INTELLIGENT COLLISION PREVENTION SYSTEM FOR SPECT DETECTORS BY IMPLEMENTING DEEP LEARNING BASED REAL-TIME OBJECT DETECTION

by

Tahrir Ibraq Siddiqui

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Engineering

Department of Electrical and Computer Engineering

Fort Wayne, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Guoping Wang, Chair
Department of Electrical and Computer Engineering

Dr. Bin Chen
Department of Electrical and Computer Engineering

Dr. Chao Chen
Department of Electrical and Computer Engineering

Approved by:
Dr. Chao Chen

Dedicated to my parents, who are always there for me.

ACKNOWLEDGMENTS

I want to thank Thomas Cullen, manager of Embedded Systems team at Siemens Healthineers, for proposing this project as a potential thesis topic, and Dennis Steibel, manager of Hardware team at Siemens Healthineers, for providing helpful guidance along the way. Acknowledgements also go out to my adviser, Dr. Guoping Wang, for his support throughout this research journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SPECT	Single-Photon Emission Computerized tomography
CT	Computed Tomography
RTOD	Real-Time Object Detection
CNN	Convolutional Neural Network
R-CNN	Region-Convolutional Neural Network
SSD	Single-shot Detection
YOLO	You Only Look Once
VOC	Visual Object Classes
COCO	Common Objects In Context
LMDB	Lightning Memory-Mapped Database

ABSTRACT

The SPECT-CT machines manufactured by Siemens consists of two heavy detector heads(~1500lbs each) that are moved into various configurations for radionuclide imaging. These detectors are driven by large torque powered by motors in the gantry that enable linear and rotational motion. If the detectors collide with large objects – stools, tables, patient extremities, etc. – they are very likely to damage the objects and get damaged as well. This research work proposes an intelligent real-time object detection system to prevent collisions between detector heads and external objects in the path of the detector's motion by implementing an end-to-end deep learning object detector. The research extensively documents all the work done in identifying the most suitable object detection framework for this use case, collecting, and processing the image dataset of target objects, training the deep neural net to detect target objects, deploying the trained deep neural net in live demos by implementing a real-time object detection application written in Python, improving the model's performance, and finally investigating methods to stop detector motion upon detecting external objects in the collision region. We successfully demonstrated that a *Caffe* version of *MobileNet-SSD* can be trained and deployed to detect target objects entering the collision region in real-time by following the methodologies outlined in this paper. We then laid out the future work that must be done in order to bring this system into production, such as training the model to detect all possible objects that may be found in the collision region, controlling the activation of the RTOD application, and efficiently stopping the detector motion.

1. INTRODUCTION

The absence of a collision detection and prevention system around the detectors of the SPECT-CT machine has led to many collision-related accidents in the research bays, production floors, and scanning rooms. This project lays the groundwork for an intelligent collision prevention system by leveraging deep learning frameworks for computer vision to detect objects in the path of the detector's motion in real-time. Significant advancements have been made in object detection technology in the past decade thanks to numerous breakthroughs in Deep Neural Nets, made possible by large datasets and powerful computers. These breakthroughs have enabled widespread use of AI-powered computer vision in medical imaging, security and surveillance, automotive industry, robotics, and various other applications. However, research on collision detection is still in its early stages and the implementation methodologies can vary a lot from one use case to another. This project tackles the challenges involved with this unique case of collision prevention.

Figure 1.1 below shows a standard SPECT-CT machine found in the prototype bay, where research and development of the machine takes place. There are two detector heads in each machine, the bottom one is not visible in this picture as it is behind the patient bed. The detectors move into the space in front of the gantry on either side for various scan configurations, and that is the region where an object can collide with the detector heads. The detection mechanism is physically implemented by having two cameras monitor the region of potential collision on either side from a fixed diagonal angle and height, where 75-80% of the video frame consists of the region of collision, which covers the full range of detector motion. The live video feed in the camera is launched from an RTOD application, which uses a trained deep learning model and an object detection framework to detect external objects entering the region of collision in real time. Once an object is detected, a stop signal is triggered to halt detector motion. The implementation of this system follows five chronological stages:

1) Collecting, Preparing, and Processing Image Dataset
2) Training Deep Learning model with Image Data
3) Running Real-Time Object Detection Application
4) Optimizing Deep Learning Model to Improve Performance
5) Stopping Detector Motion when Object is Detected in Path of Collision

Before diving into the research work and findings in each stage, a choice had to be made on which objection detection framework was best suited for this use case. The next section discusses all the deep learning object detection frameworks that were explored in order to make this choice.

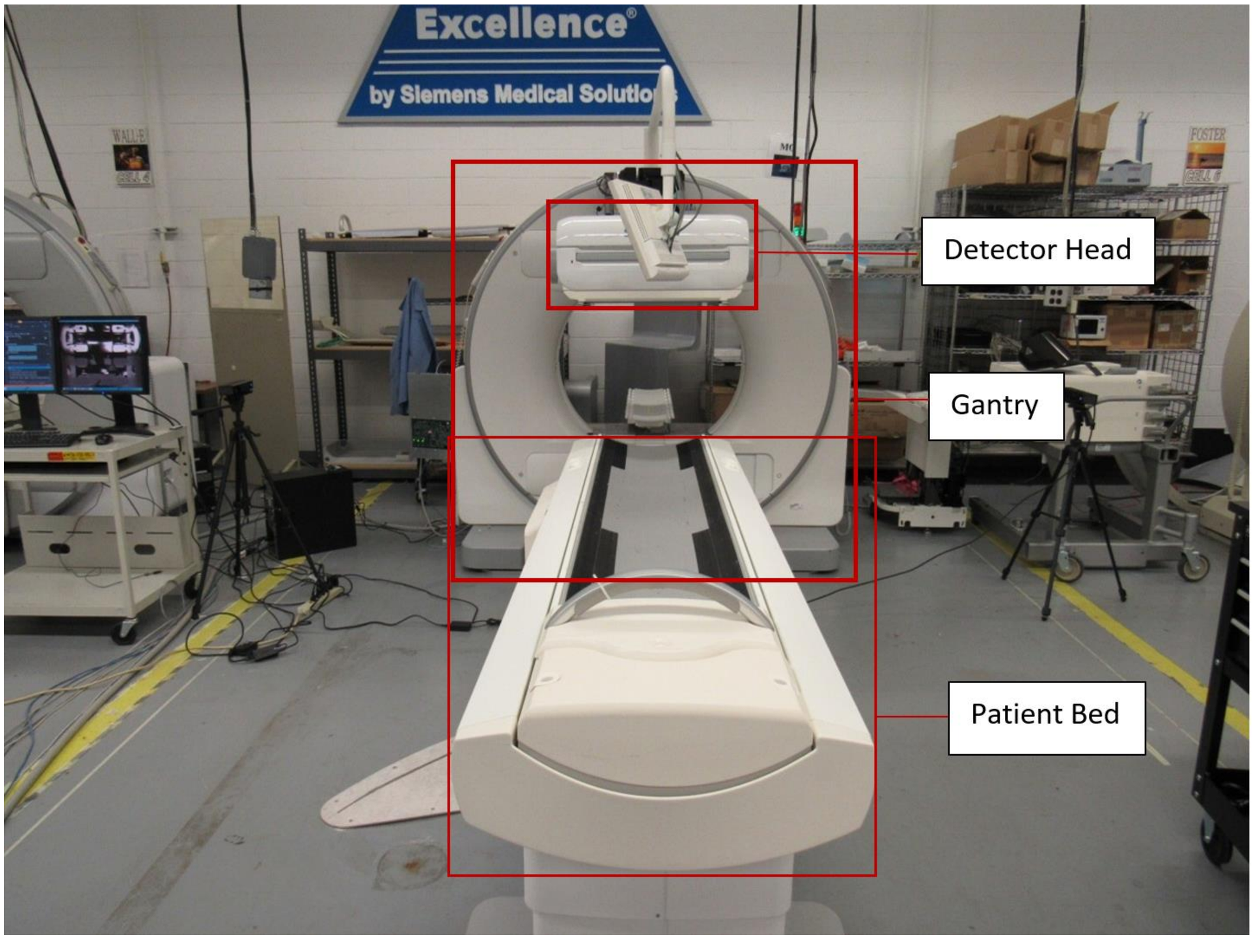


Figure 1.1. Frontal view of a SPECT-CT machine in a prototype bay with the detector head., gantry, and patient bed labelled.

2. DEEP LEARNING OBJECT DETECTORS

Object Detection can be performed with Deep Learning or other Computer Vision techniques. However, both approaches build on image classification and seek to localize the exact locations of objects within an image. When performing standard image classification, an input image is fed into the neural network to obtain a single class label, possibly with the probability score for the predicted label. This prediction is made for a single object in the entire image. When performing object detection, given an input image, we wish to obtain:

- A set of bounding boxes, or (x,y)-coordinates, locating each object in the image.
- The class labels predicted for each bounding box.
- The confidence score for each bounding box, which is the probability that the predicted object is within the bounding box.

The difference between image classification and object detection is illustrated in figure 2.

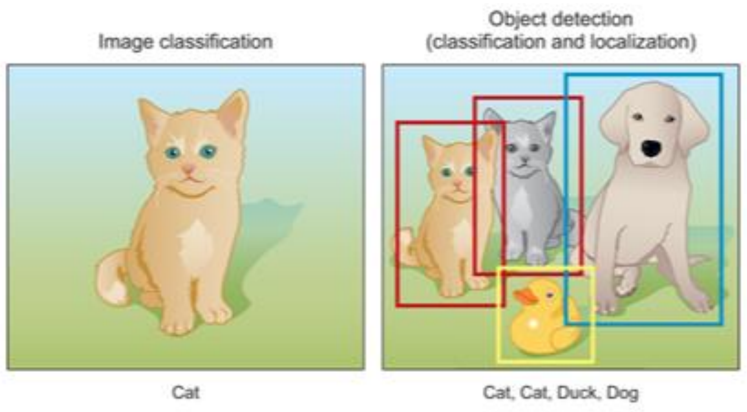


Figure 2.1. Image Classification vs. Object Detection [1].

Given a Convolutional Neural Network that has been trained to classify target objects in an image, object detection methods that do not use end-to-end deep learning utilize sliding windows in conjunction with image pyramids to locate objects. The sliding window slides from left-to-right and top-to-bottom to localize objects in different locations and the image pyramid detects the target objects at varying scales, as depicted in figure 2.2.

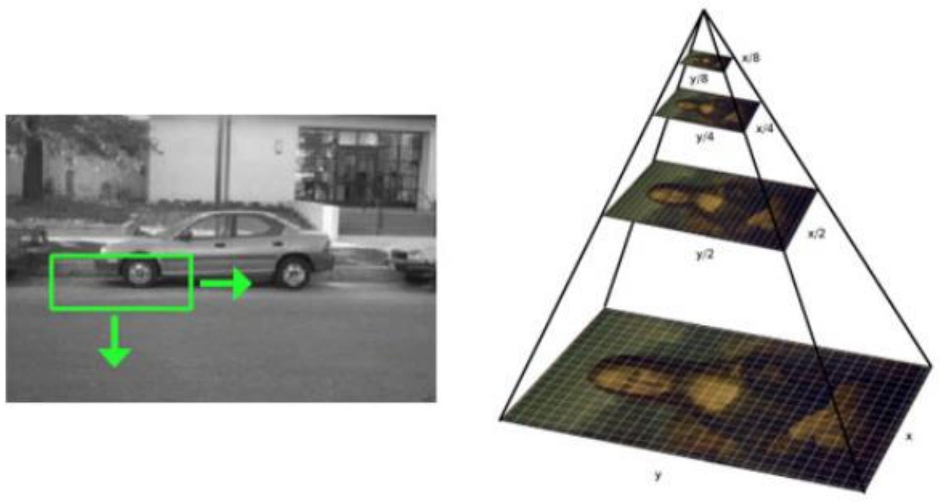


Figure 2.2. Left picture depicts Sliding Windows, right picture depicts Image Pyramids [2].

The problem with this traditional approach is that it is generally slow and a bit error prone. This method would not work for a real-time collision prevention system as a collision is very likely to occur by the time an object is detected. Therefore, implementing an end-to-end deep learning method was the best choice for this project. In this method, the pre-trained image classification network is fitted in as a **base network** within a **deep learning object detection framework** – the two core components of a deep learning object detector. The next step was to select a deep learning object detection framework that is best suited for this project. There are three well-established object detection frameworks – Faster RCNN [3], You Only Look Once (YOLO) [4], and Single-shot Detection (SSD) [5]. To decide which model is the most suitable choice, the features and detection results of each framework is explored without diving deep into the architectures.

2.1 Comparing Faster R-CNN, SSD, and YOLO

Faster R-CNN was first published in 2015, and it is the third and latest version of the R-CNN family [3]. It utilizes Region Proposal Networks (RPN) followed by Region of Interest (ROI) pooling to generate bounding boxes, then followed by feature extraction of objects within the boxes using a CNN, and finally a classification layer to predict which class the object belongs to. Although Faster R-CNN achieved state of the art accuracy, the whole process runs at a speed between 5 and 7 frames per second, making it unsuitable for real-time detection. YOLO, which has a similar architecture to Faster R-CNN, uses k-means clustering strategy on the training dataset to determine suitable anchor box sizes [4]. However, it is much faster than R-CNN with a speed of 45 frames per second and it also makes less background errors compared to Faster R-CNN. Due to the high speed of YOLO, it is quite popular in real-time object detection tasks in computer vision. However, YOLO's main weakness is that it leaves much accuracy to be desired, with relatively low recall and high localization error compared to Faster R-CNN. It also struggles to detect close objects and small objects. SSD, originally developed by Google researchers in 2016 [5], consists of the following key features:

- **Single Shot:** the tasks of object localization and classification are done in a single forward pass of the network.
- **MultiBox:** name of the technique used for bounding box regression developed by Szegedy et al.
- **Detector:** To indicate that the network is an object detector which also classifies the objects detected.

SSD uses VGG-16, an image classification network, as the base model because of its strong performance in high quality image classification tasks as well as its utility in *transfer learning*, which is discussed later. It builds on the VGG-16 architecture by removing the fully connected layers and adding a set of auxiliary convolutional layers, as depicted in figure 2.2 below.

