OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS FOR EDGE DEVICES

Submitted To:

Dr. Tamizharasan Periyasamy

CS F376

BITS Dubai

(SEPTEMBER 2023 - DECEMBER 2023)

Submitted By:

Name of Student	ID
SHAIK MOHAMMAD ANAS	2021A7PS0068U

Birla Institute of Technology and Science, Pilani

Dubai Campus

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my parents, whose unwavering support and encouragement have been my constant pillars of strength. I extend my heartfelt thanks to God for providing me with the wisdom, perseverance, and inspiration needed to undertake and work on this project.

I am indebted to my teachers Dr. Tamizharasan, for his invaluable guidance, mentorship, and constructive feedback. Their expertise and commitment to fostering academic growth have played a pivotal role in shaping the outcome of this project. I am grateful for the time and effort he invested in helping me develop my ideas and refine my work.

I also acknowledge Bits Pilani Dubai Campus for creating an environment conducive to academic growth. Thank you to Prof. Srinivasan Madapusi for their support and commitment to excellence, which played a crucial role in the completion of this project.

In conclusion, I would like to express my sincere appreciation to everyone who has played a role, big or small, in the realization of this project. Your support has been invaluable, and I am grateful for the opportunities and experiences that have contributed to my growth as a student and a researcher.

BITS Pilani, Dubai Campus

First Semester 2023-24

Project Course Code: CS F376
Semester: Semester 1 (AY 2023-2024)
Duration: 28/08/2023 – 26/12/2023
Date of Start: 28/08/2023
Date of Submission: 26/12/2023
Title of the Report: Optimization of Convolutional Neural Networks for Edge Devices

ID No. / Name of the student: 2021A7PS0068U / SHAIK MOHAMMAD ANAS
Discipline of Student: B.E Computer Science
Name of the Project Supervisor: Dr TAMIZHARASAN P.S.

Key Words: Convolutional Neural Networks, Optimization, Quantization, Pruning, MobileNetV2, LSTM, Memory Optimization, Violence Recognition.

Abstract: The rapid advancement of AI has led to the development of complex deep learning models that show high performance in different domains. Deploying AI models on edge devices has many advantages such as low latency, privacy and data security, bandwidth optimization, and reduced network dependence. Low latency is achieved due to real-time processing by instant data analysis on edge without waiting for remote server processing, this data analysis on the edge reduces transmitting data to the cloud which enhances security against breaches, reduces the bandwidth consumption, and reduces network dependence. Within the scope of this project, our focus extends to exploring diverse optimization techniques at the hardware level, aimed at efficiently utilizing resources such as memory and energy to create a lightweight model. Initial part of the project deals with studying about various optimization techniques such as LCRM, quantization, cascading and pruning. After the extensive research about the aforementioned techniques, a real-life scenario involving edge devices is taken up. In this project, Violence Detection through CCTV cameras has been implemented using a combination of MobileNetV2(CNN) and LSTM(RNN). Later, the optimization techniques are applied to better the results and improve resource management.

Signature of Student:
Date:

Signature of Supervisor
Date:

INDEX

TABLE OF FIGURES

Introduction

1.1 Overview

The rapid advancement of AI has led to the development of complex deep learning models that show high performance in different domains. Deploying AI models on edge devices has many advantages such as low latency, privacy and data security, bandwidth optimization, and reduced network dependence. Low latency is achieved due to real-time processing by instant data analysis on edge without waiting for remote server processing, this data analysis on the edge reduces transmitting data to the cloud which enhances security against breaches, reduces the bandwidth consumption, and reduces network dependence.

Efficient deployment of Convolutional Neural Network (CNN) models on resource-constrained devices such as IoT or handheld devices is crucial. Traditional optimization methods like quantization, pruning, and clustering have demonstrated their efficacy in compressing models at the expense of overall accuracy. Yet, the potential benefits of collaborative optimization techniques, including sparsity-preserving clustering, sparsity-preserving quantization, and cluster-preserving quantization, have not been fully investigated.

1.2 Aim & Scope

The core aim of this project is to improve the inference efficiency of CNN models by implementing a sequential optimization approach, capitalizing on the collaborative benefits of various techniques. This all-encompassing strategy seeks to find a harmonious equilibrium between compressing the model and maintaining the inherent sparsity and clustering structures, ultimately ensuring optimal performance on devices with limited resources.

Within the scope of this project, our focus extends to exploring diverse optimization techniques at the hardware level, aimed at efficiently utilizing resources such as memory and energy to create a lightweight model. In the initial phase, an in-depth study of various optimization methods was conducted, with practical applications applied to existing Convolutional Neural Network models. This allowed for a comparative analysis of results, facilitating the selection of the most suitable optimization technique.

The latter part of the project involves implementing the chosen optimization technique in a real-world use case, with the objective of achieving superior results in comparison to the existing model. This practical application not only validates the effectiveness of the chosen optimization strategy but also ensures its relevance and impact in real-world scenarios.

1.3 List of Definitions

Table 1: Convolutional neural network hyper-parameters.

Hyperparameters	Description
Kernel(Filter) Size	Kernel size of convolutional layer.
Number of feature map	Kernel count of convolutional layer.
Stride	Number of moving pixels of kernel when performing convolution.
Padding	Hyperparameters used to acquire characteristics of the border area of the training data.
Pooling type	How calculate the value for each patch on the feature map (average, Max).
Epoch	Number of learning iterations.
Learning rate	Amount of change in weight that is updated during learning.
Number of layers	Number of layers constituting entire network.
Number of neurons	Neuron count in fully-connected layer.
Batch Size	Group size to divide training data into several groups.
Weight initialization	Initialize the weights with small random numbers(Xavier initialization, He initialization).
Loss function	Function to calculate error (MSE, cross-entropy, etc.)
Optimiser	Is argument required for compiling the model (SGD, Adam, RMSprop, Adadelta, etc.)
Dropout rate	The method drops out units in neural network according to the desired probability.
Activation function	Neuron's activation function (ReLU, elu, sigmoid, etc.)

Literature Review

- **LCRM: Layer-Wise Complexity Reduction Method for CNN Model Optimization on End Devices- Hanan Hussain, P. S. Tamizharasan, and Praveen Kumar Yadav.**

The research paper proposes an optimization algorithm called Layer-wise Complexity Reduction Method (LCRM) to address these challenges by converting accuracy-focused CNNs into lightweight models. The authors evaluate the standard convolution layers and replace them with the most efficient combination of substitutional convolutions based on the output channel size. The primary goal is to reduce the computational complexity of the parent models and the hardware requirements. The effectiveness of the framework is assessed by evaluating its performance on various standard CNN models, including AlexNet, VGG-9, U-Net, and Retinex-Net, for different applications such as image classification, optical character recognition, image segmentation, and image enhancement. The experimental results show up to a 95% reduction in inference latency and up to 93% reduction in energy consumption when deployed on GPU. Furthermore, the LCRM-optimized CNN models are compared with state-of-the-art CNN optimization methods, including pruning, quantization, clustering, and their four cascaded optimization methods, by deploying them on Raspberry Pi-4. The profiling experiments performed on each model demonstrate that the LCRM-optimized CNN models achieve comparable or better accuracy than the parent models while providing added benefits such as a 62.84% reduction in inference latency on end devices with significant memory compression and complexity reductions.

- **U-Net: Convolutional Networks for Biomedical Image Segmentation- Olaf Ronneberger, Philipp Fischer, Thomas Brox.**

In this paper, the authors present a network and training strategy that relies on the strong use of data augmentation to leverage available annotated samples more efficiently. The architecture comprises a contracting path to capture context and a symmetric expanding path that enables precise localization. The authors demonstrate that such a network can be trained end-to-end from very few images and outperforms the prior best method (a sliding-window convolutional network) on the ISBI challenge for segmentation of neuronal structures in electron microscopic

stacks. Utilizing the same network trained on transmitted light microscopy images (phase contrast and DIC), the authors achieved a significant margin of victory in the ISBI cell tracking challenge 2015 in these categories. Furthermore, the network is noted for its speed, with segmentation of a 512x512 image taking less than a second on a recent GPU.

- **Very Deep Convolutional Networks for Large-Scale Image Recognition- Karen Simonyan, Andrew Zisserman**

In this study, the researchers investigate the impact of convolutional network depth on its accuracy in the context of large-scale image recognition. The primary contribution of the research is a comprehensive evaluation of networks with increasing depth, utilizing an architecture featuring small (3x3) convolution filters. The results demonstrate that a significant enhancement over prior state-of-the-art configurations can be achieved by increasing the depth to 16-19 weight layers. These findings served as the foundation for the ImageNet Challenge 2014 submission, where the research team secured the first and second places in the localization and classification tracks, respectively. The study also highlights that the representations developed in their work generalize well to other datasets, consistently achieving state-of-the-art results.

- **Robust Real-Time Violence Detection in Video Using CNN And LSTM - Al-Maamoon R. Abdali, Rana F. Al-Tuma**

Violence event detection is a critical component of surveillance systems that are essential to both public safety and law enforcement. As such, there has been a great deal of research done to improve the speed, accuracy, and generality of these detectors in a variety of video sources. Although earlier research concentrated on accuracy or speed, few examined the more general issue of compatibility with a wide range of video formats. In response, this study presents a deep learning based real-time violence detector, which uses LSTM for temporal relation learning and CNN as a spatial feature extractor. The suggested model, which emphasizes the trinity of overall generality, accuracy, and fast response time, achieves an astonishing 98% accuracy at 131 frames per second. A comparison with earlier studies on violence detection demonstrates how much better the model is in terms of accuracy and speed. The study concludes that the combined use of CNN and LSTM, leveraging transfer learning, represents the optimal approach for achieving

accuracy, robustness, and speed in violence detection tasks, particularly with limited datasets and computing resources. However, the authors acknowledge the scope for further improvement and recommend future work to explore or create well-balanced, large datasets encompassing different video sources for enhanced violence detection, expanding beyond mere binary classification to identify specific violence actions.

- **A hybrid model using 2D and 3D Convolutional Neural Networks for violence detection in a video dataset – Anusha Jayasimhan, Pabitha P'**

This work provides a hybrid CNN model for violence detection in surveillance films in the modern setting of the Internet of Things (IoT), where data from varied sources such edge devices and sensors, especially movies from CCTV devices, are readily available. The difficulty is in using deep learning algorithms, like Convolutional Neural Networks (CNNs), to video categorization in a way that takes into account the temporal aspects that two-dimensional (2D) CNNs frequently ignore. The hybrid model, which combines a 2D and 3D CNN, shows impressive speed and accuracy in identifying violence, with a maximum training accuracy of 84.5 percent and a validation accuracy of 99.93 percent. The need for automated surveillance to identify and stop violent occurrences highlights how important this effort is. The absence of publicly available datasets for violence detection emphasizes the need for accurate deep learning models capable of classifying videos based on violence-related activities. The proposed hybrid CNN strategically bridges the limitations of 2D and 3D CNNs, ensuring the extraction of both temporal and spatial features for precise and swift violence recognition. This review outlines the context, challenges, motivation, methodology, and results, offering a comprehensive understanding of the proposed model's efficacy.

- **Detecting Violence in Video Based on Deep Features Fusion Technique - Heyam Bin Jahlan, Lamiaa Elrefaei**

In the realm of surveillance systems and violence detection, there is a discernible demand for automated solutions given the surge in surveillance cameras across public spaces. This necessitates systems capable of swiftly and accurately identifying violent events. Prior studies have predominantly focused on either speed or accuracy, with limited attention to generality

across diverse video sources. The proposed work contributes to this domain by introducing a real-time violence detection model, combining Convolutional Neural Networks (CNNs) for spatial feature extraction and Long Short-Term Memory (LSTM) for temporal relation learning. This hybrid CNN model attains commendable accuracy of 98% and a speed of 131 frames per second, outperforming previous works in both accuracy and speed. The integration of a 3D CNN addresses the temporal aspect of video classification, while the proposed model achieves a balance between accuracy, speed, and generality across various video sources. The study underscores the importance of developing models that not only excel in accuracy and speed but also exhibit adaptability to diverse video inputs, thus contributing to the broader landscape of violence detection in surveillance scenarios.

- **Violence Recognition from Videos using Deep Learning Techniques – Mohamed Mostafa Soliman, Dina Khattab**

Within the domain of violence detection from videos, this study addresses the need for automated systems to recognize interpersonal violence, focusing on physical altercations. Traditional surveillance methods relying on human attention prove inadequate due to high costs and errors. In the pre-deep learning era, classical computer vision methods faced limitations. Leveraging the benefits of deep learning, this research proposes an end-to-end model integrating VGG-16 and Long Short-Term Memory for spatial and temporal feature extraction. A new benchmark dataset, Real-Life Violence Situations (RLVS), is introduced. The model achieves fine-tuned accuracies of 86.2%, 88.2%, and 84.0% on hockey fight, movie, and violent-flow datasets. The paper concludes with insights into related works, the proposed model, dataset descriptions, experimental results, and future directions.

