Human Suspicious Activity Detection from Surveillance Cameras Using LRCN

By- Antarjita Adhya
B.Tech, CSE

INTRODUCTION:

In recent years, the integration of advanced technologies in surveillance systems has become imperative for enhancing public safety and security. One such technology is the Long-term Recurrent Convolutional Networks (LRCN), a hybrid model that combines Convolutional Neural Networks (CNNs) for spatial feature extraction with Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) units, for temporal sequence modeling.

The LRCN framework is particularly effective in human suspicious activity detection from surveillance cameras. By leveraging the strengths of CNNs and RNNs, LRCNs can accurately capture and interpret both spatial and temporal information in video sequences. This allows for the identification of anomalous behaviors that may indicate suspicious activities, such as loitering, trespassing, or potential criminal actions.

The process begins with CNNs analyzing individual frames to extract critical spatial features, such as human shapes, poses, and interactions with the environment. These features are then passed to the LSTM units, which analyze the temporal dynamics across consecutive frames to detect irregular patterns over time. The combination of these two stages enables the system to understand complex activities and make informed predictions about potential threats.

Implementing LRCN-based systems in surveillance cameras offers significant advantages, including improved accuracy in activity recognition and real-time detection capabilities. This advancement enhances the ability of security personnel to respond promptly to suspicious activities, thereby preventing crimes

MOTIVATION:

Enhanced Public Safety in Crowded Spaces: Surveillance operators often face the challenge of monitoring multiple camera feeds simultaneously, which can lead to fatigue and oversight. An AI-driven system using LRCN can continuously and consistently monitor all feeds without distraction, significantly reducing the likelihood of human error and missed suspicious activities.

Proactive Incident Prevention: By detecting suspicious behavior patterns early, security systems can act proactively to prevent incidents before they escalate. For instance, identifying loitering in restricted areas or unusual movements around sensitive infrastructure can trigger preventive measures, mitigating potential threats.

Improved Public Confidence: The presence of an advanced real-time surveillance system can enhance public confidence in safety measures. Knowing that potential threats are being monitored and addressed promptly can create a sense of security and trust among the public.

The motivation for implementing real-time human suspicious activity detection using LRCN in surveillance cameras is deeply rooted in the need for enhanced security, rapid response, and efficient resource allocation. By leveraging cutting-edge AI technology, such systems aim to provide a safer environment, prevent incidents, and support law enforcement in maintaining public order and safety.

APPROACH:

Dataset Description:

The dataset used for this project consists of videos categorized into three action classes:

- **Fight**: Videos depicting aggressive confrontations or physical altercations. (From GitHub)
- **Running**: Videos showing individuals running or jogging. (Kth Action Dataset)
- **Walking**: Videos of people walking or strolling. (Kth Action Dataset)

Dataset Structure:

- **Classes**: Each action class (fight, running, walking) is represented in its own directory.
- **Files**: Each directory contains multiple video files capturing the respective actions. Each file has 100 videos.

WALKING FIGHTING RUNNING

Data Preprocessing:

1. **Frame Extraction**:
    - Videos are processed to extract a fixed number of frames (SEQUENCE_LENGTH) from each video.
    - Frames are uniformly sampled at intervals to ensure consistency in the sequence length.
2. **Resizing and Normalization**:
    - Extracted frames are resized to a standard size of IMAGE_HEIGHT x IMAGE_WIDTH (224 x 224 pixels) to maintain uniform input dimensions.
    - Frames are normalized by scaling pixel values to the range [0, 1].
3. **Data Preparation**:
    - Frames are collected into sequences, where each sequence contains SEQUENCE_LENGTH frames.
    - Sequences are used as input data for training the model.

4. **One-Hot Encoding**:
    - Labels corresponding to each video are one-hot encoded to represent the action class in a format suitable for classification.

