

October 2020

Edge Computing for Deep Learning-Based Distributed Real-time Object Detection on IoT Constrained Platforms at Low Frame Rate

Lakshmikavya Kalyanam
University of South Florida

Follow this and additional works at: <https://digitalcommons.usf.edu/etd>

 Part of the [Computer Engineering Commons](#)

Scholar Commons Citation

Kalyanam, Lakshmikavya, "Edge Computing for Deep Learning-Based Distributed Real-time Object Detection on IoT Constrained Platforms at Low Frame Rate" (2020). *USF Tampa Graduate Theses and Dissertations*.
<https://digitalcommons.usf.edu/etd/9541>

This Thesis is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact scholarcommons@usf.edu.

Edge Computing for Deep Learning-Based Distributed Real-time Object Detection on IoT

Constrained Platforms at Low Frame Rate

by

Lakshmikavya Kalyanam

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Srinivas Katkoori, Ph.D.
Co-Major Professor: Hao Zheng, Ph.D.
Mehran Mozaffari Kermani, Ph.D.

Date of Approval:
June 25, 2020

Keywords: Distributed System, CNN, YOLO, PYNQ, FPGA

Copyright © 2020, Lakshmikavya Kalyanam

Dedication

I dedicate this work to my family for their unconditional love and patience.

Acknowledgments

I would first like to formally acknowledge Dr. Srinivas Katkoori and Dr. Hao Zheng for providing the opportunity to work on this project. For bringing me success with their constant help and guidance, I am forever grateful to them. I would like to thank Dr. Mehran Mozaffari Kermani for serving on my thesis committee.

Finally, I must express my profound gratitude to my family and friends for providing me with unfailing support and continuous encouragement through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Table of Contents

List of Tables	iii
List of Figures.....	iv
Abstract	v
Chapter 1: Introduction and Motivation.....	1
1.1 Distributed System	2
1.2 Proposed Approach	3
1.3 Experimental Results	4
1.4 Thesis Organization.....	4
Chapter 2: Background and Related Work	5
2.1 IoT and Edge Computing.....	5
2.2 Edge Computing.....	7
2.3 Object Detection.....	8
2.4 CNN (Convolutional Neural Network).....	9
2.4.1 Structure of CNN.....	10
2.4.1.1 Convolution Layer.....	10
2.4.1.2 Pooling or Sub-sampling Layer	11
2.4.1.3 Fully Connected Layer	11
2.5 YOLO (You Only Look Once)	12
2.5.1 Tinier YOLO	13
2.6 Quantized Neural Networks	14
2.7 Distributed Object Detection	16
2.8 Object Detection for Hardware Platforms	18
2.8.1 FPGA (Field Programmable Gate Array)	18
2.8.2 GPU (Graphics Processing Unit)	20
2.9 PYNQ (Python Productivity for ZYNQ)	22
2.10 Load Balancing.....	24
2.10.1 HAProxy (High Availability Proxy)	25
2.11 Flask	25
2.12 Sockets	26
2.13 Related Work	27
2.14 Chapter Summary	31
Chapter 3: Proposed Work	33
3.1 Initial Work	35
3.1.1 Head Node	35
3.1.2 Worker Node	36
3.1.3 Loadbalancing by HAProxy.....	36
3.1.4 Dataflow	36
3.1.5 Scaling the Framework for Multiple Worker Nodes	37
3.1.6 Limitations of this Framework	37

3.2	Model 2.....	38
3.2.1	Head Node.....	39
3.2.2	Worker Node.....	40
3.2.3	Communication between Head and Worker Nodes.....	40
3.3	Chapter Summary.....	42
Chapter 4:	Experimental Results.....	43
4.1	Performance Analysis of YOLO.....	44
4.1.1	Tiny YOLO v3.....	44
4.1.2	Tinier YOLO.....	45
4.2	Evaluating BNN (Binarized Neural Network).....	47
4.3	Evaluating the Performance of Framework.....	47
4.3.1	Model 1.....	48
4.3.2	Model 2.....	48
4.3.3	Framework Overhead.....	49
4.4	Chapter Summary.....	52
Chapter 5:	Conclusions and Future Work.....	53
5.1	Future Enhancements.....	53
	References.....	55
	Appendix A: Copyright Permissions.....	62

List of Tables

Table 2.1	Comparison of CNN implementations.....	11
Table 4.1	PYNQ-Z1 SoC specifications [1].	43
Table 4.2	PYNQ-Z1 SoC memory specifications [1].....	43
Table 4.3	PYNQ-Z1 SoC USB and ethernet specifications [1].....	43
Table 4.4	PYNQ-Z1 SoC audio and video specifications [1].....	44
Table 4.5	PYNQ-Z1 SoC peripherals [1].....	44
Table 4.6	Analysis of Model 1 (in seconds)	48
Table 4.7	Analysis of Model 2 (in seconds)	49
Table 4.8	Analysis of framework in Model 2 (in seconds).....	50
Table 4.9	Analysis of the distributed framework for multiple Worker nodes to run inference on one input image (in seconds).....	50
Table 4.10	Analysis of the Distributed framework for multiple Worker nodes to run inference on one input image (in frames per second).....	50

List of Figures

Figure 2.1	Edge computing architecture.....	7
Figure 2.2	An overview of LeNet-5 architecture.	10
Figure 2.3	Tiny-YOLO vs YOLO.....	14
Figure 2.4	Xilinx Zynq-7000 SoC [2].	24
Figure 2.5	A basic socket overview.....	26
Figure 2.6	Edge computing-based object detection architecture.	28
Figure 2.7	System overview showing the data-flow.	29
Figure 2.8	Overview of architecture.	30
Figure 2.9	Comparison of image processing response time.....	31
Figure 3.1	An overview of our architecture.....	33
Figure 3.2	Block diagram of the framework.	34
Figure 3.3	Overview of Model 1 architecture.	38
Figure 3.4	Timing diagram that shows the communication between Head node and one Worker node.	41
Figure 3.5	Overview of Model 2 architecture.	42
Figure 4.1	Screenshot of Tiny YOLO v3 runtime execution.....	45
Figure 4.2	Tinier YOLO layers.	46
Figure 4.3	Tinier YOLO demo input and output image [3].	47
Figure 4.4	Performance analysis of Tinier YOLO [3].	47
Figure 4.5	Number of nodes vs. average inference time.....	51

Abstract

In the era of IoT (Internet of Things) and edge computing, there is a rising need for real-time applications in the domain of computer vision. The increase in hardware computing capabilities gave rise to applications of neural networks in various fields. Implementing IoT with neural networks in domains such as image and video recognition has shown promising performance when deployed in complex environments. There is an emerging demand for applications that require data computation in real-time with low latency. In an effort to address these issues, while keeping in mind the computing capabilities of IoT devices, we seek to develop a framework for efficient object detection on a distributed constrained platform system. In this thesis, a scalable and adaptable network for fast and easy convolutional neural network prototyping on the Xilinx PYNQ cluster has been proposed.

We employed PYNQ Z1 AP-SoC (All Programmable System-on-Chip) as the IoT edge node platform and integrated state-of-the-art algorithms for object detection. The distributed architecture is robust and exploits the heterogeneous computing capability of the PYNQ platform. The proposed work is on a wireless distributed network with minimal communication latency. We demonstrate the framework for low frame rate applications where the scenery is not changing rapidly. We were able to achieve 19.23 frames per second with three IoT nodes with Binarized Neural Network (BNN) image classification algorithm. The frames per second rate is directly proportional to the number of nodes in the network. The communication latency can be offset by the scalability offered by the distributed framework.

Chapter 1: Introduction and Motivation

Today's world of high-speed internet is ever-changing and ever-evolving. While it helps us stay connected and updated, it has also bled into every corner of our lives to include physical devices and everyday objects. This idea has coalesced into a huge network of connected devices, things, people, and even objects that collect and process the acquired data to share it over the internet's common platform to be called the Internet of Things (IoT). They have sensors and actuators which continually record information to acquire data. Depending on the requirement, these sensors and actuators can measure and record different kinds of raw data such as temperature, movement, or process this information to make meaningful and intelligent decisions or transmit it over the internet. This data can be used as input for various applications or services. IoT has moved into every corner of our lives from smart homes to wearable technology such that most of the devices can be controlled, and in turn, these devices can control our environment. When in the context of communication, it traditionally meant between humans, now it has expanded to human to machine and machine to machine communication. Machine to machine communication is based on the premise that a networked machine is more valuable than an isolated one. When multiple machines are interconnected, more autonomous and intelligent applications can be deployed. The total installed base of the Internet of Things connected devices is projected to amount to 75.44 billion worldwide by 2025, a five-fold increase in ten years, according to a report published by Statista Research in 2016 [4]. This form of communication is already being applied in various sectors such as transportation, medicine, smart grid, and manufacturing, with enormous growth

potential. The multitude of devices and sensors that communicate over the network work together to form a distributed system to provide the user a seamless experience.

1.1 Distributed System

A distributed system can be described as a collection of multiple components on various platforms that communicate and coordinate to work as a single coherent system. These components are computing elements, each of which can be referred to as a node. These nodes could be software or hardware processes, and are generally geographically dispersed. The nodes are autonomous computing elements connected to a network and have some local memory. The general characteristics of a distributed system are:

- All the components can work concurrently but may not have the same computing capabilities. They share resources over a network. They communicate and coordinate to complete a task by distributing the workload amongst them.
- They do not share a global clock, which brings in challenges with synchronization and coordination within the system. This also allows the system to be free of geographical constraints.
- A single component failure typically does not lead to the overall system failure. In general, the distributed system has high fault tolerance.

Based on architecture of a distributed system, we can classify it into the following models: peer-to-peer, client-server, three-tier, and n-tier.

Some of the benefits of a distributed system are:

- *Transparency*: Although the system can be distributed geographically, a coherent view of the system is presented to the user.
- *Openness*: The network makes a distributed system easy to modify and configure.
- *Reliability*: The system has high reliability as single node failure does not result in the system failure.
- *Scalability*: The computation on each node is independent of the other nodes, which make scaling the system relatively easy. The system is flexible and adaptable as a collection of small and replaceable nodes.

When more than one component meets the job requirements, depending on availability and responsiveness, the work load needs to be balanced. A load balancer is generally used to keep track of the resources while distributing the workload.

1.2 Proposed Approach

We propose a framework for distributed object detection on constrained platforms at a low frame rate. We have used the PYNQ board as the constrained platform and developed a distributed system for object detection with deep neural networks. We proposed two models for the object detection problem. The first model uses a third party static load balancer to distribute the load over multiple PYNQ boards. The second model is a client-server model based distributed system. The system leverages the software capabilities of the PYNQ board by using Python and the hardware capabilities of the dual-core ARM processor.

1.3 Experimental Results

We implemented the proposed framework for distributed object detection on the PYNQ Z1 APSoc (All Programmable System-On-Chip). For the first model, we implemented two state-of-the-art object detection algorithms, namely, Tiny YOLO (You Only Look Once) v3 and Tinier YOLO. For these algorithms, with a single node, we achieved a frame rate of 0.029 fps (frames per second). The achieved frame rate is very low due to serial processing nature of the model. To improve frame rate, we implemented a second model which is distributed in nature. Besides the above two algorithms, we also implemented BNN (Binarized Neural Network) algorithm. The achieved frame rates with three nodes are 0.12 fps (Tiny YOLO v3), 1.19 fps (Tinier YOLO), and 19.23 fps (BNN). The achieved frame rates are better compared to that of the first model, due to the distributed nature of the model. Further, we implemented our own task scheduler. The measured communication latency is negligible compared to the computation latency on each node, thus the frame rate can be scaled linearly with the number of nodes.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 presents an overview of edge computing, image detection on low power platforms, YOLO algorithm, object detection on FPGAs, and communication sockets. It also reviews existing architectures for real-time distributed object detection with deep neural networks on constrained platforms. Chapter 3 presents in detail two models for the proposed framework. Chapter 4 describes the experimental setup and reports the experimental results. Chapter 5 draws the conclusions and outlines directions for further enhancement.

Chapter 2: Background and Related Work

In this chapter, we will present background on Internet-of-Things (IoT), edge computing, object detection, Convolutional Neural Networks (CNNs), YOLO, distributed object detection. We also review the related work on real-time object detection primarily on constrained platforms.

2.1 IoT and Edge Computing

The evolution of IoT can be traced back to the invention of MOSFET, which serves as the building block of most modern electronic devices and the internet itself. IoT essentially is the new way of life. Machine-to-Machine communication and Human to Machine communication to transfer data over a network is the backbone of IoT. The idea is to create an interrelated system of computing devices.

IoT has expanded and evolved into the current state as various technologies, embedded systems, sensors, data analytics, and artificial intelligence have made progress in leaps and bounds. The communication between these different components is to exchange information for a wide range of applications in the cloud. The cloud in an IoT system generally manages the event queuing and messaging system that occurs between different layers. The number of IoT devices increased 31 percent year-over-year to 8.4 billion in the year 2017, and it is estimated that there will be 30 billion devices by 2020. The global market value of IoT is projected to reach 7.1 trillion by 2020, as predicted by [5]. This describes the scope of IoT devices that collect large amounts of data and exchange it over complex networks, which is conventionally sent to centralized servers. The results

of computations are sent back to the IoT devices. As more and more devices end up connected to the internet via IoT, we observe the data that needs to be also processed exponentially, increasing, which ends up pushing network resources and bandwidth to its limit. Improving the network and data centers to cater to data traffic generated by the devices is one avenue, but issues arise when applications are real-time. The clusters of clouds cause latency, which cannot be accepted as they could impact safety and emergency responses in some cases. While cloud computing has addressed the issues of the sheer volume of data with big data and computing it with deep learning, time sensitivity, and context awareness as issues where short response time is impertinent cannot be addressed with conventional cloud computing. Furthermore, to extend the lifetime of most IoT devices, it is necessary to balance power consumption by scheduling computation to devices that have higher power and computational capabilities [5].

To reduce transmission time, we could process data in the computation nodes closest to the user. The speed of data transfer depends on network traffic, and if heavy, it leads to high latency and increases power consumption cost. This makes scheduling and allocation of resources for processing a critical issue. Now, this bottleneck needs to be addressed for current networks with edge computing, which moves the processing to the edge of the network. Edge nodes respond to demands with edge computing servers, which are located closer to the user compared to the cloud server. A local edge node means less computational capability, but it can still provide better QoS (Quality of Service), and these can be integrated into the network itself [6].