Understanding Optimization Techniques

In the initial phase of the project, I delved into a comprehensive study of diverse resource optimization techniques and opted to assess their efficacy on existing models. I have worked on two pre-existing models to compare the results of the base model and optimized model.

1) UNET:

The U-Net architecture consists of a contracting path and an expansive path. The contracting path, also known as the encoder, captures context and reduces the spatial resolution of the input image. It typically involves a series of convolutional and pooling layers to extract hierarchical features. The expansive path, or decoder, performs upsampling to recover the spatial information and generate the segmentation mask.

One distinctive feature of the U-Net is the use of skip connections, where the output of a layer in the contracting path is concatenated with the corresponding layer in the expansive path. These skip connections help in preserving fine-grained details during the upsampling process and improve the network's ability to localize objects accurately.

U-Nets have been widely adopted in medical image analysis, particularly in tasks such as semantic segmentation of organs or tumors in medical images. They have also found applications in various other image segmentation tasks, such as satellite image analysis, cell segmentation in microscopy images, and more.

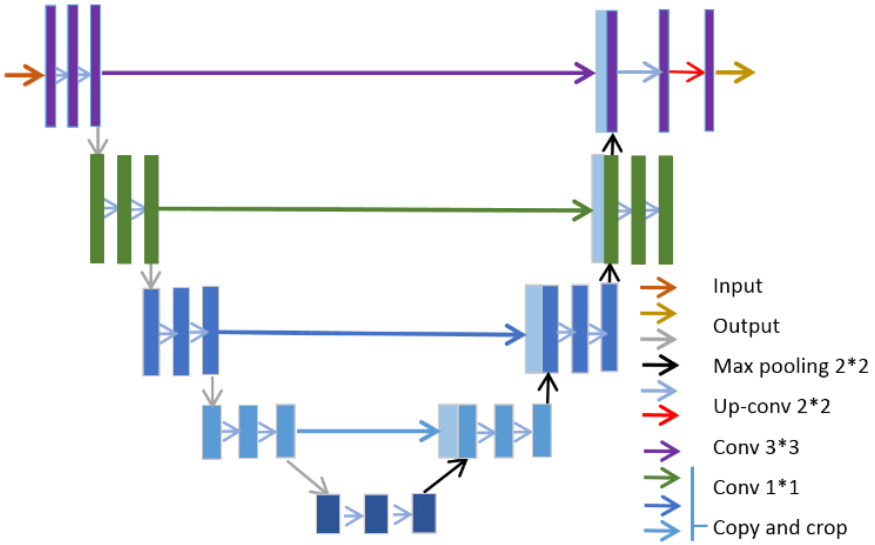


Fig. 1: U-Net Architecture

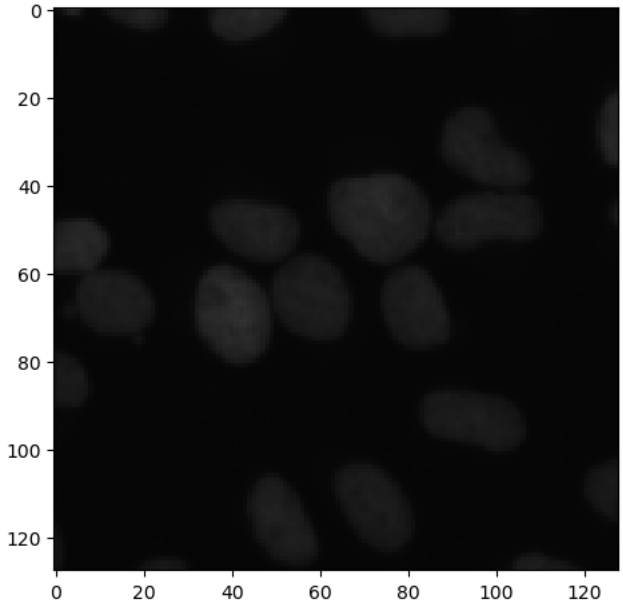
Data Preprocessing

```
print('Resizing training images and masks')
for n, id_ in tqdm(enumerate(train_ids), total=len(train_ids)):
    path = TRAIN_PATH + id_
    img = imread(path + '/images/' + id_ + '.png')[:,:,:IMG_CHANNELS]
    img = resize(img, (IMG_HEIGHT, IMG_WIDTH), mode='constant', preserve_range=True)
    X[n] = img  #Fill empty X_train with values from img
    #merging the masks images
    mask = np.zeros((IMG_HEIGHT, IMG_WIDTH, 1), dtype=bool)
    for mask_file in next(os.walk(path + '/masks/'))[2]:
        mask_ = imread(path + '/masks/' + mask_file)
        mask_ = np.expand_dims(resize(mask_, (IMG_HEIGHT, IMG_WIDTH), mode='constant',
                                      preserve_range=True), axis=-1)
        mask = np.maximum(mask, mask_)
    #y is the masked images.
    y[n] = mask

# test images
test_images = np.zeros((len(test_ids), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS), dtype=np.uint8)
sizes_test = []
print('Resizing test images')
for n, id_ in tqdm(enumerate(test_ids), total=len(test_ids)):
    path = TEST_PATH + id_
    img = imread(path + '/images/' + id_ + '.png')[:,:,:IMG_CHANNELS]
    sizes_test.append([img.shape[0], img.shape[1]])
    img = resize(img, (IMG_HEIGHT, IMG_WIDTH), mode='constant', preserve_range=True)
    test_images[n] = img

X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.33, random_state=42)
print('Split success!')
```

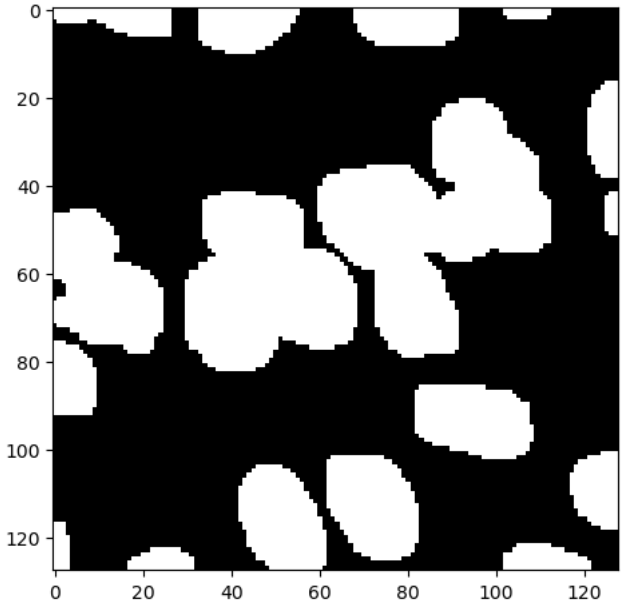
The training data consists of images and masks stored in specified directories (TRAIN_PATH). The code iterates through each training image (id_) and loads the image, resizing it to a specified height and width (IMG_HEIGHT and IMG_WIDTH). The corresponding masks are then merged into a single mask for each image, with pixel values indicating the presence or absence of certain features. The training images (X) and their corresponding masks (y) are stored in arrays.

Similarly, for the test dataset, images are loaded and resized, with their dimensions recorded. The resulting test images are stored in the test_images array.

Lastly, the code splits the training data into training and testing sets using the train_test_split function from scikit-learn, allocating 33% of the data for testing. This is a common practice in machine learning to assess model performance on unseen data. The success of the split is printed as confirmation.



X_train Image



Y_train Masked image

Model Implementation

1) *UNET(Base model)*

```
inputs = tf.keras.layers.Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))

#Contracting Path
s = tf.keras.layers.Lambda(lambda x: x/255)(inputs)

c1 = tf.keras.layers.Conv2D(16, (3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(s)
c1 = tf.keras.layers.Dropout(0.1)(c1)
c1 = tf.keras.layers.Conv2D(16, (3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c1)
p1 = tf.keras.layers.MaxPooling2D((2,2))(c1)

c2 = tf.keras.layers.Conv2D(32,(3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(p1)
c2 = tf.keras.layers.Dropout(0.1)(c2)
c2 = tf.keras.layers.Conv2D(32,(3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c2)
p2 = tf.keras.layers.MaxPooling2D((2,2))(c2)

c3 = tf.keras.layers.Conv2D(64,(3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(p2)
c3 = tf.keras.layers.Dropout(0.2)(c3)
c3 = tf.keras.layers.Conv2D(64,(3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c3)
p3 = tf.keras.layers.MaxPooling2D((2,2))(c3)

c4 = tf.keras.layers.Conv2D(128,(3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(p3)
c4 = tf.keras.layers.Dropout(0.2)(c4)
c4 = tf.keras.layers.Conv2D(128,(3,3), activation = 'relu', kernel_initializer = 'he_normal', padding = 'same')(c4)
p4 = tf.keras.layers.MaxPooling2D((2,2))(c4)

c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p4)
c5 = tf.keras.layers.Dropout(0.3)(c5)
c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c5)
```

```
#Expansive Path

u6 = tf.keras.layers.Conv2DTranspose(128,                                    ')(c5)
u6 = tf.keras.layers.concatenate([u6,c4],
c6 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(u6)
c6 = tf.keras.layers.Dropout(0.2)(c6)
c6 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c6)

u7 = tf.keras.layers.Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c6)
u7 = tf.keras.layers.concatenate([u7, c3], axis=3)
c7 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(u7)
c7 = tf.keras.layers.Dropout(0.2)(c7)
c7 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c7)

u8 = tf.keras.layers.Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(c7)
u8 = tf.keras.layers.concatenate([u8, c2], axis=3)
c8 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(u8)
c8 = tf.keras.layers.Dropout(0.1)(c8)
c8 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c8)

u9 = tf.keras.layers.Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(c8)
u9 = tf.keras.layers.concatenate([u9, c1], axis=3)
c9 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(u9)
c9 = tf.keras.layers.Dropout(0.1)(c9)
c9 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c9)

outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(c9)

model = tf.keras.Model(inputs=[inputs], outputs=[outputs])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

(parameter) activation: Any | None

Model summary:

```
Total params: 1941105 (7.40 MB)
Trainable params: 1941105 (7.40 MB)
Non-trainable params: 0 (0.00 Byte)
```

2) *UNET(with Batch Norm)*

The key idea behind Batch Normalization is to normalize the inputs of a layer by subtracting the mean and dividing by the standard deviation of the mini-batch. This is applied independently to each feature in the input. The normalization is followed by scaling and shifting operations, which introduce learnable parameters, allowing the model to adapt and potentially undo the normalization.

```
inputs = tf.keras.layers.Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))

#Contraction path
c1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(inputs)
c1 = tf.keras.layers.Dropout(0.1)(c1)
c1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c1)
b1 = tf.keras.layers.BatchNormalization()(c1)
r1 = tf.keras.layers.ReLU()(b1)
p1 = tf.keras.layers.MaxPooling2D((2, 2))(r1)

c2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p1)
c2 = tf.keras.layers.Dropout(0.1)(c2)
c2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c2)
b2 = tf.keras.layers.BatchNormalization()(c2)
r2 = tf.keras.layers.ReLU()(b2)
p2 = tf.keras.layers.MaxPooling2D((2, 2))(r2)

c3 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p2)
c3 = tf.keras.layers.Dropout(0.2)(c3)
c3 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c3)
b3 = tf.keras.layers.BatchNormalization()(c3)
r3 = tf.keras.layers.ReLU()(b3)
p3 = tf.keras.layers.MaxPooling2D((2, 2))(r3)

c4 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p3)
c4 = tf.keras.layers.Dropout(0.2)(c4)
c4 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c4)
b4 = tf.keras.layers.BatchNormalization()(c4)
r4 = tf.keras.layers.ReLU()(b4)
p4 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(r4)

c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p4)
b5 = tf.keras.layers.BatchNormalization()(c5)
r5 = tf.keras.layers.ReLU()(b5)
c5 = tf.keras.layers.Dropout(0.3)(r5)
c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c5)

#Expansive path
u6 = tf.keras.layers.Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c5)
u6 = tf.keras.layers.concatenate([u6, c4])
u6 = tf.keras.layers.BatchNormalization()(u6)
u6 = tf.keras.layers.ReLU()(u6)


u7 = tf.keras.layers.Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(u6)
u7 = tf.keras.layers.concatenate([u7, c3])
u7 = tf.keras.layers.BatchNormalization()(u7)
u7 = tf.keras.layers.ReLU()(u7)


u8 = tf.keras.layers.Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(u7)
u8 = tf.keras.layers.concatenate([u8, c2])
u8 = tf.keras.layers.BatchNormalization()(u8)
u8 = tf.keras.layers.ReLU()(u8)

u9 = tf.keras.layers.Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(u8)
u9 = tf.keras.layers.concatenate([u9, c1], axis=3)
u9 = tf.keras.layers.BatchNormalization()(u9)
u9 = tf.keras.layers.ReLU()(u9)


outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(u9)

model2 = tf.keras.Model(inputs=[inputs], outputs=[outputs])
model2.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Model Summary:

```
Total params: 1400033 (5.34 MB)
Trainable params: 1398081 (5.33 MB)
Non-trainable params: 1952 (7.62 KB)
```

We observe that the number of trainable parameters has reduced and the memory associated with it too.

