Application of Neural Networks for Weapon Detection

(MECE – 4520)

Final Project - Data Science for Mechanical Systems

Project Report Submitted by

PAVAN KASHINATH KAMATH
UNI: pkk2123

SACHIN FRANCIS DSOUZA
UNI: sfd2121

KIRAN SREENIVASA
UNI:km3714

SRIVATSA ELLORA
UNI: se2559

MUHAMMAD SAJAWAL SAGHIR
UNI: MS6339

Instructor Name
PROF. CHANGYAO CHEN

Date Submitted
December-14-2021

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Table of Contents

SL.NO.	TITLE	PAGE NO.
1.	Cover Page	1
2.	Table of Contents	2
3.	Introduction	3
4.	Image Data Set Generation	4
4.1	Result of Image Generation	5
5.	Methods Used	6
5.1	ResNet	6
5.2	GoogleNet	8
5.3	MobileNet	10
6.	Results	11
7.	Alternative Methods	13
7.1	Generative Adversarial Network (GAN)	13
7.2	You Only Look Once (YOLO)	14
8.	Future Work	14
9.	References	15
10.	Appendix	16

3. Introduction to the Problem statement

1. Every year, a large amount of the population reconciles gun-related violence all over the world.
2. In this work, we develop a computer-based fully automated system to identify basic armaments, particularly knives, handguns and rifles.
3. Recent work in the field of deep learning and transfer learning has demonstrated significant progress in the areas of object detection and recognition.
4. We have implemented ResNet, MobileNet and GoogleNet object detection model by training it on our customized dataset.
5. Applying this model in our surveillance system, we can attempt to save human life and accomplish a reduction in the rate of crime.
6. Additionally, our proposed system can also be implemented in high-end surveillance and security robots to detect a weapon or unsafe assets to avoid any kind of assault or risk to human life.

Figure 1: Introduction

4. Image Dataset Generation

Working on images and media files for data science is always a very exciting prospect. When it comes to using images, most of us retort to using an annotated dataset or use a set of images for manual annotation. Whilst this is a good approach, we must keep in mind that Machine Learning and Deep Learning algorithms need a very high volume of input training data to train a model. As easy as it is to find simple images, there would be a few cases where the dataset of images is not available or is very scarce, leading to lack of training data. And even if similar images were to be available, manually annotating them is a tedious and time-consuming task.

The idea of the project is to propose a systematic method to generate images that suit specific requirements. We focus highly on creating images that have all visual components that are expected of it and use algorithmic methods to obtain the results. Since the end goal is to generate dataset and not artistic images, we used royalty free image vectors for all the image components. The image components are chosen randomly and used, but since we know exactly which components we are adding to these images, we obtain the annotation details without additional effort. We used the python package Pillow for image manipulation. The flowchart below shows the entire process briefly.

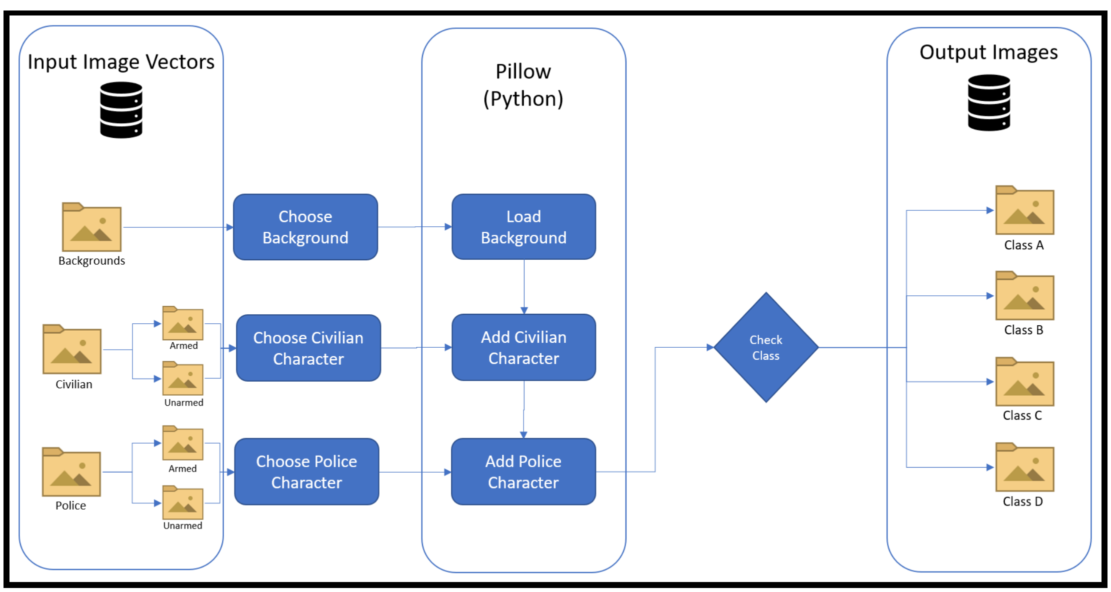

Figure 2: Image Generation Process Flowchart

We used three types of attributes: the background, the civilian character and the police character. For each of them we set a probability distribution in the form of a percentage to define how common each of the features will be. To ensure the balance a dataset, it is ideal to have equal probabilities for all the different options since we want our final model to predict all different types of cases, i.e., all the images in a particular attribute will have the same chance of being picked by the algorithm for image generation and all the generated classes of images will be equally distributed as well to ensure Equally likely events. In our project, we generated 20000 images in total.