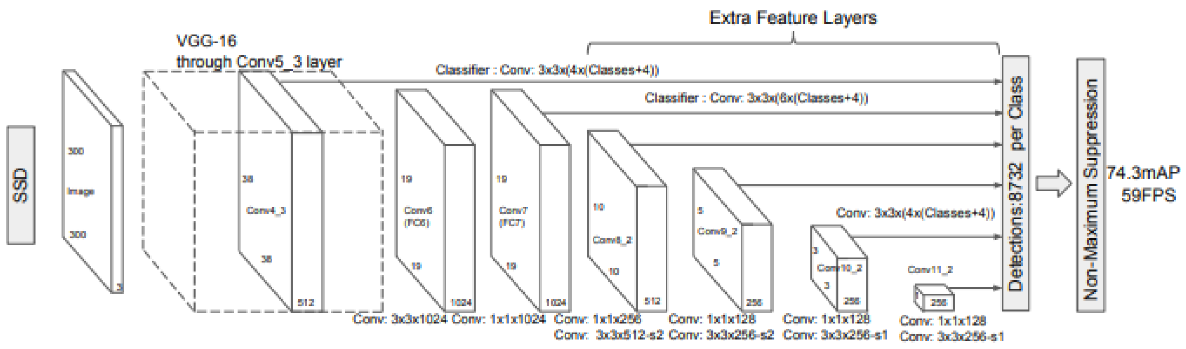


Figure 2.3 . Architecture of Single Shot Multibox Dectector (input is 300x300x3) [5].

SSD achieves accuracies that are similar, and in some cases even better, than Faster R-CNN while speeding up the detection process by eliminating RPN's and using low resolution images. SSD therefore provides a great balance between Faster R-CNN and YOLO with its high, real-time compatible speed and high detection accuracies, making it the obvious choice for this project. There are two models of SSD – SSD 300 and SSD512. SSD300 fixes input size to 300x300 and SSD512 fixes input size to 512x512. SSD300 uses lower resolution images to achieve faster processing speeds, but it is less accurate than SSD512. The table below compares the results of the three architectures on the Pascal VOC2007, an image dataset for object class recognition. Accuracy is measured in mAP(mean average precision), which is the most reliable evaluation metric in object recognition tasks, and speed is measured in FPS(frames per second) for a batch size of 1.

Table 2.1. Results of Faster R-CNN, YOLO, SSD300, and SSD512 on Pascal VOC2007 dataset. All methods used a batch size of 1.

Framework	**mAP**	**FPS**	**No. of Boxes**	**Input Resolution**
Faster R-CNN (VGG16)	73.2	7	~ 6000	~ 1000x600
YOLO (VGG16)	66.4	21	98	448x448
SSD300	74.3	46	8732	300x300
SSD512	76.8	19	24564	512x512

2.2 MobileNet SSD

As discussed in the previous section, SSD300 would be the most suitable choice for this project due its high processing speed, high mAP, and low resolution of input images. Most commercially available USB cameras would be compatible with real-time detection speeds of 30-45 FPS, making it an economic option as well. We then had to choose the most suitable image classification network that would fit in as the base network given the needs and constraints of this project. VGGNet, ResNet, MobileNet, and AlexNet are among the most reliable CNN architectures that have performed very well on large image datasets, such as ImageNet. Although SSD300 was originally built using VGGNet, this base model has a large network size which requires a high number of computations. The same applies for ResNet and AlexNet, with sizes ranging from 200 to 500 MB. MobileNet is a lightweight network architecture designed by Google researchers [6] for resource constrained devices, such as smartphones and Raspberry Pi. It has a simple architecture consisting of a 3x3 depthwise convolution followed by a 1x1 pointwise convolution as illustrated in figure 2.4, making it much lighter compared to its counterparts. This does come at the cost of lower accuracy than the heavier architectures. However, when the MobileNet architecture is combined with the SSD framework, we arrive at a fast, efficient deep-learning method for object detection. A MobileNet-SSD model that was first trained on the COCO dataset to detect 20 common objects and then *fine-tuned* on the Pascal VOC dataset reached a record high mAP of 72.7%. Thus, MobileNet-SSD stood out as the most suitable choice for indoor object detection. The next section discusses the work done to collect and process the image dataset for this project.

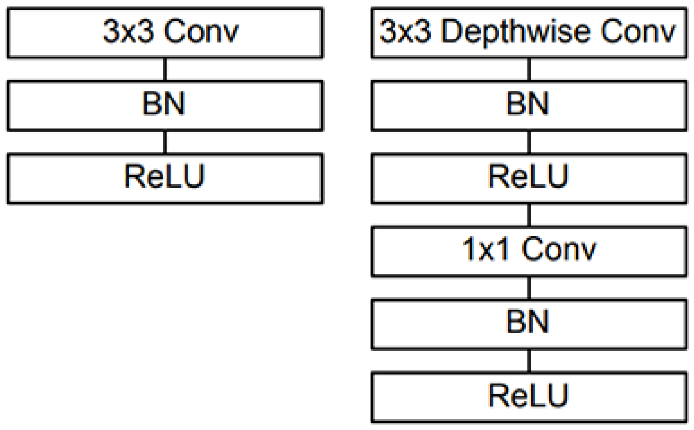


Figure 2.4. Left: Standard convolutional layer with batchnorm and ReLU. Right: depthwise separable convolutions with depthwise and pointwise layers followed by batchnorm and ReLU. [6].

3. COLLECTING, PREPARING, AND PROCESSING IMAGE DATASET

The quality of results produced by Deep Learning methods is highly dependent on the quality of data that is fed as input. This is well described by the phrase 'Garbage In, Garbage Out', which is often used among machine learning practitioners to emphasize the importance of good quality data. Even the best performing machine learning architectures will produce inaccurate, incomplete, and incoherent results if the input data is not properly standardized and processed according to the needs of the pertinent use case. This section discusses the work done to collect, prepare, and process images for this particular use case.

3.1 Collecting and Preparing Image Dataset

For our use case, we need to ensure that the detectors are not misclassified during motion when there are no external objects present in the collision region, thereby causing an unintended stop. To prevent such misclassifications, the object detector must be trained to detect various angles and configurations of the SPECT detectors with high accuracies from the camera's viewpoint. To do this, an object detection framework requires several pictures of various detector configurations as viewed by the cameras on either side. However, we need not ensure high detection accuracies for external objects entering the collision region since false classifications will also stop detector motion. For example, if a tray table in the path of collision is misclassified as a chair, it will still halt detector motion as all objects that are not detectors trigger the stop. Nonetheless, the object detector should be trained to detect a wide variety of common objects that may be present around the detectors – even for generating misclassifications.

Collecting reliable data for image classification takes a lot of time and effort. While image datasets for common objects are available on the internet, images of uncommon objects, such as SPECT detectors, must be manually taken and processed. As a rule of thumb, it is widely recommended to keep at least 1000 images per class for deep learning image classification. Due to the limitations of conducting a solo project, two external objects were selected for testing the efficacy of MobileNet-SSD in our use case – People and Collimator Carts. These are the two most common objects found near the SPECT detectors, and therefore the most likely to collide with it. If we get the desired object detection results in real-time, the same methodologies can be applied to include other objects in the future. The following sections discuss the methods used to collect images and ensure that the training data for all three object classes are well optimized for our use case.

3.1.1 SPECT Detector Images

The rule of thumb of having at least 1000 images per class was derived from the results of the ImageNet challenge, where classifiers like AlexNet was successfully trained to classify 1000 classes with high accuracies by using a dataset consisting around 1000 images per class (most classes had around seven or eight hundred). Since we are training MobileNet to correctly classify only 3 classes, it follows that a smaller fraction of the number needed to train 1000 classes should prove to be sufficient. We can further reduce the number of images needed for robust training by using a method known as *transfer learning*. Indeed, we will be using this method as discussed later. It is also important to note that training deep learning networks is a highly iterative process, where all the variables that can affect the results need to be adjusted with every iteration to obtain better results. Therefore, there is no formula to pre-determine the ideal settings for a variable without testing it. The amount of training data used per class is another such variable, for which we take a 'best guess' estimate on a good starting point and iteratively improve upon it.

After taking these factors into consideration, 300-400 images of SPECT detectors were deemed as a good starting point. The dataset had to include images of all the common detector configurations for various scans, as well as mid-motion images of the detector heads as they move into a specific configuration. This is illustrated in figure 3.1, where images of the detector were taken as they moved from the home position to CT scan position as viewed by the fixed camera positions on either side. The same method was followed for five of the other configurations, with ten photos of each configuration per side. 60 photos of various detector configurations and 40 mid-motion photos were taken from each side, summing up to 200 images of the detector heads without any external objects present. To robustly detect collision scenarios, rest of the detector images were taken with the object of interest, the collimator cart, present in the path of collision. Therefore, 180 pictures of the various detector configurations were taken with the collimator cart present in the path of collision, bringing the total number of detector images to 380. The SPECT/CT machine used for this project is a Symbia T series model located in a prototype bay. Although other models look very similar to this, they can come in different colors and sizes. If the data collection methods used for this prototype proves to be successful, the same methods can be applied to train a deep learning network to detect detector heads in all the other models and colors.

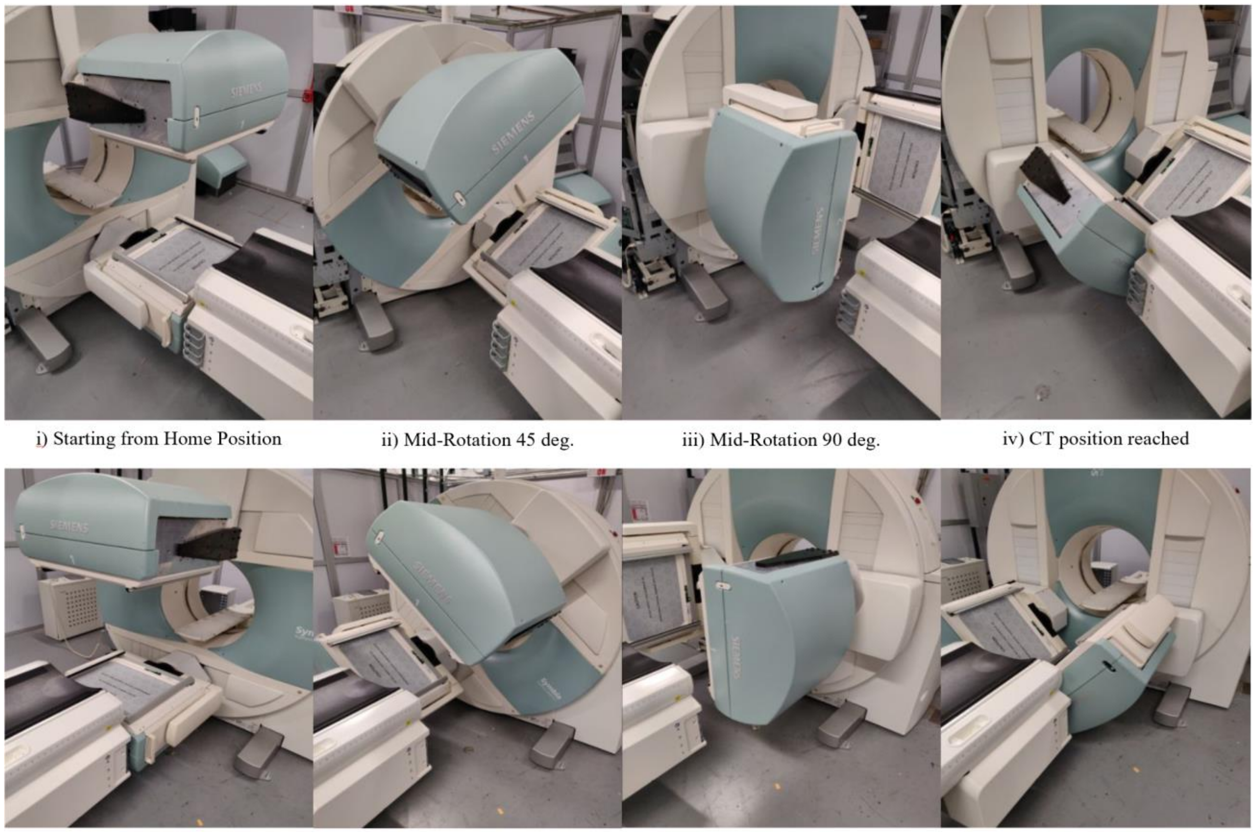


Figure 3.1. Photos of Detector Heads were taken in Home position, CT scan position, and mid-motion while moving to CT scan position from both sides.

3.1.2 Person Images

The PASCAL VOC datasets [7] contain more than 10,000 images of 20 object classes in everyday scenes, including people. The images have been standardized and annotated for object recognition training, thus saving a lot of pre-processing time. Furthermore, a MobileNet-SSD model that reached a mAP of 72.7% after 73,000 iterations of training on the VOC0712 dataset was made available for users. Pre-trained models enable the utilization of transfer learning, which is a highly effective optimization method in machine learning for accelerating the training and improving the performance of a neural network model. It is essentially a technique that allows us to bypass the time, effort, and computational resources needed for training a model from scratch on a very large dataset. The curve in figure 3.2 depicts the three ways in which transfer learning can improve training performance and inference results:

i) Higher start: The learned parameters from the original model start training of a new model from a higher skill-level.

ii) Higher slope: The rate of skill improvement is higher than it is for the original model trained from scratch.

iii) Higher asymptote: The converged skill of the new model is better than the original model.

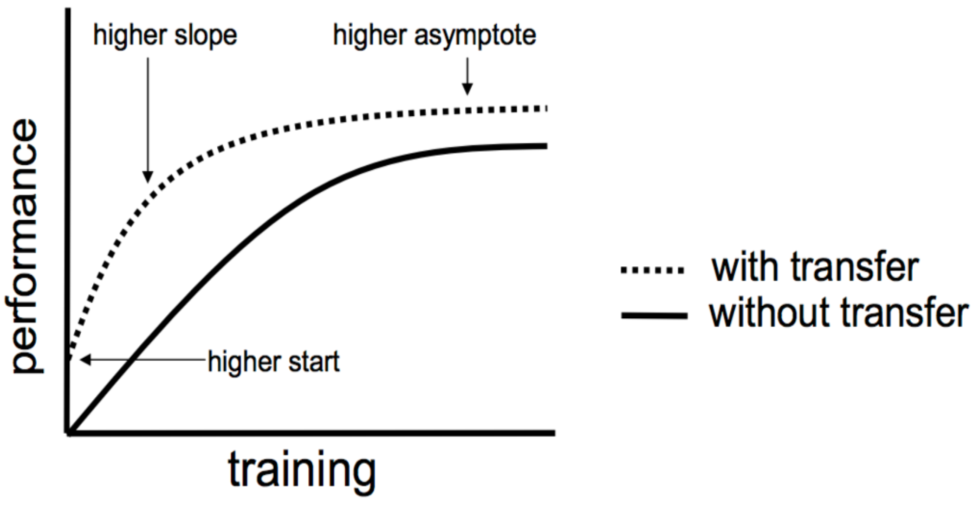


Figure 3.2. Three ways in which transfer learning improves performance. [8]

In deep learning, learning can be transferred by passing in the weights of a pre-trained model as a starting point for training a new model. Implementing transfer learning as a weight initialization scheme is especially useful for object recognition problems where other models have been trained to detect some or all target classes. However, the datasets used in the pretrained model and the new model must be of a similar *domain* to reap the benefits of transfer learning. For example, weights of a model trained to recognize indoor objects will not transfer well to a model being trained on images of CT scans and satellite photos. Since our target classes and the VOC classes are all indoor objects, we will not run into this problem. By initializing weights that have been pre-trained to detect the 'person' class, we do not need to feed the new model with all the original images of people used for training the pre-trained model. Due to all these reasons, the VOC2007 dataset was the ideal source for gathering people images for this use case.