Model training:

The chosen optimizer is Adam, a popular optimization algorithm, and the loss function is set to binary crossentropy, which is commonly used for binary classification tasks. The model is configured to monitor accuracy as a metric during training. The training process is initiated using the fit method, where the training data (X_train and Y_train) is specified along with validation data (X_test and Y_test). The batch size for each training iteration is set to 128, and the training is scheduled for 25 epochs. To prevent overfitting and enable early stopping, the training incorporates callbacks. Specifically, the EarlyStopping callback is configured to halt training if the validation loss does not improve after three consecutive epochs, while the TensorBoard callback is employed to log training metrics for visualization. The model training progress and relevant information are saved in a ‘logs’ directory.

```
Epoch 25/25
4/4 [==============================] - 17s 4s/step - loss: 0.1531 - accuracy: 0.9589 - val_loss: 0.2165 - val_accuracy: 0.9502
```

The performance of the models accuracy can be measured using the metric called Intersection over Union.

```
intersection = np.logical_and(sample_mask, predicted_mask)
union = np.logical_or(sample_mask, predicted_mask)
iou_score = np.sum(intersection) / np.sum(union)
print(iou_score)
```

```
0.741679389312977
```

3) *UNET(with LCRM)*

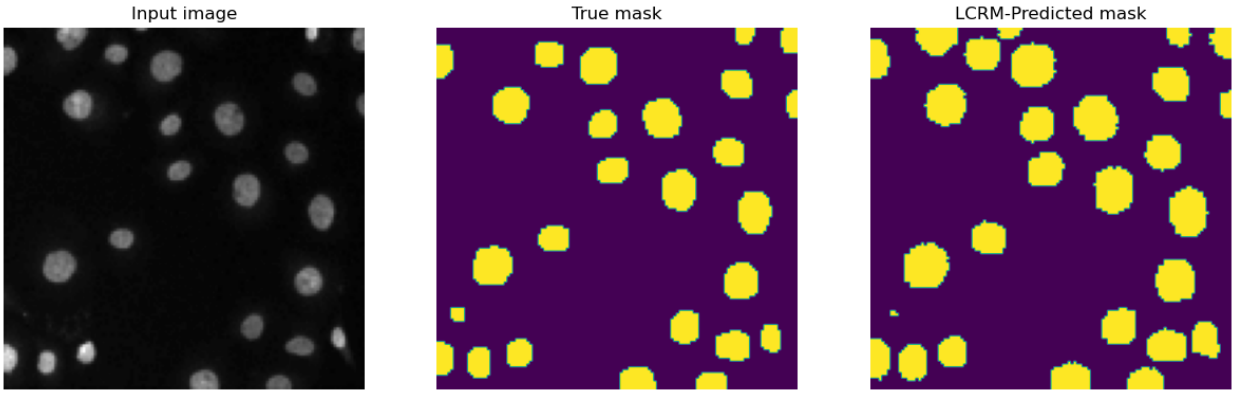
```
inputs = tf.keras.layers.Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))

#contraction path
c1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(inputs)
c1 = tf.keras.layers.Dropout(0.1)(c1)
#c1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c1)
c1 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(c1)
# b1 = tf.keras.layers.BatchNormalization()(c1)
r1 = tf.keras.layers.ReLU()(c1)
p1 = tf.keras.layers.MaxPooling2D((2, 2))(r1)


# c2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p1)
c2 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(p1)
c2 = tf.keras.layers.Dropout(0.1)(c2)
# c2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c2)
c2 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(c2)
# b2 = tf.keras.layers.BatchNormalization()(c2)
r2 = tf.keras.layers.ReLU()(c2)
p2 = tf.keras.layers.MaxPooling2D((2, 2))(r2)

# c3 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p2)
c3 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(p2)
c3 = tf.keras.layers.Dropout(0.2)(c3)
# c3 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c3)
c3 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(c3)
# b3 = tf.keras.layers.BatchNormalization()(c3)
r3 = tf.keras.layers.ReLU()(c3)
p3 = tf.keras.layers.MaxPooling2D((2, 2))(r3)

# c4 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p3)
c4 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(p3)
c4 = tf.keras.layers.Dropout(0.2)(c4)
# c4 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c4)
c4 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(c4)
# b4 = tf.keras.layers.BatchNormalization()(c4)
r4 = tf.keras.layers.ReLU()(c4)
p4 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(r4)

# c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(p4)
c5 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(p4)
# b5 = tf.keras.layers.BatchNormalization()(c5)
r5 = tf.keras.layers.ReLU()(c5)
c5 = tf.keras.layers.Dropout(0.3)(r5)
# c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal', padding='same')(c5)
c5 = tf.keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size = 3,strides = (1,1),padding='same',activation = 'relu')(c5)


#Expansive path
u6 = tf.keras.layers.Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c5)
u6 = tf.keras.layers.concatenate([u6, c4])
# u6 = tf.keras.layers.BatchNormalization()(u6)
u6 = tf.keras.layers.ReLU()(u6)

u7 = tf.keras.layers.Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(u6)
u7 = tf.keras.layers.concatenate([u7, c3])
# u7 = tf.keras.layers.BatchNormalization()(u7)
u7 = tf.keras.layers.ReLU()(u7)

u8 = tf.keras.layers.Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(u7)
u8 = tf.keras.layers.concatenate([u8, c2])
# u8 = tf.keras.layers.BatchNormalization()(u8)
u8 = tf.keras.layers.ReLU()(u8)

u9 = tf.keras.layers.Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(u8)
u9 = tf.keras.layers.concatenate([u9, c1], axis=3)
# u9 = tf.keras.layers.BatchNormalization()(u9)
u9 = tf.keras.layers.ReLU()(u9)

outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(u9)
model3 = tf.keras.Model(inputs = [inputs], outputs = [outputs])
```

Model Summary:

```
Total params: 227569 (888.94 KB)
Trainable params: 227569 (888.94 KB)
Non-trainable params: 0 (0.00 Byte)
```

Model Training

```
Epoch 22/50
28/28 [==============================] - 8s 298ms/step - loss: 0.1215 - accuracy: 0.9537 - val_loss: 0.1236 - val_accuracy: 0.9506
```



The results show that we are getting the same level (or in some cases better performance) than the base model with added benefits like reduced memory consumption.

2) **VGGNet:**

VGGNet, short for Visual Geometry Group Network, is a deep convolutional neural network architecture proposed by researchers Karen Simonyan and Andrew Zisserman from the University of Oxford. Introduced in 2015, VGGNet is characterized by its straightforward and uniform design. The architecture is notably deep, consisting of either 16 or 19 layers, making it one of the early deep neural networks. One of the distinctive features of VGGNet is its consistent use of 3x3 convolutional filters throughout the network, along with max-pooling layers for spatial dimension reduction. The stacking of multiple convolutional layers contributes to the depth, and the network culminates in fully connected layers with softmax activation for classification purposes. Despite its computational demands compared to more recent architectures, VGGNet played a pivotal role in advancing deep learning for image recognition. Different versions, such as VGG16 and VGG19, have become benchmarks in computer vision, often serving as baseline models in various applications, including image classification and feature extraction.

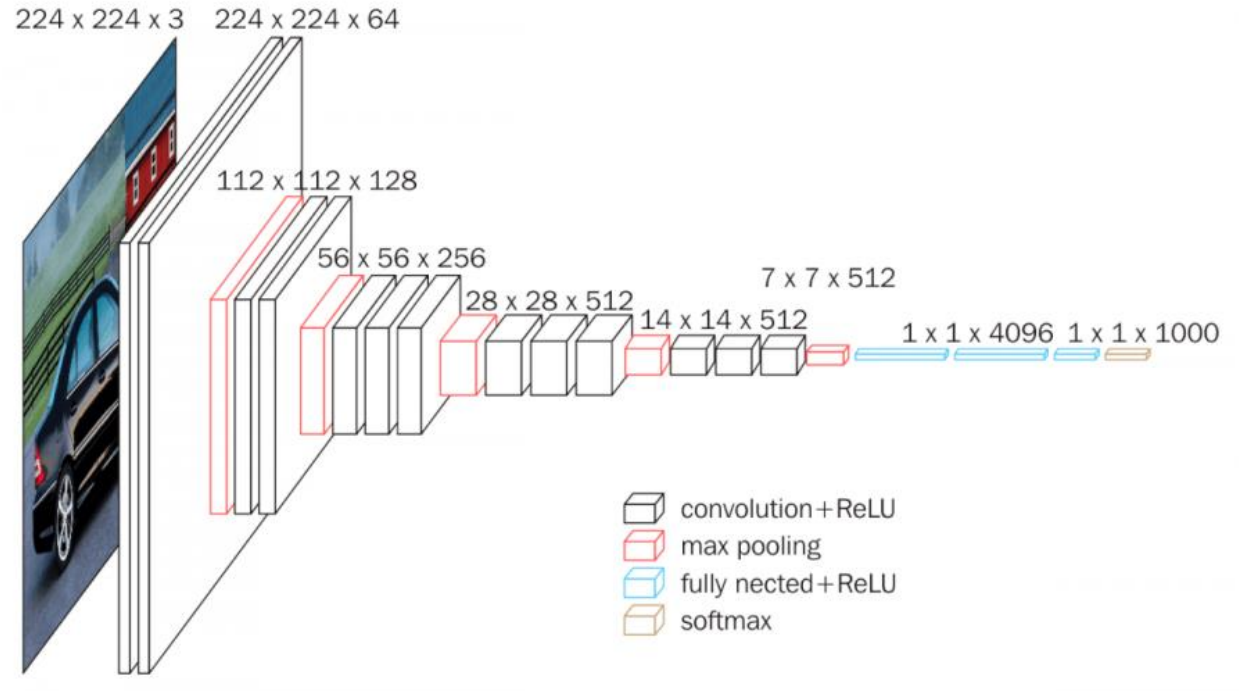


Fig. 2: VGG-Net Architecture

Data Preprocessing

```
def normalize_img(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  return tf.cast(image, tf.float32) / 255., label


def process_images(image, label):
    image = tf.image.resize(image, (224,224))
    return image, label

ds_train = train.map(normalize_img)
ds_train = train.map(process_images)

ds_test = test.map(normalize_img)
ds_test = test.map(process_images)
```

The normalize_img function takes an image and its corresponding label as input and converts the pixel values from the uint8 data type to float32, scaling them between 0 and 1 by dividing each pixel value by 255. This normalization is a common practice to bring pixel values into a range suitable for neural network training. The process_images function is designed to resize the images to a fixed size of 224x224 pixels using TensorFlow's tf.image.resize function.

Subsequently, these preprocessing functions are applied to both the training (train) and testing (test) datasets using the map method. The training dataset (ds_train) undergoes both normalization and resizing operations in sequence, first mapping the normalize_img function and then the process_images function. The same preprocessing pipeline is applied to the testing dataset (ds_test).

