Deep Convolutional Neural Network Based Object Detection Inference Acceleration Using FPGA

Solomon Negussie Tesema

THESE DE DOCTORAT DE L'ETABLISSEMENT UNIVERSITE BOURGOGNE FRANCHE-COMTE

PREPAREE A Université de Bourgogne

Ecole doctorale n°37

L'Ecole Doctorale de Sciences Physiques pour l'Ingénieur et Microtechniques (ED SPIM)

Doctorat Instrumentation et informatique de l'image

Par

Mr. TESEMA Solomon Negussie

Deep Convolutional Neural Network Based Object Detection Inference Acceleration Using FPGA

Thèse présentée et soutenue à Dijon, le 09 Septembre 2022

Composition du Jury :

Professeure	Fan YANG	Professeure de Université de Bourgogne	Président
Professeur	Camel TANOUGAST	Professeur de Université de Lorraine	Rapporteur
MCF-HDR	Samy MEFTALI	MCF-HDR Université de Lille	Rapporteur
Professeur	El-Bay BOURENNANE	Professeur de Université de Bourgogne	Directeur de thèse

école doctorale **sciences pour l'ingénieur et microtechniques** SPIM

Titre : Accélération de l'inférence de la détection d'objets basée sur un réseau neuronal convolutif profond à l'aide de FPGA

Mots clés : Apprentissage profond, CNN, Accélérateur matériel, Détection d'objets, FPGA

Résumé : La détection d'objets est l'un des domaines de recherche en vision par ordinateur les plus difficiles et pourtant essentiels. Elle consiste à étiqueter et à localiser tous les objets d'intérêt connus sur une image d'entrée en utilisant des boîtes de délimitation rectangulaires bien ajustées autour des objets. La détection d'objets, après avoir connu plusieurs évolutions et progressions, repose aujourd'hui sur les succès des réseaux de classification d'images basés sur des réseaux de neurones convolutifs profonds. Toutefois, à mesure que la profondeur et la complexité des réseaux neuronaux convolutifs augmentent, la vitesse de détection diminue et la précision augmente. Malheureusement, la plupart des applications de vision par ordinateur, comme le suivi d'objets en temps réel sur un système embarqué, nécessitent une détection d'objets légère, rapide et précise. Par conséquent, l'accélération de la détection d'objets est devenue un domaine de recherche très dynamique, avec beaucoup d'attention accordée à l'accélération basée sur le FPGA en raison de la haute efficacité énergétique, de la grande largeur de bande de données et de la programmabilité flexible du FPGA. Cette thèse de doctorat propose d'améliorer progressivement les modèles de détection d'objets en reconvertissant les détecteurs d'objets connus existants en modèles plus légers, plus précis et plus rapides. Nos modèles atteignent une précision comparable, tout en étant légers et rapides, à celles de obtenues par les meilleurs détecteurs de l'état de l'art. Nous proposons et mettons également en œuvre l'accélération de l'inférence de la détection d'objets à l'aide de cartes FPGA de différentes capacités et ressources. Nous nous concentrons sur les implémentations d'accélération d'inférence à haute efficacité énergétique et en ressources, tout en préservant les performances de précision du détecteur d'objets. Enfin, nous présentons diverses contributions auxiliaires telles qu'une technique de génération ou d'augmentation d'images synthétiques très significative pour l'entraînement d'un détecteur d'objets, ce qui est essentiel pour obtenir un détecteur d'objets performant. Dans l'ensemble, notre travail dans cette thèse comporte deux grandes parties : la conception et la mise en œuvre de modèles de détection d'objets légers et précis basés sur le CPU et le GPU et la mise en œuvre d'une accélération d'inférence de détection d'objets à haut débit, à faible consommation d'énergie et de ressources sur un FPGA.

Title: Deep Convolutional Neural Network Based Object Detection Inference Acceleration Using FPGA

Keywords: Deep Learning, CNN, Hardware Accelerator, Object Detection, FPGA

Abstract: Object detection is one of the most challenging yet essential computer vision research areas. It means labeling and localizing all known objects of interest on an input image using tightly fit rectangular bounding boxes Our models achieve a comparable accuracy while being lightweight and faster compared with some of the top state-of-the-art detectors. We also propose and implement object detection inference acceleration using FPGA

around the objects. Object detection, having passed through several evolutions and progressions, nowadays relies on the successes of image classification networks based on deep convolutional neural networks. However, as the depth and complication of convolutional neural networks increased, detection speed reduced, and accuracy increased. Unfortunately, most computer vision applications, such as real-time object tracking on an embedded system, requires lightweight, fast and accurate object detection. As a result, object detection acceleration has become a hot research area, with much attention given to FPGA-based acceleration due to FPGA's high-energy efficiency, high-data bandwidth, and flexible programmability.

This Ph.D. dissertation proposes incrementally improving object detection models by repurposing existing well-known object detectors into lighter, more accurate, and faster models. boards of different capacities and resources. We focus on high resource and energy-efficient inference acceleration implementations while preserving the object detector's accuracy performance. Last but not least, we present various auxiliary contributions such as a highly significant synthetic image generation or augmentation technique for training an object detector which is critical for achieving a high-performance object detector. Overall, our work in this thesis has two parts: designing and implementing lightweight and accurate CPU and GPU-based object detection models and implementing high-throughput, energy, and resource-efficient object detection inference acceleration on an FPGA.

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTE

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon

Dedicated to my heavenly late sister Mixuye!

Acknowledgments

First, I would like to express my sincere gratitude to my thesis supervisor Prof. El-Bay Bourennane for his unbounded patience, guidance, and support. I am forever indebted to his magnanimous personality, advice, and perpetual hardworking nature that genuinely made this thesis work a success story. I extend my heartfelt gratitude to the member of the thesis defense jury Professor Fan Yang, Professor Camel Tanougast, and Associate Professor Samey Meftali.

I would also like to extend my heartfelt gratitude to the Ethiopian Ministry of Higher Education and Campus France for sponsoring my study. I would also like to thank the University of Burgundy and the ImViA laboratory for accepting me as a Ph.D. student and offering me teaching jobs, which helped me improve my scientific research and teaching skills.

Special thanks to my friends and colleagues for the fun times, discussions, and interactions we had. Though it might be challenging to list all your names here, I must mention a few of you: Kibrom, Yousef, Rabid, Juhlyn, Ichraf, and Tedy — please accept my sincerest thank you.

It will be an unforgivable mistake not to thank my loving, caring, and beautiful wife, Lelisie Esaw Abosse, who made this study bearable. You are and will forever be the most invaluable being I genuinely adore and hold on to dearly. So thank you very much, darling, for making my study and life enjoyable. I also extend my heartfelt thank you to my parents, sisters, and brothers for staying strong and making me feel strong even though we had to go through the ordeal of our unjust loss of our beloved Mixuye to the cruel brutality of the world we live in and the evils surrounding us.

Acronyms

AI Artificial Intelligence

ANN Artificial Neural Networks

BN Batch Normalization

CNN Convolutional Neural Networks

CPU Central Processing Unit

DL Deep Learning

FPGA Field Programmable Gate Array

GPU Graphical Processing Unit

HOG Histogram of oriented gradients

ID Identifier

ML Machine Learning

MSE Mean Square Error

SVM Support Vector Machine

Terms Used

DDGNet Dense Detection Grid Network is an object detection model created by restructuring the well-known YOLOv2 model's detection head using binary encoding instead of one-hot encoding. It is our first modification to YOLOv2 in our quest to create a lightweight and fast detector which also generic or large multiclass.

DenseYOLO Is our second version object detector based on YOLOv2.

IoU stands for intersection over union and is a prime method for identifying if two overlapping bounding boxes belong to the same or different objects. It is the measure or ratio of an area of an intersection between two boxes to the area of the union of the two boxes.

mAP stands for mean average precision and is used to tell the accuracy performance of an object detector or image classification network.

MNIST is a well-known database of handwritten digits (0 through 9).

MultiGridDet Is the third incremental improvement of our object detector and is based on YOLOv3 and DenseYOLO.

SIFT is a computer vision algorithm to detect, describe, and match local features in images, invented by David Lowe in 1999.

ThreeWayNMS is custom non-max-suppression for filtering redundant and overlapping object detection bounding boxes. It is more flexible than the traditional non-max-suppression, which uses only a single threshold, whereas ThreeWayNMS uses two thresholds and area-based neighborhood clustering.

YOLO You Only Look Once, or YOLO, is a pioneer and state-of-the-art one-stage object detection model based on fully convolutional neural networks. It is the first of the successive incrementally improved versions released by the original authors of the object detector network and is usually known as YOLOv1, short for YOLO version one.

YOLOv2 YOLO Version 2.

YOLOv3 YOLO Version 3.

Contents

List of Figures

List of Tables

Chapter

1 Introduction

Chapter content

Introduction

In this chapter, we will provide the overall theme of this thesis work. We will discuss our work's motivation, objectives, and significant contributions. In the end, we present the structure of our dissertation document with an executive summary of each subsequent chapter.

1.1 Motivation

We humans can perceive any visible object in front of us effortlessly and with great detail in just a single glance or blink of an eye. These include identifying where an object part begins and ends, its color, size, and the number of objects, to list a few things we can do in a single glance. It is an age-old quest for researchers to mimic this natural ability of human beings using machines – after all, vision is one of the most valuable and dominant senses a human depends on to run his or her daily errands. If a machine can see and dependably understand what it sees without human assistance or intervention, that would undoubtedly be one of the most outstanding achievements in modern human history. Considering only the positive repercussion of this achievement, one can think of an infinite application that an intelligent machine can render to better our existence, such as surveillance and tracking, medical technology, environmental protection, text-to-audio conversion, and vice versa, uncrewed vehicles technology, to list the few. Though progress is still in the infant stage of understanding and mimicking human intelligence using computers, many breakthrough achievements have been achieved in this quest. Blind users can now explore photos by touch with Microsoft's Seeing AI [1]; AI machine has beaten humans in Image classification[2]; AI Beats Radiologists in Detecting Lung Cancer [3]; Two new planets discovered using artificial intelligence[4] — are success stories showcasing the potential of artificial intelligence. Moreover, AI successfully tackles the daunting email spam filtering task [5] and face recognition [6] functionality running on our favorite social media applications.

Even though these growths and phenomenal advancements of artificial intelligence, particularly computer vision, are admirable, the programming complexity, the vast computational resource requirement of the models, their expensive electrical power consumption, and lack of real-time operational speed are bottlenecks that we have to reckon with yet. Embedded systems

such as implantable medical chips, smart city camera systems for traffic control, wireless vision networks for farms, borders, personal property surveillance, uncrewed aerial or land vehicles, etc., are all resource-constrained platforms that could benefit from the advancement of AI. However, current AI applications are too big, high resource and power demanding, and also slow. As AI systems' accuracy increases and becomes dependable, research on making them lightweight, real-time, and less complex is getting traction and top priority. This fact brings us to the motivation of this thesis which is the design and implementation of a lightweight, fast and accurate generic object detection . Object detection is one of the most critical computer vision applications whose objective is to precisely locate all known objects on an image or a video frame. In this regard, we intend to contribute by proposing lightweight, fast, and accurate object detection and its hardware acceleration architecture suited for embedded systems using FPGA.

1.2 Statement of the Problem

Object detection one of the most critical computer vision applications. It has several real-world applications such as face detection and tracking, emotion recognition, pedestrian detection and tracking, property surveillance, video analysis, numerous military use, medical image analysis, and the list goes on. Most of these applications are needed in resource-constrained environments such as embedded systems where memory storage, battery consumption, and processor processing capacity are adversely low. For example, a path following robot, wireless sensor network of smart surveillance camera system running on a remote secure property, wearable patient monitoring and assistance system, et cetera are embedded systems that could benefit from object detection . Even if the deployment is on unconstrained resource environments, object detection should be real-time; after all, humans need computers for accuracy, speed, and ease of doing tasks.

However, the current trend in object detection focuses on boosting accuracy, with less regard for speed and used resources, by using deep learning techniques with several layers and millions of parameters. In a desktop or laptop computer system, the increasing speed and memory of graphical processing units, GPUs, along with readily available software frameworks for creating sophisticated object detectors, has made it easy for researchers to keep increasing the depth of their object detection model with anticipation of obtaining an accurate detector, of course at the cost of speed. Nowadays, finding a commendable balance between speed and accuracy while requiring fewer resources is an active computer vision research question. A framework called one-shot or one-stage object detection is often regarded as light, fast, and commendably accurate even-though usually less accurate than two-stage object detection. This thesis proposes ways to make one-shot object detection even faster, lighter, and competitively accurate against the two-stage object detectors. We implement our object detectors compare them against state-of-the-art detectors. We also implement a generic hardware accelerator for object detection based on our models using two low-cost FPGA boards.

1.3 Objectives and Major Contributions

This thesis work aims to design and implement deep learning-based lightweight, real-time, accurate, and generic object detection . We also aim to make object detection energy efficient and suitable for embedded systems, and hence we implement our lightweight and accurate object detector using FPGA for an increased inference speed. In general, the following are specific objectives and deliverables of our research work on object detection implementation using deep CNN and its inference acceleration using custom buit accelerators built on an FPGA:

- Design and implementation of CPU and GPU-based fast, accurate, and lightweight one-shot object detector. Here we propose three different deep convolutional neural network-based

object detection methods. All three models have one core concept: treating object detection as a single unified feedforward regression problem instead of the pipelined two-stage framework that relies on an underlying object region proposal network. The first model focuses on object class encoding, the second model proposes change on the use of anchor boxes in a one-stage object detector, and the third model incorporates multiscale detection for increased detection accuracy while maintaining a lightweight and high-detection speed.

- We design and implement hardware accelerators based on our object detection models on two FPGAs with different onboard resources. We compare our object acceleration with both other similar implementations and against our GPU-based implementation to investigate resource and power consumption and inference speed gain.

- We also propose an efficient and straightforward synthetic data augmentation technique for object detection. Training object detection requires an enormous amount of carefully annotated data, and obtaining or producing such data is a very tedious and cumbersome task. Instead, researchers rely on different image augmentation techniques. Our synthetic image generation algorithm proves very useful in training object detection.

Furthermore, Figure 1.1 summarizes our objective and major contributions in this Ph.D. work pictorially by grouping our core objectives and contributions into two categories. As seen from the figure, our first objective is to design and implement a new and better object detection network based on deep CNN by re-purposing an already existing well-performing model into a more lightweight, faster, and accurate detector. Under this objective, our significant contributions yield three new object detection models based on a one-stage object detection paradigm and are derived from YOLOv2 and YOLOv3, well-reputed object detection models. Moreover, we also devise a new and valuable synthetic data augmentation technique to supplement object detection training datasets. Our second core objective is to design and implement a faster, lighter, and resource and power-efficient hardware version of our object detectors. Under this objective, we achieved high resource and power efficiencies and more accurate and high-throughput object detection inference acceleration.

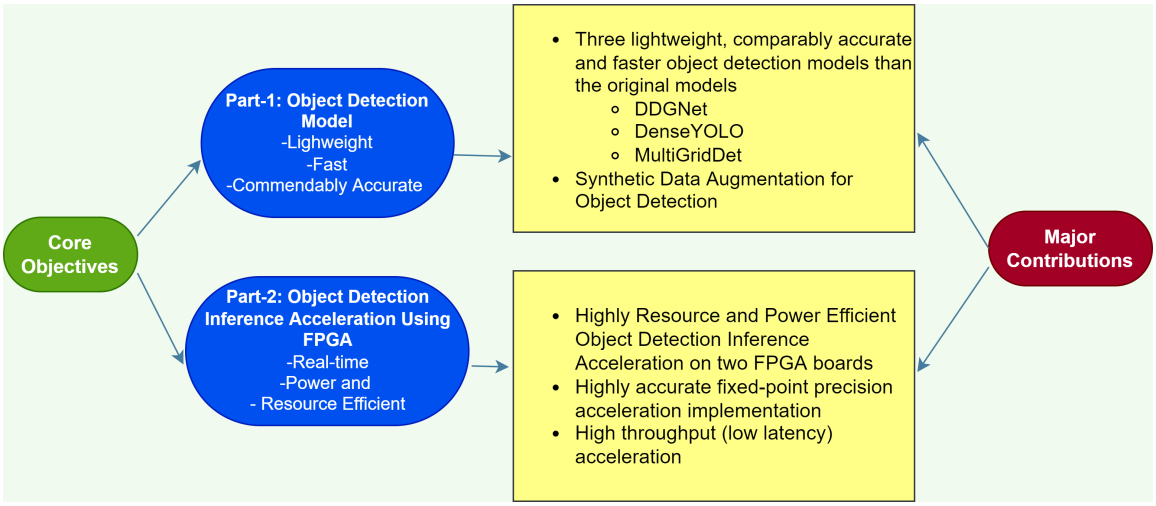


Figure 1.1: *Our objectives and major contributions*

1.4 Dissertation Structure

This thesis comprises nine chapters, including this chapter introductory chapter, where we stated our objectives and mission in this Ph.D. work. Below we summarize the focuses of each chapter

in a bulleted manner:

- Chapter 1: Introduction: presents the thesis motivation, objectives, problem statement, scopes, and limitations.
- Chapter 2: Overview of Basics of Deep Learning: introduces deep learning, particularly artificial neural networks and convolutional neural networks. Moreover, the chapter also tries to define some critical terms in machine learning. Finally, the chapter introduces the process of creating and training a deep learning model and its challenges.
- Chapter 3: Object detection and tracking: a study of current trends and state-of-the-art approaches. This chapter focuses on object detection and tracking, delves in detail into modern object detection trends and introduces some of the most well-known object detection models of the past decade. We also introduce object tracking and current object tracking implementation approaches.
- Chapter 4: FPGA for Object Detection Acceleration: presents why FPGA is used to accelerate object detection, reviews recent deep CNN acceleration using FPGA, some standard go-to approaches in implementing custom object acceleration, and finally presents the end-to-end implementation of hardware/software co-design tools and flows.
- Chapter 5: Lightweight Generic object detection using binary encoding— is the beginning of part two of this thesis document and presents our first significant contribution: an object detection model called DDGNet. DDGNet is a lightweight object detection model faster than the famous YOLOv2 object detector and slightly more accurate. This chapter also introduces our object prediction filtering technique called ThreeWayNMS.
- Chapter 6: Yet Faster, Lighter, and More Accurate YOLO, DenseYOLO, is our second version object detection model repurposed from YOLOv2. DenseYOLO introduces a unique anchor-based object detection network. It is lighter than YOLOv2 and better handles the removal of false-positive object detection. In short, it outperforms the parent YOLOv2 and some state-of-the-art detectors in detection accuracy and speed.
- Chapter 7: Our third significant thesis contribution is the multi-grid redundant bounding box annotation for accurate object detection. This chapter presents the detail of Multi-GridDet, our third model and more robust detector suited for multi-scale object detection. Multi-scale object detection is nowadays becoming the focus of object detection research. It means detecting objects of varying sizes, namely small, medium, and large, using one unified network but multiple output stages dedicated for each scale. The chapter also introduces synthetic data generation and augmentation techniques for supplementing object detection training datasets. Object detection requires a termoundous amount of datasets, and it is usually tiresome and boring, if not impossible, to create millions of datasets for object detection. Instead, some artificial dataset generation is a paramount demand nowadays, and our seemingly smooth and dynamic synthetic data generation is an outstanding contribution in this regard.
- Chapter 8: Resource and Power Efficient High-performance Object Detection Inference Acceleration Using FPGA — is our other significant contribution to this thesis. This chapter introduces hardware acceleration implementation, custom acceleration techniques, design choices, and lastly, implement and test our acceleration design using YOLOv2, DDGNet, and DenseYOLO.
- Chapter 9: Conclusion and Future Works — is the closing chapter of the dissertation. We summarize the major points of each earlier chapter, our contributions, and some anticipated

future works of the thesis. Finally, in the appendixes, we present the main components of our hardware acceleration implementation codes, and details of our synthetic data generation and augmentation are included. Moreover, references used in each chapter are presented in the biography section, which wraps up our dissertation.

Part I

Theory Overview

Chapter

2 Overview of Basics of Deep Learning

Chapter content

Overview of Basics of Deep Learning

2.1 Introduction

Out of the five sense organs in our body, vision is the most important yet complex one. Most of our intelligence depends on our ability to see and our brain's ability to process visual inputs almost effortlessly. The sole goal of computer vision is to mimic this ability of our brain in processing visual inputs. Computer vision evolved over many stages of milestones, starting from the early years of the 1950s and 60s till the present date, though we are still very far from fully mimicking the human vision system, if possible at all. Early computer vision approaches, also referred to by the umbrella term traditional computer vision methods, depended on human experts' handcrafted features and shallow classifier networks or descriptor algorithms such as SIFT[7], HOG[8], and SVM[9]. Though still widely used and sometimes best fitting for certain types of applications, the challenge with these traditional approaches was the lack of end-to-end trainability and the need for manual feature crafting by an expert for all object categories.

Nowadays, deep artificial neural networks, particularly convolutional neural networks, dominate computer vision applications. Before we delve into object detection and tracking, it is customary to briefly introduce artificial neural networks and convolutional neural networks since they are responsible for computer vision in general and object detection, in particular, be at its pinnacle of modern-day success.

2.2 Definitions

Artificial intelligence, Machine Learning, and Deep Learning

Artificial intelligence is a growingly thriving hot area of study with application in almost every science. This popularity is prone to introduce confusing terms and blurred boundaries on defi-

nitions and scopes of some core concepts in AI. Here we briefly introduce and try to clear those confusions. We begin by defining artificial intelligence, machine learning, and deep learning.

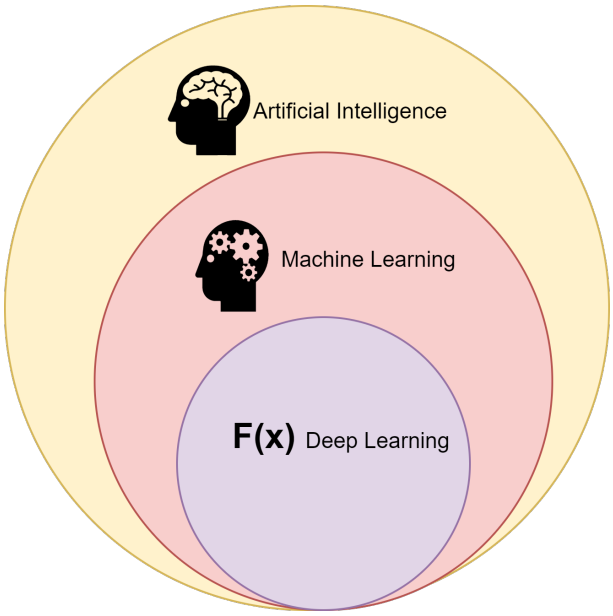


Figure 2.1: *AI vs. ML vs. DL: A Venn Diagram depicting Deep Learning as Part of Machine Learning which is, in turn, one of the ways of mimicking human intelligence artificially to machines*

Since the inception of machines that can solve complex mathematical or statistical problems, humans have been in constant quest of making these machines have the ability to think like humans or even better. This process of enabling machines to mimic human intelligence is called artificial intelligence or AI [10]. Programmable machines or computers easily outperform humans in speed and accuracy in problems with formal rules and procedures such as mathematical or logical operations, thanks to the high computing capacity of the underlying hardware and sophisticated application softwares. However, computers are way behind us,humans, on those problems we humans effortlessly solve, like naming and locating objects on an image or reading someone else's handwriting. Even though we do these tasks seemingly effortlessly, enabling computers to do the same is not a trivial task since we do not have formal rules or algorithms to solve such tasks. Instead, we expect machines to learn from representative data and generalize to other similar problems or tasks. This mimicry of human intelligence through learning from representative data is called machine learning. In general, machine learning is a subset of AI that uses statistical learning algorithms that builds a system that can predict an output from experience or intelligence obtained from prior exposure to similar, not necessarily the same, training representative data.

Based on the data and the problems the machine learning systems target, we can classify machine learning techniques into three categories: supervised, unsupervised, and reinforcement learning. Table 2.1 summarizes the distinction between the three different types of machine learnings.

Some of the most successful machine learning algorithms include artificial neural networks (ANN), support vector machines (SVM), decision trees, and Bayesian models. These algorithms are further classified into shallow learning and deep learning. Even though there is no standard as to when to call a particular algorithm shallow or deep, generally, deep learning algorithms have more connections and more than one hidden layer. Nowadays, the success of artificial neural networks, particularly the one based on convolution, namely referred to as convolutional neural network or CNN, overshadows the other traditional shallow learning networks. CNN's inception is to artificially mimic the human brain and usually has thousands or millions of connections.

In summary, deep learning is a subset of machine learning, specifically deep artificial neural network-based machine learning, where the interconnection of more than one hidden layer is used to increase a model's accuracy. The model's target could be the natural language processing, image recognition, detection, segmentation, object tracking, etc. Furthermore, machine learning, in turn, is a subset of artificial intelligence, meaning, not all artificial intelligence is machine learning intelligence. For example, a computer calculator application is an artificial intelligence but not a machine learning intelligence since the computer can solve calculation problems due to stored programs rather than the machine's learning from data. Figure 2.1 shows this relationship. By this, we claim to have cleared the ambiguity between artificial intelligence, machine learning, and deep learning and move on to further elaborate phrases directly related to our thesis work. These are image classification, object detection, segmentation, and tracking.

Supervised Learning	Supervised learning is a machine learning technique in which the input dataset has labeled, named, target output pair. For example, in a handwritten digit recognition task, all handwritten input digits have a corresponding label of 0 – 9.
Unsupervised Learning	Unsupervised learning is a machine learning technique in which the system under training is expected to distinguish patterns among the input data without being pre-fed with the expected output pair. A typical example is clustering problems, such as K-means clustering.
Reinforcement Learning	Reinforcement learning or semi-supervised learning is a class of machine learning in which learning happens by interacting with the environment, not necessarily the input and output pairs only. Here we describe the current state of the system, specify a goal, provide a list of allowable actions and their environmental constraints for their outcomes, and let the machine learning model experience the process of achieving the goal by itself using the principle of trial and error to maximize some form of cumulative reward. An example is the Markov decision process (MDP).

Table 2.1: *The three types of machine learning techniques*

Image classification, object detection, segmentation, and tracking

Computer vision is one of the most widely researched application areas of artificial intelligence where a computer system tries to mimic a human vision system from digital images or videos. Common computer vision application includes image classification, object detection or recognition, object segmentation, and object tracking.

Image classification:- is the task of labeling an image regardless of the object's position on an image. Inputs to classification networks usually has a single object, and the classifier outputs the label or name of this object or at least the dominantly visible object's name on the image. See Figure 2.2 fist image.

Object detection:- is a computer vision challenge where a model predicts the location and class of all known objects on an image and draws a tightly fit rectangular bounding box on the areas occupied by the individual objects. Simply put, it is a classification with localization, but often the image contains more than one object. Figure 2.2, second image, shows an example object detector's output.

Segmentation:- is in short, segmentation is pixel-level image classification, except instead of

labeling the image as an image classifier does, a segmentation system assigns a similar color value to each object of the same class or instance, creating a color blob resembling the objects. If a segmentation system assigns different colors for each object instance of a similar class as in Figure 2.2 (fourth image), we call it instance segmentation. In contrast, semantic segmentation assigns similar color for pixels of objects of the same class regardless of the pixels belonging to different objects even though the same class. See Figure 2.2, (the third image) for semantic segmentation.

Object Tracking:- is a computer vision application where an object tracking system, could be a single object or multi-object simultaneous tracking system, assigns a unique ID for each detected object in a video frame and follows the object's (s') progression within the video. Often a tight bounding box with a unique identifying ID is assigned to each object, and(or) a path is drawn in the video frames showing the object's trajectory. Object tracking has several applications, including surveillance, activity recognition, uncrewed vehicles (areal and land), and various military purposes. Image classification, object detection, or segmentation are typically time-insensitive, whereas object tracking has an added complication due to its need to be real-time or instantaneous. Since this thesis focuses on real-time object detection and tracking, we will discuss these two concepts in detail in the following chapters, including our proposed methods and contributions.

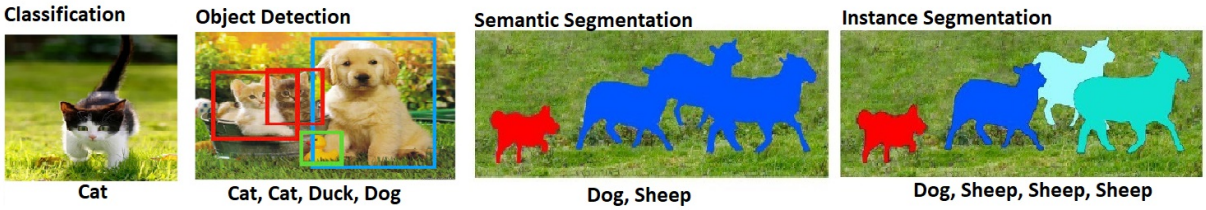


Figure 2.2: *Illustration of image classification, object detection, semantic, and instance segmentation*

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are computing systems inspired by the human brain's biological neural networks or circuits. The brain has over 10 billion interconnected neurons. Figure 2.3 (a) shows a single neuron that on its own has over 10 thousand interconnections with other close and distant neurons through the synapses. A signal from one cell to another is transmitted through a complex chemical process at the synapses forcing the receiving cell to react by raising or lowering its electrical potential (Inhibitory vs. Excitatory synapses). If this potential reaches a certain threshold, a pulse or an action potential (of fixed strength and duration) is sent down the axon. This pulse, in turn, branches through the axon to synaptic junctions with other cells. [excerpt from Lecture 8: Artificial neural networks by Natan Intrator]

Similarly, but not precisely, artificial neurons, Figure 2.3 (b), mimic this process of the biological neuron. Input vector from neurons of a preceding layer $x = x_1, x_2, \ldots, x_n$ pass through a dot product and overall summation with a weight matrix $w = w_{11}, w_{21}, \ldots, w_{n1}$ and addition of an optional bias term. Then on the resulting summation, we apply an activation function, linear or nonlinear function. The resemblance between the biological neuron and the artificial neuron is that the summation process mimics the neural summation of the potential of the signals coming from both nearby and distant neurons at the postsynaptic membrane of the cell body. Based on the potential in the cell body, whether the potential is above a certain threshold or not, and the neuron type (inhibitory or excitatory), a neurotransmitter signal is sent to the synapse junction through the axon. The activation function and its output mimic this later stage of the biological neuron.

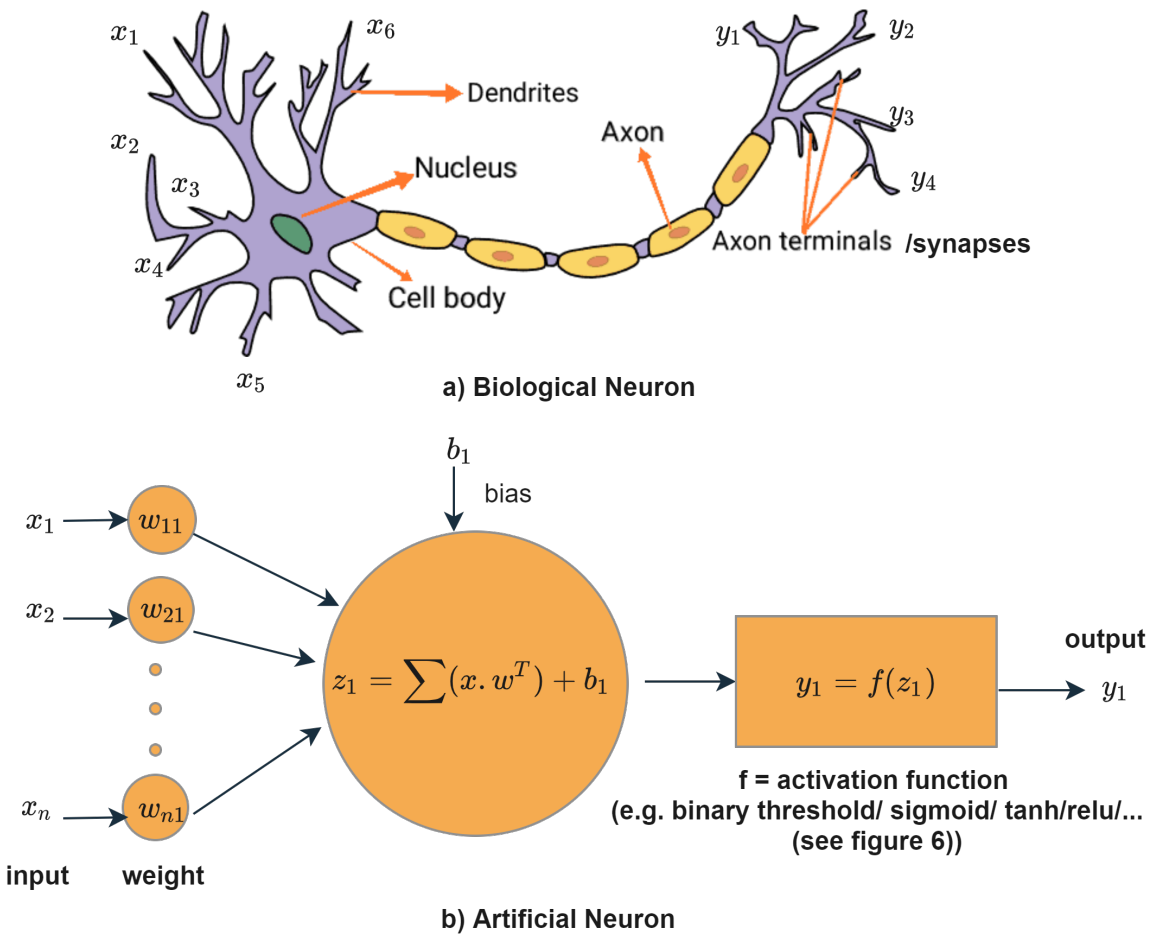


Figure 2.3: *Biological vs. Artificial Neuron*

An artificial neural network, Figure 2.4, is thus a set of connected layered artificial neurons in which each connection has an associated weight and optionally bias. During the learning phase, the neural network's task is to adjust the weights and biases of each layer so that on the output layer, a particular neuron fires more strongly than the rest of the neuron for corresponding input data. The forward per layer calculation of new neural output is called feedforward, whereas the backward process of tuning the weights and biases values is called backpropagation. Furthermore, this feedforward and backpropagation are repeated for several iterations to process all input data and learn common features (weights and biases) that better represent a given input to trigger its corresponding output neuron.

Details of working principles and mathematical equations explaining artificial neural networks and backpropagation can be referred to from Michael Nielsen's book titled "Neural Networks and Deep Learning. [11]" Here we summarize it into three steps: (1)Feedforward, (2) Error backpropagation, and (3) Updating weight and bias. To explain the underlying mathematics, we will use MNIST handwritten digits recognition as an example problem to solve using ANN. Assume the digits are 28x28-pixel images, and when flattened, it will be 784 x 1 vector. Our neural network will have two hidden layers, each with 30 neurons and the last output layer of size 10, one neuron for each digit. Figure 2.4 shows these assumptions and the training parameters of each layer (the weight and bias on each layer). Now we explain the maths of the three steps we mentioned above in an enumerated bullet points:

Step 1. The feedforward:-this is the process of traversing through the network forward while calculating new neurons from preceding layer neurons and weight and bias pairs. As explained earlier, an artificial neuron is a mathematical value calculated, as shown in Figure 2.3

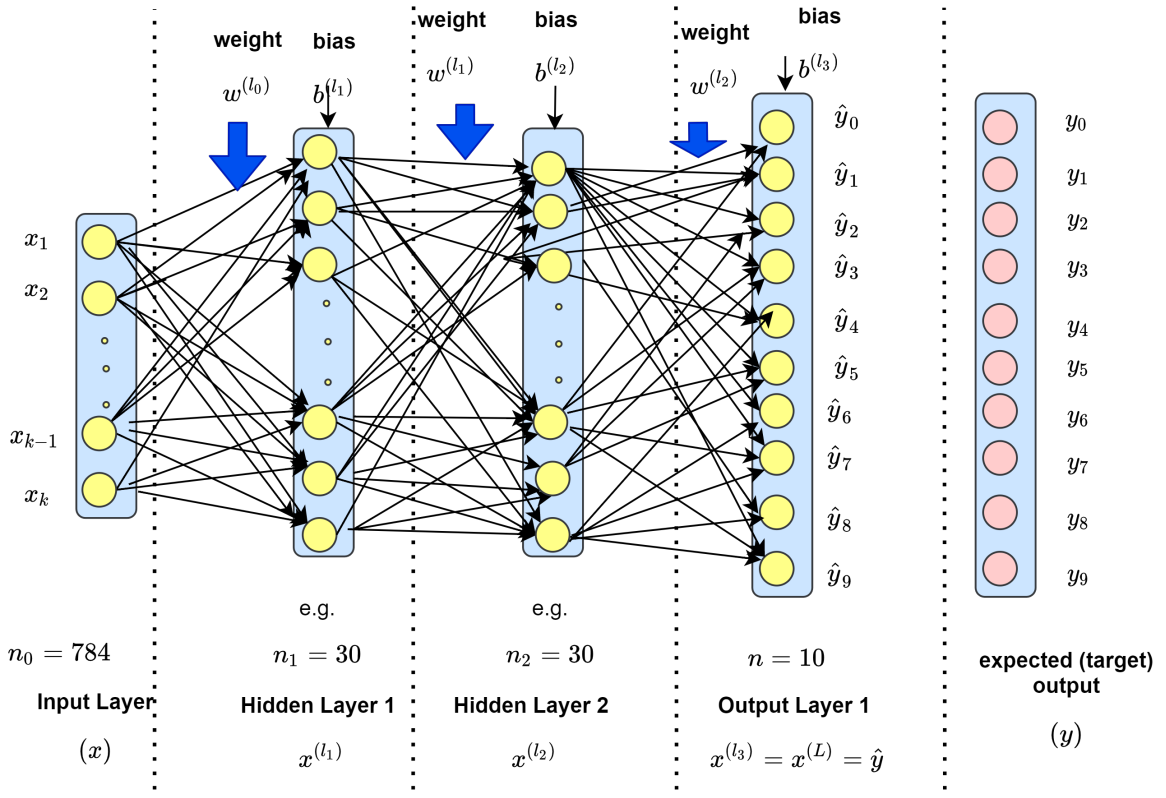


Figure 2.4: *Artificial Neural Network*

(b), and a neural network is the network of such neurons. In the input layer, the flattened vector of the image pixels, in this case, the 28×28 handwritten digit flattened to $784x1$, is our input x. Let us call this input layer with $n_0 = 784$ neurons as layer 0. This layer 0 fully connects with the following hidden layer1 neurons of $n_1 = 30$. Now since layer 0 neurons fully connect with layer 0 neurons, we will have a connection weight of shape $w^{l_0} = 784 \times 30$, because after all, by the connection, we mean the multiplying weight matrix. Initially, these weight values are initialized with small random numbers, and the sole purpose of training is to fix these randomly initialized weights with learned weights. Optionally, we will also initialize a bias parameter of shape $b^{l_1} = 30 \times 1$, one bias value for each neuron of layer 1. Now having sorted out our mathematical notations for the first layer, we explain the steps in feedforward to calculate the new neurons of layer 1 (x^{l_1}), from layer 0 neurons (x^{l_0}), and weights (w^{l_0}) and bias (b^{l_1}) as follow:

1. Calculate ${z_j}^{l_1}$, an intermediate value before calculating the actual (x^{l_1}), using the Equation 2.1 for each neuron of layer 1. In Equation 2.1, $j = 1, 2, \ldots, 30$ stands for each neurons in the first hidden layer and $i = 1, 2, \ldots, 784$ is for each neuron in the input neuron. Note that for the next hidden layer, the current j becomes i, and the neurons of the second hidden layer are j. This way, i and j interchange and Equation 2.1 are repeatedly used for all layers in the forward propagation.

$$ {z_j}^{l_1} = \sum_{i=0}^{n} \left(\left({x_i}^{l_0} . \left({w_{ij}}^{l_0} \right)^T \right) + {b_j}^{l_1} \right) \tag{2.1} $$

2. Chose an activation function f and apply it to ${z_j}^{l_1}$ to calculate the final value of the new neurons in the first hidden layer (x^{l_1}) as shown in Equation 2.2. Some common activation functions and their graph is shown in Figure 2.5.

$$ {x_j}^{l_1} = f\left({z_j}^{l_1} \right) \tag{2.2} $$

3. Repeat 1 and 2 for all layers in the forward direction taking the preceding layer as an input the following layer until we calculate the last layer neurons.

Step 2: Backpropagation:- This is where we calculate the error on the final output layer against the expected target output and propagate the error backward. All neurons contributing to the error will gradually fix their corresponding weights and biases so that the next time the network sees the same input, it will trigger its corresponding output neuron correctly. Backpropagation is the most excellent quality of ANNs against traditional machine learning techniques because it enables end-to-end learning without a need for human intervention or the introduction of other feature descriptors.

Following the same approach as in step 1, we will explain the steps in backpropagation. The final goal in backpropagation is to determine the contribution of each layer's parameters (weights and biases) to the final-layer error or loss. If a function has two or more variables, we can use a partial derivative of the function against each variable to calculate the effect (contribution) of the each variable on the function. Backpropagation depends on this mathematical principle. Now to the steps (procedures) in backpropagation:

1. Chose a cost function or loss functions. There are several cost functions for different applications and targets. Among them, mean-square-error (MSE) is the one we will use here to explain backpropagation due to its simplicity and the understandable intuition behind it. Let the final layer (L) output be $x^L = f\left(z^L\right) = \hat{y}$. The expected target vector is y. The cost function C is thus given by Equation 2.3:

$$C = \frac{1}{2n}\sum_{j=0}^{n}(y_j \; - \; \hat{y}_j)^2 \tag{2.3}$$

Since we have ten digits targets, y is a vector with a size ten, and thus the last layer has $n = 10$. We can rewrite Equation 2.3 by replacing the term $\hat{}y_j$ by Equation 2.2.

$$C = \frac{1}{2n}\sum_{j=0}^{n}\left(y_i - \; f\left({z_j}^L\right)\right)^2 \tag{2.4}$$

$$C = \frac{1}{2n}\sum_{j=0}^{n}\left(y_j \; - f\left(\sum_{i=0}^{n}\left(\left({x_i}^{L-1}.\left({w_{ij}}^{L-1}\right)^T\right) + {b_j}^L\right)\right)\right)^2 \tag{2.5}$$

As seen from Equation 2.5, the cost function depends on the weight and bias parameters considering the ${x_i}^{L-1}$ part as a constant from the previous layer.

2. Calculate the error: once we formulate the error function, we perform partial differentiation to determine the contribution of the two variable parameters (weight and bias) to the error. Similarly, we continue backward for each layer. Equation and 2.7 shows the error induced by weights from layer $L-1$ and bias on the L layer. The equation for all layers is present in Figure 2.6.

$$\frac{\partial C}{\partial w_{ij}^{(L-1)}} = \frac{\partial C}{\partial Z_j^{(L)}} \cdot \frac{\partial Z_j^{(L)}}{\partial w_{ij}^{(L-1)}} \tag{2.6}$$

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{\partial C}{\partial Z_j^{(L)}} \cdot \frac{\partial Z_j^{(L)}}{\partial b_j^{(L)}} \tag{2.7}$$

Step 3: Update the weights and biases: This step is called gradient descent since we are computing the gradients (slight changes or nudges to the weight and bias parameters) so that the overall cost descends to a minimum point zero or very close to zero. Sometimes this minimum point happens to be a local minimum rather than a global minimum forcing the system or the training stack at an undesirable stage. There are several proposed solutions to this kind of problem, such as tweaking a learning rate parameter (the rate at which the weight and bias change), restarting the training with better initial weight and bias, increasing or decreasing neurons in the hidden layer, etc. The weight and bias are updated using Equations 2.8 and 2.9, respectively. Equations 2.8 and 2.9 are for the last layer; the remaining layer is presented in Figure 2.6. Note that these formulas are for a single image as an input. However, usually, we train on batches of images at once so that 1) the training will be faster, 2) the network learns more common features from batch images and learns to focus on distinguishing features of each object (image).

$$w_{ij}^{(L-1)} = w_{ij}^{(L-1)} - \eta \frac{\partial C}{\partial w_{ij}^{(L-1)}} \tag{2.8}$$

$$b_{j}^{(L)} = b_{j}^{(L)} - \eta \frac{\partial C}{\partial b_{j}^{(L)}} \tag{2.9}$$

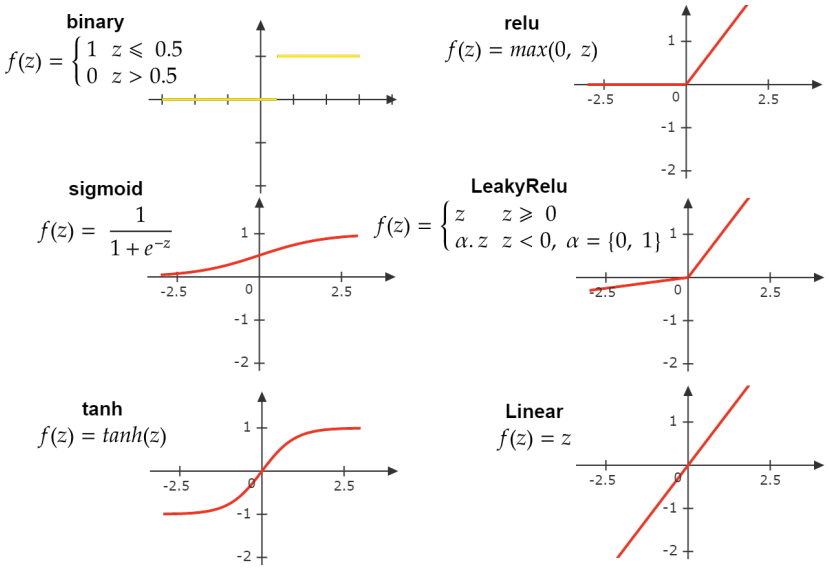


Figure 2.5: *Common Activation functions*

2.4 Convolutional Neural Networks

ANNs have been working well in classifying images, and deeper ANNs work even more but since the networks are fully connected, increasing neurons per the hidden layer means quickly escalating weight parameters. Apart from the drastic increase of weight parameters with increasing neurons, ANNs have one critical pitfall: they do not consider the spatial structure of the image while learning to classify it. Instead of learning directly from the natural arrangement of an image's pixels, ANNs have to flatten the input, ignoring the pixels' spatial distribution on an image. The problem is not only the loss of spatial information, but the fact that ANNs are fully connected (meaning all neurons of the previous layer connects to all neurons of the next layer), it is almost impossible to use bigger images as that would mean an exploding size of weight parameter. As a result, a new learning paradigm that considers the spatial location of pixels as well as suited for higher resolution images was necessary, hence a convolution-based neural network.