Although it is possible to train a new model well using as few as 20-30 images of a transferred class, 100 was determined as a prudent starting number for people images. Since the deployed model will be seeing an indoor setting with no more than 2-3 people in the frame, 100 indoor images with 1-3 people were selected from the VOC2007 dataset. Although it is ideal to include images of people standing in the collision region for our use case, doing so was not tractable during this research.

3.1.3 Collimator Cart Images

The SPECT collimator is a thick, specialized sheet that is fixed on the detector heads for capturing radiation from a patient's target area. There are different types of collimators, and each type is designed for a specific type of scan. This means that collimators on the detector heads may need to be replaced if a scan being performed requires collimators of another type. The collimator cart is a rolling cart which has four drawers, and each drawer holds a specific type of collimator. When collimators on the detector heads need to be replaced with ones in the cart, the collimator cart is rolled into a docking slot fixed underneath the patient bed and the desired collimator pair is placed into the integrated drawers from the cart's drawer before being placed on the detectors. Therefore, collimator carts are usually kept close to SPECT/CT machines, posing a potential collision risk. These carts come in one design whereas people and detectors come in various shapes, colors, and orientations. This means that we can obtain good detection results for collimator carts by using fewer images compared to people or detectors. Keeping this in mind, 200-250 images were deemed as a good starting number for collimator carts. 60 cart pictures were taken from various angles with no other target objects present, while 180 cart pictures were taken in collision scenarios alongside the detectors, totaling to 240 pictures with collimator carts in it. Figure 3.3 shows a 'collimator cart only' picture versus a 'collimator cart in collision scenario' picture from the dataset, and table 3.1 shows the final image count of images in our dataset.

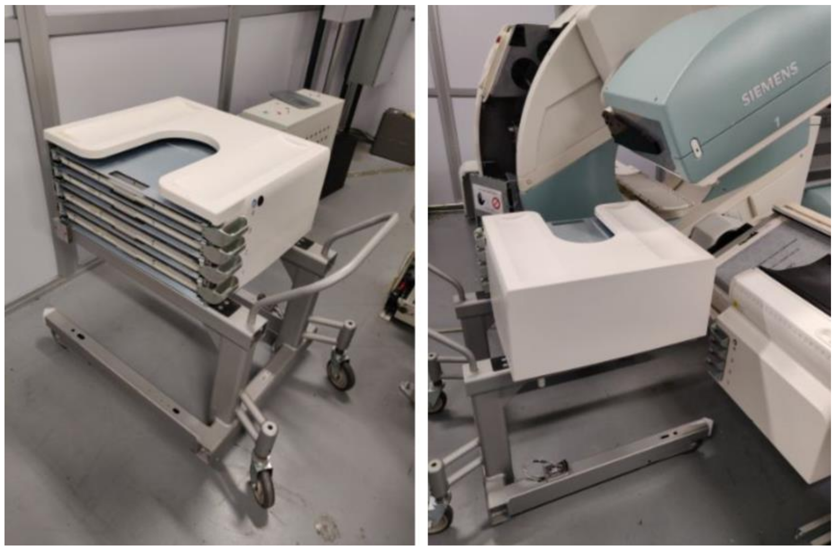


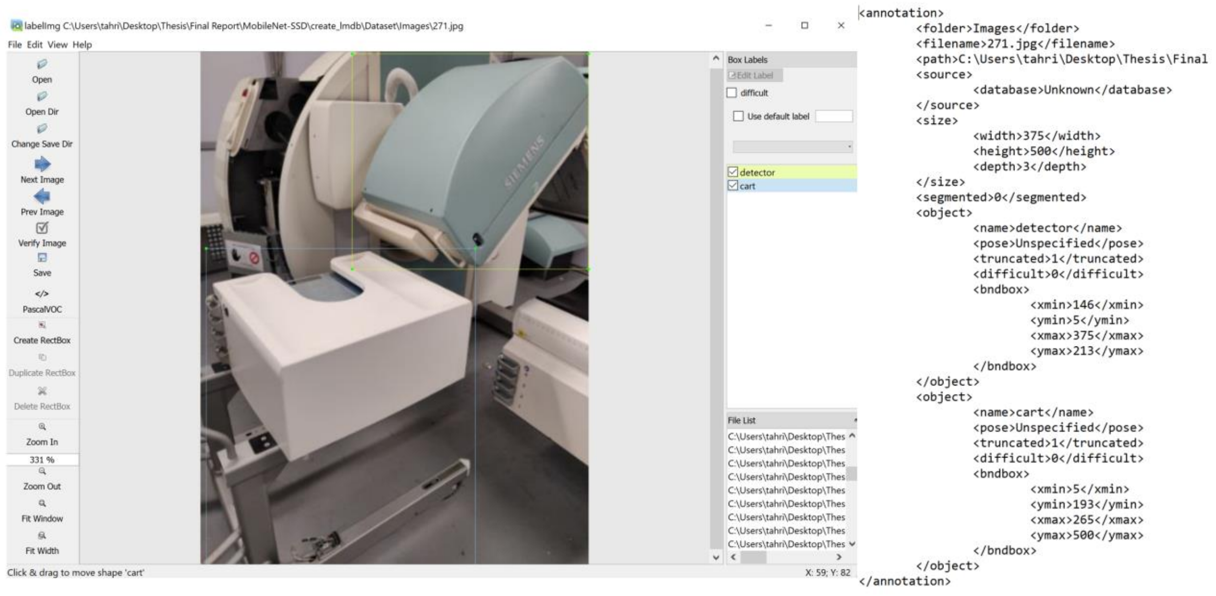
Figure 3.3. Left: Collimator Cart only. Right: Collimator Cart in collision region alongside detectors.

Table 3.1. Image Count Breakdown

Objects	Detectors only	Carts only	Detectors + Carts	People	Total Detector Images	Total Cart Images	**Overall Total**
Image Count	200	60	180	100	380	240	540

3.2 Processing Image Dataset

Once the image data set was collected, the next step was to *annotate* the images. Annotation is essentially the process of adding metadata to the dataset by tagging the data with target classes. This allows supervised machine learning models to compute errors due to differences in predictions and the ‘true’ labels read from the tagged metadata. Therefore, these labels help machine learning models recognize pixels within the annotated area as a distinct type of object. Drawing ‘bounding boxes’ around target objects in images, also known as ‘ground truth’ labels, is a common annotation technique for image datasets, and it is also used in MobileNet-SSD. For real-time videos, deep learning object detectors process video streams on a frame-by-frame basis where each frame is an image for which predictions are generated in the form of bounding boxes. Image annotation is a very time-consuming process as all target classes present in each image in the dataset must be labeled. The people images taken from the VOC2007 dataset came with annotations but the remaining 440 images in our dataset had to be labelled manually. This was done using *labelImg* [16], an image annotation tool written in Python. Figure 3.4 depicts the process of labelling an image within the tool and the XML file generated for the labelled image. Just like the people annotations, XML files for all the images were saved in the PascalVOC format since our implementation of MobileNet-SSD parses annotations in this format. Although people images did not require labelling, objects that are not included as targets had to be omitted from the annotations. This was done by simply removing all object tags that were not 'person' from the xml files.


labelImg C:\Users\tahri\Desktop\Thesis\Final Report\MobileNet-SSD\create_lmdb\Dataset\Images\271.jpg
File Edit View Help
Open
Open Dir
Change Save Dir
Next Image
Prev Image
Verify Image
Save
PascalVOC
Create RectBox
Duplicate RectBox
Delete RectBox
Zoom In
331 %
Zoom Out
Fit Window
Fit Width
Box Labels
Edit Label
difficult
Use default label
detector
cart
File List
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
C:\Users\tahri\Desktop\Thes
Click & drag to move shape 'cart'
X: 59; Y: 82
SIEMENS

```
<annotation>
        <folder>Images</folder>
        <filename>271.jpg</filename>
        <path>C:\Users\tahri\Desktop\Thesis\Final
        <source>
                <database>Unknown</database>
        </source>
        <size>
                <width>375</width>
                <height>500</height>
                <depth>3</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>detector</name>
                <pose>Unspecified</pose>
                <truncated>1</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>146</xmin>
                        <ymin>5</ymin>
                        <xmax>375</xmax>
                        <ymax>213</ymax>
                </bndbox>
        </object>
        <object>
                <name>cart</name>
                <pose>Unspecified</pose>
                <truncated>1</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>5</xmin>
                        <ymin>193</ymin>
                        <xmax>265</xmax>
                        <ymax>500</ymax>
                </bndbox>
        </object>
</annotation>
```

Figure 3.4. Left: Annotating a picture of a collision scenario in labelImg by drawing a blue bounding box around the detector and a green bounding box around the collimator cart. Right: Annotation of the image as an xml file, which includes heights and widths of each object's bounding box in pixels.

Normalizing data is the final stage of pre-processing input data prior to feeding it to a deep neural network. Normalization methods assign a common mean and unit variance to all input data samples, thereby normalizing all inputs to a standard scale. This allows learning algorithms to *converge* to optimal parameters more quickly. Luckily, the framework being implemented auto-normalizes all input images after re-scaling each image to 300x300 pixels. The framework also implements *batch normalization*, where input to each layer in the network is also normalized.

4. TRAINING AND DEPLOYING MOBILENET-SSD MODEL FOR FIRST DEMO

4.1 Caffe SSD

A *Caffe* [9] version of the original TensorFlow implementation of MobileNet-SSD was used for this project. Caffe is an open-source deep learning framework developed by Berkeley AI Research (BAIR) that offers several advantages with training deep neural networks, such as:

- Network architectures defined as human-readable plaintext schema in the form of ‘prototxt’ files [A.1], making network alterations much easier.
- Caffe is very fast at processing image data, reaching a speed of 60 million images per day with a single Nvidia K40 GPU.
- Hyperparameters and optimization methods can be tuned easily from “solver” text files [A.2] as well as from “prototxt” files.
- Switching between CPU and GPU mode by setting a single flag.

These features make the process of training a deep neural network very fast and convenient. Furthermore, some users have successfully implemented the Caffe version of MobileNet-SSD on image datasets of common objects, reaching high mAP values, and shared their pre-trained models online as downloadable ‘caffemodel’ files. By using the Caffe implementation, these pre-trained models can be leveraged to greatly speed up and optimize training. As mentioned in section 3.1.2, a caffemodel trained by GitHub user chuanqi305 [10] reached a mAP of 72.7% after 73,000 training iterations on the COCO dataset followed by finetuning on the PascalVOC dataset. The network was initialized with weights from this pre-trained caffemodel to reap the benefits of transfer learning. Since Caffe and many of its dependencies are only compatible with Linux, it was installed on an Ubuntu 20.04 machine after all its dependencies for GPU mode was installed and configured. This paper will not go over details on installing and configuring Caffe, however, it is important to note that the SSD version of the original Caffe repository was installed from source in order to implement MobileNet SSD. Wei Liu, author of the SSD paper, forked the main Caffe branch and made the requisite modifications to support SSD. Once Caffe and MobileNet-SSD were correctly installed and configured for GPU training, three additional steps were taken to prepare the image dataset for training:

i) The ‘labelmap.prototxt file’ within MobileNet SSD was modified to detect the three target classes: detector, person, and cart.

ii) The image data was split into training and testing sets with 80% of images set aside for training and validation, and 20% set aside for testing.

iii) The train and test sets were converted to LMDB format, which is one of the two formats in which Caffe reads input data.

4.2 Choosing Hyperparameters for Initial Training Run

In neural networks, *hyperparameters* comprise of all the adjustable variables that influence the learned parameters during supervised training. The learned parameters, which consist of the entire set of weights and biases for each unit(neuron) in the network, are ultimately responsible for generating predictions on unseen data. The hyperparameters that can influence training results can be categorized into two groups – ones related to network architecture and ones related to training. Network-related hyperparameters include number of hidden layers, number of units in each hidden layer, choice of activation functions etc., and training-related hyperparameters include number of iterations, learning rate, batch size, gradient descent optimization method, and many more. With so many hyperparameters that can affect results, it is impossible to know which set of hyperparameters will produce optimum results without iterating on different combinations. Thus, applied Deep Learning is a very empirical process where the choice of hyperparameters need to be adjusted iteratively until we approach optimum results. However, it is very important to start with a set of hyperparameters that serve as a good baseline for improvement. Although methods for choosing hyperparameters is still an active area of research among the deep learning community, there are various methods for selecting an initial set of hyperparameters that have proven to work very well for computer vision. Since MobileNet-SSD’s network architecture is already well designed for detecting common objects, this paper focuses on tuning hyperparameters that are specific to training. In this this section, the initial selection of key hyperparameters for training our model is explained.

4.2.1 Base Learning Rate

Given a neural network structure that is well suited for a learning task, the learning rate is often the most important hyperparameter when configuring the training process. The parameters within a neural network consist of weights and biases, and the learning process by which the network converges towards optimal parameters is known as *Gradient Descent*. In each step of gradient descent, weights and biases for each unit is updated to minimize the *Cost Function*, which computes the sum of losses for each prediction based on the differences between predicted values and true values. Moving in the direction of the steepest descent, as defined by the negative gradient of the cost function with respect to weights and biases, minimizes the cost, thereby increasing the accuracy of predictions. The learning rate determines the size of step towards this downward direction for each update. The basic equations for updating weight and bias for a given unit in each updating step is as follows:

$$w := w - \alpha \frac{dJ(w,b)}{dw} \, , \; b := b - \alpha \frac{dJ(w,b)}{db} \quad (4.1)$$

In equation 4.1, w is weight, b is bias, α is learning rate, and J(w,b) is the cost function. A value of α that is too small may result in very long training periods without ever converging near a satisfactory minimum of J(w,b), whereas a value too large may result in overshooting the minimum to end up with a sub-optimal set of parameters. This can be visualized using 2D plots of loss functions as shown in figure 4.1, where θ represents a set of parameters in the x-axis, loss is represented in the y-axis, and each arrow in the loss function J(θ) represents an updating step.

