

UNIVERSIDADE FEDERAL DO PARANÁ

ANDERSON DE LIMA LUIZ

DIGITAL SYNTHESIS OF R-CNN ELEMENTS FOR AUTOMATIC BLURRING OF
LICENCE PLATES AND FACES FOR THE EUROPEAN GDPR 2016/679
COMPLIANCE

INGOLSTADT

2022

ANDERSON DE LIMA LUIZ

DIGITAL SYNTHESIS OF R-CNN ELEMENTS FOR AUTOMATIC BLURRING OF
LICENCE PLATES AND FACES FOR THE EUROPEAN GDPR 2016/679
COMPLIANCE

Monografia apresentada ao Curso de Pós-Graduação em engenharia elétrica, Setor de Tecnologia, da Universidade Federal do Paraná, como requisito parcial à obtenção do título de Mestre em engenharia elétrica.

Orientador: Prof. Dr. Luis Henrique Assumpção Lolis

Coorientador: Prof. Dr. Alessandro Zimmer

INGOLSTADT

2022

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Luiz, Anderson de Lima.

Digital synthesis of R-CNN elements for automatic blurring of license plates and faces for the European GDPR 2016/679 compliance. / Anderson de Lima Luiz. – Curitiba, 2022.

1 recurso on-line : PDF.

Dissertação (Mestrado) – Universidade Federal do Paraná, Setor de Tecnologia, Programa de Pós-Graduação em Engenharia Elétrica e Technische Hochschule Ingolstadt.

Orientador: Prof. Dr. Luis Henrique Assumpção Lolis.

1. Engenharia elétrica. 2. Regulamentação. 3. Proteção de dados. 4. Circuitos integrados digitais. I. Lolis, Luis Henrique Assumpção. II. Universidade Federal do Paraná. Programa de Pós-Graduação em Meio Ambiente Urbano e Industrial. IV. Technische Hochschule Ingolstadt. V. Título.

Bibliotecário: Nilson Carlos Vieira Júnior CRB-9/1797



MINISTÉRIO DA EDUCAÇÃO
SETOR DE TECNOLOGIA
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO ENGENHARIA
ELÉTRICA - 40001016043P4

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação ENGENHARIA ELÉTRICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **ANDERSON DE LIMA LUIZ** intitulada: **Digital synthesis of R-CNN elements for automatic blurring of license plates and faces for the European GDPR 2016/679 compliance.**, sob orientação do Prof. Dr. LUIS HENRIQUE ASSUMPÇÃO LOLIS, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 30 de Junho de 2022.

Assinatura Eletrônica

08/07/2022 14:21:06.0

LUIS HENRIQUE ASSUMPÇÃO LOLIS

Presidente da Banca Examinadora

Assinatura Eletrônica

01/07/2022 17:34:34.0

GIDEON VILLAR LEANDRO

Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica

01/07/2022 17:34:24.0

ROBERTO ZANETTI FREIRE

Avaliador Externo (PONTIFICA UNIVERSIDADE CATÓLICA DO PARANÁ)

RESUMO

Em 25 de maio de 2018 – O Regulamento Geral de Proteção de Dados (EU) 2016/679 foi implementado a fim de assegurar o processo de tratamento dos dados dos cidadãos da EU, removendo a divulgação não autorizada de dados pessoais sem o consentimento do referido indivíduo para uso geral. Tecnologias modernas de visão computacional podem desfocar com sucesso as imagens que expõem faces ou placas de veículos usando Deep Learning Convolutional Neural Networks (DLCNNs) através de GPUs. Entretanto, devido à natureza genérica do processador GPU, o roteamento digital não é otimizado para processar eficientemente usos específicos da aplicação, tais como o algoritmo CNN. O objetivo desta pesquisa é criar um circuito integrado digital usando uma Field Programmable Gate Array (FPGA) implementando funções estratégicas de partes sintetizáveis dos algoritmos You Only Look Once (YOLO) v4, OpenPose, e PoseNet. É implementada a otimização de consumo de recursos de Convolutional Neural Networks (CNNs) já existentes para GPUs usando a estrutura de FPGAs. As comparações são realizadas considerando taxa de quadros, latência e consumo de energia para os seguintes dispositivos de processamento: a Unidade Central de Processamento (CPU) Intel® Core™ i7-10750H, GeForce® Giga Texel Shader eXtreme (GTX) 1660 Ti, e GeForce® Ray Tracing Texel eXtreme (RTX) 2080 Graphics Processing Unit (GPU) e uma FPGA ZCU104-XCZU7EV-2FFVC1156 MPSoC. Os diferentes modelos de redes neurais são implementados desde flutuantes até um ponto fixo nas plataformas citadas. As CNNs de detecção de objetos são avaliadas utilizando a métrica de precisão média (mAP). Algoritmos de keypoint detection são avaliados usando-se a similaridade de pontos-chave de objeto (OKS). Foi desenvolvido um algoritmo de poda de rede que pode, com uma taxa de compressão de 50%, melhorar a taxa de quadros do algoritmo YOLO V4 de 4 FPS (CPU) e 45 FPS (GPU) para 179 FPS na FPGA enquanto reduz o consumo de energia em 50,8% (1,192 mJ/Frame) em comparação com a CPU testada (2,423 mJ/Frame) e uma média de 32,4% (1,763 mJ/Frame) em comparação com a GPU com uma ligeira perda de precisão. Usando as métricas acima mencionadas e precisão normalizada, a PoseNet obteve uma performance média de 300 FPS, contra 37 FPS na CPU e 128 FPS na GPU, obtendo respectivamente 61,1% e 51,9% (0,429 mJ/Frame) de redução do consumo de energia relacionado às implementações da CPU (1,104 mJ/Frame) e GPU (0,872 mJ/Frame), respectivamente. O trabalho desenvolvido prova que a implementação das CNNs testadas nos dispositivos FPGA através do projeto de um circuito lógico personalizado pode aumentar significativamente o desempenho em termos de ocupação de superfície, velocidade e consumo de energia; e reduzir os custos no desenvolvimento de aplicações.