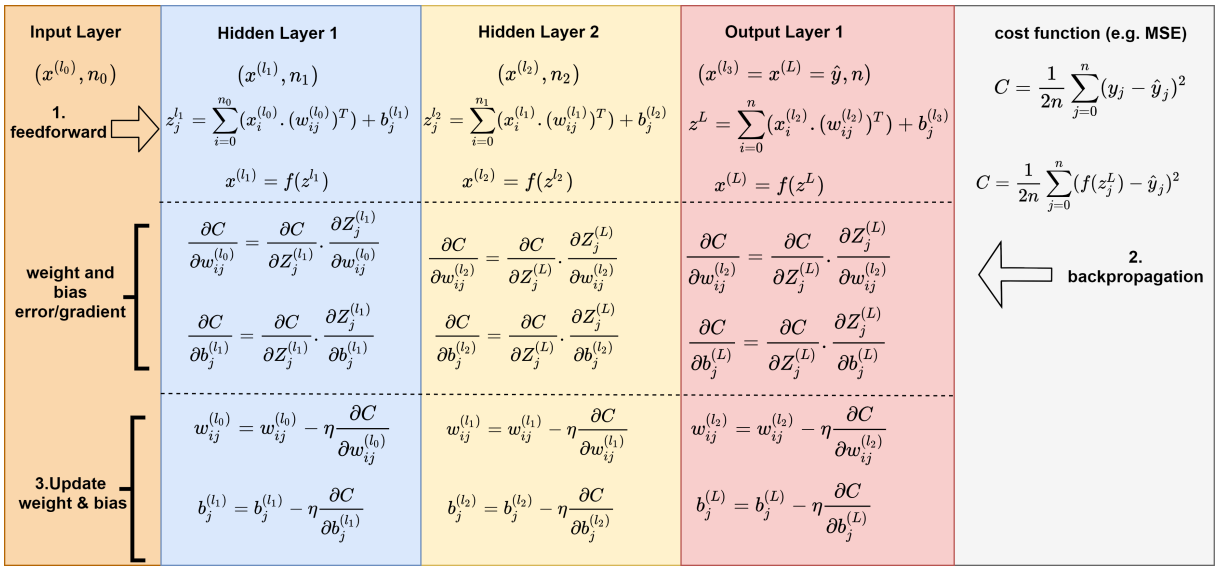


Figure 2.6: *Training ANNs: 1) Feedforward to calculate new activations (neurons), 2) backpropagate the error, 3) update weight and bias. Repeat these three steps until the cost is close or equal to zero or the stops lowering significantly*

Convolutional Neural Networks (CNNs) operate more or less like ANNs except that the layers in CNN are two or more dimensional, called feature maps, instead of the one-dimensional fully-connected layers of ANNs (see Figure 2.4). CNN is a specialized form of neural network in which at least in one of its layers, a kernel or filter window is convolved over the preceding layer feature map (input image) to produce new feature maps. CNN operates on three key features: local receptive fields, shared weights and bias, and pooling. We will explain CNN with an example, but first, let us briefly explain these critical features of CNN.

Local receptive fields:-In ANN a neuron of a layer connects with all neurons of the previous layer with a matrix of a multiplier called weights. However, a neuron in CNN is connected with a few neurons of the preceding layer restricted to a small area called the receptive field. Similar to ANN, when we say connection, we mean a multiplying weight matrix. The kernel is usually square-shaped with odd-numbered sides commonly in the range of 1 to 11.

Shared weights and biases:- During convolution, we use the same kernel or filter, hence shared weight and bias, by sliding and convolving it on the local receptive of the preceding layer. We can use more than one kernel on a given convolutional layer to produce more planes of feature maps on the following layer.

Pooling:- Pooling is a form of down-sampling the size of the feature map outputs of the convolutional layer. It reduces the memory footprint of CNN while preserving the prevalent features and disregarding the less influential features. This down-sampling enables CNNs to have more depth, which is why CNN-based networks dominate modern deep learnings. There are different types of pooling layers. The most common are max-pooling, average pooling, and l_2 pooling.

Next, we explain the mathematical aspect of the convolutional neural network. Assume an input image of size $h \times w \times c$ where h and w are the height and width of the image, respectively, and c is the image channel or depth. $c = 1$ for grayscale image whereas $c = 3$ for an RGB image. And again, let us assume that the convolution has a kernel (filter) of size $f_c \times f_c \times n_c$ and a stride of $s_c \times s_c$, the c subscript denotes that the kernel $f \times f$ or the stride $s \times s$ is associated with the convolution layer. For further clarity, see Figure 2.7. We can perform the convolution on the input image in either of two formats, valid convolution or same convolution. If we want to get an output feature map of equal height and width with the input feature map (image), we need to pad the input with zero. Otherwise, the convolved output feature map will have a

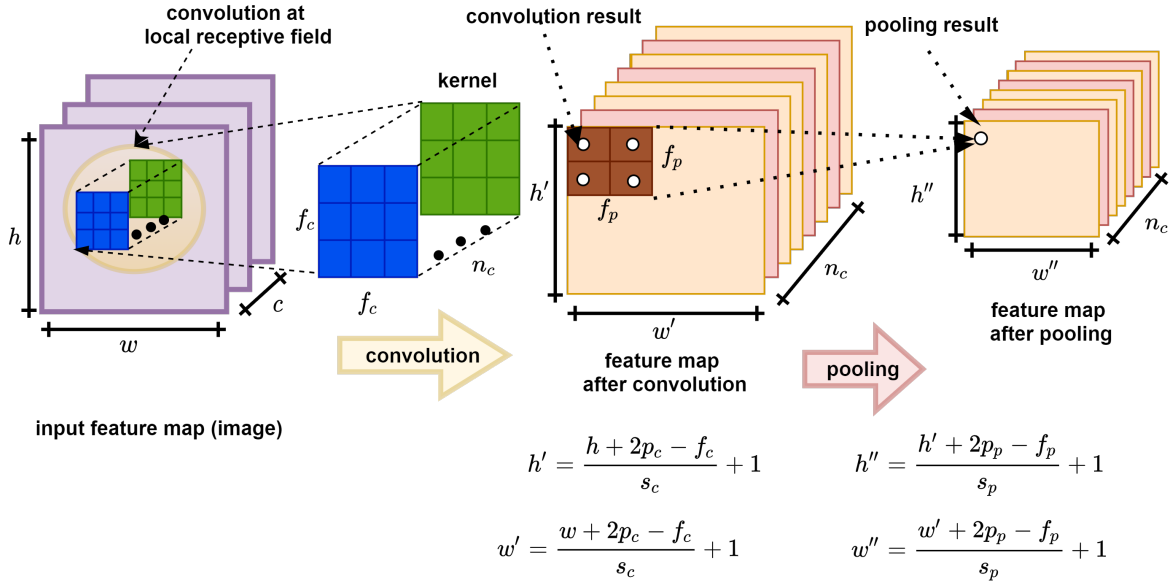


Figure 2.7: *Convolution and Pooling layer of CNN*

relatively smaller size than the input feature map. The output feature map width $w^{'}$ and height $h^{'}$ are given by Equations 2.10 and 2.11:

$$h^{'} = \frac{h + 2p_c - f_c}{s_c} + 1 \tag{2.10}$$

$$w^{'} = \frac{w + 2p_c - f_c}{s_c} + 1 \tag{2.11}$$

Similarly, for the pooling layer, we consider a pooling kernel of $f_p \times f_p$ and a pooling stride of $s_p \times s_p$ and the resulting feature map after the pooling will have a height of $h^{''}$ and width of $w^{''}$ calculated by Equations 2.12 and 2.13:

$$h^{''} = \frac{h^{'} + 2p_p - f_p}{s_p} + 1 \tag{2.12}$$

$$w^{''} = \frac{w^{'} + 2p_p - f_c}{s_p} + 1 \tag{2.13}$$

Figure 2.7 shows only the convolution and pooling layer of a CNN network. However, a typical deep CNN will have many more convolutional and other layers such as batch normalization and activation layers though the convolutional layer is its underpinning. Below we will explain some of these layers one by one.

Batch Normalization Layer

Training deep networks, either ANN or CNN, is challenging because each input batches of an image are different and have a different distribution. Moreover, since the initial state of the network is random due to the random weight initialization, each layer of the network treats the input image differently. Especially if the input image varies considerably, the network might not converge sooner, and training might take very long if it converges at all due to the need to use a lower learning rate to minimize fluctuation. This phenomenon is called internal covariate shift. The typical approach to solving such problems is to use some sort of normalization on the output of one layer before inputting it to the next layer (the next convolutional or fully connected layer).

In general, we have two standard normalization practices. (1) normalizing the data to a value between 0 and 1, using Equation 2.14 or (2) normalizing it to have a mean of zero and standard deviation of one so that the data would be a normal distribution. We use Equation 2.15 for the second type of normalization. In the equations, x is the input data, and m and s are the mean and standard deviation of the data, respectively.

$$x_{normalized} \leftarrow \frac{x - \mu}{x_{max} - x_{min}} \tag{2.14}$$

$$x_{normalized} \leftarrow \frac{x - \mu}{\sigma} \tag{2.15}$$

Moreover, we can have two distinct normalization processes based on where or when we apply the normalization in the network. First is normalization during pre-processing, and second is normalization between the layers of the network. The second type of normalization makes the normalization task part of the network like the trainable convolutional or activation layers. Here want to discuss the latter type of normalization, normalization between neural networks, and hence we will call it the normalization layer since it is part of a network, CNN or ANN, as a layer.

There are different normalization techniques proposed and tried over the years. One of the well-known, such normalization is called Batch Normalization [12] or BN for short. It is a normalization technique done between the layers of a Neural Network instead of in the raw data, and it is done along mini-batches instead of the complete data set. Its purpose is to speed up training and use higher learning rates, making learning easier. It is a slight modification of Equation 2.15 to include two learnable parameters, γ scaling factor and β a shift factor.

Let us formulate the equation of BN; Assume that x is an input data, to be exact, it is an output of a batch of a neural network layer of mini-bach size m either from a previous fully connected or convolutional layer, and it is an input to the next layer. And let $y_i = BN_{\gamma,\beta}(x)$ be the batch normalization of x. Then $BN_{\gamma,\beta}(x)$ is calculated using the following four steps (Equation 2.16 – 2.19). Equation 2.16 calculates the mean of the mini-batch, whereas equation 2.17 calculates the standard deviation of the mini-batch output of the network. Furthermore, equation 2.18 normalizes the mini-batch data x. Finally, equation 2.19 is the BN of the normalized x, including the network learnable γ and β.

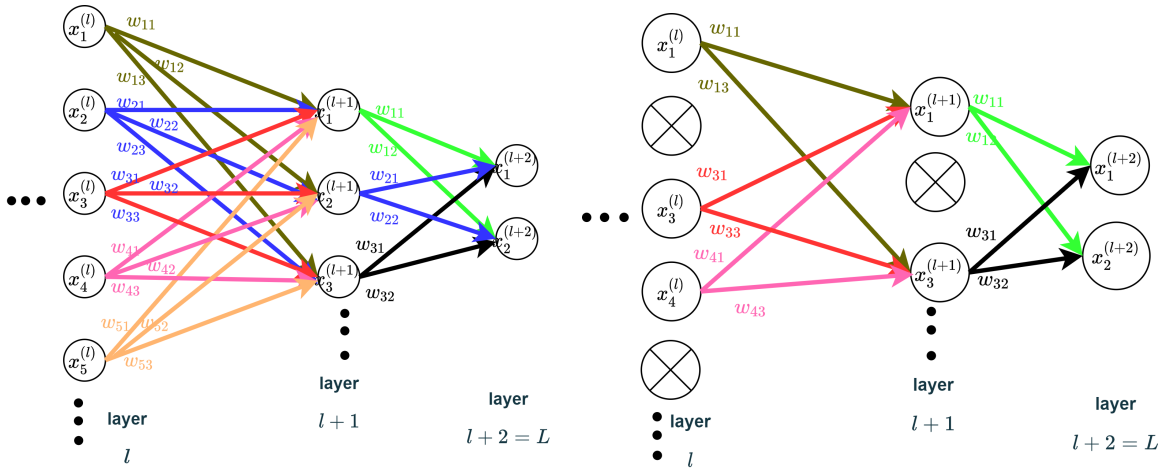
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.16}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.17}$$

$$\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.18}$$

$$y_i \leftarrow \gamma\widehat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{2.19}$$

Batch Normalization has another beneficial side-effect which is regularizing a network. Though it is not a very good regularizer, it has a regularization effect since it keeps network outputs from exploding and allows a higher learning rate. Other very recent network normalization techniques can sometimes offer a better result than batch normalization. These are weight normalization, layer normalization, instance normalization, batch-instance normalization, group normalization, and switchable normalization. All have their own merits and demerits that need to be investigated individually while using one to include in one's custom network.



(a) *Fully Connected Layer Without DropOut* **(b)** *Fully Connected Layer With DropOut*

Figure 2.8: *Fully Connected Layers without Dropout (a) and with Dropout (b) layers*

Activation Layer

The activation layer, also called the non-linearity layer, takes the convolutional layer output and produces an activation map. It is an elementwise operation over an input tensor or matrix, and hence the resulting tensor or matrix will also have the same shape and size as the input. We have already discussed some of the widely used activation layer functions in earlier section,see Figure 2.5. While using these functions in CNN, we only have to remember that the operations are on higher-order matrix or tensors such as 3D or more.

Fully Connected Layer

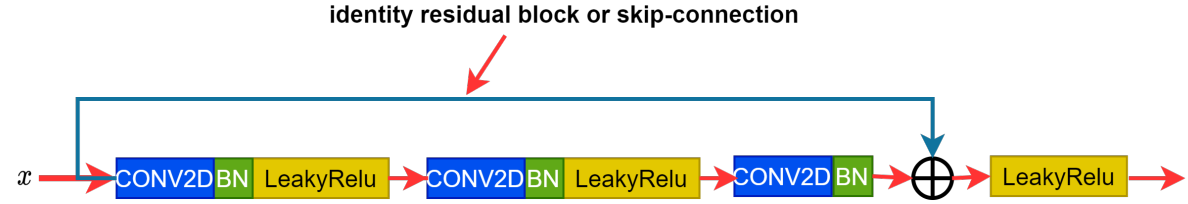
As the name suggests, Fully connected layers are a complete one-to-one connection of all previous neurons to all subsequent layer activation neurons, as shown in Figure 2.4. In CNN, fully connected layers are usually used on the last few layers and the output layer to classify the output of the CNN network into the classes of objects or images. Due to the full connection, they are the second most resource-consuming processes and lead to heavy networks.

Dropout Layer

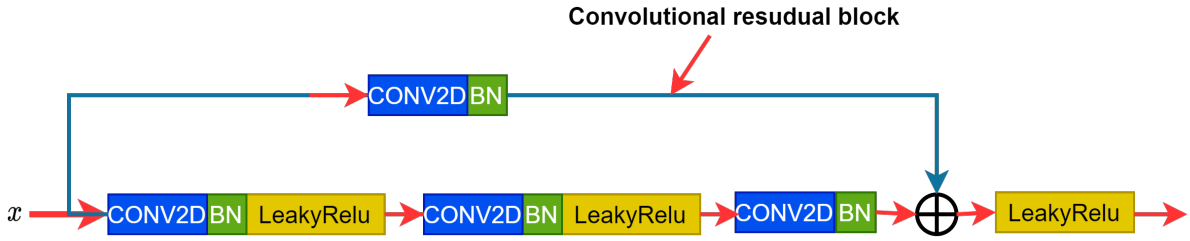
Overfitting and underfitting are two of the most common deep learning challenges. Overfitting is when the neural network is too familiarized with the training set but underperforming on the test set. Overfitting usually happens when the network has too many neurons to represent the dataset or a small training dataset. One remedy is to remove some layers or neurons of the hidden layers and retrain or increase the size of the training dataset and train further. The Dropout layer in the neural network performs precisely that. Its purpose is to turn off randomly (that is, set their values to zero) some neurons of a layer it is applied to during training so that the network trains on smaller weight parameters, preventing overfitting. This makes the layer look like another network with different nodes and connectivity to the prior layer. The effect of Dropout is closely similar to how an ensemble network works since every training iteration makes the network with Dropout appear new network. Figure 2.8 shows a simple, Fully connected network with and without Dropout layer.

Residual Blocks

A common technique to increase the performance of deep learning models is to go even deeper and use more training. However, as the depth of a model increases, the training tends to saturate, and training loss stops reducing beyond some value since the last layers tend to capture no useful feature maps from the input image. To solve this problem in deep models, researchers at Microsoft proposed adding a by-pass connection to transfer features from earlier layers to a later layer repeat the stack of such blocks to construct a very deep model. The by-pass connection is commonly called skip-connection or shortcut connection. Since the first paper, researchers have utilized different kinds of skip-connections and residual blocks. Figure 2.9 shows some of these varieties. Note how the skip connection is before the activation function; in this example, LeakyRelu is the activation we utilized since it is the most used non-linearity function. The activation is applied after adding the transferred feature map (residual block) and current layer feature map.



(a) *Identity Residual block*



(b) *Convolutional residual block*

Figure 2.9: *Residual blocks*

2.5 Training Deep Learning Model

Modern machine learning is predominantly dominated by deep learning approaches based on convolutional neural networks, especially for image and video inputs, due to CNN's ability to learn an ample amount of subtle details of its input. More importantly, it is shift and scale-invariant, meaning, once it is trained well, it can easily recognize an object in any scale, position on an image, and rotations. However, training a deep CNN model is not a trivial job since it requires a massive amount of training data, a cluster of high-performing CPUs and GPUs, and a carefully designed model, usually achieved through time-consuming trial and error-based retraining and hyperparameter tuning process. Below, we summarize some of the basic best practices and stages of training deep CNN to achieve state-of-the-art performance from one's model. However, our below discussion assumes that the developer has thoroughly thought over the problem the machine learning (ML) will solve and hence set an objective; a stage formally can be called the problem definition stage, and we will not be discussing this stage as we only focus on the machine learning part.

1. **Prepare Dataset**:- Deep learning is a hugely data-hungry learning framework. Moreover, any machine learning is as good as the quality of data it is trained on. Collecting many

data by itself may not be enough as the quality of the data for the intended purpose needs to be checked. Quality data for DL has to have at least the following features:

- Balanced data:- the training, validation, and testing dataset must not overly favor certain classes, shapes, or textures over others.
- Enough data sample to represent each class and type of data
- Sufficiently representative of the real-world data the model encounters, there should be no surprise data type that the network never saw during training but are common on real-world application of the model.

Once the data is prepared, one should divide the data into training, validation, and test set. The training set is the data the DL uses to learn appropriate weights, biases, and other model hyperparameters to minimize a given cost function. The validation set is used to optimize and control the network's training, and it should never be mixed with the training set since that would lead to overfitting or false performance reporting. However, if the dataset is small, one can use K^{th}-fold cross-validation to train both on training and validation set and report model performance on average of K iterations. However, the test set should entirely be separate and representative of the real-world data the model will encounter. Its purpose is to evaluate the performance of the model. Though it is not standard, common practice is to make 80 % of the data training, 10 % validation, and 10% test set.

The other critical process in dataset preparation is ***dataset annotation***. Dataset annotation is the process of labeling and categorizing all data in the dataset for training, validation, and testing. Data annotation is dependent on the data type (text, audio, image, or video) and the purpose (classification, detection, segmentation, natural language processing, et cetera). It is a laborious, slow, and error-prone process, especially if the purpose is detection or segmentation. The annotated dataset should also be in a simple, compressed, and faster-to-read format so that data reading will not be a bottleneck during training.

2. **Data Augmentation**:- Data augmentation is a technique for increasing the amount of data by slightly tweaking the copy of existing data through geometric transformation, color augmentation, noise addition, synthetic new data generation, or combining one or more of these techniques. It is a critical pre-processing step even when we think we have enough data to train our model. Often CNN is praised as transformation, scale, viewpoint, and (or) illumination invariant. Nevertheless, these features are achieved or boosted by the data augmentation technique we use. Data augmentation artificially exposes the model for all possible types of data the network might encounter in the real-world environment. Data augmentation can be offline, that is, before training, or online process, meaning augmentation during training.

   There are many data augmentation techniques proposed and used over the years, mostly in combination. Some of these include:

   - Geometric transformation: flipping (vertically or horizontally), rotating, zooming, cropping and padding, affine transformation, sheering
   - Color manipulation: adding or decreasing brightness, contrast, hue, and saturation
   - Adding noise, randomly eliminating a portion of data
   - Synthetic data generation: copy-paste data generation, training model to generate data using GAN (generative adversary network)

Even though data augmentation is an essential part of training DL models, over utilizing it is also counterproductive since some augmentation might introduce unrealistic data or be too much and slow the learning of the model. For example, if we do not expect vertically flipped data in the real world, then there is no need to use vertical flipping as an augmentation technique during training stages. Another significant issue is object detection data augmentation; we must not forget to augment the bounding boxes while augmenting the images. Moreover, some augmentation techniques are known to mess up bounding boxes if used as an augmentation technique for object detection, and thus proper procedure to fix the mess created by these augmenters must be taken into consideration. For example, rotation augmentations might make the bounding box a) to get out of the image boundary or b) tightly unfit the object. In general, though data augmentation is the pinnacle of the success of training deep CNN model, it should be handled with great care and proper vetting of techniques to be used.

3. **Create the Model**:- To create a deep learning model, we need to identify the objective of the model. We should ask whether we need classification, detection, segmentation, tracking, language processing, and the like. Because, based on the objective, we determine our model's objective function, also called loss, cost, or error function. We have many well-known objective functions at our disposal that we can choose based on the dataset and the task at hand. To mention a few: binary-cross entropy, categorical cross-entropy, mean square error, hinge loss, mean absolute error. Usually, in detection problems, one or more of these loss types are used together to target the classification loss and bounding-box regression loss separately.

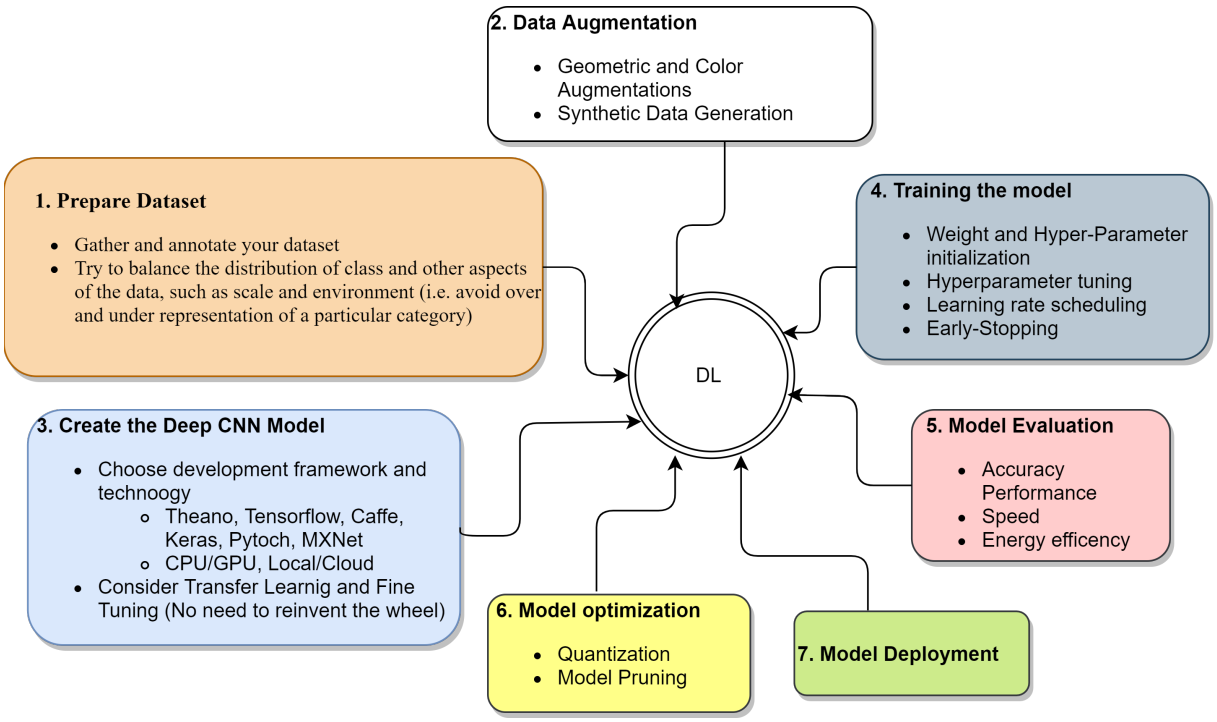


Figure 2.10: *Stages of training and deploying Deep CNN based networks*

Nowadays, there are several software frameworks and programming language choices to create a DL model. Each of them has its advantages and shortcomings that need to be analyzed individually by the developer. In general, one can consider criterions such as abstractions the frameworks offer to hide the details of creating, training, and deploying a model as a selection criterion. Alternatively, the framework's flexibility to include one's custom methods or other changes, support community it has, ease of programming, and

the like can be a criterion to select a framework. Well-known deep learning frameworks include Tensorflow, Keras, PyTorch, Theano, Caffe, OpenCV, and Matlab Deep Learning ToolBox.

Last but not least, in competitions and challenges performance of a network might be enough to win the challenge. However, in real-world computer vision problems, some other factors are as equally critical as performance. These factors include running speed, resources such as memory and processors required to train and deploy the network, power consumption, data size required to train the network, the input image resolution, or video frame rate. The underlying fact is that there is no standard way to determine or calculate the depth of layers, the number of neurons, or feature maps per layer, or filter size to use per convolutional layer. Hence, one must compromise on the acceptable trade-off between speed, performance, and resources required to design and train a model.

4. **Training the Model**:- Training a deep learning model or machine learning, in general, is a process of iteratively modifying a layered weight and bias parameters of a network so that the network can map an input to its output encoding by gradually learning to minimize a given cost function. During training, our decisions on the following core concepts matter the most on the performance of our model:

   - Weight initialization technique
   - Transfer Learning and Fine Tuning
   - Loss function
   - Optimization algorithm
   - Learning rate scheduling
   - Epoch, Batch Size, Early-Stopping

   Considering the broadness of this topic, we recommend the reader to reference books such as [Pattern recognition by Bishop [10], Deep Learning Book by Ian Goodfella [13], Neural Network and Deep Learning by Micheal Nelson [11]].

5. **Model Evaluation**:- After training or training a model, evaluating the model's performance is an unarguably critical part of model development. Model evaluation is different from model loss calculation. Loss guides the network toward maximum performance by minimizing input to output mapping error through optimization functions such as stochastic gradient descent. The loss function needs to be differentiable. However, performance measure metrics do not. The loss function can sometimes be used as performance evaluation metrics in regression-type problems, such as cosine distance, euclidian distance, or mean square error.

   Model performance evaluation metrics depend on the type of task the deep learning model tries to solve, classification, regression, or both as in detection problems. In classification challenges, we have metrics such as accuracy, recall, precision, and F1-score. In regression challenges, we have distance measurement, mean square error, or mean absolute error. We have metrics that combine classification and regression for detection challenges, such as MAP based on average precision for classification accuracy and IoU for measuring overlap between the ground-truth and the predicted bounding boxes. Since the classification and detection metrics are of critical concern in this thesis, we will explain them in more detail as follow one by one.

   Before explaining each metric, we would like to introduce an essential term to all classification metrics: ***confusion matrix***. Confusion Matrix is a tabular visualization of the ground-truth labels versus model predictions. Each row of the confusion matrix represents

the instances in a predicted class, and each column represents the instances in an actual class. It is not a performance metric on its own but sort of a basis on which other metrics evaluate the results. In order to create a confusion matrix of a classification problem, one must set the ***null hypothesis***, which is the thing our model is supposed to classify as correct classification. For example, if we want our model to look at an image and classify it as a cat image and non-cat image, the null hypothesis is "this is a cat image." The null hypothesis is also called true positive (TP). Based on this, table 2 shows the confusion matrix table for our cat and non-cat image classifier.

Table 2.2: *Classification Confusion Matrix*

		Ground-truth class	
		Cat	Not Cat
Predicted class	Cat	TP	FP
	Not Cat	FN	TN

True Positive(TP):- positive samples predicted correctly, i.e., an actual cat image predicted as cat image by the model

True Negative(TN):- negative images classified correctly, i.e., an actual not cat image is predicted as a not cat image by the model too

False Positive(FP):- negative sample classified incorrectly, i.e., not cat image predicted as cat image. This error is also called Type-1 error.

False Negative(FN):- positive image classified incorrectly, i.e., a cat image classified as not cat image. This error is also called Type-2 error.

The position of FP and FN in the confusion matrix entirely depends on our null hypothesis.

Now let us explain the classification and detection evaluation metrics:

Accuracy:- is the most basic and intuitive measure, and it is given by the number of all correct predictions divided by all predictions multiplied by 100 to convert it into a percentage.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \times 100\% \tag{2.20}$$

Precision:- we can summarize it as answering the question, "of all the predictions the model predicted as positive, how many are actually positive or correct?". It measures the hit rate of the model.

$$Precision = \frac{TP}{TP + FP} \times 100\% \tag{2.21}$$

Recall:- is another very informative metric, and it answers the question "how many of the positive samples was the model able to classify correctly?". It measures the sensitivity of the model.

$$Recall = \frac{TP}{TP + FN} \times 100\% \tag{2.22}$$

F1-Score:- is the weighted average of precision and recall. Precision considers TP and FP, whereas the recall uses only the TP and FN and F1-score instead takes into account both FP and FN in addition to the TP. This makes F1-score more interesting metrics and general but not as intuitive as accuracy.

$$F1_{score} = 2 \times \frac{Recall \times Precision}{Recall + Precision} \times 100\% \tag{2.23}$$

All the above metrics are explained from a classification point of view since we only considered the ground truth and prediction class. However, in detection challenges, the class of the object and the bounding box of the predicted object must significantly overlap with the ground truth class and bounding box. To say an object is detected, that is, a true positive (TP) detection, the class of the predicted object must match the class of the ground truth, and the bounding box of the predicted object must overlap above a certain threshold against the ground truth bounding box. The metric we commonly use to check whether a predicted bounding box overlaps against the ground truth is IoU.

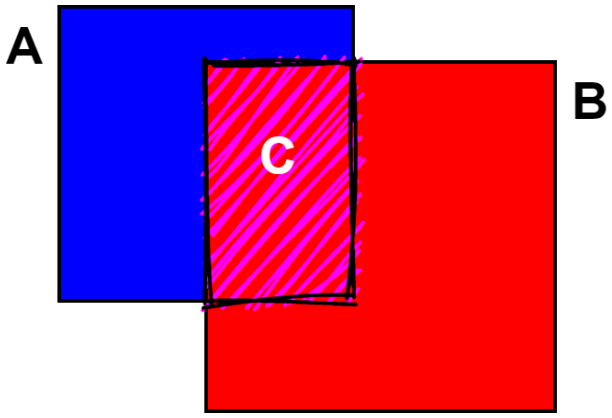


Figure 2.11: *Intersection over Union*

Assume the blue box labeled A, in figure 2.11, is the ground truth, and the red box labeled B is the predicted bounding box of object A. Both boxes A and B are the same class; for example, say both are "cat," hence the classification is correct. But for us to call cat is detected, the IoU needs to be above some preset IoU threshold, IoU_{thresh}. The shaded region labeled C is the intersection box. If A and B do not overlap, then the area of box C is zero indicating that the IoU is also zero. We calculate the IoU using Equation 2.24.

$$IoU = \frac{Area\ of\ C}{Area\ of\ A\ +\ Area\ of\ B\ -\ Area\ of\ C} \tag{2.24}$$

Image classification or object detection models are usually trained to classify or detect more than one class of objects or images, respectively. Accordingly, the standard metric in such multi-class image classification or detection metrics is an average of all classes in the dataset. One widely used metric in many classification and detection challenges is called ***mean average precision(mAP)***. Public challenges such as the Imagenet classification challenge and Pascal VOC and COCO dataset object detection challenges use **mAP** to measure performances and reward winning models.

6. **Model Optimization**:-

Model optimization in deep learning can be implemented in the overall deep learning model creation training and deployment stage. Optimization algorithms or methods to train a deep learning model might work on optimizing the loss function, error backpropagation optimizer (e.g., stochastic gradient descent(SDG), Adam, RMSprop optimizer), data representation (encoding) optimization, data feeding, and fine-tuning. Other optimization focuses on minimizing the model's size by pruning the redundant or irrelevant neurons or layers of the network or compressing model representation through quantization to transform the model from floating-point format to quantized integer format. Retraining a pruned or quantized (or both pruned and quantized) model might even solve the local minima problem of the earlier unquantized and unpruned version of the model and improve the performance of the model.

7. **Model Deployment**:-

   Developing and training a model that works with some measurable and commendable performance is the first step of the entire endeavor of creating the deep learning model, though of paramount significance. The end goal is deploying and serving the model for end customer users. Some of the major tasks during deployment involves polishing the model codes from error and for readability and future maintenance, developing a platform-independent container such as docker for the model, a client-side interface application like web-application, mobile or desktop application development, and finally deploying on the appropriate environment which can be either edge device or cloud system. Model deployment is a broad topic and dependant on the purpose of the application, and thus we leave it at this.

In general, in this section, we have tried to divide the overall stages or processes of creating and deploying a deep learning training model into seven steps, and we briefly tried to discuss the details under each section. We are not claiming that our discussions are complete and that the seven stages we mentioned are exhaustive, but we claim that the most critical topics are at least mentioned as a starting point for someone as a reference. Figure 2.10 shows these stages as equally essential stages of deep learning-based model development stages.

2.6 Conclusion

In conclusion, this chapter reviewed the basics of artificial neural networks, convolutional neural networks, and the overall process of creating and training a deep learning method to solve an artificial intelligence problem. We begin the chapter by defining and demarcating the concepts and scopes of basic terms in computer vision challenge and introducing the analogy of the relationship between biological neural networks and artificial neural networks. The arithmetics of an artificial neural network, whether based on convolutional or fully connected or mixed layer types, pass through an iterative two-stage process. These are (1) feedforward: a process where new model parameters such as weights and biases are learned layer by layer, and (2) backpropagation: a process where errors on the last layer are propagated backward, tweaking the learned parameters so that the loss on the last layer becomes as minimum as possible. A neural network can be shallow or deep based on the number of intermediate layers. Multiple research and current trends have shown that the more profound the networks are, the more accurate they are. In the next chapter, we will discuss some well-known deep convolutional neural networks for object detection and briefly introduce object tracking.

Chapter

3 Object Detection and Tracking: Study of Current Trends and State-of-the-art approaches

Chapter content

Object Detection and Tracking: Study of Current Trends and State-of-the-art approaches

3.1 Introduction

Object detection is one of the most challenging and fundamental branches of computer vision. Unlike image classifiers that only seek to label an image in its entirety into one of the predefined categories, object detectors, however, are required to precisely locate all known objects on an image using a tightly-fit bounding box around each object and label them correctly.

Traditional object detection techniques follow pipelined stages of processes. We can divide these stages into three. The first stage of a traditional object detector is to generate candidate region proposals using either method such as Selective Search [14], Sliding-Window[8], and Edge-Boxes[15]. A second stage will extract features of a fixed-length vector using feature descriptors such as HOG from the generated candidate regions. The third stage will try to classify the extracted features into one of the object classes with some measure of confidence score using methods such as SVM. These methods are not end-to-end trainable, require high expertise in feature extraction, are slow for both training and inference, and are impractical for multi-class generic object detection since more human expertise and time are needed to design features for all object classes. These changed when the neural network, particularly convolutional neural network, based methods beat these old approaches and gave birth to modern object detection

approaches dominated by CNN-based deep learning networks. In this chapter, we will dive deep into a review of these modern object detection techniques followed by object tracking based on object detection, and then finally, we introduce the role of FPGA in object detection and tracking.

3.2 Modern Object Detection Trends

The trend in current state-of-the-art object detection networks is to use a deep convolutional classifier network as a background extractor. As pointed in [16], these trends of using a deeper network are best explained by the growing trend of using families of the sophisticated and cumbersome ResNet-based classifiers in almost all best performing state-of-the-art object detectors at the cost of speed and resource expense.

Two deep CNN-based approaches, shown in figure 3.1 dominate modern era generic object detection researches. These are implementation of object detection as a two-stage or one-stage detector. Two-stage detectors perform object detection in two core steps, that is, first using a serious of CNN propose sparse candidate regions on an image, likely containing an object of interest, and second classify and score each proposed region. One-stage detectors, however, perform both the localization and classification simultaneously in one forward pass. Generally, two-stage networks are known for their high performance in detection accuracy though they are usually very slow and heavy. On the contrary, one-stage detectors are very-fast while relatively are also less accurate.

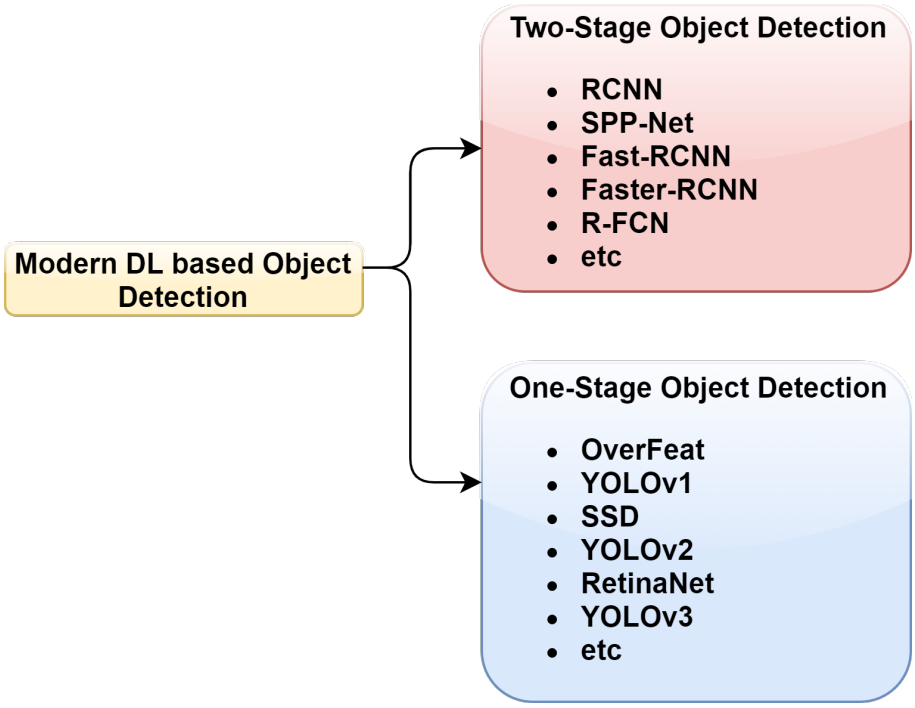


Figure 3.1: *Modern Deep-Learning Based Object Detection Categories*

3.2.1 Two-Stage Object Detection

The pioneer and undoubtedly most representative of the two-stage object detection technique is the famed R-CNN [17](region-based convolutional neural network). Operation-wise, R-CNN is almost the same as the families of the traditional object detectors, except it replaces the feature extraction stage of traditional detectors with CNN-based feature extractors, hence avoiding the

need for a human expert to generate the features for descriptors. As a result, compared to the famous traditional detector called DPM [18](Deformable Parts Model), RCNN improves mean average precision (mAP) by 21% on the PASCAL VOC2010 object detection dataset. This performance gain was the case for attracting much research attention for this line of object detection leading to successive incremental versions based on the first R-CNN.

RCNN

As seen in figure 3.2, RCNN has three main parts(Steps) minus the input image and input pre-processing. (1) Propose category-independent regions of interest by Selective Search (2k candidates per image). Those regions may contain target objects, and they are of different sizes. Other than Selective Search, EdgeBox, and BING[19] can also be also used to generate the region proposals. (2) For each object proposal of arbitrary scale, the image data is then warped into a fixed size (e.g., 227×227) and put into the deep CNN network (e.g., AlexNet) to compute a 4096-dimensional feature vector. (3) Finally, based on the feature vector extracted by the CNN network, the SVM classifiers predict the specific category of each proposal.

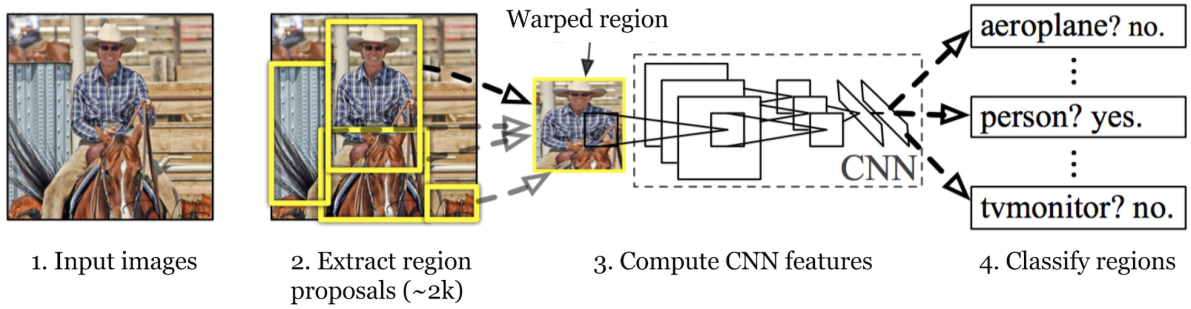


Figure 3.2: *The architecture of R-CNN. (Image source: [17])*

As typical in all modern object detection approaches, a detector is first trained on an image classification dataset such as ImageNet as a classifier and then repurposed into the detector using detection datasets such as Pascal VOC or COCO dataset. Accordingly, in RCNN's second stage, the CNN network (e.g., AlexNet) is firstly pre-trained on ImageNet and then fine-tuned on a specific object detection dataset (e.g., Pascal VOC). Because the number of object categories on ImageNet and Pascal VOC is different, the outputs of the final fully connected layer in the CNN network should be changed from 1000 of ImageNet to 21 (20 Pascal VOC classes plus the background). Based on the CNN features extracted from the trained CNN network, the linear SVM classifiers for different classes are further trained. When training the SVM classifier per class, only the ground-truth bounding box is labeled as positive. Otherwise, the proposal is negative if the IoU overlaps below 0.3 with all the ground-truth bounding boxes. Because the extracted CNN features are too large to load in memory, the bootstrap technique is used to mine the hard negatives in training SVM classifiers.

Though RCNN dramatically improves object detection performance, it has had several shortfalls. To mention the major ones, RCNN:

- detection was very slow (47 seconds per image)
- the training was slow and complex too. Separate network to generate region proposals, wrap the region proposal into fixed image size, pass through CNN for object feature generation, classify the generated features into object class using another separate procedure (SVN)
- not end to end unified training

- memory-wise the generated 2k region proposals per image were massive. An image might contain one or two objects, but the RCNN region proposal must generate about 2k proposals

SPPnet

To remove the fixed-size constraint and accelerate the detection speed of R-CNN, He et al. proposed **SPPnet** [20]. Instead of cropping or warping the image data of all the proposals before computing the CNN features, SPPnet firstly computes all the convolutional features of the whole image and then uses spatial pyramid pooling to extract the fixed-size features of each proposal. Figure 3.3 gives the illustration of the spatial pyramid pooling layer (SPP). Based on the feature maps of the last convolutional layer, SPP splits the feature maps of the proposal into 3×3 spatial bins, 2×2 spatial bins, and 1×1 spatial bins, respectively. In each spatial bin, the feature response value is calculated as the maximum of all the features which belong to the same spatial bin (i.e., max-pooling). After that, the outputs of 3×3 spatial bins, 2×2 spatial bins, and 1×1 spatial bins are concatenated as a $21c$-dimension feature vector, where $c = 256$ is the number of feature maps of the last convolutional layer. After concatenation, two fully connected layers are connected to this $21c$-dimension feature vector. Two different strategies can be adopted for training the CNN network of SPPnet: single-scale training and multiscale training. Single-scale training uses a fixed-size input (i.e., 224×224) wrapped from the input images. Multiscale training uses images of multiple different sizes, wherein each iteration, only the images of one scale are used for training the CNN network. The size of the input image is represented by $s \times s$, where s is uniformly sampled from 180 to 224. Because the multiscale training can simulate the varying sizes of images, it can improve detection accuracy. For training SVM classifiers, the specific steps are the same as that of RCNN. In the test stage, the image of an arbitrary scale can be put into SPPnet.