2.2 Edge Computing

Gartner, Inc.[7] defines edge computing as “A part of a distributed computing topology in which information processing is located close to the edge – where things and people produce or consume that information.” Edge computing aims to bring computations and data close to IoT devices and their infrastructures rather than moving it to the cloud. Edge computing helps reduce the cost of connectivity by communicating data that is preprocessed instead of raw streams of data collected by sensors. Generally speaking, the structure of edge computing can be divided into three aspects, the front layer or Edge layer, near-end or Fog layer, and far layer or Cloud layer [6], as shown in Figure 2.1.

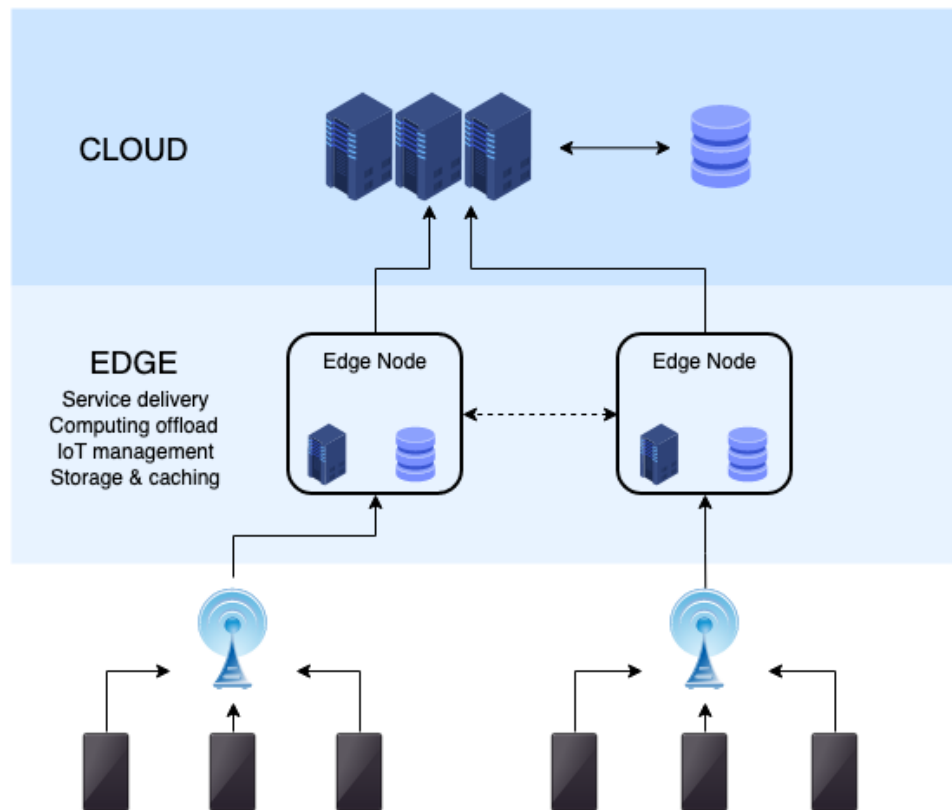


Figure 2.1: Edge computing architecture. (Reprinted from [8])

1. *Edge Layer*: This mainly comprises devices that collect raw data like sensors, actuators, etc., which are the front end. This layer mainly works to provide information from the real world, obtained from interaction with end-users in real-time. While these have some computing capacity, these devices must send data out to servers for more complex processing as resources available are limited locally.
2. *Fog Layer*: When the edge nodes need more resources than available locally, they send data to a near-end environment where most of the computational and storage requirements of the application can be fulfilled. In edge computing, the high resource needed computations like real-time data processing, data caching, and computation offloading can be achieved with a good performance with small latency.
3. *Cloud Layer*: The cloud servers can provide massive parallel data processing, big data mining, big data management, machine learning, etc. at the cost of high latency. These are generally cloud servers deployed far from the user devices, but they can provide more computational power and large storage capacity. In some cases, Cloud-Edge acts as the main repository, which collects the reports or data sets from Edge End to improve a feature or a system [6].

2.3 Object Detection

Object detection is a computer vision technique for locating instances of objects in images or videos to detect and classify objects leading to many specialized fields and applications. The main aim of this is to not just look at images but to gain insightful results that can be further used. Object detection algorithms typically leverage deep learning to predict the class of an object and identify the location of objects in an image. This is typically done by drawing bounding boxes

around objects and then assigning them class labels. The challenge is to not only develop models that are efficient and fast but also real-time. The applications of computer vision have led to developments such as autonomous cars, face detection and advancements in robotics, and so forth. To improve accuracy and make them more reliable, these networks tend to have larger, deeper layers, but this means more computations and large amounts of data. Most of the current object detection algorithms need GPUs to compute or depend on sensors and radars for input data, which are not always real-time. Thus, using them for everyday applications is not feasible.

2.4 CNN (Convolutional Neural Network)

The scope of performance that CNNs show in the computer vision is one of the primary reasons deep learning has opened up to be a feasible option. The distinguishable difference between deep-learning neural nets and neural networks is that while neural networks generally have a single hidden layer, deep learning neural nets have multiple node layers through which data needs to pass through for the process of pattern detection or image detection. The neural nets are considered shallow as they consist of one input and one output layer and utmost one hidden layer, but in deep neural nets, the output of the current layer acts as training data for the next layers. So, the deeper a deep learning network is, the better is the feature extraction as it recombines and clusters the feature information from the previous layer. These have shown distinguishable performance improvement in the field of object detection and classification. CNNs breakdown the input images by taking them as frameworks of numbers called tensors. Tensors are matrices of numbers with extra dimensions, and these CNNs process images with the help of tensors [9][10].

2.4.1 Structure of CNN

Below is an image of LeNet-5 Architecture (Figure 2.2), which depicts how the input images go through multiple layers for detection. As seen, there can be multiple layers of the same type interspersed for optimal output.

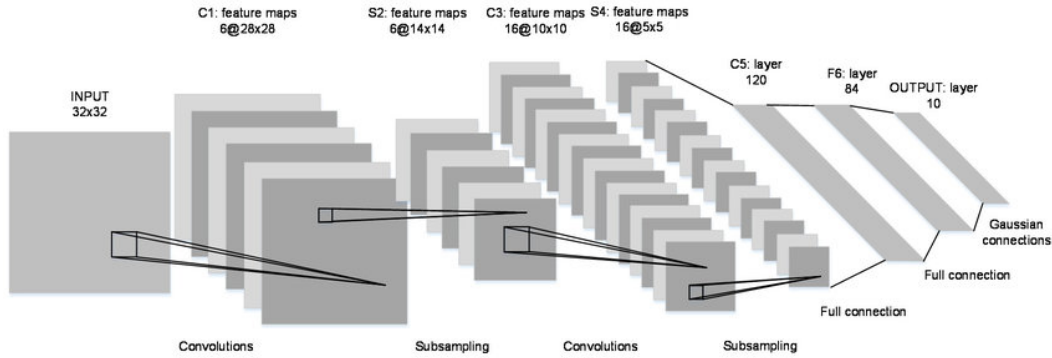


Figure 2.2: An overview of LeNet-5 architecture. (Reprinted from[11] CC-BY License. Used with Permissions)

2.4.1.1 Convolution Layer

This layer extricates high-level features from the input image. An image can be represented as a matrix of $n \times n$ dimension. A filter of $s \times s$ dimensions, also called a kernel, can be convolved over the input image to get a feature map. This stack of feature maps can be put together to get a convoluted image, which essentially is the output of this layer. There can be more than one convolution layer; that way, the initial layers would be responsible for extracting low-level features like color, edge, gradient, orientation, etc. The next ones will extract features that help the network have an overall understanding of the input image [9].

2.4.1.2 Pooling or Sub-sampling Layer

This layer is responsible for decreasing the spatial size of the input. This is to reduce the memory footprint and hence would need less computational power. There are multiple ways to pool the features, like average pooling and max pooling or minimum pooling, and depending on the requirement; one can be chosen. This layer is useful to extract dominant features by partitioning the input image into a set of non-overlapping rectangles and takes the largest value in max sampling and the average value in average pooling layers. This way, over-fitting can be controlled as the computations become manageable, and the number of parameters also reduces. This layer is often added in between successive convolutional layers [9].

2.4.1.3 Fully Connected Layer

In this layer, non-linear combinations of the outputs from the previous convolution and pooling layers are taken as inputs and learned for further processing. In the fully connected layer, high-level reasoning with neurons occurs. The neurons are connected to all activation from the previous layers, so each of these neurons is connected to one that could represent high-level features from the previous layers. This could be used to classify the image into various classes using the softmax classification technique [9][10].

Table 2.1: Comparison of CNN implementations

CNN Name	Year	Developed by	Error Rate	No. of parameters	No. of layers
LeNet [12]	1998	LeCun <i>et. al.</i>	-	60 thousand	7
AlexNet [13]	2012	A. Krizhevsky <i>et. al.</i>	15.3	60 million	8
GoogleNet/ Inception V1 [14]	2013	Szegedy <i>et. al.</i>	6.67	4 million	22
VGGNet [15]	2014	Simonyan and Zisserman	7.3	138million	16
ResNet [16]	2015	Kaiming He <i>et. al.</i>	3.57	26 million	152

2.5 YOLO (You Only Look Once)

YOLO algorithm [17] has been developed as a single regression problem for object detection with bounding boxes. The network of YOLO has separate components combined, and this network, while making decisions, takes the input as a whole. YOLO is good for real-time object detection as it works robustly with unexpected inputs, and it is also adaptable to new domains. An input image is divided into $s \times s$ grid cells, and each grid cell predicts with bounding boxes, conditional probabilities, and confidence score. The confidence score is the accuracy of prediction and probability that the objects belong to a predefined class of objects in the algorithm. The probability that an object exists is specifically evaluated for each grid, but the confidence score is evaluated only for each bounding box. If there is no object in the grid cell, then the probability is zero, and if an object exists, it is one. Conditional probabilities are predicted class probabilities, which indicate if there exists an object in the grid cell and specify how aptly the box fits the class predicted. Only one set of class probabilities is predicted per each grid cell, even with more than one bounding box in that cell. YOLO v1 is implemented as a Convolution Neural Network and Darknet framework, which is trained on the ImageNet-1000 dataset. The dataset has predefined classes labeled, which means that the detected object is classified into one of the labeled classes. The confidence score is evaluated on how well the object fits into the class. It has 2 fully connected layers preceding 24 convolutional layers and uses a 1×1 reduction layer followed by 3×3 convolution layers [17].

YOLO has several versions that have been brought on to improve the original version. This version one has trouble detecting small objects and when object density is high or if objects are clustered closely. The YOLO v2 is a faster and more robust object detection algorithm developed at the end of 2016. YOLO v1 assigns an object to the grid cell that contains the middle of the

object. So, when an overlap of objects exists, for example, a person wearing a tie, the algorithm needs to detect the person and the tie. Still, since v1 predicts only one class of probabilities per grid cell, it detects only one object. So, addressing this problem, v2 lets the grid cell use more than one bounding box to detect objects. The second version of YOLO uses a combination of both the COCO dataset and ImageNet. To improve performance in small object detection, YOLO v3 has been developed. This uses Darknet53 for feature extraction. This is a hybrid of the v1 and v2, so it uses a combination of them to get more fine-grained information for detecting small objects, but v3 has slightly lesser performance compared with earlier versions in matter of large and medium objects. YOLO v3 is fast and accurate, it has significantly better performance over other detection systems [17][18][19].

We have a modified version of this object detection algorithm called Tiny YOLO, which has been developed for running on constrained platforms. Tiny YOLO's latest improvement, a combination of better various versions of YOLO, is a smaller, faster, and better model for constrained platforms. The data sets used are a combination of COCO and PASCAL VOC, like the earlier versions of YOLO, and this uses Darknet framework [20]. Figure 2.3 shows Tiny YOLO performance compared to YOLO. Tiny YOLO has 9 convolutional layers, whereas YOLO has 24 [21].

2.5.1 Tinier YOLO

A Real-Time Object Detection Method for Constrained Environments has been proposed in [23]. Here they have improved upon the already existing Tiny YOLO v3 and added a fire module. The fire module was first introduced by UC Berkeley and Stanford researchers and used a convolution layer to compress the feature maps to reduce parameters. Tiny YOLO v3 is large and has a slow detection speed, both of which were led to a decrease in accuracy when compresses. To make it more

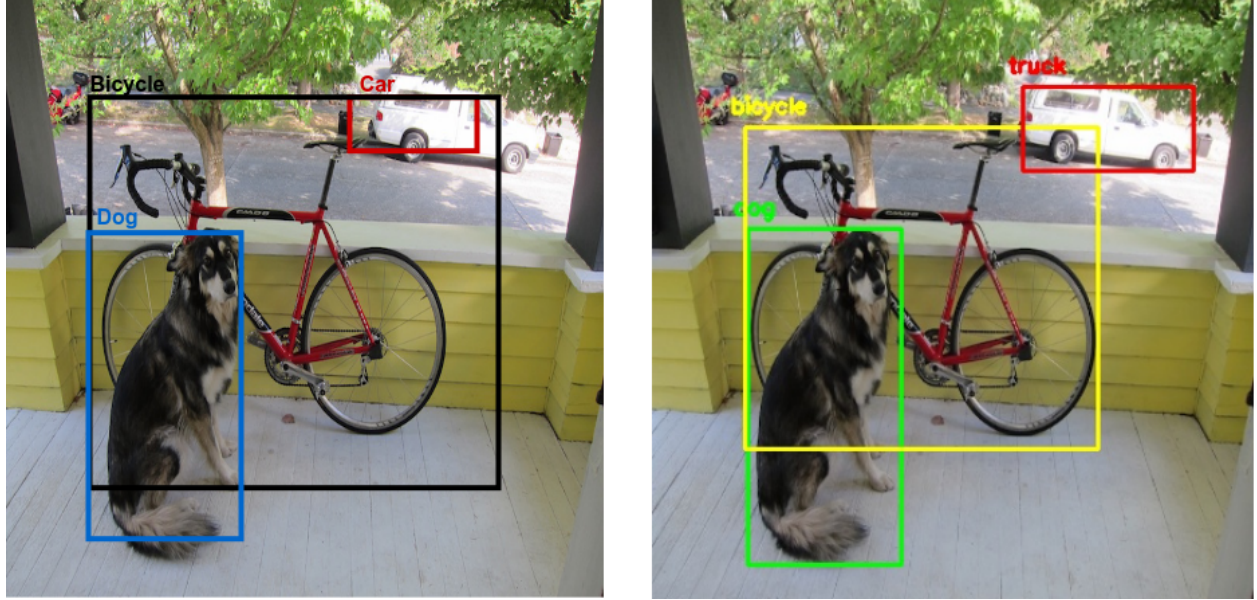


Figure 2.3: Tiny-YOLO vs YOLO. (Reprinted from [21][22]. Reprinted with permission)

viable for constrained platforms, this Tinier YOLO was introduced and trained on the PASCAL VOC dataset. The network structure is optimized by reducing the parameters and introducing the fire module to get a smaller and faster model. The middle layers are replaced by fire modules, which increase the depth and width of the network for detection accuracy and reduce the number of parameters. The real-time performance has been tested on Jetson TXI and has an average test time of 25.1 frames per second, while the Tiny YOLO v3 has 21.6 frames per second. The real-time performance has been increased by 16.2 percent compared with Tiny YOLO v3. Tinier YOLO takes 135.2 ms to detect an image, while MobileNet SSD takes 483.7 ms, and SqueezeNet SSD takes 595.7 ms, where MobileNet SSD and SqueezeNet SSD are also lightweight models for image detection [23].