Train-Test Split:

1. **Splitting Data**:
    - The dataset is divided into training and testing sets using an 75%-25% split.
    - train_test_split function from sklearn is used to randomly shuffle and split the data.
2. **Data Generators**:
    - Custom data generators (video_data_generator) are created to yield batches of video sequences and corresponding labels for training and testing.
    - These generators handle data loading, frame extraction, and batching during model training and evaluation.

Model Creation:

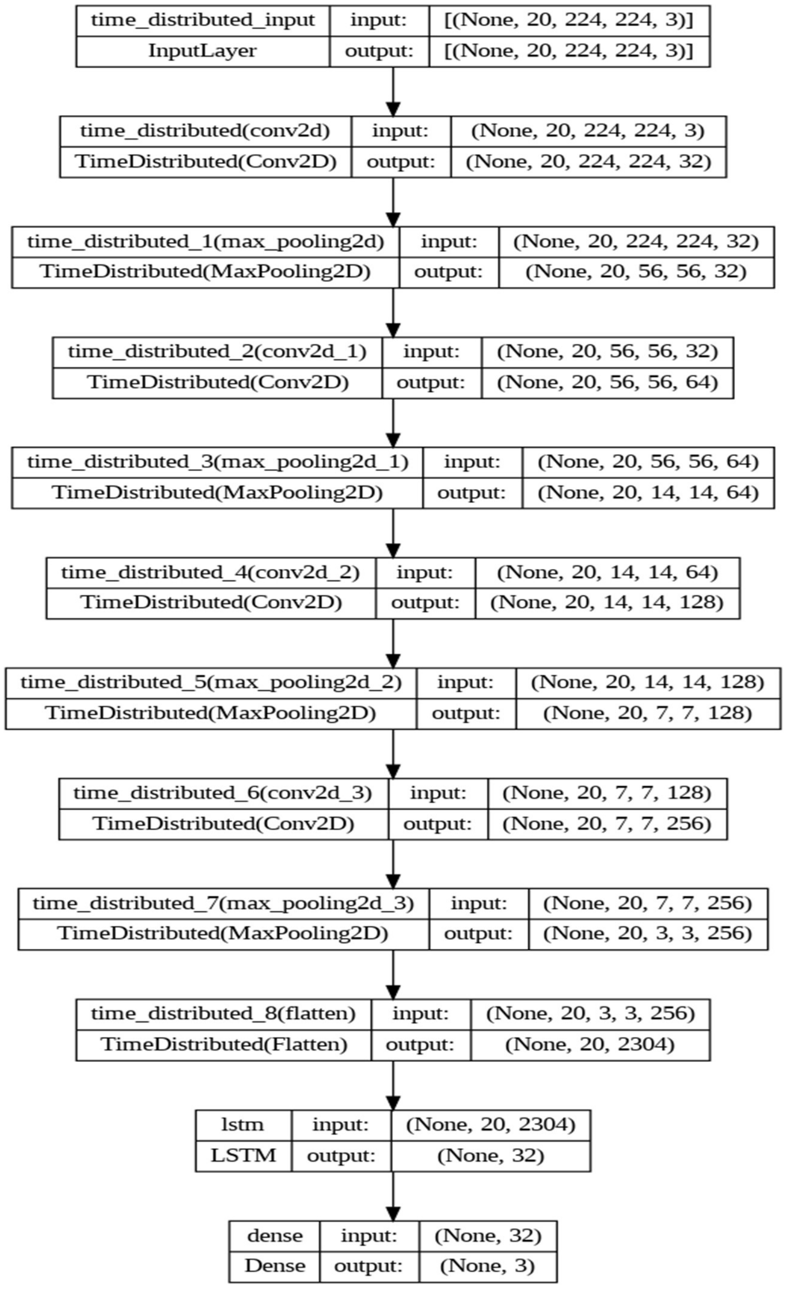
Model Overview: The code constructs a Long-term Recurrent Convolutional Network (LRCN) model using TensorFlow's Keras API. This model is designed for classifying video sequences by combining Convolutional Neural Networks (CNNs) with Recurrent Neural Networks (RNNs).