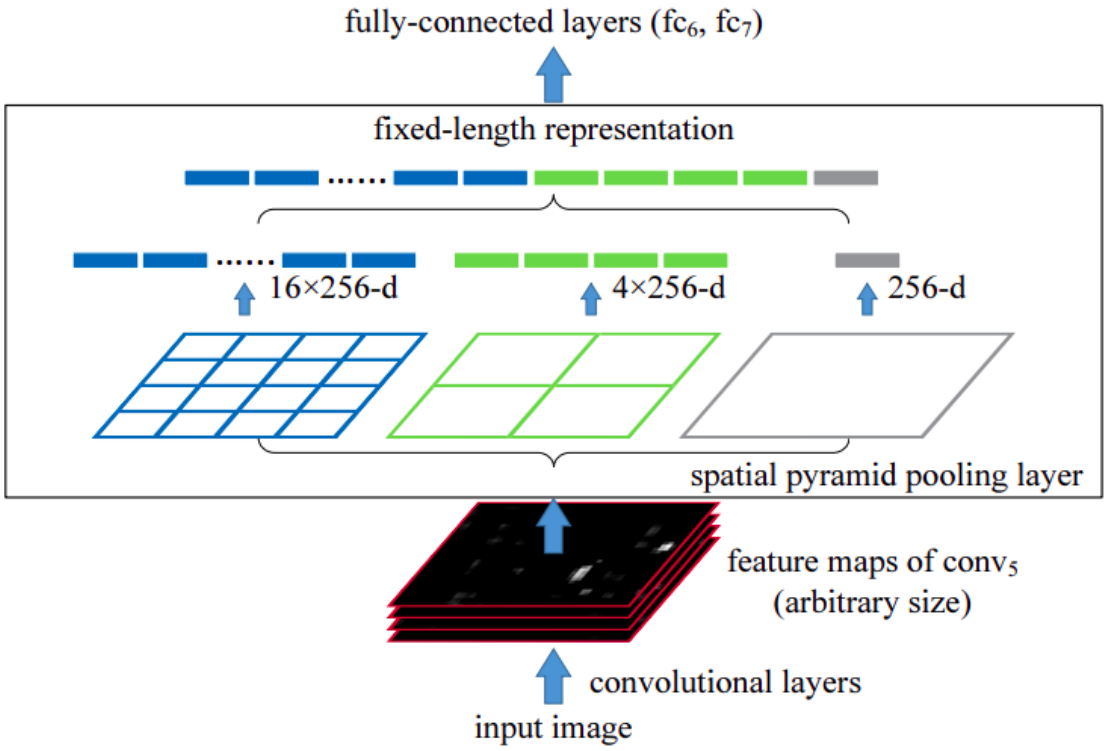


Figure 3.3: *Spatial pyramid pooling layer. The feature maps of a given proposal are pooled into 3×3 spatial bins, 2×2 spatial bins, and 1×1 spatial bins, respectively. After that, they are concatenated into a fixed-size feature vector and fed into two fully connected layers.*

Compared to RCNN, SPPnet has the following advantages: (1) All the candidate proposals share all the convolutional layers before the fully connected layers. Thus, it has a faster detection speed than R-CNN. (2) SPPnet uses multilevel spatial information of objects, which is more robust to object deformations. Meanwhile, multiscale training can enlarge the training data. Thus, it has a higher detection accuracy than R-CNN. Though SPPnet accelerates detection speed by sharing computation of all the convolutional layers, the training of SPPnet is still a multistage process similar to RCNN. Namely, they need to, respectively, fine-tune the CNN network on object detection dataset, train multiple SVM classifiers, and learn the bounding box regressors. To fix the disadvantages of R-CNN and SPPnet, Girshick and et. al, RCNN authors, further proposed **Fast RCNN**[21], which integrates the training of CNN networks, object classification, and bounding regression into a unified framework.

Fast RCNN

Figure 3.4 shows the architecture of Fast RCNN. Generally, Fast RCNN firstly calculates all the convolutional layers of the whole image. For each proposal, Fast RCNN uses an ROI pooling layer to extract the fixed-size feature maps from the feature maps of the last convolutional layer, and then feeds the fixed-size feature maps to two fully connected layers, and finally generates two sibling branches with a fully connected operation for object classification and box regression. The object classification part has c+1 outputs by softmax, where c means the number of object classes. The box regression part has $4c$ outputs, where every four outputs correspond to the box offset per class. The ROI pooling layer warps the feature maps of object proposals into the fixed-size spatial bins (e.g., 7×7) and uses a max-pooling operation to calculate the feature responses in each bin. Because Fast RCNN has two sibling outputs for object classification and box regression, the multitask training loss (i.e., L) is the joint of classification loss (i.e., L_{cls}) and regression loss (i.e.,L_{loc}) for each ROI as follows:

$$L\left(p,t\right) = L_{cls}\left(p,c\right) + \lambda\left[c \geq 1\right] L_{reg}\left(t,v\right) \tag{3.1}$$

where λ balances classification loss and regression loss and $[c \geq 1]$ equal one if the $c \geq 1$ and zero otherwise. Namely, the ROI belonging to the background class does not contribute to the regression loss. The classification loss $L_{cls}\left(p,c\right) = -\log p_c$ is the log loss for true class c. The regression loss L_{reg} is defined by the ground-truth regress target (i.e., (v_x, v_y, v_w, v_h)) and a predicted target (i.e., (u_x, u_y, u_w, u_h)) as follows:

$$L_{loc}\left(t,\ v\right) \ = \sum_{i\ \in\ x,y,w,h} smoothL_1\left(t_i, v_i\right) \tag{3.2}$$

where

$$smoothL_1\left(x\right) = \begin{cases} 0.5x^2, & \text{if } |x| \leq 1 \\ |x| - 0.5, & otherwise \end{cases} \tag{3.3}$$

Compared to $L2$ loss used in RCNN and SPPnet, the $L1$ loss is more robust to outliers.

For proposal generation, RCNN, SPPnet, and Fast RCNN are all based on selective search. The selective search method uses the handcrafted features and adopts the hierarchical grouping strategies to capture all possible object proposals. Generally, it runs at $2s$ per image on the common CPU. The detection network of Fast RCNN can run at about $100ms$ per image on the GPU. Thus, proposal generation of Fast RCNN is more time-consuming compared to the detection network of Fast RCNN. Though selective search can also be re-implemented on the GPU, proposal extraction is still isolated from the detection network of Fast RCNN. Thus, region proposal extraction becomes the bottleneck of Fast RCNN on object detection. To solve this problem, Ren et al. proposed **Faster RCNN** [22].

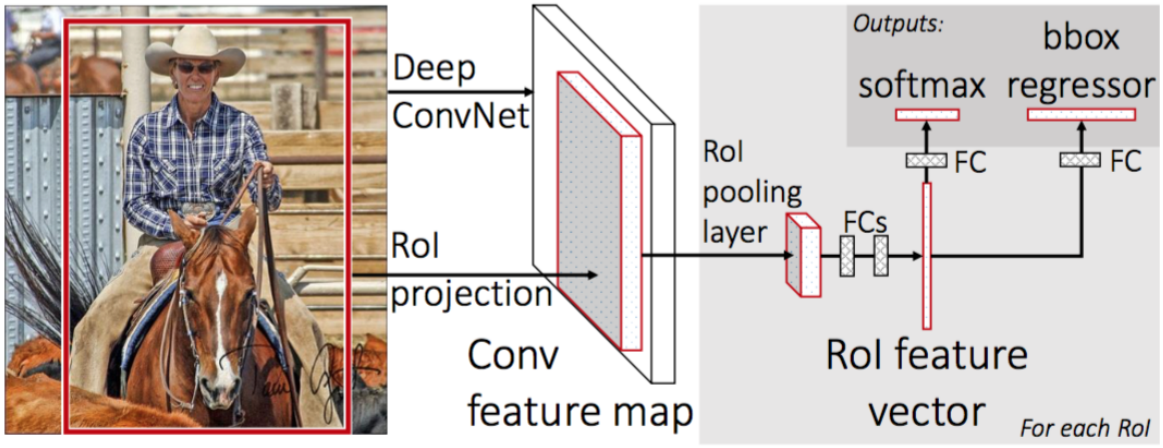


Figure 3.4: *The architecture of Fast RCNN. Compared to RCNN and SPPnet, it joins the classification and regression into a unified framework.[21]*

Faster RCNN

As shown in Figure 3.5, Faster RCNN architecture integrates proposal generation, proposal classification, and proposal regression into a unified network. It consists of two modules. One module, called **region proposal network** (i.e., **RPN**), is used to extract candidate object proposals. Another module is Fast RCNN, which aims to classify these proposals into specific categories and predict more accurate proposal locations. The two modules share the same base sub-network.

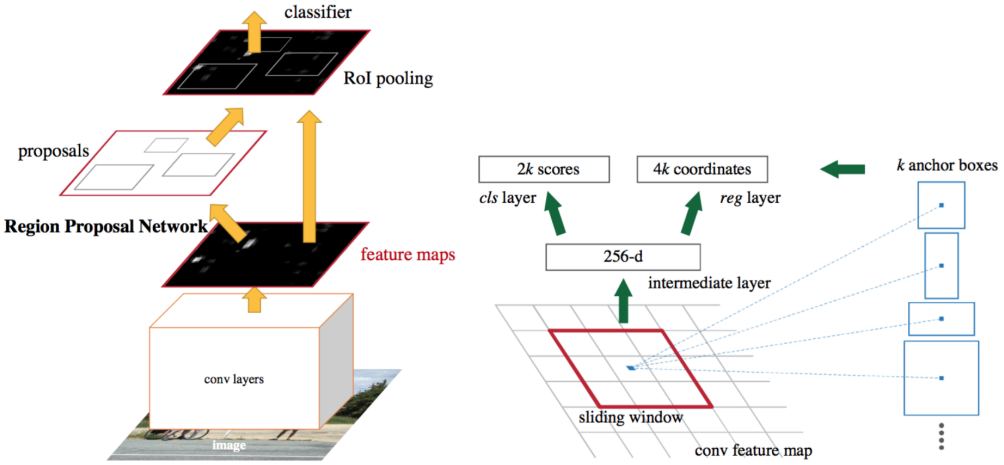


Figure 3.5: *The architecture of Faster RCNN. Proposal generation (RPN) and proposal classification (Fast RCNN) are integrated into a unified framework. [22]*

On the one hand, RPN can generate candidate object proposals using deep convolutional features, improving proposal location quality. On the other hand, Faster RCNN is an end-to-end framework with a multi-loss head. Compared to Fast RCNN, Faster RCNN can achieve better detection performance with much fewer proposals. RPN slides a small network over the output layer of the base network. The small network consists of one 3×3 convolutional layer and two siblings 1×1 convolutional layers for box regression and classification. Box classification is class-agnostic. For each sliding window location, RPN predicts multiple proposals based on the anchors of different aspect ratios and scales. Assuming that the number of anchors is k, the box

regression layer has $4k$ outputs for each sliding window, and the box classification layer has 2k outputs for each sliding window. Generally, three different aspect ratios of $\{1:2, 1:1, 2:1\}$ and three different scales of $\{0.5, 1, 2\}$ are used. Thus, there are nine (i.e., 3×3) anchors (i.e., $k = 9$) at each sliding window. The multitask loss of RPN consists of two parts: classification loss L_{cls} and regression loss L_{reg} which can be written as follows:

$$L(p, v) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, c_i) + \lambda \frac{1}{N_{reg}} [c_i \geq 1] L_{reg}(t_i, v_i) \tag{3.4}$$

where $N_{cls} = (256)$ and $N_{reg} = (\text{about } 2400)$ are the terms to, respectively, normalize classification loss and location loss, λ balances classification loss and regression loss, and $[c_i \geq 1]$ is 1 if $c_i \geq 1$ or 0 otherwise. The classification loss and regression loss are the same as that of Fast RCNN. If an anchor has an IoU overlap over 0.7 with any ground truth bounding boxes, the anchor is labeled as positive. If the anchors have an IoU overlap under 0.3 with all ground-truth bounding boxes, then the anchor is labeled as negative. The third range of overlap between 0.3 and 0.7 is left as neither negative nor positive.

R-FCN

Generally, the above state-of-the-art object detection methods (i.e., RCNN, SPPnet, Fast RCNN, and Faster RCNN) use the pre-trained CNN network on the image classification dataset (i.e., ImageNet). Dai et al. argued that this design has a dilemma to some degree. Generally, a deep CNN network for image classification favors translation invariance, while a deep CNN network for object detection needs translation variance. Dai et al. proposed R-FCN [23] to address the above dilemma between image classification and object detection. It encodes the object position information by the position-sensitive ROI pooling layer (PSROI) for the following Fast RCNN subnet. Figure 3.6 shows the architecture of R-FCN. Region proposal generation is the same as Faster RCNN. Based on the output layer of the original base network, R-FCN generates the new $k \times k$ position-sensitive convolutional banks. The convolutional banks correspond to the $k \times k$ spatial grids, respectively. In each convolutional bank, there are $c + 1$ convolutional layers (c means the number of object categories, and $+1$ means the background category). Namely, $k \times k \times (c+1)$ convolutional feature maps. At $k(C+1)$-d convolutional layer, a sibling $4k^2$-d convolutional layer is appended. Position-sensitive RoI pooling is performed on this bank of $4k^2$ maps, producing a $4k^2$-d vector for each RoI. $k^2(C+1)$ is for class prediction of the object whereas the $4k^2$-d is for bounding-box regression. Overall, R-FCN has a competitive performance with Faster RCNN while beating it in detection and training speed.

Mask RCNN

Most object detection methods only predict object locations by bounding boxes without providing more accurate segmentation information. In recent few years, some researchers proposed instance segmentation, which usually contains object detection and segmentation. Mask RCNN [24] is a famous method for instance segmentation and object detection. As shown in Figure 3.7, Mask RCNN architecture incorporates instance segmentation and object detection into a unified framework based on Faster RCNN architecture. Specifically, it adds an extra mask branch to predict the object's mask aside from the object classification and box regression branch. The mask branch has c binary masks with the size of $m \times m$. c means the number of object categories. The multi-loss head of Mask RCNN on each sampled ROI includes classification loss, regression loss, and mask loss. It can be represented as $L = L_{cls} + L_{reg} + L_{mask}$. The losses of L_{cls} and L_{reg} are the same as that of Faster RCNN. For an RoI proposal associated with the ground-truth class c, L_{mask} is only defined on the c^{th} mask and other mask outputs do not contribute to the loss. L_{mask} is defined as the average binary cross-entropy loss, only including c^{th} mask if the region is associated with the ground truth class c^{th}.

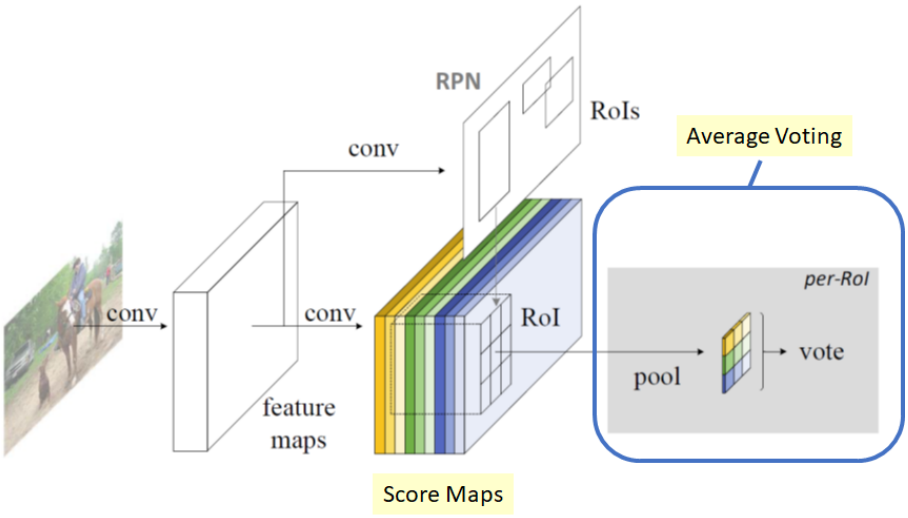


Figure 3.6: *The architecture of R-FCN. Position information is encoded into the network by position-sensitive ROI pooling (PSROI) [23]*

This design allows the network to generate masks for every class without competition among classes. In the test stage, the output mask of an object is determined by the predicted category of the classification branch. ROI pooling quantizes the floating number of ROI proposal locations into discrete values. The quantization of the ROI pool causes the misalignment between the input and the output, which negatively affects instance segmentation. To solve this problem, ROI-Align is proposed. It uses bilinear interpolation to compute the feature values of four corner locations in each spatial bin and then aggregates the feature response of each bin by max-pooling. Based on multitask learning, Mask RCNN can achieve state-of-the-art performance on object detection and instance segmentation. It means that joining the instance segmentation task with the object detection task can also help improve detection performance.

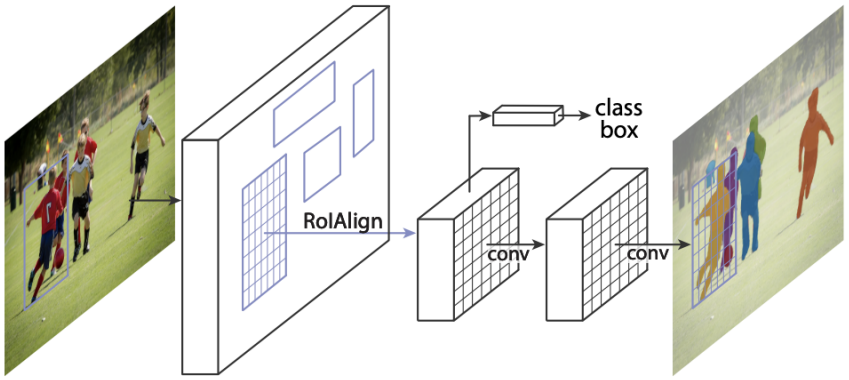


Figure 3.7: *The architecture of Mask RCNN. Apart from the detection branch of Faster RCNN, the extra branch of mask segmentation is added. [24]*

3.2.2 One-Stage Object Detection

One-stage methods aim to predict object category and object location simultaneously in one network and forward-pass without requiring a region proposal network. Compared to two-stage methods, one-stage methods have much faster detection speed and comparable detection ac-

curacy. Methods such as OverFeat [25], YOLO (YOLOv1[26], YOLOv2[27], YOLOv3[28]), SSD[29], and RetinaNet[30] are pioneers and successful one-stage detectors.

OverFeat

Overfeat is a pioneer model of integrating object detection, localization, and classification tasks into one convolutional neural network. It performs (i) multiple scales image classification at different locations on an image using a sliding window and (ii) regress bounding boxes from convolutional feature maps generated for classification using the sliding window technique. It has a model architecture similar to AlexNet but optimized for speed. Though it is a pioneer one-stage detector and ImageNet Large Scale Visual Recognition Challenge 2013 (ILSVRC2013) winner, its performance is inferior to current one-stage detectors.

YOLO

YOLO (also called YOLOv1 due to follow-up successive incremental versions), an abbreviation for You Only Look Once, is one of the first high-speed and high-performance one-stage object detectors. YOLO divides an entire image into $S \times S$ grid cells of area 32 by 32 pixels, and each grid cell detects B object bounding box coordinates, objectness confidence score, and class probabilities. The output layer of YOLO has a tensor of shape $S \times S \times (5B + C)$, meaning each of the $S \times S$ grids has a tensor of size $(5B + C)$ corresponding for the four bounding box parameters (i.e., b_x, b_y, b_w, b_h) plus one for objectness score and C for the number of class categories. The B signifies that each grid predicts B bounding boxes and confidence scores where B was two in the original implementation. In general, for an input image of size 224×224 and Pascal VOC dataset with 20 object classes, the YOLO outputs $7 \times 7 \times (5 \times 2 + 20)$. For this image size thus there will be 49 grids, and each grid predicts two bounding boxes, making 98 bounding boxes per image. Compared to RCNN and other two-stage networks we discussed above that generates over 2000 region proposals, the output of YOLO is minimal. As a result, YOLO processes relatively few bounding boxes making it one of the lightweight and fastest detectors. However, the downside of having a less dense output layer was a lower performance in today's networks standards though it was very competitive during its publication. Nonetheless, it was a state-of-the-art unified object detector that avoided the need for region proposal or complex pipelined object detection paradigm and yet came on top at its time. Figure 3.8 shows the architecture of the YOLO object detection model.

SSD

The other well-known one-stage detector is SSD [29], short for Single Shot Multibox Detection. SSD, like YOLO, presents a unified one-stage detector. However, instead of the grid-based approach of YOLO, the core feature of SSD is the multiscale object detection scheme they introduced by adding InceptioNet inspired convolutional filters on top of the VGG network. Due to these multiscale filters, SSD has better performance than YOLO in detecting objects of various scales, specifically small, medium, and large objects of the COCO dataset. Figure 3.9 shows the architecture of SSD. Assuming that the number of object classes is c and each feature map predicts k objects, it will result $(c + 4) \times k \times m \times n$ output vector for the given $m \times n$ feature maps. The objective loss of SSD is a weighted sum of location loss and confidence loss similar to that of Fast RCNN.

RetinaNet

In a bid of making one-stage detectors perform as well as two-stage object detectors in detection accuracy, a model dubbed RetinaNet [30] with its novel classification loss function called focal

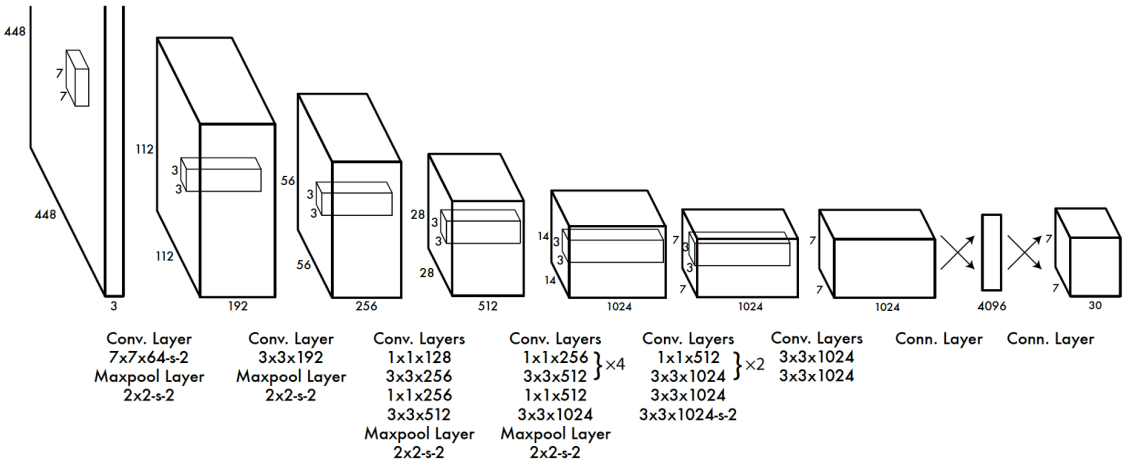


Figure 3.8: *YOLO has 24 convolutional layers followed by two fully connected layers. Alternating 1 × 1 convolutional layers reduces the features space from preceding layers. It is first trained on the ImageNet classification task at half the resolution (224 × 224 input image) and then doubles the resolution for detection. [26]*

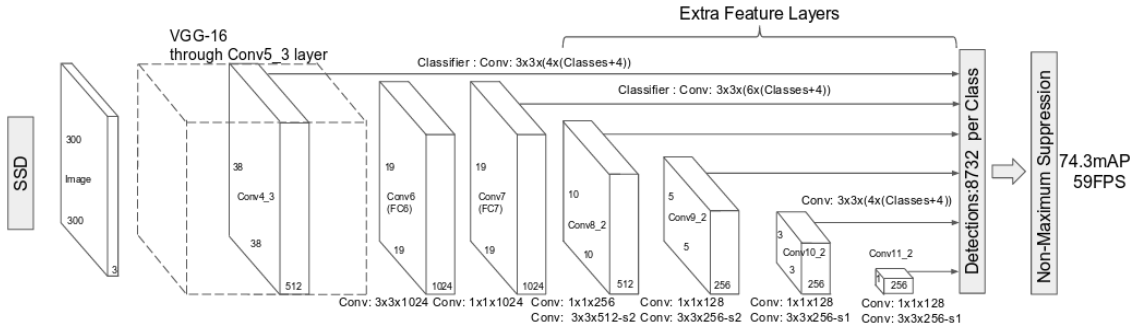


Figure 3.9: *The architecture of SSD. The base network is VGG16.[29]*

loss was proposed by Tsung-Yi Lin et al. of Facebooks AI Research. Their network predicts very dense bounding boxes, about $100k$ per image, and their cross-entropy modulating focal loss handles the class imbalance during training. The novel focal-loss proposed by RetinaNet was to solve the localization error, which is paramount in one-stage networks, by punishing the easy negative ones while simultaneously rewarding positive predictions. Focal loss is simply a modulation of cross-entropy loss, and it is given by equation 3.5.

$$FL\left(p_t\right) = -\alpha\left(1-p_t\right)^{\gamma}\log\left(p_t\right) \tag{3.5}$$

where

$$p_t = \begin{cases} p, & \text{if} c = 1 \\ 1-p, & \text{otherwise} \end{cases} \tag{3.6}$$

Figure 3.10 shows the architecture of RetinaNet with a base network is FPN. which constructs multiple feature maps. As seen from the figure, RetinaNet has two task-specific subnetworks, one subnet for performing convolutional object classification on the backbone's output and a second subnet for convolutional bounding box regression.

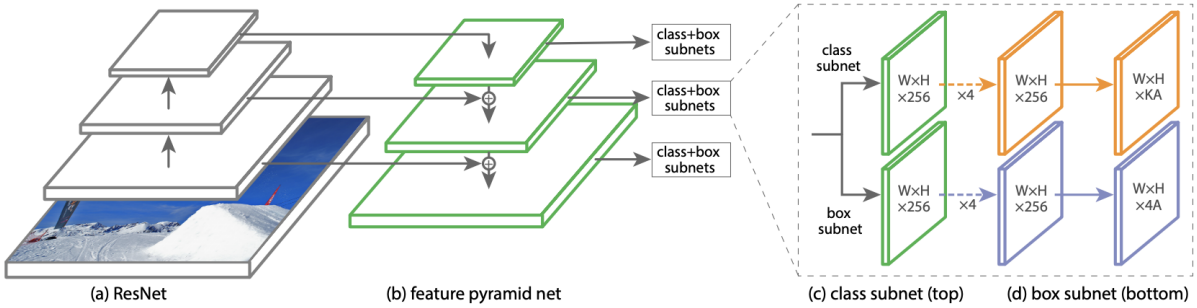


Figure 3.10: *The architecture of RetinaNet. The base network is FPN with ResNet classification backbone.[30]*

3.3 Object Tracking

Object tracking is usually treated as the next step of object detection since it involves detecting an object or objects in a video frame and locking on it (them) and following it(them) until the object(s) exits the video frames. Object tracking can be single or multi-object tracking based on the number of objects to be tracked. The primary difference between a tracker and detector model is that an object tracker must assign some form of a consistent and unique ID to the individual objects it is tracking and whereas an object detector is required to label the bounding box of the identified object without the need to assign unique ID consistently. For example, if a detector detects three cars in a frame, the object tracker must identify the three separate detections and track them across the subsequent frames with their unique IDs.

When compared to object detection, object tracking is more challenging due to 1) occlusion by other class objects, 2) occlusion by same class objects, 3) temporary frame exit and re-entry, 4) moving object scale variation, 5) non-stationary camera or viewpoint change and 6) tremendous hardship to find or annotate a tracking dataset. These are partial, not comprehensive, challenges of building a successful object tracker. For example, one essential yet most difficult challenge of building a successful tracker is the requirement of real-time object tracking. An object tracker fitted on an uncrewed vehicle or security surveillance system may not be helpful at all if it is not real-time, not to mention the danger it may pose. Due to these and many other reasons, we can safely say object tracking research is yet to mature despite the progress achieved on object detection.

Next, we shall see some of the most common and widely used object tracking techniques:

Mean Shift and Cam Shift

Mean shift [31] is a clustering algorithm that assigns the data points to the clusters iteratively by shifting points towards the mode. The mode can be understood as the highest density of data points and hence also called a mode-seeking algorithm. Given a set of data points, the algorithm iteratively assigns each datapoint towards the closest cluster centroid. Unlike the popular K-Means algorithm, meanshift does not require specifying the number of clusters in advance. The algorithm determines the number of clusters based on the data. When the mean shift is applied on a particular object(s) feature map (color, texture, histogram, or any other features of a particular object) on a given input frame, it will try to find all current locations of the previous frame's feature maps data points to locate the object(s). However, a moving object's size increases or decreases based on the direction of its movement with respect to the camera. This poses difficulty for the mean shift to locate all the data points of the feature maps. In such situations, mean shift is coupled with continuous adaptive mean shift (cam shift)[32] so that the tracking box shifts based on appearance change of the object under tracking.

Mean shift-based tracking is practical in situations with no noisy data, little or no occlusion, no significant change in illumination or color. An example would be tracking a package on a conveyor belt or in a warehouse where colors or textures remain constant. This method's best quality is that it does not require to be trained to perform object detection and then tracking.

OpticalFlow

Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by object or camera movement. It can also be defined as the distribution of apparent velocities of movement of brightness pattern in an image. In optical flow, an object is tracked using the spatio-temporal image brightness variations at a pixel level using the following assumptions as a precondition about a moving object:

- **Brightness consistency**: Brightness around a small region is assumed to remain nearly constant
- **Spatial coherence**: Neighboring points in the scene typically belong to the same surface and hence typically have similar motions
- **Temporal persistence**: Motion of a patch has a gradual change
- **Limited motion**: Points do not move very far or haphazardly

Kalman Filter

A Kalman filter is the most widely used traditional tracking algorithm still persistent in modern object tracking implementations. It recursively estimates the state of a linear system (also nonlinear system using extended Kalman filter or EKF) with a gaussian process given the previous state and the current measurement of a moving object to predict the object's next likely state (position). If one has a good object detector running along with the Kalman filter, then an object can be successfully tracked across video frames even if the object detector fails to detect the object sometimes since the Kalman filter can use the previous state of the object to predict the next state of the object. As a result, earlier object tracking methods such as SIFT, HOG, Fast Mean Shift, and Kalman filter was used to track an object.

The detailed discussion and derivation of the formulas of the Kalman filter and its use in object tracking can further be referred from [33, 34]. Here we present the overview of the Kalman filter and its application in object tracking. The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the priori estimates for the next time step. The measurement update equations are responsible for the feedback—i.e., for incorporating a new measurement into the priori estimate to obtain an improved a posteriori estimate. The time update equations can also be considered predictor equations, while the measurement update equations can be considered corrector equations. Indeed the final estimation algorithm resembles that of a predictor-corrector algorithm [35] for solving numerical problems, as shown below in Figure 3.11.

Suppose x is the state, z is the measurement, w is process noise, v is measurement noise, P is covariance error, and they are all Gaussian. The noises w and v are independent of states and measurements. Then the complete Kalman filter equation of the two processes (predict and correct) shown in figure 3.12 can be mathematically given using equations in Table 2. Q and R stand for process noise and measurement noise covariance, respectively, whereas K is Kalman filter gain.

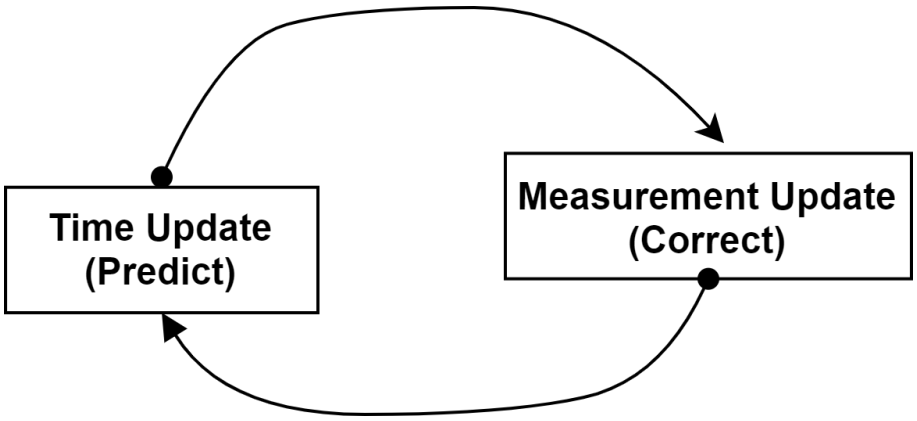


Figure 3.11: *The ongoing discrete Kalman filter cycle. The time update projects the current state estimate ahead of time. The measurement update adjusts the projected estimate by an actual measurement at that time.*

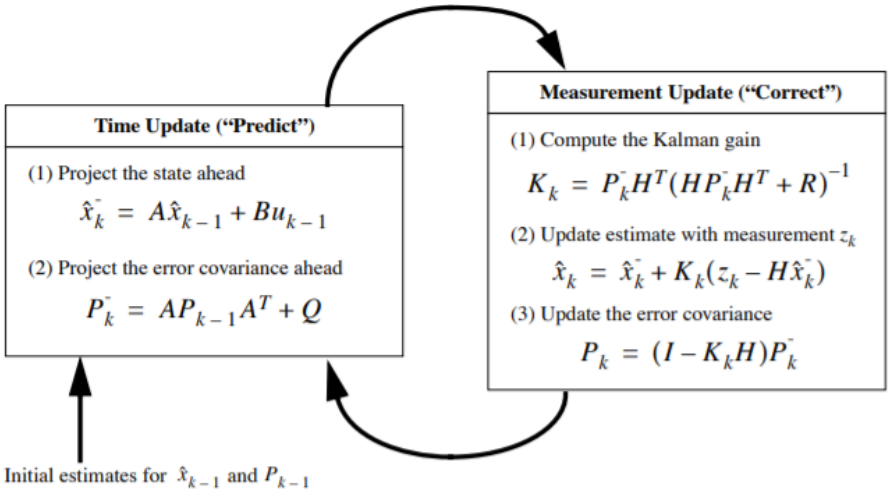


Figure 3.12: *A complete picture of the operation of the Kalman filter illustrated mathematically*

In general, the Kalman filter is a handy algorithm with applications ranging from object tracking to any applications where state prediction is required. It is usually used with other algorithms that help to distinguish one object from another object, like a detector based on neural networks or feature descriptors such as SIFT or even a background subtraction and mean shift algorithm.

SORT (Simple Online Real-Time Tracker)

SORT [36] presents a lean implementation of tracking by detection framework for the problem of multiple object tracking (MOT), where objects are detected in each frame and represented as bounding boxes. It is an online tracking system where only detections from the previous and the current frame are required to track an object, emphasizing real-time tracking speed for pedestrian tracking or autonomous driving problems. SORT has four key components:

1. **Object detection**: - CNN based appearance descriptor pre-trained on large person-re-identification dataset is used to detect person appearances (feature vectors) from frame to frame

2. **Estimation model**:- Here, the object model, i.e., the representation and the motion model based on Kalman-filter is used to propagate a target's identity into the next frame. SORT approximates the inter-frame displacements of each object with a linear constant velocity model independent of other objects and camera motion. The state of each target is modeled as:

$$x = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}]^T$$

   where u and v represent the horizontal and vertical pixel location of the center of the target, while the scale s and r represent the scale (area) and the aspect ratio of the target's bounding box, respectively. Note that the aspect ratio is considered to be constant. When detection is associated with a target, the detected bounding box updates the target state where the velocity components are solved optimally via a Kalman filter. If no detection is associated with the target, its state is predicted without correction using the linear velocity model.

3. **Data association**:- In assigning detections to existing targets, each target's bounding box geometry is estimated by predicting its new location in the current frame. The assignment cost matrix is then computed as the intersection-over-union (IoU) distance between each detection and all predicted bounding boxes from the existing targets. The assignment is solved optimally using the Hungarian algorithm. Additionally, a minimum IoU is imposed to reject assignments where the detection to target overlap is less than IoU_{min}.

4. **Creation and deletion of track identities**: When objects enter and leave the image, unique identities need to be created or destroyed accordingly. In SORT, an overlap less than IoU_{min} signifies the existence of an untracked object. The tracker is initialized using the geometry of the bounding box with the velocity set to zero. Additionally, the new tracker then undergoes a probationary period where the target needs to be associated with detections to accumulate enough evidence in order to prevent the tracking of false positives. On the opposite, tracks are terminated if an object is lost for more than T_{Lost} time or frames. This prevents an unbounded growth in the number of trackers and localization errors caused by predictions over long durations without corrections from the detector.

Overall, a simple SORT algorithm in conjunction with Kalman filter achieved a commendable performance in the MOT (multi-object tracking) challenge in 2016, just the second-best performer in the competition while still working at a very high speed.

Deep Learning-Based Object Tracking

Due to the success of CNN-based deep learning methods in object detection, researchers have been looking for ways to harvest from this success and develop a tracking algorithm that benefits from deep learning object detectors. One of the early methods that used deep learning for single object tracking is **Deep Regression-Based Object Tracking**. In this method, a DL model is trained on a dataset consisting of videos with labeled target frames. The objective of the model is to track a given object from the given image crop. To achieve this, they use a two-frame CNN architecture which uses both the current and the previous frame to regress onto the object accurately. It is a single object tracking method, where a user first sets the object to be tracked. Another elegant method to track an object is **ROLO** [37], a combination of YOLO

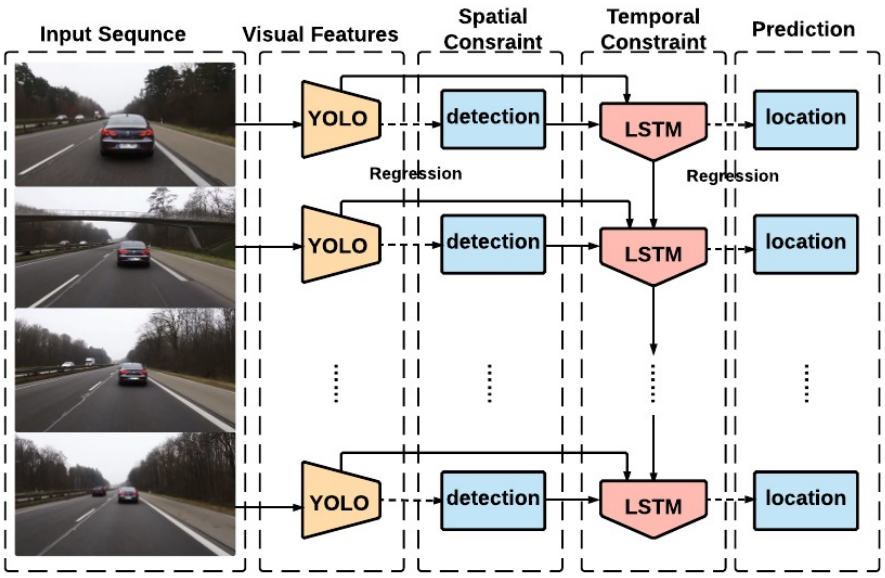


Figure 3.13: *A simplified overview of ROLO and the tracking procedure [37]*

and recurrent LSTM. The overview of ROLO depicted in figure 3.13, or its architecture in figure 3.14, shows that YOLO detections are concatenated and passed onto the LSTM cell for spatial information recording and tracking. The most popular and widely used, elegant object tracking

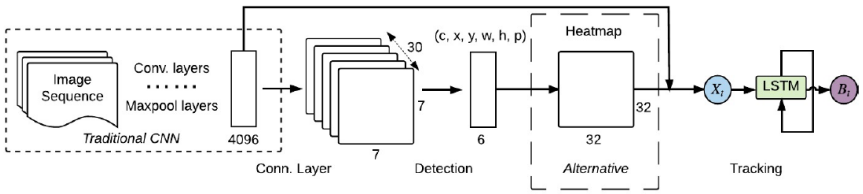


Figure 3.14: *ROLO network architecture[37]*

framework is **Deep SORT** [36], an extension to SORT (Simple Real-time Tracker). It has all the four components of SORT and one more deep learning generated component called appearance descriptor. The appearance descriptor is introduced to minimize the identity switch problem prevalent in the SORT algorithm. In SORT, the association between two objects from two successive frames is based on bounding box IoU regression only (distance calculation); however, when two similar class objects overlap, it is easy to mistake one object bounding for another hence case identity switch. In Deep SORT, in addition to the IoU overlap-based distance calculation, cosine loss between appearance description vectors from one frame to another is also considered to associate objects. Moreover, instead of Faster RCNN as an object detection background,

current fast and more accurate detectors such as YOLOv4 can be used with the Deep SORT algorithm.

3.4 Conclusion

In this chapter, we thoroughly reviewed state-of-the-art object detection and tracking approaches. We classified the current object detection trends into two well-known categories based on the approaches to implementing the detection networks. These two categories are called two-stage object detection and one-stage object detection. Two-stage object detectors perform detection in two tightly coupled stages. The first stage is a region of interest proposal or pooling, where thousands of likely objects of interest containing regions of an image are pooled and wrapped together to be fed to the next stage. The second stage is the classification and localization stage, where the wrapped regions are passed through several layers, most probably CNN layers, and scored on their likelihood of containing objects. Non-Max-Suppression and thresholding methods filter out weak predictions while preserving the most confident ones. The other set of modern object detection methods is based on one-stage detection in which classification of object class and regression of their location is done in one unified network in a single forward pass. These models are fast, lightweight, and suitable for embedded system applications. However, they are relatively less accurate since they don't have a separate network to propose regions and likely locations of objects. As a result, these types of object detectors are the center of our research work. In the coming chapters, we present different, more accurate, lightweight, and fast one-stage detectors followed by hardware acceleration implementation

Chapter

4 FPGA as Hardware Accelerator for Object Detection and Tracking

Chapter content

FPGA as Hardware Accelerator for Object Detection and Tracking

4.1 Introduction

As explained in previous chapters, deep learning techniques, mainly based on CNN, dominate computer vision applications, with breakthroughs announced nearly every month. The pace of innovation in this field is astounding. It means that new, previously unobtainable applications are likely to be enabled within the next few years. However, the challenge with CNNs is the intense amount of computation required - commonly requiring multiple TeraOPS. As a result, deep CNNs are mainly trained on a cluster of computers with high-performance GPUs, thereby consuming high power and resources. Low power and resource-constrained small electronics such as embedded systems have benefited little from the accuracy-wise advancement the deep learning is going through as the networks' depth, size, and complication kept increasing as well.

Recently, Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) are getting more attention to balance the tradeoff between the high-power demand of GPUs and the intensive computation demand of deep convolution-based learning networks. Though FPGAs and ASICs hardly reach the same or increased throughput as GPUs, their power is limited. However, compared to FPGAs, the high cost and long development period of ASICs make them unable to keep up with the rapid changes of deep CNNs. As a result, FPGA-based deep CNNs are becoming a center of attention for lightweight and real-time deep CNNs for embedded systems.

Though FPGAs are relatively power efficient and have a high level of parallelism, they have relatively more minor resources such as memory and are comparatively challenging to program. To maximize the memory and power efficiency of deep CNNs on FPGAs, recently, various methods have been proposed by researchers worldwide. These optimizations focus on reducing the size of deep CNN networks and increasing ways to exploit FPGAs' parallelism feature. Later, we will discuss some of the well adopted techniques of FPGA-based deep CNN accelerations but first let us briefly explore why one wants to consider FPGA for deep CNN acceleration.

4.2 Why FPGA?

1. **FPGAs are energy efficient**: Though Deep Learning is a resource and power-hungry task, FPGAs can provide an energy-efficient acceleration due to their customizability for a specific model and power-up required resources only [38]. FPGA can be flexibly and variably configured to use any bit size, whereas GPUs cannot be customized to use variable data sizes; they are generally suitable for either 32-bit or 64-bit floating-point precisions. The lower the data representation, the faster the computation.

2. **FPGAs have high flexibility**: It is a well-founded fact that FPGAs are highly flexible development tools. The area of Deep Learning or AI is a very fast-changing one. FPGAs' reconfigurability and re-programmability accommodate these continuous improvements and can be reprogrammed many times, saving costs. This advantage is relevant when compared with ASIC-based development, given their long development time and lack of reusability or re-programmability.

3. **FPGAs can yield high-throughput**: Since FPGAs, support variable bit quantization and deep learnings are being proved to work just fine with extreme quantization, as extreme as one or two-bit quantization, the throughput of FPGA based deep-learnings are bound to be in par or more than that of GPUs. Moreover, FPGA bandwidth is many orders faster than GPUs, meaning low latency computation for the most compute-intensive operation, the convolution layer.

Though the pros mentioned above entail good prospects for FPGA-based Deep CNN acceleration, there are some cons. These shortcomings include the relative difficulty in programming FPGA, complexity and the need for special expert knowledge in the underlying hardware, lack of free or less-cost tools and libraries for test and development, and finally, the cost of the FPGA boards themselves.

4.3 Short Review of FPGA Based Deep Learning Acceleration

Over the years, many authors have proposed and tried different alternatives for accelerating CNN-based networks, particularly the convolution layer. An extensive review of hardware acceleration methods from multiple points of view can be read from review works of [**empty citation**]. Here we briefly summarize some well-known approaches by grouping them into three subsets. 1) Optimizations based on convolution mathematics transformation 2) Optimizations based on exploiting the parallelism of convolution nested loop execution 3) Optimization based on lightweight implementation.

4.3.1 Optimizations based on transforming the algorithm of convolution

Optimizations in this category focuses on replacing the standard convolution algorithm altogether with faster algorithms such as Fast Fourier Transform (FFT) [39–41], Winograd[42, 43] or treating convolution as matrix multiplication or GEMM [44]. [39] first proposed to transform

convolution algorithm from spatially sliding kernel over input feature maps to pointwise products in the Fourier domain while reusing the same transformed feature map many times, see Figure 4.1. At the time, this method was implemented on GPU and proved to have superior performance over traditional convolution algorithm. However, it has two problems 1) it requires the inputs always to be in 2'power for optimal performance gain and thus requires zero-padding 2) kernels must be large or needs to be padded to have the same size as the input feature map. Modern kernels are spatially tiny compared to input feature maps, 1×1 or 3×3 rendering [39]'s implementation in its originality less prevalent in modern deep CNN acceleration. Instead, [40] added the concept of Overlap-and-Add to accelerate the FFT-based implementation of convolution further and also addressed the impact oc size imbalance between input feature map and kernels. Their solution is implemented on FPGA to accelerate the AlexNet model and proved successful.

On the other hand, Winograd-based methods implement convolution as Hadamard product or Element-Wise Matrix Multiplication (EWMM). Unlike FFT, they are more suitable for smaller kernels and can benefit from standard loop optimization techniques such as unrolling and pipelining. Since Winograd-based methods efficiency is associated with smaller tile sizes, on-chip buffering aided with double-buffering is necessary to reduce memory access for bringing tiles or buffering partial results.

