

REAL TIME GUN DETECTION CLASSIFIER

**RAJAS KAKODKAR
ETHAN ALBERTO
ANKITA SHIRODKAR
ASHTON RODRIGUES**

Dissertation submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER ENGINEERING

GOA UNIVERSITY



2018-2019

**DEPARTMENT OF COMPUTER ENGINEERING
PADRE CONCEICAO COLLEGE OF
ENGINEERING
VERNA GOA 403722**

REAL TIME GUN DETECTION CLASSIFIER

**RAJAS KAKODKAR
ETHAN ALBERTO
ANKITA SHIRODKAR
ASHTON RODRIGUES**

Dissertation submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER ENGINEERING

GOA UNIVERSITY



2018-2019

**DEPARTMENT OF COMPUTER ENGINEERING
PADRE CONCEICAO COLLEGE OF
ENGINEERING
VERNA GOA 403722**

PADRE CONCEICAO COLLEGE OF ENGINEERING

**Department of Computer Engineering
LOCATION IN VERNA, Verna 403722**



REAL-TIME GUN DETECTION CLASSIFIER

Bona fide record of work done by

RAJAS KAKODKAR	201510283
ETHAN ALBERTO	201510227
ANKITA SHIRODKAR	201510284
ASHTON RODRIGUES	201510093

Dissertation submitted in partial fulfillment of the requirements for the degree of
Bachelor of Engineering in Computer Engineering under the Goa University.

2018-2019

Approved By:

(Prof. ANUSHA PAI)
Project Guide

(Prof. AARTI BANDODKAR)
Project Co-Guide

(Prof. ANUSHA PAI)
HOD, Computer Department

(Dr. MAHESH B. PARAPPAGAUDAR)
Principal

Contents

Acknowledgements	iv
Abstract	iv
List of Figures	iv
List of Figures	v
1 Introduction	1
1.1 Motivation	1
1.2 Existing System	1
1.3 Transfer Learning	2
1.4 Approach	3
2 Literature Survey	4
2.1 A Brief History of Object Detection	4
2.1.1 Deep Learning	4
2.1.2 Object Detection	5
2.1.3 Related Work	9
3 System Requirements	11
3.1 Libraries	11
3.1.1 Model	11
3.1.2 Dataset	12
3.2 GPU	13
4 Dataset Description	15
4.1 Collection of Data	15
4.2 Annotations	17
4.3 Data Block API	19

5	Proposed Technique	23
5.1	Intuition	23
5.2	Technique	24
6	Implementation	26
6.1	Learning Rate Finder	29
6.2	1cycle Policy	30
6.3	Progressive Re-sizing and Fine-Tuning	31
6.4	Regularization	33
6.5	Splitting the dataset	34
7	Experiments	35
7.1	AP	35
7.2	Results	36
8	Results	37
8.1	Validation Set	37
8.2	Videos	40
9	Conclusion	46
10	Future Scope	47
10.1	Dataset	47
10.2	Splitting of the Dataset	47
10.3	AP scores	47
10.4	Application	48
	References	48

ACKNOWLEDGEMENT

We would like to take this opportunity to express our gratitude to everyone who supported us throughout the course of this project. We are thankful for their aspiring guidance, invaluable constructive feedback and friendly advice throughout the course of the project.

We would like to express deepest appreciation towards our principal Dr. Mahesh B. Parappagoudar for granting us the opportunity and providing laboratory facilities, thereby allowing us to complete our project on time and helping us through all the official work.

We are profoundly grateful to our Head of Department and Guide, Asst. Prof. Mrs. Anusha Pai and Co-Guide Asst. Prof. Mrs. Aarti Bandodkar for their support, expert guidance and continuous encouragement throughout the project.

At last we must express our sincere heartfelt gratitude to all the staff members of Computer Engineering Department who helped us directly or indirectly during this course of work.

ABSTRACT

In spite of the widespread existing use of surveillance cameras in public places, gun crime rates keep soaring relentlessly upward. Tailored convolutional neural networks designed to enhance training and testing accuracy greatly aid in real-time detection of objects. In our research, we propose a Real-Time Gun Detection system using Deep Learning. We first created a dataset of 1045 natural images with pistols and rifles. The biggest challenge in our dataset was the varying scales of the guns in the images. Since Convolutional Neural Networks (CNNs) use hierarchical feature learning, critical information can get smoothed out in the early layers and very small objects may go undetected. We chose to use RetinaNet as it carries out deep feature extraction. We classified our images as hard or easy before training. Our results show that by splitting the training data in this way, RetinaNet resulted in AP (Average Precision) score of 0.446985 for pistol class and 0.071868 for rifle class whereas training it with the entire dataset in one go resulted in AP score of 0.126551 for pistol class and 0.032240 for rifle class on NVIDIA Tesla T4. The proposed technique also takes less time to train.

Keywords: CNN; Gun detection; RetinaNet; Hard and easy images;

List of Tables

4.1	Dataset	16
7.1	Results	36

TABLE NO.	NAME	PAGE
Table 4.1	Dataset	(12)

Chapter 1

Introduction

1.1 Motivation

Violence in all forms is rampant in towns and villages all over the world. Among others, gun violence is the most prevalent of all, thus resulting in heavy casualties. Each day, numerous lives are lost on account of mass shootings, terrorist attacks, armed assaults and robberies. In the face of a worsening scenario, many countries like the United States of America have adopted liberal gun laws. Gun detection and control is of main concern, apart from enforcement of more stringent gun laws as gun violence keeps growing at an alarming rate. Despite presence of numerous surveillance cameras in public places, gun crimes still take place due to ineffective monitoring of multiple video streams, time lost in detection of hand-held weapon by manual monitoring of footage and security lapse caused by human oversight. Tackling this serious problem and taming the high numbers associated with gun violence involves bringing in major improvements in the form of strengthened security systems.

1.2 Existing System

Despite extensive use of security cameras in most places, the crimes involving guns have not reduced. The existing system for weapon detection in public places relies largely on manual monitoring of video streams by security personnel. Surveillance of multiple streams is inefficient and ineffective as supervising them can be a daunting task and cost prohibitive. In other controlled public places like airports and shopping malls, metal detectors are used to keep a check on unauthorized possession of weapons.

As an effort to strengthen security, a real-time gun detection system can be developed which would take live feed from surveillance cameras and notify the security personnel on detection of a hand held weapon (pistol or rifle) in the feed.

1.3 Transfer Learning

While classifiers output probabilities over classes, object detectors output both probabilities of class membership and the coordinates that identify the location of the objects. Conventional machine learning and deep learning algorithms, so far, have been traditionally designed to work in isolation. These algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones.

For problems like image recognition, transfer learning has been proved to be very efficient. An existing convolutional neural network commonly used for image recognition of other object classes is taken, and is adjusted to train with the target object class. Training begins with a pre-trained model, instead of starting from the beginning with no aid. Transfer learning is an optimization, a shortcut to saving time or getting better performance. Transfer learning is useful when dataset is limited for a new domain. It enables development of skillful models that simply cannot develop in the absence of transfer learning. The choice of source data or source model is an open problem and may require domain expertise and/or intuition developed via experience. Despite having access to limited images of the target object class, most of the low-level and mid-level feature definitions can be gained by using an existing CNN such as ResNet[10], pre-trained with more than 1 million images on ImageNet dataset[10].

Customizing a pre-trained model using fine tuning involves adding new classifier layers to the end and training those, while keeping the rest of the model frozen. Before unfreezing, the saved weights of the fine-tuned layers are loaded. Next, the entire model is trained jointly by unfreezing the initial layers. This allows "fine tuning" of the higher-order feature representations in the base model in order to make them more relevant for the specific task. The details about fine tuning are discussed in the subsequent chapters.

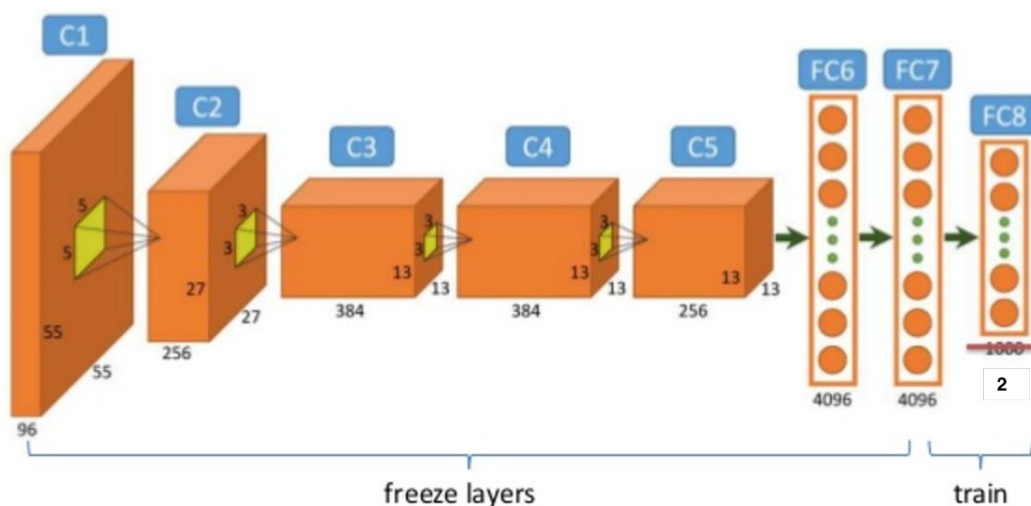


Figure 1.1: Transfer Learning

(Source:<https://www.mdpi.com/2076-3417/8/10/1768/html>)

1.4 Approach

A dataset of pistols and rifles was built from scratch and annotated manually using LabelImg tool[22]. The dataset consists of a total of 1045 images and a .json file consisting of the annotations. We have tried to build a state-of-the-art Object Detection model with limited amount of data. The model was trained on RetinaNet Architecture[21] using fastai library[6]. To test the performance, we have used the Average Precision (AP) score to evaluate the performance of each class. While training a model, usually to achieve a high AP score the model has to be trained for longer time. We propose a technique of splitting the dataset before training the model which results in better AP score and takes less time to train.

Chapter 2

Literature Survey

2.1 A Brief History of Object Detection

2.1.1 Deep Learning

Object Detection, as the name suggests, is a technique of recognizing as well as localizing objects in an image. The research carried out in this field has been growing rapidly due to advancements in Deep Learning. The foundations of Deep Learning are laid out in the Universal Approximation Theorem. This theorem states that any function can be approximated with just one hidden layer in a neural network. This follows that the same kind of approximation can be achieved for any neural network that goes deeper (contains multiple layers). Stacks of affine functions (matrix multiplications) and non linear functions can approximate any function very closely. It is because of this reason that deep learning architectures are going deeper.

For image data, a special type of affine function, Convolution is used. Convolutional Neural Networks (CNNs) convolve filters/kernels over the image during training phase. These filters are matrices of size 3x3 or 5x5 or 7x7. Every filter performs a sum of products (convolution) operation with the corresponding matrix of image pixels for the entire size of the image. The results of these convolutions are weights which are optimized during the training phase. The number of filters used and the optimization algorithm used, define the performance of CNN.

Any Neural Network is made up of three main components: data, architecture and the loss function. The loss function compares the predictions of the neural network with the ground truth and then tunes the weights. The architecture is mainly dependent on the combinations of the affine functions and the nonlinear functions. In case of CNN, the affine function is Convolution whereas the nonlinear function

used is Rectified Linear Unit (ReLU).

2.1.2 Object Detection

With recent advancements in Computer Vision, many researchers have developed efficient algorithms for Object Detection using CNNs. The algorithms such as Faster R-CNN[11] followed a two pass approach where the first stage used the classical computer vision approaches to find edges and changes of gradients to guess which parts of the image may represent distinct objects. Then second phase fit each of those into a convolutional neural network which was basically designed to figure out if that is the kind of object we are interested in. The algorithms like YOLO[8] follow a single phase approach and results in a very efficient algorithm.

We tried to study two algorithms in detail. Single Shot Multibox Detector (SSD)[24] and Focal Loss for Dense Object Detection[21] (RetinaNet).

SSD

SSD[24] is a single pass approach. The core concept is to divide the image into a number of anchor boxes of different sizes and ratios. Every anchor box is evaluated to check how much of it overlaps with the object of our interest. Consider an image showing the ground truth/objects of our interest.