With the above points in mind, we created 4 classes in our dataset:

- Class A : Civilian armed, Police armed (5000 images)
- Class B : Civilian unarmed, Police armed (5000 images)
- Class C : Civilian armed, Police unarmed (5000 images)
- Class D : Civilian unarmed, Police unarmed (5000 images)

4.1 Result of Image Generation

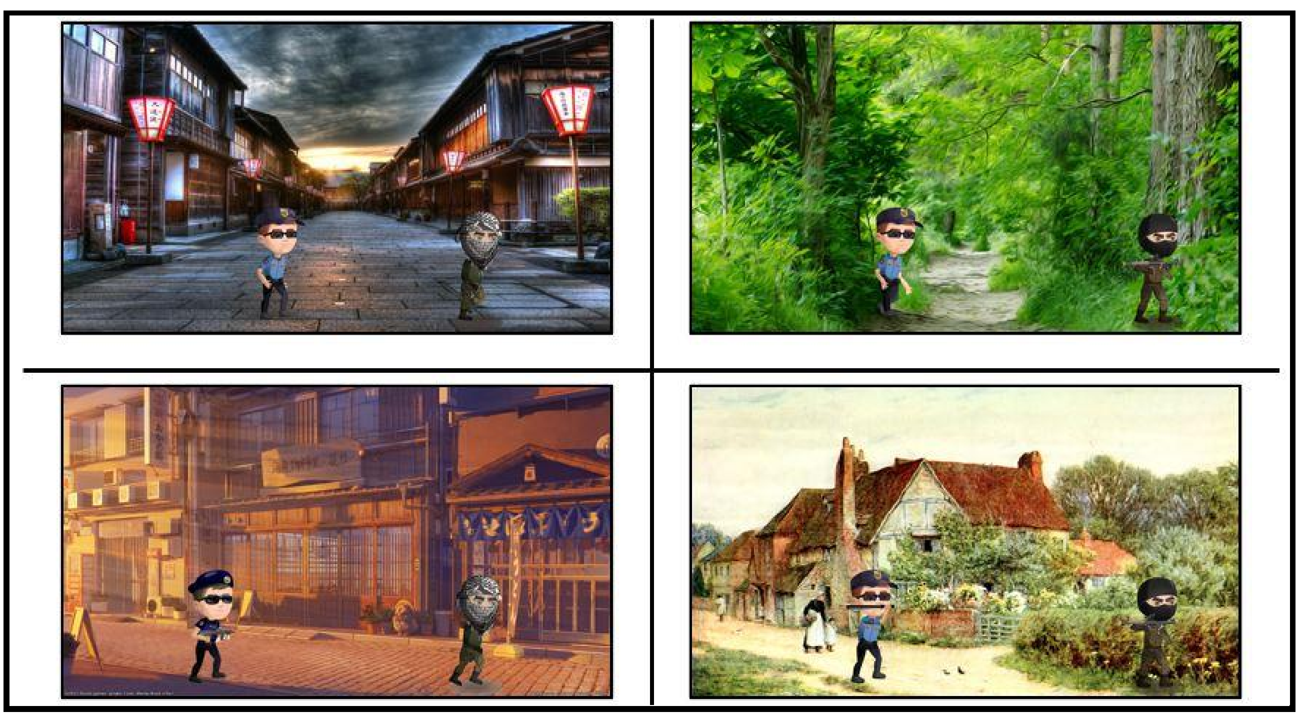

Figure 3: Image Generation

We created the images on a local machine using Python multithreading for parallelization. We created a csv file containing the annotations for all the file names for future reference. Using these images, we trained the models that we chose by splitting them into a test and train dataset. Following is a brief set of specifications of the image generation module:

- Image Resolution: 1920x1080
- No. of images: 20000
- No. of classes: 4
- Total Execution Time: ~6 minutes
- Packages used: Pillow, NumPy, tqdm, os, threading

5. Methods Used:

5.1 ResNet

For the purpose of training, this project employs complex images with intricate features to be extracted, and such features can be accurately extracted using pre-existing CNN architectures that have been tried and tested. Well known architectures such as ResNet, MobileNet, GoogleNet have proved to be successful in various competitions in the past, therefore been used in this project.

ResNet – this architecture was built by AI researchers with the objective of achieving the highest computational accuracy, close to 99%. It is a very deep network and hence computationally very expensive. The characteristic feature of ResNet is identity short connection aka residual mapping, where the input ‘x’ is added to the output of the network layer. This helps prevent the occurrence of vanishing gradients, a common issue in conventional neural networks that affects the accurate training of the first few layers, sometimes resulting in dead neurons. ResNet 50, 101 and 152 are the different variants, each with the respective number of layers.