Architecture Details:

1. **Input Layer**:
    - The model starts with a TimeDistributed wrapper for applying Convolutional and MaxPooling layers to each frame of the video sequence independently.
    - **Input Shape**: (SEQUENCE_LENGTH, IMAGE_HEIGHT, IMAGE_WIDTH, 3), where SEQUENCE_LENGTH is the number of frames in the video sequence, and IMAGE_HEIGHT and IMAGE_WIDTH are the dimensions of each frame.
2. **Convolutional and Pooling Layers**:
    - **Conv2D Layers**:
        - **32 Filters**: (3, 3) kernel size, 'same' padding, ReLU activation.
        - **64 Filters**: (3, 3) kernel size, 'same' padding, ReLU activation.
        - **128 Filters**: (3, 3) kernel size, 'same' padding, ReLU activation.
        - **256 Filters**: (2, 2) kernel size, 'same' padding, ReLU activation.

- **MaxPooling2D Layers**:
    - Pooling sizes of (4, 4), (4, 4), (2, 2), and (2, 2) are used after each convolutional layer to reduce spatial dimensions.

3. **Flattening and Recurrent Layer**:
    - **Flatten**: Applied via TimeDistributed to prepare the feature maps for the LSTM layer.
    - **LSTM**: A Long Short-Term Memory (LSTM) layer with 32 units processes the sequence of frames, capturing temporal dependencies.


time_distributed_input | input: | [(None, 20, 224, 224, 3)]
InputLayer | output: | [(None, 20, 224, 224, 3)]
time_distributed(conv2d) | input: | (None, 20, 224, 224, 3)
TimeDistributed(Conv2D) | output: | (None, 20, 224, 224, 32)
time_distributed_1(max_pooling2d) | input: | (None, 20, 224, 224, 32)
TimeDistributed(MaxPooling2D) | output: | (None, 20, 56, 56, 32)
time_distributed_2(conv2d_1) | input: | (None, 20, 56, 56, 32)
TimeDistributed(Conv2D) | output: | (None, 20, 56, 56, 64)
time_distributed_3(max_pooling2d_1) | input: | (None, 20, 56, 56, 64)
TimeDistributed(MaxPooling2D) | output: | (None, 20, 14, 14, 64)
time_distributed_4(conv2d_2) | input: | (None, 20, 14, 14, 64)
TimeDistributed(Conv2D) | output: | (None, 20, 14, 14, 128)
time_distributed_5(max_pooling2d_2) | input: | (None, 20, 14, 14, 128)
TimeDistributed(MaxPooling2D) | output: | (None, 20, 7, 7, 128)
time_distributed_6(conv2d_3) | input: | (None, 20, 7, 7, 128)
TimeDistributed(Conv2D) | output: | (None, 20, 7, 7, 256)
time_distributed_7(max_pooling2d_3) | input: | (None, 20, 7, 7, 256)
TimeDistributed(MaxPooling2D) | output: | (None, 20, 3, 3, 256)
time_distributed_8(flatten) | input: | (None, 20, 3, 3, 256)
TimeDistributed(Flatten) | output: | (None, 20, 2304)
lstm | input: | (None, 20, 2304)
LSTM | output: | (None, 32)
dense | input: | (None, 32)
Dense | output: | (None, 3)