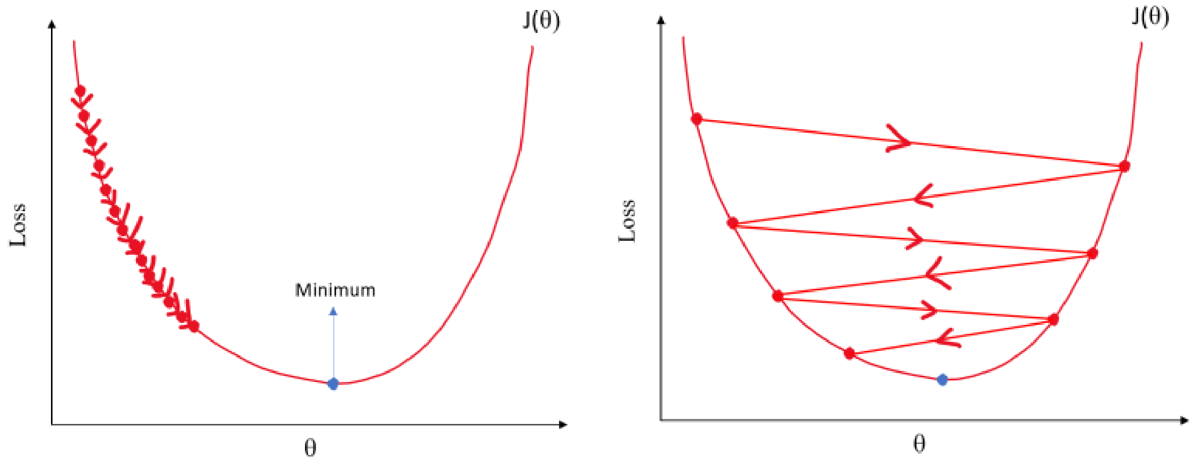


Figure 4.1. Left: α is too small, resulting in too many steps to converge near minimum. Right: α is too large, resulting in overshooting the minimum.

The base learning rate is the initial value of α for training a network. As the algorithm approaches a minimum, the learning rate can be reduced to ensure that smaller steps are taken in an attempt to end up as close to the minimum as possible. For training deep learning object detection networks from scratch, base learning rates are typically in the range of 0.01 – 0.001. Once the low-level features have been learned, the learning rate is dropped in order to fine-tune the weights for learning more nuanced, higher-level features. Since we are initializing weights from a pre-trained model, many of the generic, low-level features for classifying objects have already been learned. This means that we can bypass the initial phase of training and start from the fine-tuning phase, where a lower base learning rate is used. Therefore, a base learning rate of 0.001 was used for the first training run.

4.2.2 Learning Rate Policy

As explained earlier, it is often useful to reduce, or ‘decay’, learning rate as the training progresses. This can either be done manually, by stopping training at self-determined points, or by using pre-defined scheduling policies. In most cases, it is practically unfeasible to constantly monitor training progress and manually pause training to change the learning rate. Therefore, learning rate schedules, also known as *learning rate policy*, are often used in practice. In Caffe, the following learning rate decaying policies are available for use:

- Fixed: always return base_lr.
- Step: return base_lr * gamma ^ (floor(iter / step))
- Exponential: return base_lr * gamma ^ iter
- Inverse: return base_lr * (1 + gamma * iter) ^ (- power)
- Multistep: similar to step but it allows non uniform steps defined by stepvalue
- Poly: the effective learning rate follows a polynomial decay, to be zero by the max_iter. return base_lr (1 - iter/max_iter) ^ (power).
- Sigmoid: The effective learning rate follows a sigmoid decay return base_lr (1/(1 + exp(-gamma * (iter - stepsize))))

In these decay equations, 'base_lr' is the base learning rate, 'iter' is the current iteration, and 'max_iter' is the maximum number of iterations. Values for gamma, step, stepvalues, and power can be varied from the solver file to get desired results. The learning rate policy selected for the first run was 'multistep', with gamma set to 0.5 so that the learning rate is halved at the stepvalues. The stepvalues were set to 20,000 iterations and 40,000 iterations, so that the learning rate is halved at those intervals with maximum number of iterations set to 50,000. Prior to the very first training run, it is not possible to know the range of iterations where parameters will be approaching a minimum. Therefore, the stepvalues were selected arbitrarily and kept equidistant from each other. The minimum is usually found in the range where training loss plateaus before it starts to increase. The range for a potential minimum can therefore be identified after monitoring training loss in the first run, and stepvalues can then be adjusted accordingly in later runs.

4.2.3 Batch Size

Batch size is the number of training examples passed through the network for each step of gradient descent. Batch sizes for gradient descent can be configured in three ways:

i) Batch Gradient Descent (BGD) – Batch size is set to the total number of examples in the training set.

ii) Stochastic Gradient Descent (SGD) – Batch size is set to one.

iii) Minibatch Gradient Descent (MGD) – Batch size is set to more than one but less than the total number of training examples in the training set.

Since forward and backward computations are performed on each sample for every updating step, training time increases with batch size. However, using more examples lead to a more accurate estimation of the direction towards the minimum of the error gradient. Conversely, using fewer examples lead to faster training times at the cost of less accurate, 'noisier' steps down the error gradient. Consequently, parameter updates are noisiest for SGD as shown in figure 4.2, and the noise in updates decrease as batch size increases. Due to this tradeoff between speed and accuracy of updates, the convergence of the learning process can be very sensitive to batch size.

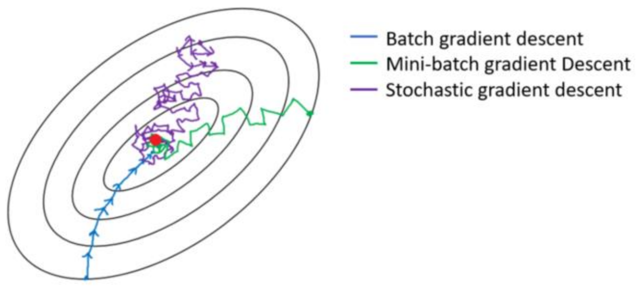


Figure 4.2. Updates in Batch vs Mini-batch vs Stochastic Gradient Descent [11].

When using large datasets, BGD is not a viable option due to restrictions in time and computational memory. On the other hand, SGD may require too many steps to converge near a minimum due to very noisy updates. Therefore, MGD is the most optimal choice for most deep learning tasks as it provides a balance between the two extremes. Batch size is typically chosen between 1 and a few hundreds, and for datasets with more than 1000 examples, 32 is a good default value [12]. Since our dataset contains 540 images with 432 used for training, batch size for the first training run was set to 12. Thus, one *epoch* will be reached in 36 iterations, and with maximum number of iterations set to 50,000, 1388 epochs will be reached after training ends.

4.2.4 Optimization Algorithm

As explained in the previous section, using minibatch results in 'noisy' steps where the estimated gradient descent step is directed away from the minimum of the cost function. Although the noise can help with generalizing the model to make correct predictions on unseen test samples, having too much noise over many steps rapidly slows down convergence towards a minimum. This is because the number of steps taken to approach the minimum increases a lot when too many steps significantly diverge away from the direction of the minimum. Gradient-based optimization algorithms speed up the convergence by 'dampening' out the noise so that each step ends up being closer to the minimum, thereby reducing the number of steps taken to approach closer to the minimum. The effect of using optimization algorithms is illustrated in figure 4.3 below, where the contours represent the error function, the red dot represents the minimum, the blue arrows represent gradient descent steps without using an optimization algorithm, and the orange arrows represent gradient descent steps when using an optimization algorithm.

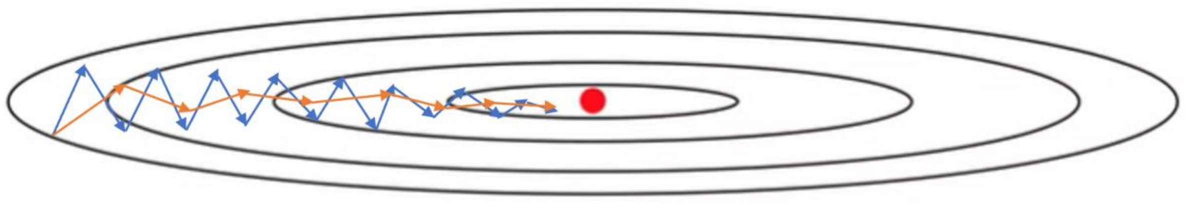


Figure 4.3. Effect of using Optimization Algorithms for Gradient Descent.

We can see in the figure that the number of steps taken to reach the same point near the minimum is greatly reduced when using a gradient-based optimization algorithm. Caffe provides implementations of six different methods for gradient-based optimizations, and all these methods aim to speed up convergence. Elaborating on the implementation details of each method is out of the scope of this paper, so we will only justify the selection of the method that is used for the first run – RMSprop. Unlike the other methods, RMSprop automatically adjusts the learning rate and chooses a different learning rate for each parameter. This makes it a suitable fit for the multistep learning rate decay policy used in the first training run.

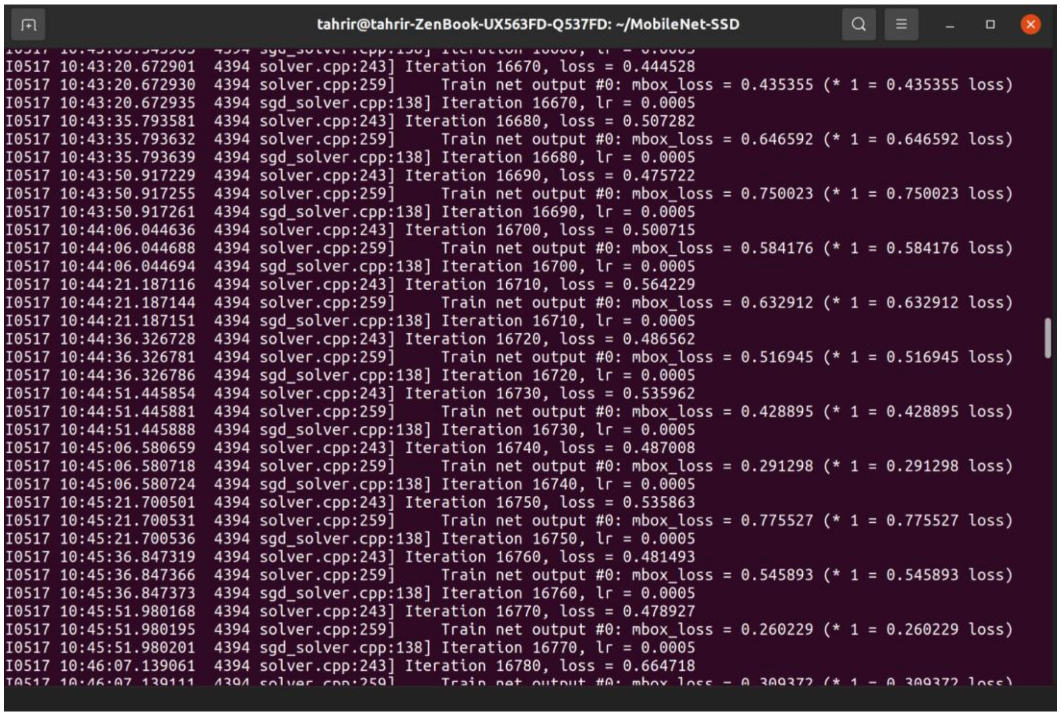
The key hyperparameter settings for the first training run can be summarized as follows:

- Base Learning Rate: 0.001
- Learning Rate Policy: Multistep. Stepvalue 1 = 20,000 iterations, Stepvalue 2 = 40,000 iterations.
- Minibatch Size: 12
- Optimization Algorithm: RMSprop.

4.3 Results of First Training Run

4.3.1 MobileNet-SSD training

Once the training hyperparameters were configured, weights from the caffemodel that was pretrained for 73,000 iterations were transferred to the network for the first training run. To monitor training progress, the iteration number, training loss, and testing loss is printed to the terminal in regular intervals, as shown in figure 4.4.



```
tahrir@tahrir-ZenBook-UX563FD-Q537FD: ~/MobileNet-SSD
I0517 10:43:20.672901  4394 solver.cpp:243] Iteration 16670, loss = 0.444528
I0517 10:43:20.672930  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.435355 (* 1 = 0.435355 loss)
I0517 10:43:20.672935  4394 sgd_solver.cpp:138] Iteration 16670, lr = 0.0005
I0517 10:43:35.793581  4394 solver.cpp:243] Iteration 16680, loss = 0.507282
I0517 10:43:35.793632  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.646592 (* 1 = 0.646592 loss)
I0517 10:43:35.793639  4394 sgd_solver.cpp:138] Iteration 16680, lr = 0.0005
I0517 10:43:50.917229  4394 solver.cpp:243] Iteration 16690, loss = 0.475722
I0517 10:43:50.917255  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.750023 (* 1 = 0.750023 loss)
I0517 10:43:50.917261  4394 sgd_solver.cpp:138] Iteration 16690, lr = 0.0005
I0517 10:44:06.044636  4394 solver.cpp:243] Iteration 16700, loss = 0.500715
I0517 10:44:06.044688  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.584176 (* 1 = 0.584176 loss)
I0517 10:44:06.044694  4394 sgd_solver.cpp:138] Iteration 16700, lr = 0.0005
I0517 10:44:21.187116  4394 solver.cpp:243] Iteration 16710, loss = 0.564229
I0517 10:44:21.187144  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.632912 (* 1 = 0.632912 loss)
I0517 10:44:21.187151  4394 sgd_solver.cpp:138] Iteration 16710, lr = 0.0005
I0517 10:44:36.326728  4394 solver.cpp:243] Iteration 16720, loss = 0.486562
I0517 10:44:36.326781  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.516945 (* 1 = 0.516945 loss)
I0517 10:44:36.326786  4394 sgd_solver.cpp:138] Iteration 16720, lr = 0.0005
I0517 10:44:51.445854  4394 solver.cpp:243] Iteration 16730, loss = 0.535962
I0517 10:44:51.445881  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.428895 (* 1 = 0.428895 loss)
I0517 10:44:51.445888  4394 sgd_solver.cpp:138] Iteration 16730, lr = 0.0005
I0517 10:45:06.580659  4394 solver.cpp:243] Iteration 16740, loss = 0.487008
I0517 10:45:06.580718  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.291298 (* 1 = 0.291298 loss)
I0517 10:45:06.580724  4394 sgd_solver.cpp:138] Iteration 16740, lr = 0.0005
I0517 10:45:21.700501  4394 solver.cpp:243] Iteration 16750, loss = 0.535863
I0517 10:45:21.700531  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.775527 (* 1 = 0.775527 loss)
I0517 10:45:21.700536  4394 sgd_solver.cpp:138] Iteration 16750, lr = 0.0005
I0517 10:45:36.847319  4394 solver.cpp:243] Iteration 16760, loss = 0.481493
I0517 10:45:36.847366  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.545893 (* 1 = 0.545893 loss)
I0517 10:45:36.847373  4394 sgd_solver.cpp:138] Iteration 16760, lr = 0.0005
I0517 10:45:51.980168  4394 solver.cpp:243] Iteration 16770, loss = 0.478927
I0517 10:45:51.980195  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.260229 (* 1 = 0.260229 loss)
I0517 10:45:51.980201  4394 sgd_solver.cpp:138] Iteration 16770, lr = 0.0005
I0517 10:46:07.139061  4394 solver.cpp:243] Iteration 16780, loss = 0.664718
I0517 10:46:07.139111  4394 solver.cpp:259]     Train net output #0: mbox_loss = 0.309372 (* 1 = 0.309372 loss)
```

Figure 4.4. Training log printed on terminal window every 10 iterations.