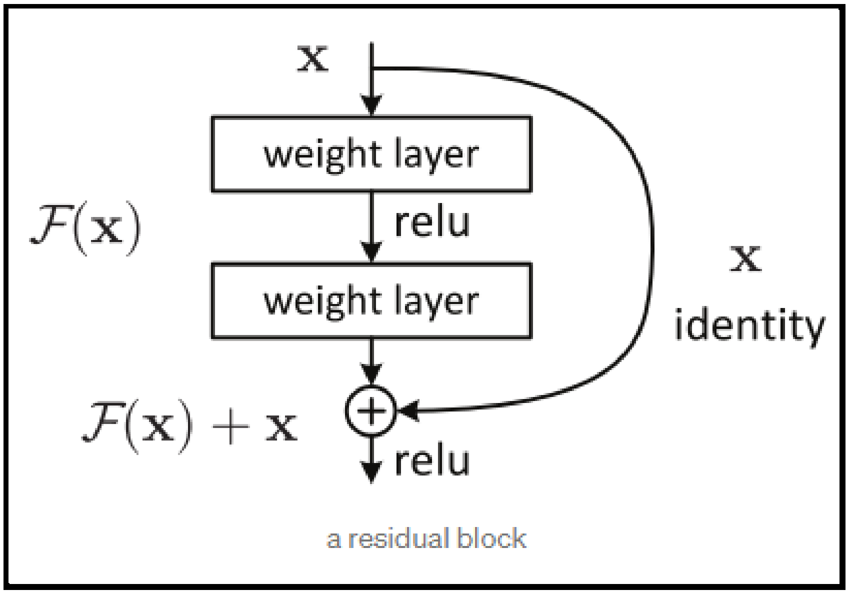

Figure 4: ResNet Block Diagram

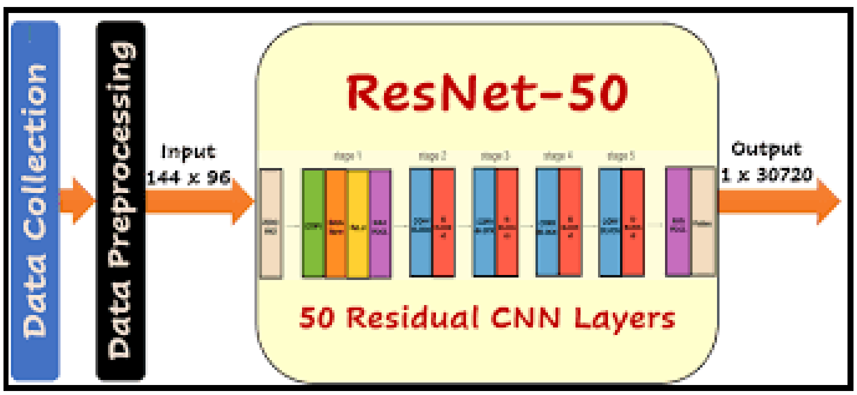

Figure 5: ResNet Architecture

Although ResNet provides near-perfect accuracy, the computational cost makes it infeasible in practical applications such as autonomous vehicles etc. This challenge is addressed by MobileNet, a computationally efficient algorithm with reasonably high accuracies. The characteristic feature of MobileNet is the 'depth wise separable convolution'. These are two-step convolutions – a depth wise convolution followed by a pointwise convolution. This feature greatly reduces the number of parameters required, thereby reducing the computational cost.

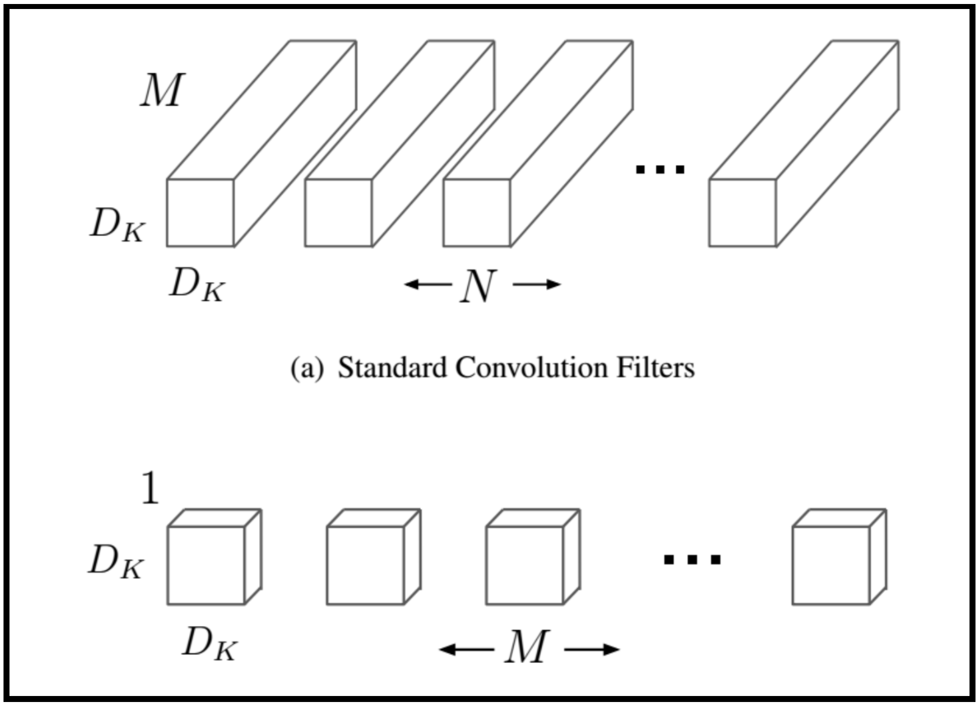

Figure 6: Depthwise Convolution Filters

5.2 Google Net

Another pre-existing architecture employed for image classification for the purpose of the project was GoogleNet. It is a 22-layer deep convolutional neural network that is a variant of the Inception network developed by researchers at Google. GoogleNet has the following versions Inception v1, Inception v2, Inception v3, Inception v4 and Inception Resnet. Inception v1 had 22 layers (27 including pooling layers) and it uses global average pooling at the end of the last inception module. While Inception v3 is a convolutional neural network for assisting in image analysis and object detection and got its start as a module for GoogleNet. The design of Inceptionv3 was intended to allow deeper networks while also keeping the number of parameters from growing too large.

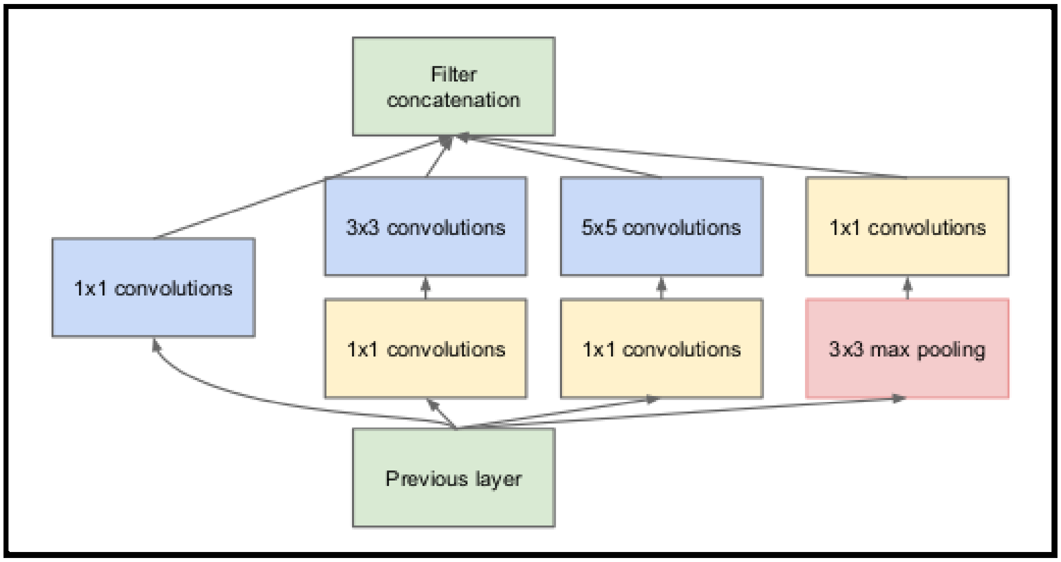

Figure 7: GoogleNet Block Diagram

Method:

Once the model is loaded, the user loads and preprocesses the images generated for prediction. Once the images are loaded in the right format, the next step is to feed to the network to train the model. Post training, we validate the results from testing the datasets.

