Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Hawk-Eye: An AI-Powered Threat Detector for Intelligent Surveillance Cameras

AHMED ABDELMOAMEN AHMED[1] (Member, IEEE) and MATHIAS ECHI[1]
[1]Department of Computer Science, Prairie View A&M University, Prairie View, TX, USA

Corresponding author: Ahmed Abdelmoamen Ahmed (e-mail: amahmed@pvamu.edu).

ABSTRACT With recent advances in both AI and IoT capabilities, it is possible than ever to implement surveillance systems that can automatically identify people who might represent a potential security threat to the public in real-time. Imagine a surveillance camera system that can detect various on-body weapons, masked faces, suspicious objects and traffic. This system could transform surveillance cameras from passive sentries into active observers which would help in preventing a possible mass shooting in a school, stadium or mall. In this paper, we present a prototype implementation of such systems, Hawk-Eye, an AI-powered threat detector for smart surveillance cameras. Hawk-Eye can be deployed on centralized servers hosted in the cloud, as well as locally on the surveillance cameras at the network edge. Deploying AI-enabled surveillance applications at the edge enables the initial analysis of the captured images to take place on-site, which reduces the communication overheads and enables swift security actions. At the cloud side, we built a Mask R-CNN model that can detect suspicious objects in an image captured by a camera at the edge. The model can generate a high-quality segmentation mask for each object instance in the image, along with the confidence percentage and classification time. The camera side used a Raspberry Pi 3 device, Intel Neural Compute Stick 2 (NCS 2), and Logitech C920 webcam. At the camera side, we built a CNN model that can consume a stream of images directly from an on-site webcam, classify them, and displays the results to the user via a GUI-friendly interface. A motion detection module is developed to capture images automatically from the video when a new motion is detected. Finally, we evaluated our system using various performance metrics such as classification time and accuracy. Our experimental results showed an average overall prediction accuracy of 94% on our dataset.

INDEX TERMS AI, Threat Detector, Surveillance Cameras, Deep Learning, Mask R-CNN, CNN.

I. INTRODUCTION

According to the nonprofit Gun Violence Archive [1], around 100 people are killed using guns, and 200 more are shot and wounded every day in the United States. In 2019, there have been more than 350 mass shootings in the United States. The effects of such gun violence extend far beyond these casualties. It reshapes the lives of millions of people who are witnessing it and living in fear of the next shooting. So, there is a growing social demand for developing effective technological methods to compact gun violence.

A traditional way to mitigate potential security threats is by using surveillance cameras, which are ubiquitously installed in almost every public place such as schools, malls, shops, residential apartments and parking lots. However, the traditional methods of surveillance need a continual human observation though dozens of monitors in realtime. This may result in missing vulnerable situations due to fatigue, lack of concentration, or loss of key information in the surveillance videos [2]. Therefore, an intelligent surveillance system is critically needed to automatically detect security threats in a stream of videos without manual intervention.

The advancement of Artificial Intelligence (AI), Machine Learning (ML), and Internet of Things (IoT) opens up an opportunity to implement such intelligent surveillance systems. Imagine an AI-powered security camera system that can alert about various weapons and guns, masked faces, and suspicious objects in real-time. This can prevent a mass shooting or terrorist attack before it happens. To detect threatening objects in a live video, advanced deep learning models must be deployed on-site to improve object detection and semantic

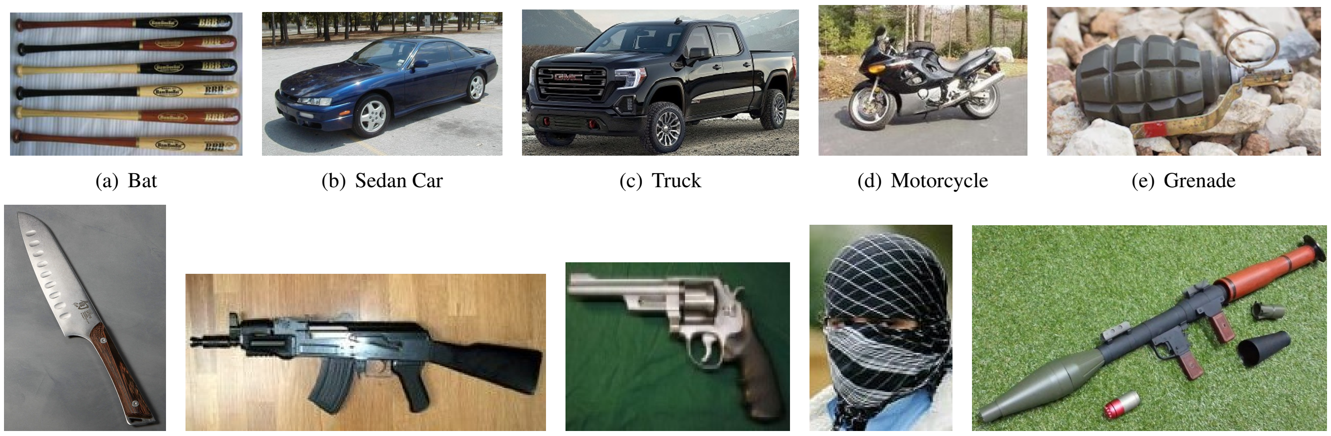

(a) Bat (b) Sedan Car (c) Truck (d) Motorcycle (e) Grenade
(f) Knife (g) Machine Guns (h) Pistol (i) Masked Face (j) RPG

FIGURE 1. Sample Examples from our Weapon Dataset Showing Various Types of Threatening Objects.

segmentation of the video content. However, it is challenging to deploy ML-enabled surveillance applications on cameras at the network edge because of its intensive computational requirements [3].

This paper presents Hawk-Eye, a lightweight AI-powered threat detector for intelligent surveillance cameras, which can be deployed on-site at the edge[1]. The goal of Hawk-Eye is to minimize communication delays, which is essential to perform sensitive and mission-critical tasks such as thread detection using surveillance cameras. Hawk-Eye is organized into two parts executing on centralized servers on the cloud, as well as locally on the surveillance cameras. We created a dataset that consists of more than 10k images for various types of threatening objects including masked faces, all kinds of guns including machine guns, grenade, Rocket-Propelled Grenade (RPG), pistol, motorcycle, cars, knife, and bats.

Figure 1 shows some examples of various types of dangerous objects from our weapon dataset. Most of our dataset images are in their natural environments because object detection is highly dependent on contextual information. Note that the car and truck images are included in the threatening objects because they may be included with other weapon objects in the same scene image. For instance, consider a masked-thief robbing a bank who escapes from the crime place using a car/truck; hence, our threat detector model could detect the whole scene as a security threat scenario. This feature is enabled because our threat detector can handle multi-class classification problems, where it can classify the input image as belonging to one or more of the nine weapon classes.

At the camera side, we created a Convolutional Neural Network (CNN) model that can feed images directly from on-site cameras, perform object detection and semantic segmentation, and display the classification results to the user via a web-based interface. We deployed the CNN model on an Intel Neural Compute Stick 2 (NCS2), which works with a Raspberry Pi 3 device, to enable real-time deep learning inferencing at the edge. At the cloud side, we built a Mask Region-based CNN (R-CNN) model [4], which is a deep learning model for object instance segmentation. Mask R-CNN can detect objects of interest in an image while generating a segmentation mask for each instance.

We also developed a motion detection module based on the frame difference method [5]. We calculate the absolute difference value between adjacent frames in the captured video. When a change in the current scene is detected, the motion detection module captures a fresh image with the highest possible quality in realtime. The captured images are then fed into the threat detector model automatically for classification.