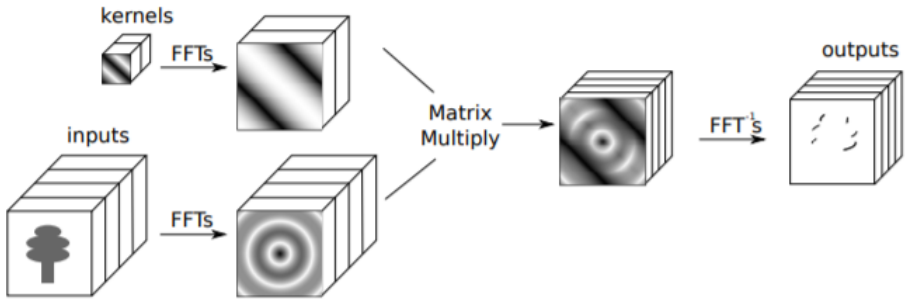


Figure 4.1: *Illustration of the convolution algorithm using FFTs [39]*

4.3.2 Loop Optimizations

The second set of optimizations exploits the different types of parallelism in the convolution's nested loops. The parallelism can be manifested between layers or within a layer but among batch-size, channel, or spatial dimensions. However, because of the resource limitation of FPGA devices, it is impossible to exploit all the parallelism patterns at once. Instead, the primary approach that state-of-the-art methods use is to map a certain level of parallel operations onto a limited number of Processing Elements (PEs) and reuse these PEs by iterating data. We find two unique approaches under this category of optimization, 1) systolic array of PEs and 2) Single Instruction on Multiple Data (SIMD) based dynamic loop optimization.

Early FPGA-based accelerators for CNN [**chakradhar2010dynamically**, 45, 46] implemented and exploited static collection of PEs, called systolic arrays, typically arranged in a 2-dimensional grid operating under the control of a CPU **??**. These static collections of PEs are independent of the kernel size of a given layer. They hence can be inefficient or incapable of implementing the convolution layer if the layer kernel is greater than the number of implemented PE sizes. Moreover, the lack of data buffering in these early systolic arrays increased memory access frequency, reducing their throughput. The inefficient hardware resource utilization coupled with high-power consumption renders this systolic array-based acceleration method less favorable in modern deep CNN acceleration.

Due to the inefficiency of static systolic arrays, flexible SIMD accelerators for CNNs on

FPGAs were proposed [47–49]. The general computation flow in these accelerators is to fetch FMs and weights from DRAM to on-chip buffers. These data are then streamed into the PEs. At the end of the PE computation, results are transferred back to on-chip buffers and, if necessary, to the external memory to be fetched in their turn to process the subsequent layers. Each PE is configurable and has its computational capabilities using DSP blocs and data caching capabilities employing on-chip registers. Standard loop optimization techniques accelerate the deep CNN layers based on the number of available resources, such as DSPs, logic cells, and on-chip BRAM memories. The number of PEs is dependent on the selected loop optimization techniques. The detail of loop optimization techniques such as loop unrolling, loop pipelining and loop order interchange as applied to CNN layers acceleration on FPGA will be discussed further in Chapter 8, but here we will introduce them briefly.

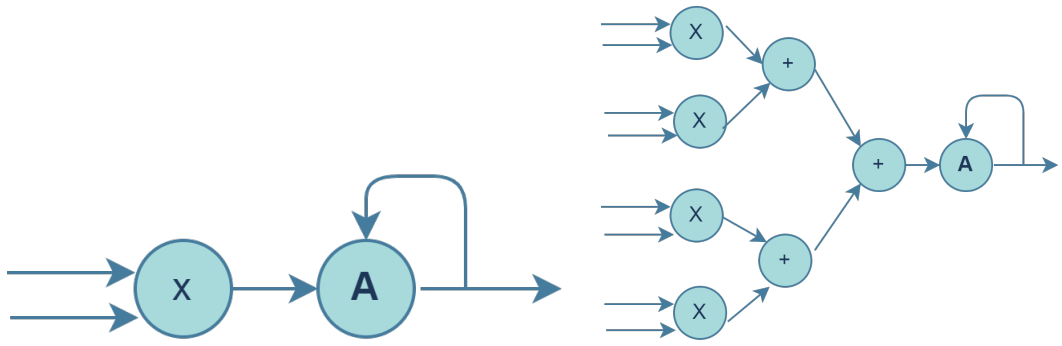
Loop Unrolling

Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, an approach known as space-time tradeoff. The goal of loop unwinding is to increase a program's speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration, reducing branch penalties; as well as hiding latencies, including the delay in reading data from memory.

Consider the following two simple for-loops side by side. As seen from the left for loop, the elements of array A depend only on the corresponding index elements of arrays B and C. One can implement the left side loop as in the right-side reducing the number of loop iteration check and implementing certain lines to run in parallel thereby increasing overall throughput of the loop execution.

```
for (i = 0; i < N; i++) {
  A[i]=B[i]*C[i];
}
```

```
for (i = 0; i < N; i=i+5) {
  A[i]=B[i]*C[i];
  A[i+1]=B[i+1]*C[i+1];
  A[i+2]=B[i+2]*C[i+2];
  A[i+3]=B[i+3]*C[i+3];
  A[i+4]=B[i+4]*C[i+4];
}
```



(a) *Without unrolling* **(b)** *Unrolled loop*

Figure 4.2: *Loop Unrolling. 4.2a 1 data-path, 1 sample per iteration and total of N iterations. 4.2b 4 data-path, 4 samples per iteration and N/4 total iterations.*

The above two implementations of the for loop will give the same result, except the second version is faster by a small factor as there will be less loop iteration, condition check, and branch penalty. Moreover, by duplicating the processes on different processing elements (PEs) in FPGA implementation, one can utilize the parallelism feature for further speed gain. The size of the loop unrolling depends on the number of resources of the FPGA and the speed gain of loop-unrolling

depends on the maximum data-path width. Loop unrolling also has its own disadvantages, such as increased program size and high power consumption due to intensive parallel computation. Moreover, one should also take into account the unrolling factor and the maximum loop size for efficient implementation.

Loop Tiling

One primary limitation of FPGA is the on-chip memory, and as a consequence, it fails to store all intermediate and input data of deep CNNs. Therefore, it is usually necessary to use external DRAMs to store the input and output feature maps, weights, bias, and a partial results from CNN layers. Therefore, loop-tiling thus the process of breaking down these data into small atomic blocks the FPGA can handle.

Below pseudocode listing show a loop before and after tiling is implemented. The inner block on the left side operates on atomic data residing on the FPGA's on-chip buffer.

```
for (i = 0; i < 100; i++) {
  ...
}
```

```
for (j = 0; j < N; j+=B) {
  for (i = j; i < min(N, j+B); i++) {
    ...
  }
}
```

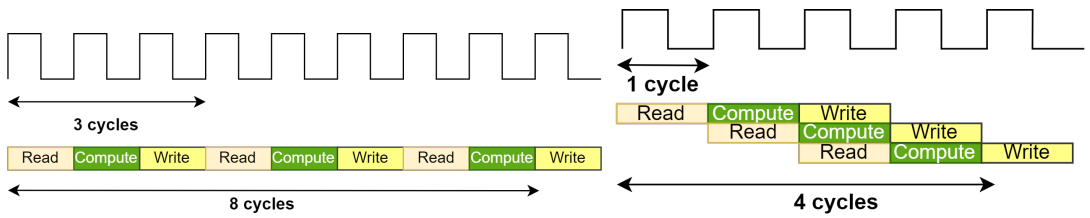
Even though loop-tiling is mandatory for optimizing convolutional neural networks on FPGAs due to the limitation of FPGA on-chip memory, it also comes with its challenges and design parameters that need careful selection. The design parameter that needs careful deliberation is the block size, as too small a block increases communication between on-chip and off-chip memory, and too big block wastes energy.

Loop Pipelining

Pipelining a loop allows the operations of the loop to be implemented concurrently, as shown in the following figure below. In the figure 4.3, part (A) shows the default sequential operation where there are three clock cycles between each input read, and it requires eight clock cycles before the final output write is performed, whereas only four clock cycles are required when using pipelining, as shown in part B of the figure. Pipelining is a valuable optimization technique, but it is constrained by instructions having dependence on each other and by the available resource.

In general, all the loop optimization techniques we discussed here (unrolling, tiling, and pipelining) can be implemented either by the programmer for every loop or automatically by the compiler, such as HLS pragmas

```
for (i = 0; i < 2; i++) {
  read();
  compute();
  write();
}
```



(a) *Without pipelining* **(b)** *With pipelining*

Figure 4.3: *Loop pipelining*

4.3.3 Optimization based on lightweight implementation

The other set of optimization focus on trimming the trained network using methods such as model quantization, pruning, compression and sparse convolution. Pruning generally means cutting down parts of the network that contribute less or nothing to the network during inference. This results in smaller models, more memory-efficient, more power-efficient, and faster networks during inference with minimal loss of accuracy. Due to the loss of connections and neurons, pruning usually results in a slight loss of accuracy compared with the unpruned network. It is customary to cyclically retrain or fine-tune a pruned network to revive the accuracy drop. The cycle continues until the fine-tuning stops improving or starts to overfit. However, the most common optimization is based on data quantization. Research has shown that the accuracy drop of using fixed-point operations is insignificant, especially considering the vast resource efficient utilization quantization can render. Nowadays, the most common FPGA implementations are 16 and 8-bit quantization of the 32-bit floating-point trained model. Extreme quantizations such as 1, 3, and 4-bit are also being proposed and some commendable performance and significantly efficient memory utilization are reported by many authors.

In summary, current hardware-acceleration implementations utilize one or more of the above techniques for maximum throughput, efficient resource utilization, and low-power consumption while maintaining accuracy drop as minimum as possible though usually these objectives conflict and results in trading one over the other.

4.4 Hardware-accelerated deep CNN design and implementation flow

Every deep CNN-based AI solution has two main stages: Training and Inference. Since the training stage requires a large dataset (usually in tens or hundreds of gigabytes), a computer system with a high-performance GPU cluster is used to create and train models in floating-point precisions. Many popular frameworks and tools exist to create deep CNN models prominently as Tensorflow, Pytorch, Caffe, and Theano are the usual ones to mention. Once a network is trained, and performance requirements are met, an optional stage of pruning and compression follows to yield a lightweight and faster model for embedded systems. However, pruning usually reduces the accuracy of a network and is thus accompanied by a cyclic fine-tuning until the performance reaches the unpruned network or stops improving. Whether pruning is done, the trained network weight, bias, and other hyperparameters are exported and quantized before hardware acceleration. Quantization is not mandatory for hardware implementation but is the most common first step in implementing energy-efficient and high-throughput hardware-accelerated inference of the deep CNN model. It also helps to reduce the hardware resource consumption of the inference model. Quantization can be done using a custom algorithm on the exported model parameters (weights and biases) using any preferred programing language such as C/C++ or can be done using Tensorflow's quantization aware training as part of training or fine-tuning. Nonetheless, the trained floating or the fixed-point quantized weight and bais must be exported and saved separately for the hardware-accelerated inference stage.

Based on the board or software vendor, one can use different IDEs to implement the accelerator for the hardware inference implementation. Full-fledged IDEs such as Intel Quartus for Intel FPGAs and Vitis and Vivado for Xilinx FPGA provide full support for creating, synthesizing, debugging, implementing, and deploying the hardware acceleration deep CNN functions. Figure 4.4 shows the Vitis hardware design flow based on C/C++ and Vitis HLS compiler. As seen from the figure, one first writes C/C++ code for inference of the deep CNN, in our case, the object detection inference, and also a C-testbench to evaluate the implemented inference C/C++ code. Once the C-simulation passes, then hardware synthesis with certain design constraints such as target clock, target device, and some other configurations of interfaces, if necessary, is

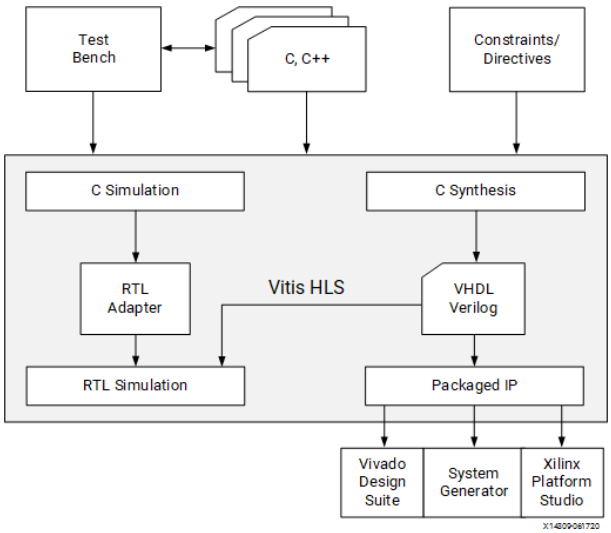


Figure 4.4: *Xilinx Vitis HLS Design Flow*

configured and synthesized. Suppose the synthesis passes and the design constraint and target performance are satisfied. One can then export the custom design as an IP block or proceed with more optimization steps to increase throughput (reduce latency), optimize resource utilization, and eliminate inefficient design choices with better ones for optimum implementation.

Moreover, synthesized hardware does not necessarily mean it works as expected and is verified by the C-simulation. Instead, one needs to perform a further simulation called C/RTL co-simulation to test whether synthesized hardware works as identical as the software version. Once all this is over, the design is exported and implemented as an IP block. The custom accelerated IP block with other IPs such as host processors, AXI interfaces to DDR memory, clock generator wizards, post and pre-process modules, etcetera, are bundled into one embedded system hardware implementation. Though we tried to view the end-to-end stages very briefly, each section, in reality, is extensive and has its dos-and-don'ts which are beyond the scope of this section. However, one can find detailed resources on each topic presented by Xilinx and other FPGA vendors in great detail, spread over different manuals and tutorials. Figure 4.5 summarizes our above discussions and the end-to-end model creation, training weights and bias, quantizing, and designing hardware acceleration to deployment on the FPGA board.

4.5 Conclusion

This chapter briefly summarized different object detection inference acceleration techniques using FPGAs. We categorized prior acceleration implementations into three categories and referenced some representative works by prior researchers falling into each category. The first category of inference accelerations implementations focuses on transforming the standard convolution operations into faster matrix multiplication or other approximate processes such as frequency domain Fourier transforms. The most common technique is optimizing the standard convolution implementation using loop optimizations such as pipelining, unrolling, and loop order interchanges. These loop optimization coupled with various data quantization, network pruning, and compression techniques comprises the modern object detection inference acceleration. Our object detection inference acceleration discussed in chapter eight details these and other state-of-the-art concepts in our quest to implement our object detection acceleration using some test development boards in our lab.

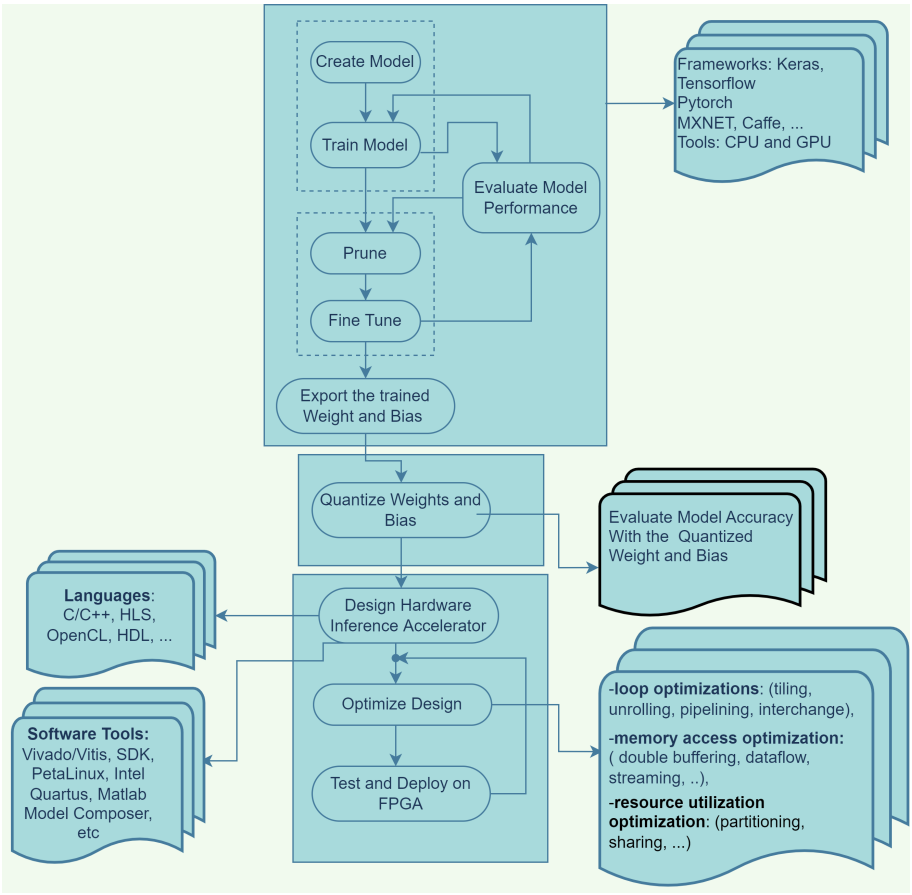


Figure 4.5: *End-to-end Deep CNN-based hardware inference acceleration overall implementation stages, tools and languages*

Part II

Experiments and Major Contributions

Chapter

5 Lightweight Generic Object Detector using binary encoding

Chapter content

Lightweight Generic Object Detector using binary encoding

5.1 Objectives

Object detection is one of the most challenging and very important branch of computer vision. Some of the challenging aspect of a detection network is the fact that an object can appear anywhere in the image, be partially occluded by another object, might appear in crowd or have greatly varying scales. Consequently, we propose a fine grained and equally spaced dense grid cells throughout an input image be responsible of detecting an object. We re-purpose an already existing deep state-of-the-art detector or classifier into deep and dense detector. Our dense object detector uses binary class encoding and hence suitable for very large multi-class object detector. We also propose a more flexible and robust non-max suppression implementation to filter out redundant detection of same object. As a result of our dense object detection implementation we have managed to increase YOLOv2's performance on Pascal VOC 2007 and COCO datasets by +2.3% and +7.2% mean average precision (mAP) respectively.

5.2 Introduction

Object detection is one of the most challenging and very important branch of computer vision. Unlike object classifiers that only seek to label an image in its entirety into one of predefined categories, object detectors, however, are required to precisely locate all known objects on an image using bounding box around each object and label them correctly.

The trend in current state-of-the-art object detection networks is to use a deep convolutional classifier network as background extractor. As pointed in [16], these trends of using deeper network is best explained by the growing trend of using families of the sophisticated and cumbersome ResNet-based classifiers in almost all best performing state-of-the-art object detectors at a cost of speed and resource expense.

In this paper we propose to densifying the output layer of a detection network in such a way that detection occur at fine grained equally spaced grid cells all over the input image. This is a similar approach to YOLO[26] except YOLO uses bigger grid cells such as 32 by 32 whereas in our case we propose using smaller grids of size 8 by 8. Densification alone might not yield a significant performance gain as it leads to overfitting, and in fact might also render implementation unfeasible due to large memory requirement on the output layer. We remedy this using a modified form of binary encoding which we refer it as **extended binary encoding** as opposed to the de-facto onehot encoding of categorical data. We also propose a new classification loss function for object detection to both handle class imbalance and foreground and background grid cell imbalance. Lastly but not least, we also propose a new and robust non-max suppression implementation suitable for filtering less confident and redundant bounding boxes detection. Our network relies on deep convolutional classifier or any high performance detector network as feature extractor backbone for improved performance. In this paper, though any high performance network could have been re-purposed into our dense detector, however, we chose YOLOv2 for its simplicity and of course no mean performance.

5.3 Related Works

OverFeat[25] for the first time modeled object detection as one stage, meaning one forward pass to determine both the class of the object and their localization, regression multi-scale sliding window problem. Since then many successful one stage object detectors followed the suite. One such network is YOLO[26] and its incrementally improved trilogies, namely YOLOv1[26], YOLOv2[27] and YOLOv3[28]. In general, YOLO divides an input image into equal grid cells, particularly 32 by 32 pixels, and each grid cell predicts one or more objects with confidence scores. Though YOLOv1 and YOLOv2 are very fast however, compared to similar scale one stage or two stage object detectors of recent time, their performance trails behind. Recently YOLOv3 by including some best practice of the current object detection trend such as using very deep feature extractor and employing concepts such as residual network, skip connection, multi-scale prediction and up sampling, they managed to significantly improve YOLO's performance.

The other well-known one stage detector is SSD [29], short for Single Shot Multibox Detection. SSD, like YOLO, presents a unified one stage detector. However, instead of grid based approach of YOLO, the core feature of SSD is the multi-scale object detection scheme they introduced by adding InceptioNet inspired convolutional filters on top of VGG network. Due to these multi-scale filters SSD was much better than YOLO in detecting objects of various scales, specifically small, medium and large objects of COCO dataset.

Most recently, in a bid of making one stage detectors perform as well as two stage object detectors in detection accuracy, a model dubbed RetinaNet [30] with its novel classification loss function called focal loss, was proposed by Tsung-Yi Lin, et al of Facebooks AI Research. Their network predicts very dense bounding boxes, about 100k per image, and their cross entropy modulating focal loss handles the class imbalance during training. However, due to the underlying

ResNet network and the dense bounding box prediction,the speed of RetinaNet is significantly slow than the above two models.

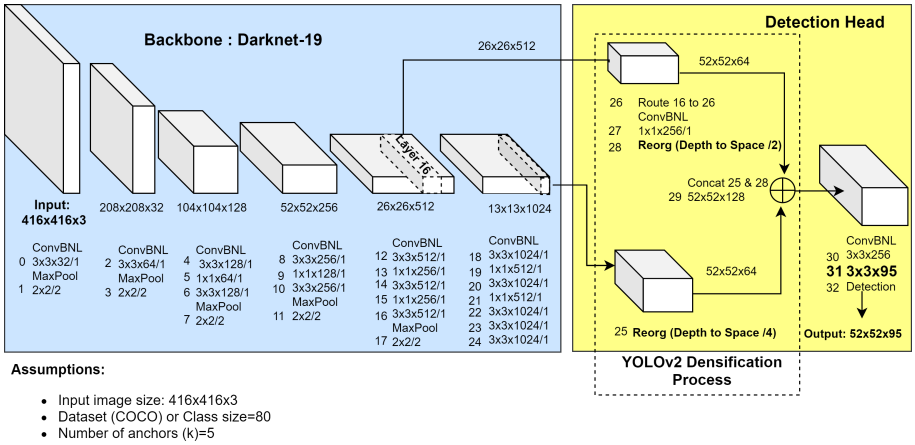


Figure 5.1: ***Dense Detection Grid Network (DDGNet).*** *YOLOv2 used as base model after stripping the last detection layer.*

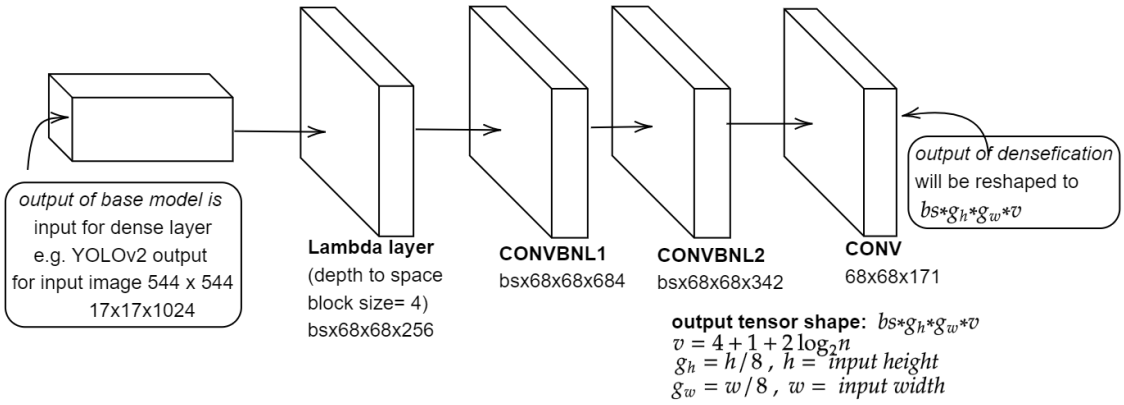


Figure 5.2: ***Grid densification.*** *After striping off the last detection layer of YOLOv2 output layer we then pass it through CONVBNL1, CONVBNL2 and CONV layers to give dense output. CONVBNL* stands for Convolution $\longrightarrow$ Batch Normalization $\longrightarrow$ LeakyRelu layers.*

5.4 Densification: Generation of Dense Detection Grids

As per [50] in COCO dataset more than 41% of the objects have size below 32 by 32 and each YOLO grids cells have an area of 32 by 32. With this grid cell size, it is possible that objects of smaller sizes appearing in group on an image , such as flock of birds or bucket of fruits can fall in the same grid cell. Though rare, two or more partially occluding objects might also fall in the same grid cell and same anchor box. When such cases happen during ground truth annotation, since any anchor per grid only annotates one object the other will be dropped out. Moreover, during inference again since one anchor per grid can only detect at a maximum one object objects of near same size and appearing in a crowd or partially occluded will be dropped out.

Tiny dense grids reduces the chance of overlap of ground truth annotations and helps in deep search of a particular image for an object at every location. In our dense grid implementation shown in Fig. 5.1, the output of the YOLOv2 network minus the last layer is fed to depth-to-space reshaping Lambda function with block size of 4 and then passed through additional light weight three convolutional blocks as shown in Fig. 5.2 where the third convolutional layer is the

Class Index	Extended Binary Encoding, n=80													
	One's Complement (width=7)							Binary (width=7)						
	6	*5*	*4*	*3*	*2*	*1*	*0*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	1	1	1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	0	1	0	0	0	0	0	1	0
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
79	0	1	0	1	1	1	1	1	0	1	0	0	0	0

Table 5.1: ***Extended Binary Encoding.*** *For class size $n = 80$ we need $m = 2 * log_2 n = 14$ size array to encode a class.*

new output head of the network. The middle two added convolutional blocks are to counter the bottleneck layer that might be created due to the folding of the depth feature map into small sized space feature map, though we merely reorder the same feature map. Since we use an 8 by 8 grid cell, our network predicts four time as many bounding as YOLOV2 detects. Due to this dense grid and dense predictinon we refer our network as **dense detection grid network** or **DDGNet**.

Unlike the de facto onehot encoding of categorical objects, we used *extended binary encoding* in our ground-truth class encoding. For n classes of nominal categorical objects, onehot encoding requires an array of size n for each object, whereas binary encoding requires only an array of size $\log_2 n$. However, since binary encoding uses fewer features to encode a class, it is expected to loose some information in describing an object when compared with onehot encoding. To compensate this information loss, we followed a novel approach to extend the binary encoding of an object by combining binary encoding of class index with its one's complement so that the final class encoding will have a size of $m = 2 * \log_2 n$. The extension of binary encoding with one's complement, as shown in table 5.1, is a simple hack to compensate the feature loss in binary encoding and it also balances the number of zeros and ones in a given class encoding.

In summary, our DDGNet final layer outputs a feature map of shape $bs * g_w * g_h * k * (4 + 1 + 2 * \log_2 n)$, where bs stands for batch size, g_w and g_h stands for total number of horizontal and vertical grid cells respectively, k for number of anchor boxes and the expression $(4+1+2*\log_2 n)$ refers to the coordinate of the bounding boxes, objectness score of each anchors per grid and the class encoding array. We followed the same approach used by YOLOv2 to annotate our ground truth of our training and validation dataset. We used $k = 9$ in our experiment.

5.5 Training

DDGNet training loss function is structurally similar with YOLO. The loss function has three parts: box coordinate prediction loss, the objectness confidence prediction loss and class encoding prediction loss.

5.5.1 Coordinate loss

The coordinate loss is given by the following equation, where (x_{ij}, y_{ij}) and (w_{ij}, h_{ij}) are the ground truth bounding box center and width and height pair resized by the original image width and height respectively. The i and j stands for the grid cell and anchor box within the grid cell respectively.

$$
\begin{aligned}
crd_{loss} = \lambda_{crd} \sum_{i=0}^{g_w * g_h} \sum_{j=0}^{k} \mathbb{1}_{ij}^{obj} \left[(x_{ij} - \hat{x}_{ij})^2 + (y_{ij} - \hat{y}_{ij})^2 \right] + \\
\lambda_{crd} \sum_{i=0}^{g_w * g_h} \sum_{j=0}^{k} \mathbb{1}_{ij}^{obj} \left[(w_{ij} - \hat{w}_{ij})^2 + \left(h_{ij} - \hat{h}_{ij} \right)^2 \right]
\end{aligned} \tag{5.1}
$$

$\mathbb{1}_{ij}^{obj}$ is the masking tensor for each anchor j of grid cell i. $\mathbb{1}_{ij}^{obj}$ is set to 1 if anchor j of grid cell i has the highest IOU overlap with the ground truth of a given object; otherwise is set to 0. λ_{crd} is coordinate loss scaling coefficient which we always set to 1. However, one can give different weights for the center coordinate loss and width and height prediction loss.

5.5.2 Objectness loss

Objectness confidence score is the measure of the probability that a given cell has detected an object. Equation (5.2) shows the objectness loss function used in DDGNet. $\sigma\left(\hat{t}_0\right)$ is the networks predicted box objectness confidence score. λ_{obj} is a weight coefficient stressing the cell responsible of detecting an object whereas λ_{noobj} is a weight coefficient set to reduce the effect of cells that are not expected to detect an object. Normally, since the majority of the cells doesn't have an object λ_{obj} must be very large compared to λ_{noobj}. In our training we experimented by setting $\lambda_{obj} = 10000$ and $\lambda_{noobj} = 10$. $IOU(b, \hat{b}) < IOU_{thrsh}$ is the IOU computation of predicted bounding box against the ground truth bounding boxes. $conf_{loss}$ is computed over all grids and anchor boxes per each grid and averaged by the total number of object detecting grid cells, however we showed a minimal representation of the equation for brevity.

$$
\begin{aligned}
conf_{loss} = & \lambda_{obj} \mathbb{1}_{ij}^{obj} \left(1 - \sigma\left(\hat{t}_0\right)\right)^2 + \\
& \lambda_{noobj} \mathbb{1}_{ij}^{noobj} \left(\sigma\left(\hat{t}_0\right)\right)^2 * IOU(b, \hat{b}) < IOU_{thrsh}
\end{aligned} \tag{5.2}
$$

5.5.3 Class prediction Loss: Ternary Cross Entropy (TCE) Loss

Before explaining our classification loss function we would like to recall binary cross entropy (BCE) loss for binary classification. BCE loss is calculated using the following equation:

$$
BCE(c, \hat{c}) = \begin{cases} -\log(\hat{c}), & \text{if } c = 1 \\ -\log(1 - \hat{c}), & \text{if } c = 0 \end{cases} \tag{5.3}
$$

In the equation (5.3) the $c \in \{0, 1\}$ is the binary ground-truth class annotation of the object to be detected whereas $\hat{c} \in [0, 1]$ is the predicted class probability of the detected object. The first term in BCE equation trains to predict binary 1 where as the second term trains to predict binary 0. However, in our case since we conduct multi-label classification, multi-label to say more than one label can be value 1 according to their extended binary encoding, we added a third term where we train the network to output a value 0.5 for grids that have no object of interest. In our dataset we don't have object annotated with 0.5 instead all objects are annotated as explained in table 5.1. Since 0.5 is the equally furthest number from both 0 and 1, we push grids that has no object to output 0.5 while pushing the other object containing grids to their extended binary encoding. Due to this three values, that is 0, 0.5 and 1, we refer our classification loss function as **ternary cross entropy** or TCE for short. Equation (5.4) shows simplified the TCE loss implemented at a given cell. c and $\hat{c}$ are ground-truth class encoding array and the sigmoid of the class prediction output of the network respectively at a given cell.

Detection Frameworks	Train	mAP
Fast R-CNN [22]	2007+2012	70
YOLO [26]	2007+2012	63.4
SSD300 [29]	2007+2012	74.3
SSD500 citessd	2007+2012	76.8
YOLOv2 416 [27]	2007+2012	76.8
YOLOv2 544 [27]	2007+2012	78.6
DDGNet 544 ours	2007+2012	**81.3**

Table 5.2: ***Performance comparison on Pascal VOC 2007 test dataset against some equivalent models.*** *As seen from the table at 544 by 544 input image DDGNet outperforms the equivalent networks in the table.*

$$TCE(c,\hat{c}) = \begin{cases} -\log(\hat{c}+0.5) - \log(1.5-\hat{c}), & \text{if } \mathbb{1}^{obj} = 0 \\ \begin{cases} -\log(\hat{c}), & \text{if } c = 1 \\ -\log(1-\hat{c}), & \text{if } c = 0 \end{cases}, & \text{if } \mathbb{1}^{obj} = 1 \end{cases} \tag{5.4}$$

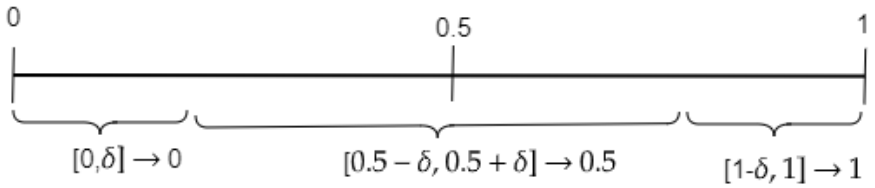


Figure 5.3: *Rounding class encoding prediction to either of the three value 0, 0.5 and 1. If the predicted value is in the range of* $[0,\delta]$ *we round the predicted value to 0, if it is in range* $(0.5-\delta, 0.5+\delta)$ *then we round to 0.5 and if in the range* $[0.5+\delta, 1]$ *we round it to 1.*

5.6 Inference

For single input image our network outputs a tensor $\hat{y}$ of shape $g_h{*}g_w{*}k{*}(4{+}1{+}c_m)$, where g_h, g_w stands for number of grid cells in height and width direction of the input image,respectively, and k is the number of anchor boxes whereas the term $(4+1+c_m)$ stands for the four parameters of bounding box coordinates, 1 grid cell objectness score or probability measure and c_m for the m size extended binary class encoding prediction.

As in as in [27] we calculate bounding box coordinates relative to grid cell from the network output corresponding the four coordinate values t_x, t_y, t_w, t_h and corresponding grid coordinate (c_x, c_y) that detected the bounding box as shown below. The (p_w, p_h) stands for the width and height of the anchor box detecting the object.

$$b_x = \sigma(t_x) + c_x \tag{5.5}$$

$$b_y = \sigma(t_y) + c_y \tag{5.6}$$

$$b_w = p_w e^{tw} \tag{5.7}$$

$$b_h = p_h e^{t_h} \tag{5.8}$$

After sigmoid we the class prediction output to the ternary values, namely 0, 0.5 or 1, using a threshold value δ chosen from range of $[0, 0.25]$. Fig. 5.3 shows the rounding process whereas equation (5.9) is the mathematical expression of the rounding process.

$$\hat{c}_j = \begin{cases} 0, & \text{if } \hat{c}_j < \delta \\ 0.5, & \text{if } \delta < \hat{c}_j \leq \delta + 0.5 \\ 1, & \text{if } \hat{c}_j > 0.5 + \delta \end{cases} \tag{5.9}$$

Once class predictions are rounded into the ternary points, then we measure **weighted euclidean distance** between predicted class encoding and class encoding of each object category of our ground-truth classes using equation (5.10).

$$d_i^2 = \sum_{j=0}^{m} (\frac{2^j}{n}) \, (\hat{c}_j - c_{ij})^2 \tag{5.10}$$

$$predclass_{index} = argmax(-d_i^2), \forall i = \{0, n-1\} \tag{5.11}$$

We also measure classification confidence score using the following equation:

$$clfn_{conf} = 1 - \frac{1}{m} \sum_{j=0}^{m} \begin{cases} 1 - \hat{c}_j, & \text{if } \hat{c}_j \geq 0.5 + \delta \\ \hat{c}_j, & \hat{c}_j \leq \delta \\ 1, & 0.5 - \delta \leq \hat{c}_j \leq 0.5 + \delta \end{cases} \tag{5.12}$$

The classification confidence score is computed from the un-rounded direct class encoding prediction output of the network. The classification confidence score simply shows how well the network class prediction is close to the extended binary encoding of the objects. It is not a probability measure of confidence of the class prediction certainty, like the common softmax or sigmoid based classifications.

Finally, we compute overall object prediction confidence score using (5.13) from classification confidence score and grid objectness confidence score g_{conf} which is simply $g_{conf} = \sigma(t_o)$, where t_o is the corresponding objectenss output of the network.

$$O_{conf} = clfn_{conf} * g_{conf} \tag{5.13}$$

Bounding boxes with overall score above certain threshold are then selected as detected object and then pass through non max suppression for further removal of redundant prediction of the same object. However, one can also set individual threshold for classification, objectness score and overall score and use the equation (5.14) to mask detected objects.

$$\begin{aligned} fltr_{mask} =& (clfn_{conf} < clfn_{thrsh}) * (g_{conf} < g_{thrsh}) * \\ & (O_{conf} < O_{thrsh}) \end{aligned} \tag{5.14}$$

5.6.1 Three Way Non-Max Suppression (3WayNMS)

Non max suppression (NMS), as its name suggests, is a mechanism of successively discarding the less confident of two bounding boxes whose IOU overlaps above certain IOU threshold, IoU_{thrsh}. However, picking a single IoU_{thrsh} for all object scale and appearance is painstaking trail and error procedure. In this paper we reduce that burden with a robust and flexible NMS implementation we dubbed **3WayNMS**. Our NMS implementation employs the usage of two IOU thresholds and a two other thresholds, namely **Intersection over smaller box area** in short **IOS** or **Intersection over larger box area** in short **IOL** threshold given by equation (5.15) and (5.16).

Let IoU_{thrsh1} and IoU_{thrsh2}, where $IoU_{thrsh1} > IoU_{thrsh2}$, be the two IOU threshold values and IoS_{thrsh} IOS threshold and IoL_{thrsh} IOL threshold. Now consider two overlapping bounding boxes, A and B, of same class. Using the following three steps we can check if the boxes are redundant detection of same object and hence leading to suppression of the less confident of the two bounding box:

1. if $IOU(A, B) > IOU_{thrsh1}$, then remove either of the box whose overall confidence score is smaller.

2. if $IoU_{thrsh2} \leq IOU(A, B) < IoU_{thrsh1}$ and the center coordinates of box A and B are within r grids apart of one another in all direction, then we remove the box with smaller confidence score. In our case we used $r = 4$, meaning a grid within maximum 4 grids of one another in any direction are considered neighbors.

3. if $IOU(A, B)$ is less than IoU_{thrsh1} but higher than IoU_{thrsh2} though the center coordinates of box A and B are not within r grids cells away of one another, if $IOS(A, B) > IoS_{thrsh}$ then remove the box with smaller grid or if $IOL(A, B) > IoL_{thrsh}$ then again remove the box with smaller threshold. The meaning of this is, if two boxes overlap above certain small threshold, that is IoU_{thrsh2} but fail from meeting the required IoU_{thrsh1}, then we check how much percent of the smaller or the larger box is in the intersection. If the smaller object's, say 75% of the area is in the intersection then we remove one of the two boxes having smaller score. However, though not frequently true we also check the ration of the intersection to the bigger area and if some part of less confident large box intersect with more confident large above certain threshold, say 50% then we remove the less confident box.

$$IoS = Intersecion(A, B)/min(area_A, area_B) \tag{5.15}$$

$$IoL = Intersecion(A, B)/max(area_A, area_B) \tag{5.16}$$

Due to these three possible stages or ways of discarding the less confident of redundant bounding box prediction, we named our suppression technique as Three Way Non-Max Suppression or 3WayNMS for short.

5.7 Experiment

We conducted numerous experiments on Pascal VOC and COCO object detection datasets in order to compare the performance of DDGNet against the original YOLOv2 and other related one stage object detectors such as SSD and RetinaNet.

To start our training we used YOLOv2's weight file of both COCO and Pascal VOC datataset from the authors website [51]. In our all test experiments we used small learning rate $1e^{-4}$ for the first 30 epochs and $1e^{-5}$ for the last 70 epochs, in total we trained for 100 epochs with early stop in place when the mean average precision stops improving. Random combination of common image augmentation techniques such as cropping,rotating, hue and saturation manipulation and different noise additions are implemented using *Imgaug* package [52] to artificially boost our training dataset.

5.7.1 Performance on Pascal VOC Object Detection Dataset

We trained on combined 2007 and 2012 Pascal VOC training dataset and tested the performance of our system on Pascal VOC 2007 test set.

As seen on Table 7.1 DDGNet trained with input image size 544 by 544 on combined Pascal VOC 2007 and 2012 shows a performance gain of +2.7% over the original YOLOv2 proving

Table 5.3: *Individual Pascal VOC 2007 dataset classes mAP. DDGNet works better in almost all objects the other networks struggles with.*

Model	data	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
Faster [22]	07+12	73.2	76.5	79.0	70.9	65.5	52.1	83.1	84.7	86.4	52.0	81.9	65.7	84.8	84.6	77.5	76.7	38.8	73.6	73.9	83.0	72.6
SSD300 [29]	07+12	74.3	75.5	80.2	72.3	66.3	47.6	83.0	84.2	86.1	54.7	78.3	73.9	84.5	85.3	82.6	76.2	48.6	73.6	73.9	83.4	74.0
SSD512 [29]	07+12	76.8	82.4	84.7	78.4	73.8	53.2	86.2	87.5	86.0	57.8	83.1	70.2	84.9	85.2	83.9	79.7	50.3	77.9	73.9	82.5	75.3
DDGNet 544	07+12	**81.3**	81.9	86.1	76.3	74.5	71.3	85.3	88.0	87.7	72.4	83.9	87.4	83.8	86.3	85.1	81.2	66.2	74.6	91.4	82.6	80.1

densification and TCE loss useful. In addition trained on, our model also out performs SSD300 [29] and SSD500 [29] by +7% and +4.5% respectively. Table 7.2 shows detail mAP performance of DDGNet against SSD300, SSD500 and against two stage object detector Faster RCNN [22]. From the table we can see that DDGNet works much better on usually occluded or partially visible and small objects such as boat, bottle, chair, table, sofa and potted plants when all other detectors struggle with these objects.

Models	**AP**	**AP50**	**AP75**	**APS**	**APM**	**APL**
Two-stage methods						
Faster R-CNN+++ [20]	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [53]	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [54]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [55]	36.8	57.7	39.2	16.2	39.8	52.1
One-stage methods						
YOLOv2 [27]	21.6	44.0	19.2	5.0	22.4	35.5
YOLOv3 [28] 608	33.0	57.9	34.4	18.3	35.4	41.9
DSSD513 [56]	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [30]	40.8	61.1	44.1	24.1	44.2	51.2
DDGNet 608(ours)	28.8	51.2	30.8	12.1	39.4	50.7

Table 5.4: *COCO dataset test result. Though DDGNet sweepingly outperforms the underlying YOLOv2, but it still remains behind the rest current state-of-the-art including behind YOLOv3.*

5.7.2 Performance on COCO Object Detection Dataset

We also trained DDGNet on COCO dataset using COCO object detection evaluation metrics. Table 7.3 shows comparison of DDGNet mean average precision against some well known and recent state-of-the-art detectors. Our DDGNet implementation again shows over +7.2% gain against YOLOv2. However, against the other recent networks DDGNet trails far behind, especially in precisely detecting of smaller objects. In fact our underlying backbone, YOLOv2, lacks some of the recent object detection networks best working and proved features proved for this purpose such as skip connection, residual network and up-sampling, which the authors included in YOLOv3. Accordingly, we believe if recent best performing base networks are used as backbone of DDGNet the detection of small objects would improve.

5.8 Conclusion

In this paper we have introduced dense object detector obtained after restructuring and adding few extra layers to an already deep and high performing backbone network. We proposed dense grid based one stage object detector in a assumption that dense and fine grained grid cells better annotates and detects crowded or occluded objects and smaller objects, consequently increasing performance of an object detector. As hypothesized we have managed to increase one of the state-of-the-art object detector YOLOv2 without inducing significant weight or reduce the speed of YOLOv2 significantly. Moreover, since our detector uses optimized class encoding it is suitable for very large multi-class object detection and we believe as detection networks are going deeper and heavier the way forward to support very large multi-class detection is probably through the use of binary encoding as hardware always has limit. We have also introduced new bounding box classification loss function called ternary cross entropy to handle the overwhelming imbalance of grids responsible of detecting an object and those which are not. We have also introduced

a new and flexible non-max suppression implementation to remove redundant detection of the same object.

Chapter

6 Yet Faster, Lighter and More Accurate YOLO

Chapter content

Yet Faster, Lighter and More Accurate YOLO

6.1 Objective

As much as an object detector should be accurate, it also has to be light and fast. However, determining acceptable tradeoff between speed and accuracy is not a simple task. One of the high speed and high performance object detector is the well known YOLOv2. YOLOv2 performs detection by dividing an input image into grids and training each grid cell to predict certain number of object bounding box parameters and their corresponding classes and objectness confidence score. We re-frame this multiple object prediction per grid cell approach of YOLOv2 into a single object prediction per grid cell by using smaller area grid cells. We train our system not only to predict box parameters and classes but also to pick an appropriate ⌋anchor box from a set of K-means generated anchors, or prior boxes, so that the predicted bounding box parameters yield higher IOU overlap against the ground-truth box. For the same input size and grid cell area, our DenseYOLO approach has $k-1$ folds of less parameters on the output layer compared to YOLOv2's output layer, assuming k is the number of anchor boxes or priors. This reduced output layer parameters directly translates to increased detection speed due to less number of redundant prediction of same object or small false-positive predictions and in general smaller

computation. Moreover, since our approach has fewer output parameter one can use large anchor box to start training the system from good initial status or increase image resolution or large class category without worrying the drastic burst of the output layer.

6.2 Introduction

One of the most important and challenging computer vision area is object detection. It is a basic first step for many real-world computer vision and robotics applications such as face detection, object tracking, pose analysis, surveillance, information retrieval and etc. Object detection has a dual goal of both precisely labeling and localizing all known objects on an image using rectangular bounding boxes.