Model Building

1) *VGG9 (Base Model):*

```
model1= keras.models.Sequential([
  keras.layers.Conv2D(filters=64, kernel_size=(11,11), strides=(4,4), input_shape=(224,224,1)),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),


  #keras.layers.DepthwiseConv2D(depth_multiplier=8,kernel_size=5,strides=(1,1),padding='same'),
  keras.layers.Conv2D(filters=128, kernel_size=(5,5), strides=(1,1), padding="same"),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),


  #keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding="same"),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),

  keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding="same"),
  #keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),

  keras.layers.Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding="same"),
  #keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),

  keras.layers.Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding="same"),
  #keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
  #keras.layers.Conv2D(filters=nclass,kernel_size = (1,1), activation='relu', padding="same"), #add number of classes here

  keras.layers.Flatten(),

  keras.layers.Dense(1024, activation='relu'),
  keras.layers.Dropout(0.5),

  keras.layers.Dense(1024, activation='relu'),
  keras.layers.Dropout(0.5),

  keras.layers.Dense(100, activation='softmax')

  ])
```

Model Summary:

```
Total params: 7895140 (30.12 MB)
Trainable params: 7891684 (30.10 MB)
Non-trainable params: 3456 (13.50 KB)
```

2) VGG9(with LCRM)

```
model2 = keras.models.Sequential([
  keras.layers.Conv2D(filters=64, kernel_size=(11,11), strides=(4,4), input_shape=(224,224,1)),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),


  keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size=5,strides=(1,1),padding='same'),
#   keras.layers.Conv2D(filters=128, kernel_size=(5,5), strides=(1,1), padding="same"),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),


  keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size=3,strides=(1,1),padding='same'),
  #keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding="same"),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),

  #keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding="same"),
  keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),

 # keras.layers.Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding="same"),
  keras.layers.DepthwiseConv2D(depth_multiplier=2,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),

  #keras.layers.Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding="same"),
  keras.layers.DepthwiseConv2D(depth_multiplier=1,kernel_size=3,strides=(1,1),padding='same'),
  keras.layers.BatchNormalization(),
  keras.layers.Activation('relu'),
  keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
  #keras.layers.Conv2D(filters=nclass,kernel_size = (1,1), activation='relu', padding="same"), #add number of classes here
  keras.layers.Flatten(),

  keras.layers.Dense(1024, activation='relu'),
  keras.layers.Dropout(0.5),

  keras.layers.Dense(1024, activation='relu'),
  keras.layers.Dropout(0.5),

  keras.layers.Dense(62, activation='softmax')

  ])
```

Model Summary:

```
...
Total params: 3244734 (12.38 MB)
Trainable params: 3241278 (12.36 MB)
Non-trainable params: 3456 (13.50 KB)
```

As VGGNet is a huge network to train and requires a lot of computation, I have not run the training and model fitting algorithm on it. But according to the results in the above-mentioned papers, VGG9 LCRM performs as good as the base model.

Implementation of Optimization Techniques

PeaceGuard: Violence Detection Platform

Introduction

"PeaceGuard: Violence Detection Platform" is a comprehensive application intended for the quick analysis and detection of violent material present in footage. The software uses deep learning techniques to evaluate video frames and identify possible instances of hostility or violence by utilizing cutting-edge machine learning models. Uploading MP4, AVI, or MKV video files allows users to use the application, which then processes and analyzes the material to forecast whether or not violence would be there.

Since this application will be used in resource constrained devices, optimisation is highly required for the faster training and detection time, less memory consumption and reduction in number of Floating-Point operations (FLOPS)[6]. For training the neural network architecture, i.e. **MoBiLSTM** (MobileNet V2 paired with Bi-directional LSTM), many state-of-the-art optimization techniques such as LCRM (Layer wise Complexity Reduction) [1], quantization [10], pruning etc. were used. The idea of using MobileNet V2 as the CNN component in this architecture was due to the results obtained in the LCRM paper. It shows that using Depthwise Separable and Pointwise Separable Convolutional Layers reduce the number of convolutions i.e. filter multiplications, hence reducing the resource (CPU, GPU) and memory consumption. The CNN, MobileNet acts as a spatial feature extractor and these features are fed to the Bi-Directional LSTM for temporal feature extraction. This is followed by number of Fully Connected Layers for binary classification. While training, techniques like quantization and Pruning were applied to decrease the training time, while keeping the accuracy almost constant. After obtaining desirable results, the model was saved as an .h5 file to be reused. An application was developed using Streamlit to test the model's functionality.

Methodology

Importing Necessary Libraries

```
import os
import shutil
import cv2
import math
import random
import numpy as np
import datetime as dt
import tensorflow
import keras
from collections import deque
import matplotlib.pyplot as plt
plt.style.use("seaborn")
import tensorflow as tf

%matplotlib inline

from sklearn.model_selection import train_test_split

from keras.layers import *
from keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from keras.callbacks import EarlyStopping
from tensorflow.keras.utils import plot_model
from keras.preprocessing.image import ImageDataGenerator, img_to_array,load_img
```

Visualizing the Data

The dataset consists of 1,000 movies showing violent incidents and another 1,000 showing peaceful scenes. Street fight scenes in a variety of settings and circumstances are accurately captured in the violent footage in their dataset. The peaceful movies, which are compiled from multiple sources, simultaneously present a range of human activities like eating, strolling, and playing sports.

```
from IPython.display import HTML
from base64 import b64encode

# To Show a Video in Notebook
def Play_Video(filepath):
    html = ''
    video = open(filepath,'rb').read()
    src = 'data:video/mp4;base64,' + b64encode(video).decode()
    html += '<video width=640 muted controls autoplay loop><source src="%s" type="video/mp4"></video>' % src
    return HTML(html)
```

This Python code defines a function called `Play_Video(filepath)` for displaying a video in a Jupyter notebook. It uses the IPython library to generate an HTML code snippet containing a

video player. The video file specified by the `filepath` parameter is read, encoded in base64 format, and embedded in the HTML video element. The resulting HTML code is then rendered using the `HTML` class from IPython.display, allowing users to play the video directly within the notebook. The video is set to autoplay, loop, and includes controls for user interaction.

```
# Classes Directories
NonViolence_dir = "/Users/anasshaik/Documents/Design Project/Violence Detection/data/rlvs/NonViolence"
Violence_dir = "/Users/anasshaik/Documents/Design Project/Violence Detection/data/rlvs/Violence"

# Retrieve the list of all the video files present in the Class Directory.
NonViolence_files_names_list = os.listdir(NonViolence_dir)
Violence_files_names_list = os.listdir(Violence_dir)

# Randomly select a video file from the Classes Directory.
Random_NonViolence_Video = random.choice(NonViolence_files_names_list)
Random_Violence_Video = random.choice(Violence_files_names_list)
```

Frame Extraction

```
# Specify the height and width to which each video frame will be resized in our dataset.
IMAGE_HEIGHT , IMAGE_WIDTH = 64, 64

# Specify the number of frames of a video that will be fed to the model as one sequence.
SEQUENCE_LENGTH = 16

dataset_path = '/Users/anasshaik/Documents/Design Project/Violence Detection/data/rlvs/'
label_list = ["NonViolence","Violence"]
```

```
def frames_extraction(video_path):

    frames_list = []

    # Read the Video File
    video_reader = cv2.VideoCapture(video_path)

    # Get the total number of frames in the video.
    video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate the the interval after which frames will be added to the list.
    skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH), 1)

    # Iterate through the Video Frames.
    for frame_counter in range(SEQUENCE_LENGTH):

        # Set the current frame position of the video.
        video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window)

        # Reading the frame from the video.
        success, frame = video_reader.read()

        if not success:
            break

        # Resize the Frame to fixed height and width.
        resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))

        # Normalize the resized frame
        normalized_frame = resized_frame / 255

        # Append the normalized frame into the frames list
        frames_list.append(normalized_frame)


    video_reader.release()

    return frames_list
```

A function named **`frames_extraction()`** has been created to generate a list comprising resized and normalized frames from a specified video path. The video file is iteratively read frame by frame, and only an evenly distributed sequence length of frames is added to the list. This was achieved with the help of a library called **OpenCV.**

Creating the Data

```
def create_dataset():

    features = []
    labels = []
    video_files_paths = []

    # Iterating through all the classes.
    for class_index, class_name in enumerate(label_list):

        print(f'Extracting Data of Class: {class_name}')

        # Get the list of video files present in the specific class name directory.
        files_list = os.listdir(os.path.join(dataset_path, class_name))

        # Iterate through all the files present in the files list.
        for file_name in files_list:

            # Get the complete video path.
            video_file_path = os.path.join(dataset_path, class_name, file_name)

            # Extract the frames of the video file.
            frames = frames_extraction(video_file_path)

            # Check if the extracted frames are equal to the SEQUENCE_LENGTH specified.
            # So ignore the vides having frames less than the SEQUENCE_LENGTH.
            if len(frames) == SEQUENCE_LENGTH:

                # Append the data to their repective lists.
                features.append(frames)
                labels.append(class_index)
                video_files_paths.append(video_file_path)

    features = np.asarray(features)
    labels = np.array(labels)
    return features, labels, video_files_paths
```

A function named **create_dataset()** has been defined, which iterates through all the classes specified in the **labels_list** list. For each class, it invokes the **frame_extraction()** function on every video file associated with the selected classes. The function returns the **frames (features)**, **class index (labels)**, and **video file path (video_files_paths)**.

```
# Saving the extracted data
np.save("features.npy",features)
np.save("labels.npy",labels)
np.save("video_files_paths.npy",video_files_paths)
```

```
features, labels, video_files_paths = np.load("features.npy") , np.load("labels.npy") ,  np.load("video_files_paths.
```

```
one_hot_encoded_labels = to_categorical(labels)
```

The arguments obtained from the function called are saved locally for preservation. At the same time, the target variable i.e. label is encoded (**one-hot encoding**).

Data Augmentation

Splitting The data

```
X_train, X_test, y_train, y_test = train_test_split(features, one_hot_encoded_labels, test_size = 0.25,
                                                    shuffle = True, random_state = 42)
```

```
print(X_train.shape,y_train.shape )
print(X_test.shape, y_test.shape)
```

```
(1500, 16, 64, 64, 3) (1500, 2)
(500, 16, 64, 64, 3) (500, 2)
```

Data Augmentation

```
trainAug = ImageDataGenerator(
    horizontal_flip = True,
    rotation_range = 30,
    zoom_range = 0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.15,
    fill_mode="nearest",
    rescale=1./255 #normalizing the pixel values to range [0,1]
)
```

```
valAug = ImageDataGenerator(rescale=1./255)
```

```
# define the ImageNet mean subtraction (in RGB order) and set the
# the mean subtraction value for each of the data augmentation
# objects
mean = np.array([123.68, 116.779, 103.939], dtype="float32")
trainAug.mean = mean
valAug.mean = mean
```

After the splitting of data into training and testing data, an instance of the `ImageDataGenerator` class, which is frequently used for data augmentation in image-based machine learning applications, especially during the training phase, is created and represented by the `trainAug' variable. In order to produce a training dataset that is more robust and diversified, data augmentation entails adding several changes to the original images. With this particular configuration, the generator can flip horizontally, rotate randomly within a 30-degree range, zoom randomly up to 15%, shift horizontally and vertically up to 20% of the total width and height, shear transform randomly up to 15 degrees, and use the "nearest" strategy to fill in missing pixels. In addition, rescaling is used to normalize the image pixel values to a range of 0 to 1. For the validation data, only normalization has been applied [5].

```
# Initialize an empty list to store augmented sequences
X_train_augmented_sequences = []
y_train_augmented = []

# Assuming trainAug is your data augmenter
for i in range(len(X_train)):
    augmented_sequence = []
    for j in range(SEQUENCE_LENGTH):
        # Apply data augmentation to each frame
        augmented_frame = trainAug.random_transform(X_train[i][j])
        augmented_sequence.append(augmented_frame)
    X_train_augmented_sequences.append(augmented_sequence)
    y_train_augmented.append(y_train[i])

# Convert the list to a numpy array
X_train_augmented_sequences = np.array(X_train_augmented_sequences)
y_train_augmented = np.array(y_train_augmented)
```

```
X_test_augmented_sequences = []
y_test_augmented = []

# Assuming trainAug is your data augmenter (make sure the variable name is consistent)
for i in range(len(X_test)):
    augmented_sequence = []
    for j in range(SEQUENCE_LENGTH):
        # Apply data augmentation to each frame
        augmented_frame = valAug.random_transform(X_test[i][j])
        augmented_sequence.append(augmented_frame)
          # Append the label for each frame in the sequence
    X_test_augmented_sequences.append(augmented_sequence)
    y_test_augmented.append(y_test[i])

# Convert the list to a numpy array
X_test_augmented_sequences = np.array(X_test_augmented_sequences)
y_test_augmented = np.array(y_test_augmented)
```

The above code is used to augment the pre-existing data. Two lists, named **X_train_augmented_sequences** and **y_train_augmented**, are set up to store the augmented sequences of frames and their corresponding labels. The code goes through each sample in the original training dataset (**X_train**) using nested loops, addressing individual frames within the sequence (controlled by the constant SEQUENCE_LENGTH). Data augmentation is applied to each frame independently using the **trainAug** data augmenter, which involves random changes like flips, rotations, shifts, and zooms. These augmented frames are then combined into a sequence (referred to as augmented_sequence) for each sample. The resulting sequences and labels are added to the respective lists. Lastly, the lists are transformed into NumPy arrays for compatibility with machine learning frameworks such as TensorFlow or Keras. Similarly, the same process has been applied to test data.

```
# Concatenate the original and augmented data
X_train_combined = np.concatenate((X_train, X_train_augmented_sequences), axis=0)
y_train_combined = np.concatenate((y_train, y_train_augmented), axis=0)
```

```
print("Original X_train shape:", X_train.shape)
print("Augmented X_train_augmented shape:", X_train_augmented_sequences.shape)
```

```
Original X_train shape: (1500, 16, 64, 64, 3)
Augmented X_train_augmented shape: (1500, 16, 64, 64, 3)
```

```
# Concatenate the original and augmented data
X_test_combined = np.concatenate((X_test, X_test_augmented_sequences), axis=0)
y_test_combined = np.concatenate((y_test, y_test_augmented), axis=0)
```

```
print("Original X_train shape:", X_test.shape)
print("Augmented X_train_augmented shape:", X_test_augmented_sequences.shape)
```

```
Original X_train shape: (500, 16, 64, 64, 3)
Augmented X_train_augmented shape: (500, 16, 64, 64, 3)
```

```
print("Combined X_train_combined shape:", X_train_combined.shape)
print("Combined X_test_combined shape:", X_test_combined.shape)

print("Combined y_train_combined shape:", y_train_combined.shape)
print("Combined y_test_combined shape:", y_test_combined.shape)
```

```
Combined X_train_combined shape: (3000, 16, 64, 64, 3)
Combined X_test_combined shape: (1000, 16, 64, 64, 3)
Combined y_train_combined shape: (3000, 2)
Combined y_test_combined shape: (1000, 2)
```

After augmentation, the amount of data in the test and train data sets have been doubled. The process of data augmentation will help the model to adapt to various conditions while training.