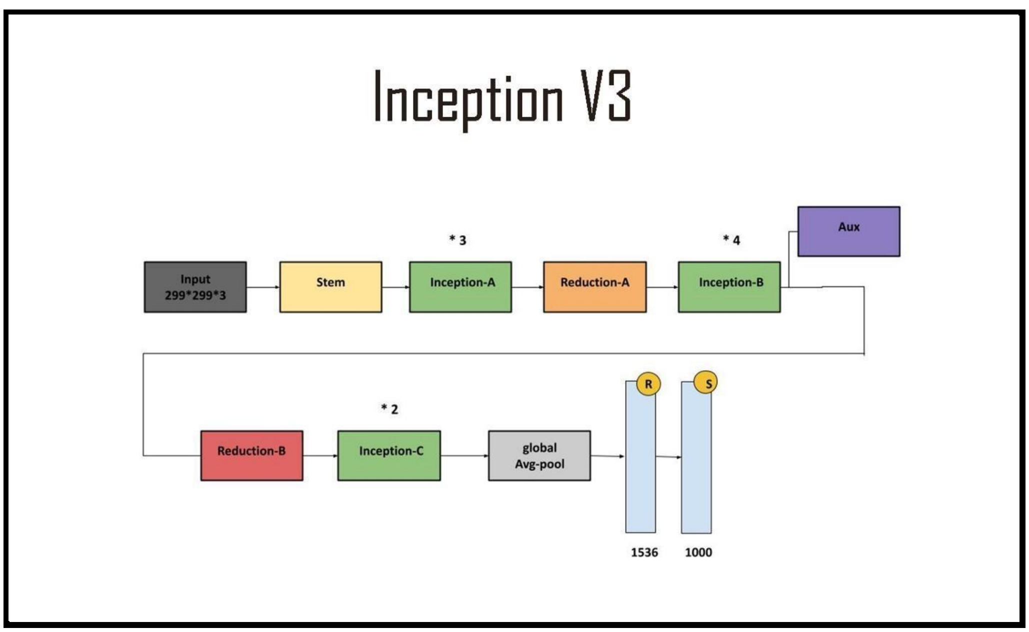

Figure 8: Inception v3 Block Diagram

5.3 MobileNet

MobileNet architecture is already small and computationally not very intensive, it has two different global hyperparameters to effectively reduce the computational cost further. One is the width multiplayer and another is the resolution wise multiplayer.

Width Multiplier: Thinner Models

For further reduction of computational cost, they introduced a simple parameter called Width Multiplier also referred to as α. For each layer, the width multiplier α will be multiplied with the input and the output channels(N and M) in order to narrow a network. Here α will vary from 0 to 1, with typical values of [1, 0.75, 0.5 and 0.25]. When α = 1, called as baseline MobileNet and α < 1, called as reduced MobileNet. Width Multiplier has the effect of reducing computational cost by α^2.

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F$$

Computational Cost: Depthwise separable convolution with width multiplier

Figure 9: Width Multiplier

Resolution Multiplier: Reduced Representation

The second parameter to reduce the computational cost effectively. Also known as ρ. For a given layer, the resolution multiplier ρ will be multiplied with the input feature map.

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$$

Computational cost by applying **width multiplier** and **resolution multiplier**

Figure 10: Resolution Multiplier

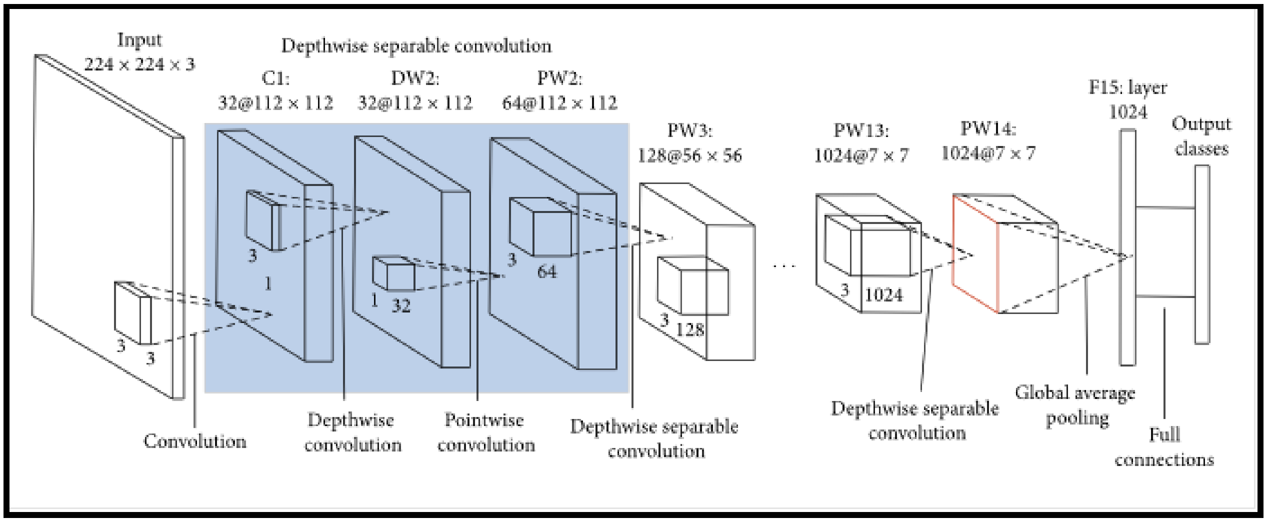

Figure 11: MobileNet Architecture

6. Results

After running our image classification using Hypernet v3, Mobile Net and Resnet 50 we got a figure for the accuracy of our results. The figure shows that Mobile Net was the most accurate with accuracy closer to 95% while HyperNet had an accuracy closer to 85%. We also obtained a figure for the training time to run each epoch. It can be seen from the figure 1 that MobileNet took the least amount of training time while HyperNet v3 took the most time.