Modern era generic object detection researches are dominated by two deep CNN based approaches. These are implementation of object detection as a two-stage or one-stage detector. Two-stage detectors perform object detection in two core steps, that is first using a serious of CNN propose sparse candidate regions on an image, likely containing an object of interest and second classify and score each proposed regions. One-stage detectors, however, perform both the localization and classification simultaneously in one forward pass. Generally, two-stage networks are known for their high performance in detection accuracy though they are usually very slow and heavy. On the contrary, one-stage detectors are very-fast while relatively are also less accurate. Since we believe that an object detection should be fast, accurate and light weight,in this paper we will focus on one-stage object detectors.

YOLO [26], an abbreviation for You Only Look Once, is one of the first high speed and high performance one-stage object detectors. YOLO divides an entire image into square grid cells of an area 32 by 32 and each grid cells detect a number of object bounding box coordinates and classes. Though YOLO, also called YOLOv1, was very fast however its detection accuracy was not in par with recent detectors. Thus, in a quest of increasing the detection performance of YOLOv1, a new model with more depth and other useful features dubbed YOLOv2[27] is proposed by authors of YOLO. YOLOv2 has more depth and predicts more bounding boxes per image compared to YOLOv1. Each grid cell in YOLOv2 predicts 5 bounding box coordinates, object classes and objectness confidence score for each prediction. Theoretically if one uses dense grid cells with smaller area than 32 by 32 or increased the number of boxes each grid cell predicts, say from 5 to 10 or more, then the chance of detecting partially occluded objects, objects appearing in group and objects of smaller size should increase. These constitutes some of the drawbacks of YOLOv2. However, increasing the density of grid cells by using smaller area grid cells, like 8 by 8, or increasing the number of boxes each grid cell predicts will drastically increase the output layer of YOLOv2 which in turn makes the network either slow or demand large memory for training.

In this paper we present a new more robust and efficient approach of implementing YOLO like, more specifically YOLOv2 like, one-stage object detection. Instead of predicting k bounding boxes per grid cells, we train our model to predict only one object per grid cell. We use smaller grid cells, an 8 by 8 grid cell instead of the 32 by 32 used in the YOLO trilogies [26], [27] and [28]. Since the grid cells are tiny it is unlikely for two or more objects to fall in the same grid cell thereby increasing the chance of detecting objects appearing in group such as flock of bird or a bucket of fruits or partially occluded objects − the type of objects YOLOv2 fails to handle very well. In YOLOv2 K-means generated k number of anchor boxes are made to predict k bounding boxes per grid cell, though only one anchor box is annotated as responsible of detecting an object during ground truth annotation. This leaves the other $k-1$ anchors per grid cells unnecessarily bloat the output layer into consuming more memory location or resort into redundant prediction of same object multiple times. In our method, which we dubbed **DenseYOLO**, instead of predicting k bounding boxes using each anchor, we train the system to predict one bounding box coordinate, in fact we predict offset from anchor boxes rather than the

coordinates directly, per grid and also predict a single anchor that yields the highest IOU overlap against the corresponding ground-truth. In short, just like each grid cell predicts coordinates and classes it also predicts an anchor to be used with the predict coordinate so that the predicted box fits the underlying ground-truth. This reduces the output layer of DenseYOLO by atleast $k-1$ fold compared to YOLOv2 and thus DenseYOLO is lighter and faster. Since the output layer is smaller in DenseYOLO, one can use large number of K-means generated anchor boxes to start the system train from more dataset representative set of anchors or prior boxes. Large set of anchor boxes, also called prior boxes, help to handle appearing in unusual scale or shape−− the other problem YOLOv2 cannot handle very well. Moreover, one can easily convert best performing deep classifiers such as ResNet[57] or VGG[58] or even YOLOv3 into DenseYOLO based detector for an increased the detection performance and speed.

6.3 Related Works

Classical object detection works have been relying on hand crafted features and shallow networks for very long time. However, the success of using deep convolutional methods for object classification and detection revolutionized by AlexNet[59], triggered deep CNNs to conquer modern object detection researches. As presented in the introduction section, current trend in object detection follows either two stage detection approach or one-stage detection. Both have their own advantages over the other and usually models based on the former approach are more accurate whereas models based on the later are significantly faster. Determining trade-of between detection speed, memory and accuracy is a prime concern in object detection. As [54] proposed, two-stage detectors can be speedup by using smaller image resolution or light and deep classifier as a backbone.

Recently, a one-stage object detection model called RetinaNet[57] targeted solving class imbalance challenge in one-stage detector in a bid of uplifting performance of one-stage detector to be in par with two-stage detectors such as Faster-RCNN. They proposed a new loss function called focal-loss to naturally handle the background-foreground class imbalance of dense bounding box perdition in one-stage detector. However, RetinaNet is very slow for real-time tasks though still faster than most two-stage networks and competitively high performing. Another well known one-stage detector called SSD[29] targets multi-scale detection so that a detector could better detect objects of varying scale.

However, in this paper we take inspiration from YOLOv2's grid based approach of object detection and maximize on its speed and performance. However, it is to be recalled that YOLOv2 lacks depth and few other current best practices such as multi-scale detection, and residual connections which the authors included in latest version YOLOv3 [28]. Theoretically, it is possible to use our approach on an already best performing detector such as [28] or turn any deep CNN based stat-of-the-art classifier into DenseYOLO based detector for increased detection performance and speed. However, for a simplistic illustration of our proposed method we focus on simple yet commendably powerful YOLOv2 than the sophisticated YOLOv3.

6.4 DenseYOLO

In DenseYOLO, we simply strip the last layer of YOLOv2 and reshape the output layer so that the out put tensor corresponds to a grid cell of an area 32 by 32 to a smaller grid of 8 by 8 area. This simply means we divide the input image width and height by 8 not 32 as YOLOv2 does. After reshaping we add two more blocks composed of convolutional layer followed by batch normalization layer followed by relu activation layer. And lastly, we add an output convolutional layer of tensor corresponding to each grid cells. Figure 6.2 shows the conversion of YOLOv2 into dense detector.

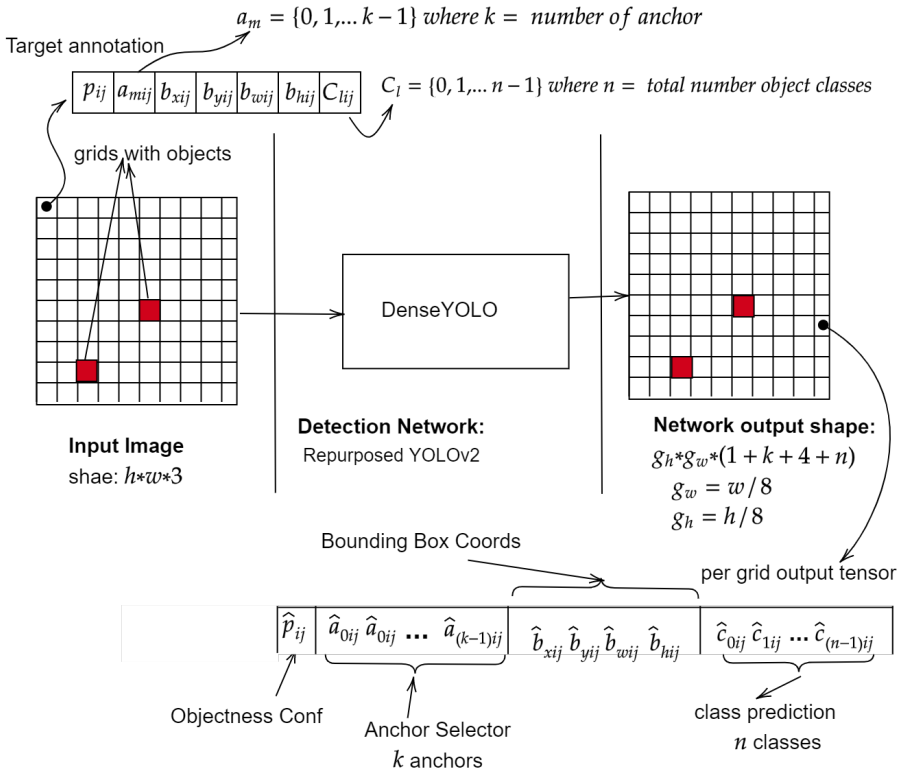


Figure 6.1: *DenseYOLO*

DenseYOLO has an output tensor of shape $g_w * g_h * (k + 1 + 4 + n)$ unlike YOLOv2's $g_w * g_h * (k + 1 + 4 + n)$. The $g_w * g_h$ is the total number of grid cells of area 8 by 8 pixels, k and n stand for the number of anchor boxes and number of object categories, respectively. The 1 stands for binary value 1 or 0 indicating that whether a grid does or does not have an object, respectively. And the 4 stands for the four bounding box coordinate parameters (x_1, y_1) top left corner and (x_2, y_2) bottom right corner. Each grid cells in $g_w * g_h$ size outputs an array of size $(k + 1 + 4 + n)$ instead of outputting $k * (1 + 4 + n)$, see figure 6.1. With the same grid size our models output layer has $k - 1$ folds of reduced weight than YOLOv2's output layer. The significance of this change is that, one can now increase the number of K-means generated anchor boxes to start the training from better representative priors. Or one can increase the input image resolution without worrying that the output layer tensor could explode into large tensor size and restricting implementation due to hardware, specifically memory shortage.

A natural intuitions to garner more accuracy from YOLOv2 like model is to increase number of anchor boxes or increase the number of grids by using smaller grid cells. However, those intuitions might lead to either training loss saturation or exploding output layer. Unlike YOLOv2, however, our model predicts only one object per grid cell instead of the k bounding boxes. Since we use a very small grid size, the probability of two or more objects center falling in the same grid cell is very small.These avoids the need to predict multiple objects per grid cell. We still use anchor boxes, but instead of forcing the network to predict one object per anchor box per grid, we train the system to try all k anchor boxes per grid cell and pick the one that led to high IOU against the ground truth. In short we train the network to predict a best fitting anchor box just the way it predicts bounding box coordinates, class, and confidence score. In the following section we discuss our training loss function and training procedures.

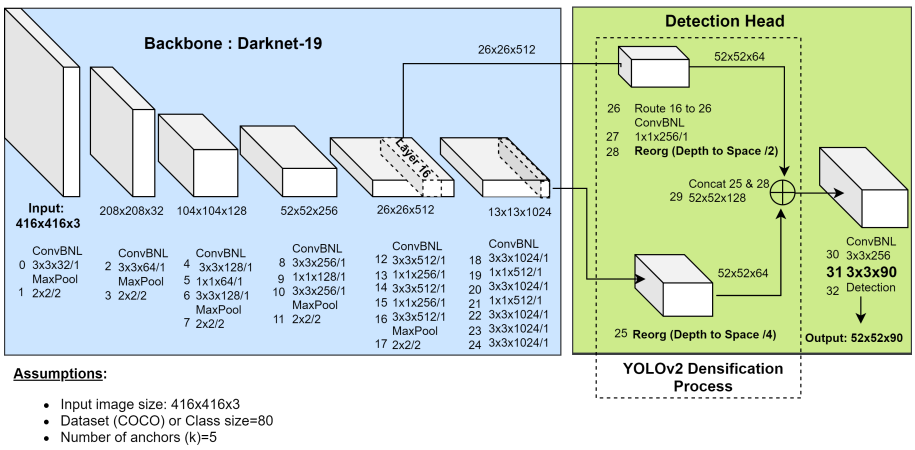


Figure 6.2: ***The repurposing of YOLOv2 into DenseYOLO****. In the figure YOLOv2 with an input image of 416 by 416 is assumed and bs stands for batch size. CONVBL1 and CONVBL2 refers to two blocks each made up of Convolution layer followed by Batch Normalization layer followed by Leaky Relu layer.*

6.5 Training

6.5.1 Ground-Truth Annotation

After re-sizing an input image into fixed input size divisible by 8, say 608 by 608 for example, we then divide each image into grids of g_w by g_h where g_w and g_h are equal to image $width/8$ and $height/8$ respectively. A grid cell with a bounding box center coordinate of an object falling in, is made responsible of detecting an object. We assign a specific anchor out of k anchors,to every bounding box based on their IOU score calculated by centering both the ground truth box and anchor box at origin, that is $(0, 0)$.

Assume (b_x, b_y) is the center coordinate of bounding box and (b_w, b_h) as pair of the width and height of the bounding box. We train the network not to predict these parameters directly, instead we train it to predict (t_x, t_y) and (t_w, t_h) for stability, the same way YOLOv2 and SSD does. The relationships between the two sets of parameters are given by the following equations:

$$t_x = b_x - c_x \tag{6.1}$$

$$t_y = b_y - c_y \tag{6.2}$$

$$t_w = \log(b_w/g_w) \tag{6.3}$$

$$t_h = \log(b_h/g_h) \tag{6.4}$$

In the equations c_x and c_y the grid cells row and column top-left corner.

The corresponding network output $(\hat{t_x}, \hat{t_y}, \hat{t_w}, \hat{t_h})$ are easily converted to $(\hat{b_x}, \hat{b_y}, \hat{b_w}, \hat{b_h})$ by the opposite process given by the following equations:

$$\hat{b}_x = \sigma(\hat{t}_x) + c_x \tag{6.5}$$

$$\hat{b}_y = \sigma(\hat{t}_y) + c_y \tag{6.6}$$

$$\hat{b}_w = \ p_w e^{\hat{t}_w} \tag{6.7}$$

$$\hat{b}_h = \ p_h e^{\hat{t}_h} \tag{6.8}$$

These bounding box parameters are then scaled back by the original image size to yield true predicted image bounding box coordinates.

6.5.2 Training Loss function

Our loss function has four parts. These are coordinate prediction loss, class prediction loss, objectness prediction loss and anchor prediction loss. We shall see each of them separately as follow:

coordinate prediction loss

Here we use the same loss used by YOLOv2 which is the mean square error of the bounding box coordinates given by the following equation:

$$\begin{aligned} crd_{loss} = \lambda_c \sum_{i=0}^{g_w} \sum_{j=0}^{g_h} \mathbb{1}_{ij}^{obj} \left[\left(t_{xij} - \hat{t}_{xij}\right)^2 + \left(t_{yij} - \hat{t}_{yij}\right)^2 \right] + \\ \lambda_c \sum_{i=0}^{g_w} \sum_{j=0}^{g_h} \mathbb{1}_{ij}^{obj} \left[\left(b_{wij} - \hat{b}_{wij}\right)^2 + \left(b_{hij} - \hat{b}_{hij}\right)^2 \right] \end{aligned} \tag{6.9}$$

$\mathbb{1}_{ij}^{obj}$ is 1 if the grid has center of bounding box and zero otherwise. λ_c is a coefficient to weigh coordinate loss and we simply set it to be 0.5.

Objectness loss

Objectness confidence score determines whether a grid has detected an object or not with an IOU above certain threshold. However, imbalance between grids that are made responsible of detecting an object and those are not expected to predict an object, because the center of the bounding box did not fall in them, is very high. And using binary cross entropy is not much of a help, as we expect the objectness confidence to match with an IOU measure of the predicted object against the ground-truth, not just only the binary information that the grid has an object or does not.

Our objective is to drive the overwhelmingly many grid cells that do not have the center of an object to quickly converge to a small confidence score while the few grid cells that have an object rise to a higher confidence as the system learn to predict a better bounding boxes. Thus, our objectness loss function has two parts, one for grids with out object and another for grids with an object. We use log-loss, equation (6.11) for grids without an object whereas IOU based conditional mean square error, equation (6.10), for grids with an object. Both equation (6.11) and equation (6.10) are applied in every grid cell. The term iou in equation (6.10) refers to the best IOU score obtained after computing k IOU against the expected ground truth using each of the k anchors.

$$conf_{obj} = \begin{cases} \lambda_{obj} \mathbb{1}^{obj} (1 - \sigma(\hat{t}_0))^2, & iou > iou_{thrsh} \\ \lambda_{obj} \mathbb{1}^{obj} (\sigma(\hat{t}_0) - iou))^2, & \text{otherwise} \end{cases} \tag{6.10}$$

$$conf_{noobj} = \lambda_{noobj} (1 - \mathbb{1}^{obj}) \log(1.0 - \sigma(\hat{t}_0)) \tag{6.11}$$

λ_{obj} and λ_{noobj} stands for coefficient to penalize grids responsible of detecting an object and those that shouldnt, respectively. Normally, λ_{obj} should be set above λ_{noobj} to emphasize more on the grids that have an object. $\hat{t}_0$ is the system output corresponding to objectenss prediction. As seen from the conditional expression of objectness confidence loss of grids that are meant to detect an object, $conf_{obj}$, whenever the network predicted box with an IOU against ground-truth above certain threshold,iou_{thrsh}, we then push confidence score of that grid to 1 like an encouragement to the network to preserve that bounding box prediction. This is one of the novel change we made to the famous YOLO loss. This change amounts to handling foreground-background imbalance prevalent in one-stage object detection.

Detection Frameworks	Train	mAP
Fast R-CNN [22]	2007+2012	70
YOLO [26]	2007+2012	63.4
SSD300 [29]	2007+2012	74.3
SSD500 citessd	2007+2012	76.8
YOLOv2 416 [27]	2007+2012	76.8
YOLOv2 544 [27]	2007+2012	78.6
DenseYOLO 608@k=9	2007+2012	**80.3**
DenseYOLO 608@k=15	2007+2012	**82.7**

Table 6.1: ***Performance comparison of DenseYOLO trained with*** $k = 9$ ***and*** $k = 15$ ***and tested on Pascal VOC 2007 test dataset against some equivalent models.***

Class prediction Loss

Class prediction loss is simply a weighted categorical cross entropy loss. However, one could use focal-loss instead for better handling of class imbalance among the dataset.

Anchor prediction Loss

Anchor prediction loss, like class prediction loss, is a weighted cross entropy loss. Here we set the target anchor to be the anchor that yielded best IOU whereas the predicted anchor tensor is a softmax output of shape $g_w * g_h * k$ extracted from the output of the network.

Our systems training loss is the mean of the four losses.

6.6 Inference

During inference, in one forward pass we evaluate three confidence measures for every grid cell. These are objectness confidence, classification confidence and anchor prediction confidence. Objectness confidence is simply sigmoid of network output corresponding to the parameter assigned to tell if the cell has an object or not. Classification prediction confidence is argmax of the SoftMax output of the network class prediction outputs. Similarly, anchor prediction confidence is the argmax of the SoftMax output of the anchor prediction tensor. The overall prediction score is the product of these confidence scores. We then filter predicted boxes using threshold and further we implement NMS (non-max suppression) to whittle out redundant prediction of same object. Following the same procedure followed by YOLOv2 we convert the predicted bounding boxes into an object bounding box coordinate on an image.

6.7 Experiment

We experiment on Pascal VOC and COCO dataset by changing the input image size and the number of k anchor boxes on the detection speed and detection performance of DenseYOLO.

In our test on Pascal VOC dataset we merged Pascal VOC 2007 and 2012 training datasets and tested the performance on Pascal VOC-2007 test set. Since we started training from YOLOv2's object detection weight file trained already on Pascal VOC and COCO dataset, we only fine tuned our system for 100 epochs starting from small learning rate of $1e^{-3}$ for the first ten epochs on the added auxiliary layer and unfreezing 22 more layers and training with learning rate $1e^{-4}$ scheduled to reduce by 10 every 50 epochs.

We generated a number of anchor boxes using K-means cluster for both COCO and Pascal VOC dataset for different k's using IOU to cluster bounding boxes. As table 6.2 shows IOU based

# of k	Average IOU on VOC
5	62.14
9	69.09
15	74.47

Table 6.2: *IOU cluster of bounding boxes of combined Pascal VOC 2007 and 2012 training set*

K-mean clustering generates more representative priors (anchors) as k increases suggesting better starting point for training if k is large enough. This might not be true if we keep increasing k as the system would struggle to quickly predict a fitting anchor box for a given shape of objects. Since YOLOv2 has already been trained on $k = 5$, here we train for $k = 9$ and $k = 15$.

In the following section we will discuss our result on performance on VOC and COCO dataset.

6.7.1 Performance on Pascal VOC Object Detection Dataset

After training for 100 epochs on combined 2007 and 2012 Pascal VOC training dataset at 608 by 608 input image resolution, our network achieved a mean Average Precision of over 80.3% for $k = 9$ and 82.7% for $k = 15$ on Pascal VOC 2007 test dataset. This is a significant increase over YOLOv2's 78.6% mAP on Pascal VOC dataset. It is a prove to our natural instinct that letting the system to dynamically pick anchor from better dataset representative anchors, definitely helped to achieve increased detection performance. However, we have also observed that as training progresses for a longer duration, though DenseYOLO's recall keep increasing, the mAP fluctuates up and down almost around the same value. Proper and smart training loss coefficient setting mechanism or more advanced backbone classifier, the kind used in YOLOv3 or RetinaNet, might convert the recall into precision. Tabel 7.3 compares the result of our DenseYOLO's mAP against YOLOv2 and other comparable networks tested on the same dataset.

Models	AP	AP50	AP75	APS	APM	APL
Two-stage methods						
Faster R-CNN+++ [20]	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [53]	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [54]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [55]	36.8	57.7	39.2	16.2	39.8	52.1
One-stage methods						
YOLOv2 [27]	21.6	44.0	19.2	5.0	22.4	35.5
YOLOv3 [28] 608	33.0	57.9	34.4	18.3	35.4	41.9
DSSD513 [56]	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [30]	40.8	61.1	44.1	24.1	44.2	51.2
DenseYOLO 608(@k=9)	27.6	47.1	29.7	11.04	30.9	38.4
DenseYOLO 608(@k=15)	29.03	50.4	32.6	13.11	33.2	40.3

Table 6.3: *COCO dataset test result.*

6.7.2 Performance on COCO Object Detection Dataset

COCO dataset has more generic and challenging 80 category datasets compared to Pascal VOC's 20 category. Moreover, COCO performance evaluation metrics is more stricter in that it requires mean average precision (mAP) to be calculated over 10 scales of IOU 0.5-0.95. Like on VOC

dataset, we also experimented on COCO datasets for $k = 9$ and $k = 15$ with an input image of size 608 by 608. Table 7.3 compares DenseYOLO against some well known detectors. mAP in both $k = 9$ and $k = 15$ significantly outperforms YOLOv2, though still behind some of the recent networks such as YOLOv3 and RetinaNet. This has certain obvious reasons such as the fact that YOLOv2 has shallower depth compared to the rest of the network leading to poor classification or feature extraction capability.

6.8 Conclusion

In this paper we introduced a new light weight, faster and more accurate implementation of YOLO like one-stage object detector. Our design approach is lighter, because for the same input image size by just reshaping the output layer into dense grid and avoiding the need to predict multiple objects per grid cell while also fully using the concept of anchor box or prior boxes, we obtain a lighter and hence faster network. We have introduced a novel concept of training an object detector to predict a prior box along bounding box coordinate and class prediction. Apart from design change, we have also introduced new loss calculation for objectness confidence prediction to balance the imbalance between grids with no object of interest and grids dedicated to predict an object.

Chapter

7 MultiGrid Redundant Bounding Box Annotation for Accurate Object Detection

Chapter content

MultiGrid Redundant Bounding Box Annotation for Accurate Object Detection

7.1 Objective

Modern leading object detectors are either two-stage or one-stage networks repurposed from a deep CNN-based backbone classifier network. YOLOv3 is one such very-well known state-of-the-art one-shot detector that takes in an input image and divides it into an equal-sized grid matrix. The grid cell having the center of an object is the one responsible for detecting the particular object. This paper presents a new mathematical approach that assigns multiple grids per object for accurately tight-fit bounding box prediction. We also propose an effective offline copy-paste data augmentation for object detection. Our proposed method significantly outperforms some current state-of-the-art object detectors with a prospect for further better performance.

7.2 Introduction

An object detection network aims to locate an object on an image using a tight-fit rectangular bounding box and label it correctly. Nowadays, there are two distinct approaches to achieve

this purpose. The first and performance-wise, the most dominant approach is two-stage object detection, best represented RCNN [17] and its derivatives [22, 60]. In contrast, the second set of object detection implementations, well acknowledged for their outstanding detection speed and light-weightiness, are referred to as one-staged networks, representative examples being [26], [29],[30]. Two-stage networks rely on an underlying region proposal network that generates candidate regions of an image likely to contain an object of interest, and a second detection head handles the classification and bounding box regression. In one-stage object detection, detection is a single, fully unified regression problem that simultaneously handles classification and localization in one complete forward pass. Due to this, usually, one-stage networks are lighter, faster, and simple to implement.

One-stage networks can be classified as anchor-based [26–30, 61–63] and anchorless [64, 65]. Anchor-based networks such as YOLOv3[28] or YOLOv4[61] mainly divide an input image into equal grid cells. Furthermore, each grid cell regresses object bounding box coordinates and classifies them into one of the predefined class categories while simultaneously scoring the predicted box's objectness confidence. Other anchor-based variants, such as SSD [29] and RetinaNet [30] employ the concept of feature pyramid to perform multi-scale detection extracted from different layers of the backbone classifier network such as VGG [58] or families of ResNet [57]. Recent anchorless entries of one-shot detectors use key-point such as corners of bounding box [65] or center coordinate or combination of both [64], replacing the use of anchor boxes by key-point pooling convolutional pipelines along their bounding box regression and classification networks.

This paper sticks to YOLO's approach, particularly YOLOv3 [28], and proposes a simple hack that simultaneously enables multiple grid cells to predict an object coordinate, class, and objectness confidence. The basic theory behind multi-grid cell assignment per object is to increase the likelihood of predicting a tight-fit bounding box by enforcing more than one cell working on the same object. Some of the advantages of multi-grid assignment includes: (a) gives the object detector a multi-perspective view of the object it is detecting rather than relying on just one grid cell to predict the class and the coordinates of an object, (b) less random and erratic bounding box prediction, meaning high precision and recall, since nearby grid cells are trained to predict same object class and coordinates,(c) reducing the imbalance between grid cells with an object of interest against grids without an object of interest. Moreover, since the multi-grid assignment is mathematical utilization of an existing parameters and does not require an extra keypoints pooling layer and post-processing to regroup keypoints to their corresponding objects like CenterNet[64] and CornerNet[65], we say it is a more natural way of achieving what anchorless or keypoint-based object detectors are trying to achieve. In addition to the multi-grid redundant annotation, we also introduce a new offline copy-paste-based data augmentation technique for accurate object detection.

7.3 Related Works

The pioneer and most successful one stage-detector YOLO [26] and its successive incremental improvements [27], [28], [61] divide an input image into grid cells of equal size. The grid that contains the center of a given object-bounding box on an image is responsible for detecting that particular object. Since YOLOv1, the authors of YOLO tried to improve the performance of their object detector by incrementally incorporating key improvements such as more network depth, more anchor boxes, slight change on loss function, and lately since YOLOv3 best practices such as multi-scale detection and skip-connections are incorporated.

The other typical one-stage detector is SSD[29]. SSD uses multi-layer feature pyramids on top of a backbone classification network, notably a VGG network, to perform a multi-scale detector that better handles objects of various scales. Using a similar concept of feature pyramids as in SSD, another famous object detector called RetinaNet [30] proposed a novel loss function called focal-loss to solve the foreground and background class imbalance prevalent in one-stage

detectors unlike two-stage networks.

Recently, anchorless one-stage object detection techniques such as [65], [64], [66], [67] aim to reduce the hurdle of determining the appropriate number and shape of anchor boxes. Networks such as DSSD [56] and RetinaNet [30] use default-boxes, also referred to as anchor boxes, amounting over tens or hundreds of thousands, resulting in slow training and brutal non-max suppression during inference. Instead, anchorless detectors add a separate layer to pool and process points on the bounding box of an image. CornerNet[65], for example, adds a pipeline that processes the corner keypoints of an object hence needing no anchor boxes. CenterNet[64], another anchorless one-stage detector, adds a third point, namely the center point of an object bounding box in addition to the corner keypoints. The center keypoint in CenterNet is to aid CornerNet to have a more global view of an object it is detecting, which was its bottleneck at first.

This paper sticks to YOLO's grid-based approach since YOLO's approach neither requires many anchor boxes like SSD, DSSD or RetinaNet nor adds a separate pipeline to process keypoints like CornerNet or CenterNet. However, unlike YOLO, we propose a mathematical way to assign an object to multiple grid cells, including the grid cell where the center of the object-bound box falls. As we stated earlier, in YOLO, the grid that contains an object's bounding box center is made responsible for detecting that particular object, hence, one grid assignment per object. In our implementation, we will show that mathematically it is possible to assign any number of grid cells to annotate an object, though we will only use the grids around the center, including the center grid. Due to the multi-grid annotation, we dubbed our object detector MultiGridDet short for Multi-Grid Detector. Our detector is light and faster than YOLOv3 mainly due to two reasons; (1) MultiGridDet has relatively less depth number of layers and (2) we use a lighter output layer, or detection head, based on the technique introduced by DenseYOLO [62].

7.4 Multi-Grid Assignment

Consider Figure 7.1 containing three objects, namely a dog, bicycle, and car. For brevity, we will explain our muti-grid assignment on one object, the dog. Figure 7.1(a) shows the three objects bounding box with more detail on the dog's bounding box. Figure 7.1(b) shows the zoomed-out region of Figure 7.1(a), focusing on the dog's bounding box center. The top-left coordinate of the grid cell containing the center of the dog bounding box is labeled by number 0, while the other eight grid cells around the grid containing the center have a label from 1 up to 8.

In YOLO and other YOLO-based detection networks, the grid labeled 0 is solely responsible for predicting the class dog and its precise bounding box coordinates (x, y, b_w, b_h), whereas in our case, we assign all grids labeled 0 to 8 to predict the class and the precise bounding box of the dog simultaneously.

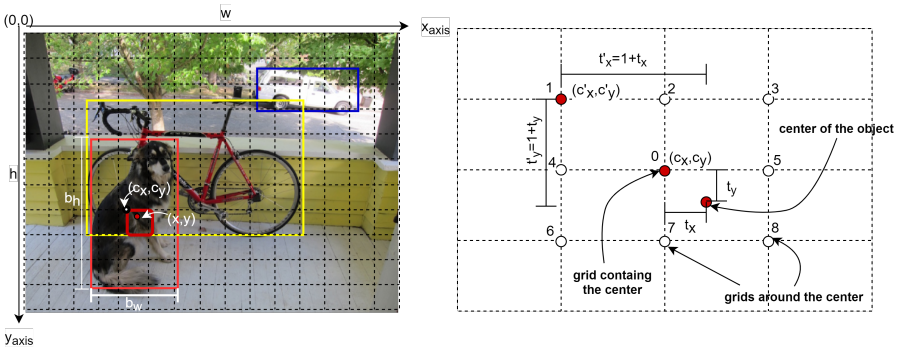


Figure 7.1: ***Multi-grid assignment***

The grid containing the center coordinate, the small red box on Figure 7.1(a), or coordinate

labeled 0 on Figure 7.1(b) is at the grid location $(c_x,\ c_y)$. We calculate the $(c_x,\ c_y)$ coordinates using the equation, $c_x = \left\lceil \frac{x}{g_w} \right\rceil$ and $c_y = \left\lceil \frac{y}{g_h} \right\rceil$, where g_w and g_h are the grid cell's width and height, respectively. In YOLOv1 and YOLOv2 g_w and g_h are both 32 pixels each, whereas in YOLOv3, due to the multi-scale detection feature tailored for small, medium and large scale object, the grid cells are also in those three scales; 8×8, 16×16, and 32×32 pixels.

Starting from YOLOv2, YOLO-based object detectors predict an offset of the bounding box from pre-generated anchor boxes instead of directly predicting the bounding box's coordinates. As a result, the ground-truth bounding box values, (x, y) and (b_w, b_h) are rescaled to smaller scales (t_x, t_y) and (t_w, t_h), respectively, for training stability using Equations (7.1) to (7.4). Note that, t_x and t_y are in the range $[0, 1]$.

$$t_x = \frac{x}{g_w} - \left\lceil \frac{x}{g_w} \right\rceil \tag{7.1}$$

$$t_y = \frac{y}{g_h} - \left\lceil \frac{y}{g_h} \right\rceil \tag{7.2}$$

$$t_w = \log\left(\frac{b_w}{a_w}\right) \tag{7.3}$$

$$t_h = \log\left(\frac{b_h}{a_h}\right) \tag{7.4}$$

, where a_w, a_h are the best-fit anchor box's width and height, respectively, generated using K-means IoU clustering.

One can easily convert the (t_x, t_y) and (t_w, t_h) parameters to the original bounding box parameters of an object, that is (x, y) and (b_w, b_h) using the reverse Equations (7.5) to (7.8):

$$x = (c_x + t_x) \times g_w \tag{7.5}$$

$$y = (c_y + t_y) \times g_h \tag{7.6}$$

$$b_w = a_w \times e^{t_w} \tag{7.7}$$

$$b_h = a_h \times e^{t_h} \tag{7.8}$$

Thus far, we have explained how the grid containing the center of an object's bounding box annotates an object's ground truth. This dependence on just one grid cell per object to do the difficult job of predicting the class and the exact tight-fit bounding box raises many questions such as (a) massive imbalance between the positive and negative grids, that is, grids with and without object's center coordinate (b) slow bounding box convergence to ground-truth, (c) lack of multi-perspective (angle) view of the object to be predicted. So one natural question to ask here is, *"obviously, most objects encompass an area of more than one grid cell, and thus would there be a simple mathematical way to assign more of those grid cells try to predict the class and coordinates of the object together with the center grid cell ?"*. Some of the advantages of doing this are (a) reduce imbalance, (b) faster training to converge to the bounding box as now multiple grid cells target the same object at once, (c) increase the chance of predicting tight fit bounding box (d) give grid-based detectors such as YOLOv3 a multi-perspective view rather than a single point view of the objects. Our multi-grid assignment tries to answers the above question, and we explain it as follow: consider (c'_x, c'_y) be any grid cell within the distance of $d \in \{-1, 0, 1\}$ from $(c_x,\ c_y)$, or mathematically $(c'_x, c'_y) = (c_x + d_x, c_y + d_y)$ where d_x and d_y are distance d in x and y directions from the (c_x, c_y) point respectively. Based on the value of d, this equation refers to all the grids marked 0 to 8 in Figure 7.1(b). Then instead of (c_x, c_y) based equations of (5-9), we can rewrite a general one using (c'_x, c'_y) that applies for any of the grids labeled 0 to 8 as in Equations (7.9) to (7.12):

$$x = (c'_x + t'_x) \times g_w \tag{7.9}$$
$$y = (c'_y + t'_y) \times g_h \tag{7.10}$$
$$b_w = a_w \times e^{t_w} \tag{7.11}$$
$$b_h = a_h \times e^{t_h} \tag{7.12}$$

Where $t'_x = \mp d + t_x$ and $t'_y = \mp d + t_y$. Note that now the bounding box parameter $(t'_x,\ t'_y)$, will have a range of $[-1, 2]$, unlike the $[0, 1]$ range of (t_x, t_y). Figure 7.2 shows the ground-truth annotation of the expected output of our object detector.

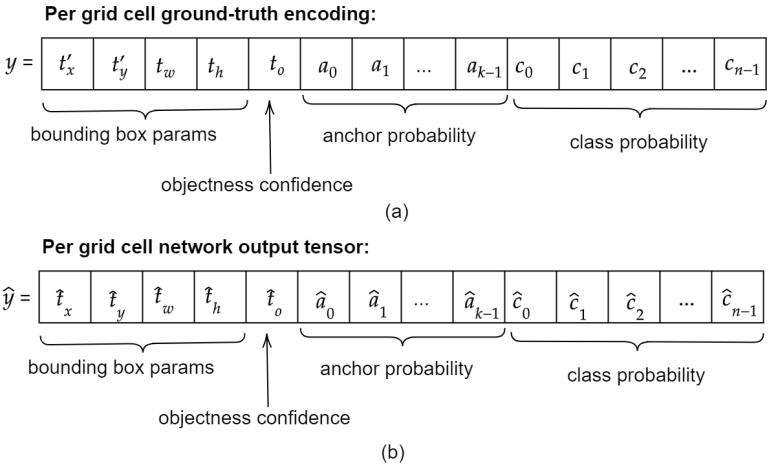


Figure 7.2: ***Ground-truth encoding***

7.5 Training

7.5.1 The Detection Network: MultiGridDet

MultiGridDet is an object detection network we repurposed by removing six darknet convolutional blocks from YOLOv3 to make it lighter and faster. A convolutional block has one Conv2D layer followed by a Batch Normalization layer followed by a LeakyRelu layer. The removed blocks are not from the classification backbone, that is, Darknet53. Instead, we removed them from the three multi-scale detection output networks or heads, two from each output network. Though usually deep networks perform well, too deep networks also tend to overfit quickly or drastically reduce the network's speed.

In addition to stripping the six convolutional blocks, we also adopt DenseYOLO's output head. In YOLOv3, each output layer has a tensor shape $g_w \times g_h \times k \times (5 + n)$, where $g_w \times g_h$ is the total grid cells, k is the number of anchors, and n is the total number of object classes. In DenseYOLO, the output layer tensor has a shape $g_w \times g_h \times (5 + k + n)$ which means approximately k times fewer parameters on the output layer. Moreover, DenseYOLO introduces a novel approach by making an *anchor box a predictable parameter* similar to an object's class and bounding box prediction. Thus in MultiGridDet, we opt for DenseYOLO's lighter approach on the output layer. In general, MultiGridDet has less convolutional block and a **lighter** output head compared to YOLOv3, thus relatively **faster**.

7.5.2 The Loss function

Like DenseYOLO, our loss function has four parts: class prediction loss, location or coordinate prediction loss, anchor prediction loss, and objectness confidence loss.

Class prediction loss (error)

Our class prediction loss is a simple binary cross-entropy loss calculated over all grid cells labeled to have contained an object of interest.

Anchor prediction loss (error)

:- Our anchor prediction loss is also a binary cross-entropy loss in which we train our network to pick an appropriate anchor out of a given set of anchors. We generated nine anchor boxes, 3 for each scale, using IoU (Intersection over Union) based K-means using the same approach as YOLOv3. For example, an anchor with the highest IoU against the ground truth bounding box is assigned to an object during ground truth annotation. And during training, we train the network to learn to pick the anchor that gives the highest IoU to a given object, a concept introduced by DenseYOLO.

Coordinate prediction loss (error)

As shown in Figure 7.2, every object bounding box has four parameters $(t'_x,\ t'_y, t_w, t_h)$ related to the actual bounding box coordinates using Equations (7.9) to (7.12). Accordingly, the network will predict the corresponding bounding box parameters $(\hat{t}_x, \hat{t}_y, \hat{t}_w, \hat{t}_h)$. As explained in an earlier section, the $(t'_x,\ t'_y)$ corresponds to an object's center coordinate and has a value in the range $[-1, 2]$. Thus the corresponding network output must pass through an activation function whose output value must also be in the same range. However, the common activation functions such as tanh have a range $[-1, 1]$, sigmoid $[0, 1]$, and Relu or LeakyRelu have a range either $[0, \infty]$ or $[-\infty, +\infty]$, respectively. We experimented with various custom activations, or simple mapping functions, but finally figured out using tanh and sigmoid activation functions in combination works very well since the output of the sum of the two functions is bounded in the range $[-1, 2]$.

Let the direct output of the detection network corresponding to $(t'_x,\ t'_y)$ before passing through an activation be $(\hat{z}_x, \hat{z}_y)$. Using Equations (7.13) to (7.14), we convert $(\hat{z}_x, \hat{z}_y)$ into $(\hat{t}_x, \hat{t}_y)$.

$$\hat{t}_x = \tanh(\beta \times \hat{z}_x)\ + \sigma(\beta \times \hat{z}_x) \tag{7.13}$$

$$\hat{t}_y = \tanh(\beta \times \hat{z}_y)\ + \sigma(\beta \times \hat{z}_y) \tag{7.14}$$

As shown in Figure 7.3, equation 7.13 and 7.14 smoothly transforms the network output $(\hat{z}_x, \hat{z}_y)$ to the desired output range. The β in equations is to horizontally expand the *tanh* and sigmoid function to prevent quick saturation of these functions. β should be picked from range $[0, 1]$ since values above 1 make bounding box prediction unstable during training. In our case we set $\beta = 0.25$ and during inference also we use the same value for β.

Finally, we calculate coordinate prediction loss using mean square error as given in eq. (7.15).

$$\begin{aligned} lcrd_{ij} = \lambda_{coord} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{x_{ij}} - \sqrt{\hat{x}_{ij}} \right)^2 + \left(\sqrt{y_{ij}} - \sqrt{\hat{y}_{ij}} \right)^2 \right] + \\ \lambda_{coord} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{b_{wij}} - \hat{b}_{wij} \right)^2 + \left(\sqrt{b_{hij}} - \sqrt{\hat{b}_{hij}} \right)^2 \right] \end{aligned} \tag{7.15}$$

$$\lambda_{coord} = -\lambda \log(IoU_{score_{ij}}) \tag{7.16}$$

$$loss_{coord} = \frac{1}{m} \sum_{i=0}^{g_w} \sum_{j=0}^{g_h} lcrd_{ij} \tag{7.17}$$

g_w and g_h are the total number of grid cells horizontally and vertically, respectively, whereas ${1^{obj}}_{ij} = 1$ if the grid cell has an object and otherwise equals zero. m refers to the batch size. IoU_{score_ij} has a value of 0 to 1 depending on how much the predicted bounding box overlaps with the ground truth. The logarithmic coefficient we introduced in the coordinate loss plays a

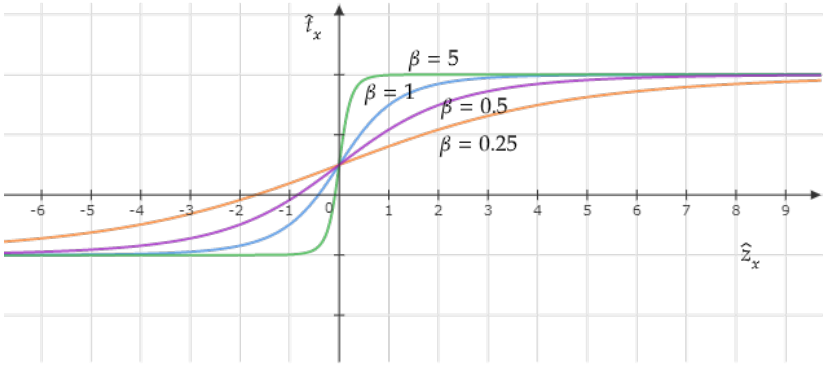


Figure 7.3: ***Coordinate activation function plot with different β values***

significant role. It penalizes incorrect bounding box prediction and rewards the more accurate ones logarithmically, similar to what focal-loss of RetinaNet intends to achieve, except ours is for localization rather than classification.

Objectness confidence loss

Fourth part of our loss function evaluates objectness confidence of the predicted bounding box. Our objectness confidence loss has two parts, as seen in equation 6, one for grids labeled to have contained an object, that is $t_0 = 1$, and those that are not, meaning $t_0 = 0$.

$$\begin{aligned} loss_{conf} = & \frac{1}{m} \sum_{i=0}^{g_w} \sum_{j=0}^{g_h} \mathbb{1}_{ij}^{obj} \times BCE_{obj_{ij}} + \\ & \frac{1}{m} \sum_{i=0}^{g_w} \sum_{j=0}^{g_h} \mathbb{1}_{ij}^{noobj} \times BCE_{noobj} \end{aligned} \tag{7.18}$$

$$BCE_{obj_{ij}} = -\log\left(\hat{t}_0\right) \tag{7.19}$$

$$BCE_{noobj_{ij}} = -\log\left(1 - \hat{y}_{[\cdots,4:]}\right) \tag{7.20}$$

The objectness confidence is similar to YOLOv3 because we both use binary cross-entropy loss, except in our case, the no object loss part tries to make classification, anchor prediction, and objectness confidence to have probability zero.

7.5.3 Offline Synthetic Data Generation

The other significant contribution of our work is our offline copy-paste-based data augmentation. As much as careful design of artificial intelligence model is essential, a neat and tremendous amount of training and validation data are also mandatory for a better performing network, especially for an object detection network. Recently copy-paste-based augmentation techniques such as simple alpha blending of two or more images MixUp, CutMix and Mosaic augmentations are reported to increase object detectors' performances. In this work, we implement our own unique and more robust offline copy-paste data augmentation to increase training data significantly.

In general, our offline copy-paste artificial training image synthesis works as follows: First, we download thousands of background objectless images, meaning images without our object of interest, from google images using a simple image search script using keywords such as landmarks,

rain, forest, amusement parks, deserts, cities, wallpapers. We then iteratively pick p number of objects and their bounding boxes from random q images of the entire training dataset. We then generate all possible combinations of the p bounding boxes selected using their index as ID. From the set of the combinations, we pick a subset of bounding boxes that satisfies the following two conditions:

- if arranged in some random order side by side, they must fit within a given target background image area
- and should efficiently utilize the background image space in its entirety or at least most part of it without the objects overlap.

Following the above approach, we generate hundreds of thousands of artificial images. Moreover, before copy-pasting an object from one image onto the background, we randomly do various common augmentations on the individual object. During training, we randomly implement simple and common augmentations to the training minibatch. Fig. 7.4 shows three sample artificially synthesized images using our offline copy paste augmentation. The figure shows that the artificial images comprise objects that often will not appear together, reassuring the training dataset's robustness. To prevent the network from learning the copy-paste edges, we add an offset of 10 to 15 pixels in all four sides of the object when copying from the source image, thus assuring the bounding box will not rest on the paste borders. As explained earlier, we also passed each object through one or more common augmentation (flip, brightness, contrast, etc.) before pasting on the background image to prevent early overfitting of the network on the training dataset. For detail of our offline supplementary artificial image generation please refer Appendix 9.2.

Figure 7.4: ***Sample Offline Copy-Paste Generated Artificial Images***

7.6 Experiment

To test our multi-grid redundant object annotation and our offline copy-paste data augmentation, we repurposed the YOLOv3 network into a lighter and faster network, dubbed MultiGridDet, as explained in the earlier section. We perform training on two well-known object detection datasets, namely Pasca-VOC (VOC 2007 + 2012) and COCO datasets.