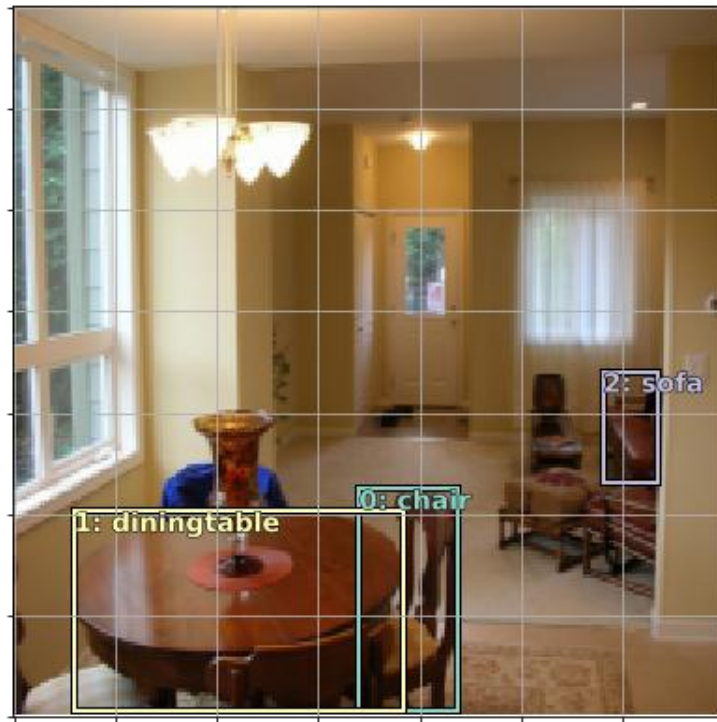


Figure 2.1: Ground Truth
(Source: <https://bit.ly/2KAErWs>)

There are three objects to be detected. This image is divided into anchor boxes. For simplicity consider only 16 anchor boxes.

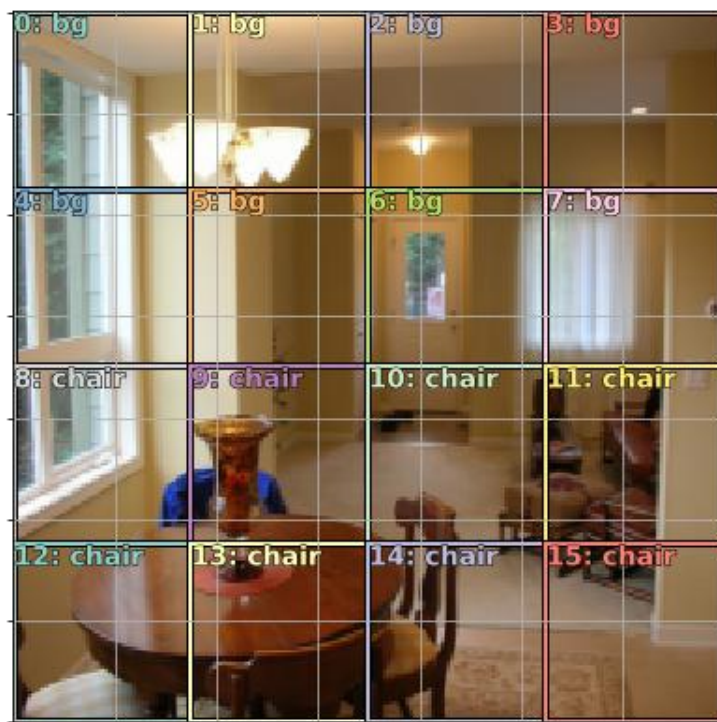


Figure 2.2: Anchor Boxes
(Source: <https://bit.ly/2KAErWs>)

Each one of these 16 boxes is considered and matched with one of these three ground truth objects that has the highest amount of overlap with a given square. A standard function for this is called Jaccard index (IoU).

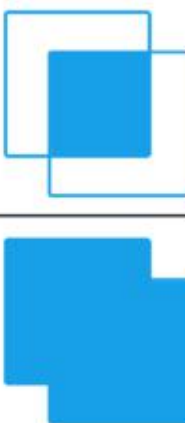

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Figure 2.3: Jaccard
(Source: <https://bit.ly/2KAErWs>)

This task of matching is done by the loss function. Classification along with regression is used to predict the class label of the object and the four bounding box coordinates respectively. This algorithm works very well yet we chose not to implement it. SSD[24] suffers from a drawback; it performs well only for large sized objects. Our dataset consisted mainly of small sized objects hence we chose to implement the RetinaNet[21] algorithm.

RetinaNet

Focal Loss for Dense Object Detection[21] research paper fixed the problem of detecting small sized objects by modifying the loss function used in SSD[24] and using a more complex architecture. The Binary Cross Entropy Loss function used in SSD[24] is multiplied by an exponential term which optimizes the algorithm and is called the Focal Loss function.

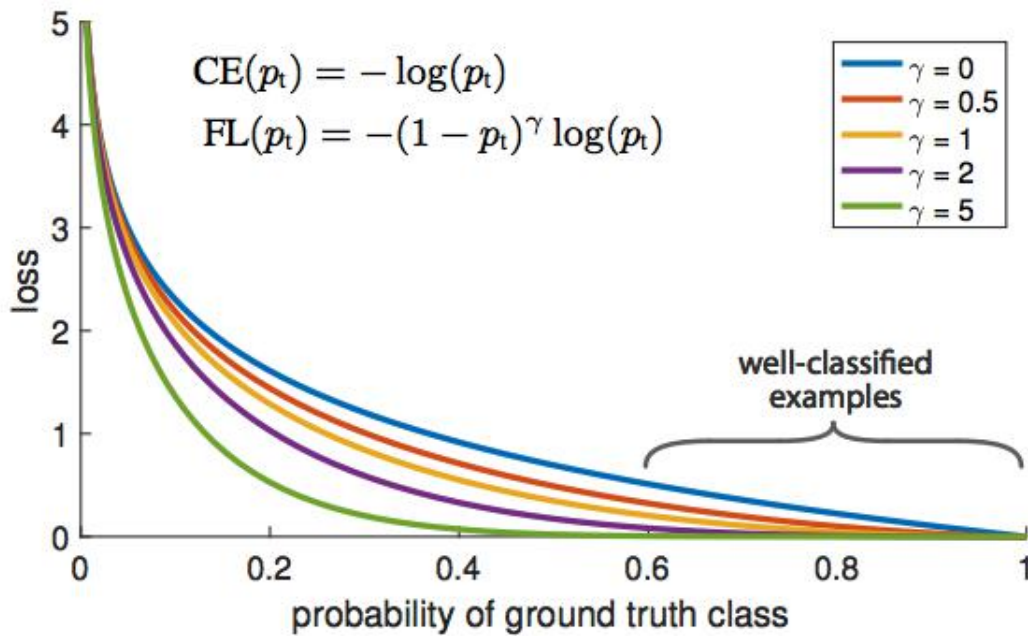


Figure 2.4: Focal Loss

(Source: <https://arxiv.org/abs/1708.02002v2>)

The blue line is the binary cross entropy loss and the purple line corresponds to Focal Loss. The importance of this loss function can be shown as follows: Suppose the Neural Network predicts an object with probability 0.4 with respect to the above graph. Binary Cross Entropy loss will report a loss between 1 and 2 whereas Focal Loss will report a loss less than 1. This will encourage the Neural Network to go ahead with the prediction which helps in predicting small sized objects.

The second most important aspect of RetinaNet[21] is the architecture. It follows a Feature Pyramid Network[20].

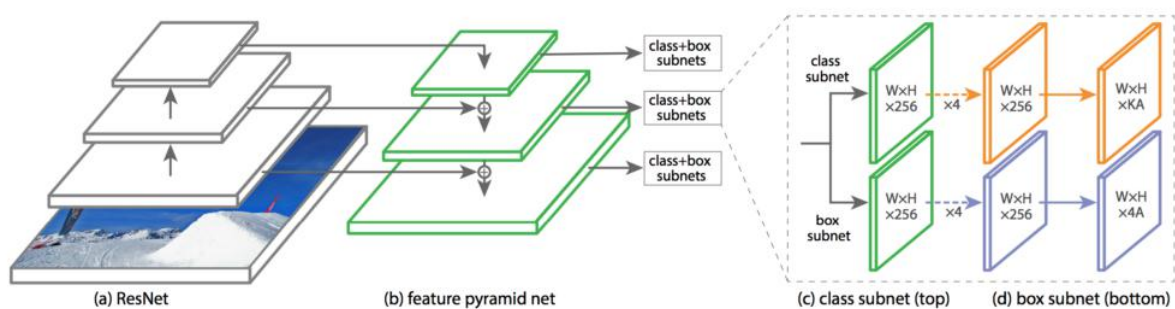


Figure 2.5: RetinaNet Architecture

(Source: <https://arxiv.org/abs/1708.02002v2>)

In this architecture, the CNN first goes deep and then wide. In the left branch the image is progressively reduced and in the right branch it is upsampled again using bilinear upsampling. There exist lateral connections between the two branches, which produces the feature maps at each level for our final predictions. The top feature maps are responsible for large objects whereas the bottom feature maps are responsible for small objects. These feature maps are given to the classification and regression subnets. The downsampling of the image is achieved by using a pretrained model. In this case we use ResNet50[10] which is pretrained on ImageNet[10] dataset.

2.1.3 Related Work

In the direction of reducing the soaring rates of gun crime, many other projects have forayed into research on weapon detection in image and video data.

The research paper, Developing a Real-Time Gun Detection Classifier by students of Stanford University [7] highlights the ways in which real-time detection and classification of weapons was carried out using a Tensorflow-based implementation of the Overfeat network. The dataset used for this purpose was not made public. It has also been shown by researchers that RetinaNet[21] architecture works better than Overfeat.

A paper which goes by the title 'Automatic Handgun Detection Alarm in Videos Using Deep Learning' by Olmos, Tabik and Herrera [17] delves into automatic gun detection in surveillance videos, triggering an alarm if the gun is detected. Their approach uses R-CNNs as a means to implement weapon detection. R-CNNs and Faster R-CNNs[11] result in high accuracy but are two-phase approaches resulting in inefficiency. SSD[24] and RetinaNet[21] have been proved to be better algorithms than R-CNNs and Faster R-CNNs[11].

A study performed by Joseph Redmon et al [8] brings to light an improved object-detection system, YOLO9000 that can detect over 9000 object categories. In the said paper, YOLOv2 is compared with YOLO9000. The novel method trains YOLO9000 simultaneously on the COCO detection dataset and the ImageNet classification dataset.

YOLO algorithm is very efficient and accurate at the same time. The difference between RetinaNet (or SSD) and YOLO is the complexity of the architecture. RetinaNet has a complex architecture. Since we had to detect small objects with very less data, we decided to go forward with RetinaNet.

It was noticed that these papers did not use the technique of splitting the dataset into easy and difficult sets. The details of the technique are discussed in the subsequent chapters.

Chapter 3

System Requirements

3.1 Libraries

3.1.1 Model

This project was coded in Python 3.6.8 which is the eighth and last maintenance release of Python 3.6. The Python 3.6 series contains many new features and optimizations.

- We used fastai 1.0.53[6] which is a wrapper of the open source deep learning platform PyTorch version 1[16]. PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. The fastai library was used to implement the Focal Loss for Dense Object Detection paper[21] and train the model. The fastai library simplifies training fast and accurate neural nets using modern best practices. It is based on research in best practices of deep learning undertaken at fast.ai, including "out of the box" support for vision, text, tabular, and collaborative filtering models. The fastai library structures its training process around the Learner class, whose object binds together a PyTorch model, a dataset, an optimizer, and a loss function; the entire learner object then will allow us to launch training.
- The fastai library internally uses PyTorch[16], matplotlib, torchvision, pandas and other computer vision libraries. Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision. Pandas is a high-level data manipulation tool which is built on the Numpy package and its key data structure is called the DataFrame. DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.

- OpenCV 3.4.3 was used to test the model on a video. It was mainly used to extract frames from a video, run the model on every frame and reconstruct the frames back into a video. OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, the library is cross-platform and free for use under the open-source BSD license. OpenCV supports the deep learning frameworks TensorFlow, Torch/PyTorch[16] and Caffe.

3.1.2 Dataset

- A web scrapper was built using Beautiful Soup library [4] to extract images from the Internet Movie Firearms Database (IMFDB) [5] which is an online database of firearms used or featured in films, television shows, video games, and anime. On the other hand, Beautiful Soup is a Python library for pulling data out of HTML and XML files. It makes it easy to scrape information from web pages. It sits atop an HTML (included in Python's standard library) or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree.
- LabelImg [22] was used to annotate the images in Pascal VOC format which resulted in XML files. LabelImg is a graphical image annotation tool. It is written in Python and uses Qt for its graphical interface. Annotations are saved as XML files in PASCAL VOC format, the format used by ImageNet[10].