2.6 Quantized Neural Networks

Deep Neural Networks have been successfully used for object detection and classification or speech detection and various other challenging problems. This involves a large number of compu-

tations in training the parameters, and so the algorithm learns to differentiate between classes. To have better performance, the data used for training is extensive, which means using them or even training them on general-purpose digital hardware is highly inefficient. It becomes a challenge to run them on constrained platforms, and they generally require specially designed GPU (Graphic Processor Units). So, there has been a considerable amount of work done to run these with a smaller footprint so that the run-time can be reduced.

Some of the approaches include compressing the layers or data reusing or reducing data precision so that there is a decrease in memory size and an increase in the speed of computations. Another is quantizing the neurons or the activation bits or weights where the arithmetic operations on the network, which are generally 32-bit floating-point numbers, are reduced to lower bit width numbers. Hubara *et.al.* [24] have proposed a Quantized Neural Network for quantizing the neurons and weights. They have replaced most of the arithmetic operations with bit-wise operations. The quantized gradients use only 6-bit numbers, and they proposed two sets of experiments, one using 6 bits and another with 1 bit per weight and activation. The second one has led to Binarized Neural Network (BNN), which claims to have reduced the time complexity by 60 percent. They have achieved comparable results to the 32-bit floating-point architectures.

Though there is a trade-off in terms of accuracy, these QNNs can be implemented on various hardware platforms such as FPGAs (Field Programmable Gate Array), and also they are more compact and are energy-efficient algorithms. Umuroglu *et.al.* [25] have proposed a framework called FINN for fast, scalable Binarized Neural Network Inference accelerators on FPGAs. They have implemented this framework on a ZC706 embedded FPGA platform and demonstrated up to 12.3 million image classifications per second with 0.31 μ s latency on the MNIST dataset with 95.8

percent accuracy, and 21,906 image classifications per second with 283 μ s latency on the CIFAR-10 and SVHN datasets with respectively 80.1 percent and 94.9 percent accuracy.

Preußer *et.al.* [26] present a case of QNN on heterogeneous all-programmable devices for real-time object detection in a live video stream. They have achieved a rate of 16 frames per second on an embedded Zynq UltraScale+ (XCZU3EG) platform. The framework involves work done on both the Tiny YOLO and the FPGA. They have developed Tincy YOLO on the Darknet framework and used customized weights and activation. The framework leverages the four available processor cores to design a combination of parallelizable and pipe-lined architecture that achieves a speedup of 160.

2.7 Distributed Object Detection

Distributed object detection has been proposed to exploit the advantages of edge computing and to reduce latency in object detection for real-time applications. Distributed object detection can reduce deployment cost and have accurate object detection models with low latency. As we move into an era of wireless devices, we have new challenges to overcome in location-specific applications. The data collected from these IoT devices in real-time and non-fixed in size. All these bring in the need for a dynamic network that is adaptive and robust.

Ren *et.al.* [27] propose a distributed edge computing-based object detection architecture for surveillance applications. Their model has three layers, end devices, edge servers, and the cloud. They interact to adapt to improve the performance of object detection. They have used the PASCAL VOC data set and trained the model on R-CNN (Recurrent-Convolutional Neural Network) models. The image captured by the end device is compressed and sent to the edge server, which runs the

object detection algorithm. This edge server is limited by the local media data from other nearby end devices. So, in the case of an unusual object that cannot be detected, it sends the image to the cloud where the information of all edge servers is gathered. So, there is a better scope of detecting the object. Once the cloud detects the object successfully, it sends the new weights and related metadata to the respective edge server to store for further reference. The detection accuracy for this model shows that it is reduced by only 2.5 percent when the image compression ratio is 60 percent [27].

In another direction, Liu *et.al.* [28] have addressed the need for dynamic clustering in edge computing on an IoT network. The network requirements of IoT devices used for edge computing are dynamic as the location of these devices is not static. This paper has proposed a solution to cluster the data from IoT devices for maximum communication performance and computing performance of edge servers. This model uses a deep reinforcement learning algorithm for object detection [28].

Zhang *et.al.* [29] have a framework with Heterogeneous Distributed Deep Neural Networks (HDDNN) overcloud, fog, and end devices. The framework attempts to take into account the memory and power limitations of each layer for better resource utilization by distributing the computing on the computing nodes. End devices use MobileNet, which is a small, fast neural network model and fog devices use ResNet and WRN (Wide Residual Network) on cloud devices. The end and edge layers are loaded with coarse and fine labels so that they classify the input image. The cloud can perform a better classification as the framework exploits the distributed hierarchy. This framework offers low response time, optimized resource allocation, privacy protection, and robustness [29].

2.8 Object Detection for Hardware Platforms

Shrinking the number of transistors per chip has powered 5 decades of advances in information technology. The number of transistors on a chip doubled about every two years, in step with Moore's law. Almost every advancement in chip manufacturing gave rise to new technological device production starting from mainframes in 1950 to embedded processors in 2010. It is the primary reason for the computer revolution and the digital age. However, this trend has come to an end, fabricating more number of transistors on a small chip resulted in more heat emission, cooling down the transistors needs more energy than the amount of energy consumed by the transistors. But the exponential increase in demand for powerful hardware computation systems leads to new architectural breakthroughs like manufacturing multi-core processors to overcome the power consumption problem, designing domain-specific architectures (DSAs). Deep neural networks are an essential part of computer vision and natural language processing (NLP) systems. They work on pre-trained datasets or by training an existing dataset. They are composed of deeply connected layers that have convolved inputs and outputs that are processed between them. Because of these high memory requirements and complexity, networks are mostly trained using powerful hardware. Today there are several types of processors, but for image processing and object detection, we can categorize them into CPU(Central Processing Unit), GPU (Graphics Processing Unit), an FPGA (Field Programmable Gate Array).

2.8.1 FPGA (Field Programmable Gate Array)

The FPGA was mainly developed as a result of rapidly evolving complex hardware requirements. They are the preferred choice for neural network applications for their performance in terms

of execution time, power consumption, and flexibility to adapt to real-world use cases. The basic working of the neural networks involves the classification of common nodes by weighted inputs and clustering them after convolution. This constitutes a large number of computational operations.

As addressed by Koromilas et al., in [30], the number of operations required to process input images is close to 100 billion operations as the computational complexity, and the size of the input images are directly proportional. FPGA's have high-level parallelism and the capability to simplify logic with respect to the computational process of a Neural Net without affecting the accuracy of the neural network. And as FPGA's gradually became candidate platforms, they were found to be able to reach higher energy efficiency than CPUs and GPUs. Hardware accelerators have been developed to address the problem of increasing bottlenecks like networking and storage.

In 2016 Xilinx released a framework called Re-Configurable Acceleration Stack. This aims to deploy an FPGA accelerator for hyper-scale data centers. So, this can deliver up to 20x acceleration over traditional CPUs with a flexible, re-programmable platform for rapidly evolving workloads and algorithms.

Wang and Wang [30] proposed a framework for seamless utilization of hardware accelerators for heterogeneous all programmable SoCs based on PYNQ (Python Productivity for ZYNQ) framework. Their framework is based on logistic regression that is connected to processors through the interface and integrated it with the Spark framework to support PL (Programmable Logic) on SoC. They claim to have achieved an 11x speedup and that this framework can be used to support any hardware accelerator. It can be used in fog and edge computing as it allows for fast deployment of embedded and cyber-physical systems.

While a lot of work has been done to leverage the hardware for optimal performance, Xiao *et. al.* [31], have explored heterogeneous algorithms for accelerating deep CNNs on FPGAs by leveraging software design to fuse multiple layers in CNNs and reusing intermediate data. They also developed an automated toolchain for mapping from the Caffe model to FPGA bitstream using Vivado HLS and achieved up to 1.99x speedup when compared to other fusion-based FPGA accelerators for CNNs.

For real-time object detection with CNNs for IoT intelligence on the Edge, Hao *et. al.*[32] have a design methodology for co-design flow. They proposed to simultaneously modify the DNN and design an FPGA accelerator using PYNQ-Z1 FPGA. Their design is to meet performance constraints on IoT platforms and has achieved 2.5 times better efficiency and has consumed 40 percent less power than the first place winner of an LPODC (Low Power Object Detection Challenge) that was conducted by DAC.

Sharma, Singh, and Rani [33] demonstrate the deployment of CNN on ARM embedded Xilinx Zynq based FPGA using Xilinx PYNQ framework for real-time object detection. They have evaluated four object detectors for custom data sets and video feed inputs. They concluded that a combination of SSD (Single Shot Detector, which is an architecture predominantly used for object detection) and Inception V2 model has the most accurate results and can work in a real-time frame rate and has exhibited efficient performance for embedded design development.

2.8.2 GPU (Graphics Processing Unit)

The evolution of Deep CNNs for object detection techniques has to be attributed to the development of GPU's computational power. GPU's ability to offer peak performance makes them

an ideal option for neural network acceleration. Neural nets are trained on GPUs traditionally as they have hundreds of cores and enormous amounts of memory on a single chip. Large data sets are compiled and used for training as background noise in the images can lead to false positives. GPUs have small shallow caches that can be leveraged for parallel operations to compute a single function using vector processing units. This enables parallel operation's computation to execute a single piece of code in multiple pieces of data in parallel.

Technologists exploited advancement in multi-core architectures by designing domain-specific architectures to run compute-heavy neural networks like Amazon Deep Lens (a deep learning enabled video camera) for developing computer vision applications based on a deep learning model etc.

Coates *et.al.* [34] proposed an object detection system to scale large datasets and leverage the computational power of GPUs. Nvidia's Cuda development library is used to implement the significant test time components for real-time object detection. They compute on a custom dataset for small objects and have performed object detection under 5 seconds.

Nvidia in [35] has run inference benchmarks across popular models and a performance evaluation of various deep learning networks with Jetson Nano and TensorRT, using 16-bit floating-point precision and batch size one. It shows how each of the networks for object detection scales against the others with respect to frames per second. Among various platforms and algorithms, we see Tiny YOLO v3 being compared to various version of SSD like SSD ResNet-18 (Object Detection algorithm), SSD Mobilenet-V2 (Object Detection algorithm), MobileNet-v2 (Classification algorithm), and VGG-19 (Image Classification algorithm) and the performance of Tiny YOLO v3 on NVIDIA Jetson Nano is 25 frames per second.

2.9 PYNQ (Python Productivity for ZYNQ)

In 2016, Xilinx released the PYNQ framework structure that permits the quick programming of the heterogeneous all-programmable SoC. Utilizing the Python language and libraries, planners can abuse the advantages of programmable rationale and chip in Zynq to manufacture progressively skilled and energizing embedded frameworks. Programmable rationale circuits are exhibited as equipment libraries called overlays.

The key distinction is the fabricate once, reuse many times paradigm. Overlays, like programming libraries, are intended to be configurable and re-utilized as regularly as conceivable in various applications. Zynq stage has an elite performance interface for the direct communication of the ARM cores with the programmable rationale part. The augmented bus is based on the ARM AMBA 3.0 interconnection that has numerous focal points, for example, QoS, multiple-outstanding transactions, and low-latency ways. The characteristic parallelism of two-dimensional convolutions coordinates well with FPGAs' solid parallel-processing capacity. It prompts a higher data handling throughput, and additionally, it consumes less power compared to more traditional embedded platforms.

The PYNQ-Z1 board contains a Xilinx Zynq-7000 ZC7020. This MPSoC (Multiprocessor System-on-Chip) comprises a dual-core ARM A9 hard-core processor, outfitted with different peripherals, as Processing System (PS). The image 2.4 shows an internal view of Xilinx Zynq-7000 SoC, here we can observe how the various interconnects and peripherals are connected. Besides, the ZC7020 incorporates the Programmable logic (PL). Because of the information-driven progression of CNNs, high information throughput is required. The data transfer of convolutional layers using AXI4-Stream-based interfaces rely upon payload information and further parameters for the pro-

cessing. Memory-mapped AXI4 interfaces with a balanced burst size are used to move the weight and bias parameters. Because of the weight sharing notion of convolutional layers, the information throughput on this port is a lot lower contrasted with the input data port of the layer. The hardware implementations of the CNN layers require a bigger number of assets than offered by the target device. Also, the resources are shared between the CNN equipment and the rest of the subsystem [2].

In this way, DPR (Dynamic Partial Reconfiguration) is applied to share the FPGA assets between numerous layers of the CNN. DPR allows the user to re-program a part of the FPGA fabric while the remaining logic resources continue to operate without interruption. DPR empowers a decrease in the timing overhead by decreasing the measure of reconfigured rationale. The reconfiguration time and area are major constraints. To support the deployment of dynamically reconfigured hardware accelerators on the programmable logic, the FPGA area is divided into two main regions: a static region and a re-configurable region. Reconfiguration operation can occur while the FPGA logic is in a reset state (static) or running (dynamic). Within the FPGA fabric, DPR enables the exchange of logic partitions, called Re-configurable Partitions (RP). Thereby, the system can be adapted for an optimized execution according to the application, as well as the application phase. This is a major advantage for designing efficient hardware architectures able to adjust the hardware due to characteristics of CNNs, Recurrent Neural Networks (RNNs), or combinations of both, but also due to the characteristic of layers within these networks. Moreover, the integration of DPR enables the exploration of dynamic adaptations [2][36][37][38].