We developed a web-based interface using Python Flask Framework [6], which enables users to upload an image or capture an image directly from the camera. The web application runs on top of both the Mask R-CNN model at the cloud side and the CNN model on the camera side. Also, the application displays the confidence percentage and classification time taken to process the image.

The contributions of this paper are fourfold. First, we propose a distributed AI-Powered system that can help security personnel detect various types of weapons in real-time, preventing a potential crime. The system implementation at the camera side uses the Intel NCS 2 device to boost the processing capabilities of limited-resource IoT devices (e.g., Raspberry Pi 3 device) for deploying powerful machine learning models locally on the surveillance camera without cloud compute dependence. Second, we developed a user-friendly interface on top of both CNN and Mask R-CNN to allow users to interact with the threat detector system conveniently at the camera and cloud sides. Third, a novel motion detection module is proposed for detecting moving objects in surveillance videos in real-time. The developed module is integrated seamlessly with both the camera and cloud sides. Fourth, the system is designed to be generic,

[1]Both the source code and dataset are available online: https://github.com/ahmed-pvamu/Hawk-Eye

making it applicable to different fields requiring real-time processing and using cameras and IoT sensors such as in the transportation field.

The rest of the paper is organized as follows: Section II presents related work. Sections IV and III present the design and prototype implementation of the system, respectively. Section V experimentally evaluates Hawk-Eye in terms of classification time and accuracy. Finally, Section VI summarizes the results of this work.

II. RELATED WORK

The existing video surveillance approaches from both academia and industry, to the best of our knowledge, offload the captured video to the cloud for further analysis. This precludes the initial analysis of videos to take place on-site, which leads to increased communication overheads and slows down security actions [3]. This section focuses on existing work on video surveillance from different perspectives such as human-weapon activity recognition [7, 8], crime detection [9], traffic monitoring [10], monitoring indoor surveillance video [11], moving object tracking [12] and face identification [13, 14].

Lim et al. [7] collected an annotated dataset consists of 5,500 gun images in various settings extracted from 250 recorded Closed Circuit Television Systems (CCTV) videos. The authors used the collected dataset to train a single-stage object detector using a multi-level feature pyramid network. Experimental results indicated that the collected dataset increases the precision accuracy by 18% when compared to two other existing datasets. Both training and validation phases are carried out offline on a centralized server equipped with two NVIDIA Quadro P5000 GPUs with a combined video memory of 32 GB hosted in the cloud.

Another crime detection system based on CCTV images was proposed in [9] using deep learning models. When the system detects a potential crime, it sends an SMS message to the human supervisor to take the necessary actions. The system can only detect two types of weapons, namely manual gun and knife, using the pre-trained VGGNET19 model. However, the model suffers from poor prediction accuracy due to the limited dataset used to train the model.

Grega et al. [8] proposed an algorithm for automatic firearms detection using recorded CCTV image analysis and situation recognition. The algorithm is designed to raise a flag when a firearm is detected. The authors used 1,000 positive examples and 3,500 negative examples to train a CNN model. The classification output of the CNN model is analyzed by a MPEG-7 classifier to determine whether the target object is a firearm. However, the collected imagery dataset suffered from poor quality and low resolution, which degraded the firearms detection accuracy.

In the field of traffic monitoring, Zhang et al. [10] proposed a vehicle detection and annotation algorithm based on a fine-tuned CNN model. The algorithm can identify vehicle positions and extract vehicle properties on highways from recorded traffic videos. The detection algorithm can predict the bounding-boxes for vehicles and infer their attributes such as pose, color and type.

Focusing on indoor surveillance, Liu et al. [11] proposed a video surveillance method for indoor environments based on a pre-trained model of the Mask R-CNN model using Microsoft COCO dataset [15]. The authors used a video recorded by a surveillance camera in a school lab to create the training dataset. Similar to the work done in [9], the indoor surveillance system was trained and validated using only 100 and 40 images, respectively. Given that small training dataset, the model was underfitting, and consequently, the classification accuracy was relatively low.

Object tracking is an essential step in object recognition in video surveillance. For instance, in [12], Cui et al. proposed a multiple granular cascaded model of object tracking in surveillance videos, where various frames from different granularity levels are analyzed. At the coarse level, the authors used a Gaussian mixture model to extract the foreground region of the frame. At a finer scale, they used a cascaded multi-source feature fusion method to classify the region proposal and obtain the location of the target. Experimental results showed that the system is able to obtain an accuracy of 61% on images that have multiple objects.

Interesting work was presented in [13, 14] for face recognition using surveillance videos. In [14], Ding et al. proposed a trunk-branch ensemble CNN platform for face recognition using recorded surveillance videos. The platform can extract complementary information from holistic face images. Another method for face recognition in real-world surveillance videos using deep learning was presented in [13], where a human face dataset is created automatically and incrementally during the face detection process. The authors fine-tuned the pre-trained VGG face model [16] with a dataset collected from web scraping consists of around 2,600 human faces. The fine-tuned VGG face model increased the recognition accuracy by 9% compared to the original VGG model.

In summary, most of the existing video surveillance approaches work on either recorded videos [7, 8, 11, 14] or offload the video data into a centralized server in the cloud [9, 10, 13], which is not appropriate for real-time threat detection. Furthermore, none of the current video surveillance approaches can be deployed on the security camera due to its limited computational capabilities, which precludes taking advantage of minimizing the communication delays in such mission-critical security tasks.

III. SYSTEM DESIGN

As illustrated in Figure 2, the distributed run-time system for Hawk-Eye is organized with parts executing on IoT devices at the camera side, as well as on centralized servers at the cloud side. The hardware components of the camera side are shown in layer 1, which contains a Raspberry Pi 3, NCS2 device and webcam. Layer 2 describes the deep learning models used in Hawk-Eye, which includes the CNN and Mask R-CNN models. It also shows the Intermediate Representation (IR) model, which runs on the NCS2 device at

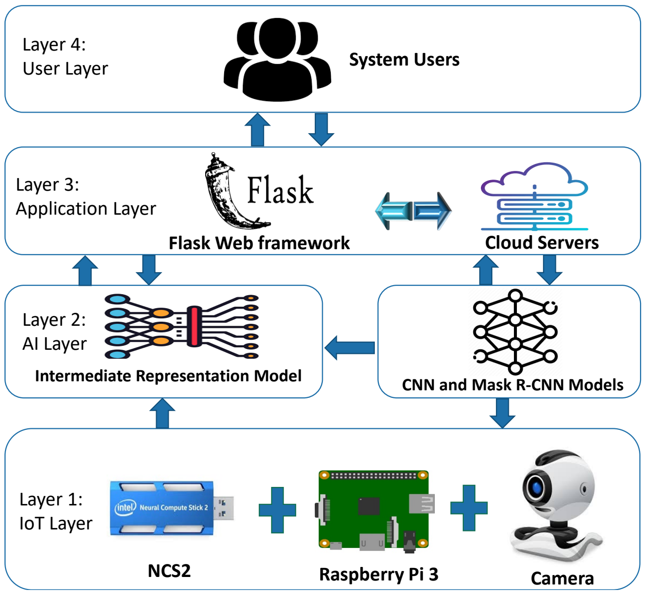


FIGURE 2. System Architecture

the camera side. Layer 3 illustrates the web-based interface of Hawk-Eye running on the cloud server. We used Python Flask Framework to develop a user-friendly web application that enables users (shown in layer 4) to interact with the system. In the rest of this section, we discuss these two parts separately.

