dense2'
)(drop3)
```

Output Layer

The final layer outputs valence and arousal predictions:

```
# Output layer for emotion prediction (valence and arousal)
outputs = Dense(
    units=2,                # Two output neurons for valence and arousal
    activation='linear',    # Linear activation for regression task
    name='emotion_output'
)(dense2)
```

We use:

- **Linear activation**: For regression task (predicting continuous emotion values)
- **2 output neurons**: One for valence (pleasure) and one for arousal (energy)

Model Compilation & Optimization

Model Creation

After defining all layers, we create the complete model:

```
from tensorflow.keras.models import Model

model = Model(inputs=inputs, outputs=outputs, name='EmotionCRNN')
model.summary()
```

Loss Functions

For emotion recognition as a regression task, we use:

Mean Squared Error (MSE) - Default Choice

```
from tensorflow.keras.losses import MeanSquaredError

# Standard MSE - treats valence and arousal equally
loss = MeanSquaredError()

# Custom weighted MSE if one dimension is more important
def weighted_mse(y_true, y_pred):
    valence_weight = 1.0  # Weight for valence prediction
    arousal_weight = 1.2  # Weight for arousal prediction (slightly higher)

    valence_loss = tf.reduce_mean(tf.square(y_true[:, 0] - y_pred[:, 0])) * valence_weight
    arousal_loss = tf.reduce_mean(tf.square(y_true[:, 1] - y_pred[:, 1])) * arousal_weight
```

```
    return valence_loss + arousal_loss
```

Huber Loss - Robust to Outliers

```
from tensorflow.keras.losses import Huber

# Huber loss is less sensitive to outliers than MSE
loss = Huber(delta=1.0)  # delta controls transition point between linear and quadratic
```

Optimizers

We primarily use Adam optimizer for its adaptive learning rates:

```
from tensorflow.keras.optimizers import Adam

# Standard Adam configuration
optimizer = Adam(
    learning_rate=0.001,    # Learning rate
    beta_1=0.9,             # Exponential decay rate for 1st moment estimates
    beta_2=0.999,           # Exponential decay rate for 2nd moment estimates
    epsilon=1e-7,           # Small constant for numerical stability
    amsgrad=False           # Whether to apply AMSGrad variant
)
```

Alternative optimizers include:

- **RMSprop**: Good for RNNs/LSTMs
- **SGD with Momentum**: Sometimes achieves better final generalization

Learning Rate Schedules

To improve training, we often use learning rate schedules:

```
from tensorflow.keras.callbacks import ReduceLROnPlateau

# Reduce learning rate when validation loss plateaus
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',      # Metric to monitor
    factor=0.5,             # Multiply LR by 0.5
    patience=5,             # Wait 5 epochs without improvement
    min_lr=1e-6,           # Don't go below this
    verbose=1              # Print when LR changes
)
```

Metrics

We track multiple metrics to evaluate model performance:

```python
from tensorflow.keras.metrics import RootMeanSquaredError, MeanAbsoluteError

# Standard regression metrics
metrics = [
    RootMeanSquaredError(name='rmse'),
    MeanAbsoluteError(name='mae'),
]

# Custom emotion metrics
def valence_mae(y_true, y_pred):
    """Mean Absolute Error for valence only"""
    return tf.reduce_mean(tf.abs(y_true[:, 0] - y_pred[:, 0]))

def arousal_mae(y_true, y_pred):
    """Mean Absolute Error for arousal only"""
    return tf.reduce_mean(tf.abs(y_true[:, 1] - y_pred[:, 1]))