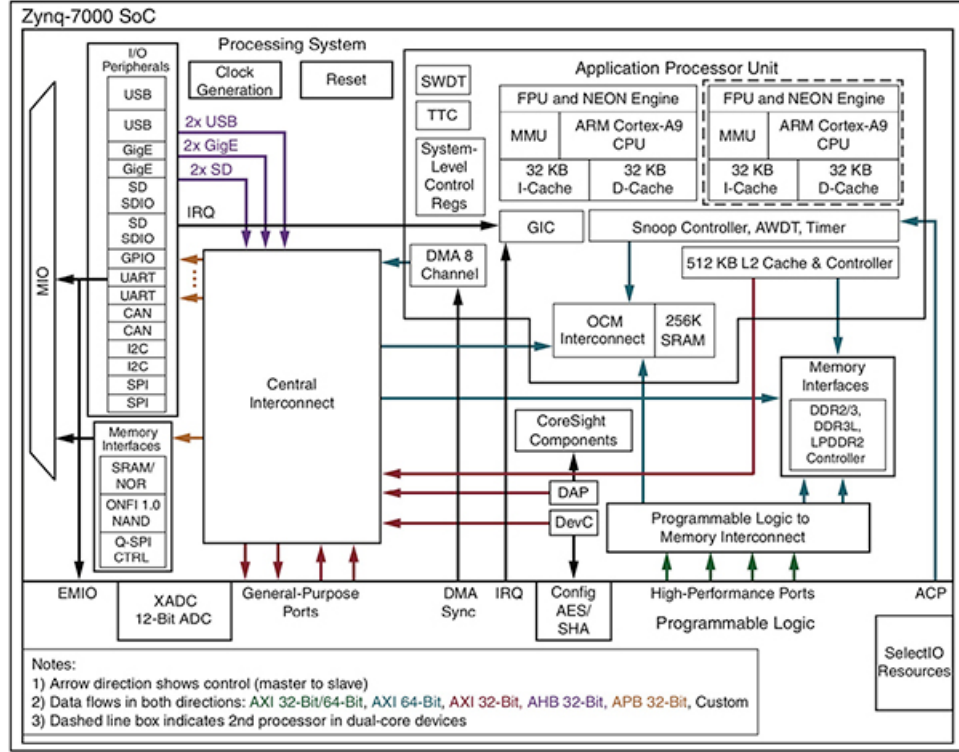


Figure 2.4: Xilinx Zynq-7000 SoC [2]. (Internal View) Reprinted with permission.

2.10 Load Balancing

Load balancing is understanding and ensuring the performance of servers, websites, and various other applications during a large influx of traffic. This traffic spike needs to make sure the applications using a load balancer are available, have uptime, and better performance. Load balancing can be categorized into static and dynamic. Static load balancers distribute the load uniformly irrespective of the state of the node or computing component. While they have their own uses, in some scenarios, we need to take into account the state of each node and disperse the load depending on their computing capabilities. These are dynamic load balancers. Distinct algorithms are used to decide which routing algorithm to route to by the network administrators depending on the requirements of the detailed application or site. The round-robin algorithm sends a request to a

server and then moves to the next server after a period. It's generally used when all the applications have similar processes or memories. Some other algorithms are the least connection method, least response time method, and IP (Internet Protocol) hash method.

2.10.1 HAProxy (High Availability Proxy)

HAProxy stands for High Availability Proxy. It offers high availability and load balancing and is an open-source software TCP/HTTP (Transmission Control Protocol/ Hypertext Transfer Protocol) proxying solution which can be run on multiple operating systems like Linux and Solaris. It is used in situations with particularly high traffic and improves performance and reliability. It is easy to integrate into existing architectures and is the most widely used software load balancer. It is a powerful and reliable method to distribute load among servers. HAProxy is considered less risky, easy to use, and does not exhibit fragile web servers to the internet. It offers various load balancing algorithms which can be implemented according to the user requirement like Round-Robin, Least Connections, Source, URI, and URL Parameter [39][40][41].

2.11 Flask

Flask is a micro web framework, which means that it has all the tools, technologies, and libraries that one needs to build a web application. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine that is lightweight and has the capability to scale up to complex applications. Sockets can be used in flask applications, and these enable a bi-directional, low latency communication application between a server-client pair.

2.12 Sockets

Sockets facilitate communication between processes located on the same machine or different machines on a common network. We use socket programming to create a bidirectional connection for each of the server-client pair. Sockets are primarily used as a way of connecting two nodes on the same network to where the server waits for the client to make a request, and then the server responds to the request by sending the requested data to the client. Sockets have one node generally listening on a particular port on an IP (Internet Protocol) address and another node that comes forward with a connection request. The server then responds to the request and establishes a connection for communication. Figure 2.5 shows how the client-server relation is established with a request-response protocol over a port [42].

Sockets are reliable and have in-order-data delivery; that is, the order in which the data is sent is the order in which the data is received. When the data is sent as a continuous stream of characters, it is categorized as a stream socket. Here the communication protocol is TCP (Transmission Control Protocol), and if the entire message is transmitted over in a message-oriented UDP (Unix Datagram Protocol), they are categorized as data-gram sockets [43].

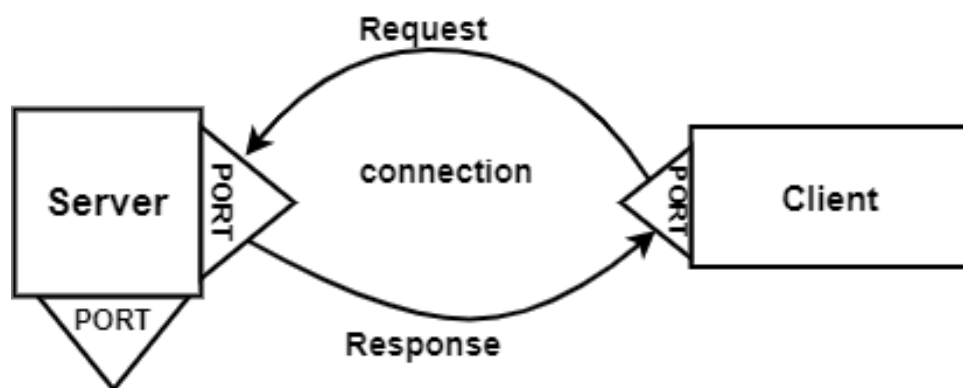


Figure 2.5: A basic socket overview.

2.13 Related Work

We review the existing architectures for distributed object detection with deep neural networks on constrained platforms for edge computing in IoT in real-time.

The data flow starts from the edge end devices, which capture the real-time data and transmit it to the fog end or edge servers. Here the object detection is performed, and for objects detected with low confidence, the data is further sent to the cloud end for better results.

In the architecture proposed by Ren *et.al.* in [27], the end devices are responsible for data compression as they propose to use wireless networks instead of using high-speed fiber networks and transmitting to the edge servers. The edge servers have multiple distributed infrastructures like local servers that use the data from the edge end devices and run object detection algorithms. These algorithms need to be lightweight without compromising accuracy, and each of them serves a few end nodes/devices. The advantage of this layer is that it is equipped for real-time data and adapts to the compression algorithm of end devices for optimal object detection results. The cloud layer integrates the edge model and keeps the parameters updated. If the detection on edge servers fails, the data is forwarded to the cloud end. The cloud is better equipped for object detection as it has a collection of parameters from all edge servers. The cloud end also has better computing capability because it is not limited by size, and it keeps the edge servers or fog end devices updated with relevant parameters.

As the Figure 2.6 shows the structure of the architecture, the end device used in this experimental platform uses an image compression method named BRE (BING [44] RoI Extraction algorithm), based on Region-of-Interest (RoI) extraction method. BRE essentially detects the region of object and background. It compresses the former minimally and the latter sufficiently enough for

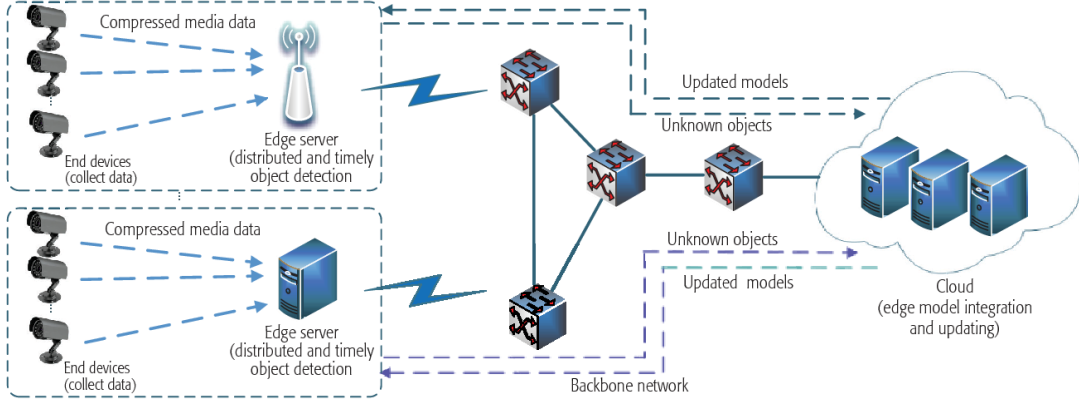


Figure 2.6: Edge computing-based object detection architecture. (Reprinted from [27]. © 2018 IEEE)

effective wireless communication. The edge server has fast R-CNN as the object detection model, and the cloud trains on the end devices input images and updates the rest of the layers so that new objects can be detected easily on the fog end devices [27].

Cheng *et.al.* [45] have proposed an object detector with dynamic workload balancing for an FPGA. Their algorithm is based on one of the more widely used algorithms used for object detection with a special application for face detection, by Viola and James [46]. They propose an architecture to map this onto an FPGA for dynamic workload balancing among the computational units in its architecture for optimal resource utilization. They leverage multiple processing elements and redistribute workload among them for effective hardware utilization. They have addressed the ineffective hardware utilization on FPGAs for real-time object detection with a novel approach.

Jokic *et.al.* [47] have proposed a streaming camera system for on-device real-time image recognition on an FPGA using BNN (Binary Neural Network). They have introduced a BinaryEye streaming camera for onboard image recognition to avoid data streaming through the communication interface. This helps in reducing energy consumed and latency. It uses BNNs to classify RoI

(Regions-of-Interest) for large scale data reduction. Their case study achieves a camera speed of 20 kfps for edge processing and is implemented on the MNIST dataset for a handwritten image classifier. The pure streaming implementation by the camera leads to reduced data transmission and allows for a high frame rate without congesting the communication interface. The low memory footprint of BNNs, which have simplified computations, lets the FPGA implementation be power efficient. This has been implemented on the Xilinx PYNQ board, and the input image from the camera is transferred directly to the USB interface on the board, which accounts for less latency.

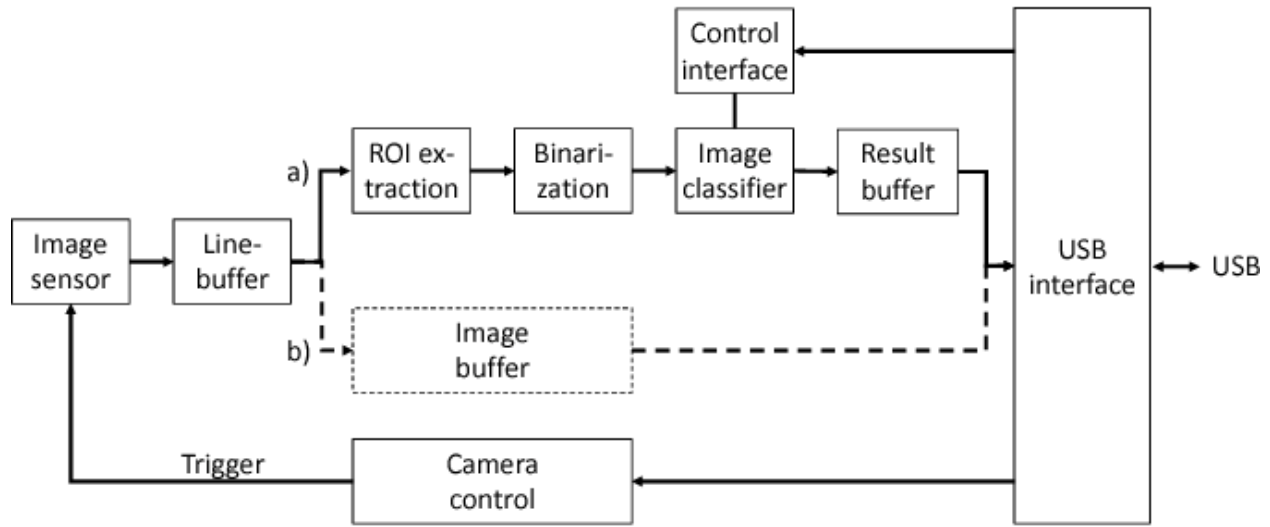


Figure 2.7: System overview showing the data-flow. (Reprinted from [47]. © 2018 IEEE)

The data flow depicted in Figure 2.7 starts at the image sensor gets triggered by the camera control block through a USB command. In one streaming application, all data is transferred to the USB interface directly. A DRAM buffer is used to store the images to compensate for the speed at which the images are produced. BinaryEye extracts RoI and completes binarization of input image and then classifies it. The result is stored in the result buffer before sending it to the USB interface. The camera control block does the trigger scheduling and image-sensor configuration functions. The framework by Garry and Molloy in [48] is implemented on Xilinx Zynq SDSoC (software-defined

system-on-a-chip) for real-time image analysis on a software/hardware Co-design framework on IoT edge devices.