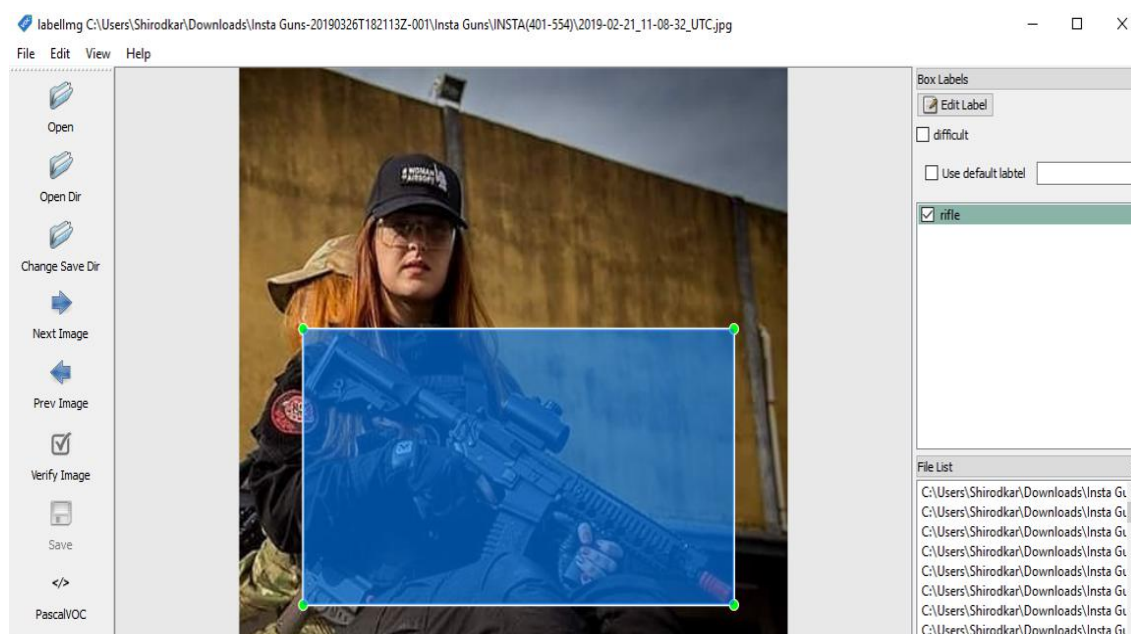


Figure 3.1: LabelImg Annotation Tool

- JSON was used to convert the XML files into a single JSON file. JSON supports only text and number data type. XML support many data types such as text, number, images, charts, graphs etc.

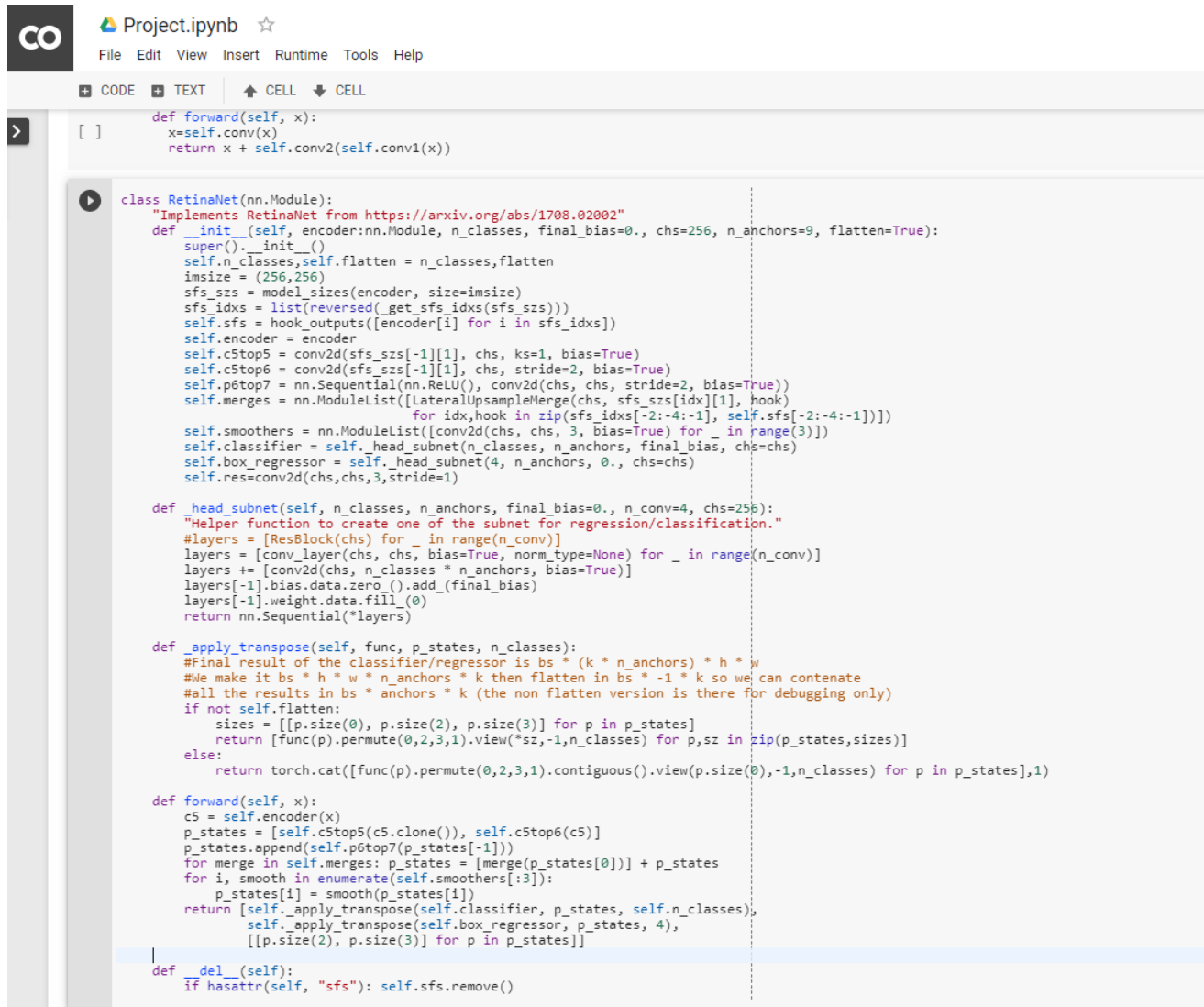
3.2 GPU

The architecture used in RetinaNet[21] requires high compute power to perform matrix multiplications in parallel. This requires the use of NVIDIA Graphic Processing Units (GPUs) with support for cuda. We used a single NVIDIA Tesla T4 of 12 GB provided by Google Colaboratory.

The entire project was implemented in Google Colaboratory which provides a Jupyter notebook like interface. This can be used extensively to experiment with the code and the fastai library in general. Google Colaboratory has extended support to the fastai library hence dependencies need not be installed every time code is run.

The dataset was uploaded on Kaggle and accessed on the Google Colab notebooks by installing the Kaggle API and authenticating the user. A Google Colab session lasts for 9 hours hence after the completion of 9 hours, the runtime would get disconnected and the dataset had to be uploaded again.

The following screenshot shows the interface of Google Colaboratory.



```

def forward(self, x):
    x=self.conv(x)
    return x + self.conv2(self.conv1(x))

class RetinaNet(nn.Module):
    """Implements RetinaNet from https://arxiv.org/abs/1708.02002"""
    def __init__(self, encoder:nn.Module, n_classes, final_bias=0., chs=256, n_anchors=9, flatten=True):
        super().__init__()
        self.n_classes,self.flatten = n_classes,flatten
        imsize = (256,256)
        sfs_szs = model_sizes(encoder, size=imsize)
        sfs_idx = list(reversed(_get_sfs_idx(sfs_szs)))
        self.sfs = hook_outputs([encoder[i] for i in sfs_idx])
        self.encoder = encoder
        self.c5top5 = conv2d(sfs_szs[-1][1], chs, ks=1, bias=True)
        self.c5top6 = conv2d(sfs_szs[-1][1], chs, stride=2, bias=True)
        self.p6top7 = nn.Sequential(nn.ReLU(), conv2d(chs, chs, stride=2, bias=True))
        self.merges = nn.ModuleList([LateralUpsampleMerge(chs, sfs_szs[idx][1], hook)
                                     for idx,hook in zip(sfs_idx[-2:-4:-1], self.sfs[-2:-4:-1])])
        self.smoothers = nn.ModuleList([conv2d(chs, chs, 3, bias=True) for _ in range(3)])
        self.classifier = self._head_subnet(n_classes, n_anchors, final_bias, chs=chs)
        self.box_regressor = self._head_subnet(4, n_anchors, 0., chs=chs)
        self.res=conv2d(chs,chs,3,stride=1)

    def _head_subnet(self, n_classes, n_anchors, final_bias=0., n_conv=4, chs=256):
        """Helper function to create one of the subnet for regression/classification."""
        #layers = [ResBlock(chs) for _ in range(n_conv)]
        layers = [conv_layer(chs, chs, bias=True, norm_type=None) for _ in range(n_conv)]
        layers += [conv2d(chs, n_classes * n_anchors, bias=True)]
        layers[-1].bias.data.zero_().add_(final_bias)
        layers[-1].weight.data.fill_(0)
        return nn.Sequential(*layers)

    def _apply_transpose(self, func, p_states, n_classes):
        #Final result of the classifier/regressor is bs * (k * n_anchors) * h * w
        #We make it bs * h * w * n_anchors * k then flatten in bs * -1 * k so we can concatenate
        #all the results in bs * anchors * k (the non flatten version is there for debugging only)
        if not self.flatten:
            sizes = [[p.size(0), p.size(2), p.size(3)] for p in p_states]
            return [func(p).permute(0,2,3,1).view(*sz,-1,n_classes) for p,sz in zip(p_states,sizes)]
        else:
            return torch.cat([func(p).permute(0,2,3,1).contiguous().view(p.size(0),-1,n_classes) for p in p_states],1)

    def forward(self, x):
        c5 = self.encoder(x)
        p_states = [self.c5top5(c5.clone()), self.c5top6(c5)]
        p_states.append(self.p6top7(p_states[-1]))
        for merge in self.merges: p_states = [merge(p_states[0])] + p_states
        for i, smooth in enumerate(self.smoothers[:3]):
            p_states[i] = smooth(p_states[i])
        return [self._apply_transpose(self.classifier, p_states, self.n_classes),
                self._apply_transpose(self.box_regressor, p_states, 4),
                [[p.size(2), p.size(3)] for p in p_states]]

    def __del__(self):
        if hasattr(self, "sfs"): self.sfs.remove()

```

Figure 3.2: Google Colaboratory

Chapter 4

Dataset Description

4.1 Collection of Data

Our dataset consists of two classes of firearms, Pistols and Rifles. We used multiple sources to collect the raw data.

Internet Movie Firearms Database (IMFDB)[5] is a large repository of firearms shown in popular movies. We built a web scrapper using BeautifulSoup library in Python to download images of pistols and rifles shown in any movie. Unusable images, like the ones having unrelated content were discarded.



The screenshot shows a GitHub repository for a script named 'get_images.py'. The repository has 1 contributor and was last updated on Dec 29, 2018. The script is 13 lines long (12 sloc) and 544 bytes. The code uses BeautifulSoup and urllib to scrape images from the IMFDB website. It defines a function 'get_images(url)' that fetches the HTML content, finds all image tags, and downloads the images to a local directory. The script is as follows:

```
1 from bs4 import BeautifulSoup
2 import re
3 from urllib.request import *
4 import urllib.request
5 def get_images(url):
6     html = urlopen(url)
7     bs = BeautifulSoup(html, 'html.parser')
8     images = bs.find_all('img', {'src': re.compile('.jpg')})
9     for image in images:
10         img_name=image['src'][(x.start() for x in re.finditer(r'/', image['src']))[4]+1:re.search('.jpg', image['src']).end()]
11         im='http://www.imfdb.org'+image['src'][:7]+image['src'][13:re.search('.jpg', image['src']).end()]
12         urllib.request.urlretrieve(im, img_name)
```

Figure 4.1: IMFDB-image-scraper

Data from other websites like Pexels, gettyimages and Google Images[15] were downloaded using a tool called ImageScrapper 2.0.7[1]. while some were manually downloaded.

While browsing for more data to add to our dataset, we came across a research paper Automatic Handgun Detection with Alarm System[17] with links to a webpage having pistol images stored in compressed folders. To add to our dataset, we extracted images from the webpage.