We can see from the figure that the interval for printing training loss was set to 10 iterations and the interval for printing test accuracy, measured in mAP, was set to 1000 intervals. Trained weights were being saved as caffemodel files in the 'snapshot' folder after every 1000 iterations. Once training was complete after 50,000 iterations, we went over the test mAP values at every 1000 iterations to identify the three caffemodels with the highest mAP values. These were the models trained for 16000, 20000, and 26000 iterations. The next step was to deploy these models in a real-time video stream of the collision region by launching the Real-Time Object Detection (RTOD) application.

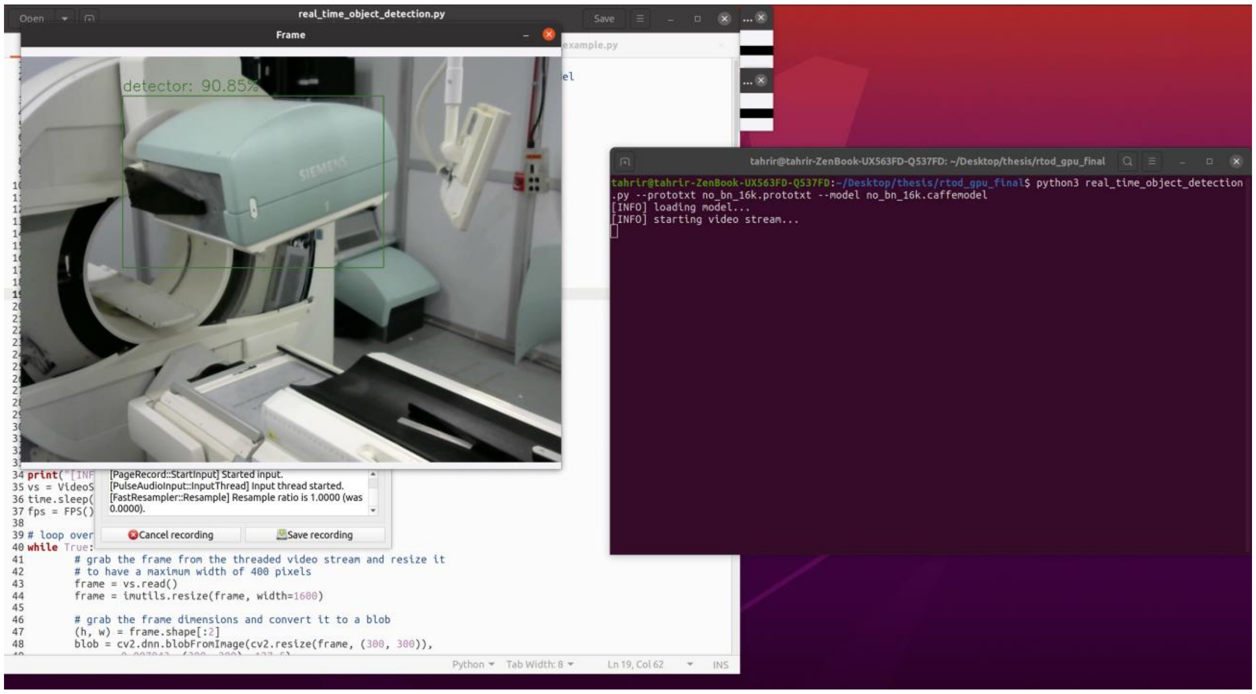
4.3.2 Real-time Object Detection Application Demo

The Real-Time Object Detection (RTOD) application, real_time_object_detection.py, along with the three caffemodels and their deployment prototxt files were moved into a folder for the demo. The RTOD python script, originally taken from pyimagesearch [13] and altered for this use case, imports various OpenCV libraries for accessing the camera, deploying a pre-trained deep neural network within a live video stream, and displaying customized bounding boxes of detected objects. The fully commented code of the application can be found in Appendix B [B.1]. The script is launched from the terminal window, and it takes two mandatory arguments – the pre-trained caffemodel file being deployed, the deployment prototxt file defining the network. For example, the command for running the script with the caffemodel trained for 20000 iterations can be as follows: "$ python3 real_time_object_detection.py --prototxt no_bn_20k.prototxt --model mobilenet_iter_20000.caffemodel". The prototxt file being deployed has the prefix "no_bn" because the *BatchNorm* layers were removed. Batch normalization layers are only required for training and redundant for deployment purposes. The minimum confidence threshold for detecting objects were set to 30% in the code, meaning any bounding boxes with a lower confidence score will not be detected. For the demos, each of the three caffemodels were separately deployed from the application and the results were evaluated.

A Logitech C270 720p Webcam was used to monitor the collision regions for the demos. With a throughput of 30fps, the webcam is highly compatible with MobileNet-SSD's detection frame rate. Once the camera was fixed in the correct position looking into the collision region from one side, the RTOD application was launched for the three caffemodels in three separate scenarios:

1) Only the detectors are present as the gantry rotates.
2) A cart is pushed into the region of collision.
3) A person walks into the region of collision.

After testing all three caffemodels by running multiple demos per model, the one trained for 16000 iterations produced the best results. The demos were screen recorded and the links to the demos have been attached in Appendix C1.

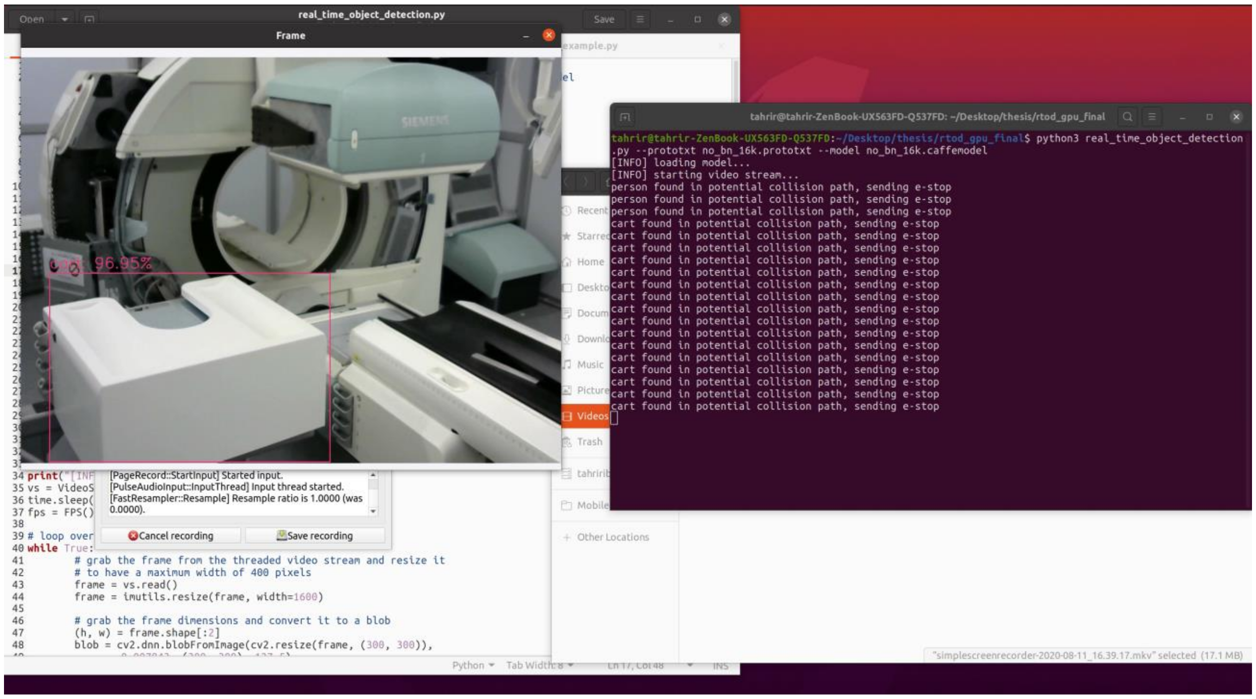


```
print("[INF
vs = VideoS
time.sleep(
fps = FPS()

# loop over
while True:
        # grab the frame from the threaded video stream and resize it
        # to have a maximum width of 400 pixels
        frame = vs.read()
        frame = imutils.resize(frame, width=1600)

        # grab the frame dimensions and convert it to a blob
        (h, w) = frame.shape[:2]
        blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
```

```
[PageRecord::StartInput] Started input.
[PulseAudioInput::InputThread] Input thread started.
[FastResampler::Resample] Resample ratio is 1.0000 (was
0.0000).
```

```
tahrir@tahrir-ZenBook-UX563FD-Q537FD:~/Desktop/thesis/rtod_gpu_final$ python3 real_time_object_detection
.py --prototxt no_bn_16k.prototxt --model no_bn_16k.caffemodel
[INFO] loading model...
[INFO] starting video stream...
```

Figure 4.5. RTOD demo with no objects entering collision region.

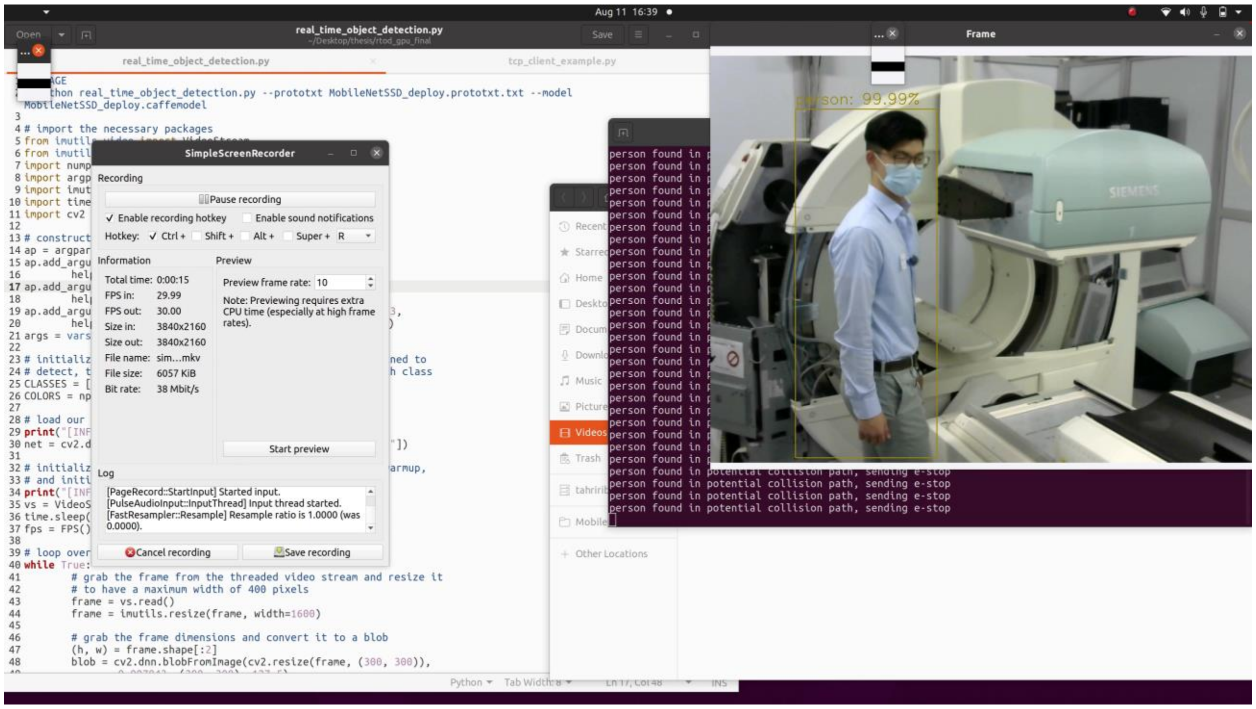


```
print("[INF
vs = VideoS
time.sleep(
fps = FPS()

# loop over
while True:
        # grab the frame from the threaded video stream and resize it
        # to have a maximum width of 400 pixels
        frame = vs.read()
        frame = imutils.resize(frame, width=1600)

        # grab the frame dimensions and convert it to a blob
        (h, w) = frame.shape[:2]
        blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
```

```
[PageRecord::StartInput] Started input.
[PulseAudioInput::InputThread] Input thread started.
[FastResampler::Resample] Resample ratio is 1.0000 (was
0.0000).
```

```
tahrir@tahrir-ZenBook-UX563FD-Q537FD:~/Desktop/thesis/rtod_gpu_final$ python3 real_time_object_detection
.py --prototxt no_bn_16k.prototxt --model no_bn_16k.caffemodel
[INFO] loading model...
[INFO] starting video stream...
person found in potential collision path, sending e-stop
person found in potential collision path, sending e-stop
person found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
cart found in potential collision path, sending e-stop
```

Figure 4.6. RTOD demo with cart entering collision region.



```
AGE
thon real_time_object_detection.py --prototxt MobileNetSSD_deploy.prototxt.txt --model
MobileNetSSD_deploy.caffemodel

# import the necessary packages
from imutil
from imutil
import nump
import argp
import imut
import time
import cv2

# construct
ap = argpar
ap.add_argu
        hel
ap.add_argu
        hel
ap.add_argu                                                     3,
        hel                                                     )
args = vars

# initializ                                                 ned to
# detect, t                                                 h class
CLASSES = [
COLORS = np

# load our
print("[INF
net = cv2.d                                                     "])

# initializ                                                 armup,
# and initi
print("[INF
vs = VideoS
time.sleep(
fps = FPS()

# loop over
while True:
        # grab the frame from the threaded video stream and resize it
        # to have a maximum width of 400 pixels
        frame = vs.read()
        frame = imutils.resize(frame, width=1600)

        # grab the frame dimensions and convert it to a blob
        (h, w) = frame.shape[:2]
        blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
```

SimpleScreenRecorder

Recording

Pause recording

✓ Enable recording hotkey ☐ Enable sound notifications

Hotkey: ✓ Ctrl + ☐ Shift + ☐ Alt + ☐ Super + R

Information

Total time: 0:00:15
FPS in: 29.99
FPS out: 30.00
Size in: 3840x2160
Size out: 3840x2160
File name: sim...mkv
File size: 6057 KiB
Bit rate: 38 Mbit/s

Preview

Preview frame rate: 10

Note: Previewing requires extra CPU time (especially at high frame rates).