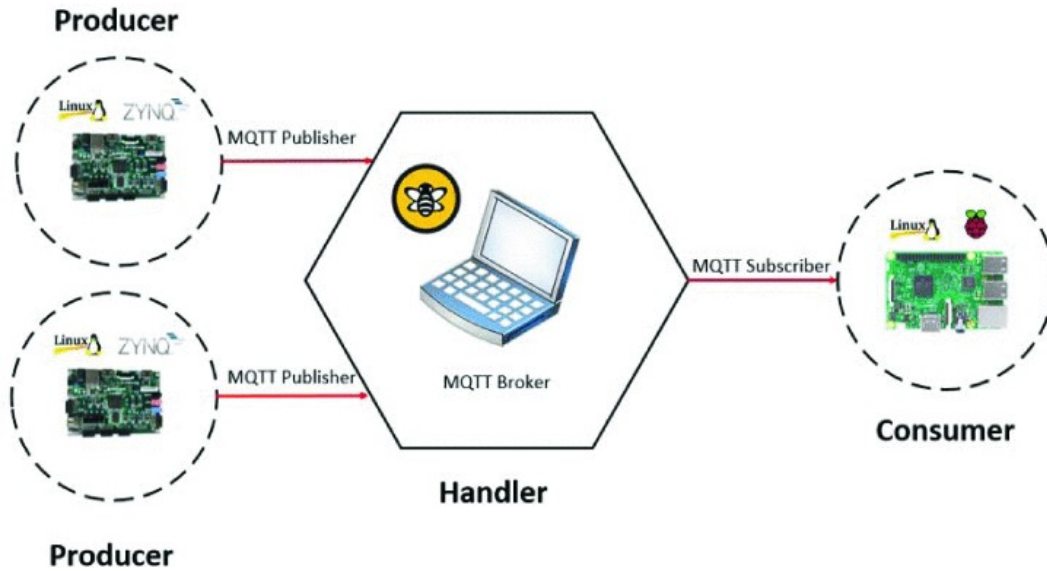


Figure 2.8: Overview of architecture. (Reprinted from [48]. © 2019 IEEE)

Figure 2.8 depicts an architectural overview of proposed in [48]. It has a producer, a handler, and a consumer.

- The producer in this is the Xilinx Zynq SoC PYNQ, which captures the input images with an HD camera to estimate the traffic flow. Its main function is to do local image processing and analysis.
- The handler is the intermediate agent mainly used for additional processing and data storage, in this application, it is a Ubuntu Linux virtual machine.
- The consumer in this application is a Raspberry Pi SBC. This is an IoT device that acts on the data received from the intermediate handler and works as a display unit or an actuator depending on the requirement.

This framework was applied to a distributed motorway vehicle counting application in real-time. They have used the PYNQ board's peripherals and developed a custom IP (Intellectual Property) in the PL (Programmable Logic) of the Soc and uses image analysis techniques to determine if a vehicle has crossed an arbitrary line. These analysis data are sent to the handler, which stores it until a subscriber requests it and then redirects the data. The subscriber, when receives this message, can act accordingly and take action like change the speed or gain insightful information about an accident or so. The frame rate of the video used was 25 fps (frames per second) in the test video and the graph 2.9 shows the response time for processing each frame in the test video. The PL was tested for edge detection, greyscale filter, and vehicle count. Different devices have been tested, the PYNQ board, the Raspberry Pi, and an Intel-based HP laptop.

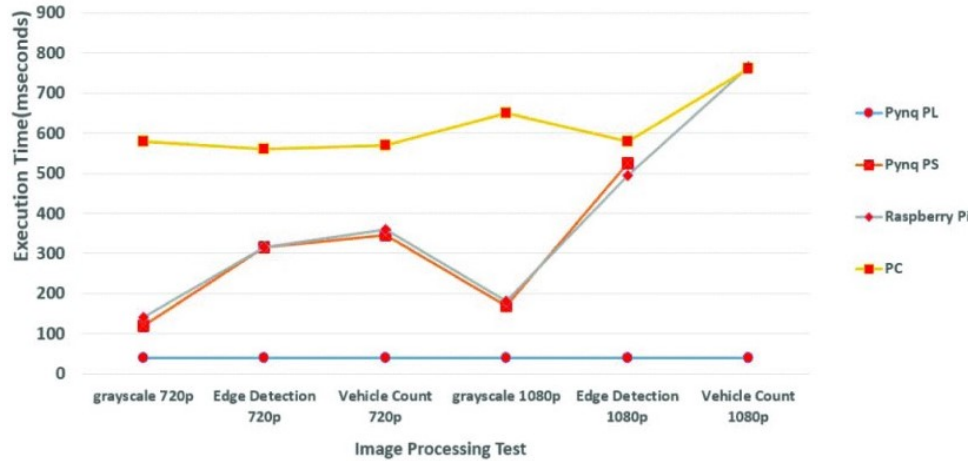


Figure 2.9: Comparison of image processing response time. (Reprinted from [48]. © 2019 IEEE.)

2.14 Chapter Summary

This chapter discusses the various technologies that have been introduced in the previous chapter and how each of them is related to each other. We have looked at image detection and then CNNs and a state-of-the-art algorithm for image detection called YOLO, then QNNs. We

presented in detail the need for distributed object detection on various platforms and looked at the PYNQ Z1 board as a candidate platform for the same. We talked about other topics, like Flask and Sockets, which have been used to develop our framework, which we will go over in detail in the next chapter. We have reviewed existing architectures for distributed object detection in edge computing on FPGAs, such as the PYNQ board. The edge computing architecture has been the central concept of the reviewed work. We have also looked at the varied neural networks used for object detection in a wide range of applications.

Chapter 3: Proposed Work

The idea is to create a framework to effectively load balance a distributed object detection with FPGA as a platform. The framework needs to be robust and dynamically allocate resources for proficient deployment of object detection on low power heterogeneous MPSoC (Multiprocessor System-on-Chip) FPGA called PYNQ. On-board processing for image detection has been a well-researched topic, where the image is captured with a USB camera, and an image detection algorithm is run to recognize objects. This can be called a node.

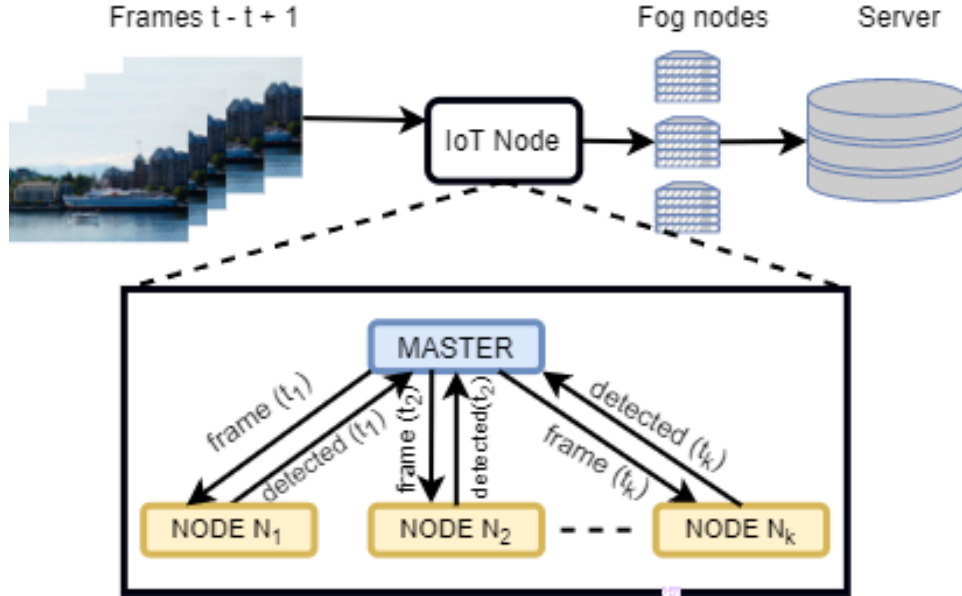


Figure 3.1: An overview of our architecture. Traditional approaches perform per frame detection on a single IoT node.

In the case of surveillance systems, the number of images captured is large, which requires fast processing. One way to achieve that would be to optimize the image detection algorithm it-

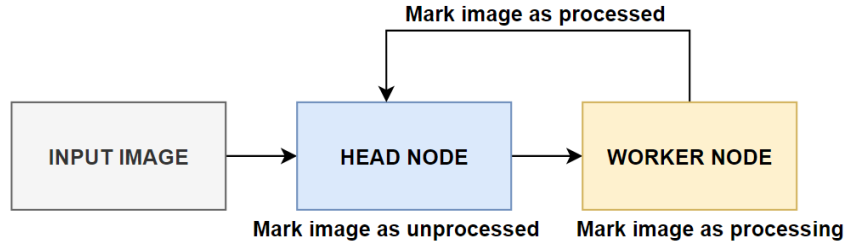


Figure 3.2: Block diagram of the framework.

self; another would be to use distributed computing. Some of the challenges to overcome in that scenario would be scaling it with multiple nodes, observing the communication and coordination between nodes. One more obstacle would be to make sure the resources are available. Then comes the question of how they would work together as a cohesive distributed system that can utilize the various resources available. Balancing the load across multiple nodes includes optimizing the response time, making certain for a uniform distribution of load, and maintaining less overhead. Using a distributed system model for our framework (Figure 3.2), we take advantage of the distributed system's characteristics and define the requirements of the framework:

1. Real-time image processing: We need to process the images in real-time.
2. Workload distribution: The work should be uniformly distributed among the available resources.
3. Object inference: An independent module that runs an object detection algorithm to detect objects.

Based on the framework's requirements, we can assign the tasks to computational nodes and define them as below:

- One node called the Head node caters to the interaction between nodes and responsible for acquiring and organizing input images. It communicates seamlessly with numerous nodes at the edge of the network to distribute the load and utilize the available resources.
- The other node, the Worker node, mainly focuses on establishing a connection with the Head node to receive new inputs. It runs an object detection algorithm and returns an image with objects detected in it. Once the resultant image is sent to the Head node, the Worker node terminates the connection with the Head node.

3.1 Initial Work

We now describe the first model that was developed and how various components work together. The model has one Head node and supports multiple Worker nodes. The main idea behind multiple Worker nodes is to simultaneously run object detection algorithm on multiple images.

3.1.1 Head Node

The Head node's primary responsibility is to maintain and keep track of images that are captured, sent, and received. It essentially responds to the request of the Worker node like a server by sending an image. And once it receives the processed image with the object detected, it responds to another Worker node and so on. The server is initially in a listening state where it is ready to establish a connection. It is to dynamically load balance between the Worker nodes and utilize the available resources for the least possible latency. The Head node needs to be able to accommodate and adapt to an increase in Worker nodes and traffic.

3.1.2 Worker Node

The Worker node, when initialized by the HAProxy, requests the Head node. Once the input image is received, an object detection algorithm is run on it. We use Tiny YOLO v3 for constrained platforms, which takes an input image and draws bounding boxes around the detected objects. The Worker node then sends this image back to the Head node and goes back to a state of wait until the HAProxy pings it. The Worker node's main goal is to procure an image and run Tiny YOLO and send the image with detected objects back to the Head node.

3.1.3 Loadbalancing by HAProxy

HAProxy load balances the incoming requests of Worker nodes to an IP address, which in our model would be spread out the work on multiple PYNQ boards. Now the HAProxy is configured to listen to incoming HTTP requests on the port number set up earlier for Worker nodes. Each PYNQ board is connected to the network with a Flask application. The HAProxy will now visit each application one after the other in a round-robin strategy. As each Worker node application is hit, it requests the Head node for an image. Load balancing done by HAProxy is responsible for making sure that the Worker nodes establish a connection to the Head node one at a time. The time taken by a Worker node is not constant. It falls on the load-balancer to make sure that the Worker nodes which do not have an active connection are only invoked. Without a load-balancer, the Head node is not equipped to reach out to the Worker nodes by itself to establish a connection.

3.1.4 Dataflow

There are one Head node and multiple Worker nodes that work cohesively for a robust distributed detection, which is dynamic and effectively balances the load of the input images for

optimal use of resources. The input images could be a custom data set that is created by the user or real-time captured on the PYNQ board with the help of a USB camera. The rate of on-board processing to capture images on the PYNQ board needs to be adaptable to accommodate the number of Worker nodes on the network. The Head node and Worker node are in a state of wait, and HAProxy invokes the flask application that is running on the Worker node. The Worker node now sends a request to the Head node and runs an object detection algorithm on the received image, and sends the object detected image back to the Head node. The Worker node is now in a state of wait again until the HAProxy activates it.

3.1.5 Scaling the Framework for Multiple Worker Nodes

When there are multiple Worker nodes, and once the first Worker node is done communicating with the Head node, the HAProxy invokes the next Worker node in a round-robin manner. As long as HAProxy is configured to Worker nodes and each of the Worker nodes has a flask application running on the same local network, the framework stands until there are no input images on the Head node. The communication in this model works as a client-server system.

3.1.6 Limitations of this Framework

- While this model has a Head node that routes the traffic requests to each Worker node, it does so sequentially. This would, in effect, be wasting the resources of the rest of the Worker nodes while they are waiting for one Worker node to process the image.
- The Head node cannot keep track of the Worker nodes status as HAProxy is required to invoke the Worker node's web application requests. The Worker node may have been in an idle state before the load balancer hits it or, it could not be available on the network.

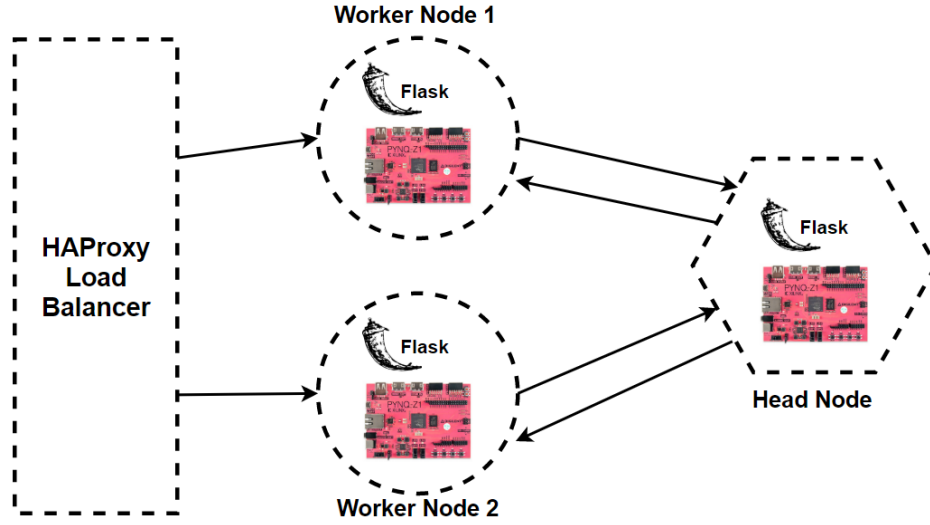


Figure 3.3: Overview of Model 1 architecture.