Palavras-chave: Aceleração FPGA. DLCNN. Detecção de objetos. Detecção de keypoints.

ABSTRACT

On May 25th, 2018 – The General Data Protection Regulation (EU) 2016/679 was implemented in order to secure the handling process of EU citizens' data by removing unauthorized disclosure of personal data without the referred individual's consent for general use. Modern computer vision technology can successfully blur images exposing faces or vehicle plates using deep learning convolutional neural networks (DLCNNs) via Graphics Processing Units (GPUs). However, due to the generic nature of the GPU processor, the digital routing is not optimized to efficiently process-specific application uses, such as the CNN algorithm. The goal of this research is to create a digital integrated circuit using a Field-programmable gate array (FPGA) implementing strategic functions of synthesizable parts of the You Only Look Once (YOLO) v4, OpenPose, and PoseNet algorithms. Optimization of the already extant GPU implemented Convolutional Neural Networks (CNNs) in terms of resource consumption using the FPGA device structure is implemented. Comparisons are carried out considering frame rate, latency, and energy consumption, for the following processing devices: the Intel® Core™ i7-10750H Processor Central Processing Unit (CPU), GeForce® Giga Texel Shader eXtreme (GTX) 1660 Ti, and GeForce® Ray Tracing Texel eXtreme (RTX) 2080 Graphics Processing Units (GPUs) and a ZCU104-XCZU7EV-2FFVC1156 MPSoC FPGA. The different neural network models are implemented from floating to a fixed point onto the cited platforms. Object detection CNNs are evaluated using the mean Average Precision (mAP) metric. Pose algorithms are evaluated using Object Keypoint Similarity (OKS). A network pruning algorithm has been developed that can, with a 50% compression rate, improve the frame rate of the YOLO V4 algorithm from 4 FPS (CPU) and 45 FPS (GPU) to 179 FPS on the FPGA while reducing the energy consumption by 50.8% (1.192 mJ/Frame) compared to the tested CPU (2.423 mJ/Frame) and a 32.4% (1.763 mJ/Frame) means compared to the GPU with a slight accuracy loss. Using the aforementioned metrics and normalized accuracy, PoseNet has obtained a mean performance of 300 FPS, against 37 FPS on the CPU and 128 FPS on the GPU, and obtaining respectively 61.1% and 51.9% (0.429 mJ/Frame) energy consumption reductions related to the CPU (1.104 mJ/Frame) and GPU (0.872 mJ/Frame) implementations, respectively. The developed work proves that implementing the tested CNNs on FPGA devices by designing a customized logic circuit can significantly increase performance in terms of surface occupation, speed, and energy consumption; and reduce overall costs in project development.

Keywords: FPGA acceleration. DLCNN. Object detection. Keypoint detection.