A. CAMERA SIDE

At the camera side, we trained a CNN model with 4 convolutional layers, one input layer and one output layer. $I = [i_1, i_2, .., i_r]$ and $O = [o_1, o_2, .., o_h]$ represent the input and output vectors, respectively, where r represents the number of elements in the input feature set and h is the number of classes. The main objective of the network is to learn a compressed representation of the dataset. In other words, it tries to approximately learns the identity function F, which is defined as:

$$F_{W,B}(I) \simeq I \tag{1}$$

where W and B are the whole network weights and biases vectors.

A log sigmoid function is selected as the activation function f in the hidden and output neurons. The log sigmoid function s is a special case of the logistic function in the t space, which is defined by the formula:

$$s(t) = \frac{1}{1 + e^-t} \tag{2}$$

The weights of the network create the decision boundaries in the feature space, and the resulting discriminating surfaces can classify complex boundaries. During the training process, these weights are adapted for each new training image. In general, feeding the CNN model with more images can recognize the weapons and other threatening objects more accurately. We used the back-propagation algorithm, which has a linear time computational complexity, for training the CNN model.

The input value Θ going into a node i in the network is calculated by the weighted sum of outputs from all nodes connected to it, as follows:

$$\Theta_i = \sum(\omega_{i,j} * \Upsilon_j) + \mu_i \tag{3}$$

where $\omega_{i,j}$ is the weight on the connections between neuron j to i; Υ_j is the output value of neuron j; and μ_i is a threshold value for neuron i, which represents a baseline input to neuron i in the absence of any other inputs. If the value of $\omega_{i,j}$ is negative, it is tagged as inhibitory value and excluded because it decreases net input.

The training algorithm involves two phases: Forward and Backward phases. During the forward phase, the network's weights are kept fixed, and the input data is propagated through the network layer by layer. The forward phase is concluded when the error signal e_i computations converge as follows:

$$e_i = (d_i - o_i) \tag{4}$$

where d_i and o_i are the desired (target) and actual outputs of ith training image, respectively.

In the backward phase, the error signal e_i is propagated through the network in the backward direction. During this phase, error adjustments are applied to the CNN network's weights for minimizing e_i.

We used the gradient descent first-order iterative optimization algorithm to calculate the change of each neuron weight $\Delta\omega_{i,j}$, which is defined as follows:

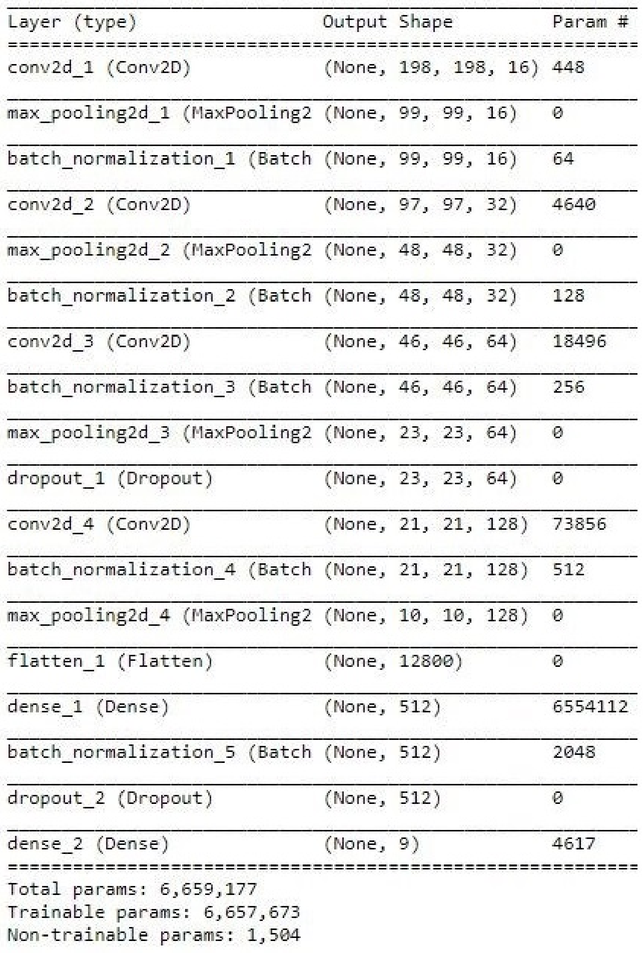
$$\Delta\omega_{i,j} = -\eta \frac{\delta\varepsilon(n)}{\delta e_j(n)} y_i(n) \tag{5}$$

where $y_i(n)$ is the intermediate output of the previous neuron n, η is the learning rate, and $\varepsilon(n)$ is the error signal in the entire output. $\varepsilon(n)$ is calculated as follows:

$$\varepsilon(n) = \frac{1}{2} \sum_j e_j^2(n) \tag{6}$$

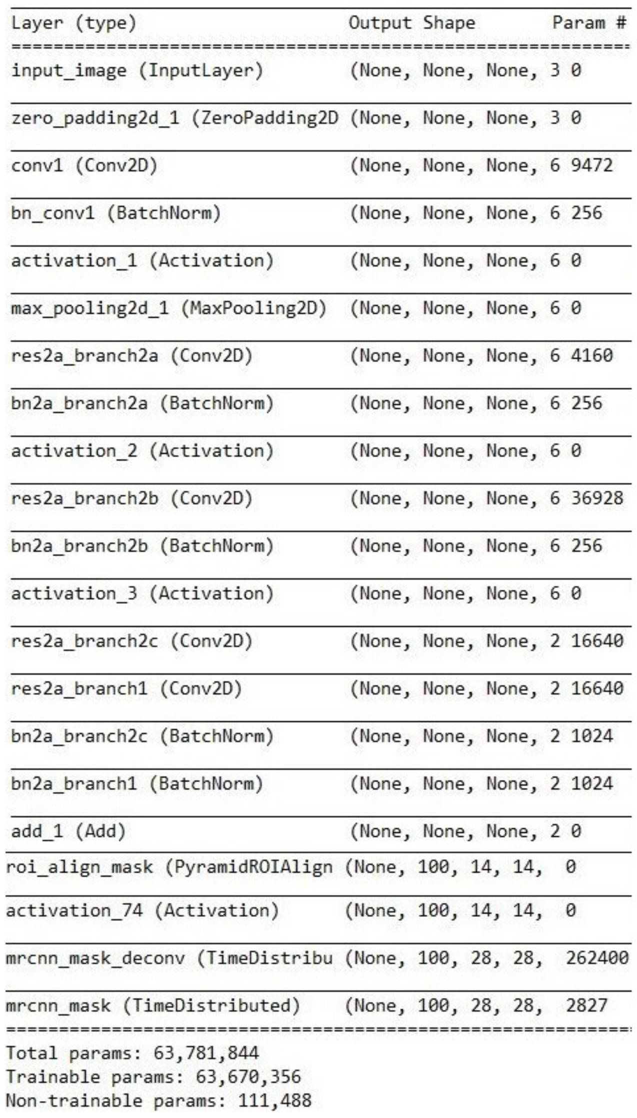
The CNN network has two types of layers: convolution and pooling. Each layer has a group of specialized neurons that perform one of these operations. The convolution operation means detecting the visual features of objects in the input image such as edges, lines, color drops, etc. The pooling process helps the CNN network to avoid learning irrelevant features of objects by focusing only on learning the essential ones. The pooling operation is applied to the output of the convolutional layers to downsampling the generated feature maps by summarizing the features into patches. Two common pooling methods are used: average-pooling and max-pooling. In this paper, we used the max-pooling method, which calculates the maximum value for each patch of the feature map as the dominant feature.