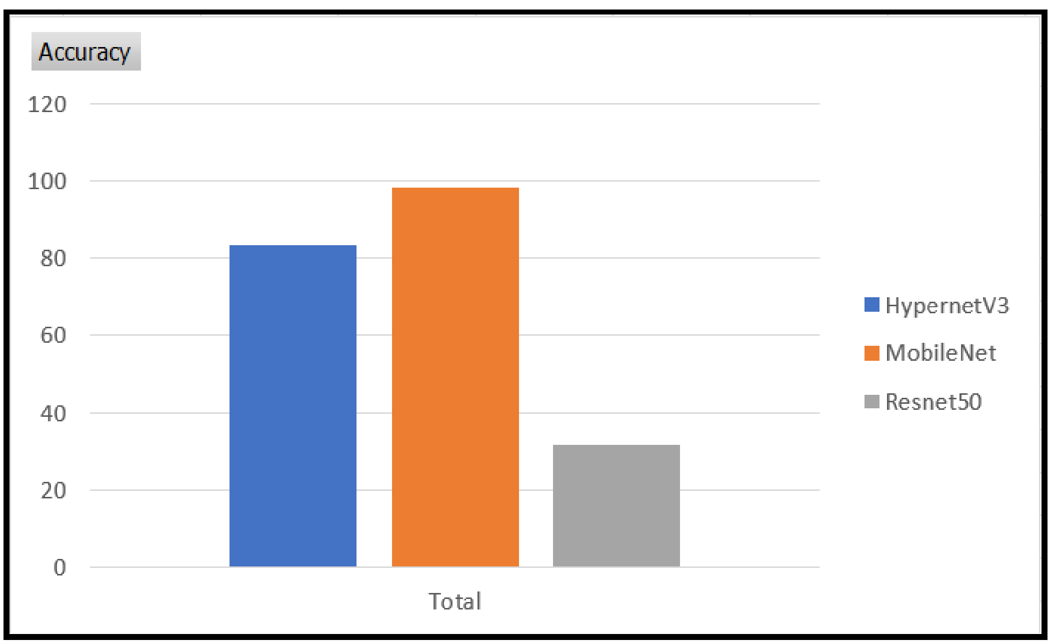

Figure 12: Accuracy Comparison of the models used

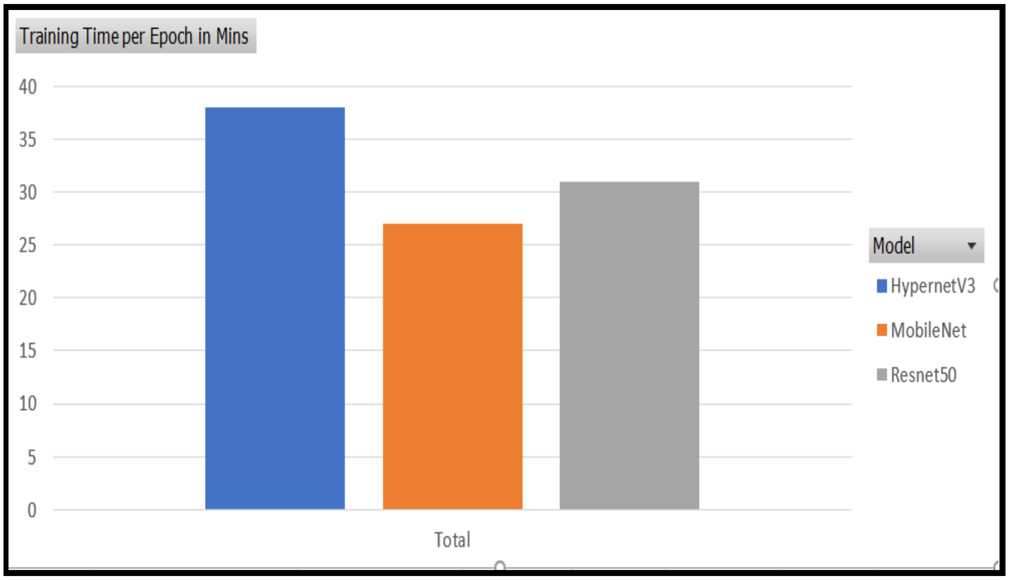

Figure 13: Training Time Comparison of the models used

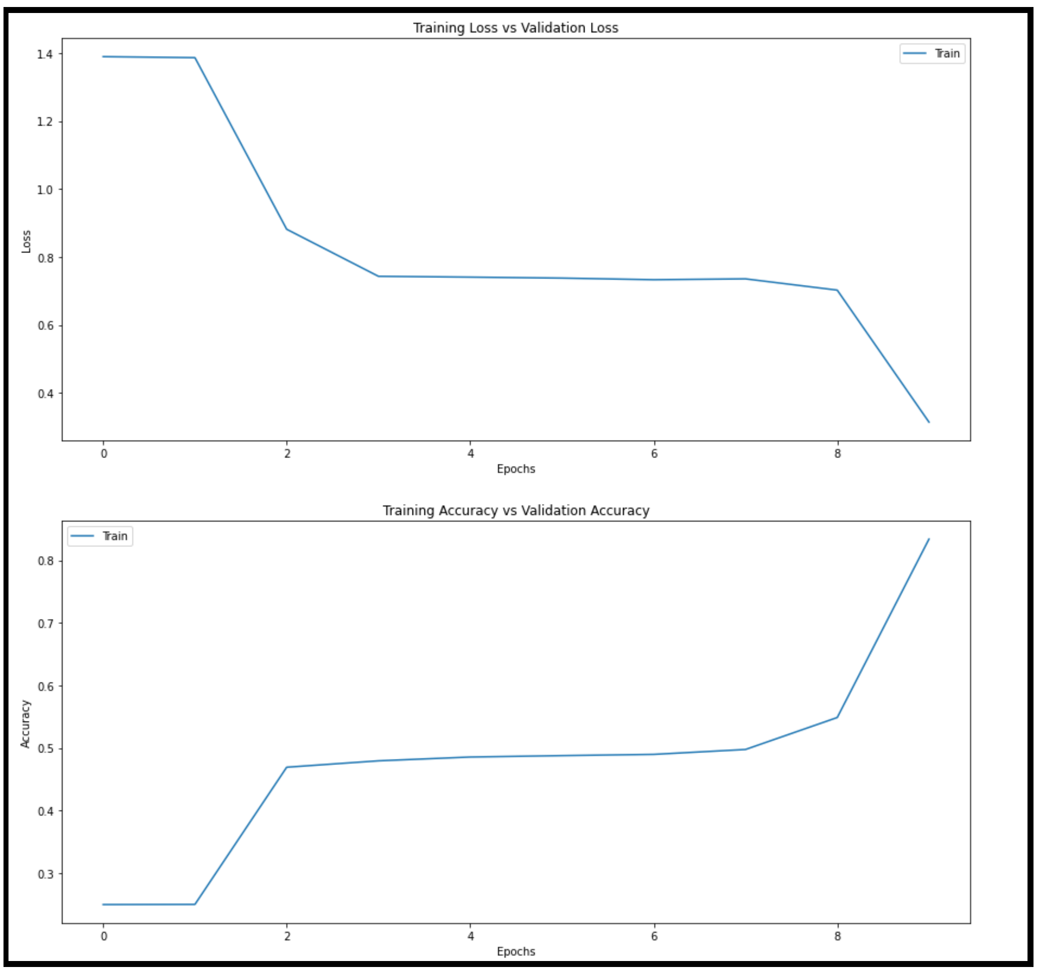

Figure 14: Hypernet v3 Training Loss and accuracy

7. Alternate Methods

There are other ways to generate the images as per our requirements in order to create our own dataset. One of the most common methods used is Generative Adversarial Network (GAN). This method can be used to generate a new synthetic image that looks least superficial to human beings. Apart from the methods discussed we could use You Only Look Once (YOLO) algorithm for weapon detection. Both of these methods are briefly discussed as follows:

7.1 Generative Adversarial Network (GAN)

This is a class of machine learning frameworks where two networks contest in the form of a zero-sum game in which the gain of one network is the loss of another network. This framework learns to generate a