- The case of multiple requests at one instance does not occur as it responds to each Worker node one after the other in a round-robin algorithm.
- The Head node does not know if the Worker node is idle or busy; it does not account for the image if the Worker node fails to process it. That input image is lost, as the Head node assumes it has been processed while the Worker node fails to send a viable result back to the Head node.

3.2 Model 2

Now, we change the approach of the Head node and Worker node; until now, each Worker node needed to be invoked. In a step to change it, the Worker node itself makes a request every time it is in an idle state. Then we see that the Head node is not equipped to handle multiple requests at once. So, now we leverage the dual-core ARM hard-core processor of the PYNQ board to create multiple threads for handling multiple requests from more than one Worker node at one instance.

We now describe the framework and how the various components work together. The framework has one Head node and supports multiple Worker nodes. The main idea behind multiple Worker nodes is to run object detection algorithms on numerous images simultaneously. In Figure 3.5, we can see the Head node with a camera and three Worker nodes connected.

3.2.1 Head Node

We have three modules in the Head node. One module captures input images with a USB camera, and one module is responsible for allowing connections to Worker nodes and another that monitors the input images.

The *camera module* works autonomously to capture images from the real-world. The *connection module* works as a server to listen to requests on a designated IP address. As each of the Worker nodes sends a request for a connection, the Head node creates a socket connection. The Head node's connection module spawns a thread to respond to a Worker node's request for an image. The new thread takes care of the communication with the corresponding Worker node and gets terminated when the Worker node ends the connection. In the case of multiple Worker nodes making requests, the Head node spawns a thread for each of the requests.

The Head node is equipped to sustain multiple connections from Worker nodes and works to distribute the load uniformly among the available Worker nodes. This means that each connection is independent of the other so that when a new Worker node joins the network, it can make a connection with the Head node without causing any disruption to the already in-progress connections. This makes the Head node scalable and open to modification to multiple connections as it can run each socket connection independently on the same network. In the case of a Worker node crashing or

losing contact, the Head node's main task is to ensure that the rest of the Worker nodes equally share the workload of the dead Worker node. This makes the framework reliable in the event of any interruption.

The *organizing module* keeps track of images. This module supervises the images, so all images are sent for image detection and mark it, so another thread does not have access to it. The input image goes through three states, they are marked unprocessed when captured by the camera, and they move to process when the Head node sends the image. Once the result has been obtained, the image is marked processed. If the Worker node fails to process the image or when the result is not received, the Head node sends the images marked as processing again to another Worker node.

3.2.2 Worker Node

The worker node makes a connection to the Head node and acquires an input image. It runs an object detection algorithm and gives an output image with bounding boxes around the recognized objects. It sends the resultant image back to the Head node and terminates the connection with the Head node. Then, it repeats the process over again by promptly making a new connection with the Head node. All Worker nodes run autonomously and analogously for coherent image detection.

3.2.3 Communication between Head and Worker Nodes

In the timing diagram shown in Figure 3.4, the two nodes and their communication with respect to time from top to bottom can be observed. At the first step, 1(a) the Worker node sends a request to the Head node to establish a connection. In parallel, at step 1(b), the Head node is waiting for possible connection requests from Worker nodes. In step 1(c), the camera thread is spawned for capturing images. The next step, 2(a), shows the Head node accepting a connection

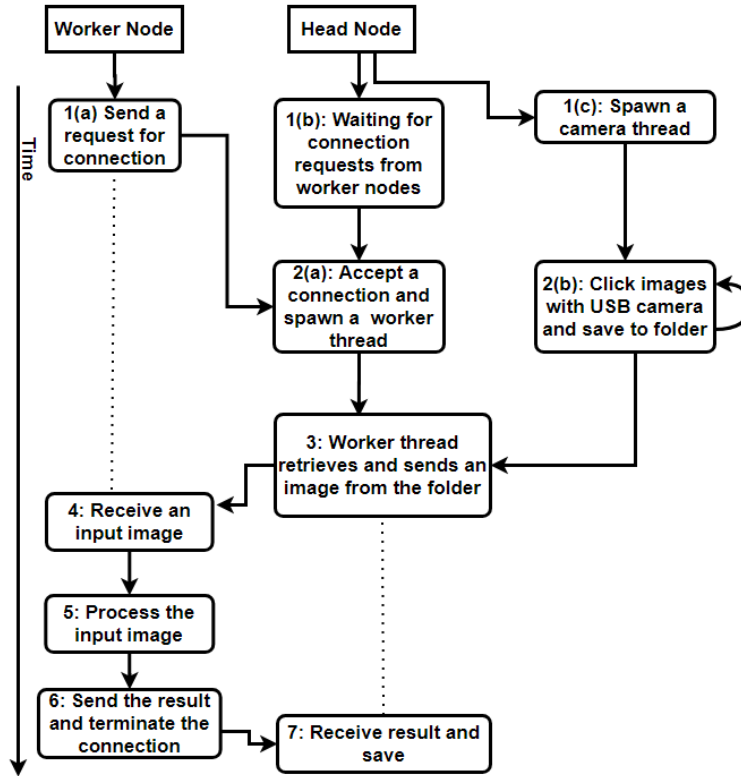


Figure 3.4: Timing diagram that shows the communication between Head node and one Worker node.

from the Worker node and spawns a Worker thread that is responsible for the communication with that Worker node. During step 2(b), the USB camera captures an image and saves it to a folder. And this step is repeated by the camera thread continually to capture images in real-time. In the next step at 3, the Worker thread retrieves a saved image from the folder and sends it to the Worker node that had sent a request. In step 4, the Worker node receives the input image, and then in step 5, an object detection algorithm is run. The result is sent back to the Head node in step 6, and the connection with the Head node is terminated. In step 7, the Head node receives the result and saves it, and then the Worker thread is terminated. The load balancing is effectively done by the Head node, as it works as a server to accommodate each Worker node's request. In Figure 3.5, the Head node is connected to three Worker nodes over WiFi. The Head node maintains active

connections with the Worker nodes and utilizes the available resources for the least possible latency. The nodes work together seamlessly and independently as a single coherent system exploiting the transparency characteristic in the framework.

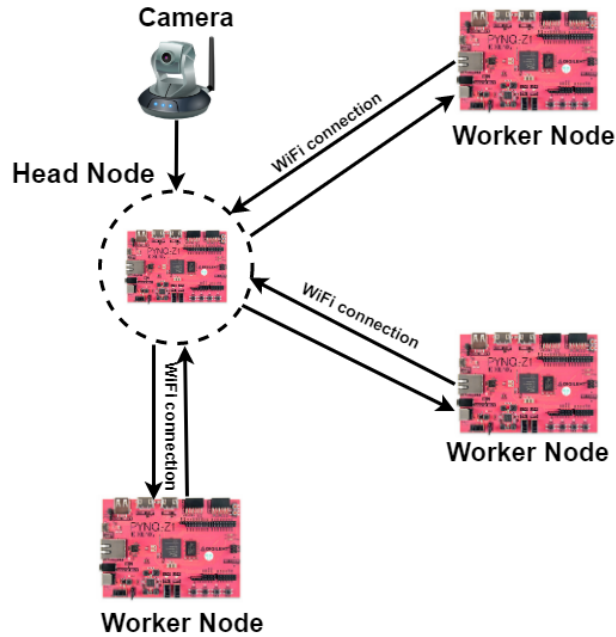


Figure 3.5: Overview of Model 2 architecture.

3.3 Chapter Summary

We proposed two models for a distributed system for object detection with a modified algorithm called YOLO on constrained platforms. The first model uses an external load balancer to distribute load among the nodes. This model has a client-server distributed system architecture. To address a few challenges in the first model, we developed the second model. This model has peer-to-peer distributed system architecture. A strong load balancing framework for a multi-PYNQ board cluster has been developed. We are exploiting the software capabilities of APSoC (All Programmable System-on-Chip) using the Python language and libraries in Zynq to build more adept embedded systems.

Chapter 4: Experimental Results

In this chapter, we report the experimental results performed on the designed framework. We used ZYNQ XC7Z020-1CLG400C SoC which has two cores of Cortex A9 processor, for the experimental validation. The specifications of this card is as shown in Table 4.1.

Table 4.1: PYNQ-Z1 SoC specifications [1].

650MHz dual-core Cortex-A9 processor
DDR3 memory controller with 8 DMA channels and 4 High Performance AXI3 Slave ports
High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
Low-bandwidth peripheral controller: SPI, UART, CAN, I2C
Programmable from JTAG, Quad-SPI flash, and microSD card
Programmable logic equivalent to Artix-7 FPGA
630 KB of fast block RAM
4 clock management tiles, each with a phase-locked loop (PLL) and mixed-mode clock manager (MMCM)
220 DSP slices
On-chip analog-to-digital converter (XADC)

Table 4.2: PYNQ-Z1 SoC memory specifications [1].

512MB DDR3 with 16-bit bus each at 1050Mbps
16MB Quad-SPI Flash with factory programmed 48-bit globally unique EUI-48/64™ compatible identifier
microSD slot

Table 4.3: PYNQ-Z1 SoC USB and ethernet specifications [1].

Gigabit Ethernet PHY
USB-JTAG Programming circuitry
USB-UART bridge
USB OTG PHY (supports host only)

Table 4.4: PYNQ-Z1 SoC audio and video specifications [1].

HDMI sink port (input)
HDMI source port (output)
Microphone with PDM interface
PWM driven mono audio output with 3.5mm jack

Table 4.5: PYNQ-Z1 SoC peripherals [1].

4 push-buttons
2 slide switches
2 RGB LEDs
Two standard Pmod ports 16 Total FPGA I/O

4.1 Performance Analysis of YOLO

We first evaluate the YOLO models used in the framework in terms of time taken to process an input image.

4.1.1 Tiny YOLO v3

Figure 4.1 shows the runtime console screenshot of Tiny YOLO v3; it clearly shows the 23 layers and the corresponding number of filters, the size of filters, input, and output sizes. The inference time is for an input image is 17690.17 milliseconds. Tiny YOLO v3 is a combination of various versions of YOLO and is a smaller, faster, and better model for constrained platforms. The data sets used are a combination of COCO and PASCAL VOC, like the earlier versions of YOLO, and this uses Darknet framework [20].

layer	filters	size	input				output			
0 conv	16	3 x 3 / 1	224 x	224 x	3	->	224 x	224 x	16	0.043 BF
1 max		2 x 2 / 2	224 x	224 x	16	->	112 x	112 x	16	0.001 BF
2 conv	32	3 x 3 / 1	112 x	112 x	16	->	112 x	112 x	32	0.116 BF
3 max		2 x 2 / 2	112 x	112 x	32	->	56 x	56 x	32	0.000 BF
4 conv	64	3 x 3 / 1	56 x	56 x	32	->	56 x	56 x	64	0.116 BF
5 max		2 x 2 / 2	56 x	56 x	64	->	28 x	28 x	64	0.000 BF
6 conv	128	3 x 3 / 1	28 x	28 x	64	->	28 x	28 x	128	0.116 BF
7 max		2 x 2 / 2	28 x	28 x	128	->	14 x	14 x	128	0.000 BF
8 conv	256	3 x 3 / 1	14 x	14 x	128	->	14 x	14 x	256	0.116 BF
9 max		2 x 2 / 2	14 x	14 x	256	->	7 x	7 x	256	0.000 BF
10 conv	512	3 x 3 / 1	7 x	7 x	256	->	7 x	7 x	512	0.116 BF
11 max		2 x 2 / 1	7 x	7 x	512	->	7 x	7 x	512	0.000 BF
12 conv	1024	3 x 3 / 1	7 x	7 x	512	->	7 x	7 x	1024	0.462 BF
13 conv	256	1 x 1 / 1	7 x	7 x	1024	->	7 x	7 x	256	0.026 BF
14 conv	512	3 x 3 / 1	7 x	7 x	256	->	7 x	7 x	512	0.116 BF
15 conv	255	1 x 1 / 1	7 x	7 x	512	->	7 x	7 x	255	0.013 BF
16 yolo										
17 route	13									
18 conv	128	1 x 1 / 1	7 x	7 x	256	->	7 x	7 x	128	0.003 BF
19 upsample		2x	7 x	7 x	128	->	14 x	14 x	128	
20 route	19 8									
21 conv	256	3 x 3 / 1	14 x	14 x	384	->	14 x	14 x	256	0.347 BF
22 conv	255	1 x 1 / 1	14 x	14 x	256	->	14 x	14 x	255	0.026 BF
23 yolo										
Total BFLOPS 1.615										
Loading weights from yolov3-tiny.weights...										

Figure 4.1: Screenshot of Tiny YOLO v3 runtime execution

4.1.2 Tinier YOLO

A modified version of Tiny YOLO called Tinier YOLO was developed by Xilinx for PYNQ Z1/Z2 and Ultra 96 [3]. As shown in Figure 4.2, it has 10 layers and uses 1-bit weights and 3-bit activation for middle layers. The layers in the middle, in the color pink, are run on the programmable logic of the PYNQ board, while the first and last layers are executed on Python. The inference time is for an input image is 1.39 seconds [49].

The layers are initialized on the Darknet framework, and Figure 4.3 shows the input on the top and output image at the bottom. We can see the bounding boxes around the predicted

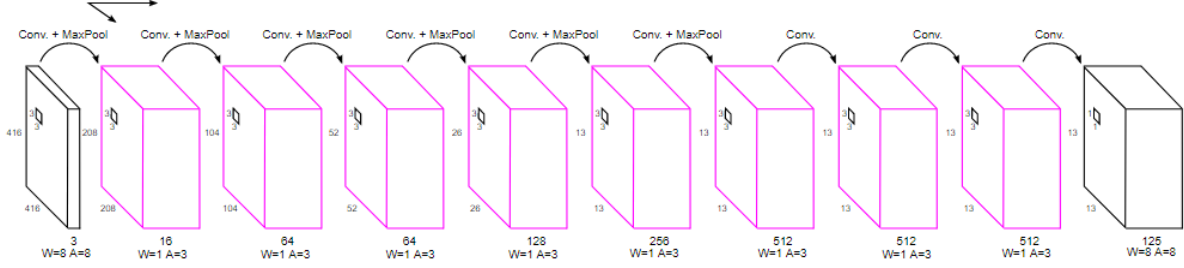


Figure 4.2: Tinier YOLO layers. (Reprinted from [50]. BSD 3-Clause License. Used with Permissions)

object and the prediction class. This has been trained on the PASCAL VOC dataset, so twenty classes of objects can be recognized by this object detection algorithm. The first layer has not been quantized. Thus it needs to be run on python while the layers followed are executed by the Hardware Accelerator of the board. The last layer is responsible for drawing the bounding boxes and is also run in Darknet with python bindings. The hardware in this context is the programmable logic, and the software is python on the PYNQ board.