To add to our growing dataset of rifle images, we downloaded images from Instagram using instaloader 4.2.4.

Next, we extracted data in the form of videos from YouTube and trimmed these videos to snippets of short sequences containing gun scenes. This was done using tools from VLC Media Player and another useful tool called Hesetube. Frames were then extracted from these short snippets using OpenCV. A code in Python was run to generate the frames sequentially at the rate of 40 frames per second, using OpenCV.

Around 2000 images were analyzed and finally 1045 images were selected. The images which were discarded contained frames not consisting pistols or rifles or frames containing firearms not of our interest. Entire dataset was divided into easy and difficult images. Easy images are the ones which have lateral view of pistols and rifles. Difficult set consisted of images where pistols were partly occluded or the object into consideration was small sized. This technique was used during implementation.

Initially, the easy dataset consisted of 491 images of which 20% i.e 98 images were added to the validation set and the remaining 393 images were part of the training set. Also, out of the 551 images of the difficult dataset, 20% i.e 111 images were in the validation set while 440 images were in the training set.

Table 4.1: Dataset

Type of Split	Number of images in Train	Number of images in Validation
Easy	393	98
Difficult	440	111
Combined	836	209

Here is an example from easy and difficult set.



Figure 4.2: Easy and Difficult Image

4.2 Annotations

Any object detection model requires the image as the input along with the annotation. The annotation tells the loss function the class label of the object(s) contained in the image along with the four coordinates of its location. These are the top left and the bottom right coordinates of the bounding box surrounding the object into consideration.

Using LabelImg[22], an annotation tool which annotates images in Pascal VOC format and XML notation, we made bounding box annotations for the images and frames. This tool generates one XML file for every image. An example is given below.

```
<annotation>
  <folder>Definitiva</folder>
  <filename>armas (1485)</filename>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>500</width>
    <height>281</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>pistol</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>66</xmin>
      <ymin>13</ymin>
      <xmax>446</xmax>
      <ymax>247</ymax>
    </bndbox>
  </object>
</annotation>
```

Figure 4.3: Annotation in XML

The fields of our importance are filename, size of the image, name of the object and the four bounding box coordinates. It becomes a very cumbersome task to extract these fields from every file for 1045 images. Hence to improve efficiency we converted all the xml files into a single .json file. The advantage of using .json file is that, from a single file, the above fields can be extracted into a Python dictionary data structure.

fastai has implemented a very useful function `get_annotations()` which maps useful information from .json annotation file to Python list data structure. Here is an example

```
[1] from fastai import *
    from fastai.vision import *

[3] images, lbl_box=get_annotations('combined.json')

[4] images[:5]

↳ ['rifle454.jpg',
    'rifle379.jpg',
    'rifle461.jpg',
    'armas (1333).jpg',
    '2019-02-07_10-55-46.UTC.jpg']

[5] lbl_box[:5]

↳ [[[561, 840, 709, 946], [480, 92, 713, 715]], ['rifle', 'rifle']],
    [[224, 167, 421, 617]], ['rifle']],
    [[362, 198, 469, 730], [273, 324, 321, 553], [255, 335, 281, 449]],
    ['rifle', 'rifle', 'rifle']],
    [[108, 1, 552, 992]], ['pistol']],
    [[223, 379, 502, 1063], [586, 4, 788, 131]], ['rifle', 'pistol']]]
```

Figure 4.4: get_annotations()

Every filename is mapped to the corresponding bounding box coordinates and the class labels. The bounding box coordinates along with class labels is the dependent variable for the CNN.

4.3 Data Block API

After preparing the dataset, the next most important task is to create the model data loader which can supply the data to the model as and when it wants and in proper format. The data has to be preprocessed, split into train and validation datasets. A CNN works with one batch of the data at a time. Deep Learning suffers from a drawback of using the same size of inputs, i.e all the images in a batch should be of the same size. All these tasks were performed with the very flexible Data Block API [26] provided by the fastai[6] library.

The data block API [26] lets you customize the creation of a DataBunch by isolating the underlying parts of that process in separate blocks, mainly:

- Where are the inputs and how to create them?
- How to split the data into a training and validation sets?
- How to label the inputs?
- What transforms to apply?
- How to add a test set?
- How to wrap in dataloaders and create the DataBunch?

Each of these may be addressed with a specific block designed for your unique setup. Your inputs might be in a folder, a csv file, or a dataframe. You may want to split them randomly, by certain indices or depending on the folder they are in. You can have your labels in your csv file or your dataframe, but it may come from folders or a specific function of the input. You may choose to add data augmentation or not. A test set is optional too. Finally you have to set the arguments to put the data together in a DataBunch (batch size, collate function...)

Here is an example of how we used the data block API in our implementation.

```
[ ] path=Path('guns')
    images_trn, lbl_bbox_trn=get_annotations(path/'train.json')
    images_val, lbl_bbox_val=get_annotations(path/'valid.json')
    images, lbl_bbox=images_trn+images_val, lbl_bbox_trn+lbl_bbox_val
    img2bbox = dict(zip(images, lbl_bbox))
    get_y_func = lambda o:img2bbox[o.name]

[ ] def get_data(bs, size):
    src = ObjectItemList.from_folder(path/'images')
    src = src.split_by_files(images_val)
    src = src.label_from_func(get_y_func)
    src = src.transform(get_transforms(), size=size, tfm_y=True)
    src = src.databunch(path=path, bs=bs, collate_fn=bb_pad_collate)
    return src.normalize(imagenet_stats)

▶ learn.data=get_data(8,224)
```

Figure 4.5: Data Block API

`get_data()` function uses the data block API [26]. An `ObjectItemList` consists of items in which the dependent variable consists of bounding box coordinates as well as class labels. The images are searched in a folder called 'images'. Next, the train and validation dataset is split using the files contained in `images_val` which is extracted from the `valid.json` annotation file using `get_annotations()` function. The images are then labelled using the `get_y_func` function defined.

During training, the model cannot see the valid dataset labels, hence the loss on validation dataset helps in generalizing the model (reducing overfitting). Transforms are applied on the fly when an item is grabbed. They also may change each time we ask for the same item in the case of random transforms. These transforms augment the data with various versions of the same image. This helps in improving the dataset. Data augmentation is perhaps the most important regularization technique when training a model for Computer Vision: instead of feeding the model with the same pictures every time, we do small random transformations (a bit of rotation, zoom, translation, etc...) that don't change what's inside the image (to the human eye) but do change its pixel values. Models trained with data augmentation will then generalize better. An example of applying transforms is shown below.

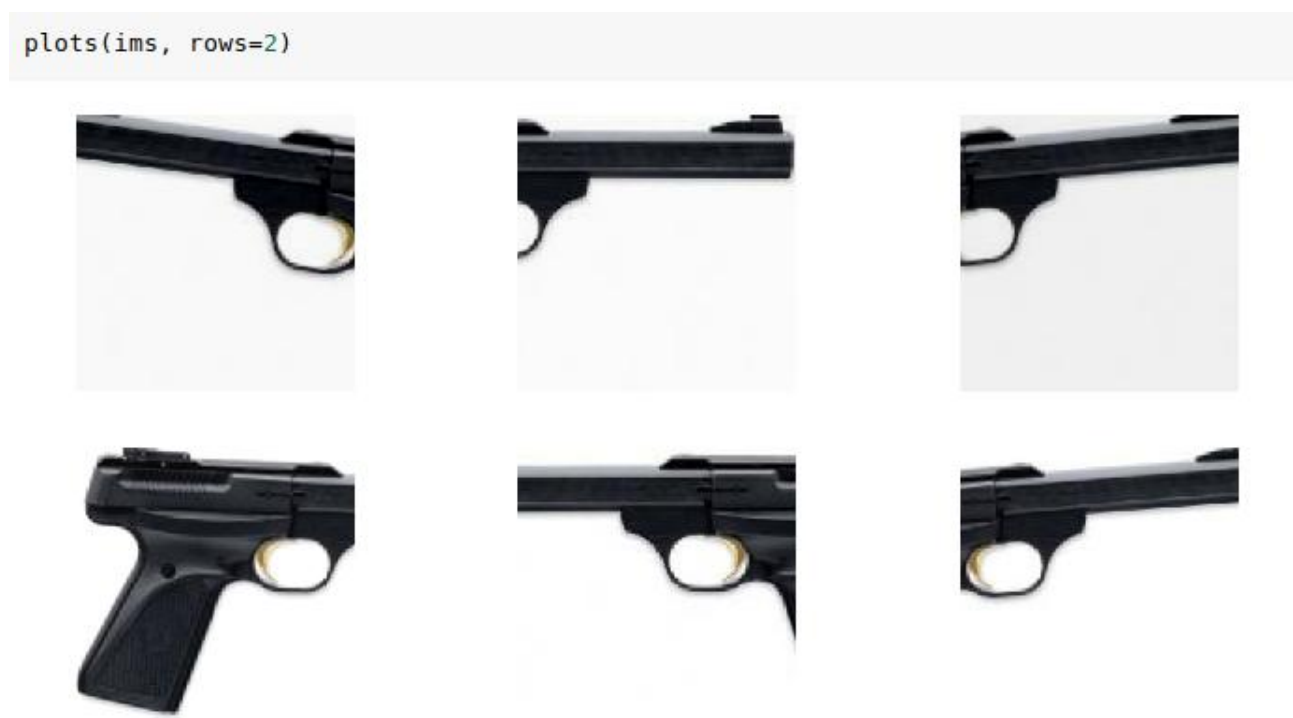


Figure 4.6: Data Augmentation

The default transformations defined in fastai [6] worked very well for our dataset. The images are also resized to the value contained in size variable provided as an argument to get_data() function. The next step is to convert the data into a databunch which contains items of training and validation sets of a particular batch size given by the bs variable which is an argument to the get_data() function. The final step is a preprocessing task of normalizing the data. The image data is normalized to achieve mean of zero and standard deviation one. The values of mean and standard deviation used to normalize the data are set to ImageNet stats. We use ImageNet stats[10] because the architecture uses ResNet50 [10] as its backbone which was trained on ImageNet [10].

Once these steps are completed successfully, our data is ready to be given to the model for training. We can look at the ground truth using the show_batch() function.

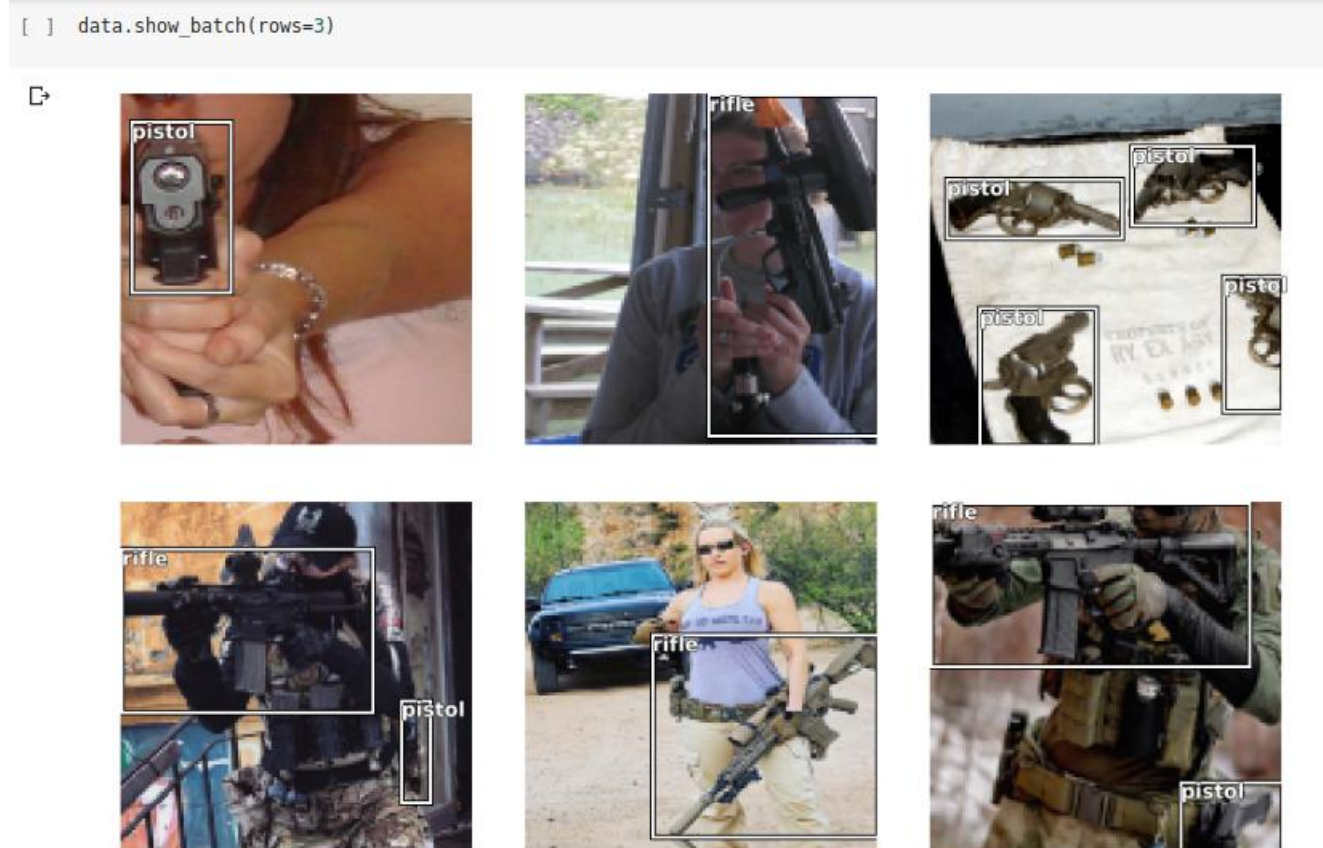


Figure 4.7: Batch

Chapter 5

Proposed Technique

5.1 Intuition

Training a neural network to detect objects is analogous to a person learning a complex course. It is easier to grasp a topic if the basics are studied first and then the complex topics. This forms the basic idea behind our technique of splitting the dataset into easy and difficult images.