Start preview

Log

```
[PageRecord::StartInput] Started input.
[PulseAudioInput::InputThread] Input thread started.
[FastResampler::Resample] Resample ratio is 1.0000 (was
0.0000).
```

Cancel recording   Save recording

```
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in p
person found in potential collision path, sending e-stop
person found in potential collision path, sending e-stop
person found in potential collision path, sending e-stop
person found in potential collision path, sending e-stop
```

Figure 4.7. RTOD demo with person entering collision region.

4.3.3 Inferring Results of Demos

At the time of running this demo, the issue of stopping detector motion upon detecting external objects was not yet resolved. To simulate a stop, the following message was printed to the terminal “[object] found in potential collision path, sending e-stop”. The results and corresponding inferences of the demos are as follows:

1) Detectors are not being detected in some configurations while being detected with low confidence scores in other configurations. We can see in the ‘16k-30-detector’(16000 iterations and default confidence of 30% being used in RTOD script) demo that the bottom detector is not detected while the top detector is detected with low confidence scores. As the gantry rotates in an anti-clockwise direction, the confidence score keeps increasing. This means the model is good at detecting the top of the detectors around the 45-degree angle, but it was not trained well enough to detect them in other configurations. These poor detections are primarily due to training with an insufficient number of images for every configuration. Only 15-20 images were used per configuration, such as Gurney mode, CT mode, 180 degree, and so on. However, the model should be trained with many more images per configuration to ensure high confidence detections at different angles.

Fortunately, we are not concerned with failure to detect the detectors at some angles for this use case. The detectors have been included only to prevent misclassifications that would stop the detector motion when no external objects are present.

2) The cart and person classes are being detected almost instantaneously (within milliseconds) with high confidence scores as they enter the region of potential collision, as seen in the ‘16k-30-person’ and ‘16k-30-cart’ demos. Thus, the model was successfully trained to detect these two classes in real-time, and the same methodologies can be applied for training MobileNet-SSD models to detect more objects. Again, the number of images required for robustly detecting an object depends on the dimensions and color patterns of the object. Since the dimensions and color patterns of carts are highly consistent, 240 images from various angles proved to be enough to generate high confidence detections.
3) The application incorrectly labels random regions with the ‘person’ class, as seen in the person and cart demos. This is the most concerning issue, as we do not want the model to unnecessarily stop motion due to incorrect detections. These misclassifications are most likely being caused by the model overfitting the training data due to an imbalance in the dataset. Weights for the person class trained for 73000 iterations were initiated in training whereas the detector class was trained from scratch with 380 images. This may be causing a biased training process where the network leans heavily towards “person” predicitons. Investigating and resolving this issue required a deeper dive into the methodologies behind improving the performance of a deep learning model, which is tackled in the next section.

5. IMPROVING AND OPTIMIZING MODEL PERFORMANCE

5.1 Image Augmentation

As discussed in the previous section, the model may have been overfitted with the person class as evident from the misclassifications in the demos. The best strategy for counteracting this imbalance is training the model with more images of the detector heads, while reducing the number of person images. We reduced the number of person images to 50, but manually taking hundreds of detector images is a highly cumbersome process. Image augmentation is a very useful technique by which a dataset can be artificially expanded without taking pictures manually. This is done by altering the existing pictures of an object to create more pictures of the same object through transformations such as rotation, cropping, shifting, flipping, etc. The altered images also add more variety to the model, which can result in better generalization of unseen test data. Therefore, the existing pictures of the detector heads were augmented to add more images to the training data.

The Keras deep learning neural network library provides image augmentation capabilities via the *ImageDataGenerator* class. The code that was written to implement the library on detector images can be found in Appendix B [B.2]. By using the ImageDataGenerator class, we applied random rotation, shifting, cropping, and horizontal flipping with only a few lines of code. 200 images of the detector heads were augmented to generate 200 modified versions, bringing the total number of detector images to 580 and the training split to 464. Figure 5.1 shows some of the original images and their augmented versions side by side.

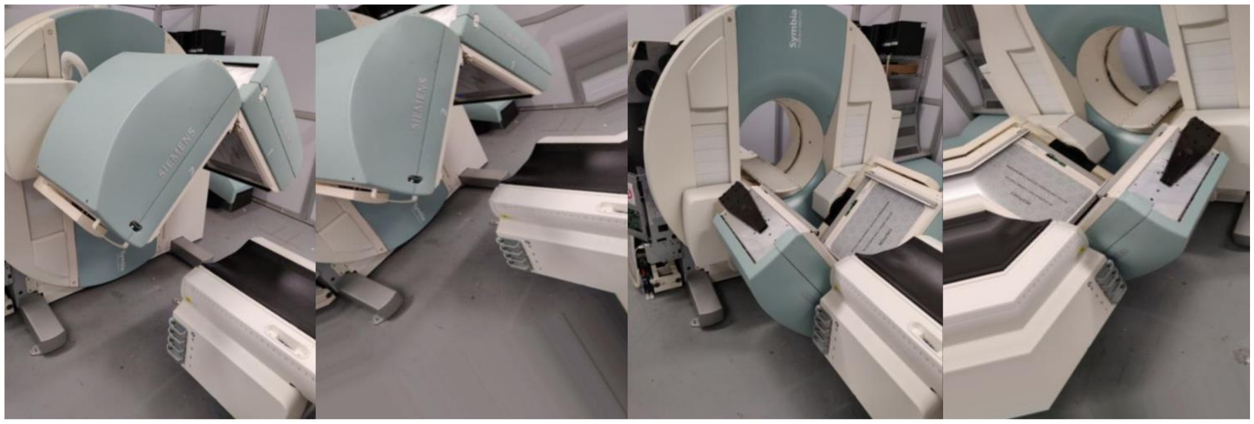


Figure 5.1. Second from left: randomly rotated and cropped version of leftmost image. Rightmost: Randomly rotated and flipped version of second from right image.

Table 5.1. Image Count Breakdown for Optimized Training Runs

Objects	Detectors only	Carts only	Detectors + Carts	People	Total Detector Images	Total Cart Images	**Overall Total**
Image Count	400	60	180	50	580	240	690

5.2 Regularization

After adding more data, we can implement a technique called *Regularization* to prevent overfitting. This technique reduces the complexity of a model by shrinking weights of parameters, resulting in a classifier that fits more "loosely" with the training data which helps the model generalize better with unseen test data. Figure 5.2 illustrates the basic concept with a simple example where price is predicted from size. We see that using a linear classifier causes underfitting and using a 4th degree polynomial causes overfitting, while a 2nd degree polynomial provides just the "right fit". Regularization helps prevent a model from becoming overly complex by reducing variance and increasing bias, thereby bringing a classifier closer to the "right fit".

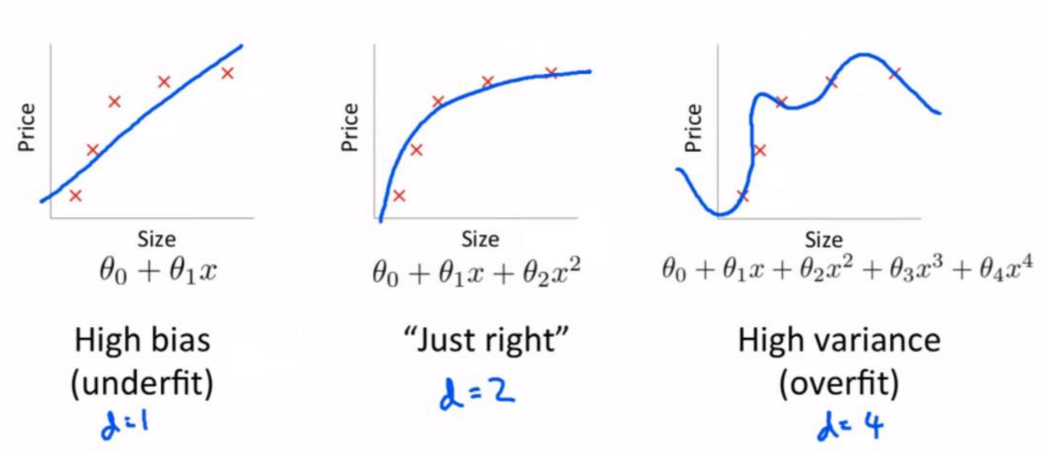


Figure 5.2. Bias and Variance tradeoff [14].

It is important to note we need several parameters to form the non-linearities required to solve complex deep learning problems, unlike the figure above. So instead of eliminating parameters, regularization works by adding a penalty to the loss function that shrinks, or even nullifies, the weights associated with “less important” parameters that do not weigh heavily on a specific outcome. For this use case, we are using “L2” regularization which is also known as *Ridge Regression*. The penalty term added to the loss function for this is: **weight_decay*sum(w^2)**, where weight decay determines how dominant the regularization term will be during gradient computation, and sum(w^2) is the sum of the squares of all the weights for the layer being computed. As a rule of thumb, the higher the number of training examples, the weaker this term should be, and the deeper the neural net, the higher this term should be. Since we do not have many training examples and we are training a very deep neural network, we increased the weight decay from 0.00005 in the first run to 0.001 for the upcoming runs.

5.3 Hyperparameter Tuning

As discussed in section 4, the key hyperparameters which dictate the performance of a deep learning model are learning rate, batch size, and optimization algorithm. To improve model performance, we need to tune these hyperparameters based on the results of the first run. After choosing a set of hyperparameter settings, we will train models on all these combinations in the upcoming runs to determine which one delivers the best results. The following sub-sections explain the settings chosen for these hyperparameters.

5.3.1 Tuning Learning Rate

For the first run, the base learning rate was set to 0.001 and it was consecutively halved at 20,000 iterations and 40,000 iterations, with the maximum number of iterations set to 50,000. However, we reached the highest mAP values on the testing set between 16,000 and 26,000 iterations, and the model trained for 16,000 iterations performed best in the live demo. This clearly indicates that we reach the optimal region by 20,000 iterations when setting the base learning rate to 0.001. Also, the longer we train beyond 26,000 iterations, the higher the chances of overfitting. Therefore, we will be setting the multistep values to 10,000 and 20,000 iterations, and the maximum number of iterations to 30,000 for this base learning rate. Halving the learning rate prior to 16000 iterations should result in the model approaching closer to the minimum.

To see if we can improve convergence by starting with a higher learning rate and halving it after 20,000 iterations, we will also try increasing the base learning rate to 0.01 while setting the multistep values to 20,000 and 30,000 iterations, and the maximum number of iterations to 40,000. Unlike the first training run, we will be plotting the training loss and test accuracy against number of iterations with the help of a python script in the upcoming runs. Based on the results of these plots, we can change the stepvalues to further optimize the model.

5.3.2 Tuning Batch Size

As discussed in 4.2.3, the batch size was set to 12 to strike a good balance between training speed and optimal convergence. The first training run took a very long time to finish as it was performed on a CPU. The upcoming runs will be performed on GPU processors, which perform better when batch sizes are powers of 2 [15]. Therefore, we will be trying batch sizes of 4, 8, and 16 in the upcoming runs to see which delivers the best results. As explained earlier, we will not use batch sizes of 32 or more since we are using less than 1000 images in our dataset.

5.3.3 Using Momentum and Adam Optimizer

In 4.2.4, we justified the selection of RMSprop as our gradient descent optimizer. Although RMSprop dampens out oscillations that are directed away from a minimum, it does not accelerate the search that is directed towards a minimum. *Momentum* is a very effective technique that can be used to accomplish the latter. Momentum implements exponentially weighted averages in which the gradients of previous updates are weighed in during each updating step. This has the effect of making the updates less sensitive to points of high curvature in the loss function where the search could be directed significantly away from a minimum.

Adam, or Adaptive Moment Optimization, combines both Momentum and RMSprop to reap the benefits of both techniques. Adam does this by incorporating both algorithms in the updating step, thereby impeding the search in the direction of oscillations while accelerating it in the direction of minima. This enables us to use a higher base learning rate and a lower maximum number of iterations. Although this speeds up the training process, it comes at the risk of converging to a *sharp* minimum. If the minimum is sharp, the loss is very sensitive to changes in parameters near the minimum. Flatter minima tend to generalize better with unseen samples than sharp minima [17] and for this reason, we will be implementing Momentum, Adam, and RMSprop separately for each combination of hyperparameters in our upcoming runs to determine which optimizer delivers the best results.

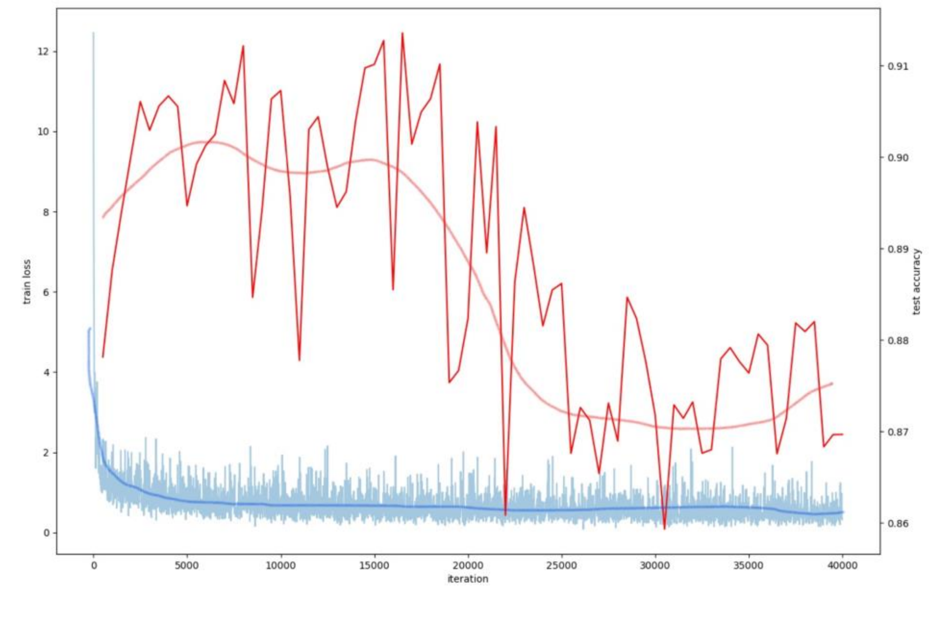
5.4 Results of Optimized Training Runs

The table below summarizes all the combinations of hyperparameter settings that have been selected. The terminal output for each of these training runs will be copied to a log file, which will then be used to plot graphs of training and testing loss versus number of iterations. The plots will tell us how each combination is performing. Ideally, we want the training loss to decrease to a plateau while test accuracies are above 0.75 mAP. An indication of poor performance is when the training loss does not decrease steadily until reaching a plateau. Based on the plots, we will shortlist the combinations that performed the best and deploy those models in live demos.