As shown in Figure 3, the output of every *Conv2D* and *MaxPooling2D* layer is a 3D form tensor (height, width,



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 198, 198, 16)      448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 99, 99, 16)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 99, 99, 16)        64
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 97, 97, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 48, 48, 32)        0
_________________________________________________________________
batch_normalization_2 (Batch (None, 48, 48, 32)        128
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 46, 46, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 46, 46, 64)        256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 23, 23, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 23, 23, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 21, 21, 128)       73856
_________________________________________________________________
batch_normalization_4 (Batch (None, 21, 21, 128)       512
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 10, 10, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 12800)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               6554112
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 9)                 4617
=================================================================
Total params: 6,659,177
Trainable params: 6,657,673
Non-trainable params: 1,504
```

FIGURE 3. The Structure of the CNN model



```
Layer (type)                    Output Shape         Param #
==============================================================
input_image (InputLayer)        (None, None, None, 3 0
______________________________________________________________
zero_padding2d_1 (ZeroPadding2D (None, None, None, 3 0
______________________________________________________________
conv1 (Conv2D)                  (None, None, None, 6 9472
______________________________________________________________
bn_conv1 (BatchNorm)            (None, None, None, 6 256
______________________________________________________________
activation_1 (Activation)       (None, None, None, 6 0
______________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, None, None, 6 0
______________________________________________________________
res2a_branch2a (Conv2D)         (None, None, None, 6 4160
______________________________________________________________
bn2a_branch2a (BatchNorm)       (None, None, None, 6 256
______________________________________________________________
activation_2 (Activation)       (None, None, None, 6 0
______________________________________________________________
res2a_branch2b (Conv2D)         (None, None, None, 6 36928
______________________________________________________________
bn2a_branch2b (BatchNorm)       (None, None, None, 6 256
______________________________________________________________
activation_3 (Activation)       (None, None, None, 6 0
______________________________________________________________
res2a_branch2c (Conv2D)         (None, None, None, 2 16640
______________________________________________________________
res2a_branch1 (Conv2D)          (None, None, None, 2 16640
______________________________________________________________
bn2a_branch2c (BatchNorm)       (None, None, None, 2 1024
______________________________________________________________
bn2a_branch1 (BatchNorm)        (None, None, None, 2 1024
______________________________________________________________
add_1 (Add)                     (None, None, None, 2 0
______________________________________________________________
roi_align_mask (PyramidROIAlign (None, 100, 14, 14,  0
______________________________________________________________
activation_74 (Activation)      (None, 100, 14, 14,  0
______________________________________________________________
mrcnn_mask_deconv (TimeDistribu (None, 100, 28, 28,  262400
______________________________________________________________
mrcnn_mask (TimeDistributed)    (None, 100, 28, 28,  2827
==============================================================
Total params: 63,781,844
Trainable params: 63,670,356
Non-trainable params: 111,488
```

FIGURE 4. The Structure of the Mask R-CNN model

channels). The width and height dimensions tend to shrink as we go deeper into the network. The third argument (e.g., 16, 32, 64 or 128) controls the number of output channels for each *Conv2D* layer. During the training phase, the CNN model generated more than 6.6 million trainable parameters.

Before moving the trained CNN model to the NCS 2 device, we had to convert it into an optimized Intermediate Representation (IR) model based on the trained network topology, weights and biases values. We used the Intel Open-VINO toolkit [17] to generate the IR model, which is the only format that the inference engine at NCS 2 accepts and understands. The conversion process involved removing the convolution and pooling layers that are not relevant to the inference engine at the stick. In particular, OpenVINO splits the trained model into two types of files: XML and Bin extension. The XML files contain the network topology, while the BIN files contain the weights and biases binary data.

B. CLOUD SIDE

At the cloud side, we used the pre-trained Mask R-CNN model [4] to construct segmentation masks on different Region of Interests (RoI) of weapon objects in the input image. The Mask R-CNN model extends the Faster R-CNN model [18] by adding a new module for predicting segmentation masks on each RoI, in addition to the existing modules for feature extractions, RoI classification, and bounding box regression. However, the operation of creating the segmentation masks is time-consuming, making the model relatively slower than Faster R-CNN and CNN models. Therefore, it is not applicable to be deployed on IoT devices at the edge.

The Mask R-CNN model adopts the same two-stage procedure of the Faster R-CNN model: object detection and semantic segmentation. Semantic segmentation, also called instance segmentation, is the operation of identifying the object outlines at the pixel level. First, Mask R-CNN uses the CNN feature generator to extract the essential features from the input image. Then, it creates a region proposal map for the target RoIs by using the CNN Region Proposal Network

(RPN), which transforms the input feature maps into candidate bounding boxes. Like CNN, Mask R-CNN applies the pooling operation to the bounding boxes to wrap them into a fixed dimension. It then feeds the pooled boxes into fully connected convolution layers to perform the classification and boundary box prediction.

Figure 4 shows the structure of our Mask R-CNN model, with over 63 million trainable parameters. As shown in the figure, we trained the model with four fully connected convolutional layers, one input layer, one classification layer, and two convolution layers to build the segmentation masks. The figure also shows the key operations' output shapes, such for instance segmentation, RoI creation and pooling, and RPN construction.