Figure 4.4 shows the performance analysis in terms of latency and throughput. The first graph shows the Hardware vs. Software performance on the y-axis and execution time in milliseconds on the x-axis. The second graph shows the number of operations that are performed on hardware and software. The x-axis shows the number of million operations per second (MOPS) on the x-axis, and they are compared in terms of hardware and software. As we can see, there are more operations performed on hardware than software. The third graph shows how many MOPS per second does the hardware and the software execute to analyze performance. And even though the number of operations on hardware is higher, it consumes less execution time than software.

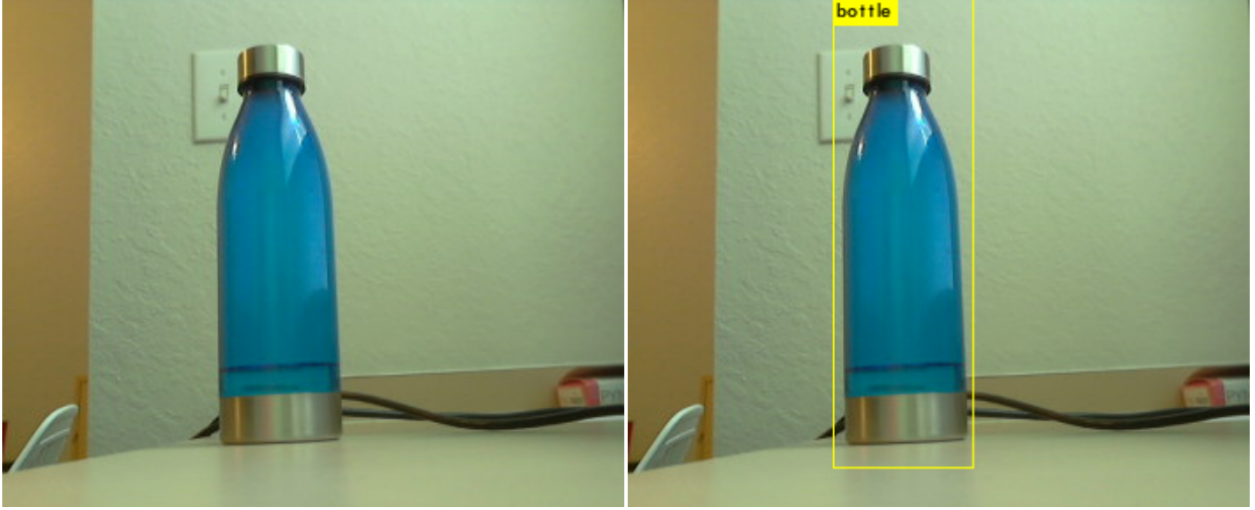


Figure 4.3: Tinier YOLO demo input and output image [3].

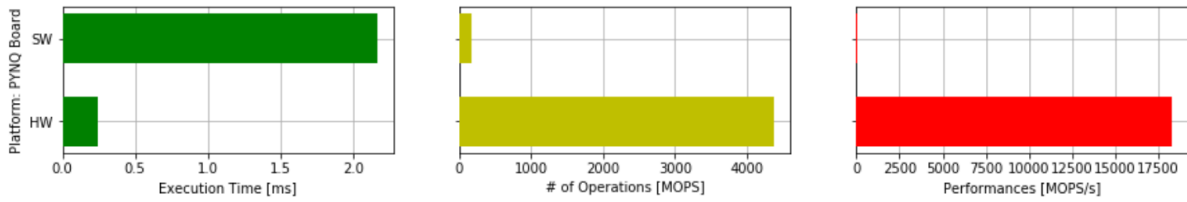


Figure 4.4: Performance analysis of Tinier YOLO [3].

4.2 Evaluating BNN (Binarized Neural Network)

We ran a BNN (Binarized Neural Network), which has been adapted from [25]. This has 1-bit weights and 1-bit activation and has been inspired by VGG-16. It has 6 convolutional layers, 3 max pool layers, and 3 fully connected layers. This is a classifier network which runs inference on MNIST, CIFAR-10, and SVHN datasets. The inference time for an image is 0.057 seconds [51].

4.3 Evaluating the Performance of Framework

The communication over sockets helps us parallel process input images over multiple Worker nodes seamlessly while adding minimal overhead to the system.

4.3.1 Model 1

To test the framework, we ran Model 1 with Tinier YOLO and Tiny YOLO v3 over the COCO-2017 test dataset. We also ran it over a custom data set of 50 images, which were captured in real-time with a USB camera attached to the PYNQ board. The results are reported in Table 4.6, which is the average time in seconds each Worker node has taken to process one image.

Table 4.6: Analysis of Model 1 (in seconds)

Time taken for	COCO dataset (in seconds)	custom dataset (in seconds)
Tinier YOLO	33.96	34.32
Tiny YOLO v3	50.54	50.87

4.3.2 Model 2

Model 2 has been evaluated on multiple object detection algorithms, as mentioned above. In this instance, the object detection algorithm used is Tinier YOLO [3]. The Worker node first receives the size of the image, and then the input image captured by the Head node with a USB camera. Then the object detection algorithm is run on this image, and the confidence of the prediction is obtained. The time it has taken for running the algorithm, Tinier YOLO takes 2.30 seconds on this one image on average. The result is sent to the Head node, which is a combination of class and probability of prediction. The image (one like the bottom image of Figure 4.3) is also sent to the Head node to store and display. A timed analysis of Model 2 is shown in Table 4.7, with the average time taken for each image in seconds. The framework has been evaluated on COCO test dataset and a custom dataset. The custom dataset refers to the images captured by the Head node with the help of a USB camera. From Table 4.7 and the section above where we evaluated the YOLO algorithm, we can see that the time taken for the object detection algorithm without the framework

and each Worker node in the framework are comparably close. This leads to the conclusion that our framework does not add any latency. From comparing Tables 4.6 and 4.7, we can see that the overhead the frameworks has reduced. Our framework in the Model 2 considerably less extra time, and in fact by using a distributed object detection, we can scale the time down by the number of Worker nodes in the network. Here, we can see that by adding more Worker nodes to the framework we can scale the time taken for each input image.

Table 4.7: Analysis of Model 2 (in seconds)

Time taken for	COCO dataset (in seconds)	custom dataset (in seconds)
Tinier YOLO	2.30	2.32
Tiny YOLO v3	17.69	18.23

4.3.3 Framework Overhead

In the above sections, we have analysed the performance of various object detection algorithms, and each of the developed models. The Model 2 has significantly reduced the time taken by the framework for processing an image adding negligible overhead. To see how much time the frameworks takes, we have compared the time taken by the Head node for each Worker node in Table 4.8. In this table we see the time taken by the Head node and the time taken by a Worker node to process an image. The time taken by the Head node is calculated after the image is being clicked to until the result is received to until the result is received from the Worker node. The time for the Worker node is the time it takes to receive an image from the Head node to sending the results back to Head node after running the object detection algorithm.

Table 4.8: Analysis of framework in Model 2 (in seconds)

Time taken for	Head node (in seconds)	One Worker node (in seconds)
Tiny YOLO v3	24.36	23.50
Tinier YOLO	2.31	2.24
BNN	0.09	0.05

Table 4.9: Analysis of the distributed framework for multiple Worker nodes to run inference on one input image (in seconds)

Time taken to process	1 Worker node	2 Worker nodes	3 Worker nodes
Tiny YOLO v3	24.36	15.54	8.32
Tinier YOLO	2.31	1.23	0.84
BNN	0.091	0.068	0.052

Table 4.9 refers to the average time taken to process one image by one Worker node, two Worker nodes and three Worker nodes respectively. It shows how each node added to the framework reduces the time taken to process an image. We can deduce from Figure 4.5, the correlation between time taken to process an image and the number of Worker nodes in the framework. It shows the time taken on an average to run inference on one image and how it decreases as we put it in our Distributed Load Balancing (DLB) framework. We can observe that by adding more Worker nodes, we can optimize the time taken to process an image.

$$Frames\ per\ second = \frac{Number\ of\ Worker\ Nodes}{Computational\ Latency + Communication\ Latency} \quad (4.1)$$

Table 4.10: Analysis of the Distributed framework for multiple Worker nodes to run inference on one input image (in frames per second).

Frames per second	1 Worker node	2 Worker nodes	3 Worker nodes
Tiny YOLO v3	0.041	0.064	0.120
Tinier YOLO	0.43	0.81	1.19
BNN	11.1	14.7	19.23

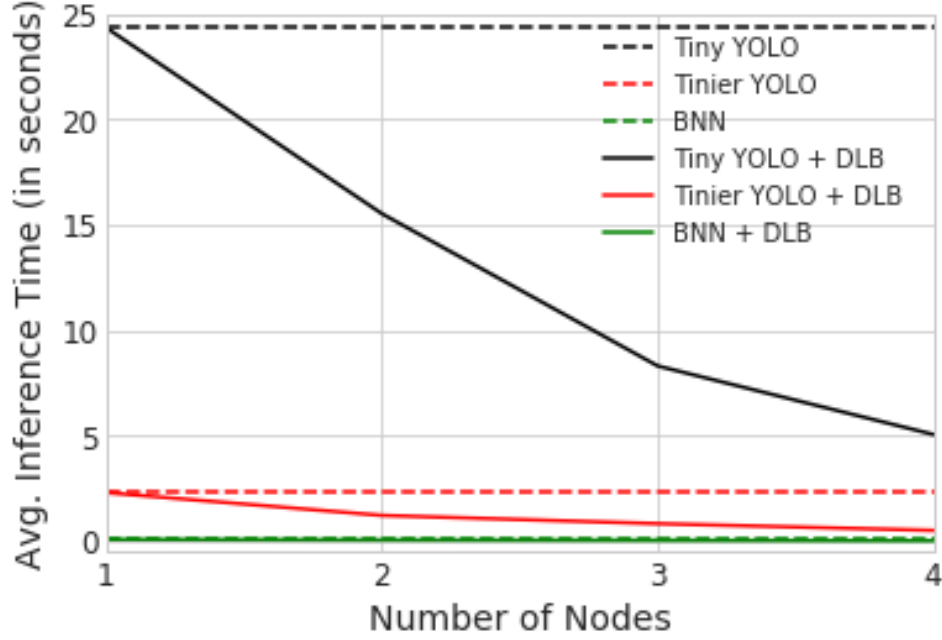


Figure 4.5: Number of nodes vs. average inference time. DLB stands for Distributed Load Balancing.

Using Equation 4.1, we can calculate the number of frames our framework can process in one second. The number of frames per second in the proposed framework is proportional to the number of Worker nodes in the network. It takes a marginal amount of time to run the image detection on this distributed system framework. The number of frames per second the proposed framework can process is dependent on the number of nodes in the network as shown in Table 4.10. According to the numbers seen above, we can scale the network and increase the rate of image processing by a factor that is equivalent to the number of nodes in the framework. As each Worker node takes the same amount of time uniformly to process an image and the Head node keeps distributing the images, we can apply this framework to any real-time distributed system application. The Head node is equipped to handle the Worker nodes independently and dynamically, which allows for adding a new Worker node to the framework.

4.4 Chapter Summary

Various object detection and classification algorithms, mainly variants of YOLO and BNN algorithms, have been run on the PYNQ board for object detection. Model 1, which is a preliminary version of the framework, has been analyzed, and results have been discussed. To overcome the overhead, and the various limitations of Model 1, we developed Model 2. This Model 2 consumes less time for each image as the additional delay caused by the framework itself has reduced. We have reduced the latency in Model 2, and the experimental results have been tabulated. We developed a framework that can be scaled to add Worker nodes to process more input images in parallel.

Chapter 5: Conclusions and Future Work

As we ran the Tinier YOLO on both the models and observe, there is virtually no latency involved in transfer the data via a socket connection. With a Python-based programming interface, the framework aims to combine the convenience of high-level abstraction with the speed of optimized FPGA implementation. The framework can be scaled by adding more Worker nodes to achieve a parallel and seamlessly distributed object detection system. A software-down development is needed to accommodate the processor driven and heterogeneous growth of FPGAs. We are currently working on a new framework for a multi-FPGA cluster with efficient load balancing. There are several platforms this framework can be tailored to, and optimizing the time taken for the neural nets to run could be done. The end goal is to be able to exploit all the resources of the APSoC (All Programmable System-on-Chip) and have that framework can be applied to real-time scenarios for various application instances. The integration of real-time instances into the framework would be based on the high-level abstraction with the speed of optimized FPGA implementation.

5.1 Future Enhancements

The following are suggestions for enhancing the proposed framework for a distributed system on FPGAs for running deep neural networks.

- To decrease the latency of image detection algorithm for real-time applications.
- Making the system more robust by implementing the Head node to check the status of all

the available Worker nodes and take necessary actions like sending the notifications if Worker nodes are down.

- Extending the testing from two-Worker nodes to test function to multi-Worker node test function.
- Tracking the time taken for the image under the processing state and attempt to process the image if it was put under processing state for more than the expected time.
- Finding ways to increase the efficiency by optimizing the other resources which occupy more space in memory.
- Implement a more dynamic Head node for better control over the data flow. To reduce the overhead of communication and make it more robust for adding a Worker node dynamically.
- To explore other SoCs as platforms for this proposed framework. Applying the current framework to a different domain like handwriting detection.
- And also to tailor the tinier YOLO algorithm for better resource utilization.