Detection Frameworks	mAP
Fast R-CNN [22]	70
YOLO [26]	63.4
SSD300 [29]	74.3
SSD500 citessd	76.8
YOLOv2 416 [27]	76.8
YOLOv2 544 [27]	78.6
MultiGridDet 416x416 (ours)	**83.5**

Table 7.1: ***Performance Comparison on Pascal VOC 2007 test set***

We supplement both datasets with artificial images we generated using our offline copy-paste augmentation. We downloaded about 10,000 background images from Google Images using simple search keywords and a script. Then using these background images and total Pascal VOC 2007 and 2012 training and validation images, which constitute about 16K images, we generate an additional 200K artificial images for pascal VOC object detection. In total, we increased the initial 16K Pascal VOC 2007 + 2012 training + validation set images to 216K images and used a validation split of 0.2 so that 80% of the total data are used for training while the remaining 20% are for validation.

Similarly, we used the same 10K background images and the original 118K COCO images to generate another 200K artificial COCO images. This increases our COCO dataset to 318K images. Similar to the strategy we used on the Pascal VOC dataset, we used a validation split of 0.2 to train object detection on the COCO dataset as well. It is good to note that, though we artificially generated hundreds of thousands of new images, our artificially generated images objects are from the same dataset, picked randomly, individually augmented, and pasted on a randomly picked background image.

We used the Darknet53 weight file from YOLOv3 authors to train the detector. For the first 50 epochs, we trained only the detection head by freezing the Darknet53 weight file and 150 more epochs by unfreezing the whole network. We start the training with learning rate $1e^{-4}$, and after the 75^{th} epochs, we started using cosine decay to update the learning rate. Throughout the training, we used Adam optimizer. We had access only to 2 Tesla V100 32 GB Nvidia GPU, which limited our training batch size to 64 (32 per GPU) and made the training take longer, restricting us from testing our network performance with other backbones such as ResNet. Next, we will discuss our experiment's result on the two datasets and compare them with other well-known object detection networks.

Pascal VOC 2007 test set:- Pascal VOC 2007 test set has about 5k test images in 20 class categories. It is one of the widely used generic object detection datasets for comparing the performance of general-purpose object detectors. Accordingly, we tested our MultiGridDet performance using Pascal VOC mean average precision (mAP) metrics at IOU (intersection over union) 0.5. Table 7.1 shows the mAP performance comparison of MultiGridDet against other state-of-the-art one-stage detectors and equivalent two-stage detectors. As seen from the table, our detector significantly outperforms all older versions of YOLO, YOLOv1, and YOLOv2, including all variants of SSD, RetinaNet, and Fast RCNN. Table 7.2 further shows our detector's per class mAP score in detail. The authors of YOLOv3 never reported the performance of the original YOLOv3 on the Pascal VOC dataset. However, following their training and data augmentation approaches detailed in paper [28], we retrained YOLOv3 on the combined Pascal VOC 2007 and 2012 training set and achieved a maximum mAP of 77.63% at input image size 608. This score is much lower than our MultiGridDet score of 83.5% mAP at an input image resolution 416.

Model	data	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
Faster [22]	07+12	73.2	76.5	79.0	70.9	65.5	52.1	83.1	84.7	86.4	52.0	81.9	65.7	84.8	84.6	77.5	76.7	38.8	73.6	73.9	83.0	72.6
SSD300 [29]	07+12	74.3	75.5	80.2	72.3	66.3	47.6	83.0	84.2	86.1	54.7	78.3	73.9	84.5	85.3	82.6	76.2	48.6	73.6	73.9	83.4	74.0
SSD512 [29]	07+12	76.8	82.4	84.7	78.4	73.8	53.2	86.2	87.5	86.0	57.8	83.1	70.2	84.9	85.2	83.9	79.7	50.3	77.9	73.9	82.5	75.3
MultiGridDet	07+12+CP	**83.5**	87	93	84	70	73	96	88	93	65	83	80	90	92	92	87	55	85	78	93	84

Table 7.2: *Individual Pascal VOC 2007 dataset classes mAP.*

MS COCO test set:- COCO dataset is the most challenging dataset for object detection, typically due to its massively unfair under and over-representation of object class categories and object scale imbalance in the dataset. Nonetheless, it is a more generic dataset consisting of 80 class categories and more robust performance measurement metrics; an mAP averaged over 11 IoU ranges $[0.5 - 1.0]$ referred to as AP(average precision). Accordingly, we trained our MultiGridDet on the COCO dataset and obtained an AP of 31.8%, a little less than YOLOv3's 33% AP at 608×608 input image size as shown in Table 7.3. However, on large images, that is objects with a bounding box area above 96^2 according to COCO metrics, MultiGridDet by far outperforms YOLOv3's AP 41.9% by +15.5%, scoring AP 57.4%. MultiGridDet is poor on small and medium image detections; only 11%AP against YOLOv3's 18.3% AP on small objects and 24.6% AP on medium images against YOLOv3 35.5% AP. Objects such as bottles usually appear in smaller sizes and crowed, whereas objects such as boats and potted plants usually appear in widely irregular shapes and sizes. These are objects MultiGridDet struggled to detect correctly.

Finally, to illustrate the quality of MultiGridDet bounding box prediction, we visualize the prediction of six randomly picked images from the Pascal VOC 2007 test dataset. Fig. 7.5 shows these visualization. As seen from the figure, almost in all cases, the predicted unfiltered bounding boxes overlap perfectly, proving tight-fit bounding box prediction.

In summary, since small objects have tiny widths or height, some even smaller than the 8 grid size, thus they are effectively annotated with a single grid cell, whereas larger objects will have a maximum of 9 grid cells to annotate and predict them. Probably, the addition of more fine-grained output layers, typically an output layer with 2 and/or 4 grid cell sizes in addition to the 8, 16, and 32 grid cells, might help to increase the performance of MultiGridDet on small object detections.

We have also compared MultiGridDet inference speed with YOLOv3 on a standard personal laptop with Nvidia GPU Geforce 1060 and 16 GB RAM Intel Core i7-7700HQ CPU 2.80GHz processor. On average, YOLOv3 at 416×416 input image for 80 class COCO dataset takes 0.149 seconds to infer an image, the overall time spent from reading input image, preprocess, predict, and draw bounding boxes back on the image and display it or save it in a directory. For MultiGridDet at the same input resolution and same dataset, it takes only 0.103 seconds. On video object detection at 416×416 80 Class COCO dataset YOLOv3 reaches detection speed of 6FPS (frame per second) whereas MultiGridDet reaches upto 9FPS. Note that the computer we used is not the same as the one the author used, and it has a much slower GPU. In general, in the speed test, MultiGridDet is significantly faster than YOLOv3.

Models	**AP**	**AP50**	**AP75**	**APS**	**APM**	**APL**
Two-stage methods						
Faster R-CNN+++ [20]	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [53]	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [54]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [55]	36.8	57.7	39.2	16.2	39.8	52.1
One-stage methods						
YOLOv2 [27]	21.6	44.0	19.2	5.0	22.4	35.5
YOLOv3 [28] 608	33.0	57.9	34.4	18.3	35.4	41.9
DSSD513 [56]	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [30]	40.8	61.1	44.1	24.1	44.2	51.2
DDGNet [63]	28.8	51.2	30.8	12.1	39.4	50.7
DenseYOLO[62]	29.03	50.4	32.6	13.11	33.2	40.3
MultiGridDet @608x608	31.8	52.1	40.7	11.0	24.6	57.4

Table 7.3: *AP Performance on COCO test set*

Figure 7.5: ***Sample MultiGridDet output on randomly selected Pascal VOC 2007 test set images.*** *As seen from the figure the first row shows six input images, whereas the second row shows the prediction of the network before non-max-suppression (NMS) and the last row shows the final bounding box prediction of MultiGridDet on the input image after NMS thresholding.*

7.7 Conclusion

In this paper, we proposed a new and alternative object detection implementation for one-stage YOLO-like object detectors that rely on a matrix of grid cells. It is a lightweight, faster, and commendably accurate detector with a prospect for further improvement that addresses the poor performance on the infinitesimally tiny objects of the COCO dataset. A straightforward technique is probably to add finer output scales, for example, 2×2 or 4×4, so that the multi-grid annotation could also be implemented on those tiny objects. Another significant contribution we achieved in this work is our unique data augmentation technique that vastly increases object detection training sets without needing additional external dataset. Finally, as future work, we would like to tackle small object detection challenges and try to use MultiGridDet on object tracking and segmentation challenges.

Chapter

8 Resource and Power Efficient High-Performance Object Detection Inference Acceleration Using FPGA

Chapter content

Resource and Power Efficient High-Performance Object Detection Inference Acceleration Using FPGA

8.1 Objective

The success of deep convolutional neural networks in solving age-old computer vision challenges, particularly object detection, came with high requirements in terms of computation capability, energy consumption, and a lack of real-time processing capability. However, FPGA-based inference accelerations have recently been receiving more attention from academia and industry due to their high energy efficiency and flexible programmability. This paper presents resource-efficient yet high-performance object detection inference acceleration with detailed implementation and design choices. We tested our object detection acceleration by implementing YOLOv2 on two FPGA boards and achieved up to 184 GOPS with limited resource utilization.

8.2 Introduction

Object detection is one of the most critical areas of computer vision due to its vast applications in surveillance and security, medical imaging, media and entertainment, and transport automation, to name a few. Though it has been an old and challenging quest for researchers and academia to perfect object detection performance, it is only in recent years that significant progress has been made due to the success of convolutional neural networks in image classification [59]. The current trend in object detection relies on the use of very deep image classification convolutional neural network(s) (CNNs) repurposed to perform detection tasks [22, 26, 29, 30]. However, the challenge with deep CNN-based detectors is the intensive computation these require in the order of multiple GOPs, which can only be rendered by utilizing high-performance computers and GPUs that consume high energy and resources. On the other hand, most applications require real-time inference capability with a constrained power source for real-time decision-making. Thus, low energy and resource-constrained small electronics such as embedded systems have benefited little from the leap in the accuracy of object detectors as the achievement also required more advanced machines or clusters of machines [68].

Nonetheless, recently field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs) are gaining increased attention as energy-efficient and real-time time alternatives [68–71]. Although FPGAs and ASICs hardly reach the same or increased throughput as GPUs, they consume less energy. On the other hand, compared to FPGAs, the high cost and long development period of ASICs also make them unfavorable as it is challenging to keep them up with the rapid changes of deep CNNs. As a result, FPGA-based deep CNN inference accelerations are becoming a center of focus for lightweight and real-time deep CNNs for embedded systems.

Despite FPGA-based machine learning implementations generally gaining traction, the progress is slow and marked by disjoined and irregular individual efforts, unlike the software world where there is a broad community base and frameworks. Recent hardware acceleration implementations exhaustively but inefficiently consume onboard resources, such as DSPs, BRAMs, and logic cells, sometimes beyond what is recommended by development boards. Such implementations lead to high power consumption and are costly in terms of energy. On the other hand, extreme data quantization, typically one to three-bit quantization, has been tried to accelerate CNN on FPGA. However, although such quantization quickly achieves more than real-time speed, their accuracy loss is significant. This paper, however, presents a detailed end-to-end hardware acceleration implementation while maintaining high performance and speed and—at the same time—highly efficient resource utilization. Although we demonstrate our accelerator design based on the well-known YOLOv2 detector, our object detection implementation is easily customizable to different YOLO-like one-stage accelerators. The analysis of our custom inference accelerator on DenseYOLO and DDGNet by comparing against this YOLOv2 acceleration implementation is explained in Appendix 9.2.

8.3 Related Works

Increasing accuracy performance has been at the center of computer vision challenges for a long time. In this quest for increased accuracy, object detection networks, or CNN-based networks in general, have become very deep, complex, heavy, resource-wise expensive, and energy inefficient. Top state-of-the-art object detection networks are based on deep CNN networks and have tens or hundreds of layers and over 50 million parameters [28, 30]. Moreover, at the core of these heavy models is a convolution operation taking the most resource and computation time, reportedly over 90% [72] models' execution time. On the other hand, many real-world problems of computer vision demand real-time and lightweight detectors that fit on an embedded system. As a result, FPGA's support for high parallelism and CNN's suitability for such high parallelism elevates the prospect of FPGA becoming the leading hardware solution for accelerating computer vision applications. Unfortunately, most top-performing object detectors are too big to fit into most FPGA's on-chip memory, making it difficult or impossible to fully exploit the parallelism support in FPGA and the convolution process.

Over the years, many authors have proposed and tried different alternatives for accelerating CNN-based networks, particularly the convolution layer. An extensive review of hardware acceleration methods from multiple points of view can be read from the review works of [73, 74]. Some optimization methods include replacing the standard convolution algorithm altogether with faster algorithms such as fast Fourier transform (FFT) [40, 41] or Winograd [42, 43]. Other methods based on the transformation of convolution computation include performing convolution as matrix multiplication [44].

However, most optimization methods nowadays focus on bettering the standard convolution by exploiting its parallelism capability via common loop optimization techniques such as loop unrolling, pipelining, and interchanges [47]. In addition to loop optimization concepts such as maximizing data reuse, employing double-buffering to minimize memory access bottleneck or streamlined dataflows are integral parts of modern hardware acceleration designs [75]. Algorithms such as roofline modeling [47] have been used to pick optimum design parameters such as tile size and unroll factors and exploring design spaces.

Furthermore, recent works have also considered data quantization, model pruning, and compression—a core first step of deep CNN implementations on FPGAs as lighter models tend to be faster and inexpensive in terms of resources. These approaches include quantizing the trained weights and biases to smaller precisions (bits), as small as one-bit quantization [68]. Although such quantizations are highly hardware efficient or fast, they are also prone to severe accuracy loss. Another related optimization mechanism is to exploit the sparsity of trained networks

weights [76].

In summary, current hardware-acceleration implementations utilize one or more of the above techniques for maximum throughput, efficient resource utilization, and low-power consumption while maintaining the smallest possible drop in accuracy. However, these objectives largely contradict one another, and researchers end up with designs that are inefficient in terms of their accuracy, resource use and power efficiency. However, in this article, we give an in-depth explanation of our design and implementation of an object detection accelerator with the objective of fair resource utilization while preserving the highest possible accuracy and detection speed. After all, object detection should be fast and accurate, not only fast or accurate.

8.4 Background

8.4.1 Overview of Object Detection Models

Two deep CNN-based approaches dominate modern generic object detection implementations: two-stage [17, 22] and one-stage object detectors [26, 29, 30]. As the names imply, two-stage object detectors perform detection in two core stages; the first stage proposes the regions and the second stage classifies and scores each proposed region by object class and location. One-stage detectors, however, complete both the localization and classification in one forward pass using one unified deep CNN network. Due to one-stage detectors' unified single-network approach, they are relatively less complex, lightweight, and faster although they can be somewhat though not significantly less accurate. As a result, many hardware acceleration implementations of object detection networks concentrate on these network types [77]. One well-known and widely implemented object detection network is YOLO [26], particularly YOLO versions 2 and 3, or YOLOv2 [27] and YOLOv3 [28] as they are commonly known, respectively. As a result, we also target one-stage object detector YOLO, particularly YOLOv2, as the basis for our hardware-accelerated object detection design and implementation.

Commonly, an object detection model is a repurposed image classification network obtained after removing the output layer of a classifier and adding a few more convolution layers tailored toward detection. For example, YOLOv2 repurposes a classification network called Darknet-19, a network with 19 convolutional layers—hence the name Darknet-19—into a unified object detection network with a few extra layers, as shown in Figure 8.1 or in greater detail in Table 8.1. YOLOv2 has 31 layers, excluding the batch normalization and activation layers. The 31 layers comprise 23 convolutional layers, 5 max-pooling layers, 1 concatenation layer, 1 route layer, and 1 space-to-depth reorganization layer. Moreover, there is an associated batch normalization and the Leaky Relu activation layer following each convolutional layer, except the final detection head, where the activation is linear.

We will then briefly summarize the working principle of YOLO-based object detectors. YOLO generally perceives an input image as divided into $S \times S$ grids of equal sizes, and each grid cell predicts at least a K object class, confidence score, and bounding box parameters. K is the number of pre-prepared anchor boxes generated from training sets using K-means clustering. In post-processing, the predictions are filtered out using objectness confidence thresholding and non-max suppression mechanisms.

Recent versions of YOLO such as YOLOv3 and YOLOv4 and their derivatives such as Multi-GridDet [78] have multiscale output and are better at handling the detection of varying scales of objects while also very deep and unfortunately heavy for small-scale FPGAs and other embedded systems. There have been various efforts to reduce the size of YOLO while harvesting the benefit of the progressive increase in the network's depth and complexity with no or minimal accuracy loss. Some of these modifications include removing some convolutional layer(s) or batch normalization layers from the original implementation [79], reshaping the output layer [78, 80] or converting the one-hot encoding into binary encoding [81]. Following this section, we briefly

summarize some of the core layers of YOLOv2-based object detection networks.

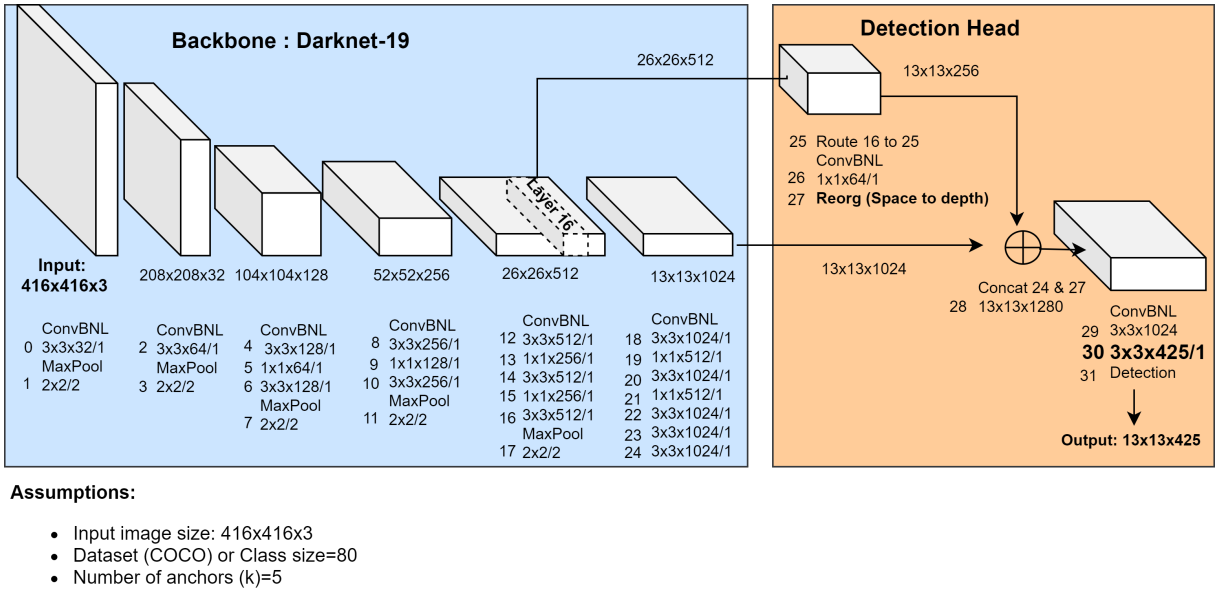


Figure 8.1: *YOLOv2 object detection model layers and their corresponding tensor shapes. ConvBNL stands for convolution followed by batch normalization and Leaky Relu activation layers. Numbers 0–31 show the YOLOv2 layers. For a detailed understanding of each layer's parameter size, refer to Table 8.1.*

8.4.2 Convolution Layer

The convolution layer is the core and computation-intensive part of CNN-based networks, reportedly taking over 90% of the network's execution time [72]. Consider Figure 8.2 showing a particular convolutional layer with an input feature map (IFM) tensor of shape $X = N_{if} \times N_{ix} \times N_{iy}$, weight kernel of shape W $= N_{of} \times N_{if} \times N_{kx} \times N_{ky}$ and an output feature map (OFM) of shape $O = N_{of} \times N_{ox} \times N_{oy}$. The subscripts $_{of}$, $_{ox}$, $_{oy}$ stand for the output feature map depth, row (height), and column (width) of the output feature map. Similarly, subscripts $_{if}$, $_{ix}$, $_{iy}$ serve the same purpose but for the input feature map. We will stick to these notations throughout the paper for consistency.

Convolution is thus a process of repeated multiply and accumulate operations of a pre-trained weight kernel of shape $N_{if} \times N_{kx} \times N_{ky}$ against an input feature map or an input image of a shape $N_{if} \times N_{ix} \times N_{iy}$ by striding the weight kernel across the surface of the input with a stride of size S. This process is repeated N_{of} times—once for each of the N_{of} different kernels yielding an output of size $N_{of} \times N_{ox} \times N_{oy}$. The Equation (8.1) mathematically describes this convolution process.

$$O[m][x][y] = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \sum_{n=0}^{N-1} (X[n][xi][yi] \times W[m][n][i][j]) + B[m] \tag{8.1}$$

where

$$xi = x \times S + i$$

$$yi = y \times S + j$$

$$K = N_{kx} = N_{ky}, \quad N = N_{if}$$

Table 8.1: *YOLOv2 layers and their input and output sizes presented in detail.*

	Layer	Layer Type	Filters	Size/Stride	Input Size	Output Size
Darknet-19 Backbone	0	ConvBNL	32	$3 \times 3/1$	$416 \times 416 \times 3$	$416 \times 416 \times 32$
	1	Max Pool		$2 \times 2/2$	$416 \times 416 \times 32$	$208 \times 208 \times 32$
	2	ConvBNL	64	$3 \times 3/1$	$208 \times 208 \times 32$	$208 \times 208 \times 64$
	3	Max Pool		$2 \times 2/2$	$208 \times 208 \times 64$	$104 \times 104 \times 64$
	4	ConvBNL	128	$3 \times 3/1$	$104 \times 104 \times 64$	$104 \times 104 \times 128$
	5	ConvBNL	64	$1 \times 1/1$	$104 \times 104 \times 128$	$104 \times 104 \times 64$
	6	ConvBNL	128	$3 \times 3/1$	$104 \times 104 \times 64$	$104 \times 104 \times 128$
	7	Max Pool		$2 \times 2/2$	$104 \times 104 \times 128$	$52 \times 52 \times 128$
	8	ConvBNL	256	$3 \times 3/1$	$52 \times 52 \times 128$	$52 \times 52 \times 256$
	9	ConvBNL	128	$1 \times 1/1$	$52 \times 52 \times 256$	$52 \times 52 \times 128$
	10	ConvBNL	256	$3 \times 3/1$	$52 \times 52 \times 128$	$52 \times 52 \times 256$
	11	Max Pool		$2 \times 2/2$	$52 \times 52 \times 256$	$26 \times 26 \times 256$
	12	ConvBNL	512	$3 \times 3/1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
	13	ConvBNL	256	$1 \times 1/1$	$26 \times 26 \times 512$	$26 \times 26 \times 256$
	14	ConvBNL	512	$3 \times 3/1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
	15	ConvBNL	256	$1 \times 1/1$	$26 \times 26 \times 512$	$26 \times 26 \times 256$
	16	ConvBNL	512	$3 \times 3/1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
	17	Max Pool		$2 \times 2/2$	$26 \times 26 \times 512$	$13 \times 13 \times 512$
	18	ConvBNL	1024	$3 \times 3/1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
	19	ConvBNL	512	$1 \times 1/1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$
	20	ConvBNL	1024	$3 \times 3/1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
	21	ConvBNL	512	$1 \times 1/1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$
	22	ConvBNL	1024	$3 \times 3/1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
	23	ConvBNL	1024	$3 \times 3/1$	$13 \times 13 \times 1024$	$13 \times 13 \times 1024$
Detection Head	24	ConvBNL	1024	$3 \times 3/1$	$13 \times 13 \times 1024$	$13 \times 13 \times 1024$
	25	Route 16				$26 \times 26 \times 512$
	26	ConvBNL	64	$1 \times 1/1$	$26 \times 26 \times 512$	$26 \times 26 \times 64$
	27	Reorg		/2	$26 \times 26 \times 64$	$13 \times 13 \times 256$
	28	Concat 24 and 27				$13 \times 13 \times 1280$
	29	ConvBNL	1024	$1 \times 1/1$	$13 \times 13 \times 1280$	$13 \times 13 \times 1024$
	30	ConvBNL	425	$1 \times 1/1$	$13 \times 13 \times 1024$	$13 \times 13 \times 425$
	31	Detection-head (output post-processing)				

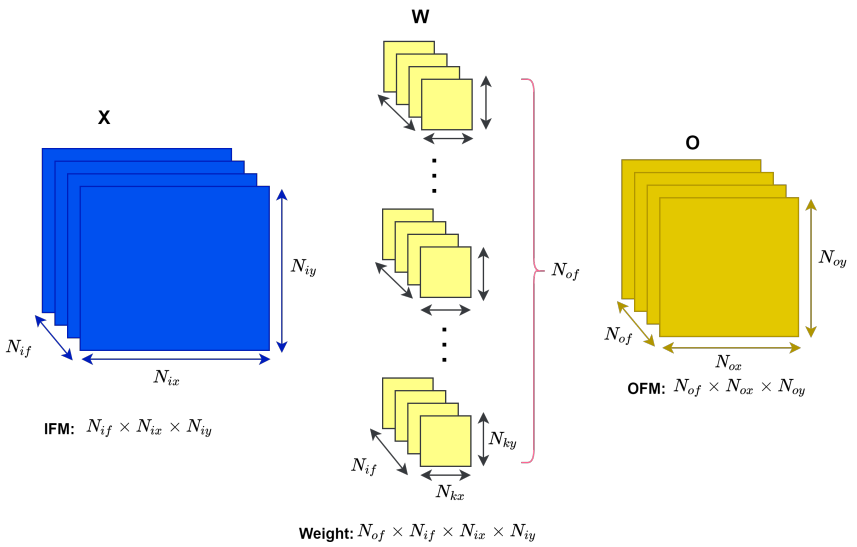


Figure 8.2: *Feature maps and weight tensors representation of a particular convolution layer. Although not indicated in the figure, usually convolution layers have also learned bias (B) parameters of size equal to the number of output channels, that is N_{of}. That is one bias value per output channel.*

$$m \in \{0, N_{of}\}, \quad x \in \{0, N_{ox}\}, \ \& \ y \in \{0, N_{oy}\}$$

Equation (8.1) assumes that the width and height of the weight kernels to be equal as is the case with YOLOv2 and almost all modern CNN-based networks. The relationship between the input and output feature map width and height is also determined using Equations (8.2) and (8.3). The P in the equation stands for the zero-padding of the input feature map so that the resulting output feature map will have either a 'valid' or 'same' shape. Valid is for when the input is not padded, meaning that $P = 0$ and the output will have a slightly shorter width and height compared to the input feature map, whereas in the 'same' convolution, the output and input will have the same width and height and hence P is different from zero.

$$N_{ox} = \frac{N_{ix} + 2P - N_{kx}}{S} + 1 \tag{8.2}$$

$$N_{oy} = \frac{N_{iy} + 2P - N_{ky}}{S} + 1 \tag{8.3}$$

The pseudocode in Listing 8.1 demonstrates that the unoptimized convolution will have six nested loops for a single-input image or input feature map. From this, we can understand that there are $N_{of} \times N_{if} \times N_{ox} \times N_{oy} \times N_{kx} \times N_{ky}$ total multiply–accumulate (MAC) operations for every convolution layer. The X, W, B and the O in the pseudocode stands for IFM, weight, bias and OFM, respectively.

```
for (m=0; m<Nof;m++){
  for (y=0; y<Noy;y+=S){
    for (x=0; x<Nox;x+=S){
      for (n=0; n<Nif;n++){
        for (ky=0; ky<Nky;ky++){
          for (kx=0; kx<Nkx;kx++){
            O[m][x][y]+= X[ni][S*x+kx][S*y+ky] * W[m][n][kx][ky];
          }
        }
      }
      O[m][x][y] += B[m];
    }
```

```
    }
  }
```

Listing 8.1: *Unoptimized standard convolution pseudocode for batch-size = 1.*

8.4.3 Pooling Layer

Another common layer type in a modern object detection CNN network is a pooling layer. A pooling layer reduces the preceding layer's spatial dimensions and facilitates the prospect of a deeper network. Moreover, it also increases the network's translation invariance by omitting pixels from a feature map through either maximum or average pooling. It also minimizes, to a lesser extent, network overfitting to the training dataset. It is worth noting that the pooling layer has no trainable parameter. Accordingly, more recent state-of-the-art models utilize alternative layers such as up-sampling and down-sampling to enable learned pooling. The pooling layer, particularly the max-pooling layer, has three nested loops as depicted in pseudocode Listing 8.2.

```
for (no=0; no<Nof;no++)
  for (y=0; y<Noy;y+=S)
    for (x=0; x<Nox;x+=S)
      O[no][x][y]=Max(X[n0][x:x+S][y:y+S])
```

Listing 8.2: *Original max-pooling pseudocode.*

8.4.4 Depth-to-Space or Space-to-Depth Reorganization Layer

The other layer type in YOLOv2 is a reorganization layer, known in the TensorFlow framework as the space-to-depth or depth-to-space layer. These reorganization processes reshuffle the previous layer's feature maps into either channel-wise deeper feature maps, shown in Figure 8.3, or spatially wider feature maps, shown in Figure 8.4. Reorganization is commonly performed for the facilitation of the concatenation of two or more layers of different shapes. In our case, layer 27 of YOLOv2 is a space-to-depth reorganization of layer 26 with a block-size of $B = 2 \times 2$ (seen Figure 8.1 or Table 8.1). The following layer, layer 28, concatenates the output of layers 24 and 27. Note that, in the reorganization layer, there are no learned or learnable parameters (or hyperparameters).

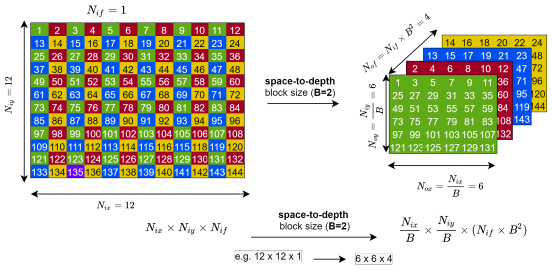


Figure 8.3: *Space-to-Depth*

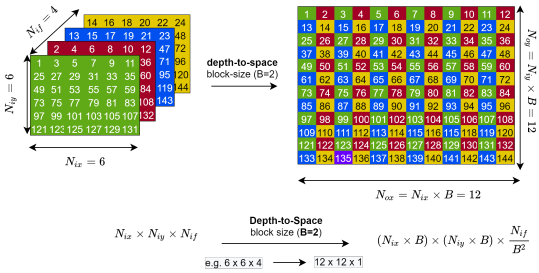


Figure 8.4: *Depth-to-Space*

8.4.5 Batch Normalization Layer

The batch normalization layer is inter-layer data normalization, which differs from input normalization during pre-processing, to accelerate object detection training convergence by minimizing internal variance among layers. This usually comes after the convolution layer, just before the non-linear activation layer. In short, batch normalization involves four mathematical steps: (1) calculating the mean of an output of the convolution layer Equation (8.4); (2) calculating the variance of an output of the convolution layer, Equation (8.5); (3) normalizing the convolution output so that its mean and variance become 0 and 1, respectively, Equation (8.6); and finally

(4) scaling and shifting the normalized data using learned hyperparameters γ and β, Equation (8.7). The value after the fourth step will be input to the next layer, which is going to be Leaky Relu in the YOLOv2 object detector.

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{8.4}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{8.5}$$

$$\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{8.6}$$

$$y_i \leftarrow \gamma\widehat{x_i} + \beta \equiv BN_{\gamma,\beta}\left(x_i\right) \tag{8.7}$$

8.4.6 Leaky Relu Activation Layer

In YOLOv2, the Leaky Relu activation function given by Equation (8.8) is used for the non-linear transformation of the feature map pixels yielded from the preceding layer—in our case, the batch normalization layer.

$$y = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & otherwise, where\ \alpha \in [0,1] \end{cases} \tag{8.8}$$

8.5 The Proposed Hardware Acceleration of Object Detection Inference

8.5.1 General Overview

We propose a hardware–software coprocessing dual system where the computation-intensive layers, namely all convolution, max-pooling, and activation layers, are offloaded to an FPGA (Programmable Logic or PL) to benefit from FPGA's parallelism capabilities. In contrast, layers that are non-computation oriented, such as the reorg and route layers, are processed by a processor onboard our test system (processor system or PS), typically an ARM processor. Moreover, the PS supervises the overall control of the detection network's end-to-end flow, including the pre- and post-processing stages.

Figure 8.5 shows the overall architecture of our proposed object detection accelerator. As seen from the figure, a pre-trained YOLOv2 weight, bias and input-images are stored on a DDR memory of the host system which also contains the processor and the software accelerated portions of our object detection network. All contents of the DDR memory are 16-bit quantized. An AXI-DMA interface connects the host systems' PS and DDR memory with the PL side's custom accelerator, where the heavy-duty arithmetic of the convolution, max-pooling and Leaky Relu are executed. In general, the core features of our hardware-accelerated object detection inference includes:

- A highly hardware resource-efficient and optimized convolution and max-pool processors based on standard optimization techniques such as loop tiling, unrolling and convolution loop reordering;
- Per-layer dynamic 16-bit data quantization of the weight, bias, IFM and OFM;

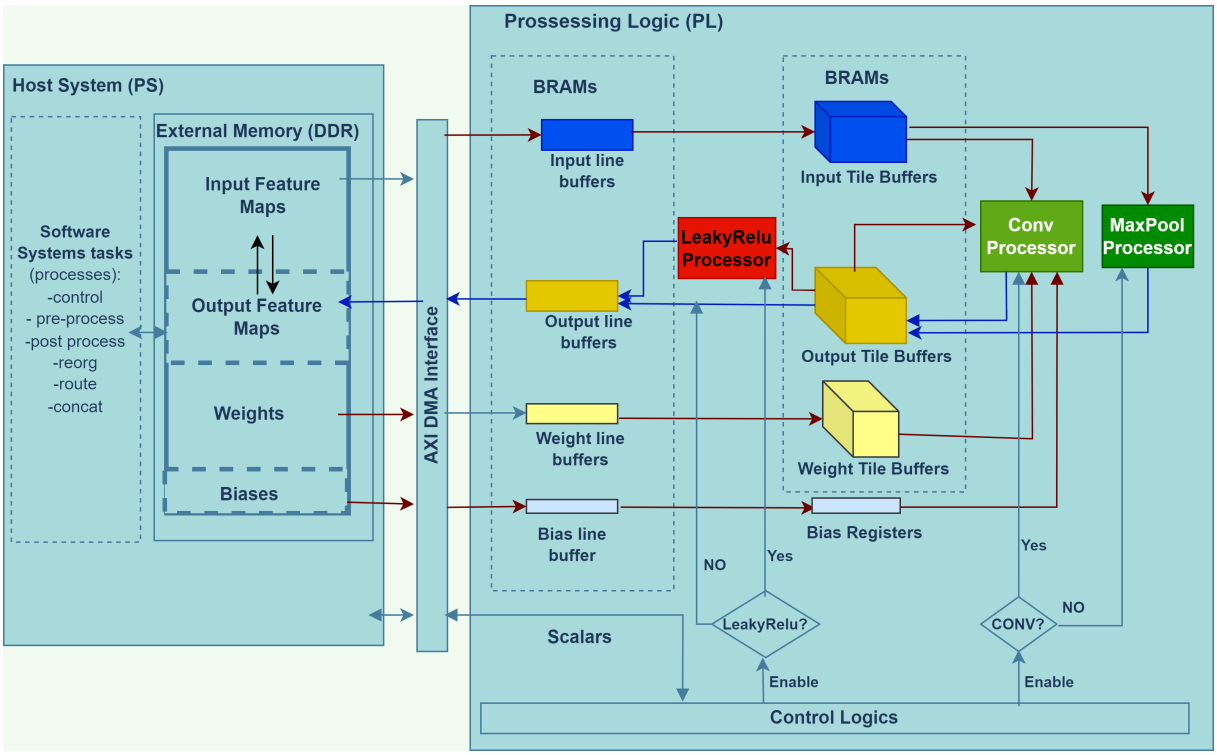


Figure 8.5: *Overall architecture of the proposed HW/SW co-design of the inference acceleration system.*

- Double buffering-based memory read, computation and writeback for smooth convolution acceleration, one that avoids memory access from becoming its bottleneck.

We shall then discuss these features of our design choice one by one in detail.

8.5.2 Loop Tiling

As discussed in earlier sections, current state-of-the-art object detectors are deep and have millions of trainable parameters and tens or hundreds of megabytes. As a result, breaking the inputs and outputs into FPGA-manageable chunks of blocks is an inevitable part of the hardware-accelerated implementation of these state-of-the-art models. Recall how Figure 8.2 shows a particular convolutional layer with an input tensor of shape $X = N_{if} \times N_{ix} \times N_{iy}$, weight kernel of shape $W = N_{of} \times N_{if} \times N_{kx} \times N_{ky}$ and an output feature map (OFM) of shape $O = N_{of} \times N_{ox} \times N_{oy}$. To better illustrate loop tiling, we return to our earlier Figure 8.2; however, this time, we include the loop tiling information, as seen in Figure 8.6, with the white-shaded regions indicating the tile sizes.

The two following equations give the relationship between the input and output tiles' width and height:

$$T_{ix} = (T_{ox} - 1)S + N_{kx} \tag{8.9}$$

$$T_{iy} = (T_{oy} - 1)S + N_{ky} \tag{8.10}$$

Some prior works relied on custom-built algorithms such as roofline modeling to determine the optimum tile size parameters. Instead, we opt for a simplistic but intuitive strategy or criterion to specify the appropriate tile sizes that guarantee data reuse and optimized resource utilization. Our simplistic yet intuitive strategy is based on the following assumptions or criteria:

1. For the efficient utilization of the scarce on-chip memory of the FPGA (that is, the BRAM or block random access memory), the max-pooling and convolution layers shall use the

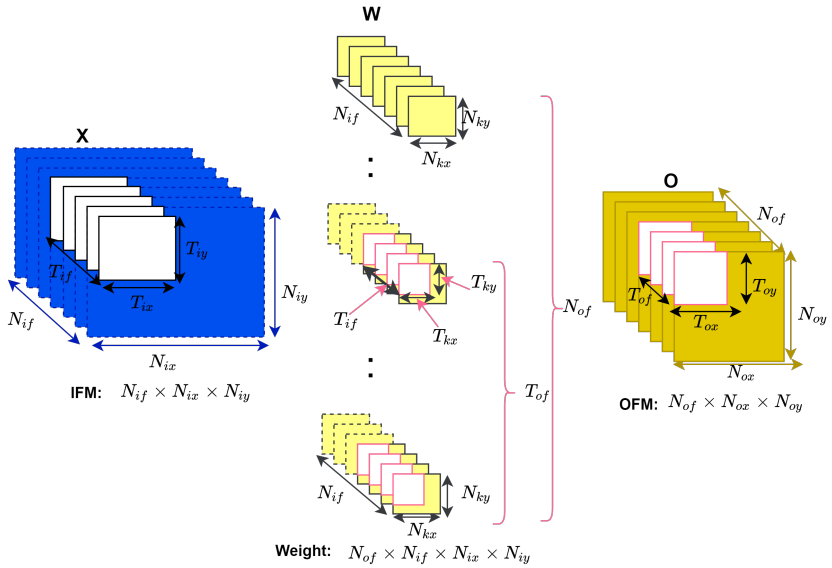


Figure 8.6: *Convolution layer with loop tiling of the input, output and weight 'pixels' or 'feature maps'.*

same memory blocks for buffering. This is possible since the two layers never happen simultaneously but one after another. Thus, we enforce resource-sharing among the two core processing elements.

2. The bigger the data that we can fit on the on-chip memory through burst transfer is, the better it is to avoid frequent external memory access because external memory access is relatively slow compared to the actual computation.

3. Determining the buffer sizes should not be solely based on the layers with the biggest width, height and/or depth. Instead, tile sizes should be a common divisor of all or most layers so as not to assign excessively-big buffers for most of layers, thereby wasting on-chip memory and energy or excessively small buffers, increasing external memory transaction frequencies.

In YOLOv2, the convolution stride (S) equals one, whereas the max-pooling stride is two. Based on our strategy of using shared buffers for max-pooling and convolution and the fact that max-pooling requires a buffer size almost twice that required by convolution for the same output tile of size $T_{of} \times T_{ox} \times T_{oy}$, we base our tile size selection based on the demands of max-pooling layers. By substituting the value of $S = 2$, we can then rewrite Equations (8.9) and (8.10) as follows:

$$T_{ix} = (T_{ox} - 1) \times 2 + 2 = 2 \times T_{ox} \tag{8.11}$$

$$T_{iy} = (T_{oy} - 1) \times 2 + 2 = 2 \times T_{oy} \tag{8.12}$$

Table 8.2 shows the tensor shapes, corresponding tile sizes and the number of external memory read- or write-access iterations. The number of BRAMs (on-chip buffers) required for each tile is calculated as:

$$Number\ of\ BRAM\ per\ Tile = \frac{Tile\ Size \times Data\ Width}{Size\ of\ One\ BRAM} \tag{8.13}$$

However, depending on the convolution loop arrangement and array partitioning, the actual required BRAM would be larger than what we obtain by Equation (8.13). Moreover, as seen from

Table 8.2: *Loop tile sizes and memory read–write access iterations to or from the tile buffers.*

Tensors	Original Shape	Tile Sizes (Shapes)	Number of External Memory Access (Either to Read from or Write to DDR Memory)
IFM	$N_{if} \times N_{ix} \times N_{iy}$	$T_{if} \times T_{ix} \times T_{iy}$	$\lceil \frac{N_{of}}{T_{of}} \rceil \times \lceil \frac{N_{if}}{T_{if}} \rceil \times \lceil \frac{N_{ox}}{T_{ox}} \rceil \times \lceil \frac{N_{oy}}{T_{oy}} \rceil$
OFM	$N_{of} \times N_{ox} \times N_{oy}$	$T_{of} \times T_{ox} \times T_{oy}$	$\lceil \frac{N_{of}}{T_{of}} \rceil \times \lceil \frac{N_{ox}}{T_{ox}} \rceil \times \lceil \frac{N_{oy}}{T_{oy}} \rceil$
Weights	$N_{of} \times N_{if} \times N_{kx} \times N_{ky}$	$T_{of} \times T_{if} \times T_{kx} \times T_{ky}$	$\lceil \frac{N_{of}}{T_{of}} \rceil \times \lceil \frac{N_{if}}{T_{if}} \rceil$
Biases	N_{of}	T_{of}	$\lceil \frac{N_{of}}{T_{of}} \rceil$

the overall architecture in Figure 8.5, each tile has an associated line buffer for burst transfer, adding up the total BRAM utilization of the hardware solution.

Finally, according to first of the aforementioned criteria, the input and output tile buffer sizes (only the width and height, T_{ix} and T_{iy} for input tile, and T_{ox} and T_{oy} for output tile) are determined based on the max-pooling layer and Equations (8.11) and (8.12). However, T_{if} and T_{of}'s choices require considering the implemented custom convolution accelerator and available resources, such as the DSPs and logic cells and the aforementioned criteria. We analyzed the YOLOv2 layers for setting T_{if} and observed that N_{ix}'s minimum and maximum values are 3 and 1280, corresponding to the input and layer 29, respectively. Similarly, the minimum and maximum values of N_{of} are 32 and 1024, respectively. Although we would like to assign as big a buffer as possible for the tiles according to the second of the aforementioned criteria, we should also respect condition 3, i.e., assigning a suitable buffer for all the layers of YOLOv2. Accordingly, we selected $T_{if} = 4$, which is neither excessively larger than the minimum nor excessively small, causing frequent memory access. However, T_{of} can be set to 32 or more based on the available BRAM and DSP, considering we designed a convolution processor with $T_{if} \times T_{of}$ simultaneous MACs (explained under Section 8.5.5). The final tile size choices of our implementation are discussed in Results and Discussions section, Section 8.6.

8.5.3 Double Buffering

To further increase the throughput of our hardware accelerator, we use the concept of double buffering, also called ping-pong buffering. Double buffering helps to overlap memory read, compute, and writeback operations, solving the memory access bottleneck. It also requires twice as much memory as implementation without double buffering, resulting in high resource consumption. We implement double-buffering using an approach similar to that in [47]. We implement a two-stage ping-pong: one for reading input tiles (weight and input feature maps) and another for writing back the final convolution results. As seen in Figure 8.7, during the first iteration of the innermost loop, the input feature map and weight tiles are brought to their corresponding buffers (IFM_buffer0, Weight_buffer0). In the next iteration, while the convolution processor simultaneously performs a convolution operation on the earlier inputs, the next batch of inputs are loaded onto the second set of corresponding buffers (IFM_buffer1, Weight_buffer1). The convolution results are kept on either OFM_buffer0 or OFM_buffer1 until the innermost loop is completed. The Algorithm 1 shows the ping-pong process more precisely and briefly. Two Boolean variables (pingpong_ifm, pingpong_ofm) control the double buffering sequencing, while the input read, compute and output writeback stages are controlled by loop iteration

checks, omitted from the pseudocode for brevity. In general, there are $\lceil \frac{N_{if}}{T_{if}} \rceil + 1$ input tile reads for each output tile writeback and in total there are $\lceil \frac{N_{of}}{T_{of}} \rceil + 1$ writebacks.