Model Architecture

The model for this particular project has two components i.e. CNN (convolutional neural networks) and RNN (recurrent neural networks). CNN is used to extract spatial information and after the extraction of these particular features the data is sent to the RNN for temporal information extraction. MobileNetV2 architecture has been used as the CNN. The concept of **transfer learni**ng has been applied in this case, since the CNN architecture has used the pretrained weights of **imagenet** dataset (comprising of 100 million images).

```
from keras.applications.mobilenet_v2 import MobileNetV2

mobilenet = MobileNetV2(include_top=False , weights="imagenet")

#Fine-Tuning to make the last 40 layer trainable
mobilenet.trainable=True

for layer in mobilenet.layers[:-40]:
    layer.trainable=False
```

The top layer of the architecture has been excluded as the purpose of this project is different as compared to the **imagenet** dataset. Afterwards, a fine-tuning strategy is used to train the final 40 layers of the MobileNetV2 model. This entails freezing the layers that come before it and setting the 'trainable' attribute to 'True' for these particular layers. The MobileNetV2 model's layers are iterated through in the for loop, with the exception of the final 40 layers, which are marked as non-trainable. By strategically fine-tuning, the model can take advantage of the knowledge it acquired from pre-training on the larger ImageNet dataset to adapt and specialize to the peculiarities of a particular task.

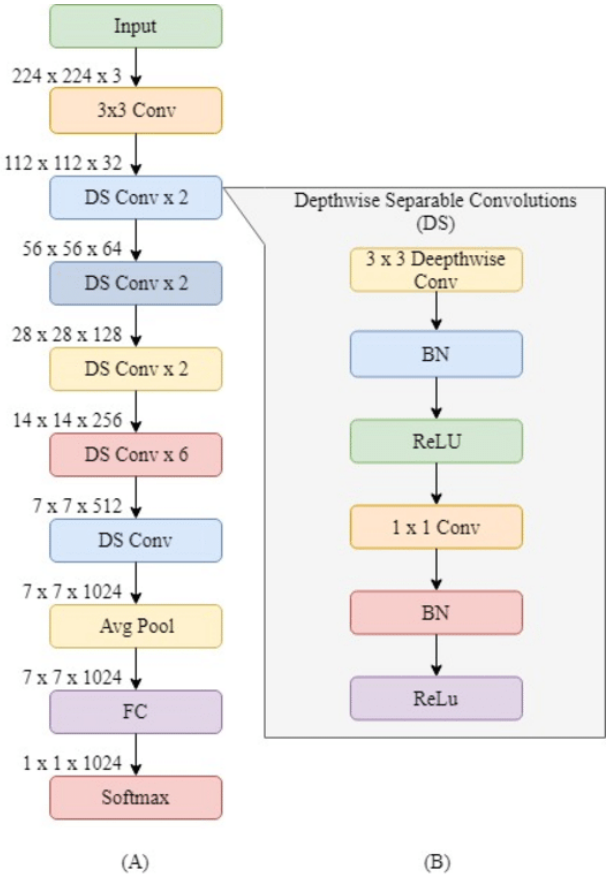


Fig. 3: MobileNetV2 Architecture

```
Total params: 2257984 (8.61 MB)
Trainable params: 1681536 (6.41 MB)
Non-trainable params: 576448 (2.20 MB)
```

These are the total number of parameters. As expected, the number of trainable parameters is less as compared to the non-trainable parameters due to the usage of pre-trained weights.

Quantization of MobileNet Architecture

```
import tensorflow_model_optimization as tfmot
```

```
# Helper function uses `quantize_annotate_layer` to annotate that only the
# Dense layers should be quantized.
# https://github.com/tensorflow/model-optimization/blob/0f6dd5aeb818c5f61123fc1d5642435ea0f5cd70/tensorflow_model_op

quantizable_layers = [
        tf.keras.layers.Dense,
        tf.keras.layers.DepthwiseConv2D,
        tf.keras.layers.AveragePooling2D,
        tf.keras.layers.GlobalAveragePooling2D,
        tf.keras.layers.Flatten,
        tf.keras.layers.ReLU,
        tf.keras.layers.Softmax,
        tf.keras.layers.Dropout
    ]

def apply_quantization_to_layers(layer):
    if any(isinstance(layer, layer_type) for layer_type in quantizable_layers):
        return tfmot.quantization.keras.quantize_annotate_layer(layer)
    return layer
```

```
# Use `tf.keras.models.clone_model` to apply `apply_quantization_to_dense`
# to the layers of the model.
annotated_mobilenet = tf.keras.models.clone_model(
    mobilenet,
    clone_function=apply_quantization_to_layers,
)

quant_aware_mobilenet = tfmot.quantization.keras.quantize_apply(annotated_mobilenet)
quant_aware_mobilenet.summary()
```

In the code section above, quantization is applied to particular layers of a MobileNetV2 model using the **TensorFlow Model Optimization (tfmot)** module. **Dense, DepthwiseConv2D, AveragePooling2D, GlobalAveragePooling2D, Flatten, ReLU, Softmax,** and **Dropout layers** are among the layers that can be quantized according to a list called '**quantizable_layers**'. Next, to annotate quantization to layers according to their kind, a function called '**apply_quantization_to_layers**' is made. The specified function applies the quantization annotations, and the code then uses TensorFlow's '**clone_model**' function to create an annotated version of the original MobileNetV2 model. Lastly, by calling '**quantize_apply**' from **tfmot**, the quantization-aware MobileNetV2 model ('**quant_aware_mobilenet**') is obtained. An overview of the model architecture following quantization is given by the summary of the quantization-aware model that is shown. In some cases, quantization does not reduce the memory or the number of parameters drastically, but it does help in decreasing the training time.

Building the MoBiLSTM

```
def create_model():

    model = Sequential()


    model.add(Input(shape = (SEQUENCE_LENGTH, IMAGE_HEIGHT, IMAGE_WIDTH, 3)))

    #passing mobilenet in TimeDistributed Layer to handle the sequence

    model.add(TimeDistributed(quant_aware_mobilenet))

    model.add(Dropout(0.1))

    model.add(TimeDistributed(Flatten()))

    lstm_fw = LSTM(units=32)
    lstm_bw = LSTM(units=32, go_backwards = True)

    model.add(Bidirectional(lstm_fw, backward_layer = lstm_bw))

    model.add(LayerNormalization())

    model.add(Dense(256,activation='relu'))
    model.add(LayerNormalization())

    model.add(Dense(128,activation='relu'))
    model.add(LayerNormalization())

    model.add(Dense(64,activation='relu'))
    model.add(LayerNormalization())

    model.add(Dense(32,activation='relu'))
    model.add(LayerNormalization())


    model.add(Dense(len(label_list), activation = 'softmax'))

    model.summary()

    return model
```

The code shown above describes a sequential neural network model for video analysis using the Keras API of TensorFlow. A Sequential model is initialized and its architecture is configured via the '**create_model**' function. It uses an input form in five dimensions that represents video sequences with predefined color, width, and height channels. Within a **TimeDistributed** layer, a MobileNetV2 model is introduced, enhanced with quantization annotations, to manage the sequential nature of video frames. To reduce overfitting, a **dropout layer** is added. Next, a

second **TimeDistributed** layer **flattens** the result. Sequence temporal dependencies are captured by bidirectional **Long Short-Term Memory (LSTM)** layers that have forward and backward components. Non-linearity and hierarchical features are added via additional layers, such as **LayerNormalization** and tightly coupled layers with different units. The final layer employs the **softmax** activation function to produce predictions across the defined label classes.

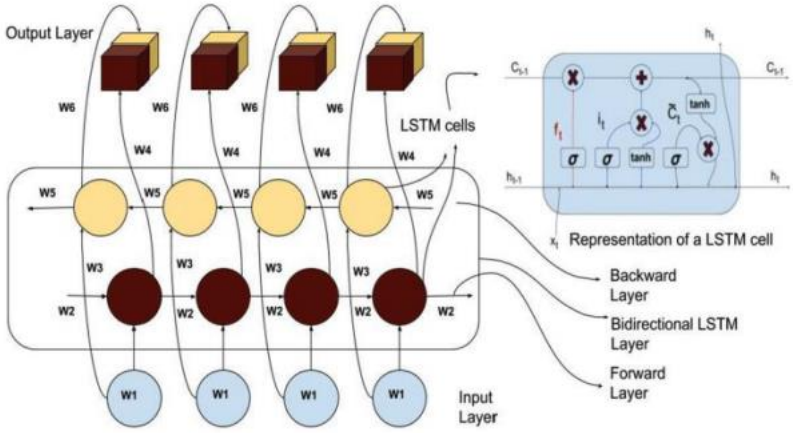


Fig. 4: Bi-Directional LSTM

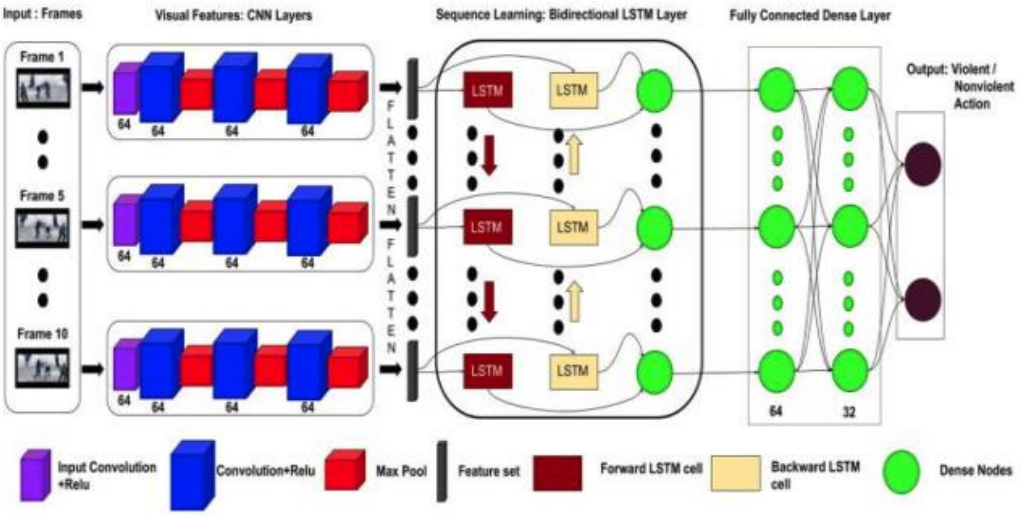


Fig. 5: Complete Architecture Map

Callbacks

```
callbacks = [
        tf.keras.callbacks.EarlyStopping(monitor = 'val_accuracy', patience = 5, restore_best_weights = True),
        tf.keras.callbacks.TensorBoard(log_dir='logs'),
        tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', # Create ReduceLROnPlateau Callback to reduce overf
                                             factor=0.6,
                                             patience=5,
                                             min_lr=0.00005,
                                             verbose=1)]
```

Among the callbacks are **ReduceLROnPlateau**, **TensorBoard**, and **EarlyStopping**. In an effort to avoid overfitting, the **EarlyStopping** callback tracks the validation accuracy and terminates the training procedure if no improvement is shown in a predetermined amount of time. Additionally, it puts the model's weights back to optimal. Training metrics are logged by the **TensorBoard** callback for analysis and display. Finally, in response to a plateau in validation loss, the **ReduceLROnPlateau** callback dynamically modifies the learning rate during training. Through the adjustment of the learning rate, this mechanism guides the model towards a more ideal parameter configuration, hence optimizing performance. All things considered, these callbacks support the stability of the model, guard against overfitting, and enable perceptive examination of the training procedure.