References

- [1] PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC. <https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/>. Accessed: 2020-06-04.
- [2] F. Kästner, B. Janßen, F. Kautz, M. Hübner, and G. Corradi. Hardware/Software Codesign for Convolutional Neural Networks Exploiting Dynamic Partial Reconfiguration on PYNQ. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 154–161, 2018.
- [3] Xilinx/QNN-MO-PYNQ. <https://github.com/Xilinx/QNN-MO-PYNQ/tree/master/qnn>. Accessed: 2020-05-23.
- [4] 2016 Published by Statista Research Department, Nov 27. <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>. Accessed: 2020-05-23.
- [5] Internet of things. https://en.wikipedia.org/wiki/Internet_of_things#Intelligenceqnn. Accessed: 2020-05-23.
- [6] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang. A Survey on the Edge Computing for the Internet of Things. *IEEE Access*, 6:6900–6919, 2018.
- [7] What is edge computing and why it matters. <https://www.networkworld.com/article/3224893/what-is-edge-computing-and-how-it-s-changing-the-network.html>. Accessed: 2020-05-23.

- [8] File:Edge computing infrastructure. https://commons.wikimedia.org/wiki/File:Edge_computing_infrastructure.png. Accessed: 2020-06-01.
- [9] R. Haridas and R. L. Jyothi. Convolutional Neural Networks: A Comprehensive Survey. *International Journal of Applied Engineering Research*, 14(3), 2019.
- [10] A. Khan, A. Sohail, U. Zahoor, and A. Qureshi. A Survey of the Recent Architectures of Deep Convolutional Neural Networks, 2019.
- [11] V. Tra, K. Jaeyoung, S. Khan, and K. Jong-Myon. Bearing Fault Diagnosis under Variable Speed Using Convolutional Neural Networks and the Stochastic Diagonal Levenberg-Marquardt Algorithm. *Sensors*, 17:2834, 12 2017.
- [12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.
- [14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.
- [15] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. 2014.
- [16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

- [17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.
- [18] J. Redmon and A. Farhadi. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.
- [19] J. Redmon and A. Farhadi. YOLOv3: An Incremental Improvement, 2018.
- [20] H. Zheyu, C. Huang, L. Wei, L. Li, and A. Guo. TF-YOLO: An Improved Incremental Network for Real-Time Object Detection. *Applied Sciences*, 9:3225, 08 2019.
- [21] GstInference with Tinyyolov3 architecture. https://developer.ridgerun.com/wiki/index.php?title=GstInference/Supported_architectures/TinyYoloV3. Accessed: 2020-05-23.
- [22] Darknet. <https://github.com/pjreddie/darknet/blob/master/data/dog.jpg>. Accessed: 2020-05-23.
- [23] W. Fang, L. Wang, and P. Ren. Tinier-YOLO: A real-time object detection method for constrained environments. *IEEE Access*, 8:1935–1944, 2020.
- [24] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations, 2016.
- [25] Y. Umuroglu, Nicholas J. J. F, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17*, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery.

- [26] T. B. Preußer, G. Gambardella, N. Fraser, and M. Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 833–838, 2018.
- [27] J. Ren, Y. Guo, D. Zhang, Q. Liu, and Y. Zhang. Distributed and Efficient Object Detection in Edge Computing: Challenges and Solutions. *IEEE Network*, 32(6):137–143, 2018.
- [28] Q. Liu, L. Cheng, T. Ozcelebi, J. Murphy, and J. Lukkien. Deep Reinforcement Learning for IoT Network Dynamic Clustering in Edge Computing. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 600–603, 2019.
- [29] Z. Zhang, T. Song, L. Lin, Y. Hua, X. He, Z. Xue, R. ma, and journal=IEEE Transactions on Big Data H. Guan. Towards Ubiquitous Intelligent Computing: Heterogeneous Distributed Deep Neural Networks. pages 1–1, 2018.
- [30] E. Koromilas, I. Stamelos, C. Kachris, and D. Soudris. Spark acceleration on FPGAs: A use case on machine learning in PYNQ. In *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, 2017.
- [31] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.
- [32] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W. Hwu, and D. Chen. FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

- [33] A. Sharma, V. Singh, and A. Rani. Implementation of CNN on Zynq based FPGA for Real-time Object Detection. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7, 2019.
- [34] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng. Scalable learning for object detection with GPU hardware. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4287–4293, 2009.
- [35] Jetson Nano: Deep Learning Inference Benchmarks. <https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks>. Accessed: 2020-05-23.
- [36] B. Janßen, P. Zimprich, and M. Hübner. A dynamic partial reconfigurable overlay concept for PYNQ. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.
- [37] E. Wang, J. J. Davis, and P. Y. K. Cheung. A PYNQ-Based Framework for Rapid CNN Prototyping. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 223–223, 2018.
- [38] A. Jahanshahi. TinyCNN: A Tiny Modular CNN Accelerator for Embedded FPGA, 2019.
- [39] Load Balancing with HAProxy. <https://blog.stackpath.com/load-balancing-haproxy/>. Accessed: 2020-05-23.
- [40] An Introduction to HAProxy and Load Balancing Concepts. <https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts>. Accessed: 2020-05-23.

- [41] An Introduction to HAProxy and Load Balancing Concepts. <http://www.haproxy.org/#desc>. Accessed: 2020-05-23.
- [42] What is a Socket? https://www.tutorialspoint.com/unix_sockets/what_is_socket.htm. Accessed: 2020-05-23.
- [43] Sockets Tutorial. <https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html>. Accessed: 2020-05-23.
- [44] M. Cheng, Z. Zhang, W. Lin, and P. Torr. BING: Binarized Normed Gradients for Objectness Estimation at 300fps. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3286–3293, 2014.
- [45] C. Cheng and C. Bouganis. An FPGA-based object detector with dynamic workload balancing. In *2011 International Conference on Field-Programmable Technology*, pages 1–4, 2011.
- [46] Robust real-time face recognition. In *2013 Africon*, pages 1–5, 2013.
- [47] P. Jokic, S. Emery, and L. Benini. BinaryEye: A 20 kfps Streaming Camera System on FPGA with Real-Time On-Device Image Recognition Using Binary Neural Networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–7, 2018.
- [48] C. Garry and D. Molloy. A Software/Hardware Co-Design Framework for the ‘Internet of Eyes’. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 133–138, 2019.
- [49] V. Y. Çambay, A. Uçar, and M. A. Arserim. Object Detection on FPGAs and GPUs by Using Accelerated Deep Learning. In *2019 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–5, 2019.

- [50] Xilinx/QNN-MO-PYNQ. <https://github.com/Xilinx/QNN-MO-PYNQ/blob/master/notebooks/Tinier-{\YOLO}-topology.svg>. Accessed: 2020-05-23.
- [51] Xilinx/BNN-PYNQ. <https://github.com/Xilinx/BNN-PYNQ>. Accessed: 2020-07-22.


Appendix A: Copyright Permissions

The permission below is for the use of Figure 2.1, from Chapter 1. It is licensed under the creative commons attribution-share alike 4.0 international license.

Licensing [\[edit \]](#)

I, the copyright holder of this work, hereby publish it under the following license:

This file is licensed under the [Creative Commons Attribution-Share Alike 4.0 International](#) license.



SOME RIGHTS RESERVED

You are free:

- **to share** – to copy, distribute and transmit the work
- **to remix** – to adapt the work

Under the following conditions:

- **attribution** – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **share alike** – If you remix, transform, or build upon the material, you must distribute your contributions under the [same or compatible license](#) as the original.

The permission below is for the use of Figure 2.2, from Chapter 2. It is an open sourced article distributed under the terms and conditions of creative commons attribution license.

PMC full text: [Sensors \(Basel\). 2017 Dec; 17\(12\): 2834.](#)

Published online 2017 Dec 6. doi: [10.3390/s17122834](#)

[Copyright/License](#) [Request permission to reuse](#)

[Copyright](#) © 2017 by the authors.

Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).

The permission below is for the use of Figure 2.3, from Chapter 2. It is public domain licensed.

RidgeRun website material copyright permission Inbox x



Diego Chaverri

Hi Lakshmikavya, I'm CC'ing Todd Fischer, RidgeRun's Head of Engineering. You can discuss with Todd regarding your copyright permission request. Regards



kavya kalyanam

Hello Todd, I am a Master's student and I would like to use one of the images from your website in my thesis. I have adapted some of the content and would like



Todd Fischer <todd.fischer@ridgerun.com>

to Miguel, me, Diego, Todd, Clark, support ▾

Hi Kavya,

The original image is from

<https://github.com/pjreddie/darknet/blob/master/data/dog.jpg>

Which is part of darknet. Darknet has a public domain license.

<https://github.com/pjreddie/darknet/blob/master/LICENSE>

Which uses wording that is a bit unusual:


0. Darknet is public domain.
1. Do whatever you want with it.
2. Stop emailing me about it!

So, do whatever you want with it. You can credit darknet.




Todd Fischer
VP Engineering
RidgeRun

The permission below is for the use of Figure 2.3, from Chapter 2. It is public domain licensed.

Branch: master **darknet / LICENSE** Find file Copy path

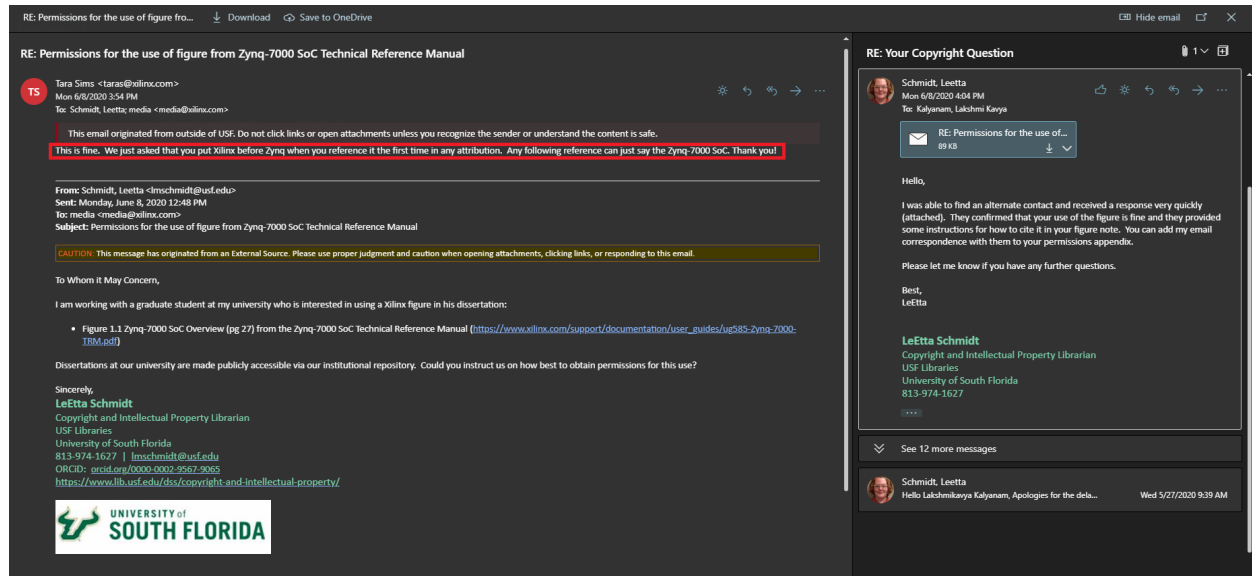
 **pjreddie** Update LICENSE e6b3006 on Jul 29, 2016

1 contributor



12 lines (10 sloc) 515 Bytes Raw Blame History   

```
1          YOLO LICENSE
2          Version 2, July 29 2016
3
4  THIS SOFTWARE LICENSE IS PROVIDED "ALL CAPS" SO THAT YOU KNOW IT IS SUPER
5  SERIOUS AND YOU DON'T MESS AROUND WITH COPYRIGHT LAW BECAUSE YOU WILL GET IN
6  TROUBLE HERE ARE SOME OTHER BUZZWORDS COMMONLY IN THESE THINGS WARRANTIES
7  LIABILITY CONTRACT TORT LIABLE CLAIMS RESTRICTION MERCHANTABILITY. NOW HERE'S
8  THE REAL LICENSE:
9
10 0. Darknet is public domain.
11 1. Do whatever you want with it.
12 2. Stop emailing me about it!
```


The permission below is for the use of Figure 2.4, from Chapter 2. They have specified the reference format.



The permission below is for the use of Figure 2.6, Figure 2.7, Figure 2.8, Figure 2.9, from Chapter 2. The format requirements for reference is specified below.



[Home](#) [Help](#) [Email Support](#) [Sign in](#) [Create Account](#)



Low Power FPGA-SoC Design Techniques for CNN-based Object Detection Accelerator

Conference Proceedings:
2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)

Author: Heekyung Kim
Publisher: IEEE
Date: Oct. 2019

Copyright © 2019, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

- 1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
- 2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
- 3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

- 1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
- 2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
- 3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

[BACK](#) [CLOSE WINDOW](#)

The permission below is for the use of Figure 4.2, from Chapter 4. The permissive licence is under BSD 3-Clause "New" or "Revised" License.

Xilinx / QNN-MO-PYNQ
Unwatch 21
Star 158
Fork 76

Code
Issues 12
Pull requests 0
Actions
Projects 0
Wiki
Security 0
Insights

Branch: master
QNN-MO-PYNQ / LICENSE
Find file
Copy path

Xilinx/QNN-MO-PYNQ is licensed under the **BSD 3-Clause "New" or "Revised" License**
A permissive license similar to the BSD 2-Clause License, but with a 3rd clause that prohibits others from using the name of the project or its contributors to promote derived products without written consent.

Permissions	Limitations	Conditions
<ul style="list-style-type: none"> ✓ Commercial use ✓ Modification ✓ Distribution ✓ Private use 	<ul style="list-style-type: none"> ✗ Liability ✗ Warranty 	<ul style="list-style-type: none"> ① License and copyright notice

This is not legal advice. [Learn more about repository licenses.](#)

Initial commit
0dabe2a on Feb 13, 2018

1 contributor

29 lines (23 sloc) | 1.47 KB
Raw
Blame
History

```

1  BSD 3-Clause License
2
3  Copyright (c) 2018, Xilinx
4  All rights reserved.
5
6  Redistribution and use in source and binary forms, with or without
7  modification, are permitted provided that the following conditions are met:
8
9  * Redistributions of source code must retain the above copyright notice, this
10     list of conditions and the following disclaimer.
11
12  * Redistributions in binary form must reproduce the above copyright notice,
13     this list of conditions and the following disclaimer in the documentation
14     and/or other materials provided with the distribution.
15
16  * Neither the name of the copyright holder nor the names of its
17     contributors may be used to endorse or promote products derived from
18     this software without specific prior written permission.
19
20  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
21  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
22  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
23  DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
24  FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
25  DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
26  SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
27  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
28  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
29  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

```