This technique can be understood well by looking at how a Sentiment Analyzer was designed by Jeremy Howard in his massive open online course (MOOC) on Deep Learning[6]. The task was to analyze IMDB movie reviews as positive or negative. Instead of training a neural network from scratch, a language model was first trained. A language model is a neural network that predicts the subsequent words in a sentence. This language model basically learns English and how people write sentences in English. The sentiment analyzer was built on top of this language model using the technique of Transfer Learning. It turned out to analyze the movie reviews very efficiently as the neural network already knew the nuances of English language.

5.2 Technique

The steps involved in the new technique of training a model for Object Detection are as follows:

- Split the dataset into easy and difficult images. The main criteria used to split is to identify images which had large and single objects. The images with such objects go in the easy dataset and the images with small and multiple objects go into the difficult dataset.
- Train the model with the easy set with a validation set consisting of the random 20% of the easy set.
- Train the model using progressive re-sizing, which is a technique mentioned in subsequent chapters.
- Change the input of the model to difficult set. The weights are saved and only the input is changed.

These steps are shown in the following diagram.

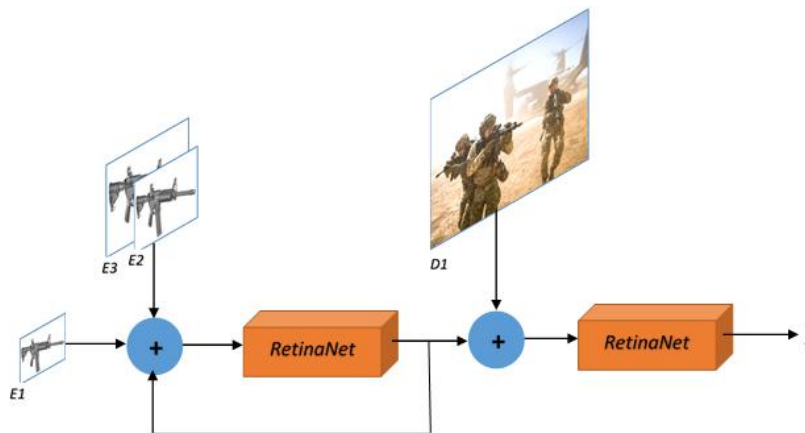


Figure 5.1: Technique

E1,E2,E3 represent 128x128,192x192 and 256x256 respectively. These are the sizes of the images given to the model when it is trained with the easy dataset. This method is called progressive re-sizing in fastai mooc, which is discussed in detail in the next chapter. The feedback loop in the diagram represents how the model was

trained for each of these sizes. D1 represents 512x512. Images of size D1 from the difficult set was given to the model which was already trained on the easy set.

To understand why this technique really works, we have to study it in more detail.

With the type of architecture (RetinaNet)[21] that we have used to train the model, it is an easy task to detect large and single objects. Detecting small and multiple objects is a challenging task. The model can detect these objects easily if it already knows how a pistol looks like or what are the different positions in which a person can hold a rifle. This is achieved by training the model with the easy set first. As the model is trained, it understands the differences between pistol and rifles, how people hold a gun and at the same time the model is getting good at predicting bounding box coordinates. Basically it learns the English of pistols and rifles.

Just when the model is getting better in detecting easy objects, it is supplied with the difficult data. We do not want the model to fit the easy data very well to reduce overfitting (or to generalize better). Since the neural network already knows most of the details about pistols and rifles, now it tries to detect objects which maybe visible partly or multiple objects maybe visible in a single frame.

The hypothesis is that with this technique, training time should be reduced and the accuracy should increase. Multiple experiments were run which proved the hypothesis. These experiments are discussed in the following chapters.

Chapter 6

Implementation

As mentioned before, any neural network requires three main components, data, architecture and the loss function. The architecture and the loss function was implemented in fastai[6] as and how it is mentioned in the Focal Loss for Dense Object Detection research paper[21] with no modifications added. The implementation was inspired from Sylvian Guggur's implementation[29] of the same paper in fastai. The RetinaNet[21] architecture is defined as follows:

```

1  class RetinaNet(nn.Module):
2      """Implements RetinaNet from https://arxiv.org/abs/1708.02002"""
3      def __init__(self, encoder:nn.Module, n_classes, final_bias=0., chs=256,
4          n_anchors=9, flatten=True):
5          super().__init__()
6          self.n_classes, self.flatten = n_classes, flatten
7          imsize = (512,512)
8          sfs_szs = model_sizes(encoder, size=imsize)
9          sfs_idxxs = list(reversed(_get_sfs_idxxs(sfs_szs)))
10         self.sfs = hook_outputs([encoder[i] for i in sfs_idxxs])
11         self.encoder = encoder
12         self.c5top5 = conv2d(sfs_szs[-1][1], chs, ks=1, bias=True)
13         self.c5top6 = conv2d(sfs_szs[-1][1], chs, stride=2, bias=True)
14         self.p6top7 = nn.Sequential(nn.ReLU(), conv2d(chs, chs, stride=2, bias=
15             True))
16         self.merges = nn.ModuleList([LateralUpsampleMerge(chs, sfs_szs[idx][1],
17             hook)
18             for idx, hook in zip(sfs_idxxs[-2:-4:-1],
19                 self.sfs[-2:-4:-1])]
20         )
21         self.smoothers = nn.ModuleList([conv2d(chs, chs, 3, bias=True) for _ in
22             range(3)])
23         self.classifier = self._head_subnet(n_classes, n_anchors, final_bias,
24             chs=chs)
25         self.box_regressor = self._head_subnet(4, n_anchors, 0., chs=chs)
26         self.res=conv2d(chs,chs,3,stride=1)
27
28     def _head_subnet(self, n_classes, n_anchors, final_bias=0., n_conv=4, chs
29         =256):
30         """Helper function to create one of the subnet for regression/
31             classification."""
32         #layers = [ResBlock(chs) for _ in range(n_conv)]

```

```

24         layers = [conv_layer(chs, chs, bias=True, norm_type=None) for _ in range
25                     (n_conv)]
26         layers += [conv2d(chs, n_classes * n_anchors, bias=True)]
27         layers[-1].bias.data.zero_().add_(final_bias)
28         layers[-1].weight.data.fill_(0)
29         return nn.Sequential(*layers)
30
31     def _apply_transpose(self, func, p_states, n_classes):
32         #Final result of the classifier/regressor is bs * (k * n_anchors) * h *
33         #w
34         #We make it bs * h * w * n_anchors * k then flatten in bs * -1 * k so we
35         #can concatenate
36         #all the results in bs * anchors * k (the non flatten version is there
37         #for debugging only)
38         if not self.flatten:
39             sizes = [[p.size(0), p.size(2), p.size(3)] for p in p_states]
40             return [func(p).permute(0,2,3,1).view(*sz,-1,n_classes) for p,sz in
41                     zip(p_states, sizes)]
42         else:
43             return torch.cat([func(p).permute(0,2,3,1).contiguous().view(p.size
44                                     (0),-1,n_classes) for p in p_states],1)
45
46     def forward(self, x):
47         c5 = self.encoder(x)
48         p_states = [self.c5top5(c5.clone()), self.c5top6(c5)]
49         p_states.append(self.p6top7(p_states[-1]))
50         for merge in self.merges: p_states = [merge(p_states[0])] + p_states
51         for i, smooth in enumerate(self.smoothers[:3]):
52             p_states[i] = smooth(p_states[i])
53         return [self._apply_transpose(self.classifier, p_states, self.n_classes)
54                 ,
55                 self._apply_transpose(self.box_regressor, p_states, 4),
56                 [[p.size(2), p.size(3)] for p in p_states]]
57
58     def __del__(self):
59         if hasattr(self, "sfs"): self.sfs.remove()

```

The Focal Loss Function is defined as follows

```

1 class RetinaNetFocalLoss(nn.Module):
2
3     def __init__(self, gamma:float=2., alpha:float=0.25, pad_idx:int=0, scales:
4         Collection[float]=None,
5         ratios:Collection[float]=None, reg_loss:LossFunction=F.
6             smooth_l1_loss):
7         super().__init__()
8         self.gamma, self.alpha, self.pad_idx, self.reg_loss = gamma, alpha, pad_idx,
9             reg_loss
10        self.scales = ifnone(scales, [1,2*(-1/3), 2*(-2/3)])
11        self.ratios = ifnone(ratios, [1/2,1,2])
12
13    def _change_anchors(self, sizes:Sizes) -> bool:
14        if not hasattr(self, 'sizes'): return True
15        for sz1, sz2 in zip(self.sizes, sizes):

```

```

13         if sz1[0] != sz2[0] or sz1[1] != sz2[1]: return True
14     return False
15
16     def _create_anchors(self, sizes: Sizes, device: torch.device):
17         self.sizes = sizes
18         self.anchors = create_anchors(sizes, self.ratios, self.scales).to(device)
19
20     def _unpad(self, bbox_tgt, clas_tgt):
21         i = torch.min(torch.nonzero(clas_tgt - self.pad_idx))
22         return tlbr2cthw(bbox_tgt[i:]), clas_tgt[i:] - 1 + self.pad_idx
23
24     def _focal_loss(self, clas_pred, clas_tgt):
25         encoded_tgt = encode_class(clas_tgt, clas_pred.size(1))
26         ps = torch.sigmoid(clas_pred.detach())
27         weights = encoded_tgt * (1 - ps) + (1 - encoded_tgt) * ps
28         alphas = (1 - encoded_tgt) * self.alpha + encoded_tgt * (1 - self.alpha)
29         weights.pow_(self.gamma).mul_(alphas)
30         clas_loss = F.binary_cross_entropy_with_logits(clas_pred, encoded_tgt,
31                                                         weights, reduction='sum')
32         return clas_loss
33
34     def _one_loss(self, clas_pred, bbox_pred, clas_tgt, bbox_tgt):
35         bbox_tgt, clas_tgt = self._unpad(bbox_tgt, clas_tgt)
36         matches = match_anchors(self.anchors, bbox_tgt)
37         bbox_mask = matches >= 0
38         if bbox_mask.sum() != 0:
39             bbox_pred = bbox_pred[bbox_mask]
40             bbox_tgt = bbox_tgt[matches[bbox_mask]]
41             bb_loss = self.reg_loss(bbox_pred, bbox_tgt - self.anchors[bbox_mask])
42         else: bb_loss = 0.
43         matches.add_(1)
44         clas_tgt = clas_tgt + 1
45         clas_mask = matches >= 0
46         clas_pred = clas_pred[clas_mask]
47         clas_tgt = torch.cat([clas_tgt.new_zeros(1).long(), clas_tgt])
48         clas_tgt = clas_tgt[matches[clas_mask]]
49         return bb_loss + self._focal_loss(clas_pred, clas_tgt) / torch.clamp(
50             bbox_mask.sum(), min=1.)
51
52     def forward(self, output, bbox_tgts, clas_tgts):
53         clas_preds, bbox_preds, sizes = output
54         if self._change_anchors(sizes): self._create_anchors(sizes, clas_preds.device)
55         n_classes = clas_preds.size(2)
56         return sum([self._one_loss(cp, bp, ct, bt)
57                     for (cp, bp, ct, bt) in zip(clas_preds, bbox_preds,
58                                                  clas_tgts, bbox_tgts)]) / clas_tgts.size(0)

```

The source code for our project can be found at <https://github.com/kakods/CNN-based-Gun-detection>.