complete set of new data using the knowledge of statistics from the training set. In other words, if a GAN is trained on photographs then it can generate new photographs with realistic characteristics. GAN's can be used to generate photographs of human faces, human poses, objects such as weapons and backgrounds. Using GAN's the image data set can be generated which further can be used to train models of our choice to detect weapons.

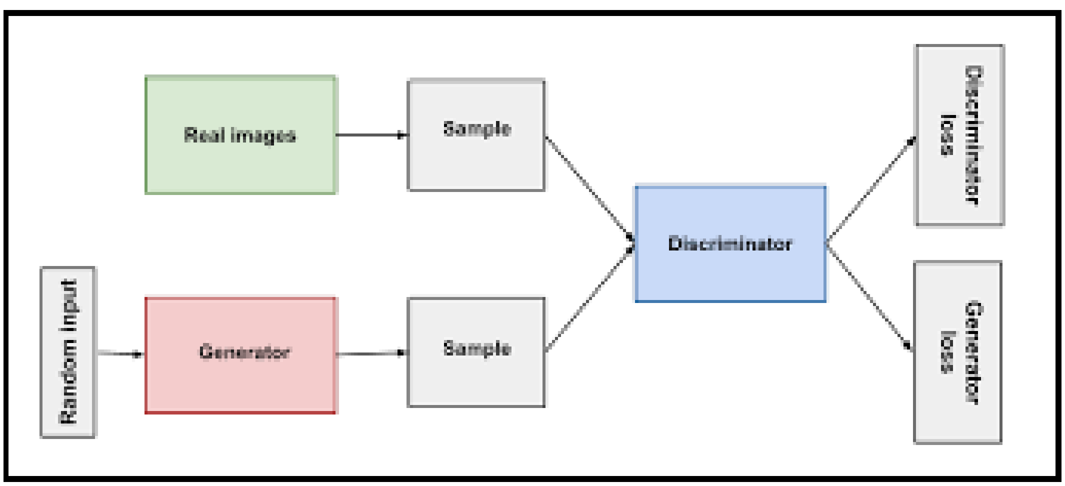

Figure 15: Overview of GAN Structure

7.2 You Only Look Once (YOLO)

This algorithm uses a Neural Network that detects and recognizes various objects in an image. Convolutional Neural Network is employed to detect the objects. YOLO requires only one single forward propagation to detect objects as the name suggests "You Only Look Once". YOLO can be used to detect objects in real time. This algorithm is widely popular because of its speed and accuracy. In our application we can use YOLO to detect weapons in real time.