ABSTRAKT

Am 25. Mai 2018 wurde die Allgemeine Datenschutzverordnung (EU) 2016/679 eingeführt, um den Umgang mit den Daten von EU-Bürgern zu sichern, indem die unbefugte Weitergabe personenbezogener Daten ohne die Zustimmung der betreffenden Person für die allgemeine Verwendung verhindert wird. Moderne Computer-Vision-Technologien können mit Hilfe von Deep Learning Convolutional Neural Networks (DL-CNNs) über Graphics Processing Units (GPUs) erfolgreich Bilder unscharf machen, auf denen Gesichter oder Fahrzeugkennzeichen zu erkennen sind. Aufgrund des generischen Charakters des GPU-Prozessors ist das digitale Routing jedoch nicht für die effiziente Verarbeitung spezifischer Anwendungen, wie z. B. des CNN-Algorithmus, optimiert. Das Ziel dieser Forschungsarbeit ist es, eine digitale integrierte Schaltung mit Hilfe eines Field-programmable Gate Array (FPGA) zu entwickeln, die strategische Funktionen der synthetisierbaren Teile der Algorithmen You Only Look Once (YOLO) v4, OpenPose und PoseNet implementiert. Es wird eine Optimierung der bereits existierenden GPU-implementierten Convolutional Neural Networks (CNNs) in Bezug auf den Ressourcenverbrauch unter Verwendung der FPGA-Bausteinstruktur durchgeführt. Es werden Vergleiche hinsichtlich der Bildrate, der Latenz und des Energieverbrauchs für die folgenden Verarbeitungsgeräte durchgeführt: die Intel® Core™ i7-10750H Prozessor Central Processing Unit (CPU), GeForce® Giga Texel Shader eXtreme (GTX) 1660 Ti und GeForce® Ray Tracing Texel eXtreme (RTX) 2080 Grafikverarbeitungseinheiten (GPUs) und ein ZCU104-XCZU7EV-2FFVC1156 MPSoC FPGA. Die verschiedenen neuronalen Netzwerkmodelle werden auf den genannten Plattformen von der Fließkomma- zur Festkommaberechnung implementiert. Die CNNs zur Objekterkennung werden mit der Metrik der mittleren durchschnittlichen Präzision (mAP) bewertet. Die Pose-Algorithmen werden anhand der Object Keypoint Similarity (OKS) bewertet. Es wurde ein Netzwerk-Beschneidungsalgorithmus entwickelt, der mit einer Kompressionsrate von 50% die Bildrate des YOLO V4 Algorithmus von 4 FPS (CPU) und 45 FPS (GPU) auf 179 FPS auf dem FPGA verbessern kann, während der Energieverbrauch um 50,8% (1,192 mJ/Frame) im Vergleich zur getesteten CPU (2,423 mJ/Frame) und um 32,4% (1,763 mJ/Frame) im Vergleich zur GPU mit einem leichten Genauigkeitsverlust reduziert wird. Unter Verwendung der oben genannten Metriken und der normalisierten Genauigkeit hat PoseNet eine durchschnittliche Leistung von 300 FPS erreicht, gegenüber 37 FPS auf der CPU und 128 FPS auf der GPU, und eine Senkung des Energieverbrauchs um 61,1% bzw. 51,9% (0,429 mJ/Frame) in Bezug auf die CPU- (1,104 mJ/Frame) und GPU-Implementierungen (0,872 mJ/Frame) erzielt. Die entwickelte Arbeit beweist, dass die Implementierung der getesteten CNNs auf FPGA-Bausteinen durch den Entwurf einer angepassten Logikschaltung die Leistung in Bezug auf Flächenbelegung, Geschwindigkeit und Energieverbrauch erheblich steigern und die Kosten senken kann.

Keywords: FPGA-Beschleunigung. DLCNN. Objekterkennung. Keypoint Erkennung.

List of Figures

1	Different techniques used for license plate detection.	21
2	Energy consumption per frame of the main three data processing units for different image kernel operations – [1]	22
3	Demonstration of the YOLOv4 algorithm detecting different individual details of a license plate alongside a license plate for multiple objects – [2].	24
4	Representation of the acceleration function.	25
5	Structure of the FPGA high level language interpreter.	26
6	Representation of the ReLU functions for CNNs.	38
7	Layers of the Darknet-53 structure – [3].	40
8	Feature structure for a 608x608 pixel input – [4].	42
9	PoseNet modules – [5].	46
10	Architecture of the OpenPose CNN.	49
11	Block diagram used to represent the data flow for [6].	52
12	License plate detection using the YOLO algorithm – [7].	53
13	Workflow used for the development of the project.	54
14	General structure of each stage.	55
15	Characteristics of the first stage.	56
16	Datasets used for training each of the studied algorithms in the first stage.	56
17	Differences of the synthesized model for different frameworks.	57
18	Overview of the optimization stage.	60
19	Relation between accuracy and model weight reduction (compression) – [8]	60
20	Overview of the quantization stage.	62
21	Demonstration of the initial 32 float variable used to calculate variables up to the optimization step.	63
22	Example of the word length sweep on the uniform test for the word length. In the example diagram, the green numbers are the ones that have already been tested and found relevant to assert accuracy above the established threshold level. Blue characters are the ones that haven't been tested in the current iteration.	63
23	Example of the word length sweep on the random test for the word length. In the example diagram, the green numbers are the ones that have already been tested and found relevant to assert accuracy above the established threshold level. Blue characters are the ones that haven't been tested in the current iteration.	64
24	Overview of the compilation stage.	64
25	Hardware validation structure workflow.	66
26	Steps executed inside the HLS interface to emulate and deploy the code.	67
27	XIR workflow.	69
28	Compilation stage workflow.	71
29	Detail representation of the RTL schematic for one of the convolution registers.	72
30	Hardware interface structure defined for the object detection algorithm.	78