C. MOTION DETECTION

Given that most of the surveillance cameras are stationary, the background of its video stream is static and relatively stable over consecutive frames of the video. Therefore, we used the background subtraction [5] method to detect any moving weapon objects in the captured video in realtime. We represented the video background using the Gaussian Mixture model [19]. Our motion detection algorithm monitors the modeled background for any substantial changes, which corresponds to a motion on the video.

The Gaussian model works on pixel points (x, y) in the background regions of both the current frame $f_k(x, y)$ and previous frame $f_{k-1}(x, y)$ to obtain the difference values. If the absolute difference value $D_k(x, y)$ exceeds an experimentally predefined threshold value τ, motion is detected between the two consecutive frames. This operation is described formally as follows:

$$D_k(x,y) = \begin{cases} 1, & \text{if } |f_k(x,y) - f_{k-1}(x,y)| > \tau \\ 0, & \text{if } |f_k(x,y) - f_{k-1}(x,y)| \le \tau \end{cases} \tag{7}$$

In order to ensure the reliability of motion detection, we update the current background model $B_t(x, y)$ frequently. We continuously integrate the previous background frame $B_{t-1}(x, y)$ and the new incoming video frame $I_t(x, y)$ with $B_t(x, y)$. This adaptive background modeling is attained using a simple adaptive filter, as follows:

$$B_t(x,y) = (1-\beta)B_{t-1}(x,y) + \beta I_t(x,y) \tag{8}$$

where β is an experimentally adjustable parameter. When selecting a large coefficient value of β, the model updates $B_t(x, y)$ at higher speed; it also leads to the creation of artificial trails behind moving objects in the background model. If some objects remain stationary in the video for a long time, they become part of the background model $B_t(x, y)$.

IV. IMPLEMENTATION

This section presents the implementation details of Hawk-Eye at the camera and cloud sides.

TABLE 1. The Number of Images used in the Training, Validation, and Testing Phases Across Weapon Classes

Class	Training	Validation	Testing	Total
Grenade	756	302	151	1,209
Machine Gun	674	308	156	1,138
Pistol	1417	400	370	2,137
RPG	581	200	104	885
Mask	500	236	100	836
Motorcycle	750	350	240	1,390
Car	754	352	341	1,447
Knife	382	201	97	680
Bat	249	84	42	375
	6,063	2,433	1,601	10,097

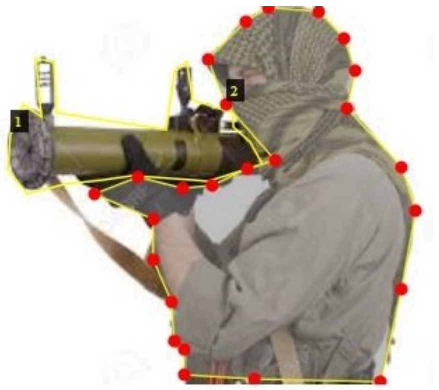


FIGURE 5. Annotated Image used in Training the Mask R-CNN Model

A. HAWK-EYE TRAINING

Although standard object detection datasets (e.g., Microsoft COCO [15]) exhibit volume and variety of examples, they are not suitable for gun detection as they annotate a set of object categories not include guns. Therefore, we collected more than labeled 10k weapon images for training the CNN and Mask R-CNN models from different sources such as Kaggle [20] and Google Web Scraper [21]. Many images in our dataset are in their natural environments because object detection is highly dependent on contextual information.

Our dataset is divided into three parts: training, validation and testing. Table 1 shows the number of images used in the three phases across the nine weapon classes. The number of images in each phase is determined based on the fine-tuned hyperparameters and structure of the CNN model.

For training the Mask-RCNN model, we used the VIA Annotation tool [22] to annotate 210 images for each weapon class. We had to manually define and describe spatial regions of the target objects in all training images. Figure 5 illustrates an annotated image with two threatening objects (i.e., RPG and Mask). As shown in the figure, the image region occupied by the RPG is defined using a polygon and described using the VIA annotation editor. We exported the spatial dimensions of each polygon in the input image in JSON format, which is then fed to the Mask-RCNN model as a one-dimensional array of the mask.

Both the CNN and Mask R-CNN models are imple-

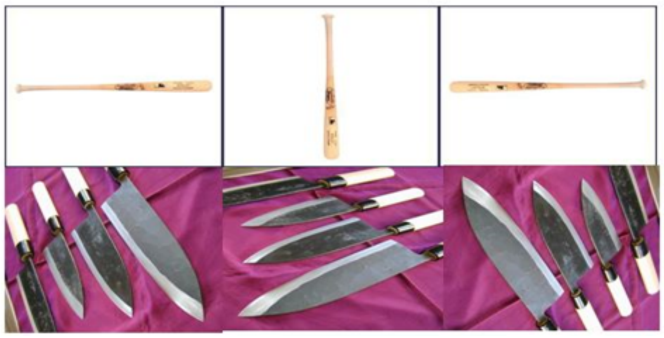


FIGURE 6. Dataset Augmentation

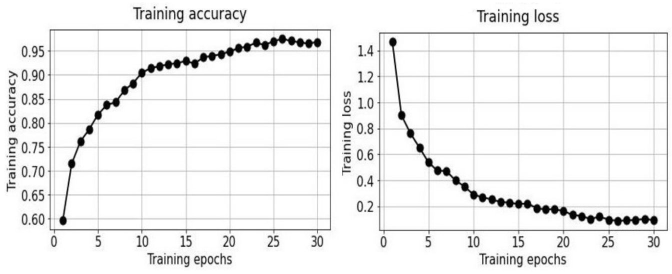


FIGURE 7. The Training Accuracy and Loss of the CNN Model

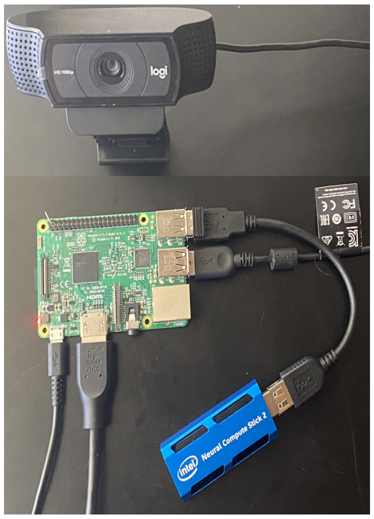


FIGURE 8. The Physical Implementation of Hawk-Eye at the Camera Side

mented using Keras development environment [23]. Keras is an open-source neural network library written in Python, which uses TensorFlow [24] as a back-end engine. Keras libraries running on top of TensorFlow make it relatively easy for developers to build and test deep learning models written in Python. For instance, we used the `keras.preprocessing.image.ImageDataGenerator` library to augment some images in our dataset for a few disease classes that lack an insufficient training set and have a lot of background noise, such as the knife and bat.

We used several random geometric transformations so that our model would never see twice the same image. This helps to avoid overfitting and allows the model to generalize better. As shown in Figure 6, the geometric transformations applied to these classes were horizontal flipping, $-45°$ to $45°$ rotation, 1.5x scaling, filling with nearest neighbor regions, zoom with range 0.2, width and height shifts with a relative scale of 0.3, and cropping some image manually.

The training images must have the same size before feeding them as input to the model. Our model was trained with colored (RGB) images with resized dimensions of 200×200 pixels. We set the batch size and number of epochs to be 150 images and 30 epochs, respectively. The model training was carried out using a server computer equipped with a 4.50GHz Intel Core™ i7-16MB CPU processor, 16GB of RAM, and CUDA GPU capability. The training phase took approximately 3 days to run 30 epochs on both models. We took a snapshot of the trained weights every 5 epochs to monitor the progress. The training error and loss are calculated using this equation:

$$M = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2 \tag{9}$$

where M is the mean square error of the model, y is the value calculated by the model, and x is the actual value. M represents the error in object detection and the construction of a mask over the wrong object.

Given our threat detector model is considered a multi-class classification problem, where it classifies the input image as belonging to one or more of the nine weapon classes, we used the softmax activation function at the output layer and the cross-entropy as the loss function. Figure 7 illustrates the calculated training error and loss graphically. As shown in the figure, the mean squared error loss is decreasing over the 30 training epochs, while the accuracy increases consistently. We can see that our model converged after the 25th epoch, which means that our dataset and the fine-tuned parameters were a good fit for the model.

B. PHYSICAL IMPLEMENTATION

Figure 8 shows the physical prototype implementation of Hawk-Eye on the camera side. Next, we describe the system components on the camera side.