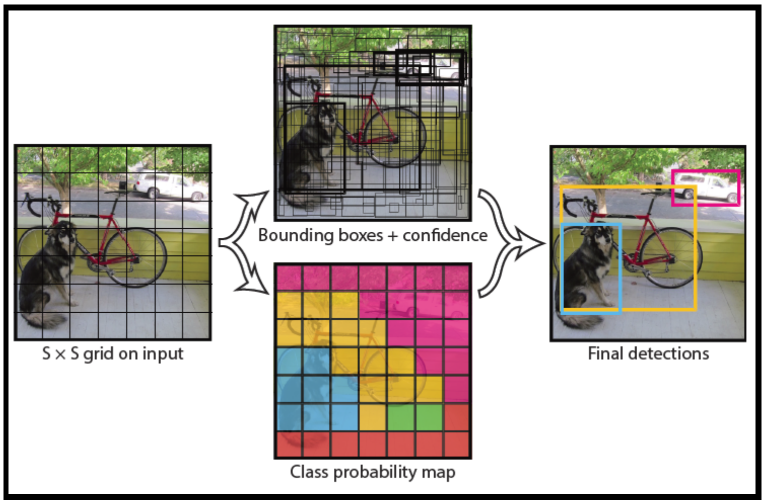

Figure 16: YOLO v1 Object Detection

8. Future Work

Our aim to develop a model that detects weapons using Neural Networks can be enhanced by incorporating the following points:

1. We can improvise our model by using realistic images with proper character orientation generated using one of the GAN techniques. This will enhance the accuracy of weapon detection using any of the object detection methods mentioned in our work.
2. Develop a high accuracy custom weapon detection model robust enough to detect weapons in lower quality images. The images generated by security cameras or CCTV are of lower quality, in order to implement the model in real time, our model must be robust enough to detect weapons in low quality images.
3. Train Real Time Object Detection Models like YOLO to detect weapons.
4. Implement Real Time Weapon Detection on CCTV cameras or Security Cameras.

5. If a weapon is detected the system can warn the security system or police, this enables security personnel and law enforcement to respond to threats in real-time.

9. References

1. He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
2. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).
3. https://towardsdatascience.com/algorithmically-generated-image-dataset-71aee957563a
4. Jose L. Salazar González, Carlos Zaccaro, Juan A. Álvarez-García, Luis M. Soria Morillo, Fernando Sancho Caparrini, Real-time gun detection in CCTV: An open problem, Neural Networks, Volume 132, 2020 (pp. 297-308).
5. Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement." arXiv preprint arXiv:1804.02767 (2018).
6. A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta and A. A. Bharath, "Generative Adversarial Networks: An Overview," in IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 53-65, Jan. 2018, doi: 10.1109/MSP.2017.2765202.

10. Code:

a. Image Generation:

```
from os import path, mkdir, makedirs
from PIL import Image
import pandas as pd
import numpy as np
from tqdm.notebook import tqdm
import concurrent.futures
import os
import threading
import random

#creating all the class output subfolders if not already present
output_folder = "output/a"
if not path.exists(output_folder):
   os.makedirs(output_folder)
output_folder = "output/b"
if not path.exists(output_folder):
   os.makedirs(output_folder)
output_folder = "output/c"
if not path.exists(output_folder):
   os.makedirs(output_folder)
output_folder = "output/d"
if not path.exists(output_folder):
   os.makedirs(output_folder)

#defining two variables based on the options available for creating the images
police_options = ['equipped_p','unarmed_p']
civilian_options = ['equipped_c','unarmed_c']

#counting the number of image vectors available for each of the image elements
backgrounds_count = 0
equipped_p_count = 0
unarmed_p_count = 0
equipped_c_count = 0
unarmed_c_count = 0
for root, dirs, files in os.walk(".", topdown=False):
   for name in files:
      #print(os.path.join(root, name))
      if "backgrounds" in os.path.join(root, name):
         backgrounds_count += 1
      if "equipped_c" in os.path.join(root, name):
         equipped_c_count += 1
      if "unarmed_c" in os.path.join(root, name):
         unarmed_c_count += 1
      if "equipped_p" in os.path.join(root, name):
```

```
            equipped_p_count += 1
        if "unarmed_p" in os.path.join(root, name):
            unarmed_p_count += 1
terrorist_count = equipped_c_count + unarmed_c_count
police_count = equipped_p_count + unarmed_p_count

def generate_image(classes, background_file_name, police_file_name, civilian_file_name, file_name): #function for generating one image based on arguments indicating presence of various elements
    background_file = path.join("backgrounds", background_file_name) #background file path
    background_image = Image.open(background_file) #open background image in pillow
    background_image = background_image.resize((1920,1080)) #set resolution for background image

    police_character_file = path.join("characters/police_chars", police_file_name) #police image file path
    police_character_image = Image.open(police_character_file)#open police image in pillow
    police_character_image = police_character_image.resize((int(police_character_image.size[0]/3),int(police_character_image.size[1]/3))) #set resolution for police image
    police_character_coordinates = (int(1920/2-police_character_image.width*1.6), int(1080-police_character_image.height*1.1)) #x, y co-ordinates for police image

    civilian_character_file = path.join("characters/civilian_chars", civilian_file_name)#civilian image file path
    civilian_character_image = Image.open(civilian_character_file)#open civilian image in pillow
    civilian_character_image = civilian_character_image.resize((int(civilian_character_image.size[0]/3),int(civilian_character_image.size[1]/3)))#set resolution for civilian image
    civilian_character_coordinates = (int(1920/2+civilian_character_image.width*1.6), int(1080-civilian_character_image.height)) #x, y co-ordinates for civilian image

    background_image.paste(police_character_image, police_character_coordinates, mask=police_character_image) #paste police image onto background
    background_image.paste(civilian_character_image, civilian_character_coordinates, mask=civilian_character_image) #paste civilian image onto background

    output_file = path.join("output/"+classes, f"{file_name}") #create output path based on arguments and classes
    background_image.save(output_file) #save to the output directory

def generate_random_imgs(total_imgs,num): #function for generating multiple images as passed through arguments
    df = pd.DataFrame(columns = ["image", "class", "civilian_present?", "civilian_armed?", "police_present?","police_armed?", "action"]) #create a table for  future reference
    for img in tqdm(range(num, num+total_imgs)): #running for the number of iterations as chosen
        background_character_number = random.randint(0,backgrounds_count-1) #choose a random background image
        background_file_name = "background" + str(background_character_number) + ".png" #find the background image in the input folder
        police_option = random.choice(police_options) #choose a random police image
        police_character_number = random.randint(0,globals()['%s_count'% police_option]-1) #randomly choose armed or unarmed police character
        police_file_name = police_option + "/" + police_option + "_" + str(police_character_number) + ".png" #find the police image in the input folder
        civilian_option = random.choice(civilian_options) #choose a random civilian image
        civilian_character_number = random.randint(0,globals()['%s_count'% civilian_option]-1) #randomly choose armed or unarmed civilian character
```

```
        civilian_file_name = civilian_option + "/" + civilian_option + "_" + str(civilian_character_number) + ".png"
#find the civilian image in the input folder

        civilian_armed = 1 if civilian_option == "equipped_c" else 0 #set variables for classification
        police_armed = 1 if police_option == "equipped_p" else 0 set variables for classification
        #create classes based on equipment status of the characters
        if civilian_armed==1 and police_armed==1:
            classes = "a"
        if civilian_armed==1 and police_armed==0:
            classes = "b"
        if civilian_armed==0 and police_armed==1:
            classes = "c"
        if civilian_armed==0 and police_armed==0:
            classes = "d"

        generate_image(classes, background_file_name, police_file_name, civilian_file_name, f"img{img}.png")
#generate the image

        data = [f"img{img}.png",classes, 1,civilian_armed,1,police_armed,"TBD"]
        s = pd.Series(data, index=df.columns)
        df = df.append(s, ignore_index=True) #append the row data of the specific image to the table

    df.to_csv('data' + str(num)+ '.csv', index=False) #creating a datafile for reference on inidividual objects

#using multithreading on 20 threads for parallel processing
if __name__ == "__main__":
    #generate_all_imgs()
    for i in range(1,number_of_threads+1):
        globals()['t%s'% str(i)] = threading.Thread(target=generate_random_imgs, args=(1000,(i-1)*1000))
    # starting threads
    for i in range(1,number_of_threads+1):
        globals()['t%s'% str(i)].start()
    # wait until threads are completely executed
    for i in range(1,number_of_threads+1):
        globals()['t%s'% str(i)].join()
    print("All threads successfully executed")

#execute the code
main()
```