31	Logic circuit utilization without implementation. Each section is divided for specific functions such as convolution, pooling, sum and main OS interface – [9].	79
32	Representation of the logic elements inside the FPGA device with a group of logic gates used for convolution.	79
33	Representation of the routing targets for a convolution register.	80
34	Representation of different parts of an image filter in [9] after implementation (routing and positioning).	80
35	GUI used for power consumption estimation over various iterations	82
36	Parameters of the CPU used for testing.	83
37	Parameters of the GPU used for testing.	84
38	ZYNQ ZCU104 Ultrascale + MPSoC board used for testing.	85
39	Plate detection seen by drawing a bounding box around the AI generated coordinates in a 608x608 pixel image.	86
40	Accuracy decay and frame rate enhancement for different compression rates of the YOLO v3 algorithm.	87
41	Performance comparison among different platforms for the YOLO v3 algorithm (optimisation normalized to prevent accuracy loss).	88
42	Accuracy decay and frame rate enhancement for different compression rates of the YOLO v4 algorithm.	90
43	Performance comparison among different platforms for the YOLO v4 algorithm (optimisation normalized to prevent accuracy loss).	91
44	Demonstration of the target 14 keypoint detectors on a human body (HumanEval dataset).	93
45	Accuracy decay and frame rate enhancement for different compression rates of the PoseNet algorithm.	94
46	Performance comparison among different platforms for the PoseNet algorithm (optimisation normalized to prevent accuracy loss).	95
47	Demonstration of OpenPose 24-keypoint detection.	96
48	OKS and frame rate of the OpenPose algorithm for different compression rate.	97
49	Performance comparison among different platforms for the OpenPose algorithm (optimisation normalized to prevent accuracy loss).	98
50	Global resource utilization in DPUCZDX8G FPGA.	99
51	Resource utilization for the YOLO v4 algorithm.	100
52	Power estimation for the YOLO v4 algorithm using analytical models.	101
53	RTL representation of a multiplexer inside the implemented object detection design.	102
54	RTL representation of the 53-convolution backbone inside the implemented object detection design.	103
55	Representation of the different blocks used to synthesize the HDL design (Synthesis).	104
56	Schematic of the implemented design for the ZCU104 design.	105
57	Implemented design visualization for the ZCU104 system.	106
58	Schematic of the implemented design for the ZCU104 design.	107

59	Resource utilization before and after implementation.	107
60	Routed elements of a convolution filter for the YOLO v4 algorithm (imple- mentation step).	108
61	Routed elements of a convolution filter for the YOLO v4 algorithm (zoomed on individual elements).	109
62	Example of random erase technique in 4 different images – [10].	123
63	Example of the <i>cut out</i> technique used on the CIFAR-10 dataset – [11]. . . .	124
64	Example of the hide-and-seek technique on a dog image – [12].	125
65	Examples of various iterations Cutout (first two on the left), Hide and Seek (subsequent two images) and the grid mask method (last) – [13].	125
66	Application of the grid mask technique using different parameters for image erasure.	126
67	"(a) input image to a convolutional neural network. The green regions in (b) and (c) include the activation units which contain semantic information in the input image. Dropping out activations at random is not effective in removing semantic information because nearby activations contain closely related in- formation. Instead, dropping continuous regions can remove certain semantic information (e.g., head or feet) and consequently enforcing remaining units to learn features for classifying input image" – [14].	127
68	Original images to be used in the mix up process – [15].	128
69	Mix up process of the two previous images – [15].	129
70	Demonstration of the cut mix process – [16].	129
71	Classification of a texture and a cat using ResNet-50 and the resultant texture- shape cue conflict created by the style transference between the initial images – [17].	130
72	Demonstration of different accuracies for five different images without the cure conflict effect – [17].	130
73	Two attention modules are used in the SAM technique – [18].	134
74	The input feature goes through pooling steps, the shared MLP and is then included into the channel attention module. The spatial module uses two outputs pooled together before transferring it to a convolution layer – [18]. . .	134
75	Structure of the modified SAM architecture – [18].	135
76	Skip connection.	135
77	"Representing hypercolumn classifiers as a neural network. Layers of the original classification CNN are shown in red, and layers that we add are in blue." – [19].	136
78	"The top-down architecture with skip connections, where predictions are made on the finest level. Bottom: our model that has a similar structure but leverages it as a <i>feature pyramid</i> , with predictions made independently at all levels." – [20].	137
79	"A building block illustrating the lateral connection and the top-down path- way, merged by addition." – [20].	138
80	The original image in displays three identifiable COCO classes: a dog, a bicycle and a car.	144