- Raspberry Pi 3 Model B+ is used as an IoT edge processing unit on the camera side. The Raspberry Pi is equipped with a 64-bit quad-core processor running at 1.4GHz, dual-band 2.4GHz, and 5GHz wireless LAN. We installed Ubuntu 18.04 on the Raspberry Pi, which can host and run the Keras development environment. We connected an Intel NCS 2 and Logitech C920 webcam to the Raspberry Pi via the General Purpose I/O (GPIO) pins.
- Logitech C920 webcam is used to simulate the surveillance camera at the edge. We selected this webcam because it has several surveillance features such as high-

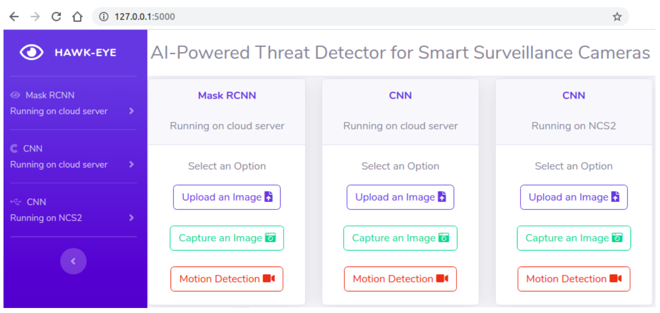


FIGURE 9. A Snapshot of the Web-based GUI of Hawk-Eye

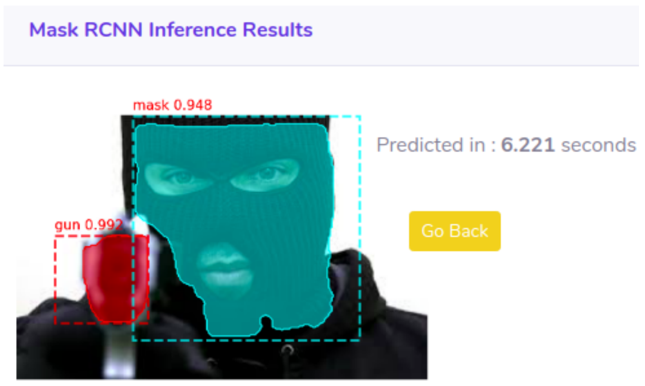


FIGURE 10. A Snapshot of the Inference Result of the Mask R-CNN Model at the Cloud Side

resolution live streaming with USB-C capability, high-dynamic-range capability, digital zoom, autofocus, and infrared sensor technology, as well as working smoothly with Ubuntu OS.

- Intel Neural Compute Stick 2 (NCS 2) is used as our inference engine for accelerating the rapid prototyping, validation, and deployment of our deep learning models at the edge. The NCS 2 device is equipped with Intel Movidius Myriad X Vision Processing Unit (VPU) and supports USB 3.0.
- Display screen is connected to the Raspberry Pi via its HDMI port as an output device to render the web interface to users.
- Keyboard and mouse are connected to the Raspberry Pi via its USB ports as input devices.

C. USER INTERFACE

The user interface is developed as a responsive, mobile-first, and user-friendly web application to enhance user experience using the system. We built the web application using Python Flask Framework, HTML5, CSS3, JavaScript, and JSON. All web pages are designed to be device-agnostic that can accommodate visitors using mobile devices, desktops, or televisions to visit the web site. Figure 9 shows the homepage of the web application, which is divided into three modules: Mask R-CNN running on the cloud server, CNN running on the cloud server, and CNN running on the NCS 2 at the edge. Each module allows users to upload saved images or capture fresh photos and videos using the webcam.

To run the web application on top of the CNN and Mask R-CNN models, we had to wrap both models, implemented on Keras, as a REST[2] API using the Flask web framework. In other words, the communication between Keras and Flask is coordinated through that REST API. When the user captures an image using the camera, Flask uses the `POST` method to send the image from the user browser to Keras via an HTTP header. The Flask service can be accessed by the IP address and port number of the web server without an extension as follows: http://127.0.0.1:5000 at both the Raspberry Pi at the camera side and the computer server at the cloud side.

Given that the Mask R-CNN model is not deployed at the camera side, our implementation supports transferring images from the camera side to the cloud side for utilizing the masking capability of the model. The Flask service, running on the Raspberry Pi, uses the REST API to send the image to the computer server on the cloud as an HTTP request via the `POST` method. Then, the classification results and the masked image are returned back to the Raspberry Pi as a JSON response.

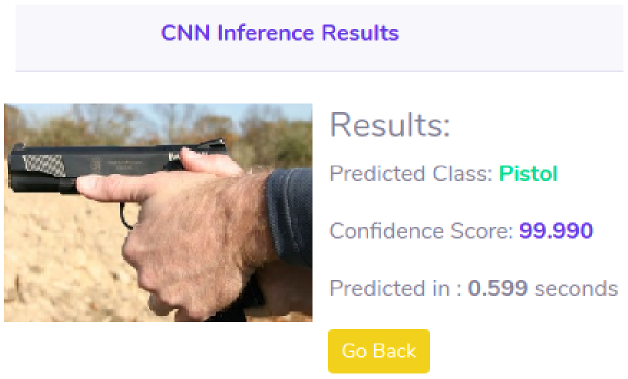
Figure 10 illustrates an example of the inference result of the Mask R-CNN model on the cloud side. As shown in the figure, the model created segmentation masks around the gun and face-mask with a confidence percentage of 99.2% and 94.8%, respectively. The processes of class prediction, mask construction, and displaying results took around 6 seconds. A significant portion of the 6 seconds is consumed to identify the regions of interest and create segmentation masks around them.

Figure 11 illustrates an example of the inference result of the CNN model on the camera side. The CNN model classified the pistol image correctly with a confidence score of 99.99%. With the help of NCS 2, the operations of class prediction and displaying results took around 0.6 seconds. This shows that Hawk-Eye can be used as a threat detector for live videos at the edge.

D. MOTION DETECTION

Our implementation of the motion detection module enables users to capture up to 12 images every minute in the video stream. A motion detection mask is also created around the moving objects in the video, as shown in Figure 12. We used the multi-threading capabilities in Python to enable parallel processing of images from the input camera. In other words, while the motion detection module generates the motion frames, our inference models are classifying the output images to the various weapon classes. The motion detection module used various Python OpenCV [25] libraries including CV2, Pandas and Shutils.

[2]Representational State Transfer (REST) is a software architectural style that is used to provide interoperability between heterogeneous computer systems connected via the internet.


CNN Inference Results

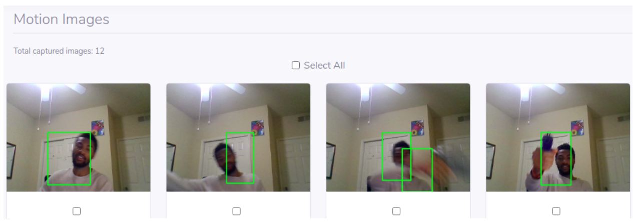
Results:

Predicted Class: Pistol

Confidence Score: 99.990

Predicted in : 0.599 seconds

FIGURE 11. A Snapshot of the Inference Result of the CNN Model at the Camera Side


Motion Images

Total captured images: 12

FIGURE 12. A Snapshot of the Motion Detection Result

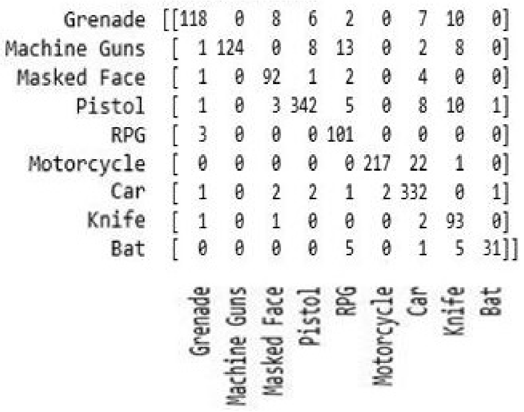
The video stream is sliced into individual frame images, which are stored into a `static_back` stack. A `motion_list` is also created to keep track of the start and end times of the detected motions. We then convert the RGB colored images stored in `static_back` to a gray-scale format before feeding them to `GaussianBlur`, which is a Gaussian filter for blurring images which helps in identifying the moving objects in frames.

The CV2 `absdiff` function is used to calculate the absolute difference value between the static background model and the current video frame. If the difference exceeds the predefined threshold value, which means that a new motion is observed, we start finding the contour of moving objects. A green-color rectangle is constructed around the found contours using the CV2 library. Finally, a new entry with the start and end times of the detected motion is appended to the `motion_list`.

V. EXPERIMENTAL EVALUATION

We experimentally evaluated our prototype implementation regarding classification accuracy and performance.

Figure 13 shows the confusion matrix for the CNN model at both the camera and cloud sides. It demonstrates that our model, in most cases, can differentiate between the weapon classes and achieve high levels of prediction accuracy. For the two most common types of weapons, pistol and knife, the model achieves accuracies above 92% and 95%, respectively. Knives appear easier to identify than pistols, which seems to make sense since pistols can be easily confused with other metal weapons.