Training without Quantization

```
MoBiLSTM_model.compile(loss = 'categorical_crossentropy',
                       optimizer = tf.keras.optimizers.legacy.Adam(), metrics = ["accuracy"])

MobBiLSTM_model_history = MoBiLSTM_model.fit(x = X_train, y = y_train, epochs = 20, batch_size = 8 ,
                                             shuffle = True, validation_split = 0.2, callbacks = callbacks)
```

```
Epoch 1/20
150/150 [==============================] - 157s 989ms/step - loss: 0.4598 - accuracy: 0.8092 - val_loss: 0.6730 - v
al_accuracy: 0.7467 - lr: 0.0010
Epoch 2/20
150/150 [==============================] - 156s 1s/step - loss: 0.4134 - accuracy: 0.8342 - val_loss: 1.1117 - val_
accuracy: 0.5517 - lr: 0.0010
Epoch 3/20
150/150 [==============================] - 153s 1s/step - loss: 0.3826 - accuracy: 0.8442 - val_loss: 1.0572 - val_
accuracy: 0.5267 - lr: 0.0010
Epoch 4/20
150/150 [==============================] - 151s 1s/step - loss: 0.3962 - accuracy: 0.8292 - val_loss: 0.7672 - val_
accuracy: 0.5367 - lr: 0.0010
Epoch 5/20
150/150 [==============================] - 143s 956ms/step - loss: 0.3716 - accuracy: 0.8392 - val_loss: 1.1987 - v
al_accuracy: 0.5667 - lr: 0.0010
Epoch 6/20
150/150 [==============================] - ETA: 0s - loss: 0.2740 - accuracy: 0.8958
Epoch 6: ReduceLROnPlateau reducing learning rate to 0.0006000000284984708.
150/150 [==============================] - 165s 1s/step - loss: 0.2740 - accuracy: 0.8958 - val_loss: 0.9172 - val_
accuracy: 0.6400 - lr: 0.0010
```

Without Quantization, we observe that the accuracy is 89.6% on the training set. Also note the training time of each epoch (on an average is around 140 seconds).

Pruning, Clustering and Quantized Aware Training

```
import tensorflow_model_optimization as tfmot

prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude

pruning_params = {
      'pruning_schedule': tfmot.sparsity.keras.ConstantSparsity(0.5, begin_step=0, frequency=100)
  }

callbacks = [
  tfmot.sparsity.keras.UpdatePruningStep()
]

pruned_model = prune_low_magnitude(model, **pruning_params)

# Use smaller learning rate for fine-tuning
opt = tf.keras.optimizers.Adam(learning_rate=1e-5)

pruned_model.compile(
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  optimizer=opt,
  metrics=['accuracy'])

# Fine-tune model
pruned_model.fit(
  train_images,
  train_labels,
  epochs=3,
  validation_split=0.1,
  callbacks=callbacks)

stripped_pruned_model = tfmot.sparsity.keras.strip_pruning(pruned_model)

print_model_weights_sparsity(stripped_pruned_model)
```

```
import tensorflow_model_optimization as tfmot
from tensorflow_model_optimization.python.core.clustering.keras.experimental import (
    cluster,
)

cluster_weights = tfmot.clustering.keras.cluster_weights
CentroidInitialization = tfmot.clustering.keras.CentroidInitialization

cluster_weights = cluster.cluster_weights

clustering_params = {
  'number_of_clusters': 8,
  'cluster_centroids_init': CentroidInitialization.KMEANS_PLUS_PLUS,
  'preserve_sparsity': True
}

sparsity_clustered_model = cluster_weights(stripped_pruned_model, **clustering_params)

sparsity_clustered_model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

print('Train sparsity preserving clustering model:')
sparsity_clustered_model.fit(x = X_train, y = y_train, epochs = 20, batch_size = 8 ,
                                          shuffle = True, validation_split = 0.2, callbacks = callbacks)

stripped_clustered_model = tfmot.clustering.keras.strip_clustering(sparsity_clustered_model)
```

```
# QAT
qat_model = tfmot.quantization.keras.quantize_model(stripped_clustered_model)

qat_model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
qat_model.fit(x = X_train, y = y_train, epochs = 20, batch_size = 8 ,
                                        shuffle = True, validation_split = 0.2, callbacks = callbacks)

# PCQAT
quant_aware_annotate_model = tfmot.quantization.keras.quantize_annotate_model(
              stripped_clustered_model)
pcqat_model = tfmot.quantization.keras.quantize_apply(
              quant_aware_annotate_model,
              tfmot.experimental.combine.Default8BitClusterPreserveQuantizeScheme(preserve_sparsity=True))

pcqat_model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
pcqat_model.fit(x = X_train_combined, y = y_train_combined, epochs = 20, batch_size = 8 ,
                                        shuffle = True, validation_split = 0.2, callbacks = callbacks)
```

The above code snippet showcases a comprehensive optimization pipeline using TensorFlow Model Optimization (tfmot) for a neural network model.

1. **Pruning**:
    - The code initiates with magnitude-based weight pruning, where less important weights are pruned to achieve a sparsity level of 50%. This is achieved through the **prune_low_magnitude** function, with pruning parameters specifying a constant sparsity schedule.
2. **Fine-tuning**:
    - Following pruning, the pruned model undergoes a fine-tuning process. A smaller learning rate (1e-5) is employed during fine-tuning to counteract potential accuracy loss that might occur due to pruning.
3. **Clustering**:
    - Weight clustering is introduced, a technique where similar weights are grouped together. In this case, the model's weights are clustered into eight groups using the K-means++ initialization method. Importantly, sparsity is preserved during clustering to maintain efficiency.
4. **Training of Clustered Model**:
    - The model with clustered weights is trained to adapt to the clustered configuration. The training process is facilitated using the cluster_weights function.
5. **Stripping Clustering**:
    - After training with clustered weights, the clustering is stripped from the model to obtain a more compact and efficient representation of the neural network.
6. **Quantization-Aware Training (QAT)**:
    - Quantization-aware training (QAT) is applied to the model, introducing quantization techniques that reduce the precision of weights and activations. This process aims to decrease the model size without compromising predictive accuracy.

7. **Post-Training Quantization with Clustering (PCQAT):**
    - Post-training quantization with clustering and quantization-aware training (PCQAT) is performed. This involves combining clustering and quantization strategies to further reduce the model size while maintaining sparsity characteristics.
8. **Training of PCQAT Model:**
    - The model resulting from PCQAT is trained using the specified data (`X_train` and `y_train`) for 20 epochs with a batch size of 8. This step ensures that the model adapts to the combined effects of clustering and quantization.

```
l_accuracy: 0.8867 - lr: 6.0000e-04
Epoch 15/20
150/150 [==============================] - 63s 419ms/step - loss: 0.1178 - accuracy: 0.9625 - val_loss: 0.2741 - va
l_accuracy: 0.9033 - lr: 6.0000e-04
Epoch 16/20
150/150 [==============================] - 63s 419ms/step - loss: 0.0781 - accuracy: 0.9742 - val_loss: 0.2909 - va
l_accuracy: 0.9067 - lr: 6.0000e-04
Epoch 17/20
150/150 [==============================] - 63s 422ms/step - loss: 0.0843 - accuracy: 0.9733 - val_loss: 0.2116 - va
l_accuracy: 0.9200 - lr: 6.0000e-04
Epoch 18/20
150/150 [==============================] - 64s 424ms/step - loss: 0.0593 - accuracy: 0.9825 - val_loss: 0.2149 - va
l_accuracy: 0.9167 - lr: 6.0000e-04
Epoch 19/20
150/150 [==============================] - 63s 418ms/step - loss: 0.0409 - accuracy: 0.9883 - val_loss: 0.2669 - va
l_accuracy: 0.9100 - lr: 6.0000e-04
Epoch 20/20
150/150 [==============================] - 63s 418ms/step - loss: 0.0517 - accuracy: 0.9858 - val_loss: 0.2919 - va
l_accuracy: 0.9067 - lr: 6.0000e-04
```

After applying the PCQAT **Collaborative Optimization,** we notice a huge jump in accuracy (also attributed to the usage of Augmented Data) along with the time required to train per epoch. In this case, there is nearly a 50% reduction as compared to the previous training history.

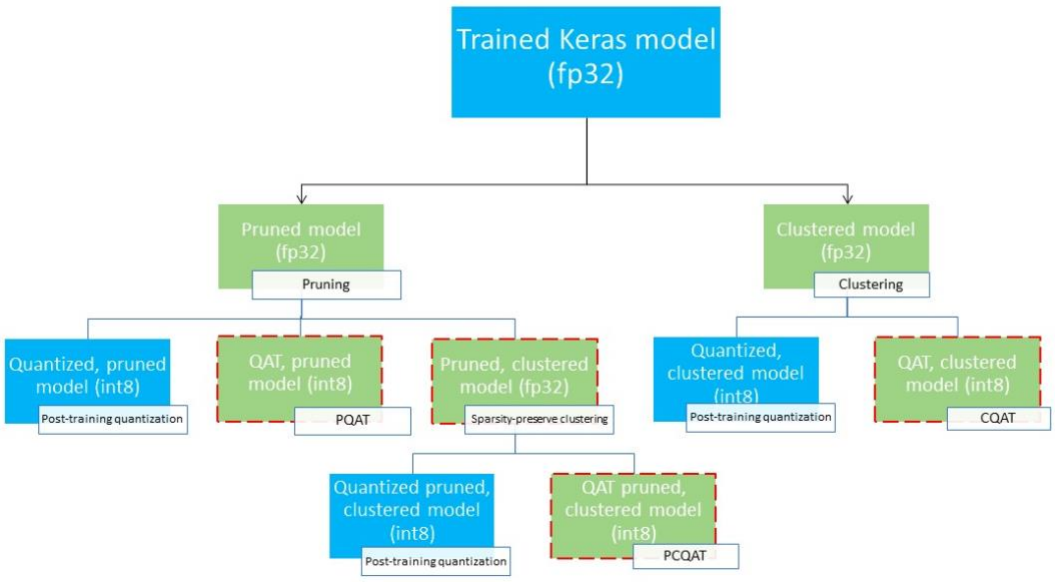


Fig. 6: Collaborative Optimization Framework

The third level of the deployment tree above is where the fully optimized model should be reached; however, any of the other optimization levels may be sufficient to meet the necessary inference latency, compression, and accuracy targets, in which case no additional optimization would be required. It is advised that the model be trained iteratively through the deployment tree levels that apply to the target deployment scenario. If the model does not meet the optimization requirements, it should be further compressed using the corresponding collaborative optimization technique. This process should be repeated until the model is fully optimized (pruned, clustered, and quantized), if necessary.

Model Evaluation

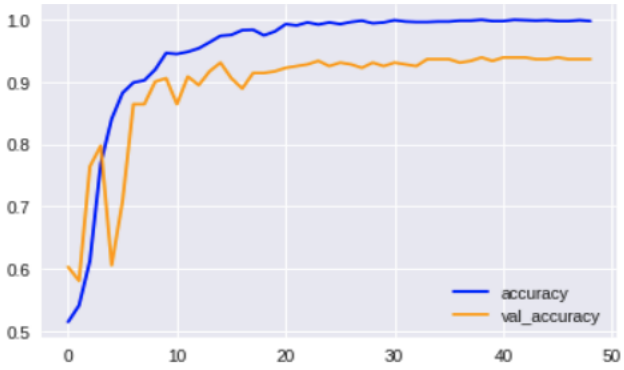


Fig.7: Training accuracy vs Validation Accuracy

Fig.8: Training Loss vs Validation Loss

Predicting the Test Set

```
labels_predict = loaded_model.predict(X_test)
```

```
16/16 [==============================] - 9s 549ms/step
```

```
# Decoding the data to use in Metrics
labels_predict = np.argmax(labels_predict , axis=1)
labels_test_normal = np.argmax(y_test , axis=1)
```

```
labels_test_normal.shape , labels_predict.shape
```

```
((500,), (500,))
```

```
from sklearn.metrics import accuracy_score
AccScore = accuracy_score(labels_predict, labels_test_normal)
print('Accuracy Score is : ', AccScore*100)
```

```
Accuracy Score is :  90.8
```

The accuracy of the model on the test set is around 91%.