b. Training and Testing code:

```
import tensorflow as tf
import tensorflow.keras
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
```

```
from tensorflow.keras.applications import imagenet_utils
from tensorflow.keras.layers import Dense,GlobalAveragePooling2D
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.applications.mobilenet import preprocess_input
import numpy as np
from IPython.display import Image
from tensorflow.keras.optimizers import Adam
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
from sklearn.metrics import confusion_matrix, classification_report,ConfusionMatrixDisplay,
classification_report

def prepare_image(file):
    img_path = ''
    img = image.load_img(img_path + file, target_size=(1920,1080))
    img_array = image.img_to_array(img)
    img_array_expanded_dims = np.expand_dims(img_array, axis=0)
    return tensorflow.keras.applications.{model_name}.preprocess_input(img_array_expanded_dims) ##changing
the model name as needed for multimodel comparison
##model_name used: MobileNet,ResNet50,ResNet152,InceptionV3

def load_image(img_path, show=False):
    img = image.load_img(img_path, target_size=(1920, 1080))
    img_tensor = image.img_to_array(img)                    # (height, width, channels)
    img_tensor = np.expand_dims(img_tensor, axis=0)         # (1, height, width, channels), add a dimension because
the model expects this shape: (batch_size, height, width, channels)
    img_tensor /= 255.                                      # imshow expects values in the range [0, 1]
    if show:
        plt.imshow(img_tensor[0])
        plt.axis('off')
        plt.show()

    return img_tensor

#define the base model
base_model = tf.keras.applications.{model_name}(weights = 'imagenet', include_top = False, input_shape =
(1920,1080,3)) ##changing the model name as needed for multimodel comparison

x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x) #we add dense layers so that the model can learn more complex functions and
classify for better results.
x=Dense(1024,activation='relu')(x) #dense layer 2
x=Dense(512,activation='relu')(x) #dense layer 3
x = Dense(1000, activation='relu')(x)
preds = Dense(4, activation = 'softmax')(x)
model=Model(inputs=base_model.input,outputs=preds)
model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics=['accuracy']) # Adam optimizer, loss
function will be categorical cross entropy, evaluation metric will be accuracy

for layer in base_model.layers:
    layer.trainable = False
```

```
#using the inbuilt function to read the train and test images
train_datagen=ImageDataGenerator(preprocessing_function=preprocess_input) #included in our dependencies

train_generator=train_datagen.flow_from_directory('/home/kiran/mece/train',
                                                 target_size=(1920,1080),
                                                 color_mode='rgb',
                                                 batch_size=2,
                                                 class_mode='categorical',
                                                 shuffle=True)


step_size_train=train_generator.n//train_generator.batch_size
history = model.fit_generator(generator=train_generator,
                   steps_per_epoch=step_size_train,
                   epochs=10)

test_generator=train_datagen.flow_from_directory('/home/kiran/mece/test',
                                                 target_size=(1920,1080),
                                                 color_mode='rgb',
                                                 batch_size=2,
                                                 class_mode='categorical',
                                                 shuffle=True)

evaluate = model.evaluate_generator(generator=test_generator)

#plotting the resulsts
fig, axs = plt.subplots(2, 1, figsize=(15,15))
axs[0].plot(history.history['loss'])
axs[0].plot(history.history['val_loss'])
axs[0].title.set_text('Training Loss vs Validation Loss')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend(['Train','Val'])
axs[1].plot(history.history['accuracy'])
axs[1].plot(history.history['val_accuracy'])
axs[1].title.set_text('Training Accuracy vs Validation Accuracy')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Accuracy')
axs[1].legend(['Train', 'Val'])

#testing to see if the prediction works
img_path = 'img23.png'
new_image = load_image(img_path,show=True)
pred = model.predict(new_image)
pred.argmax()

#predicting the class for all test images
y_pred = []
for i in tqdm(test_generator):
    pred = model.predict(i[0])
    label = np.argmax(pred)
    y_pred.append(label)
y_pred_4000 = y_pred[:4000] #for some reason, the predictor was running beyond the test generator max index.
Cutting it down to first 4000 values
```

```
#confusion matrix
cm = confusion_matrix(test_generator.labels, y_pred_4000 )
disp = ConfusionMatrixDisplay(cm)
disp.plot()
plt.show()
```