Table 5.2. Selection of Hyperparameter Settings

Combinations	Base Learning Rate	Stepvalue 1	Stepvalue 2	Maximum number of iterations	Batch Size	Optimizer
Combo 1	0.001	10,000	20,000	30,000	4	Momentum
Combo 2	0.001	10,000	20,000	30,000	8	Momentum
Combo 3	0.001	10,000	20,000	30,000	16	Momentum
Combo 4	0.01	20,000	30,000	40,000	4	Momentum
Combo 5	0.01	20,000	30,000	40,000	8	Momentum
Combo 6	0.01	20,000	30,000	40,000	16	Momentum
Combo 7	0.001	10,000	20,000	30,000	4	RMSprop
Combo 8	0.001	10,000	20,000	30,000	8	RMSprop
Combo 9	0.001	10,000	20,000	30,000	16	RMSprop
Combo 10	0.01	20,000	30,000	40,000	4	RMSprop
Combo 11	0.01	20,000	30,000	40,000	8	RMSprop
Combo 12	0.01	20,000	30,000	40,000	16	RMSprop
Combo 13	0.001	10,000	20,000	30,000	4	Adam
Combo 14	0.001	10,000	20,000	30,000	8	Adam
Combo 15	0.001	10,000	20,000	30,000	16	Adam
Combo 16	0.01	20,000	30,000	40,000	4	Adam
Combo 17	0.01	20,000	30,000	40,000	8	Adam
Combo 18	0.01	20,000	30,000	40,000	16	Adam

After completing all 18 training sessions, loss plots were generated from the logs of each session. Upon analyzing all the plots, combo 5 and combo 15 stood out as the most promising candidates. We can see in figure 5.3 that the average training loss in both plots steadily decrease until levelling out. The erratic spikes in the loss plots are due to noisy updates associated with using mini-batches, so we are only looking at the average loss represented by lines of best-fit. As discussed in 4.2.3, smaller mini-batch sizes result in noisier updates. Since the batch size of combo 5 is half of that of combo 15, the loss plot has larger and more erratic spikes.



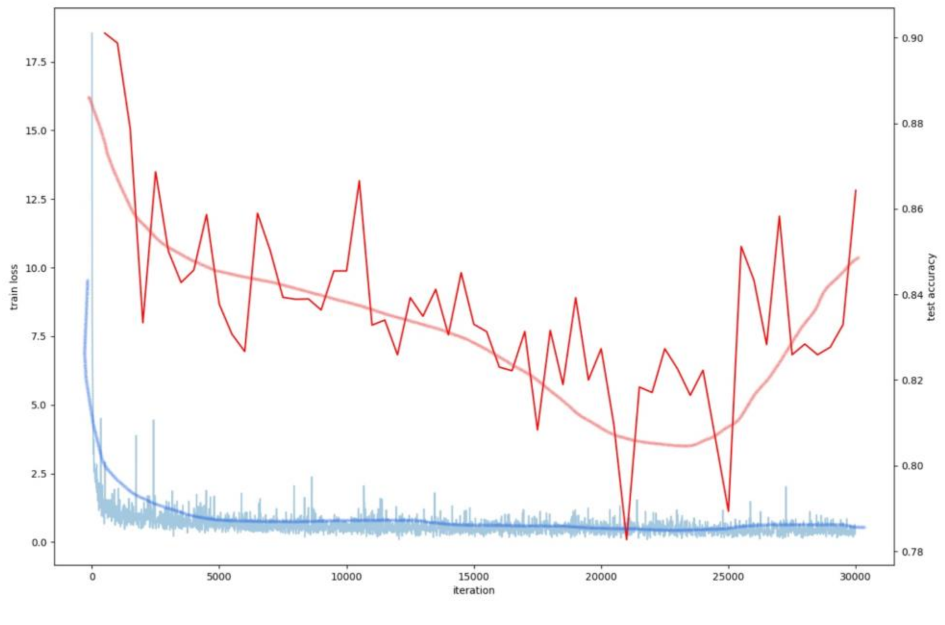


Figure 5.3. Top: Loss plot for Combo 5. Bottom: Loss plot for Combo 15. The spiky red lines are training loss, the spiky blue lines are test accuracy, and the smooth lines represent average loss/accuracy.

The links to the demos recorded for both combos have been attached in Appendix C2.

The training loss for combo 5 reached its lowest average level between 25000 and 35000 iterations while the training loss for combo 15 reached its lowest average level between 18000 and 25000 iterations before it began rising. The test accuracy of combo 5 remained steady at around 0.86 mAP while the test accuracy of combo 15 remained steady at around 0.78 mAP. When the training loss reaches a plateau, the risk of overfitting keeps increasing with training duration. Therefore, we will be deploying the caffemodels trained up till the point where training loss just enters the plateau and go from there. For combo 5, this is around 25000 iterations and for combo 15, this is around 18000 iterations.

Finally, both caffemodels were deployed in live demos. The screen-recorded demos are viewable via the links attached in Appendix C2. In both demos, we can see that the top detector heads were being correctly detected without any misclassifications. Thus, both caffemodels were successful in preventing misclassifications. However, the detections in combo 15 were more frequent and generally had higher confidence scores than those in combo 5. Furthermore, combo 15 performed better than combo 5 when detecting carts and people. Therefore, the hyperparameter settings in combo 15 serve as a better baseline than combo 5 for all future training.

6. STOPPING DETECTOR MOTION

Once an object is detected in the path of collision, detector motion must be stopped in real-time (less than or equal to 1 second). The only way to stop detector motion with a software-generated signal is by sending a stop command to the *SCONA* board, which is the central control board of the SPECT/CT camera. The host computer communicates with SCONA via *CPI* commands. CPI, which stands for 'Camera Primitives Interface', is the interface protocol which defines the format in which commands are processed by the camera control. The command for stopping camera motion, called System-Stop, is '0001mD'.

For a device to be registered as a host computer, it must be configured with the requisite proprietary software to communicate with SCONA, and this requires multiple security authorizations. Due to these security restrictions, it was not possible to configure a host computer with the software required to run the RTOD script, and any non-host device running this script cannot send CPI commands to SCONA. However, it is possible to send the command as a TCP packet to the *Gateway Processor* within SCONA, shown in figure 6.1.

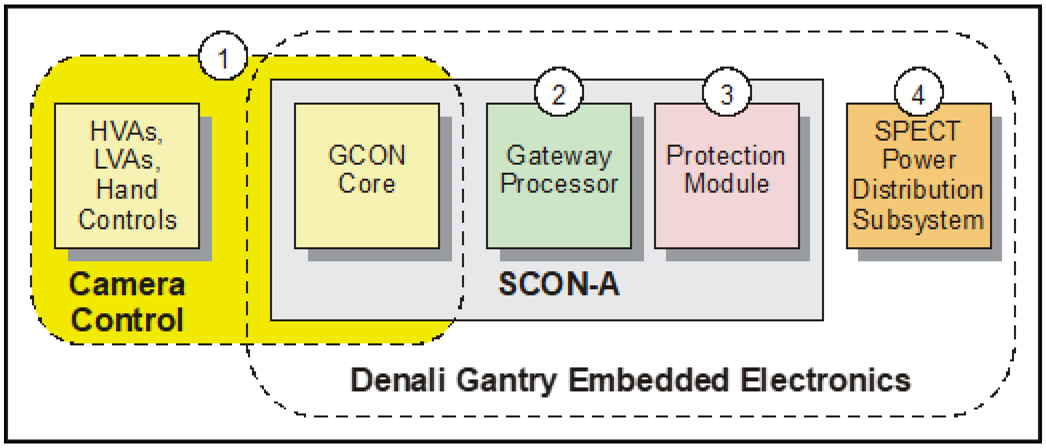

1
HVAs, LVAs, Hand Controls
GCON Core
2
Gateway Processor
3
Protection Module
4
SPECT Power Distribution Subsystem
Camera Control
SCON-A
Denali Gantry Embedded Electronics

Figure 6.1. Architecture of SCONA and Camera Control.

The GCON Core in SCONA is responsible for processing all the commands being sent to camera control. As indicated by circle 1 in figure 6.1, it is the only module within SCONA that can send commands to the camera control. Camera control includes all the amplifiers and motors that drive camera motion. Therefore, the stop command sent to the Gateway as a TCP packet must be extracted and sent to the GCON Core. At the time of undertaking this project, there was no way to send CPI commands from the Gateway to the GCON Core, so this had to be set aside as a future assignment. However, a Python script was written for sending the stop command as a TCP message to the Gateway [A3]. The script is invoked as a subprocess within the if condition in the RTOD script that is set to true when an external object is detected. The host IP address and port number of the Gateway are passed into the “tcp_send_command.py script” along with the “system stop” CPI command as three separate arguments. The “execute_tcp_cmd” function within the script then opens a TCP socket for sending the stop command to the Gateway.

A potential workaround for bypassing the SCONA to send a stop command is setting up an external circuit that toggles the stop button located on the machine. The script would trigger an electric signal within the external circuit when an external object is detected, which would then toggle the stop button. If the script can trigger the electric signal in one second or less, this method would meet real-time requirements. During the undertaking of the project, it was not possible to work on this method due to logistical and temporal restrictions.

7. CONCLUSION AND FUTURE WORK

This thesis paper laid out a comprehensive end-to-to end deep learning method for real-time object detection to prevent collisions in a real-world scenario. The experiments carried out in this project successfully implemented the methodologies described in the paper to detect external objects that are in the path of motion of the SPECT-CT detectors in real-time. Consequently, this project laid the groundwork for expanding on these successful methodologies to include all potential objects that could be found in the path of the detector’s motion. However, three issues must be resolved before the collision prevention system can be deployed in production.

7.1 Training Models to Detect More Objects and Detectors

The methodologies outlined in this project can be applied to collect and process images of several indoor objects. For example, using publicly available datasets and pretrained models can be utilized for several common objects as it was for the person class, while the collection and annotation techniques used for the cart and detector heads can be applied to other custom objects. It is important to reiterate that false positives on external objects are desirable for our use case. Training models on a wide range of common indoor objects would allow false positives for uncommon objects that may be found in scanning rooms or production floors.

However, it is imperative to prevent false positives on the detector heads and to do so, models must be trained to detect all types of detector heads that come with different models of SPECT-CT cameras. For each model, using more images per configuration than the amount used in this project will also improve the chances of robust detections while reducing the likelihood of misclassifications. Models must also be trained to detect any other unique objects that are found in scanning rooms and productions floors, such as the collimator cart. Once a list of all potential objects is finalized, the next step is to collect images and prepare the dataset for MobileNet-SSD training, as demonstrated in this research. During the training phase, many of the model optimization methods outlined in this paper can be utilized to improve the accuracy of detections.

7.2 Efficiently Stopping Detector Motion

As discussed in section 6, detector motion can be halted either by sending a system stop command through SCONA or by setting up an external circuit that can toggle the stop button. Both these methods must be tested in order to determine which triggers a faster stop. Although a script was written to send the command through the Gateway, it may introduce an additional lag that fails real-time requirements. To avoid this lag, the RTOD application can be set up within a host computer. However, configuring all the necessary third-party software in a host computer can be very difficult due to security restrictions. Setting up an external circuit may be the fastest method and one that does not entail multiple security steps to install and configure the requisite third-party software. Therefore, the external method should be explored first.

7.3 Controlling the Activation of the RTOD script

There may be some scenarios where we do not want to stop detector motion whenever an external object is detected, such as when a patient is on the bed or standing by the detectors during a scan. In such scenarios, the application should not label the patient as an external object. To account for these scenarios, we must add more conditions to the code based on the position of the patient and the type of scan. Otherwise, we could simply disable the script during such scans if no collision hazards are expected.

APPENDIX A. CAFFE MOBILENET SSD RAW FILES

A.1 Caffe Prototxt Files

Caffe prototxt files are text files that hold information about the structure of the neural network and processing of input data:

- Resizing, scaling, and normalization of input.
- Batch size, batch sampling, and data format.
- The list of layers in the neural network.
- The parameters of each layer, such as its name, type, input and output dimensions.
- The connections between layers.

Network structure and parameters can be changed by adding or removing layers and modifying parameters inside layers. The table below sequentially shows contents of the prototxt file generated for the training model used in the first run, starting with the data layer later up till the activation layer of the first convolution, “conv0”.