Saving the model

The above-mentioned model architecture and weights can be stored in a **.h5 file** as the training process has completed. By saving in an .h5 file, the model's weights can be used in the future for prediction purposes.

Predicting Violence Frame by Frame

```
def predict_frames(video_file_path, output_file_path, SEQUENCE_LENGTH):

    # Read from the video file.
    video_reader = cv2.VideoCapture(video_file_path)

    # Get the width and height of the video.
    original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))
    original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))

    # VideoWriter to store the output video in the disk.
    video_writer = cv2.VideoWriter(output_file_path, cv2.VideoWriter_fourcc('m', 'p', '4', 'v'),
                                   video_reader.get(cv2.CAP_PROP_FPS), (original_video_width, original_video_height

    # Declare a queue to store video frames.
    frames_queue = deque(maxlen = SEQUENCE_LENGTH)

    # Store the predicted class in the video.
    predicted_class_name = ''

    # Iterate until the video is accessed successfully.
    while video_reader.isOpened():

        ok, frame = video_reader.read()

        if not ok:
            break

        # Resize the Frame to fixed Dimensions.
        resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))

        # Normalize the resized frame
        normalized_frame = resized_frame / 255

        # Appending the pre-processed frame into the frames list.
        frames_queue.append(normalized_frame)

        # We Need at Least number of SEQUENCE_LENGTH Frames to perform a prediction.
        # Check if the number of frames in the queue are equal to the fixed sequence length.
        if len(frames_queue) == SEQUENCE_LENGTH:

            # Pass the normalized frames to the model and get the predicted probabilities.
            predicted_labels_probabilities = loaded_model.predict(np.expand_dims(frames_queue, axis = 0))[0]

            # Get the index of class with highest probability.
            predicted_label = np.argmax(predicted_labels_probabilities)

            # Get the class name using the retrieved index.
            predicted_class_name = label_list[predicted_label]

        # Write predicted class name on top of the frame.
        if predicted_class_name == "Violence":
            cv2.putText(frame, predicted_class_name, (5, 100), cv2.FONT_HERSHEY_SIMPLEX, 3, (0, 0, 255), 12)
        else:
            cv2.putText(frame, predicted_class_name, (5, 100), cv2.FONT_HERSHEY_SIMPLEX, 3, (0, 255, 0), 12)

        # Write The frame into the disk using the VideoWriter
        video_writer.write(frame)

    video_reader.release()
    video_writer.release()
```

```
plt.style.use("default")

ax = plt.subplot()

# To show Random Frames from the saved output predicted video (output predicted video doesn't show on the notebook b
def show_pred_frames(pred_video_path):

    plt.figure(figsize=(20,15))

    video_reader = cv2.VideoCapture(pred_video_path)

    # Get the number of frames in the video.
    frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

    # Get Random Frames from the video then Sort it
    random_range = sorted(random.sample(range (SEQUENCE_LENGTH , frames_count ), 12))

    for counter, random_index in enumerate(random_range, 1):

        plt.subplot(5, 4, counter)

        # Set the current frame position of the video.
        video_reader.set(cv2.CAP_PROP_POS_FRAMES, random_index)

        ok, frame = video_reader.read()

        if not ok:
            break

        frame = cv2.cvtColor(frame , cv2.COLOR_BGR2RGB)

        plt.imshow(frame);ax.figure.set_size_inches(20,20);plt.tight_layout()

    video_reader.release()
```

```
# Construct the output video path.
test_videos_directory = 'test_videos'
os.makedirs(test_videos_directory, exist_ok = True)

output_video_file_path = f'{test_videos_directory}/Output-Test-Video.mp4'
```

```
input_video_file_path = "/Users/anasshaik/Documents/Design Project/Violence Detection/data/rlvs/Violence/V_256.mp4"

# Perform Prediction on the Test Video.
predict_frames(input_video_file_path, output_video_file_path, SEQUENCE_LENGTH)

# Show random frames from the output video
show_pred_frames(output_video_file_path)
```

Using a neural network that has already been trained, the function **predict_frames** in the given code has been created to evaluate videos for potential violence. Every frame of the movie is examined, its size is changed, and predictions are made depending on the order in which the frames appear. After that, the results, such "Violence" or "NonViolence," are incorporated into the frames to create a fresh video featuring these annotations. OpenCV is used for both the annotation creation and video processing.

Furthermore, a utility method called **show_pred_frames** has been added, which allows for the random selection of frames from the output video with annotations. This feature makes it easier to visually examine how the model performs in various scenes in the video.

Towards the end of the algorithm, a particular test video is chosen for violence predictions, and a few randomly selected frames from the annotated output video are used to illustrate the

outcomes. This practical example shows how a trained model may be used to detect violence in videos and provides a visual depiction of the model's performance.

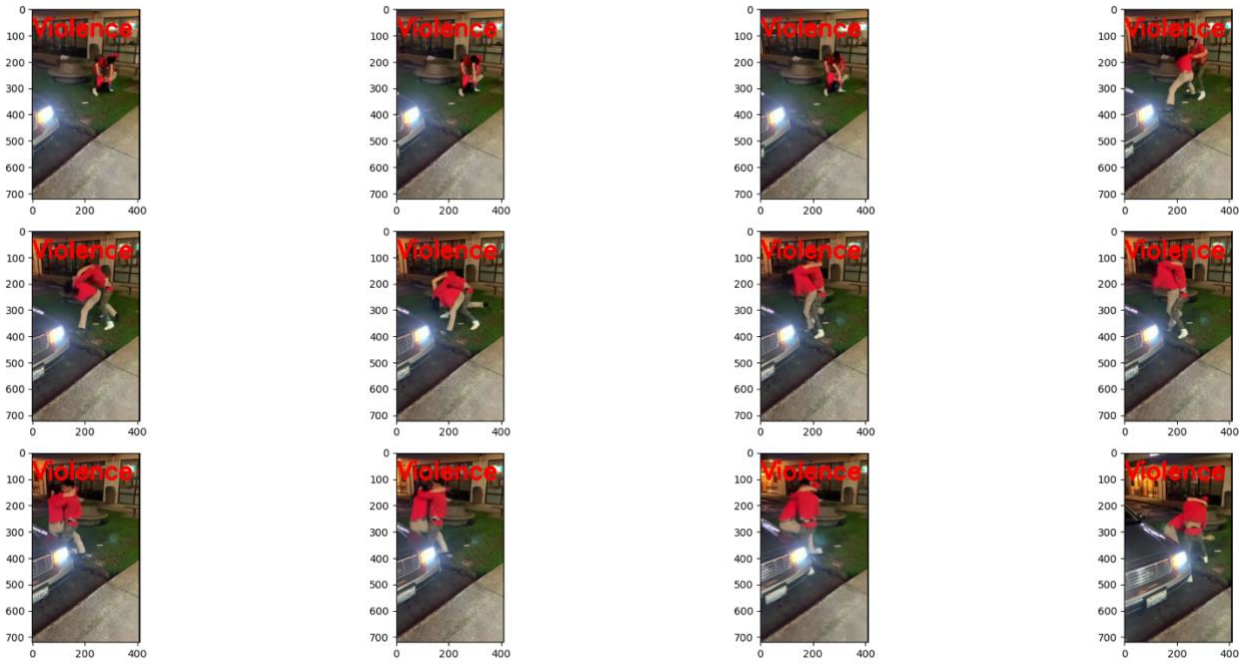


Fig. 9: Violence Frame by Frame Prediction

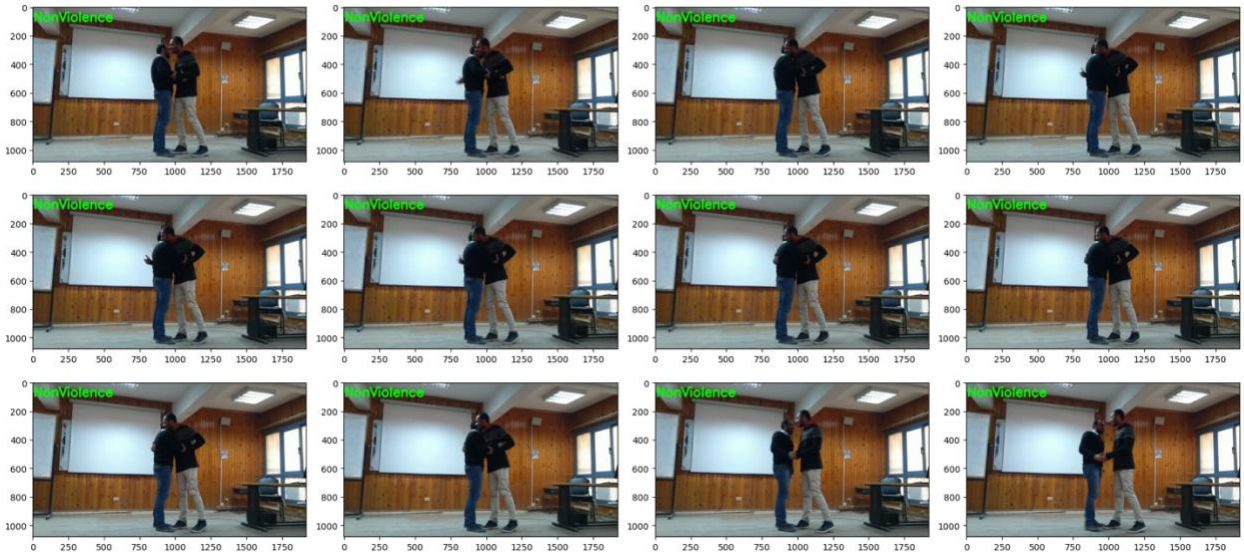


Fig. 10: Non-Violence Frame by Frame Prediction

Prediction for the Video

```
def predict_video(video_file_path, SEQUENCE_LENGTH):

    video_reader = cv2.VideoCapture(video_file_path)

    # Get the width and height of the video.
    original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))
    original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))

    # Declare a list to store video frames we will extract.
    frames_list = []

    # Store the predicted class in the video.
    predicted_class_name = ''

    # Get the number of frames in the video.
    video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate the interval after which frames will be added to the list.
    skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH),1)

    # Iterating the number of times equal to the fixed length of sequence.
    for frame_counter in range(SEQUENCE_LENGTH):

        # Set the current frame position of the video.
        video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window)

        success, frame = video_reader.read()

        if not success:
            break

        # Resize the Frame to fixed Dimensions.
        resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))

        # Normalize the resized frame.
        normalized_frame = resized_frame / 255

        # Appending the pre-processed frame into the frames list
        frames_list.append(normalized_frame)

    # Passing the  pre-processed frames to the model and get the predicted probabilities.
    predicted_labels_probabilities = loaded_model.predict(np.expand_dims(frames_list, axis = 0))[0]

    # Get the index of class with highest probability.
    predicted_label = np.argmax(predicted_labels_probabilities)

    # Get the class name using the retrieved index.
    predicted_class_name = label_list[predicted_label]

    percentage = predicted_labels_probabilities[predicted_label] * 100

    # Display the predicted class along with the prediction confidence.
    print(f'Predicted: {predicted_class_name}\nConfidence: {percentage:.3f} %')

    video_reader.release()
```

```
# Specifying video to be predicted
input_video_file_path = "/Users/anasshaik/Documents/Design Project/Violence Detection/data/rlvs/Violence/V_276.mp4"

# Perform Single Prediction on the Test Video.
predict_video(input_video_file_path, SEQUENCE_LENGTH)

# Play the actual video
Play_Video(input_video_file_path)
```

```
1/1 [==============================] - 0s 52ms/step
Predicted: Violence
Confidence: 99.159 %
```

In the provided code snippet, a function named **predict_video** has been created to facilitate the prediction of violence in a given video file. The function employs OpenCV to read frames from the input video and extracts a sequence of frames based on a specified length **(SEQUENCE_LENGTH).** The frames are processed by resizing them to fixed dimensions **(IMAGE_HEIGHT and IMAGE_WIDTH)** and normalizing the pixel values. These preprocessed frames are then appended to a list named **frames_list** for further analysis.

Subsequently, the function utilizes a pre-trained model (**loaded_model**) to predict the likelihood of violence in the video. The frames from the **frames_list** is passed through the model, and the resulting predicted probabilities are obtained. The class label with the highest probability is determined, and its corresponding class name is retrieved from the predefined **label_list**. Additionally, the confidence level of the prediction is calculated as a percentage.

To provide insights into the prediction outcomes, the function prints the predicted class and its associated confidence level. It is important to note that this code snippet operates within the context of a larger application or script designed for violence detection in videos.

Upon completing the prediction process, the function releases the video reader resources, ensuring proper cleanup.