In this chapter we discuss how the model was trained and how our technique was incorporated in the process. This is illustrated by the following techniques:

- Learning Rate Finder
- 1cycle policy
- Progressive re-sizing and Fine Tuning
- Regularization
- Splitting the dataset

6.1 Learning Rate Finder

Hyperparameters are the variables that have to be tuned before applying any learning algorithm to a dataset. Learning rate is a very important hyperparameter which determines the rate at which the weights will be updated in a CNN to attain a minimum in the Gradient Descent algorithm[6]. A very high learning rate leads to weights overshooting which results in an inaccurate model. On the other hand, a very low learning rate leads to very slow training time. It is very much important to use the optimum learning rate. fastai[6] implements a method called `lr_find()` which uses the algorithm developed by Leslie Smith in the research paper 'Cyclical learning rates for training neural networks[12]. Learning rate finder plots learning rate vs loss relationship for a Learner. The idea is to reduce the amount of guesswork on picking a good starting learning rate. Overview:

- First run `lr_find`
- Plot the learning rate vs loss `learn.recorder.plot()`
- Pick a learning rate before it diverges then start training

Consider the following image to be the output of the learning rate finder.

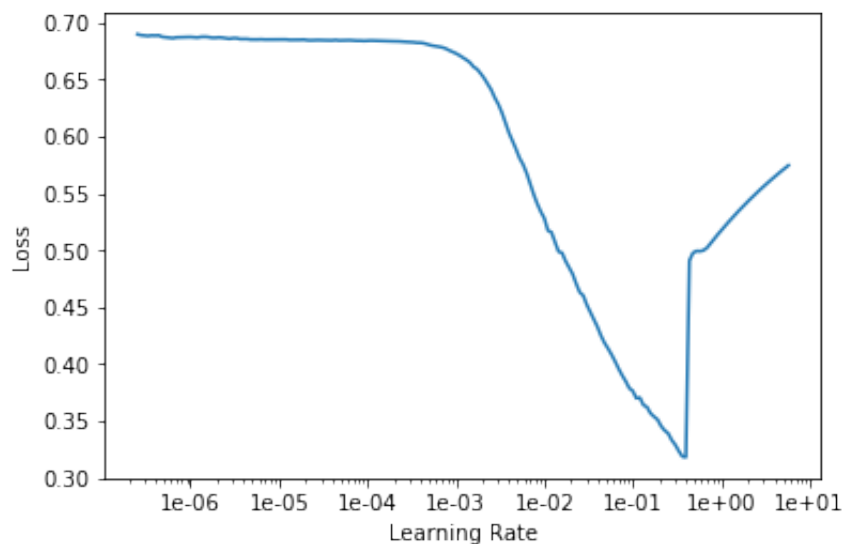


Figure 6.1: Loss vs Learning Rate plot

Then, we choose a value that is approximately in the middle of the sharpest downward slope. This is given as an indication by the LR Finder tool, an optimal learning rate for this plot would be $1e-2$. Picking a value before the downward slope results in slow training. Picking the minimum value overshoots the weights and reduces accuracy.

6.2 1cycle Policy

1cycle policy[12] is another algorithm developed by Leslie Smith in his paper "A disciplined approach to neural network hyper-parameters: Part 1 learning rate, batch size, momentum, and weight decay" [13]. This algorithm helps in training with higher learning rates for a longer time. To use 1cycle policy in fastai we need an optimum learning rate. We can find this learning rate by using a learning rate finder. The 1cycle policy has three steps:

1. We progressively increase our learning rate from `lr_max/div_factor` to `lr_max` and at the same time we progressively decrease our momentum from `mom_max` to `mom_min`.
2. We do the exact opposite: we progressively decrease our learning rate from `lr_max` to `lr_max/div_factor` and at the same time we progressively increase our momentum from `mom_min` to `mom_max`.
3. We further decrease our learning rate from `lr_max/div_factor` to `lr_max/(div_factor x 100)` and we keep momentum steady at `mom_max`.

Momentum[25] is another hyperparameter which is intended to help speed the optimization process.

Here is the schedule of the lrs (left) and momentum (right) that the 1cycle policy uses.

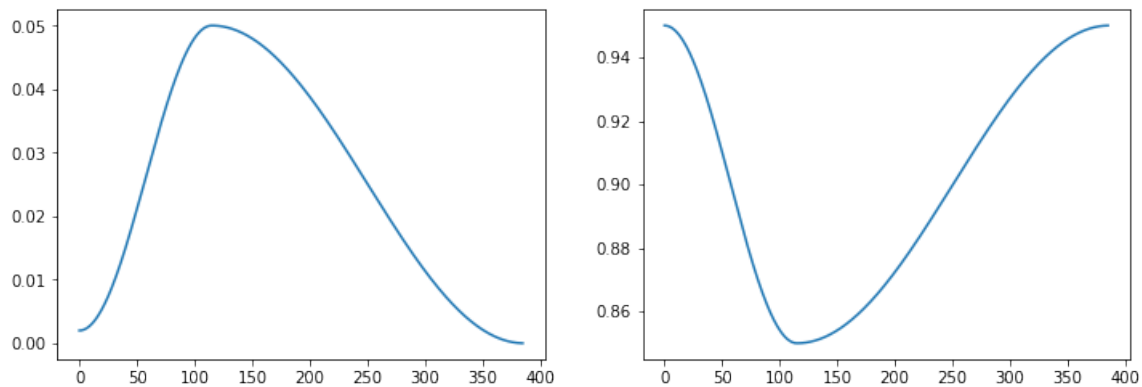


Figure 6.2: 1cycle Scheduler

We can get a far better accuracy and a far lower loss in the same number of epochs by using 1cycle policy.

6.3 Progressive Re-sizing and Fine-Tuning

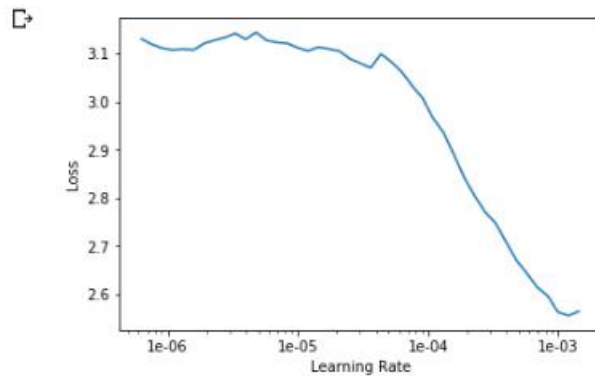
Progressive re-sizing is a method introduced by Jeremy Howard in the fastai MOOC[6]. The basic concept is that, it is easier to train a model when the size of the images is small, also the process of training is faster.

This is achieved by implementing the following steps.

- Start with image size as 128x128.
- Find optimal learning rate and use `fit_one_cycle` method to train using 1cycle policy.
- After going through some epoch, unfreeze the model and train it further. This process is called Fine Tuning. Initially when the model is trained, the weights of ResNet50, which is the backbone of our architecture, are not changed; only the recent layers added to the CNN are trained. Unfreezing the model trains the initial layers, but they are trained at a lower learning rate.
- The above steps are repeated for image size 192x192, 256x256 and 512x512.

The following snippets from the project source code shows how fine tuning is implemented.

```
[ ] learn.lr_find()
    learn.recorder.plot()
```



```
learn.fit_one_cycle(5,max_lr=(3e-4))
```

epoch	train_loss	valid_loss	time
0	3.179905	5.580485	00:13
1	2.931734	1.832770	00:13
2	2.701104	2.471909	00:14
3	2.674396	2.176059	00:13
4	2.610040	2.112298	00:13

Figure 6.3: Training before unfreezing for size 128x128

```
[ ] learn.unfreeze()
```

```
[ ] learn.fit_one_cycle(5,max_lr=slice(1e-6,1e-5))
```

epoch	train_loss	valid_loss	time
0	1.467258	1.844544	00:15
1	1.486819	1.723604	00:14
2	1.490281	1.632644	00:13
3	1.472066	1.626378	00:13
4	1.451755	1.628380	00:13

Figure 6.4: Training after unfreezing for size 128x128

As can be seen from the train_loss and valid_loss values, fine tuning helps in generalizing the model. Progressive re-sizing and fine tuning have been one of the most important methods used to train our object detection model.

6.4 Regularization

Overfitting is the phenomenon which occurs when the model performs very well on the training dataset and very badly on the test(validation) dataset. The entire purpose of using optimal learning rate, 1cycle policy[12], fine tuning, changing of hyperparameters and most of the techniques mentioned above is to stop the model from overfitting. To counter overfitting, researchers have come up with Weight decay [27], Batch Normalization [28], data augmentation [26] and many other methods.

fastai uses 0.01 as the default value for weight decay, a Batch Normalization layer by default after the convolution layer and the various transforms to augment data in the data block API [26]. The fastai documentation[6] provides a list of defaults used in `get_transforms()` method.

```
get_transforms [test] [source]

get_transforms ( do_flip : bool = True , flip_vert : bool = False ,
max_rotate : float = 10.0 , max_zoom : float = 1.1 , max_lighting : float = 0.2 ,
max_warp : float = 0.2 , p_affine : float = 0.75 , p_lighting : float = 0.75 ,
xtra_tfms : Optional [ Collection [ Transform ] ] = None ) → Collection [ Transform ]
```

Utility func to easily create a list of flip, rotate, `zoom`, warp, lighting transforms.

- `do_flip`: if True, a random flip is applied with probability 0.5
- `flip_vert`: requires `do_flip=True`. If True, the image can be flipped vertically or rotated by 90 degrees, otherwise only an horizontal flip is applied
- `max_rotate`: if not None, a random rotation between `-max_rotate` and `max_rotate` degrees is applied with probability `p_affine`
- `max_zoom`: if not 1. or less, a random zoom between 1. and `max_zoom` is applied with probability `p_affine`
- `max_lighting`: if not None, a random lightning and contrast change controlled by `max_lighting` is applied with probability `p_lighting`
- `max_warp`: if not None, a random symmetric warp of magnitude between `-max_warp` and `maw_warp` is applied with probability `p_affine`
- `p_affine`: the probability that each affine transform and symmetric warp is applied
- `p_lighting`: the probability that each lighting transform is applied
- `xtra_tfms`: a list of additional transforms you would like to be applied

This function returns a tuple of two lists of transforms, one for the training set and the other for the validation set (which is limited to a center crop by default).

Figure 6.5: Default transforms used
(Source: <https://docs.fast.ai/vision.transform.html>)

These defaults provided by the fastai[6] library worked very well for our model. During training these values were not changed and it helped a great deal to counter overfitting.

6.5 Splitting the dataset

fastai has a very flexible training API for computer vision. This helped us to implement our technique to train the model while using all the above techniques simultaneously. Overview can be give in following steps.

- Create model data from the easy dataset and set validation set to random 20%.
- Train the model over easy dataset. Start with size 128x128
 - Obtain the optimal learning rate, train for few epochs.
 - Unfreeze the model
 - Call learning rate finder
 - Use a lower learning rate for initial layers and for the recent layers use the (initial learning rate)/10
 - stop training when the training loss keep going down but validation loss starts increasing
 - Repeat the above steps for size 192x192 and 256x256
- Create model data from the difficult dataset and set validation set to the one defined in the dataset.
- Train the model according to the above steps for size 512x512

Chapter 7

Experiments

7.1 AP

To test our technique of training, we had to implement a metric to test the results. In object detection, evaluation is non trivial, because there are two distinct tasks to measure:

- Determining whether an object exists in the image (classification).
- Determining the location of the object (localization, a regression task).

So a simple accuracy-based metric will introduce biases. It is also important to assess the risk of misclassifications. Thus, there is the need to associate a confidence score or model score with each bounding box detected and to assess the model at various level of confidence. In order to address these needs, the Average Precision (AP) was introduced.

To understand the AP, it is necessary to understand the precision and recall of a classifier. precision measures the false positive rate or the ratio of true object detections to the total number of objects that the classifier predicted. If you have a precision score of close to 1.0 then there is a high likelihood that whatever the classifier predicts as a positive detection is in fact a correct prediction. Recall measures the false negative rate or the ratio of true object detections to the total number of objects in the data set. If you have a recall score close to 1.0 then almost all objects that are in your dataset will be positively detected by the model.