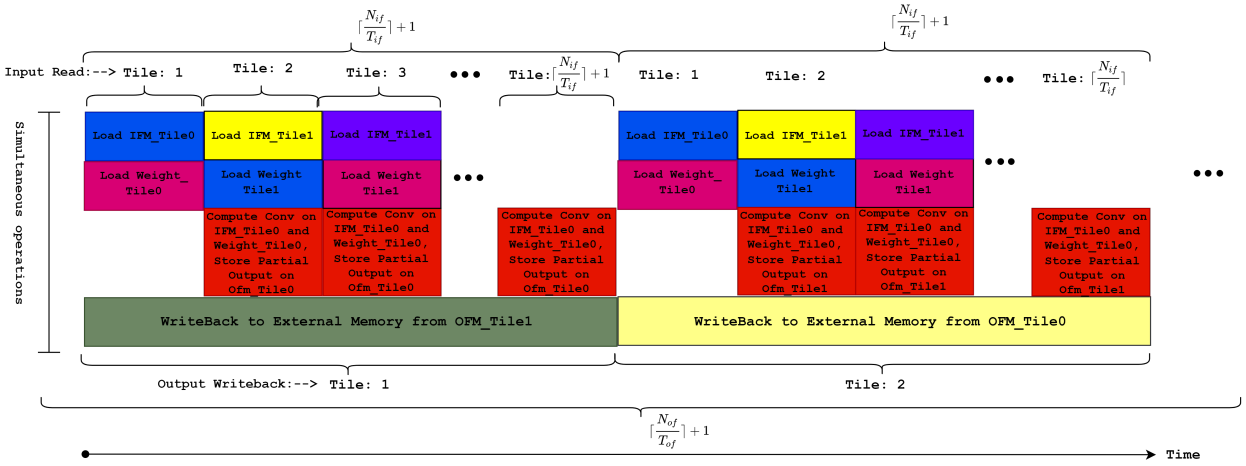


Figure 8.7: *Illustration of double-buffering sequencing.*

8.5.4 Data Quantization and Weight Reorganization

As state-of-the-art object detections model sizes steadily increase to achieve increased performance, the network becomes slower and more resource-demanding. Consequently, the model quantization of trained weights and biases has become an integral part of hardware acceleration implementation. As discussed in our Related Works section, extreme quantizations yield a high-speed model. However, the accuracy loss is usually not worth the speed gain for most real-world application areas of computer vision since a detector should be not only fast, but fast as well as accurate. As a result, instead of extreme quantization, we opted for the 16-bit quantization of the trained weights, biases, and input feature maps.

Quantization converts the trained network parameters from the de facto 32-bit floating-point precision into an $m-bit$ fixed-point precision binary string. The quantized model will be lighter in size and hence faster. To mathematically describe the quantization process, let us consider $W_{float32}$ as the 32-bit (also called single) precision IEEE 754 standard number, and its 16-bit quantized equivalent as $W_{quant16}$. To quantize $W_{float32}$ into $W_{quant16}$, we first need to determine an integer Q, such that the integer part of $W_{float32}$ could be represented by$n \geq (m - Q)$ bits, and in our case m is 16 since we target 16-bit quantization. For example, if $W_{float32} = 3.24$, the integer part is $+3$, a small number that can be represented by n = 2 bits. However, considering the potential of $W_{float32}$ as negative, we leave at least three bits for the portion before the decimal point. This leaves our Q to be 13. Once the Q value is determined, the quantized $W_{quant16}$ is calculated as:

$$W_{quant16} = \lfloor W_{float32} \times 2^{Q} \rfloor \tag{8.14}$$

In our example, substituting the Q value gives $W_{quant16} = 3.24 \times 2^{13} = 26542$. Using Equation (8.15), one can reverse the quantized value back into floating precision though a slight difference is expected due to rounding. In fact, the quantization error can also be calculated using the Equation (8.16).

$$W'_{float32} = \lfloor W_{quant16} \times 2^{-Q} \rfloor \tag{8.15}$$

$$error = \mid W'_{float32} - W_{float32} \mid \tag{8.16}$$

Algorithm 1: Illustration of our double-buffering implementation

```
   /* 1. ping-pong write-back or double buffering the output write-back */
 1 for ( tor = 0; tor < N_ox; tor+ = T_ox ) {
 2     for ( toc = 0; toc < N_oy; toc+ = T_oy ) {
 3         for ( tof = 0;tof < (N_of + T_of);tof+ = T_of ) {
 4             pingpong_ofm=false;
 5             for ( tof = 0; tof < (N_of + T_of); tof+ = T_of ) {
 6                 compute_flag = (tof < N_of) ? true: false;
 7                 write_flag = tof > 0 ? true: false;
 8                 if (pingpong_ ofm) then
 9                     compute_conv(ifm, weight, bias, ofm_buffer1,
10                             ifm_buffer0, ifm_buffer1, weight_buffer0,
11                             weight_buffer1,tof1, compute_flag, ...);
12                     writeback_convoutput(ofm_buffer0,ofm, write_flag, ...);
13                     pingpong_ofm=false;
14                 else
15                     compute_conv(ifm, weight, bias, ofm_buffer0,
16                             ifm_buffer0, ifm_buffer1, weight_buffer0,
17                             weight_buffer1, compute_flag, ...);
18                     writeback_convoutput(ofm_buffer1, ofm, write_flag, ...);
19                     pingpong_ofm=true;
   /* the following sequence is inside compute_conv function */
   /* 2. ping-pong tile reads and convolution computation or double buffering of input read
      */
20 pingpong_ifm = false;
21 load_bias(bias, bias_buffer, ...);
22 for ( tin = 0; tin < N_if + T_if; tin+ = T_if ) {
23     if (pingpong_ ifm) then
24         load_convinputtile(ifm, ifm_buffer1,..., tin < N_if);
25         load_weight(weight,weight_buffer1,..., tin < N_if);
26         conv_tile(ifm_buffer0, ofm_buffer, weight_buffer0, bias_buffer, ...,tin > 0);
27         pingpong_ifm=false;
28     else
29         load_convinputtile(ifm,ifm_buffer0,...,tin < N_if);
30         load_weight(weight,weight_buffer0,...,tin < N_if);
31         conv_tile(ifm_buffer1, ofm_buffer, weight_buffer1, bias_buffer,..., tin > 0);
32         pingpong_ifm=true;
```

Algorithm 2: Per-layer 16-bit dynamic quantization of weight

Input: $W_{float32}$
Output: $W_{quantized}, WeightQ$

```
/* Iterate through all N layers of the network.                                   */
for ( n = 0; n < N; n++ ) {
    /* for all convolution layer                                                  */
    if (layerIsConvLayer) then
        read(w_float32, W_float32, n * float32);
        minValue ← 0x7FFF ;                    // 16-bit 2's complement maximum range
        maxValue ← 0x8000;                     // 16-bit 2's complement minimum range
        /* within a layer search the minimum and maximum weight entry!            */
        for ( k = 0; k < (N_of × N_if × N_kx × N_ky); k++ ) {
            if (minValue > w_float32[k]) then
                minValue ← w_float32[k];
            else if (maxValue < w_float32[k]) then
                maxValue ← w_float32[k];
        /* Search the quantization Q value for the layer.                         */
        for ( i = 16; i > 0; i-- ) {
            if (minValue > 0x8000 * 2^-i and maxValue < 0x7FFF * 2^-i) then
                Q ← i;
                break;
            else
            /* min and max values are not in the range of 16-bit, very unlikely.  However,
               one can truncate the numbers to within the range.                  */
        /* tile by tile read, reorganize, quantize and save the quantized weight
           parameters, and its corresponding Q value.                             */
        open(W_quantized, 'w');
        for ( nof = 0; nof < N_of; nof+ = T_of ) {
            for ( nif = 0; nif < N_if; nif+ = T_if ) {
                wbuf ← w_float32[nof : nof + T_of][nif : nif + T_if];
                for ( tk = 0; tk < N_kx × N_ky; tk++ ) {
                    for ( tof = 0; tof < T_of; tof++ ) {
                        for ( tif = 0; tif < T_if; tif++ ) {
                            Wbuf_quantized[tk × T_of × T_if + tof × T_of + tif] ←
                              short(wbuf[tk][tof][tif] × 2^Q);
                write(W_quantized, Wbuf_quantized, short);// Write a tile
                write(WeightQ, Q, short);
```

Similarly to the above explanation, we implemented the weight, input, and output feature map and bias quantization using 16-bit per-layer dynamic quantization. For example, the 16-bit dynamic weight quantization is presented in the pseudocode listing of Algorithm 2. Furthermore, after quantizing, we reorganized the weight tensor from its original 4D shape of $N_{of} \times N_{if} \times N_{kx} \times N_{ky}$, as shown in Figure 8.2, to a 3D shape $N_{kxy} \times N_{of} \times N_{if}$, as seen in Figure 8.8. N_{kxy} is the product of the width and height of the kernel, that is $N_{kx} \times N_{ky} = N_{kxy}$. Hereafter, in our hardware accelerator design, we refer to the weight tensor in this 3D shape rather than its original 4D shape. The quantized weight tensor is saved in the DDR memory in the order of tiles that the convolution processor expects so that a continuous high-speed burst transfer is made to the on-chip buffer.

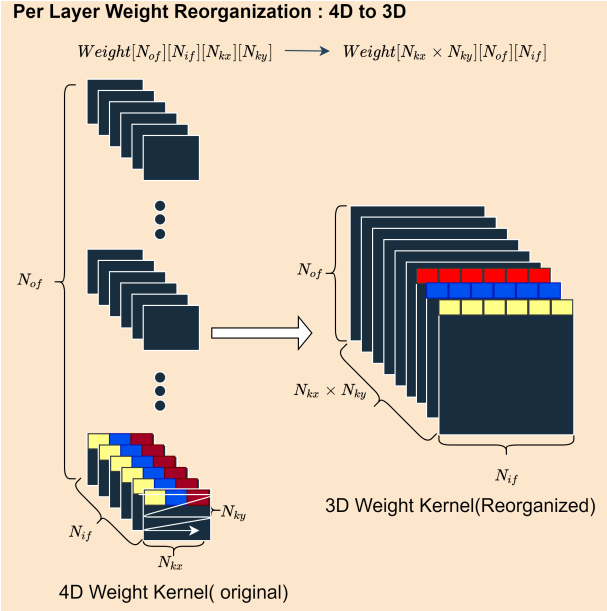


Figure 8.8: *Weight 4D–3D reorganization. The colors are only to show a sample of the corresponding pixels' positions before and after the reorganization of the weight tensor.*

8.5.5 Convolution Processor

The convolution layer is the most resource-demanding and computation-intensive part of the object detector CNN network. As shown in Listing 8.1, the unoptimized convolution has six nested loops, even though they must not always be in the same sequence. We use standard loop tiling, unrolling, and interchange to design an optimized hardware-accelerated version of the convolution. Convolution in fixed-point precision is no longer only an MAC (multiply and accumulate); instead, it is multiply, right shift, and accumulate. Thus, we like to refer to it as MSA operations, not MAC. The amount of right shift is calculated from the Q values of the input quantization Q_X, weight quantization Q_W, and an intermediate value Q_I. We will explain this better with a diagrammatic depiction. Figure 8.9 shows the smallest processing element (PE) unit of our fixed-point convolution implementation. In the figure, two 16-bit numbers with different Q, that is, Q_X for the input pixel and Q_W for the weight 'pixel' pass through the multiplier followed by the right-shift operator and then the accumulator. Had it been a floating-point precision, the decimal point would have been placed at Q_{XW} of the resulting product. However, since this is a fixed-point precision operation, we replace the decimal point with a two's power division or right-shift. Right shift with $Q = Q_{XW}$ would completely discard the fractional points from the result of the product. Instead, we perform a right-shift operation using $Q = Q_{IXW} = Q_{XW} - Q_I$. The best Q_I for 16-bit quantization is $Q_I = 15$ since this value

leaves the maximum room for the decimal parts without completely discarding the fractional value. One might refer to this as an intermediate or partial sum quantization. Note that we also perform an output quantization after Leaky Relu to convert the 32-bit partial sum back to 16-bit and write back the result of the output quantization to the DDR memory through a pipelined burst-transfer.

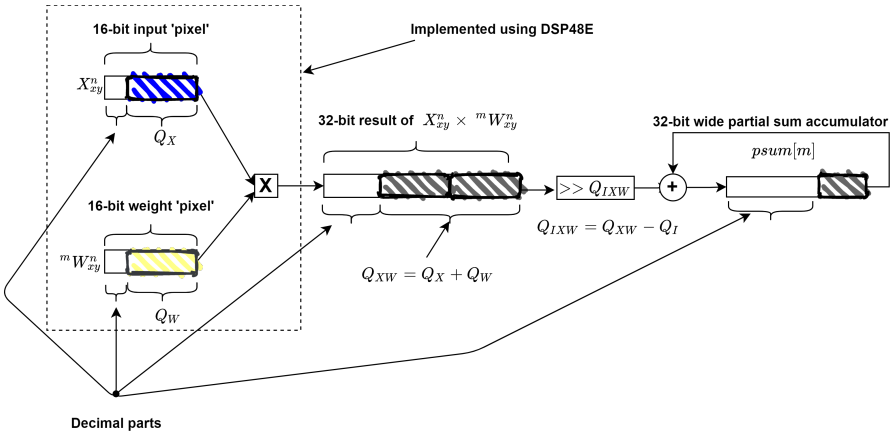


Figure 8.9: *Convolution processing element and its working procedure.*

In general, our convolution processor has $T_{of} \times T_{if}$ fully unrolled multipliers followed by fully unrolled $T_{of} \times T_{if}$ right-shift operation and $T_{of} \times T_{if}$ partial adder trees fully unrolled in the T_{of} dimension and pipelined with the smallest possible initiation interval (II = 1) in T_{if} dimensions. The overall architecture of the designed convolution processor is shown in Figure 8.10.

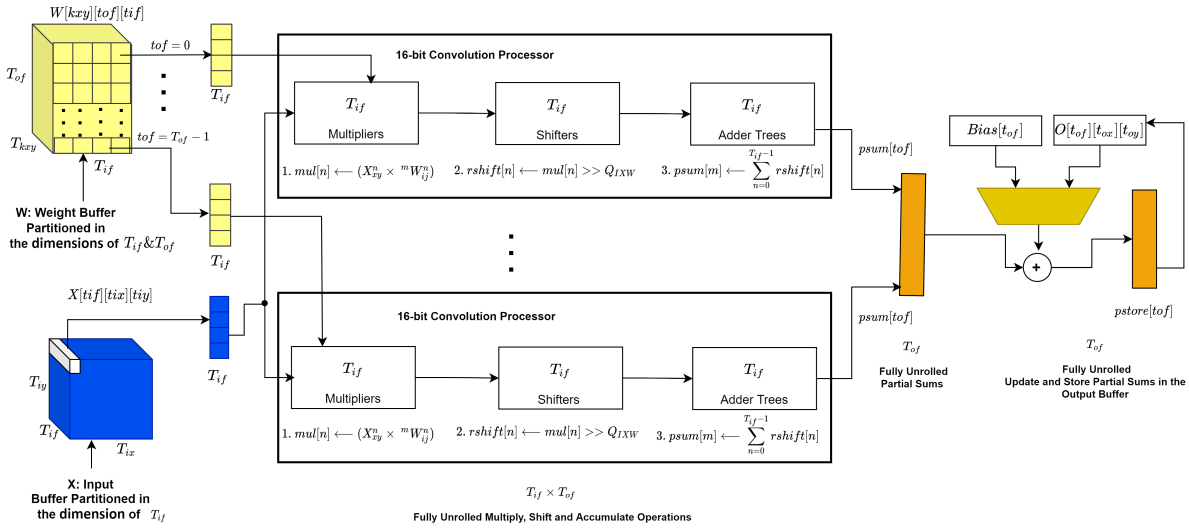


Figure 8.10: *Convolution processor architecture.*

Given the overall design of the convolution processor, the next target was to determine the optimum sequence of the nested loops of convolution. An optimum design for the convolution loops needs to minimize the number of partial sum store and read operations, utilize fewer logic cells, and take full advantage of the redundant onboard resources of the FPGA and DSPs for parallelism, all while being energy efficient. To this point, we tested many possible arrangements of the convolution nested loops, and we finally came down to two contending choices given the limited resources of our development boards. These two competing implementations of the convolution compute function, also briefly mentioned under the double buffering section (see Algorithm 1), are

given by Listing 8.3 and 8.4. In the first version, we obtain the lowest partial sum read and write. However, the convolution kernels are not fixed for all convolution layers. Instead, they alternate between 1×1 and 3×3 in YOLOv2. As a result, placing the loops labeled _nki and _nkj in the middle of the nested loops increases the iteration control hardware, consumes more logic cells and increases latency. We compared it against the second version given by Listing 8.4 and found that Listing 8.3 is three times slower. Our final optimized convolution accelerator was thus chosen to be the one mentioned in Listing 8.4.

To summarize some of the core features of our convolution accelerator, we mention the following key points:

- Per block (tile), the convolution compute latency is given by the Equation (8.17) below:

$$(N_{kx} \times N_{ky} \times T_{ox} \times T_{oy} + C) \times \frac{1}{F_{clk}} \tag{8.17}$$

  C stands for the 'constant' referring to the number of cycles needed to perform the fully unrolled inner operations commented 1–4 in the pseudocode Listing 8.4 and loop iterations control logic. In our implementation, C is equal to either 13 or 21 based on the kernel types, 1×1or 3×3, respectively. F_{clk} stands for clock frequency.

- The total compute latency for a convolution layer is calculated as:

$$\lceil \frac{N_{of}}{T_{of}} \rceil \times \lceil \frac{N_{ox}}{T_{ox}} \rceil \times \lceil \frac{N_{oy}}{T_{oy}} \rceil \times \lceil \frac{N_{if}}{T_{if}} \rceil \times (N_{kx} \times N_{ky} \times T_{ox} \times T_{oy} + C) \times \frac{1}{F_{clk}} \tag{8.18}$$

- The total number of multiply, shift and accumulate operations per convolution layer is calculated as:

$$3 \times \lceil \frac{N_{of}}{T_{of}} \rceil \times \lceil \frac{N_{ox}}{T_{ox}} \rceil \times \lceil \frac{N_{oy}}{T_{oy}} \rceil \times \lceil \frac{N_{if}}{T_{if}} \rceil \times (N_{kx} \times N_{ky} \times T_{ox} \times T_{oy} \times T_{of} \times T_{if}) \tag{8.19}$$

```
int32_t lineinput[Tif];
int32_t pmul[Tif];
int32_t rshift[Tif];
int32_t psum[Tof];
int32_t pstore[Tof];
_trconv:for(tr = 0;tr < min(Tox,Nox);tr++){
  _tcconv:for(tc = 0;tc < min(Toy,Noy);tc++){
    //1. clear
    _pmulclear:for(tm = 0;tm <Tof;tm++){
      #pragma HLS unroll //PIPELINE II=1
      psum[tm]=0;
    }
    //2. compute multiply, shift and accumulate
    _nkiconv:for(i =0;i < Nkx; i++){
      _nkjconv:for(j = 0;j < Nky; j++){
        tix = tr*Kstride + i;
        tiy = tc*Kstride + j;
        tkxy = i*Ksize + j;
        _tnminiInput:for(tn = 0;tn <Tn;tn++){
          #pragma HLS unroll
          lineinput[tn]= X[tn][tix][tiy];
        }
        _tmconv:for(tm = 0;tm < Tof;tm++){
          #pragma HLS unroll
          _tnconv1:for(tn = 0;tn <Tif;tn++){
            #pragma HLS unroll
```

```
              pmul[tn]= W[tkxy][tm][tn]*lineinput[tn];
            }
            _tnconv2:for(tn = 0;tn <Tif;tn++){
              #pragma HLS unroll
              rshift[tn]= pmul[tn]>>Qixw;
            }
            _tnconv3:for(tn = 0;tn <Tif;tn++){
              #pragma HLS unroll
              psum[tm]+= rshift[tn];
            }
          }
        }
      }
      //3. update
      _psupdate:for(tm = 0;tm <Tof;tm++){
        #pragma HLS unroll
        if(n==0){
          pstore[tm] = B[tm] + (psum[tm]);
        }
        else{
          pstore[tm] = O[tm][tr][tc]+ (psum[tm]);
        }
      }
      //4. store
      _psstore:for(tm = 0;tm <Tof;tm++){
        #pragma HLS unroll
        O[tm][tr][tc]= pstore[tm];
      }
    }
  }
```

Listing 8.3: *Version 1: Optimized convolution pseudocode on input and weight tile.*

```
int32_t mul[Tif];
int32_t rshift[Tif];
int32_t psum[Tof];
int32_t pstore[Tm];
_nkiconv:for(i =0;i < Nkx; i++){
  _nkjconv:for(j = 0;j < Nky; j++){
    _trconv:for(tr = 0;tr < min(Tox,Nox);tr++){
      _tcconv:for(tc = 0;tc < min(Toy,Noy);tc++){
        //1. clear partial sum
        _pmulclear:for(tm = 0;tm<Tof;tm++){
          #pragma HLS unroll
          msa[tm]=0;
        }
        //2. compute multiply, shift and accumulate
        _tmconv:for(tm = 0;tm < Tof;tm++){
          #pragma HLS unroll
          //2.1 multiply
          _tnmultiply:for(tn = 0;tn <Tif;tn++){
            #pragma HLS unroll
            mul[tn]= W[i*Nkx+j][tm][tn]*
            X[tn][tr*S + i][tc*S + j];
          }
          //2.2 right-shift for decimal point consideration
          _tnshift:for(tn = 0;tn <Tif;tn++){
            #pragma HLS unroll
            rshift[tn]= mul[tn]>>Qixw;
          }
          //2.3 accumulate to partial sum
          _tnaccumulate:for(tn = 0;tn <Tif;tn++){
            #pragma HLS unroll
```

```
            psum[tm]+= rshift[tn];
          }
        }
        //3. update stored partial sum
        _pupdate:for(tm = 0;tm <Tof;tm++){
          #pragma HLS unroll
          if(i ==0 && j==0 && n==0){
            pstore[tm] = B[tm] + (psum[tm]);
          }
          else{
            pstore[tm] = O[tm][tr][tc]+ (psum[tm]);
          }
        }
        //4. store partial sum
        _pstore:for(tm = 0;tm <Tof;tm++){
          #pragma HLS unroll
          O[tm][tr][tc]= pstore[tm];
        }
      }
    }
  }
}
```

Listing 8.4: *Version 2: Optimized convolution pseudocode on input and weight tile.*

8.5.6 Max-Pooling Processor

As explained earlier, YOLOv2 has five 2×2 max pool layers with a stride of $S = 2$, each following a Leaky Relu activation layer. Although max-pooling does not have an intensive computation complexity, it could benefit from FPGA's parallelism since it works on the individual 'pixels' of the input feature maps. Likewise, we designed a pipelined max-pool accelerator with three selectors and comparators, as seen in Figure 8.11. The input tile size for max-pool has the same depth as the convolution's input feature map depth, which is T_{if}. The pseudocode for the hardware-accelerated max-pool on an input tile is given in Listing 8.5.

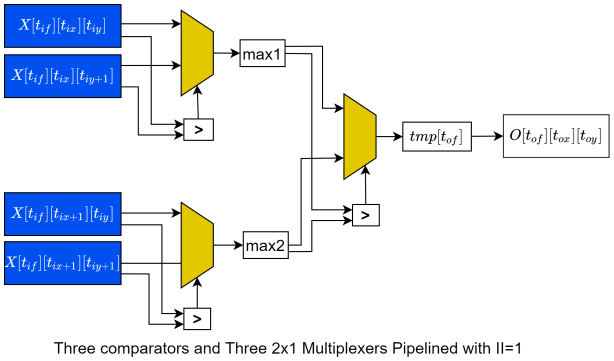


Figure 8.11: *Max-pool processor.*

```
int16_t tmp[Tif];
int16_t tmp1, tmp2,tmp3, tmp4, max1,max2;
_toxmax:for(_tox = 0;_tox < min(Tox,Nox);_tox++){
  _toymax:for(_toy = 0;_toy < min(Toy,Noy);_toy++){
    _tofmax:for(_tof = 0; _tof < min(Tif,N_{if}); _tof++){
      #pragma HLS PIPELINE II=1
      tmp1=X[_tof][_tox*S][_toy*S];
      tmp2=X[_tof][_tox*S][_toy*S+1];
      max1 = (tmp1 > tmp2) ? tmp1 : tmp2;

```

```
        tmp3=X[_tof][_tox*S+1][_toy*S];
        tmp4=X[_tof][_tox*S+1][_toy*S+1];
        max2 = (tmp3 > tmp4) ? tmp3 : tmp4;

        tmp[_tof] = max1 > max2 ? max1 : max2;

      }
      maxstore:for(_tof = 0; _tof < min(Tif,Nif); _tof++){
        #pragma HLS PIPELINE II=1
        O[_tof][_tox][_toy] = tmp[_tof];
      }
    }
  }
```

Listing 8.5: *Optimized max-pool processor for 2 x 2 kernel stride.*

8.5.7 Leaky Relu Hardware Processor

In YOLOv2, following every convolution layer comes a Leaky Relu activation, except for the last convolution layer, which is linear activation. The floating-point equivalent of Leaky Relu was discussed earlier and described using Equation (8.8). In the equation, the constant α is set to 0.1 for YOLOv2, and since we are working on 16-bit fixed precision, we convert the multiplying $\alpha = 0.1$ into 16-bit fixed-point quantized binary string using $Q = 15$. The quantized α is equivalent to base ten 32768 or hex $0xCCC$. In general, the hardware equivalent of Leaky Relu is implemented using the following expression:

```
  tmp_out[i]= (tmp_in[i] < 0) ? (tmp_in[i]*0xccc)>>15 : tmp_in[i];
```

where *tmp_in* is a pixel from the output buffer, and *tmp_out* is the 'pixel' after passing through a Leaky Relu processor. The overall architecture can be seen in Figure 8.5 for clarity.

8.6 Results and Discussions

Although we mainly discussed the FPGA implementation of object detection using YOLOv2, our implementation can be easily configured for other types of similar networks such as DenseYOLO and DDGNet, which are even more lightweight and accurate. We implemented the proposed hardware accelerator using C++, Vitis HLS 2021.1, and Vivado 2021.1. The convolution, max pooling, and Leaky Relu layers are implemented as FPGA accelerated functions. In contrast, the remaining space-to-depth reorganization, concatenation, and route layers, including the input and output pre-processing and post-processing, are performed on the ARM processor onboard our test boards. Following every convolution layer, the batch normalization layer computations were already included in generating the quantized weights and biases, avoiding the need to construct a hardware-equivalent one.

We targeted two Xilinx boards, namely ZYNQ-7000 SoC, specifically Z-7020CGL484-1 and ZCU102 development boards from ZYNQ UltraScale+ MPSoC for the implementation of YOLOv2-based object detection inference. As seen in Table 8.3, the Z-7020CGL484-1 has minimal resources compared to ZCU102. Since double buffering requires twice as many on-chip buffers than an implementation without double-buffering, we had to use different tile sizes for the two boards.

Table 8.4 shows our tile-size design choices and implementation clock frequencies for the two boards. The table also shows the total resources consumed by our hardware accelerator. Both implementations required resources well under the range of the design guidelines of the boards, proving efficient implementation. We also achieved a clock frequency of 150 MHz and 300 MHz for ZYNQ-7020 and ZCU-102, respectively. By combining Equations (8.18) and (8.19), we calculated an overall throughput (giga operations per second (GOP/S)) of 51.06 GOP/S and

Table 8.3: *Available resources onboard ZYNQ-7020 and ZCU102.*

Boards	Z-7020CGL484-1	ZCU102-XCZU9EG-2FFVB1156E
Flip flops (FF)	106,400	548,160
LUT	53,200	274,080
BRAM_18Kb	280	1824
DSP	220	2520

184.06 GOP/S for ZYNQ7020 and ZCU102, respectively. Another helpful metric called DSP efficiency, as coined by [77, 82], measures how efficiently the DSPs in the convolution accelerator are utilized. These define DSP efficiency as a ratio of effective operation or the actual operation that the layer requires over the actual number of operations that the implemented convolution processor performed. According to this definition, our tile size choices and accelerator loop arrangement, the DSP efficiency is 100% for both boards, except for the first and last YOLOv2 layers. Such a high DSP efficiency is partly because of the uniformity of YOLOv2's layers.

Table 8.4: *Design parameter choices and performance measures.*

Boards		ZYNQ 7020	ZCU102
Tile sizes	T_{of}	32	64
	T_{if}	4	4
	T_{ox}	26	52
	T_{oy}	26	52
	T_{ix}	52	104
	T_{iy}	52	104
Resource utilization	FF	22,239(20.9%)	34,076(6%)
	LUT	28,333(53.2%)	97,971 (35%)
	BRAM(18 Kb)	170 (60.7%)	1008(55%)
	DSP	180(81.8%)	291(11%)
Clock (MHz)		150	300
GOP		44.36	44.96
GOPS		51.06	184.06
Power (Watt)		2.78	5.376

Furthermore, we also analyzed the per layer execution latency of YOLOv2 layers for the two boards, as shown in Figure 8.12 for the two implementations. For ZYNQ-7020, the total execution time for end-to-end YOLOv2 object detection inference processing takes 0.868 s. In contrast, the ZCU102 only takes 0.244 s for a single 416×416 RGB image of the COCO object detection dataset. From the figure, layer 29 of YOLOv2 is the slowest, taking up to 26 and 104 ms on ZCU102 and ZYNQ-7020, respectively. On a personal laptop computer of Intel(R) Core i7-7700HQ CPU @ 2.80GHz 16GB RAM Ubuntu 20.04, our YOLOv2 inference takes a maximum of 7 s to infer all bounding boxes and object classes on a single-core single-thread CPU for a single batch of image from COCO dataset with size 416×416. Thus, our FPGA implementation accelerates YOLOv2 inference by up to 28.68 and 8.06 times for ZCU102 and ZYNQ-7020, respectively, compared to the software version on the personal laptop. All this consumes 2.78 watt for ZYNQ-7020 and 5.376 watt on ZCU102, evidencing how our implementation is much more efficient than the other implementations we compared it with.

We compared our YOLOv2 object detection inference implementation with other closely related works, and Table 8.5 summarizes the comparison using different metrics or criteria.

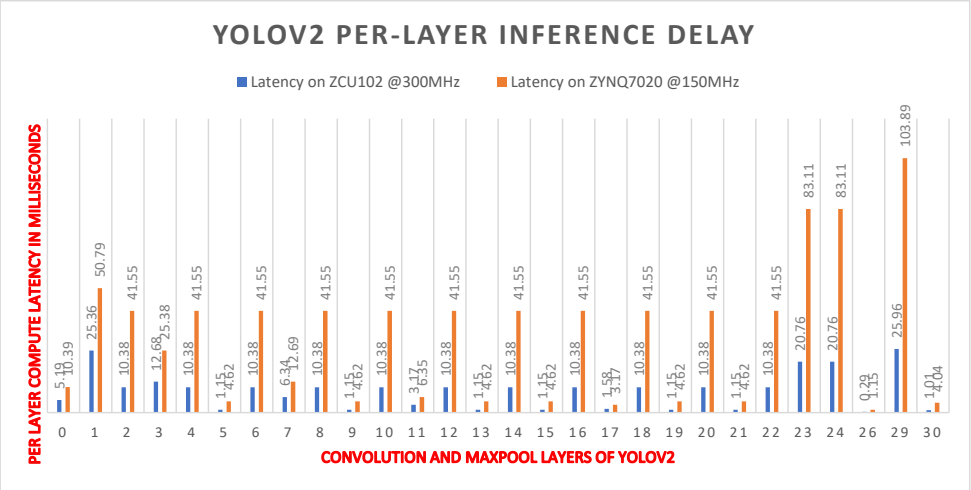


Figure 8.12: *Per-layer latency of YOLOv2 inference on ZYNQ 7020 and ZCU102.*

Although there are many FPGA-based inference accelerations, the main reasons we picked these sample references to compare against our work are that (1) these works are recent; (2) all are one-stage object detection inference accelerations (4) based on YOLO versions and 1 based on SSD); and (3) all are abundantly cited prior works with close resemblance to our approach. As the table shows, our implementation maintains the most resource and power-efficient performance while still having a commendable GOP/S at a frequency as high as 300 MHz and higher DSP efficiency. Moreover, though some entries in the table never reported their accuracy performance, our implementation of YOLOv2 inference on the Pascal VOC 2007 dataset at a resolution of 416 × 416 yielded an mAP of 76.21%, a little below the baseline 32-bit floating precision's 76.8% mAP of the original YOLOv2. The 16-bit quantization of the data and the fixed-point arithmetic of our custom convolution processor explained by Figure 8.9 played a significant role in increasing the mean average precision of our accelerator.

In general, we obtained an efficient hardware-acceleration design scheme that preserves the scarce and precious resources of an FPGA while yielding higher performance at low-energy consumption. We used a shared double-buffered on-chip buffer to conserve memory and avoid memory access becoming a bottleneck to our hardware convolution accelerator. Compared to [77] consuming 100 Watt energy and approximately fifteen times more DSPs than our implementation, we achieve a commendable 0.244 s in execution latency of YOLOv2 at a mere 5.376 Watt and 291 DSPs utilized. Given the fact that we used a 16-bit fixed-point precision, there is a reasonable prospect for our implementation to achieve real-time acceleration by changing our quantization strategy to an 8-bit or mixed precision as well as save more resources and power while still managing to maintain the minimum possible loss in detection accuracy.

Finally, Figure 8.13 shows the sample output of our hardware accelerator performing impeccably well with high accuracy as good as the full 32-bit floating-point precision implemented on our laptop.

8.7 Conclusions

This paper implemented the YOLOv2 inference accelerator on two Xilinx development boards with varying available resources and achieved a resource- and power-efficient accelerator. Our best-performing implementation achieved a commendable throughput of 184 GOP/S and 0.244 s inference time per image using 16-bit fixed point dynamic quantization and consuming only 5.376 watts. In future work, we intend to test different quantization strategies without compromising

Table 8.5: *Comparison of our implementation against other prior works using several metrics.*

	[83]	[84]	[85]	[86]	[77]	**This Work**	**This Work**
Device	Virtex-7 VC707	ZCU102	Zedboard	Intel Arria 10	Intel Stratix 10	ZYNQ -7020	ZCU102
Models	Sim-YOLOv2	YOLOv2	YOLOv3 tiny	YOLOv2	SSD300	YOLOv2	YOLOv2
Design tool	OpenCL	Vivado HLS	Vivado HLS	OpenCL	RTL	Vitis HLS	Vitis HLS
Design scheme	HW	HW/SW	HW/SW	HW/SW	HW/SW	HW/SW	HW/SW
Precision (bits)	1–6	16	16	8–16	8–16	16	16
Frequency (MHz)	200	300	100	200	300	150	300
FF Utilization	115 K (18.9%)	90,589	46.7 K	523.7 K	-	22.2 K(20.9%)	34,076 (6%)
LUT Utilization	155.2 K (51.1%)	95136	25.9 K	360 K	532 K	28.3 K(53.2%)	97,971 (35%)
DSP Utilization	272 (9.7%)	609	160	410	4363	180 (81.8%)	291(11%)
BRAM(18Kb) utilizations	1144 (55.5%)	491	185	1366*	3844*	170 (60.7 %)	1008 (55%)
Throughput (GOP/S)	1877	102.5	464.7	740	2178	51.06	184.06
Power	18.29	11.8	3.36	27.2	100	2.78	5.376
Latency (ms)	-	288	532	-	29.11	868	244
Accuracy (mAP)	64.16	-	-	73.6	76.94	76.21	76.21
Input image size	416×416	416×416	-	416×416	300×300	416×416	416×416

* Intel FPGA with BRAM 20 Kb.

Figure 8.13: *Sample YOLOv2 inference output of our hardware accelerator.*

accuracy and energy efficiency so that our implementation achieves real-time inference.

Part III

Conclusions and Future Works

Chapter

9 Conclusion and Future Works

9.1 General Conclusion

Since the early ages of computers, it has been an early-on quest for humankind to be able to mimic the biologically acquired effortless vision system it has from birth and relies on in its everyday daily life. Though the biological vision system appears seemingly effortless at recognizing objects on images and video, enabling a machine that level of effortlessness and confidence is challenging, if not outright impossible! However, thanks to the success of convolutional neural networks, machines can now mimic some of our audio-visual capability in processing audio, images, and video inputs though they are still very far from being confidently reliable. Moreover, detecting all known objects on an image and localizing them is not trivial compared to image classification due to a) lack of extensive and carefully labeled detection datasets, b) significant randomness of object scale and position distribution on an image, and c) an imbalance of object representation within a dataset. Nonetheless, deeper and more sophisticated CNN-based networks have commendable performance in the accuracy of detecting objects.

The current trend in object detection implementation appears to have two distinctive forms: object detection as a two-step process and another approach that treats object detection as a unified one-stage or one-shot task. Since early object detection challenges focused on maximizing accuracy, both one-shot and two-stage object detections have become immensely deep and easily have millions of trainable parameters. These resulted in slow and heavy networks, both during training and inference. However, nowadays, researchers give speed and efficiency the due focus it deserves as much as object detection accuracy. In this arena, a one-shot object detector unified and holistic approach to solving object detection problems has garnered much more attention than two-stage object detectors. However, the accuracy of one-stage object detectors usually trails behind two-stage detectors. Thus, it is an active research endeavor to balance the speed, accuracy, and complexity of one-stage detectors so that one metric's success will not significantly trail the other metrics.

Considering this observation, we researched ways to improve the speed and accuracy of current state-of-the-art object detection implementations before proposing and implementing an FPGA acceleration of the object detectors. We found YOLO-like detectors as an intuitive approach to localization and classification problems, meaning object detection, due to their uniform network construct and global reasoning of object localization and classifications. The uniform construct of the YOLO-based object detection network structure means easy mapping to hardware implementation. Moreover, the detection approach appears perceptive and has a commendable balance of speed versus accuracy trade-off compared to other relevant detectors. This assessment justified our choice of detection implementation paradigms and motivated us to propose a serious of gradually improving object detection models with higher accuracy, less weight, and faster.

This thesis work proposed three object detection models, two based on YOLOv2 and one based on YOLOv3. As detailed in chapter 5, our first object detection model, DDGNet, proposed restructuring the YOLOv2 detection using binary encoding instead of the defacto one-hot encoding. Moreover, restructuring the detection head also meant using fine-grained grids to

deep search objects of interest throughout the surface of an input image or frame of video, unlike YOLOv2's coarse-grained grid sizes. We also proposed a robust non-max-suppression technique called ThreeWayNMS to filter out the same object's redundant predictions and minimize false positives. Our DDGNet model achieved higher accuracy while being lighter. Moreover, since DDGNet uses a modified (extended) binary encoding, its output layer has fewer parameters and can easily be extended to perform a true generic object detector with thousands of object categories.

Our second object detection model, DenseYOLO, detailed in chapter 6, proposes a unique approach for YOLO-based object detection. The original YOLOv2 divides an input image into equal grids of size 32 pixels by 32 pixels and expects or trains each grid to predict up to five objects. Each predicted object has a vector representation constituting objectness confidence probability, one-hot encoded class prediction probability, and the bounding box parameters: the left-top coordinate pair and the right-bottom coordinate pair. YOLOv2, instead of predicting the bounding box coordinates directly, predicts offsets to a pre-generated dataset representative set of boxes called anchor boxes. There are five anchor boxes in YOLOv2. Since there are five anchor boxes, each grid predicts five offsets, one for each anchor box. This explodes the output layer of YOLOv2 in addition to increasing false positives that slow down the post-processing non-max-suppression. Instead, in DenseYOLO, the anchor box is a trainable parameter. Our model is trained to pick or predict a best-fitting anchor box and its associated offsets from the ground-truth bounding box. This avoided the need to make a redundant prediction of one object using each pre-generated anchor box and, therefore, made DenseYOLO less complex, lighter, and faster during training and inference than the original YOLOv2. Moreover, since we incorporated the idea of fine-grained grid cells from DDGNet, DenseYOLO outperforms the state-of-the-art YOLOv2 in detection accuracy.

The third model we proposed utilizes the more recent and robust YOLO implementation called YOLOv3, authored by the creators of the YOLO model. This latest version includes modern best practices such as multi-scale object detection, skip-connections, and upsampling. Unfortunately, this latest work is deeper and has hundreds of layers, which led to a considerably slower detector against YOLOv2 but immensely more accurate and better performing in detection speed compared to most of the top-performing state-of-the-art object detectors. In our quest for improved detection accuracy, we yet again repurposed the new YOLOv3 by incorporating our unique approach from DenseYOLO and another novel method of annotating object detection ground truths. Our third model is dubbed MultiGridDet, or simply DenseYOLOv2. MultiGridDet's primary feature is assigning multiple nearby grid cells to detect an object of interest, unlike the previous models that lay the responsibility of predicting an object to the grid cell that contains the center coordinate of an object. As a result, MultiGridDet has the potential of a multi-perspective view of an object of interest, increasing its chance of predicting a tightly-fit object bounding box.

Moreover, in addition to the multigrid annotation of an object ground-truth, we also proposed a powerful offline synthetic data generation and augmentation to supplement the object detection dataset. One of the significant challenges in training an object detection model is the lack of a large, well-annotated, class-balanced dataset. The most common approach to treating dataset scarcity is artificial data augmentations such as rotating, flipping, contrast and brightness manipulation, etc. However, these geometrical and texture transformations alone will not address an image's random combination of real-world object occurrence. As a result, many researchers seek ways to generate additional artificial training sets using methods such as copy-paste. We contributed an offline seamless artificial data generation and augmentation technique using copy-paste method. The main features of our data generation are dynamic and seamless stitching of different objects from the dataset, the capability to change the background of generated artificial images, and each object can be an augmented, and dataset class imbalances are addressed. In general, through the help of our data augmentation technique and redundant annotation of object

ground-truth MultiGridDet outperforms YOLOv3 in detection accuracy. Moreover, given the lightweight detection head we incorporated from DenseYOLO, our MultiGridDet implementation is faster and lighter.

Following the series of successful object detection implementations on computer and GPU systems, we focused on the second theme of our study in this thesis work: hardware acceleration of deep convolutional neural network-based object detection. Due to the high resources and energy consumption of computation of current state-of-the-art deep CNN networks, many real-world applications are yet to benefit from the advancement of deep learning. This lag is because many real-world applications have strict requirements such as less memory, less energy consumption, and real-time performances due to the limited resources onboard their embedded platforms. Fortunately, FPGA's high data bandwidth, efficient power consumption, parallelism, and flexible programmability attracted many researchers' attention to address these requirements. We proposed high-throughput energy and resource-efficient acceleration of object detection inference using FPGA to play our part. Finally, our object detection inference acceleration incorporated double-buffering, dynamic fixed-point quantization, loop tiling, loop optimizations, and buffer sharing to achieve a high-throughput, low resource, and energy-efficient inference acceleration. Using Xilinx's Vitis Unified Software Platform, we successfully tested our proposed object detection acceleration by implementing YOLOv2, DDGNet, and DenseYOLO models on at least two FPGA boards.

9.2 Future Works and Perspectives

In this Ph.D. dissertation work, we proposed, designed, and implemented lightweight, fast, and accurate object detection models using a deep convolutional neural network-based and implemented their acceleration using FPGA. Our models' accuracy is commendable and comparable to current state-of-the-art detectors. Furthermore, our hardware acceleration design and implementation showed superior performance against some comparable prior related works measured using metrics such as power efficiency, resource efficiency, and throughput.

Even though we listed several laudable contributions in this thesis work, we still have some prospects for improvement or extension work. Particular future works of our contributions are already mentioned in their corresponding chapters and sections. However, the general prospect is to enable our object detection models to perform object segmentation, both semantic and instance object segmentation. Object segmentation is a valuable area of computer vision where many sensitive real-world applications such as medical institutions could benefit greatly. Moreover, we also intend to extend our study to include object tracking based on deep CNN networks. On the hardware acceleration, in the future, we would like to experiment with extreme quantization and mixed data width quantization for higher speed while maintaining detection accuracy. Moreover, due to time limitations, we did not implement and test our third object detection model, MultiGridDet, on an FPGA board; however, we intend to do so in the future. Last but not least future work is to design a user-friendly and configurable framework that, based on available resources onboard a given FPGA board, maps the software-based deep CNN networks into hardware-accelerated version with high-throughput, efficient resources and power consumption as its core value.