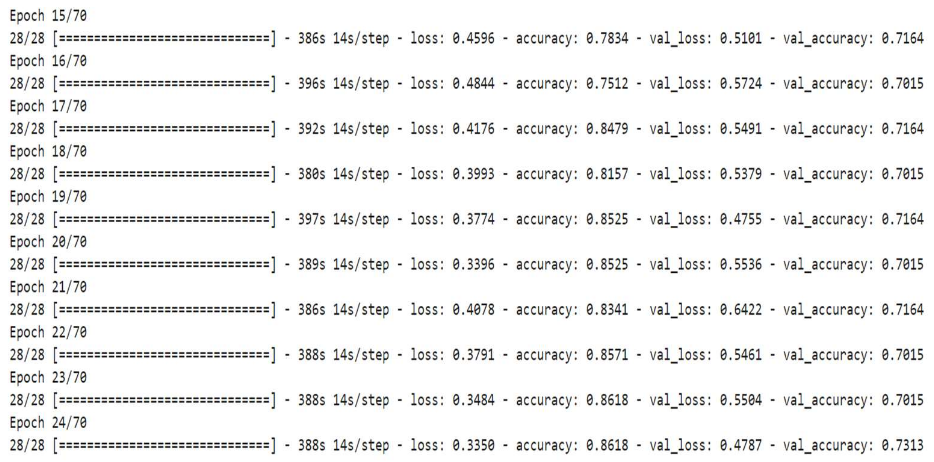
4. **Output Layer**:
    - **Dense**: A fully connected layer with softmax activation, where the number of units equals the number of action classes (len(CLASSES_LIST)), outputs class probabilities.
5. **Model Summary and Visualization**:
    - The model architecture is summarized using model.summary(), providing a detailed view of each layer's output shape and parameter count.
    - The architecture is visualized and saved as an image (lrcn_model.png) with plot_model(), showing the network structure.

MODEL TRAINING:

Training Setup: The model is trained using TensorFlow's Keras API with the following configuration:

1. **Early Stopping Callback**:
    - An EarlyStopping callback is used to monitor the validation accuracy (val_accuracy) during training.
    - **Patience**: Set to 10 epochs, meaning training will stop if no improvement is observed in validation accuracy for 10 consecutive epochs.
    - **Mode**: max, indicating that training will stop when the monitored metric (validation accuracy) reaches its maximum value.
    - **Restore Best Weights**: Ensures that the model weights are restored to the best weights observed during training when early stopping occurs.
2. **Model Compilation**:
    - **Loss Function**: categorical_crossentropy, suitable for multi-class classification problems.
    - **Optimizer**: Adam, an adaptive learning rate optimization algorithm that adjusts the learning rate during training.
    - **Metrics**: Accuracy is used to evaluate the model's performance.
3. **Model Training**:
    - **Data Generator**: Training and validation data are provided through the train_generator and test_generator, respectively, which yield batches of video frames and labels.
    - **Epochs**: The model is set to train for up to 70 epochs.
    - **Steps per Epoch**: The number of steps per epoch is defined as the number of training samples divided by the batch size.

- **Validation Steps**: The number of validation steps is defined similarly for the test data.
- **Callbacks**: The early_stopping_callback is included to manage training duration and prevent overfitting.