Finally, it is very important to note that the there is an inverse relationship between precision and recall and that these metrics are dependent on the model score

threshold that you set (as well as of course, the quality of the model). To calculate the AP, for a specific class the precision-recall curve is computed from the model's detection output, by varying the model score threshold that determines what is counted as a model-predicted positive detection of the class. In order to evaluate the model on the task of object localization, we must first determine how well the model predicted the location of the object. For all of these cases, the localization task is typically evaluated on the Intersection over Union threshold (IoU). The basic idea is that it summarizes how well the ground truth object overlaps the object boundary predicted by the model. The mean Average Precision or mAP score is calculated by taking the mean AP over all classes and/or over all IoU thresholds. For our model we used IoU threshold of 0.35. The experiment results show only the AP score.

7.2 Results

We recorded the time and AP score after training the model in one go and then using our technique of splitting the dataset. The experimental results mirrored the hypothesis. All these experiments were conducted on NVIDIA Tesla T4 GPU in Google Colab. The following table provides the summary of our findings.

Table 7.1: Results

Method	Training Time(s)				Time(s)	AP	
	128x128	192x912	256x256	512x512		pistol	rifle
SD	244	412	252	1233	1921	0.519	0.063
CD	548	872	599	2307	4326	0.028	0.053

SD denotes the technique of training by splitting the dataset and CD denotes the technique of training the entire dataset in one go. As can be seen from the above table, training by splitting the dataset takes considerably less time and the improvement in AP of each class is significant.

Chapter 8

Results

8.1 Validation Set

Our object detection model predicts the class label, the bounding box coordinates. To display the results, we have also included the confidence score predicted by the model which tells how confident the model is to predict that particular class label.

In this section we present the results of our model on the validation set and compare them with results achieved by training the entire dataset in one go. The confidence threshold was set to 0.35. The output image displays the class label as well as confidence score.

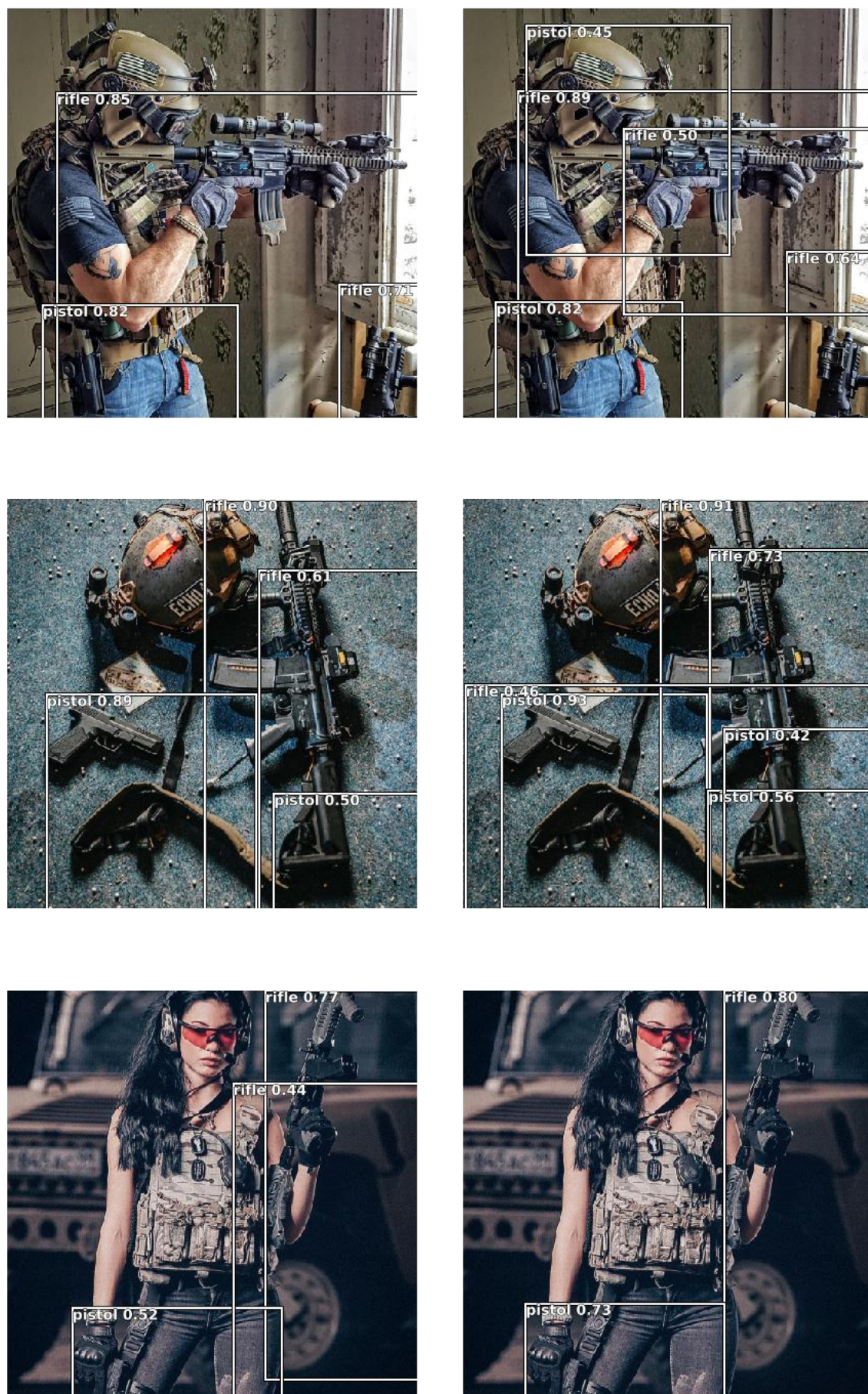


Figure 8.1: Combined training vs Our technique of Splitting

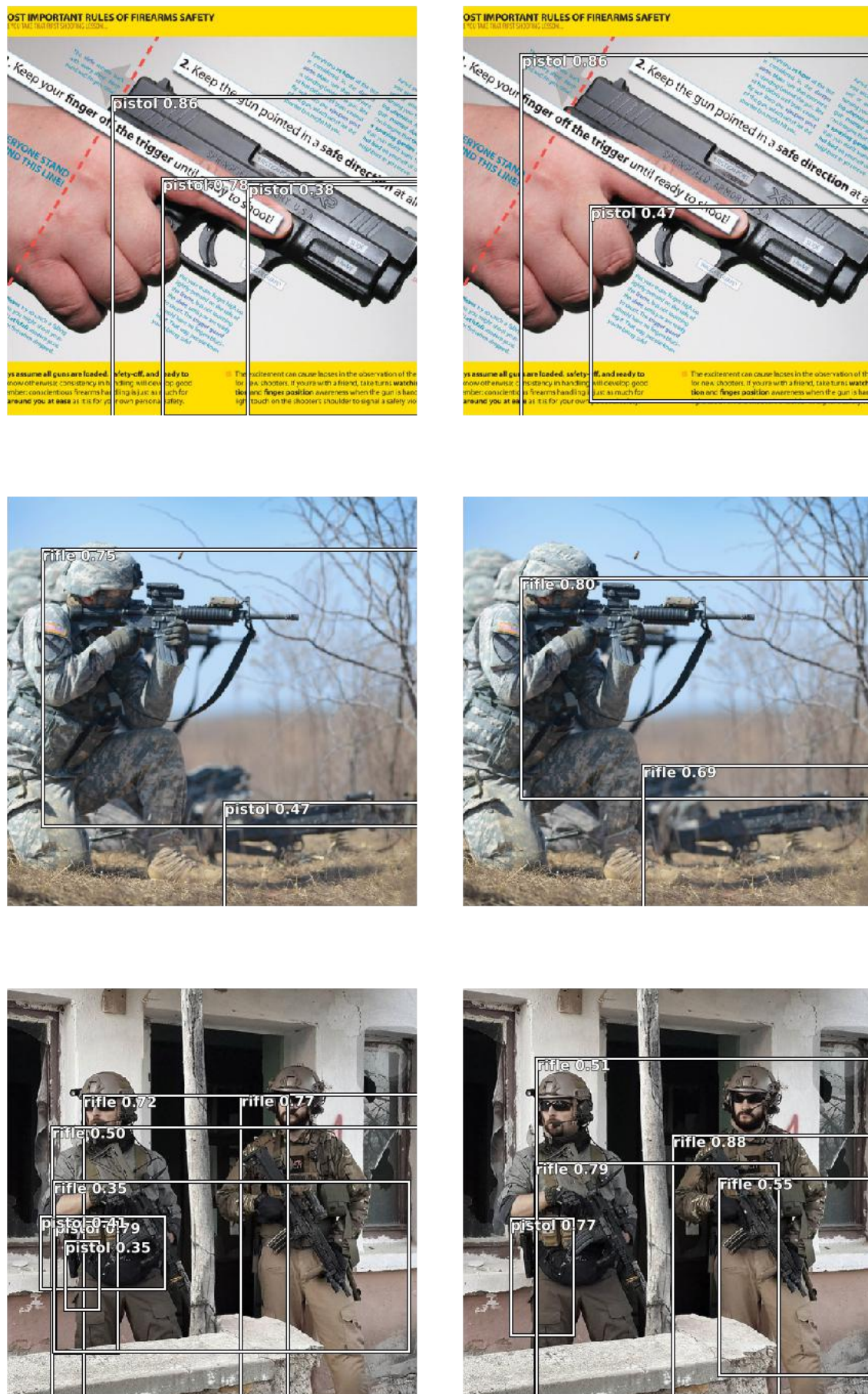


Figure 8.2: Combined training vs Our technique of Splitting

8.2 Videos

To carry out detection on videos we followed a simple algorithm.

- Extract the frames from the video using OpenCV library at the rate 45 frames per second.
- Run the object detection model on every frame to predict class label and bounding box coordinates.
- Print the class label and the confidence score on the frame and draw the bounding box using OpenCV library.
- Generate a new video using this new set of frames.

The video used for testing can be found at <https://www.youtube.com/watch?v=j9xS0AxwxK0>. In this section we present some screenshots from the video output. Confidence threshold of 0.5 was used to generate the following output.