List of Publications

Conference publications

1. Solomon Negussie Tesema and El-Bay Bourennane. "Towards General Purpose Object Detection: Deep Dense Grid Based Object Detection". In: *2020 14th International Conference on Innovations in Information Technology (IIT)*. IEEE. 2020, pp. 227–232

2. Solomon Negussie Tesema and El-Bay Bourennane. "DenseYOLO: Yet Faster, Lighter and More Accurate YOLO". in: *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. 2020, pp. 0534–0539

3. Solomon Negussie Tesema and El-Bay Bourennane. "Multi-Grid Redundant Bounding Box Annotation for Accurate Object Detection". In: *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE. 2021, pp. 145–152

Journal publications

4. Solomon Negussie Tesema and El-Bay Bourennane. "Resource- and Power-Efficient High-Performance Object Detection Inference Acceleration Using FPGA". in: *Electronics* 11.12 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11121827

Bibliography

References for Chapter 1: Introduction

[1] Brien Posey. *Microsoft 'Seeing AI': Imagining a New Use for Computer Vision*. https://redmondmag.com/articles/2021/03/12/microsoft-seeing-ai.aspx. Accessed: 2022-06-10 *Cited on page 1.*

[2] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034 *Cited on page 1.*

[3] Diego Ardila et al. "End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography". In: *Nature medicine* 25.6 (2019), pp. 954–961 *Cited on page 1.*

[4] Anne Dattilo et al. "Identifying exoplanets with deep learning. ii. two new super-earths uncovered by a neural network in k2 data". In: *The Astronomical Journal* 157.5 (2019), p. 169 *Cited on page 1.*

[5] Emmanuel Gbenga Dada et al. "Machine learning for email spam filtering: review, approaches and open research problems". In: *Heliyon* 5.6 (2019), e01802. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2019.e01802 *Cited on page 1.*

[6] Yaniv Taigman et al. "Deepface: Closing the gap to human-level performance in face verification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 1701–1708 *Cited on page 1.*

References for Chapter 2: Overview of Basics of Deep Learning

[7] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110 *Cited on page 8.*

[8] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. Vol. 1. Ieee. 2005, pp. 886–893 *Cited on pages 8, 27.*

[9] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297 *Cited on page 8.*

[10] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006 *Cited on pages 9, 23.*

[11] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015 *Cited on pages 12, 23.*

[12] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 448–456 *Cited on page 18.*

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016 *Cited on page 23.*

References for Chapter 3: Object Detection and Tracking: Study of Current Trends and State-of-the-art approaches

[8] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. Vol. 1. Ieee. 2005, pp. 886–893 *Cited on pages 8, 27.*

[14] Jasper RR Uijlings et al. "Selective search for object recognition". In: *International journal of computer vision* 104.2 (2013), pp. 154–171 *Cited on page 27.*

[15] C Lawrence Zitnick and Piotr Dollár. "Edge boxes: Locating object proposals from edges". In: *European conference on computer vision*. Springer. 2014, pp. 391–405 *Cited on page 27.*

[16] Licheng Jiao et al. "A survey of deep learning-based object detection". In: *IEEE Access* 7 (2019), pp. 128837–128868 *Cited on pages 28, 54.*

[17] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587 *Cited on pages 28, 29, 74, 88.*

[18] Pedro F Felzenszwalb, Ross B Girshick, and David McAllester. "Cascade object detection with deformable part models". In: *2010 IEEE Computer society conference on computer vision and pattern recognition*. IEEE. 2010, pp. 2241–2248 *Cited on page 29.*

[19] Ming-Ming Cheng et al. "BING: Binarized Normed Gradients for Objectness Estimation at 300fps". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014 *Cited on page 29.*

[20] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 *Cited on pages 30, 62, 71, 83.*

[21] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448 *Cited on pages 31, 32.*

[22] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99 *Cited on pages 31, 32, 58, 61, 62, 70, 74, 81, 82, 86, 88.*

[23] Jifeng Dai et al. "R-fcn: Object detection via region-based fully convolutional networks". In: *Advances in neural information processing systems* 29 (2016) *Cited on pages 33, 34.*

[24] Kaiming He et al. "Mask r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969 *Cited on pages 33, 34.*

[25] Pierre Sermanet et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks". In: *arXiv preprint arXiv:1312.6229* (2013) *Cited on pages 35, 54.*

[26] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 *Cited on pages 35, 36, 54, 58, 65, 70, 74, 81, 86, 88.*

[27] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 *Cited on pages 35, 54, 58, 62, 65, 70, 71, 74, 81, 83, 88.*

[28] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018) *Cited on pages 35, 54, 62, 65, 66, 71, 74, 81, 83, 87, 88.*

[29] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37 *Cited on pages 35, 36, 54, 58, 61, 62, 66, 70, 74, 81, 82, 86, 88.*

[30] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988 *Cited on pages 35, 37, 54, 62, 71, 74, 75, 83, 86–88.*

[31] Yizong Cheng. "Mean shift, mode seeking, and clustering". In: *IEEE transactions on pattern analysis and machine intelligence* 17.8 (1995), pp. 790–799 *Cited on page 37.*

[32] Gary R Bradski. "Computer vision face tracking for use in a perceptual user interface". In: (1998) *Cited on page 37.*

[33] Greg Welch, Gary Bishop, et al. "An introduction to the Kalman filter". In: (1995) *Cited on page 38.*

[34] Peter S Maybeck. *Stochastic models, estimation, and control*. Academic press, 1982 *Cited on page 38.*

[35] Kai Diethelm, Neville J Ford, and Alan D Freed. "A predictor-corrector approach for the numerical solution of fractional differential equations". In: *Nonlinear Dynamics* 29.1 (2002), pp. 3–22 *Cited on page 38.*

[36] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. "Simple online and realtime tracking with a deep association metric". In: *2017 IEEE international conference on image processing (ICIP)*. IEEE. 2017, pp. 3645–3649 *Cited on pages 40, 41.*

[37] Guanghan Ning et al. "Spatially supervised recurrent convolutional neural networks for visual object tracking". In: *2017 IEEE international symposium on circuits and systems (ISCAS)*. IEEE. 2017, pp. 1–4 *Cited on page 41.*

References for Chapter 4: FPGA as Hardware Accelerator for Object Detection and Tracking

[38] Eriko Nurvitadhi et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 5–14. ISBN: 9781450343541. DOI: 10.1145/3020078.3021740 *Cited on page 44.*

[39] Michael Mathieu, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through ffts". In: *arXiv preprint arXiv:1312.5851* (2013) *Cited on pages 44, 45.*

[40] Chi Zhang and Viktor Prasanna. "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 35–44 *Cited on pages 44, 45, 87.*

[41] Hanqing Zeng et al. "A framework for generating high throughput CNN implementations on FPGAs". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 117–126 *Cited on pages 44, 87.*

[42] Chun Bao et al. "A power-efficient optimizing framework FPGA accelerator based on winograd for YOLO". In: *IEEE Access* 8 (2020), pp. 94307–94317 *Cited on pages 44, 87.*

[43] Utku Aydonat et al. "An OpenCL(TM) Deep Learning Accelerator on Arria 10". In: *CoRR* abs/1701.03534 (2017). arXiv: 1701.03534 *Cited on pages 44, 87.*

[44] Yap June Wai et al. "Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA". In: *International Journal of Advanced Computer Science and Applications* 9.10 (2018). DOI: 10.14569/IJACSA.2018.091062 *Cited on pages 44, 87.*

[45] Murugan Sankaradas et al. "A Massively Parallel Coprocessor for Convolutional Neural Networks". In: *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. 2009, pp. 53–60. DOI: 10.1109/ASAP.2009.25 *Cited on page 45.*

[46] Clement Farabet et al. "CNP: An FPGA-based processor for Convolutional Networks". In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 32–37. DOI: 10.1109/FPL.2009.5272559 *Cited on page 45.*

[47] Chen Zhang et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks". In: *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 2015, pp. 161–170 *Cited on pages 46, 87, 96.*

[48] Jiantao Qiu et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 26–35. ISBN: 9781450338561. DOI: 10.1145/2847263.2847265 *Cited on page 46.*

[49] Mohammad Motamedi et al. "Design space exploration of FPGA-based Deep Convolutional Neural Networks". In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 575–580. DOI: 10.1109/ASPDAC.2016.7428073 *Cited on page 46.*

References for Chapter 5: Lightweight Generic Object Detector using binary encoding

[16] Licheng Jiao et al. "A survey of deep learning-based object detection". In: *IEEE Access* 7 (2019), pp. 128837–128868 *Cited on pages 28, 54.*

[20] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 *Cited on pages 30, 62, 71, 83.*

[22] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99 *Cited on pages 31, 32, 58, 61, 62, 70, 74, 81, 82, 86, 88.*

[25] Pierre Sermanet et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks". In: *arXiv preprint arXiv:1312.6229* (2013) *Cited on pages 35, 54.*

[26] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 *Cited on pages 35, 36, 54, 58, 65, 70, 74, 81, 86, 88.*

[27] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 *Cited on pages 35, 54, 58, 62, 65, 70, 71, 74, 81, 83, 88.*

[28] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018) *Cited on pages 35, 54, 62, 65, 66, 71, 74, 81, 83, 87, 88.*

[29] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37 *Cited on pages 35, 36, 54, 58, 61, 62, 66, 70, 74, 81, 82, 86, 88.*

[30] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988 *Cited on pages 35, 37, 54, 62, 71, 74, 75, 83, 86–88.*

[50] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755 *Cited on page 55.*

[51] *YOLO: Real-Time Object Detection*. https://pjreddie.com/darknet/yolov2/. Accessed: 2020-08-12 *Cited on page 60.*

[52] Alexander B. Jung et al. *imgaug*. https://github.com/aleju/imgaug. Online; accessed 01-Feb-2020. 2020 *Cited on page 60.*

[53] Tsung-Yi Lin et al. "Feature pyramid networks for object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2117–2125 *Cited on pages 62, 71, 83.*

[54] Jonathan Huang et al. "Speed/accuracy trade-offs for modern convolutional object detectors". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7310–7311 *Cited on pages 62, 66, 71, 83.*

[55] Abhinav Shrivastava et al. "Beyond skip connections: Top-down modulation for object detection". In: *arXiv preprint arXiv:1612.06851* (2016) *Cited on pages 62, 71, 83.*

[56] Cheng-Yang Fu et al. "Dssd: Deconvolutional single shot detector". In: *arXiv preprint arXiv:1701.06659* (2017) *Cited on pages 62, 71, 75, 83.*

References for Chapter 6: Yet Faster, Lighter and More Accurate YOLO

[20] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 *Cited on pages 30, 62, 71, 83.*

[22] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99 *Cited on pages 31, 32, 58, 61, 62, 70, 74, 81, 82, 86, 88.*

[26] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 *Cited on pages 35, 36, 54, 58, 65, 70, 74, 81, 86, 88.*

[27] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 *Cited on pages 35, 54, 58, 62, 65, 70, 71, 74, 81, 83, 88.*

[28] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018) *Cited on pages 35, 54, 62, 65, 66, 71, 74, 81, 83, 87, 88.*

[29] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37 *Cited on pages 35, 36, 54, 58, 61, 62, 66, 70, 74, 81, 82, 86, 88.*

[30] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988 *Cited on pages 35, 37, 54, 62, 71, 74, 75, 83, 86–88.*

[53] Tsung-Yi Lin et al. "Feature pyramid networks for object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2117–2125 *Cited on pages 62, 71, 83.*

[54] Jonathan Huang et al. "Speed/accuracy trade-offs for modern convolutional object detectors". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7310–7311 *Cited on pages 62, 66, 71, 83.*

[55] Abhinav Shrivastava et al. "Beyond skip connections: Top-down modulation for object detection". In: *arXiv preprint arXiv:1612.06851* (2016) *Cited on pages 62, 71, 83.*

[56] Cheng-Yang Fu et al. "Dssd: Deconvolutional single shot detector". In: *arXiv preprint arXiv:1701.06659* (2017) *Cited on pages 62, 71, 75, 83.*

[57] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 *Cited on pages 66, 74.*

[58] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) *Cited on pages 66, 74.*

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 *Cited on pages 66, 86.*

References for Chapter 7: MultiGrid Redundant Bounding Box Annotation for Accurate Object Detection

[17] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587 *Cited on pages 28, 29, 74, 88.*

[20] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 *Cited on pages 30, 62, 71, 83.*

[22] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99 *Cited on pages 31, 32, 58, 61, 62, 70, 74, 81, 82, 86, 88.*

[26] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 *Cited on pages 35, 36, 54, 58, 65, 70, 74, 81, 86, 88.*

[27] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 *Cited on pages 35, 54, 58, 62, 65, 70, 71, 74, 81, 83, 88.*

[28] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018) *Cited on pages 35, 54, 62, 65, 66, 71, 74, 81, 83, 87, 88.*

[29] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37 *Cited on pages 35, 36, 54, 58, 61, 62, 66, 70, 74, 81, 82, 86, 88.*

[30] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988 *Cited on pages 35, 37, 54, 62, 71, 74, 75, 83, 86–88.*

[53] Tsung-Yi Lin et al. "Feature pyramid networks for object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2117–2125 *Cited on pages 62, 71, 83.*

[54] Jonathan Huang et al. "Speed/accuracy trade-offs for modern convolutional object detectors". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7310–7311 *Cited on pages 62, 66, 71, 83.*

[55] Abhinav Shrivastava et al. "Beyond skip connections: Top-down modulation for object detection". In: *arXiv preprint arXiv:1612.06851* (2016) *Cited on pages 62, 71, 83.*

[56] Cheng-Yang Fu et al. "Dssd: Deconvolutional single shot detector". In: *arXiv preprint arXiv:1701.06659* (2017) *Cited on pages 62, 71, 75, 83.*

[57] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 *Cited on pages 66, 74.*

[58] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) *Cited on pages 66, 74.*

[60] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448 *Cited on page 74.*

[61] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "Yolov4: Optimal speed and accuracy of object detection". In: *arXiv preprint arXiv:2004.10934* (2020) *Cited on page 74.*

[62] Solomon Negussie Tesema and El-Bay Bourennane. "DenseYOLO: Yet Faster, Lighter and More Accurate YOLO". In: *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. 2020, pp. 0534–0539 *Cited on pages 74, 75, 83.*

[63] Solomon Negussie Tesema and El-Bay Bourennane. "Towards General Purpose Object Detection: Deep Dense Grid Based Object Detection". In: *2020 14th International Conference on Innovations in Information Technology (IIT)*. IEEE. 2020, pp. 227–232 *Cited on pages 74, 83.*

[64] Kaiwen Duan et al. "Centernet: Keypoint triplets for object detection". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 6569–6578 *Cited on pages 74, 75.*

[65] Hei Law and Jia Deng. "Cornernet: Detecting objects as paired keypoints". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 734–750 *Cited on pages 74, 75.*

[66] Zhi Tian et al. "Fcos: Fully convolutional one-stage object detection". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 9627–9636 *Cited on page 75.*

[67] Lichao Huang et al. "Densebox: Unifying landmark localization with end to end object detection". In: *arXiv preprint arXiv:1509.04874* (2015) *Cited on page 75.*

References for Chapter 8: Resource and Power Efficient High-Performance Object Detection Inference Acceleration Using FPGA

[17] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587 *Cited on pages 28, 29, 74, 88.*

[22] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99 *Cited on pages 31, 32, 58, 61, 62, 70, 74, 81, 82, 86, 88.*

[26] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 *Cited on pages 35, 36, 54, 58, 65, 70, 74, 81, 86, 88.*

[27] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 *Cited on pages 35, 54, 58, 62, 65, 70, 71, 74, 81, 83, 88.*

[28] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018) *Cited on pages 35, 54, 62, 65, 66, 71, 74, 81, 83, 87, 88.*

[29] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37 *Cited on pages 35, 36, 54, 58, 61, 62, 66, 70, 74, 81, 82, 86, 88.*

[30] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988 *Cited on pages 35, 37, 54, 62, 71, 74, 75, 83, 86–88.*

[40] Chi Zhang and Viktor Prasanna. "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 35–44 *Cited on pages 44, 45, 87.*

[41] Hanqing Zeng et al. "A framework for generating high throughput CNN implementations on FPGAs". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 117–126 *Cited on pages 44, 87.*

[42] Chun Bao et al. "A power-efficient optimizing framework FPGA accelerator based on winograd for YOLO". In: *IEEE Access* 8 (2020), pp. 94307–94317 *Cited on pages 44, 87.*

[43] Utku Aydonat et al. "An OpenCL(TM) Deep Learning Accelerator on Arria 10". In: *CoRR* abs/1701.03534 (2017). arXiv: 1701.03534 *Cited on pages 44, 87.*

[44] Yap June Wai et al. "Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA". In: *International Journal of Advanced Computer Science and Applications* 9.10 (2018). DOI: 10.14569/IJACSA.2018.091062 *Cited on pages 44, 87.*

[47] Chen Zhang et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks". In: *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 2015, pp. 161–170 *Cited on pages 46, 87, 96.*

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 *Cited on pages 66, 86.*

[68] Mohammad Rastegari et al. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European conference on computer vision*. Springer. 2016, pp. 525–542 *Cited on pages 86, 87.*

[69] Hiroki Nakahara et al. "A lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on field-programmable gate arrays*. 2018, pp. 31–40 *Cited on page 86.*

[70] Amr Suleiman and Vivienne Sze. "Energy-efficient HOG-based object detection at 1080HD 60 fps with multi-scale support". In: *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2014, pp. 1–6 *Cited on page 86.*

[71] Jos IJzerman et al. "AivoTTA: an energy efficient programmable accelerator for CNN-based object recognition". In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 2018, pp. 28–37 *Cited on page 86.*

[72] Jason Cong and Bingjun Xiao. "Minimizing computation in convolutional neural networks". In: *International conference on artificial neural networks*. Springer. 2014, pp. 281–290 *Cited on pages 87, 89.*

[73] Kamel Abdelouahab et al. "Accelerating CNN inference on FPGAs: A survey". In: *arXiv preprint arXiv:1806.01683* (2018) *Cited on page 87.*

[74] Kai Zeng et al. "FPGA-based accelerator for object detection: a comprehensive survey". In: *The Journal of Supercomputing* (2022), pp. 1–41 *Cited on page 87.*

[75] Yufei Ma et al. "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 45–54 *Cited on page 87.*

[76] Zixiao Wang et al. "Sparse-YOLO: hardware/software co-design of an FPGA accelerator for YOLOv2". In: *IEEE Access* 8 (2020), pp. 116569–116585 *Cited on page 88.*

[77] Yufei Ma et al. "Algorithm-hardware co-design of single shot detector for fast object detection on FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8 *Cited on pages 88, 106–108.*

[78] Solomon Negussie Tesema and El-Bay Bourennane. "Multi-Grid Redundant Bounding Box Annotation for Accurate Object Detection". In: *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE. 2021, pp. 145–152 *Cited on pages 88, 115, 131.*

[79] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. "YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 2503–2510 *Cited on page 88.*

[80] Solomon Negussie Tesema and El-Bay Bourennane. "DenseYOLO: Yet Faster, Lighter and More Accurate YOLO". In: *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. 2020, pp. 0534–0539 *Cited on pages 88, 115, 131.*

[81] Solomon Negussie Tesema and El-Bay Bourennane. "Towards General Purpose Object Detection: Deep Dense Grid Based Object Detection". In: *2020 14th International Conference on Innovations in Information Technology (IIT)*. IEEE. 2020, pp. 227–232 *Cited on pages 88, 115, 131.*

[82] Xuechao Wei et al. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6 *Cited on page 106.*

[83] Duy Thanh Nguyen et al. "A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8 (2019), pp. 1861–1873 *Cited on page 108.*

[84] Shiguang Zhang et al. "An fpga-based reconfigurable cnn accelerator for yolo". In: *2020 IEEE 3rd International Conference on Electronics Technology (ICET)*. IEEE. 2020, pp. 74–78 *Cited on page 108.*

[85] Zhewen Yu and Christos-Savvas Bouganis. "A parameterisable FPGA-tailored architecture for YOLOv3-tiny". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2020, pp. 330–344 *Cited on page 108.*

[86] Shuai Li et al. "A novel FPGA accelerator design for real-time and ultra-low power deep convolutional neural networks compared with titan X GPU". In: *IEEE Access* 8 (2020), pp. 105455–105471 *Cited on page 108.*

References for Chapter 9.2: List of Publications

[78] Solomon Negussie Tesema and El-Bay Bourennane. "Multi-Grid Redundant Bounding Box Annotation for Accurate Object Detection". In: *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE. 2021, pp. 145–152 *Cited on pages 88, 115, 131.*

[80] Solomon Negussie Tesema and El-Bay Bourennane. "DenseYOLO: Yet Faster, Lighter and More Accurate YOLO". In: *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. 2020, pp. 0534–0539 *Cited on pages 88, 115, 131.*

[81] Solomon Negussie Tesema and El-Bay Bourennane. "Towards General Purpose Object Detection: Deep Dense Grid Based Object Detection". In: *2020 14th International Conference on Innovations in Information Technology (IIT)*. IEEE. 2020, pp. 227–232 *Cited on pages 88, 115, 131.*

[87] Solomon Negussie Tesema and El-Bay Bourennane. "Resource- and Power-Efficient High-Performance Object Detection Inference Acceleration Using FPGA". In: *Electronics* 11.12 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11121827 *Cited on page 115.*

Appendix

Synthetic Image Generator(SIG) for Supplementing Object Detection and Segmentation Datasets

Even though research on object detection has immensely improved since the success of deep convolutional neural networks, the success is yet not on par with image classification. One of the challenges in object detection is the lack of a large, carefully annotated, balanced dataset, as one can nowadays easily find for training image classifiers or produce one at ease. However, preparing an object detection dataset is difficult since finding object(s) of interest, putting a bounding box around each object, and labeling them is very cumbersome, tiresome, and error-prone. Especially in generic object detection, preparing object class and scale balanced datasets from scratch is near impossible considering the sheer amount of images required to train an accurate object detector.

Due to this lack of robust and abundant object detection datasets, data augmentation has become integral to training deep learning models, often significantly influencing model performance based on the quality of the chosen augmentation techniques. Some of the standard practices in data augmentation include random resizing or cropping, randomly altering the brightness, contrast, or saturation of an image, rotating an image, and adding noises or dropping out certain parts or pixels of an image. However, the new images produced via these augmentation techniques maintain the relative positions of objects or will not change the set of objects on an image unless an object is entirely cropped out of an image. Hence, even if the augmented image is more bright or darker, noised or rotated from the respective source image, the objects are still the same and in the same viewpoints from one another. Moreover, these geometrical or morphological augmentations will not correct the dataset's class and scale imbalance of object representations.

In addition to the geometrical and morphological online data augmentations we mentioned earlier, copy-paste-based augmentations such as CutMix and alpha-blending-based or more advanced approaches based on generative adversarial networks (GANs) are becoming alternative augmentation techniques. As the name suggests, copy-paste augmentation is simply copying an object from one image and populating it on a new background image. The copy-pasting could be context-aware or random. The challenge with the copy-paste method is that the pasting process leaves visual artifacts that the network quickly picks, misleading the detector into looking for the artifacts rather than the object's core features, hence lacking generalization. To alleviate this, mask-based blending is required, which is another time-consuming process since producing a mask is even more difficult than annotating bounding boxes. The other approaches, such as modeling visual contexts or training networks to produce artificial images, require sophisticated designing and building of a network that trains and creates artificial images suitable for specific purposes.

This work presents our synthetic image generator (SIG) based on copy-paste and OpenCV's seamless clone functionality. We develop a unique and robust algorithm to select objects from

an existing dataset and create a new, well-annotated, photo-realistic supplementary dataset to boost the existing object detection dataset. The OpenCV's seamless clone of images is based on Poisson image editing. Though the cloning leaves few artifacts that could distract the object detection training from focusing on the primary object features, we try to remove or alleviate the effect of the artifacts by using various filterings such as Gaussian Blurring or Median Blurring. We randomly apply contrast sharpening and size augmentation on small objects to boost their resolution and visibility. Moreover, to avoid the redundancy of objects in the dataset, we apply random geometric and morphological augmentation on every object we clone onto a new background.

Below we explain our SIG step-by-step:

Step 1: Prepare Background Images:- We write a script that runs on web browsers such as google chrome to download thousands of images with minimal objects in them. We run our script on Flickr, Pinterest, and Google Images with search keywords such as a landmark, weather, empty city, streets, etc. We download the URLs and the images. We use these images as a background to populate our cloned objects from the object detection dataset.

Step 2: Prepare Annotated Object Detection Dataset:- Our SIG requires an existing object detection annotation, a text file containing a list of all training (and validation) image paths with all bounding box information of all objects on an image. The annotation should be written in Pascal VOC format as shown below:

Image1_path x_{11}, y_{11}, x_{12}, y_{12}, c_1 x_{21}, y_{21}, x_{22}, y_{22}, c_2 x_{31}, y_{31}, x_{32}, y_{32}, c_3 ...
Image2_path x_{11}, y_{11}, x_{12}, y_{12}, c_1 ...
Image2_path x_{11}, y_{11}, x_{12}, y_{12}, c_1 x_{21}, y_{21}, x_{22}, y_{22}, c_2 ...
...

Each line in the annotation represents an image and the bounding boxes of all objects on the image. x_{ij} or y_{ij} corresponds to the top-left and bottom-right coordinates of each object on the image, as shown in Figure 1. The c_i in the annotation stands for the object's class.

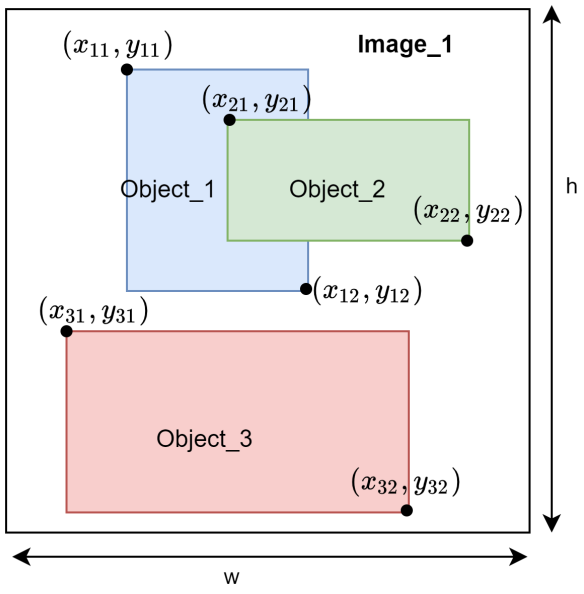


Figure 1: *Illustration of Bounding Box Ground-Truth Annotation*

In addition to the training images, one should also prepare object masks, though not

mandatory but will significantly impact the quality of the seamless clone. In our case, we used the Pascal VOC and COCO datasets and the segmentation masks provided within the datasets.

Step 3: Populating the objects:- Once we prepare the background images and some preexisting object detection datasets (e.g., Pascal VOC or COCO datasets) with segmentation masks, we follow a step-by-step procedure to randomly select objects from ground truth and clone them onto a new background to create a new seamless and photo-realistic dataset. This step-by-step flow is best explained using a flowchart as shown in figure 2. As seen on the flowchart, selecting objects, followed by additional augmentations and placement on to background image, involves several stages. To explain the flowchart briefly:

1. We randomly pick objects from an image until the cumulative sum of the IoU of the selected objects can at least cover the entirety of the background image. We calculate the IoU of the objects against the background image by assuming all objects top left corner to be pasted at the index (0,0) of the image. If the total IoU of the selected objects is greater or equal to 1.0, then that guarantees that the selected objects, if pasted on the background in some order, will cover the area of the object. That is not a requirement but is a means to control how many objects to select and not to underutilize the background image area. If the IoU is less than 1.0, then we repeat this step with a different image so that the newly created image has objects mixed from various images. Note that we also apply random augmentations from the standard image processing techniques to ensure the robustness of our dataset.
2. After picking a certain number of objects with overall IoU $>= 1.0$, we apply a fairly complex algorithm to determine paste coordinates for the selected objects. The algorithm ensures that no objects overlap or at least not significantly, and the background image space is utilized efficiently.
3. Finally, after populating the objects onto the background image, we apply random histogram equalization techniques and gamma correction to smooth out the brightness and contrast of the newly created image. At last, we save our new image and the bounding box coordinates and object classes and repeat the process all over again to generate as many new images as possible.

Figure 3 shows sample outputs of our synthetic image generator or SIG for short. The figure shows the newly generated images on the left and the bounding boxes drawn on each object on the right for better visibility and to showcase the quality of our generator. The flowchart shown in Figure 2 is simplistic as it leaves many little but vital details for brevity. For example, when cloning an object, e.g., a sofa, a bus, or a horse being ridden by a person, etc., it is evident that there will be part of objects that will be brought with these objects and cloned on the new image. Thus, if a significant portion of inner objects is cloned(copied) with other objects, we must also annotate the coordinates of these piggybacked objects as well, given that these inner objects also are our objects of interest. Figure 3 also shows an example of cases where such inner objects are annotated with a bigger and outer object around them.

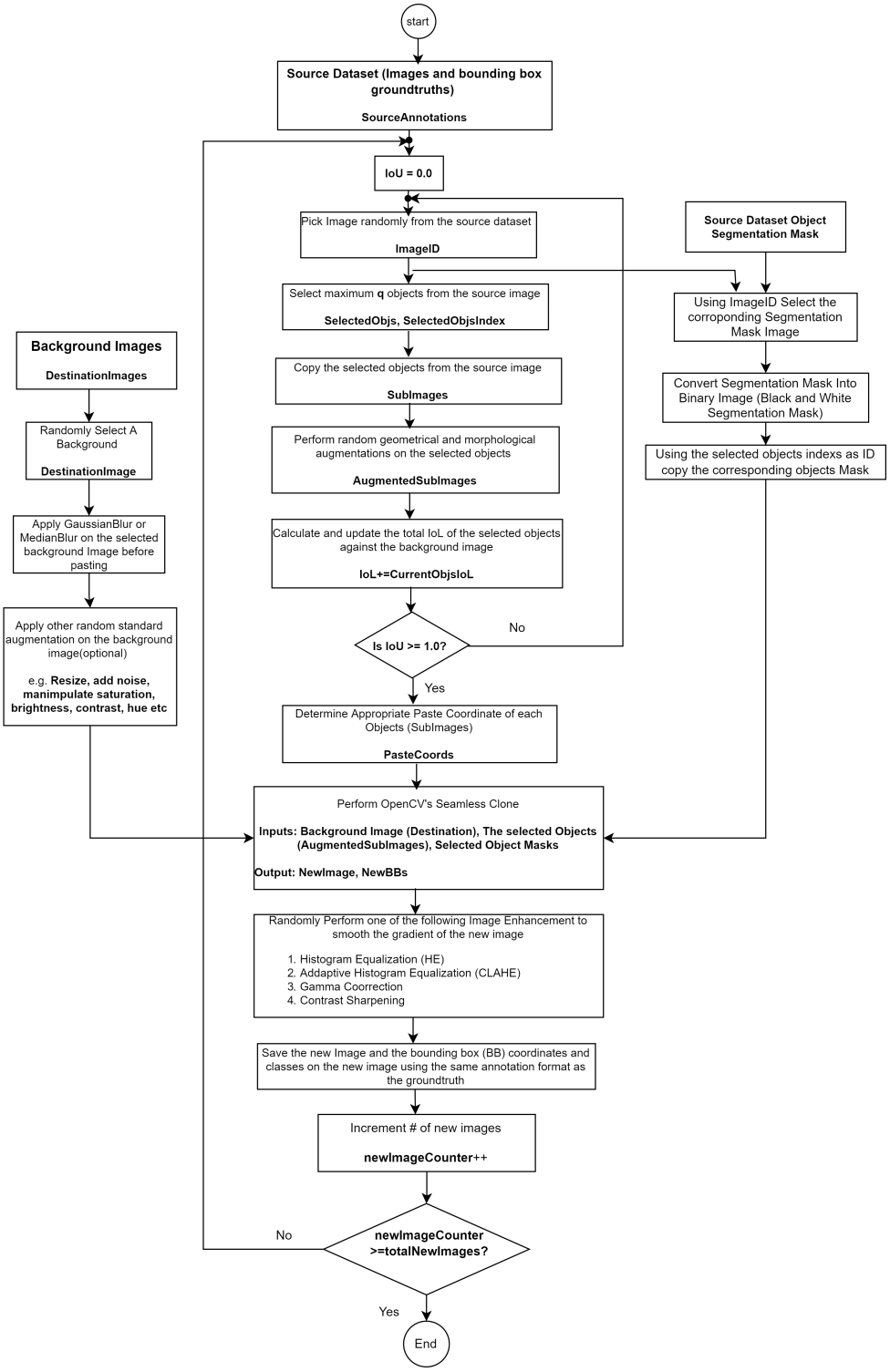


Figure 2: *Minimal flowchart showing the overall process flow of our synthetic image generator (SIG). Note that SIG is an offline data augmentation, meaning the augmented or artificial images are generated before training is started not during training. Using SIG we can generate unlimited unique images to supplement an existing object detection dataset with seamlessly generated photo-realistic new images. We have used SIG in training MultiGridDet, explained in Chapter 7 briefly. There is slight change, though than the version we presented in chapter 7. The version presented in chapter 7 does not use object mask and hence relatively leaves visible edges on an image since it is a simple copy-paste. However, the version presented in this chapter uses seamless cloning using Poisson Image Editing and Various gradient and Histogram Equalization based image processing. As a result current SIG version is more powerful and generates more seamless images at extremely high speed.*

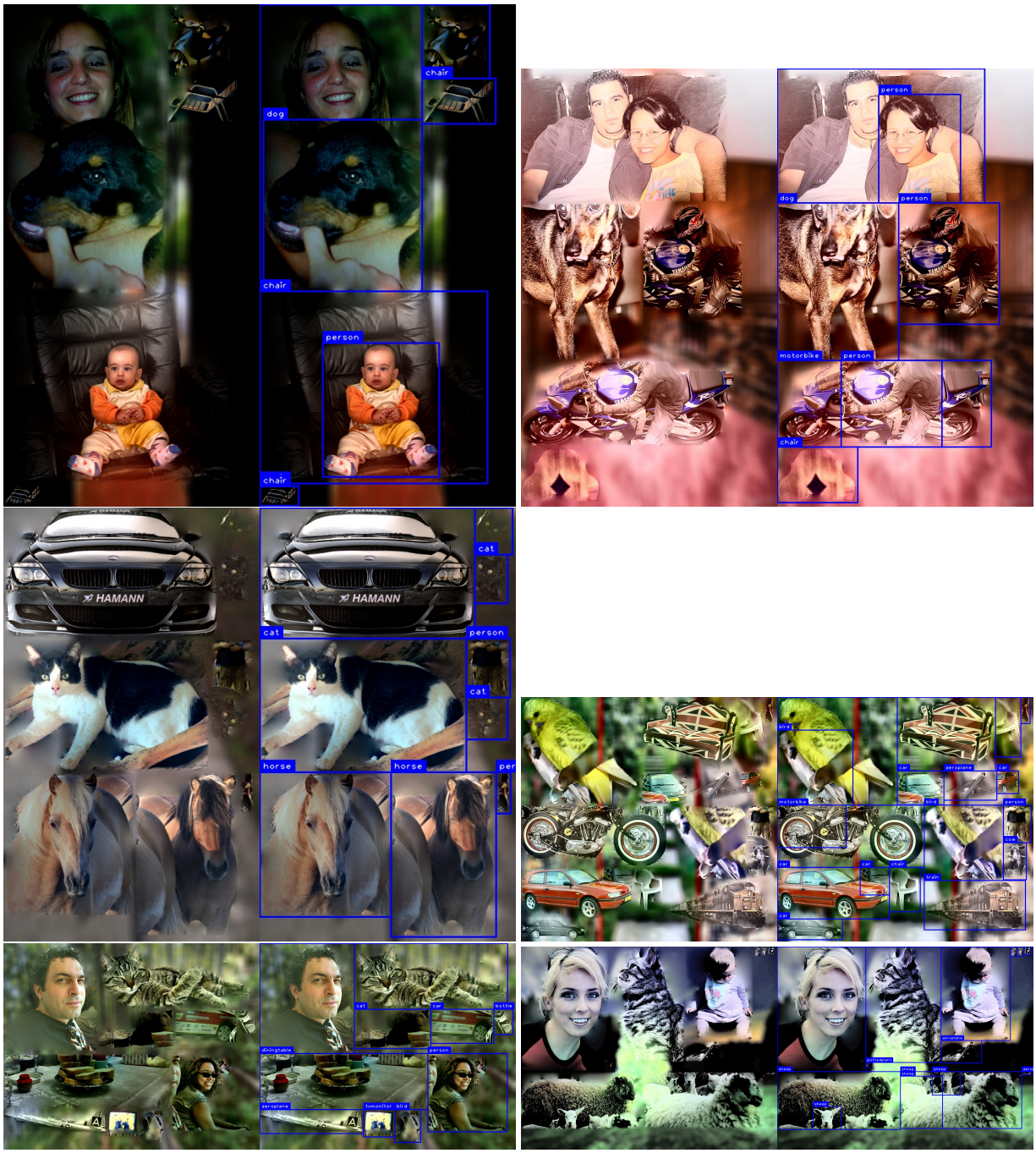


Figure 3: *Sample synthetically generated images using our SIG Algorithm. Note that, each image depicted here are duplicates , one without bounding boxes on the left and the other pair with bounding boxes drawn around each objects on the right. And also note how no objects overlap and each output image have randomly varying sizes.*

Appendix

Analysis of Our Inference Acceleration Implementation on DDGNet, DenseYOLO and YOLOv2

In chapter 8, we explained our inference accelerator by implementing YOLOv2 object detection acceleration and published our result in the journal of Electronics [78]. However, even though we presented our result on YOLOv2 we also tested our inference implementation on DDGNet [81] and DenseYOLO [80], our own object detection models presented in Chapter 5 and Chapter 6, respectively. The Vivado interface view of our implementation on ZCU102 is depicted in Figure 4. The figure shows the AXI-DMA interface view between the processor system and custom accelerator labeled $FPGA_Acc_0$ to say FPGA accelerator and the DDR memory interface through AXI smart interconnect.

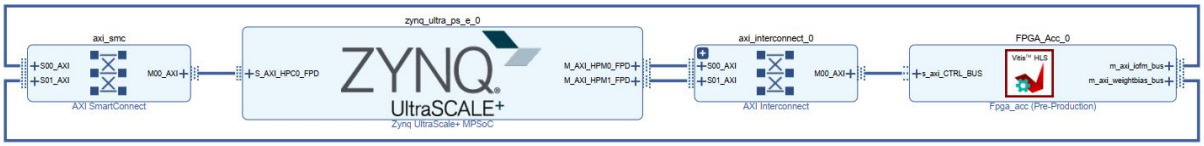


Figure 4: *Vivado Interface View of our inference accelerator.*

Next, we present the details of our comparison of inference acceleration tested with all three models mentioned earlier. Table 1 presents the parameters of the three models, and as such, the three models have the same backbone but different detection heads and, as a result, different post-processing. Moreover, Table 2 shows the comparison of throughput or latency, GOPS (Giga Operations Per Seconds), and DSP Efficiency of the three models on ZYNQ 7020 and ZCU 102. As explained in Chapter 8 we calculate latency and GOP using Equation 8.18 and 8.19 respectively. Accordingly, as seen from the result in Table 2, our two models, DDGNet and DenseYOLO are slightly faster and have lighter output detection heads. DDGNet and DenseYOLO give an output prediction in 229 milliseconds, whereas it takes 244 milliseconds for YOLOv2. However, these delays are before the post-processing, that is, before the non-max-suppression and thresholding of the output layer. Given our models' lighter output heads, further speed gains are an obvious expectation.

Table 1: *YOLOv2 vs DDGNet vs DenseYOLO models layer parameter comparisons and tile read or write access cycles. As explained in Chapter 8 the tile sizes for acceleration implementation on ZYNQ 7020 and ZCU102 boards are different. See chapter 8 Table 8.4 for details of the tile size choices of each boards. As seen in this table the three models have similar backbone network,Darknet-19, and different detection head. As a result, the inference implementation discussed in chapter 8 fully applies to all three models.*

												ZYNQ 7020			ZCU102		
	Layer Type	Layers	N_{ix}	N_{iy}	N_{if}	N_{kx}	N_{ky}	N_{ox}	N_{oy}	N_{of}	$\frac{N_{if}}{T_{if}}$	$\frac{N_{ox}}{T_{ox}}$	$\frac{N_{oy}}{T_{oy}}$	$\frac{N_{of}}{T_{of}}$	$\frac{N_{ox}}{T_{ox}}$	$\frac{N_{oy}}{T_{oy}}$	$\frac{N_{of}}{T_{of}}$
	Conv	0	416	416	3	3	3	416	416	32	1	16	16	8	8	1	1
Darknet-19 Back Bone	Max	1	416	416	32	2	2	208	208	32	8	8	8	4	4	1	1
	Conv	2	208	208	32	3	3	208	208	64	8	8	8	4	4	2	1
	Max	3	208	208	64	2	2	104	104	64	16	4	4	2	2	2	1
	Conv	4	104	104	64	3	3	104	104	128	16	4	4	2	2	4	2
	Conv	5	104	104	128	1	1	104	104	64	32	4	4	2	2	2	1
	Conv	6	104	104	64	3	3	104	104	128	16	4	4	2	2	4	2
	Max	7	104	104	128	2	2	52	52	128	32	2	2	1	1	4	2
	Conv	8	52	52	128	3	3	52	52	256	32	2	2	1	1	8	4
	Conv	9	52	52	256	1	1	52	52	128	64	2	2	1	1	4	2
	Conv	10	52	52	128	3	3	52	52	256	32	2	2	1	1	8	4
	Max	11	52	52	256	2	2	26	26	256	64	1	1	1	1	8	4
	Conv	12	26	26	256	3	3	26	26	512	64	1	1	1	1	16	8
	Conv	13	26	26	512	1	1	26	26	256	128	1	1	1	1	8	4
	Conv	14	26	26	256	3	3	26	26	512	64	1	1	1	1	16	8
	Conv	15	26	26	512	1	1	26	26	256	128	1	1	1	1	8	4
	Conv	16	26	26	256	3	3	26	26	512	64	1	1	1	1	16	8
	Max	17	26	26	512	2	2	13	13	512	128	1	1	1	1	16	8
	Conv	18	13	13	512	3	3	13	13	1024	128	1	1	1	1	32	16
	Conv	19	13	13	1024	1	1	13	13	512	256	1	1	1	1	16	8
	Conv	20	13	13	512	3	3	13	13	1024	128	1	1	1	1	32	16
	Conv	21	13	13	1024	1	1	13	13	512	256	1	1	1	1	16	8
	Conv	22	13	13	512	3	3	13	13	1024	128	1	1	1	1	32	16
	Conv	23	13	13	1024	3	3	13	13	1024	256	1	1	1	1	32	16
	Conv	24	13	13	1024	3	3	13	13	1024	256	1	1	1	1	32	16
YOLOv2 Detection Head	Route	25	Layer 16 to Layer 25								0	0	0	0	0	0	0
	Conv	26	26	26	512	1	1	26	26	64	128	1	1	1	1	2	1
	Reorg	27	26	26	64			13	13	256	16	1	1	1	1	8	4
	Concat	28						13	13	1280	0	1	1	1	1	40	20
	Conv	29	13	13	1280	3	3	13	13	1024	320	1	1	1	1	32	16
	Conv	30	13	13	1024	1	1	13	13	425	256	1	1	1	1	14	7
	head	31	13	13	**425**		post processing				0	0	0	0	0	0	0
DDGNet Detection Head	Reorg	25	13	13	1024			52	52	64	256	2	2	1	1	2	1
	Route	26	Layer 16 to Layer 26								0	0	0	0	0	0	0
	Conv	27	26	26	512	1	1	26	26	256	128	1	1	1	1	8	4
	Reorg	28	26	26	256			52	52	64	64	2	2	1	1	2	1
	Concat	29						52	52	128	0	2	2	1	1	4	2
	Conv	30	52	52	128	3	3	52	52	256	32	2	2	1	1	8	4
	Conv	31	52	52	256	1	1	52	52	95	64	2	2	1	1	3	2
	head	32	52	52	**95**		post processing				0	0	0	0	0	0	0
DenseYOLO Detection Head	Reorg	25	13	13	1024			52	52	64	256	2	2	1	1	2	1
	Route	26	Layer 16 to Layer 26								0	0	0	0	0	0	0
	Conv	27	26	26	512	1	1	26	26	256	128	1	1	1	1	8	4
	Reorg	28	26	26	256			52	52	64	64	2	2	1	1	2	1
	Concat	29						52	52	128	0	2	2	1	1	4	2
	Conv	30	52	52	128	3	3	52	52	256	32	2	2	1	1	8	4
	Conv	31	52	52	256	1	1	52	52	90	64	2	2	1	1	3	2
	head	32	52	52	**90**		post processing				0	0	0	0	0	0	0

Table 2: *YOLOv2 vs DDGNET vs DenseYOLO Latency, GOPS and DSP efficiency Comparison.*

	Layer Type	Layers	Execute Latency ZCU102	Execute Latency ZYNQ 7020	GOP ZCU102	GOP ZYNQ 7020	DSP Efficiency ZCU102	DSP Efficiency ZYNQ 7020
Darknet-19 Back Bone	Conv	0	5.19	10.39	1.20	0.60	0.14	0.56
	Max	1	25.36	50.79	0.00	0.00	1	1
	Conv	2	10.38	41.55	2.39	2.39	1	1
	Max	3	12.68	25.38	0.00	0.00	1	1
	Conv	4	10.38	41.55	2.39	2.39	1	1
	Conv	5	1.15	4.62	0.27	0.27	1	1
	Conv	6	10.38	41.55	2.39	2.39	1	1
	Max	7	6.34	12.69	0.00	0.00	1	1
	Conv	8	10.38	41.55	2.39	2.39	1	1
	Conv	9	1.15	4.62	0.27	0.27	1	1
	Conv	10	10.38	41.55	2.39	2.39	1	1
	Max	11	3.17	6.35	0.00	0.00	1	1
	Conv	12	10.38	41.55	2.39	2.39	1	1
	Conv	13	1.15	4.62	0.27	0.27	1	1
	Conv	14	10.38	41.55	2.39	2.39	1	1
	Conv	15	1.15	4.62	0.27	0.27	1	1
	Conv	16	10.38	41.55	2.39	2.39	1	1
	Max	17	1.58	3.17	0.00	0.00	1	1
	Conv	18	10.38	41.55	2.39	2.39	1	1
	Conv	19	1.15	4.62	0.27	0.27	1	1
	Conv	20	10.38	41.55	2.39	2.39	1	1
	Conv	21	1.15	4.62	0.27	0.27	1	1
	Conv	22	10.38	41.55	2.39	2.39	1	1
	Conv	23	20.76	83.11	4.78	4.78	1	1
	Conv	24	20.76	83.11	4.78	4.78	1	1
YOLOv2 Detection Head	Route	25	0.00	0.00	0.00	0.00	1	1
	Conv	26	0.29	1.15	0.07	0.07	1	1
	Reorg	27	0.00	0.00	0.00	0.00	1	1
	Concat	28	0.00	0.00	0.00	0.00	1	1
	Conv	29	25.96	103.89	5.98	5.98	1	1
	Conv	30	1.01	4.04	0.23	0.23	0.95	0.95
	head	31	0.00	0.00	0.00	0.00	1	1
	Total GOP/S		244.23	868.87	44.96 184.10	44.37 51.06	0.97	0.98
DDGNet Detection Head	Reorg	25	0	0	0	0	1	1
	Route	26	0	0	0	0	1	1
	Conv	27	1.15	4.62	0.27	0.27	1	1
	Reorg	28	0	0	0	0	1	1
	Concat	29	0	0	0	0	1	1
	Conv	30	10.38	41.55	2.39	2.39	1.00	1.00
	Conv	31	1.15	3.46	0.27	0.20	0.74	0.99
	head	31	0.00	0.00	0.00	0.00	1.00	1.00
	Total GOP/S		229.67	809.42	41.61 181.17	40.94 50.58	0.97	0.99
DenseYOLO Detection Head	Reorg	25	0	0	0	0	1	1
	Route	26	0	0	0	0	1	1
	Conv	27	1.15	4.62	0.27	0.27	1	1
	Reorg	28	0	0	0	0	1	1
	Concat	29	0	0	0	0	1	1
	Conv	30	10.38	41.55	2.39	2.39	1	1
	Conv	31	1.15	3.46	0.27	0.20	0.70	0.94
	head	31	0	0	0	0	1	1
	Total GOP/S		229.67	809.42	41.61 181.17	40.94 50.58	0.96	0.99