def emotion_accuracy(y_true, y_pred, threshold=0.5):
    """
    Emotion prediction accuracy within threshold
    Considers prediction correct if both valence and arousal are within threshold
    """
    valence_correct = tf.abs(y_true[:, 0] - y_pred[:, 0]) <= threshold
    arousal_correct = tf.abs(y_true[:, 1] - y_pred[:, 1]) <= threshold
    both_correct = tf.logical_and(valence_correct, arousal_correct)
    return tf.reduce_mean(tf.cast(both_correct, tf.float32))
```

Hyperparameter Tuning

Key Hyperparameters

The most important hyperparameters to tune (in order of importance):

1. **Learning Rate**
   - Starting point: 0.001 (default for Adam)
   - Typical range: 0.0005 - 0.005
2. **Architecture Size**
   - LSTM units: 64, 128, 256
   - Conv1D filters: (32, 64), (64, 128), (128, 256)
   - Dense layer sizes: (32, 16), (64, 32), (128, 64)
3. **Regularization Parameters**
   - Dropout rates: 0.2, 0.3, 0.4, 0.5
   - L2 regularization: 0.001, 0.01, 0.1
   - Batch normalization momentum: 0.9, 0.99, 0.999
4. **Training Parameters**
   - Batch sizes: 16, 32, 64
   - Sequence lengths: 400, 500, 600, 800

Tuning Methods

Two common approaches for hyperparameter tuning:

1. **Grid Search** - Comprehensive but slow
   - Tests all combinations of specified hyperparameters
   - Guaranteed to find best combination within the grid
2. **Random Search** - More efficient
   - Tests random combinations of hyperparameters
   - Often finds good solutions faster than grid search
   - Better for exploring larger hyperparameter spaces

Training Strategies

Data Splitting

For our dataset of 1802 songs, we recommend:

- 70% training (1261 songs)
- 15% validation (270 songs)
- 15% testing (271 songs)

Using stratified splitting ensures balanced emotion distribution across splits.

Callbacks

We use several callbacks during training:

1. **Early Stopping**

```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=15,           # Wait 15 epochs without improvement
    restore_best_weights=True,
    verbose=1
)
```

1. **Model Checkpointing**

```
checkpoint = ModelCheckpoint(
    'best_emotion_crnn.h5',
    monitor='val_loss',
    save_best_only=True,
    verbose=1
)
```

1. **Learning Rate Reduction**

```
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
```

```
    patience=7,
    min_lr=1e-6,
    verbose=1
)
```

Evaluation Metrics

We evaluate our models using multiple metrics to get a comprehensive view of performance:

1. **Root Mean Squared Error (RMSE)**
   - Measures overall prediction error
   - Penalizes larger errors more heavily
   - Main metric for comparing models
2. **Mean Absolute Error (MAE)**
   - Average absolute difference between predicted and actual values
   - More interpretable in terms of actual emotion scale
3. **R^2 Score (Coefficient of Determination)**
   - Indicates how much variance in emotions is explained by the model
   - Range: 0 to 1 (higher is better)
   - Calculated separately for valence and arousal
4. **Emotion Accuracy**
   - Custom metric measuring predictions within a threshold
   - More intuitive for understanding practical accuracy

Advantages of CRNN for Emotion Recognition

The CRNN architecture offers several key advantages for music emotion recognition:

1. **Hierarchical Feature Learning**
   - Convolutional layers extract local patterns (notes, chords, timbres)
   - Recurrent layers model how these patterns evolve over time
   - Dense layers map these features to emotion space
2. **Time-Series Processing**
   - Properly handles the sequential nature of music
   - Captures both short-term and long-term patterns
   - Models emotional evolution through a piece
3. **Feature Abstraction**
   - Automatically learns relevant features from raw audio representations
   - Reduces need for complex hand-crafted features
   - Adapts to different musical styles and genres
4. **Joint Learning**
   - Learns valence and arousal dimensions simultaneously
   - Captures correlations between these dimensions
   - More efficient than separate models for each dimension
5. **Regularization Options**

- Multiple regularization techniques (dropout, L2, batch normalization)
- Prevents overfitting despite limited training data
- Improves generalization to new music

These advantages make the CRNN architecture particularly well-suited for the complex task of mapping music to continuous emotion dimensions, outperforming traditional approaches and simpler neural network architectures.