Figure 8.3: Screenshot 1



Figure 8.4: Screenshot 2



Figure 8.5: Screenshot 3



Figure 8.6: Screenshot 4

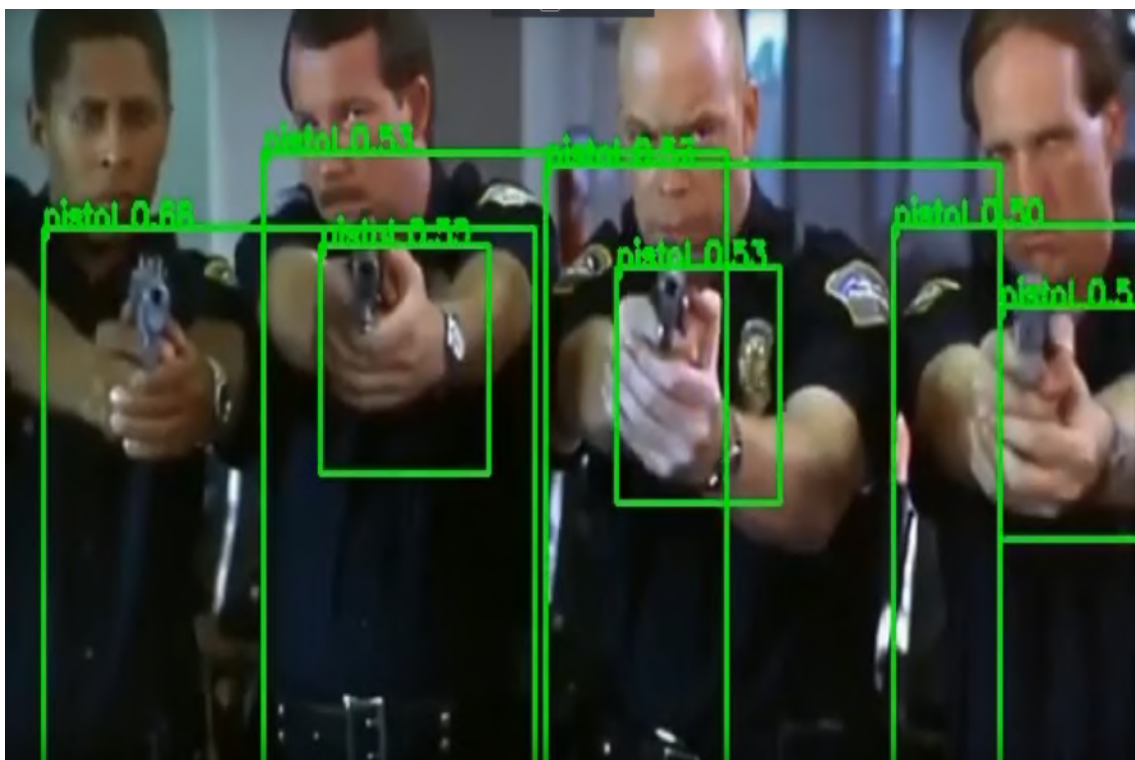


Figure 8.7: Screenshot 5



Figure 8.8: Screenshot 6



Figure 8.9: Screenshot 7

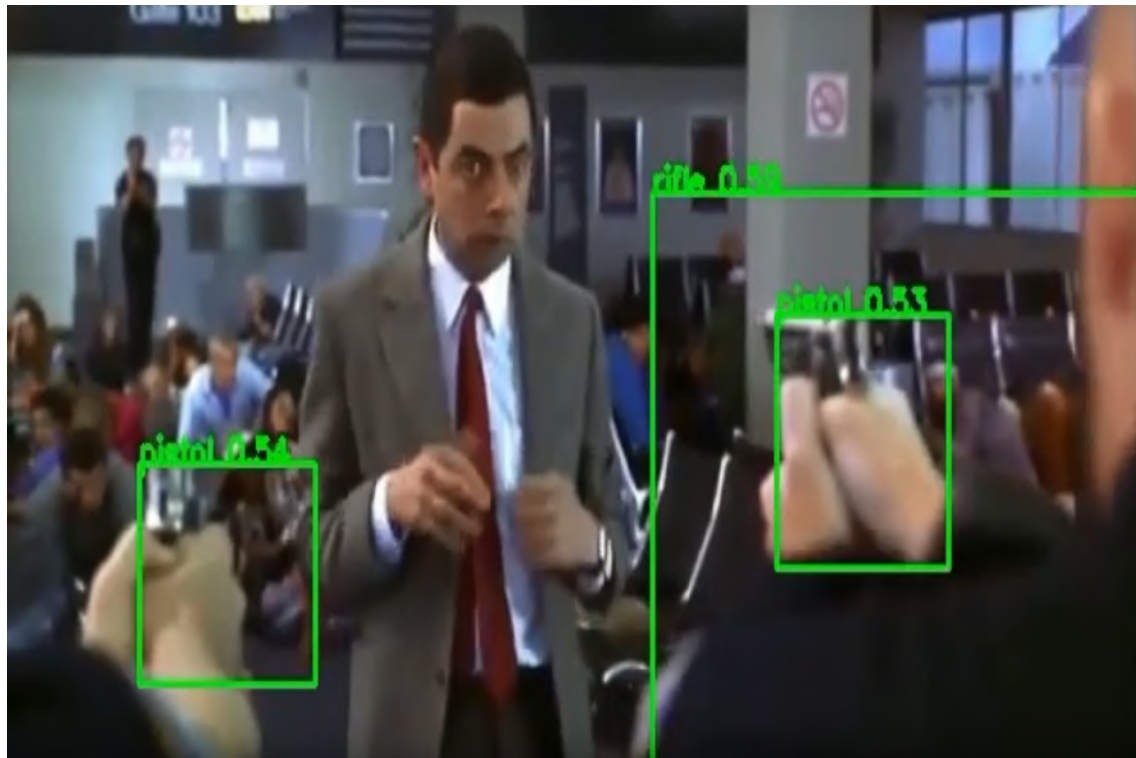


Figure 8.10: Screenshot 8

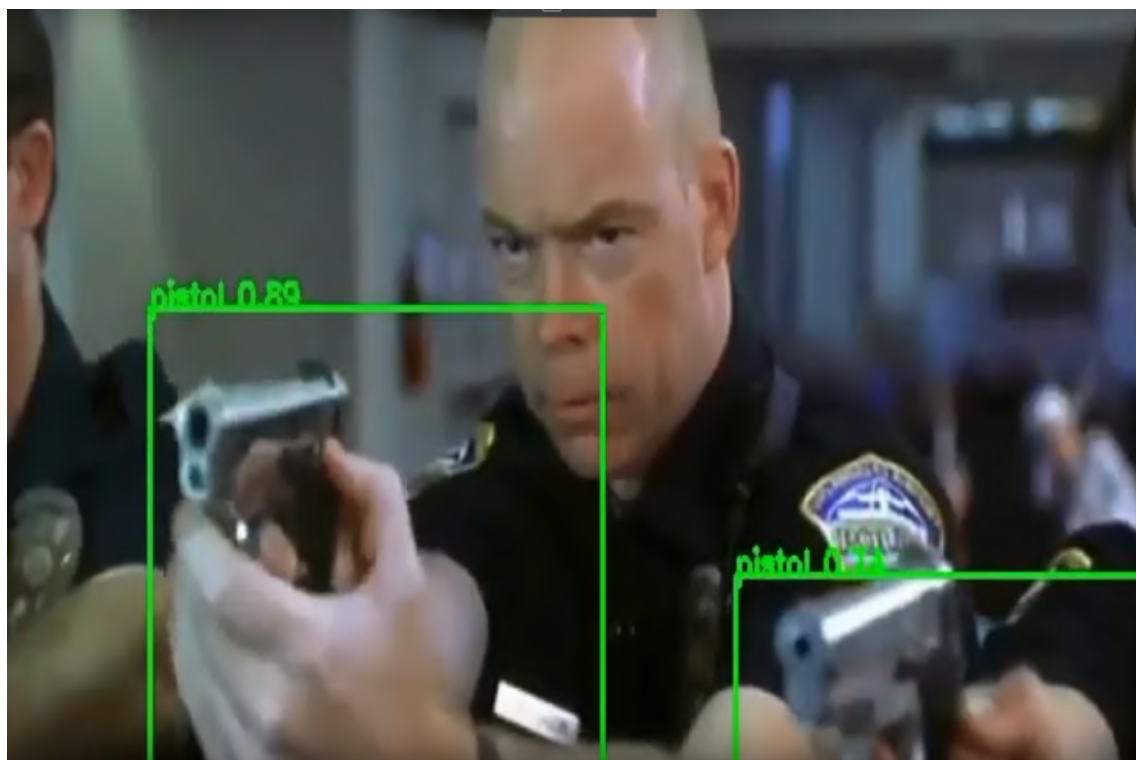


Figure 8.11: Screenshot 9



Figure 8.12: Screenshot 10



Figure 8.13: Screenshot 11

Chapter 9

Conclusion

To build a Real Time Gun Detection Classifier, a custom dataset of 1045 images was created and annotated from scratch. The dataset consisted of two classes, pistols and rifles. Various Learning Algorithms were analyzed and finally RetinaNet[21] was implemented in the fastai[6] library. We developed a technique of training Object Detection models by splitting the dataset. Techniques such as LR Finder and 1cycle policy were used to train the model efficiently. Regularization techniques such as Weight Decay [27], Batch Normalization [28], Progressive Re-sizing and Data Augmentation [26] were used to reduce overfitting. Experiments were conducted to compare the training time and accuracy of combined dataset and split dataset.

The results from our experiments showed that splitting the dataset into easy and difficult images improves AP score and takes less time to train. It was also observed that normalizing the data and applying the transforms provided by fastai[6] library helps in training a better model.

We suggest using Progressive Re-sizing and the regularization techniques of weight decay and Batch Normalization to train any computer vision model. We conclude that using the dataset splitting technique, the AP score for the pistol class was improved by 0.491099 and the AP score for rifle class was improved by 0.010574 and a training improvement of 2045 seconds.

Chapter 10

Future Scope

The scope for this project can be divided into the following sections.

10.1 Dataset

The Dataset can be updated with more classes of guns. Also different types of pistols and rifles can be added. This will give rise to fine-grained object detection. More images will have to be added and annotated.

10.2 Splitting of the Dataset

Our implementation splits the dataset manually. Human supervision is needed to split the dataset into easy and difficult images. This task can be performed automatically using neural networks. Classification task on the images will give rise to probability scores of predictions. Image with low score goes in the difficult set and the remaining in the easy set. This is just a proposal but more research and experiments should be carried out in this field to split the dataset automatically.

10.3 AP scores

The results from our experiments show that AP increases by a significant amount for rifle class but not at the same rate for pistol class, when training is performed using our technique. The exact reason why this occurs is still unknown. It may be because of class imbalance but extensive investigation should be carried out with respect to this observation.

10.4 Application

Due to time constraints we could not get a full fledged application for our project. An application should be developed in the form of Google Notebooks where if video link is provided, the application generates the output video.

References

- [1] A. Natarajan S, ImageScraper 2.0.7. PyPI, 2015. Available: <https://pypi.org/project/ImageScraper/>
- [2] Alexander Graf, Andr Koch-Kramer, instaloader 4.2.4, PyPI, 2019. Available: <https://pypi.org/project/instaloader/>
- [3] Arthur Ouaknine, Review of Deep Learning Algorithms for Object Detection, 2018.
- [4] "Beautiful Soup Documentation Beautiful Soup 4.4.0 documentation", Crummy.com, 2019. [Online].
- [5] "Internet Movie Firearms Database - Guns in Movies, TV and Video Games", Imfdb.org. [Online].
- [6] J. Howard, "Deep Learning For Coders36 hours of lessons for free", Course.fast.ai, 2018.
- [7] J. Lai and S. Maples, "Developing a Real-Time Gun Detection Classifier", Stanford University, 2017.
- [8] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger", 2016.
- [9] K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", Visual Geometry Group, Department of Engineering Science, University of Oxford, 2015.
- [10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep residual learning for image recognition", In CVPR, 2016.
- [11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", Cornell University, 2016.

- [12] Leslie N Smith. "Cyclical learning rates for training neural networks". In Applications of Computer Vision (WACV), 2017. IEEE Winter Conference on, pp. 464472. IEEE, 2017.
- [13] Leslie N Smith, "A disciplined approach to neural network hyperparameters: Part 1 learning rate, batch size, momentum, and weight decay", US Naval Research Laboratory Technical Report, 2018. Available: <https://arxiv.org/pdf/1803.09820.pdf>
- [14] "Object Detection using Deep Learning Approaches: An End to End Theoretical Perspective", mc.ai, 2018.
- [15] "Open Images Dataset V4", Storage.googleapis.com. [Online].
- [16] "PyTorch", GitHub. [Online]. Available: <https://github.com/pytorch/pytorch>.
- [17] Roberto Olmos, Siham Tabik and Francisco Herrera, "Automatic Handgun Detection Alarm in Videos Using Deep Learning", Soft Computing and Intelligent Information Systems research group, Department of Computer Science and Artificial Intelligence, University of Granada, 2017. [Dataset] Available: <https://sci2s.ugr.es/weapons-detection>.
- [18] Ryan Daws, Fastai is a Python library aiming to make AI simpler, 2018.
- [19] T. Arlen, "Understanding the mAP Evaluation Metric for Object Detection - Medium", Medium, 2018. [Online]. Available: <https://medium.com/@timothycarlen/understanding-the-map-evaluation-metric-for-object-detection-a07fe6962cf3>. [Accessed: 25- Jun- 2019]
- [20] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan and S. Belongie, "Feature pyramid networks for object detection", In CVPR, 2017.
- [21] T.-Y.Lin, P.Goyal, R.Girshick, K. He, and P. Dollar. "Focal loss for dense object detection", 2017.
- [22] Tzotalin. LabelImg. Git code, 2015. [Online]
- [23] V. Angelo, "Toolkit to download and visualize single or multiple classes from the huge Open Images v4 dataset", 2018.
- [24] W. Liu et al., "SSD: Single Shot MultiBox Detector", Cornell University, 2016.

- [25] Wiki.fast.ai, 'Lesson 4 Notes', 2016. [Online]. Available: http://wiki.fast.ai/index.php/Lesson_4_Notes#Momentum. [Accessed: 25-July- 2019].
- [26] "fastai", Docs.fast.ai, 2019. [Online]. Available: <https://docs.fast.ai/>. [Accessed: 26- Jun- 2019]
- [27] Ilya Loshchilov, Frank Hutter, "Decoupled Weight Decay Regularization", ICLR, 2019.
- [28] Sergey Ioffe, Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML, 2015.
- [29] S. Gugger, "sgugger/Deep-Learning", GitHub, 2019. [Online]. Available: <https://github.com/sgugger/Deep-Learning/blob/master/Retina>