Streamlit Application

The "**PeaceGuard: Violence Detection Platform**" Streamlit application has features for detecting violence in videos. Using the **load_model** function, the pre-trained violence detection model is loaded. **TensorFlow Model Optimization** quantizes the loaded model to increase deployment efficiency.

Uploading video files that are limited to **mp4, avi, or mkv** using the Streamlit interface is how users interact with the service. The uploaded video must be locally saved via the **save_video_locally** method in order to enable further processing. After that, viewers can see a visual representation of the uploaded video by using Streamlit's **st.video** function to see the saved video.

The **predict_video** function implements the fundamental features for violence detection. The pre-trained model receives the video frames that have been saved, read, and preprocessed by this function. To get predictions, frames are chosen periodically. Next, the Streamlit interface shows the projected class and the confidence %. The program displays the confidence level that corresponds with the anticipated class, the input video, and a video player.

The **main** function serves as the entry point for the Streamlit application, establishing the application title, providing user guidance on acceptable file formats, and creating an interface for video file uploads. Upon video upload, the application saves the video locally and proceeds to display the predicted results using the **predict_video** function.

In summary, the application enables users to upload videos, process them through the model, and visualize real-time predictions.

Code

```
import streamlit as st
import os
import pickle
import tensorflow as tf
import cv2
import numpy as np
import tensorflow_model_optimization as tfmot

label_list = ["NonViolence","Violence"]

# Function to load the pre-trained model
@st.cache_resource
def load_model():
    # model = tf.keras.models.load_model('MoBiLSTM.h5')
    # return model
    with tfmot.quantization.keras.quantize_scope():
        loaded_model = tf.keras.models.load_model('MoBiLSTM.h5')
        return loaded_model


def save_video_locally(uploaded_file, custom_file_name):
    # Specify the file name with the desired extension
    file_extension = uploaded_file.name.split(".")[-1]
    saved_file_name = f"{custom_file_name}.{file_extension}"
    # Save the video to a temporary file on disk
    with open(saved_file_name, "wb") as f:
        f.write(uploaded_file.getbuffer())
    return saved_file_name
```

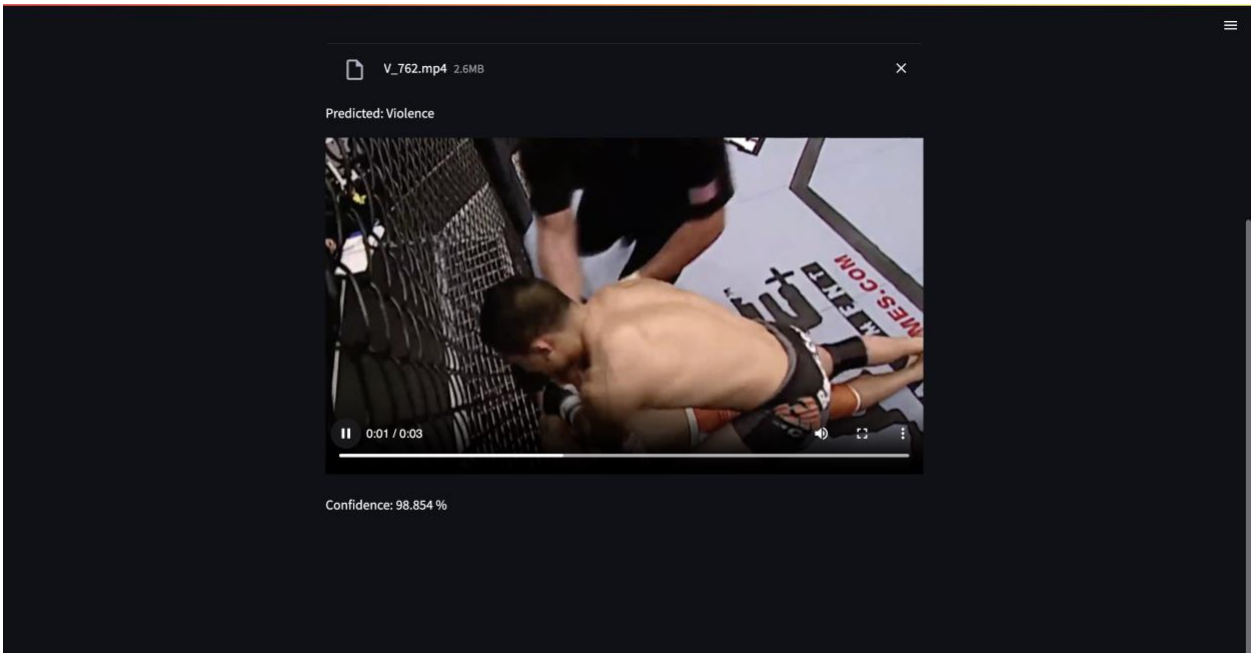
```
def main():
    st.title("PeaceGuard: Violence Detection Platform")
    model = load_model()
    st.warning("Please upload MP4 files only", icon="⚠")
    uploaded_file = st.file_uploader("Choose a video file", type=["mp4", "avi", "mkv"])

    if uploaded_file is not None:

        # Save the video file locally
        # save_button = st.button("Predict")

        saved_file_name = save_video_locally(uploaded_file, "input_video")
        # video_file = open('Violence Detection/input_video.mp4', 'rb')
        # video_bytes = video_file.read()
        # st.video(video_bytes)
        try:
            st.video(f"Violence Detection/{saved_file_name}")
        except Exception as e:
            print(f"Error: {e}")
            # Print the full traceback for more details
            import traceback
            traceback.print_exc()

        # After saving the video, display the results on a new page
        predict_video("/Users/anasshaik/Documents/Design Project/Violence Detection/input_video.mp4", model)

def save_video(uploaded_file):
    with open(uploaded_file.name, "wb") as f:
        f.write(uploaded_file.getbuffer())
    st.success(f"Video '{uploaded_file.name}' saved locally!")
```

```python
def save_video(uploaded_file):
    with open(uploaded_file.name, "wb") as f:
        f.write(uploaded_file.getbuffer())
    st.success(f"Video '{uploaded_file.name}' saved locally!")

def display_results_page():
    # Load the pre-trained model

    # Display the results on a new page
    st.subheader("Predicted Results")
    st.write("Replace this with your predicted result display logic")
    # st.write(f"Predicted Result: {predicted_result}")
```

```python
def predict_video(video_file_path, model, SEQUENCE_LENGTH = 16):

    video_reader = cv2.VideoCapture(video_file_path)

    # Get the width and height of the video.
    original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))
    original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))

    # Declare a list to store video frames we will extract.
    frames_list = []

    # Store the predicted class in the video.
    predicted_class_name = ''

    # Get the number of frames in the video.
    video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate the interval after which frames will be added to the list.
    skip_frames_window = max(int(video_frames_count/16),1)

    # Iterating the number of times equal to the fixed length of sequence.
    for frame_counter in range(16):

        # Set the current frame position of the video.
        video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window)

        success, frame = video_reader.read()

        if not success:
            break

        # Resize the Frame to fixed Dimensions.
        resized_frame = cv2.resize(frame, (64, 64))

        # Normalize the resized frame.
        normalized_frame = resized_frame / 255

        # Appending the pre-processed frame into the frames list
        frames_list.append(normalized_frame)
```

```
    # Passing the  pre-processed frames to the model and get the predicted probabilities.
    predicted_labels_probabilities = model.predict(np.expand_dims(frames_list, axis = 0))[0]

    # Get the index of class with highest probability.
    predicted_label = np.argmax(predicted_labels_probabilities)

    # Get the class name using the retrieved index.
    predicted_class_name = label_list[predicted_label]

    percentage = predicted_labels_probabilities[predicted_label] * 100

    # Display the predicted class along with the prediction confidence.
    print(f'Predicted: {predicted_class_name}\nConfidence: {percentage:.3f} %')

    st.write(f'Predicted: {predicted_class_name}')

    st.video("/Users/anasshaik/Documents/Design Project/Violence Detection/input_video.mp4")

    st.write(f'Confidence: {percentage:.3f} %')

    video_reader.release()

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

Application


V_762.mp4 2.6MB
Predicted: Violence
0:01 / 0:03
Confidence: 98.854 %

Conclusions and Future Scope

The imperative need for optimizing neural networks to suit resource-constrained devices, including the burgeoning demand for IoT devices and mobile phones, remains undeniable in our contemporary landscape. This project not only prompted contemplation on the software/application level optimization of neural networks but also provided valuable insights into the hardware considerations associated with implementing CNN models. The journey commenced with an extensive study of diverse optimization techniques, primarily focused on resource management.

Upon gaining a profound understanding of these optimization concepts, the project transitioned into a practical application within the realm of Violence Detection, a context where CCTV cameras represent resource-constrained devices. Leveraging collaborative optimization APIs, the model exhibited an impressive performance of 91%, closely aligning with the outcomes proposed in the research paper titled '**Robust Real-Time Violence Detection in Video Using CNN And LSTM**' [9].

The project's success underscores the potential for further refinement of this architecture. Future enhancements could involve the utilization of software-based hyperparameter tuning tools like **Keras Tuner** and exploration of alternative matrix multiplication algorithms, such as the **Strassen Algorithms**. These avenues for improvement contribute to the ongoing evolution of optimized neural networks, paving the way for more efficient and effective implementations in real-world scenarios.

References

[1]. Hussain, H., Tamizharasan, P. S., & Yadav, P. K. (2023). LCRM: Layer-Wise Complexity Reduction Method for CNN model optimization on end devices. *IEEE Access*, *11*, 66838–66857. https://doi.org/10.1109/access.2023.3290620

[2]. Soliman, M., Kamal, M. H., Nashed, M., Mostafa, Y. M., Chawky, B. S., & Khattab, D. (2019). Violence Recognition from Videos using Deep Learning Techniques. *2019 IEEE Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*. https://doi.org/10.1109/icicis46948.2019.9014714

[3]. Ronneberger, O., Fischer, P., & Brox, T. (2015). U-NET: Convolutional Networks for Biomedical Image Segmentation. In *Lecture Notes in Computer Science* (pp. 234–241). https://doi.org/10.1007/978-3-319-24574-4_28

[4] Jayasimhan, A., & Pabitha, P. (2022). A hybrid model using 2D and 3D Convolutional Neural Networks for violence detection in a video dataset. *2022 3rd International Conference on Communication, Computing and Industry 4.0 (C2I4)*. https://doi.org/10.1109/c2i456876.2022.10051324

[5]. Jahlan, H. M. B., & Elrefaei, L. A. (2022). Detecting violence in video based on deep features fusion technique. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2204.07443

[6]. Dhouibi, M., Salem, A. K. B., & Saoud, S. B. (2021). Optimization of CNN model for image classification. *2021 IEEE International Conference on Design & Test of Integrated Micro & Nano-Systems (DTS)*. https://doi.org/10.1109/dts52014.2021.9497988

[7]. Arish, S., Sinha, S., & Smitha, K. G. (2019). Optimization of Convolutional Neural Networks on Resource Constrained Devices. *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. https://doi.org/10.1109/isvlsi.2019.00013

[8]. Akash, S., Moorthy, R. S. S., Esha, K., & Nathiya, N. (2022). Human violence detection using deep learning techniques. *Journal of Physics*, *2318*(1), 012003. https://doi.org/10.1088/1742-6596/2318/1/012003

[9]. Abdali, A. R., & Ghani, R. F. (2019). Robust Real-Time Violence Detection in Video Using CNN And LSTM. *2019 2nd Scientific Conference of Computer Sciences (SCCS)*. https://doi.org/10.1109/sccs.2019.8852616

[10]. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. W., Adam, H., & Kalenichenko, D. (2017). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1712.05877

[11]. Khan, H. M., Ramzan, M., Khan, H. U., Iqbal, S., Khan, M. A., Choi, J., Nam, Y., & Kadry, S. (2021). Real-Time violent action recognition using key frames extraction and deep learning. *Computers, Materials & Continua*, *69*(2), 2217–2230. https://doi.org/10.32604/cmc.2021.018103

[12]. Ullah, I., Hussain, T., Iqbal, A., Yang, B., & Hussain, A. (2022). Real time violence detection in surveillance videos using Convolutional Neural Networks. *Multimedia Tools and Applications*, *81*(26), 38151–38173. https://doi.org/10.1007/s11042-022-13169-4

[13]. Iandola, F., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and. *arXiv (Cornell University)*. https://arxiv.org/pdf/1602.07360.pdf

[14]. Kiran, P. S., Reddy, B. T., Keerthi, G., Avanigadda, V. S., Vinnakota, C. P., & Sri, D. (2023). Violence Prediction System Using Bi-LSTM. *2023 3rd International Conference on Pervasive Computing and Social Networking (ICPCSN)*. https://doi.org/10.1109/icpcsn58827.2023.00010