```
      Grenade [[118   0   8   6   2   0   7  10   0]
 Machine Guns  [  1 124   0   8  13   0   2   8   0]
  Masked Face  [  1   0  92   1   2   0   4   0   0]
       Pistol  [  1   0   3 342   5   0   8  10   1]
          RPG  [  3   0   0   0 101   0   0   0   0]
   Motorcycle  [  0   0   0   0   0 217  22   1   0]
          Car  [  1   0   2   2   1   2 332   0   1]
        Knife  [  1   0   1   0   0   0   2  93   0]
          Bat  [  0   0   0   0   5   0   1   5  31]]
```

Grenade
Machine Guns
Masked Face
Pistol
RPG
Motorcycle
Car
Knife
Bat

FIGURE 13. The Confusion matrix for the CNN Model

As shown in the matrix, our model, in some cases, confuses between RPG and machine guns because they have a similar body structure. A similar situation happens between cars and motorcycles as they can share some common attributes such as pose, color, and type. Note that CNN can still identify face-masks quite well because of its discriminative features compared to the other classes in our dataset. Most notably, although bats and knives are considered non-linearly separable classes because of their similar shape properties (e.g., length, width and frame), our model was able to separate them effectively.

Figure 14 depicts the Receiver Operating Characteristic (ROC) curve for our CNN model. The ROC curve is a graphical plot that illustrates the model's ability to discriminate between the learned classes at all classification thresholds. The ROC curve is created by plotting the true positive rate against the false-positive rate at various threshold settings. The Area Under the ROC Curve (AUC) measures the two-dimensional area underneath the ROC curve from (0,0) to (1,1). AUC ranges in value from 0 to 1. A model whose predictions are 100% correct has an AUC of 1.0. As shown in the figure, most of the weapons classes are approaching an AUC's value of 1.0, which means that our CNN model could minimize most of the false positives.

The precision vs. recall curve, shown in Figure 15, summarizes the trade-off between the true-positive rate and the positive predictive value for our CNN model using different probability thresholds. Precision represents the positive predictive value of our model, while recall is a measure of how many true positives are identified correctly. As shown in the figure, the precision vs. recall curve tilts towards 1.0, which means that our CNN model achieves high accuracy while minimizing the number of false negatives.

The precision ratio describes the performance of our model at predicting the positive class. It is calculated by dividing the number of True Positives (TPs) by the sum of TPs and False Positives (FPs), as follows:

$$Precision = \frac{TPs}{TPs + FPs} \tag{10}$$

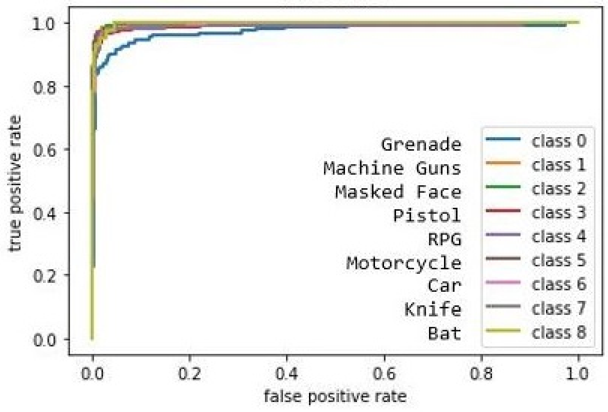


FIGURE 14. The ROC Curve of the CNN Model

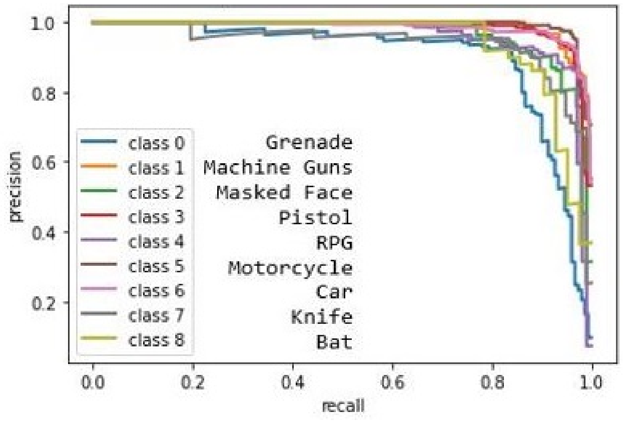


FIGURE 15. The Precision vs. Recall Curve of the CNN Model

The recall ratio is calculated as the ratio of the number of true positives divided by the sum of TPs and the False Negatives (FNs), as follows:

$$Recall = \frac{TPs}{TPs + FNs} \tag{11}$$

The overall classification accuracy of our model is calculated as the ratio of correctly predicted observation (i.e., the sum of TPs and True Negatives (TNs)) to the total observations (i.e., the sum of TPs, FPs, FNs, and TNs) using this equation:

$$Accuracy = \frac{TPs + TNs}{TPs + FPs + FNs + TNs} \tag{12}$$

Table 2 compares between the Mask R-CNN and CNN models in terms of classification accuracy and prediction time across the nine classes. The Mask R-CNN model achieved an overall average classification accuracy of 93.09%, and for the CNN model was a slightly better 94.20%. The average prediction time of the Mask R-CNN and CNN models was measured to be 4.18s (seconds) and 5.10ms (milliseconds), respectively. It is evident that the CNN model, running on the edge, outperforms the Mask R-CNN in most of the classes in terms of both the classification accuracy and prediction time.

TABLE 2. The Average Classification Accuracy and Prediction Time of Mask R-CNN vs. CNN

	Classification Accuracy		Prediction Time	
Class	**Mask R-CNN**	**CNN**	**Mask R-CNN**	**CNN**
Grenade	92.0%	94.0%	4.3s	5ms
Machine Gun	93.3%	100.0%	4.1s	6ms
Pistol	93.3%	98.9%	4.1s	5ms
RPG	91.4%	94.7%	4.1s	4ms
Mask	92.2%	93.2%	4.5s	7ms
Motorcycle	92.7%	94%	4.0s	4ms
Car	94.2%	90.2%	4.3s	6ms
Knife	91.8%	88.8%	4.3s	4.4ms
Bat	96.9%	94%	4.0s	4.5ms

Compared to CNN, Mask R-CNN is considered a significantly slower model because of the operations of identifying RoIs and creating segmentation masks around them. That is why Mask R-CNN is not adequate to be deployed on the camera side. In contrast, the CNN model, on average, requires around 5ms only to detect a security thread in an image captured locally from the camera. To put these numbers in some context, consider a commercial surveillance camera that can run at 30 Frames Per Second (fps), which means it can generate up to one image every 33ms. Our CNN model can process more than 6 images every 33ms, or support a super-resolution camera that runs at the rate of 180 fps.

We noted that the prediction accuracy of machine guns was 100% using the CNN model. With the fact that machine guns are the most used weapon in the deadliest mass shootings in the United States [1], Hawk-Eye can be used as a useful tool to mitigate mass shootings if deployed on the various surveillance camera at public places such as schools, malls, restaurants, etc.

VI. CONCLUSIONS

This paper presented the design and implementation of Hawk-Eye, an AI-enabled threat detector for realtime video surveillance. The developed prototype showed that Hawk-Eye could be deployed locally on the camera at the edge with high prediction accuracy and response time. We believe that this system would create a better opportunity for security personnel to detect various types of weapons in real-time, which could prevent a potential crime. The system implementation at the camera side used the Intel NCS 2 device to boost the processing capabilities of a Raspberry Pi 3 device for deploying the CNN model locally on the surveillance camera without cloud compute dependence. We also implemented Mask R-CNN, the state-of-the-art model for instance segmentation, on the cloud side to sketch bounding boxes around the weapon objects in the input image. A user-friendly interface was developed on top of both models to allow users to interact with the system conveniently at the camera and cloud sides. We also developed a motion detection module that can detect moving objects in surveillance videos in realtime. We carried out several sets of experiments for evaluating the performance and classification accuracy of our system, paying particular attention to the prediction time at the camera side. Most notably, our CNN model could process a number of images per second more than sextupled of the commercial fps. This proves that Hawk-Eye is suitable for real-time inference at the edge with offline generated deep learning models.

We expect that this research would increase the open-source knowledge base in the area of deep learning on the edge and real-time video surveillance by publishing the source code and dataset to the public domain: https://github.com/ahmed-pvamu/Hawk-Eye.