Table A.1. Data layer and Conv0 layers in MobileNetSSD_train.prototxt

Data Transformation	Batch Processing	Conv0 Layers
name: "MobileNet-SSD"	data_param {	layer {
layer {	source: "trainval_lmdb/"	name: "conv0"
name: "data"	batch_size: 12	type: "Convolution"
type: "AnnotatedData"	backend: LMDB	bottom: "data"
top: "data"	}	top: "conv0"
top: "label"	annotated_data_param {	param {
include {	batch_sampler {	lr_mult: 0.1
phase: TRAIN	max_sample: 1	decay_mult: 0.1
}	max_trials: 1	}
transform_param {	}	convolution_param {
scale: 0.007843	batch_sampler {	num_output: 32
mirror: true	sampler {	bias_term: false
mean_value: 127.5	min_scale: 0.3	pad: 1
mean_value: 127.5	max_scale: 1.0	kernel_size: 3
mean_value: 127.5	min_aspect_ratio: 0.5	stride: 2
resize_param {	max_aspect_ratio: 2.0	weight_filler {
prob: 1.0	}	type: "msra"
resize_mode: WARP	sample_constraint {	}
height: 300	min_jaccard_overlap: 0.1	}
width: 300	}	}
interp_mode: LINEAR	max_sample: 1	layer {
interp_mode: AREA	max_trials: 50	name: "conv0/bn"
interp_mode: NEAREST	}	type: "BatchNorm"

```
      interp_mode: CUBIC
      interp_mode: LANCZOS4
    }
    emit_constraint {
      emit_type: CENTER
    }
    distort_param {
      brightness_prob: 0.5
      brightness_delta: 32.0
      contrast_prob: 0.5
      contrast_lower: 0.5
      contrast_upper: 1.5
      hue_prob: 0.5
      hue_delta: 18.0
      saturation_prob: 0.5
      saturation_lower: 0.5
      saturation_upper: 1.5
      random_order_prob: 0.0
    }
    expand_param {
      prob: 0.5
      max_expand_ratio: 4.0
    }
  }
    batch_sampler {
      sampler {
        min_scale: 0.3
        max_scale: 1.0
        min_aspect_ratio: 0.5
        max_aspect_ratio: 2.0
      }
      sample_constraint {
        min_jaccard_overlap: 0.3
      }
      max_sample: 1
      max_trials: 50
    }
    batch_sampler {
      sampler {
        min_scale: 0.3
        max_scale: 1.0
        min_aspect_ratio: 0.5
        max_aspect_ratio: 2.0
      }
      sample_constraint {
        min_jaccard_overlap: 0.5
      }
      max_sample: 1
      max_trials: 50
    }
    batch_sampler {
      sampler {
        min_scale: 0.3
        max_scale: 1.0
        min_aspect_ratio: 0.5
        max_aspect_ratio: 2.0
      }
      sample_constraint {
        min_jaccard_overlap: 0.7
      }
      max_sample: 1
      max_trials: 50
    }
    batch_sampler {
      sampler {
        min_scale: 0.3
        max_scale: 1.0
        min_aspect_ratio: 0.5
        max_aspect_ratio: 2.0
      }
      sample_constraint {
        min_jaccard_overlap: 0.9
      }
      max_sample: 1
      max_trials: 50
    }
    batch_sampler {
      sampler {
        min_scale: 0.3
        max_scale: 1.0
  bottom: "conv0"
  top: "conv0"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
}
layer {
  name: "conv0/scale"
  type: "Scale"
  bottom: "conv0"
  top: "conv0"
  param {
    lr_mult: 0.1
    decay_mult: 0.0
  }
  param {
    lr_mult: 0.2
    decay_mult: 0.0
  }
  scale_param {
    filler {
      value: 1
    }
    bias_term: true
    bias_filler {
      value: 0
    }
  }
}
layer {
  name: "conv0/relu"
  type: "ReLU"
  bottom: "conv0"
  top: "conv0"
}
layer {
  name: "conv1/dw"
  type: "Convolution"
  bottom: "conv0"
  top: "conv1/dw"
  param {
    lr_mult: 0.1
    decay_mult: 0.1
  }
  convolution_param {
    num_output: 32
    bias_term: false
```

	min_aspect_ratio: 0.5
	max_aspect_ratio: 2.0
	}
	sample_constraint {
	max_jaccard_overlap: 1.0
	}
	max_sample: 1
	max_trials: 50
	}
	label_map_file: "labelmap.prototxt"
	}
	}
		pad: 1
		kernel_size: 3
		group: 32
		engine: CAFFE
		weight_filler {
		type: "msra"
		}
		}
		}

A.2 Caffe Solver Files

Solver files hold information of hyperparameter settings used for training and testing. Table A.2 shows the solver files for training and testing used in the initial run.

Table A.2. Solver_train and Solver_test file contents for Initial Run

Solver_train	Solver_test
train_net: "example/MobileNetSSD_train.prototxt"	train_net: "example/MobileNetSSD_train.prototxt"
test_net: "example/MobileNetSSD_test.prototxt"	test_net: "example/MobileNetSSD_test.prototxt"
test_iter: 673	test_iter: 22
test_interval: 10000	test_interval: 1000
base_lr: 0.001	base_lr: 0.001
display: 10	display: 10
max_iter: 120000	max_iter: 0
lr_policy: "multistep"	lr_policy: "multistep"
gamma: 0.5	gamma: 0.5
weight_decay: 0.00005	weight_decay: 0.00005
snapshot: 1000	snapshot: 1000
snapshot_prefix: "snapshot/mobilenet"	snapshot_prefix: "snapshot/mobilenet"
solver_mode: GPU	solver_mode: GPU
debug_info: false	debug_info: false
snapshot_after_train: true	snapshot_after_train: false
test_initialization: false	test_initialization: true
average_loss: 10	average_loss: 10
stepvalue: 20000	stepvalue: 10000
stepvalue: 40000	stepvalue: 30000
iter_size: 1	iter_size: 2
type: "RMSProp"	type: "RMSProp"
eval_type: "detection"	eval_type: "detection"
ap_version: "11point"	ap_version: "11point"

APPENDIX B. CODE FILES

B.1 RTOD Script

```python
# import the necessary packages
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-p", "--prototxt", required=True,
   help="path to Caffe 'deploy' prototxt file")
ap.add_argument("-m", "--model", required=True,
   help="path to Caffe pre-trained model")
ap.add_argument("-c", "--confidence", type=float, default=0.3,
   help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# initialize the list of class labels MobileNet SSD was trained to
# detect, then generate a set of bounding box colors for each class
CLASSES = ["background", "person", "detector", "cart"]
COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))

# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])

# initialize the video stream, allow the camera sensor to warmup,
# and initialize the FPS counter
print("[INFO] starting video stream...")
vs = VideoStream(src=4).start()
time.sleep(2.0)
fps = FPS().start()

# loop over the frames from the video stream
while True:
   # grab the frame from the threaded video stream and resize it
   # to have a maximum width of 400 pixels
   frame = vs.read()
   frame = imutils.resize(frame, width=1600)

   # grab the frame dimensions and convert it to a blob
   (h, w) = frame.shape[:2]
   blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
      0.007843, (300, 300), 127.5)

   # pass the blob through the network and obtain the detections and
   # predictions
   net.setInput(blob)
```

```
    detections = net.forward()

    # loop over the detections
    for i in np.arange(0, detections.shape[2]):
        # extract the confidence (i.e., probability) associated with
        # the prediction
        confidence = detections[0, 0, i, 2]

        # filter out weak detections by ensuring the `confidence` is
        # greater than the minimum confidence
        if confidence > args["confidence"]:
            # extract the index of the class label from the
            # `detections`, then compute the (x, y)-coordinates of
            # the bounding box for the object
            idx = int(detections[0, 0, i, 1])
            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
            (startX, startY, endX, endY) = box.astype("int")

            # draw the prediction on the frame
            label = "{}: {:.2f}%".format(CLASSES[idx],
                confidence * 100)
            cv2.rectangle(frame, (startX, startY), (endX, endY),
                COLORS[idx], 2)
            y = startY - 15 if startY - 15 > 15 else startY + 15
            cv2.putText(frame, label, (startX, y),
                cv2.FONT_HERSHEY_SIMPLEX, 1.5, COLORS[idx], 2)
            if CLASSES[idx] != "detector":
                #msg = "0001mD"
                #host = "192.168.1.1"
                #port = 55
                #response = execute_tcp_cmd(msg, host, port)
                print(CLASSES[idx]+" found in potential collision path, sending
e-stop")

    # show the output frame
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF

    # if the `q` key was pressed, break from the loop
    if key == ord("q"):
        break

    # update the FPS counter
    fps.update()

# stop the timer and display FPS information
fps.stop()
print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```

B.2 Augmentation Script

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img,
img_to_array, load_img
import os

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

imgs = os.listdir('/home/tahrir/Desktop/thesis/Augmented_detector_pics')

for pic in imgs:
    #print(pic)
    img =
load_img('/home/tahrir/Desktop/thesis/Augmented_detector_pics/'+pic)
    x = img_to_array(img)
    x = x.reshape((1,) + x.shape)

    i = 0
    for batch in datagen.flow(x, batch_size=1, save_to_dir='preview',
save_prefix='aug', save_format='jpg'):

        i += 1
        if i > 1:
            break
```

B.3 tcp_send_command.py Script

```
#!/usr/bin/env python3

import subprocess
import sys
import os
import time
import struct
import socket
import select
import datetime


#----------------------------------------------------
#   constants

#defaults
HOST_DEFAULT                 = "192.168.0.1"
PORT_DEFAULT               = 2050
TIMEOUT_SECS = 15


#----------------------------------------------------
```

```
#  misc globals
socket_is_open = False   #main command socket

# poll for incoming data ready on a socket using select()
def is_data_ready( sock, timeout=0.0 ):
   rlist = [sock]
   wlist = []
   xlist = []
   (rr,ww,xx) = select.select( rlist, wlist, xlist, timeout )
   for s in rr:
      if s is sock:
         return True
   return False

def wait_for_incoming_data( seconds, sock, verbosity=False,
sleep_between_polls=True ):
   if verbosity:
      print( "waiting on socket:", sock )
   data = b''
   timestart   = datetime.datetime.now()
   done= False;
   while not done:
      #print( "polling..")
      if is_data_ready( sock ):
         #data = sock_cmd.recv( 2048 )
         data = sock.recv( 2048 )
         str_recvd = data.decode( "utf-8")
         if( verbosity ):
            print( "recvd %d bytes: '%s'" % ( len(data), str_recvd ) )
         done = True;
      #print( "done polling")

      time_now = datetime.datetime.now();
      dt = time_now - timestart
      if( dt.total_seconds()  >= seconds ):
         string_recvd = '!! timed out !!'
         print( "timed out waiting for response")
         done=True
      if sleep_between_polls:
         time.sleep( 0.01 )
   return( data )

#  executes, returns response in binary form
#  msg is a string
#  verbosity values:
#     0: silent
#     1: print only errors
#     2: normal
#     3: lots of stuff  (i.e.debug)
def execute_tcp_cmd(   msg, host, port, verbosity=2,
close_socket_when_done=True  ):
   global sock_cmd, socket_is_open
   ip_descr = host


   if verbosity>=2:
      print( "Executing dpi cmd: '%s' to %s:%d ..." % (msg, HOST, PORT )  )
   if not socket_is_open:
      sock_cmd = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
      server_addr_port = ( ip_descr, port )
      try:
         sock_cmd.connect( server_addr_port )
      except Exception as e:
         print( "connect to %s:%d failed" %  (ip_descr, port) )
         print(e)
         exit()
   if verbosity>=3:
      print( "connection established to %s:%d" % (ip_descr, port) )
   socket_is_open = True

   if verbosity>=3:
      print( "sending '%s'" % msg )

   bytes_to_send = msg.encode( "utf-8")
   bytes_to_send +=  b'\00'

   sock_cmd.sendall( bytes_to_send )
   if verbosity>=3:
      print( "sent '%s'" % msg )

   bin_recvd = wait_for_incoming_data( TIMEOUT_SECS, sock_cmd )

   if close_socket_when_done:
      if verbosity>=3:
         print( "closing socket")
      sock_cmd.close()
      socket_is_open = False

   if verbosity>=3:
      string_recvd = bin_recvd.decode( "utf-8" )
      print( "Received: %d bytes: '%s'" %  (len(bin_recvd), string_recvd) )
   return( bin_recvd )


def usage_bail():
   print( "usage: tcp_client_example.py server[:port] cmd [cmd_args]" )
   exit()

def main( argv ):
   #global HOST, PORT

   if len(argv)<3:
      usage_bail()

   host_port = argv[1]

   if ':' in host_port:
      try:
         HOST,port_str = host_port.split( ':' )
         PORT = int(port_str)
      except:
         print( "invalid host[:port] : '%s'" % host_port)
         usage_bail()
   else:
      HOST=host_port
      PORT=PORT_DEFAULT

   cmd  = ' '.join( argv[2:] )

   response = execute_tcp_cmd(  cmd, HOST, PORT )
   print( "command returned:", response )

#------------------------------
if __name__ == "__main__":
   main( sys.argv )
   exit()
```

APPENDIX C: LINKS TO DEMOS

C.1 Link to Demos after Initial Training Run

1) Link to the Demo of only detectors with no object entering the collision regions:
   16k_30_Detector
2) Link to the Demo of cart entering collision region:
   16k_30_Cart
3) Link to the Demo of person entering collision region:
   16k_30_Person

C.2 Links to Combo 5 and Combo 15 Demos

1) Demo of combo 5 trained for 18000 iterations:
   18k_Combo5
2) Demo of combo 15 trained for 25000 iterations:
   25k_Combo15

REFERENCES

[1] Elgendy, Mohamed. “Grokking deep learning for computer vision.” *Manning*, Manning Publications, 2019, livebook.manning.com/book/grokking-deep-learning-for-computer-vision/chapter-7/v-8/7.

[2] Rosebrock, Adrian. “A gentle guide to deep learning object detection.” *Pyimagesearch*, 14 May 2018, pyimagesearch.com/2018/05/14/a-gentle-guide-to-deep-learning-object-detection/.

[3] Shaoqing Ren, et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." (2016).

[4] Joseph Redmon, et al. "You Only Look Once: Unified, Real-Time Object Detection." (2016).

[5] Liu, Wei et al. "SSD: Single Shot MultiBox Detector". *Lecture Notes in Computer Science*. (2016): 21–37.

[6] Andrew G. Howard, et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." (2017).

[7] Everingham, M. et al. "The Pascal Visual Object Classes (VOC) Challenge". *International Journal of Computer Vision* 88. 2(2010): 303–338

[8] Olivas, Emilio Soria et al. *Handbook Of Research On Machine Learning Applications and Trends: Algorithms, Methods and Techniques - 2 Volumes*.. Information Science Reference - Imprint of: IGI Publishing, 2009.

[9] Jia, Yangqing et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". arXiv preprint arXiv:1408.5093. (2014).

[10] Chuanqi305. 2017. MobileNet-SSD. github.com/chuanqi305/MobileNet-SSD.

[11] Little, Z^2. “Gradient Descent: Stochastic vs. Mini-batch vs. AdaGrad vs. RMSProp vs. Adam.” *Medium*, 25 July 2020, xzz201920.medium.com/gradient-descent-stochastic-vs-mini-batch-vs-batch-vs-adagrad-vs-rmsprop-vs-adam-3aa652318b0d.

[12] Yoshua Bengio, . "Practical recommendations for gradient-based training of deep architectures." (2012).

[13] Rosebrock, Adrian. “Real-time object detection with deep learning and opencv.” *Pyimagesearch*, 18 September 2017, pyimagesearch.com/2017/09/18/real-time-object-detection-with-deep-learning-and-opencv.

[14] Ng, Andrew. Lecture 10.5 – “Regularization and bias/variance.” *Machine Learning* offered by Stanford University on Coursera.

[15] Intel AI Developer Program. “Cifar-10 Classification using Intel ® Optimization for TensorFlow.” *Intel*, 13 December 2017, software.intel.com/content/www/us/en/develop/articles/cifar-10-classification-using-intel-optimization-for-tensorflow.html.

[16] Tzutalin. 2017. LabelImg. github.com/tzutalin/labelImg.

[17] Nitish Shirish Keskar, et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." (2017).