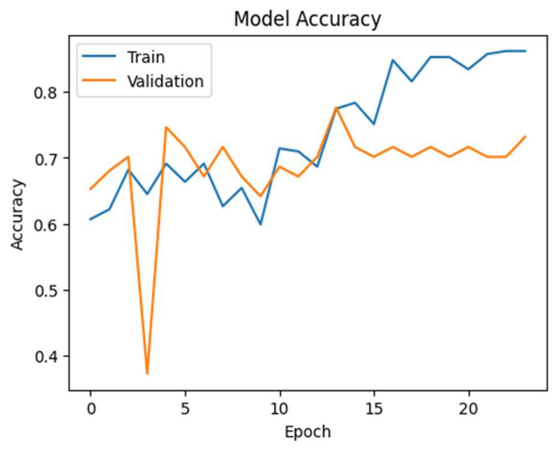
```
Epoch 15/70
28/28 [==============================] - 386s 14s/step - loss: 0.4596 - accuracy: 0.7834 - val_loss: 0.5101 - val_accuracy: 0.7164
Epoch 16/70
28/28 [==============================] - 396s 14s/step - loss: 0.4844 - accuracy: 0.7512 - val_loss: 0.5724 - val_accuracy: 0.7015
Epoch 17/70
28/28 [==============================] - 392s 14s/step - loss: 0.4176 - accuracy: 0.8479 - val_loss: 0.5491 - val_accuracy: 0.7164
Epoch 18/70
28/28 [==============================] - 380s 14s/step - loss: 0.3993 - accuracy: 0.8157 - val_loss: 0.5379 - val_accuracy: 0.7015
Epoch 19/70
28/28 [==============================] - 397s 14s/step - loss: 0.3774 - accuracy: 0.8525 - val_loss: 0.4755 - val_accuracy: 0.7164
Epoch 20/70
28/28 [==============================] - 389s 14s/step - loss: 0.3396 - accuracy: 0.8525 - val_loss: 0.5536 - val_accuracy: 0.7015
Epoch 21/70
28/28 [==============================] - 386s 14s/step - loss: 0.4078 - accuracy: 0.8341 - val_loss: 0.6422 - val_accuracy: 0.7164
Epoch 22/70
28/28 [==============================] - 388s 14s/step - loss: 0.3791 - accuracy: 0.8571 - val_loss: 0.5461 - val_accuracy: 0.7015
Epoch 23/70
28/28 [==============================] - 388s 14s/step - loss: 0.3484 - accuracy: 0.8618 - val_loss: 0.5504 - val_accuracy: 0.7015
Epoch 24/70
28/28 [==============================] - 388s 14s/step - loss: 0.3350 - accuracy: 0.8618 - val_loss: 0.4787 - val_accuracy: 0.7313
```

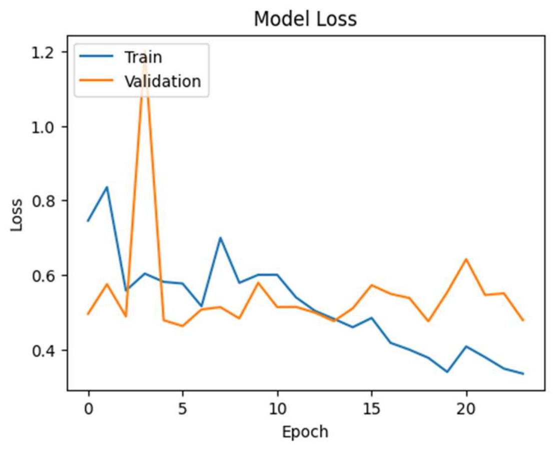
Training and Validation Metrics:

- **Accuracy Plot**: Shows training and validation accuracy over epochs, indicating model performance improvements.
- **Loss Plot**: Displays training and validation loss over epochs, illustrating how the model's loss decreases.

Test Results:

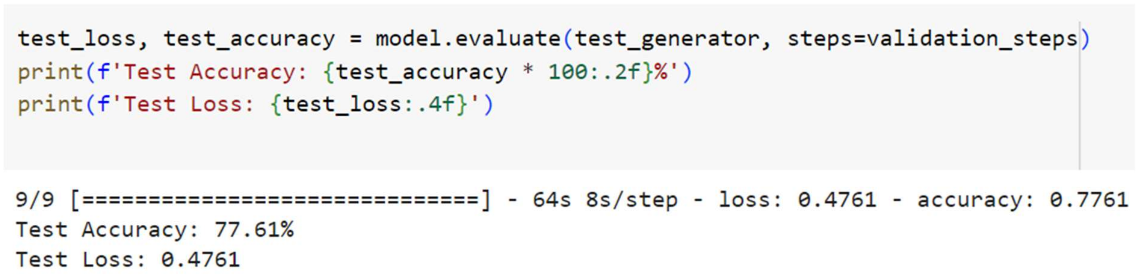
- **Test Accuracy**: 77.61%





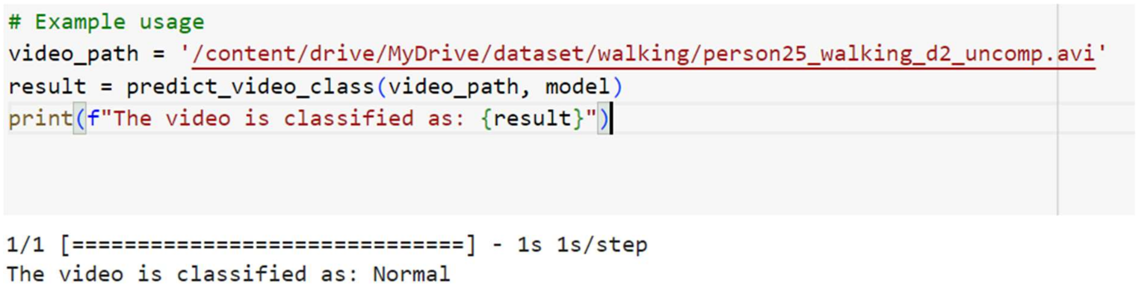
Conclusion:

The implemented code successfully loads a pre-trained LRCN model and uses it to classify video sequences. It provides 77.61% accuracy. By preprocessing video data—resizing, normalizing, and padding/truncating frames—it ensures compatibility with the model's input requirements. The predict_video_class function processes a video, makes predictions, and maps the result to a class label ("Suspicious" or "Normal"). The example demonstrates the model's capability to classify video content effectively, offering practical application for real-time activity detection and classification.



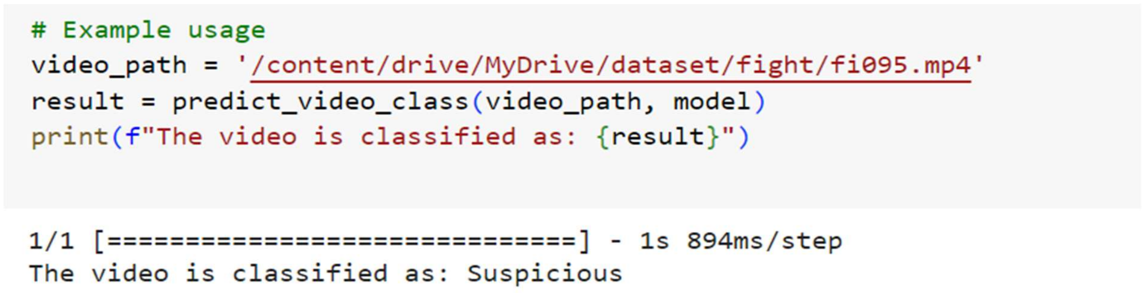
```
test_loss, test_accuracy = model.evaluate(test_generator, steps=validation_steps)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
print(f'Test Loss: {test_loss:.4f}')
```

```
9/9 [==============================] - 64s 8s/step - loss: 0.4761 - accuracy: 0.7761
Test Accuracy: 77.61%
Test Loss: 0.4761
```



```
# Example usage
video_path = '/content/drive/MyDrive/dataset/walking/person25_walking_d2_uncomp.avi'
result = predict_video_class(video_path, model)
print(f"The video is classified as: {result}")
```

```
1/1 [==============================] - 1s 1s/step
The video is classified as: Normal
```



```
# Example usage
video_path = '/content/drive/MyDrive/dataset/fight/fi095.mp4'
result = predict_video_class(video_path, model)
print(f"The video is classified as: {result}")
```

```
1/1 [==============================] - 1s 894ms/step
The video is classified as: Suspicious
```

References:

[1] Donahue, J., Anne, M., & Xie, S. (2015). "Long-term Recurrent Convolutional Networks for Visual Recognition and Description." IEEE Transactions on Pattern Analysis and Machine Intelligence.

[2] Simonyan, K., & Zisserman, A. (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition." arXiv preprint arXiv:1409.1556.

[3] Karpathy, A., Toderici, G., Shetty, S., & Leung, T. (2014). "Large-Scale Video Classification with Convolutional Neural Networks." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).