In ongoing work, we are looking into opportunities to generalize our approach to commercial surveillance cameras. Also, we are examining the composition of our work on multi-modal sensing, ModeSens [26], with ShareSens [27] to support the sharing of data between applications with multi-modal sensing requirements. ShareSens is a mechanism that opportunistically economizes on collecting sensor data by merging sensing requirements of multiple applications, thereby achieving significant power and energy savings. The problem of cumulating demand on sensors is a substantial barrier to simultaneously serving multiple sensing applications on battery-powered devices. Our work addressing this challenge has the potential of making it possible to serve numerous sensing-heavy applications on a single IoT device simultaneously. The composition of ModeSens and ShareSens will help support the sensing needs of a wide range of researches [28, 29, 30] and applications [31, 32]. Finally, experiments with more massive datasets are needed to study our system's robustness at a large scale and improve the prediction accuracy of the less performing classes.

ACKNOWLEDGMENTS

This research work is supported in part by the National Science Foundation (NSF) under grant # 2011330. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect NSF's views.

DATA AVAILABILITY STATEMENT

The data and source code that support the findings of this study are openly available at: https://github.com/ahmed-pvamu/Hawk-Eye

REFERENCES

[1] "Gun violence archive," accessed April 15, 2021. [Online]. Available: https://www.gunviolencearchive.org/

[2] G. F. Shidik, E. Noersasongko, A. Nugraha, P. N. Andono, J. Jumanto, and E. J. Kusuma, "A systematic review of intelligence video surveillance: Trends, techniques, frameworks, and datasets," *IEEE Access*, vol. 7, no. 1, pp. 457–473, 2019.

[3] J. Chen, K. Li, Q. Deng, K. Li, and P. S. Yu, "Distributed deep learning model for intelligent video surveillance systems with edge computing," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 1, pp. 1–8, 2019.

[4] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision*, ser. ICCV '17, 2017, pp. 2980–2988.

[5] S. Huang, "An advanced motion detection algorithm with video quality analysis for video surveillance systems," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 1, pp. 1–14, 2011.

[6] "Flask framework: A web-based framework written in python," accessed April 15, 2021. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/

[7] J. Lim, M. I. Al Jobayer, V. M. Baskaran, J. M. Lim, K. Wong, and J. See, "Gun detection in surveillance videos using deep neural networks," in *Proceedings of the IEEE Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, ser. APSIPA ASC '19, 2019, pp. 1998–2002.

[8] M. Grega, S. Łach, and R. Sieradzki, "Automated recognition of firearms in surveillance video," in *Proceedings of the IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support*, ser. CogSIMA '13, 2013, pp. 45–50.

[9] U. Navalgund and K. Priyadharshini, "Crime intention detection system using deep learning," in *Proceedings of the IEEE International Conference on Circuits and Systems in Digital Enterprise Technology*, ser. ICCSDET '18, 2018, pp. 1–6.

[10] Y. Zhou, L. Liu, L. Shao, and M. Mellor, "Fast automatic vehicle annotation for urban traffic surveillance," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 6, pp. 1973–1984, 2018.

[11] Y. Liu, Y. Yang, A. Shi, P. Jigang, and L. Haowei, "Intelligent monitoring of indoor surveillance video based on deep learning," in *Proceedings of the IEEE International Conference on Advanced Communication Technology*, ser. ICACT '19, 2019, pp. 648–653.

[12] H. Cui, Z. Wei, P. Zhang, and D. Zhang, "A multiple granular cascaded model of object tracking under surveillance videos," in *Proceedings of the ACM International Conference on Algorithms, Computing and Artificial Intelligence*, ser. ACAI ' 18, 2018.

[13] Ya Wang, Tianlong Bao, Chunhui Ding, and Ming Zhu, "Face recognition in real-world surveillance videos with deep learning method," in *Proceedings of the IEEE International Conference on Image, Vision and Computing (ICIVC)*, 2017, pp. 239–243.

[14] C. Ding and D. Tao, "Trunk-branch ensemble convolutional neural networks for video-based face recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 1002–1014, 2018.

[15] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *Computer Vision*, vol. 1405.0312, no. 1, 2014.

[16] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1701–1708.

[17] "Openvino: A toolkit for optimizing deep learning models," accessed April 15, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html

[18] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017.

[19] Indrabayu, R. Y. Bakti, I. S. Areni, and A. A. Prayogi, "Vehicle detection and tracking using gaussian mixture model and kalman filter," in *Proceedings of the International Conference on Computational Intelligence and Cybernetics*, 2016, pp. 115–119.

[20] "Kaggle: Machine learning and data science community," accessed April 15, 2021. [Online]. Available: https://www.kaggle.com/

[21] "Google web scraper," accessed April 15, 2021. [Online]. Available: https://chrome.google.com/webstore/detail/web-scraper/jnhgnonknehpejjnehehllkliplmbmhn?hl=en

[22] A. Dutta and A. Zisserman, "The via annotation software for images, audio and video," in *Proceedings of the 27th ACM International Conference on Multimedia*, ser. MM '19, 2019, pp. 76—79.

[23] "Keras: A python deep learning api," accessed April 15, 2021. [Online]. Available: https://keras.io/

[24] "Tensorflow: A machine learning platform," accessed April 15, 2021. [Online]. Available: https://www.tensorflow.org/

[25] "Opencv: A python library for real-time computer vision," accessed April 15, 2021. [Online]. Available: https://pypi.org/project/opencv-python/

[26] A. Abdelmoamen and N. Jamali, "A model for representing mobile

distributed sensing-based services," in *Proceedings of the IEEE International Conference on Services Computing*, ser. SCC '18, San Francisco, USA, 2018, p. 282–286.

[27] A. A. Moamen and N. Jamali, "ShareSens: An approach to optimizing energy consumption of continuous mobile sensing workloads," in *Proceedings of the 2015 IEEE International Conference on Mobile Services (MS '15)*, New York, USA, 2015, pp. 89–96.

[28] A. Abdelmoamen, D. Wang, and N. Jamali, "Approaching actor-level resource control for akka," in *Proceedings of the IEEE Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '18, Vancouver, Canada, 2018, pp. 1–15.

[29] A. A. Moamen and N. Jamali, "CSSWare: A middleware for scalable mobile crowd-sourced services," in *Proceedings of MobiCASE*, Berlin, Germany, 2015, pp. 181–199.

[30] A. A. Ahmed, A. Olumide, A. Akinwa, and M. Chouikha, "Constructing 3d maps for dynamic environments using autonomous uavs," in *Proceedings of the 2019 EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous '19)*, Houston, USA, 2019, pp. 504–513.

[31] A. Abdelmoamen, "A modular approach to programming multi-modal sensing applications," in *Proceedings of the IEEE International Conference on Cognitive Computing*, ser. ICCC '18, San Francisco, USA, 2018, pp. 91–98.

[32] A. A. Ahmed, S. A. Omari, R. Awal, A. Fares, and M. Chouikha, "A distributed system for supporting smart irrigation using iot technology," *Engineering Reports*, vol. e12352, pp. 1–13, 2020.