81	The image is divided into a 13x13 grid.	144
82	Each one of the divided cells in the 13x13 grid predicts different number of boxes and confidences. It is should be noted that bounding boxes can have different sizes and use multiple cells to constitute it.	145
83	The algorithm predicts the box size with a level of confidence.	145
84	The algorithm selects a new bounding box region to analyse size and confidence.	145
85	New bounding boxes are predicted by the algorithm	146
86	At the end of the process, many different bounding boxes are present in the picture – predicting different possible objects.	146
87	Simultaneously, each cell predicts different class probabilities.	146
88	The combination of the box and class predictions exposes all the different predictions made by each cell.	147
89	Using the non-maximum suppression technique (NMS) and a minimum confidence threshold, the process eliminates bounding boxes that have a low confidence level and are not compliant with the background context. The background context is partially defined by the level of confidence of each cell.	147
90	Representation of the YOLO darknet architecture.	148
91	Representation of the bounding box with class.	148
92	Representation of the attribution of the dog class to the specific cell using a centered x,y position. The box central position is going to define the class attributed to each cell on the grid.	149
93	The entire cell is defined as belonging to a specific class after the center of the bounding box falls to that area.	149
94	Out of all the bounding boxes falling with the (x,y) coordinates to that specific cell, the best one is chosen and then adjusted – increasing the total confidence of the bounding box.	149
95	Some of the bounding boxes is a specific location doesn't have an object in that area, so the level of confidence in that region is reduced.	150
96	Trained on the COCO dataset, the YOLO algorithm can predict a great variety of different objects – including different simultaneous predictions. . . .	150
97	Representation of a pass through layer.	155
98	Structure of an artificial neuron.	158
99	Types of activation functions.	159
100	Representation of the loss functions.	162
101	Different scales of learning rate.	166
102	Pooling layers represented by a 2x2 kernel.	168
103	Representation of the flattening operation.	169
104	Totally connected layer – a classical ANN.	170
105	Simplified architecture of a CNN.	170
106	Representation of the Intersection Over Union (IOU) between two lanes. Process used when selecting the best bounding box detection.	175
107	Representation of the non-max suppression.	176

List of Tables

1	Different methods of license plate and keypoint identification.	21
2	Computation time of license plate detection for a dedicated graphics card (GeForce RTX 2080 Ti) and and embedded computing board (NVIDIA Jetson TX2) – [2]	23
3	Hyperparameters used to train the license plate detector with the YOLO v4 algorithm from [21].	52
4	Hyperparameters used to train the license plate detector with the YOLO v4 algorithm from [21].	61
5	Parallelism for different FPGAs developed by Xilinx, Inc. – [22].	73
6	Resource utilization for different FPGAs developed by Xilinx, Inc. – [22]. . .	74
7	Extra LUTs of DPU with channel augmentation. – [22].	74
8	Number of blocks of random access memory of 36 kylobytes (BRAM36K) Blocks in Different Architectures for Each DPU Core – [22].	74
9	Representation of the performance and energy consumption evaluation metric for each platform. PC represents power consumption.	86
10	Performance data for the YOLO v3 algorithm for license plate detection. Accuracy data set after the optimization and quantization steps in 93.29 % was used to compare the differences in execution in each implementation platform at 50 % compression.	88
11	Performance data for the YOLO v4 algorithm for license plate detection. Accuracy data set after the optimization and quantization step in 95 % was used to compare the differences in execution in each implementation platform.	92
12	Performance data for the PoseNet algorithm for keypoint detection. Accuracy data set after the optimization and quantization step in 66 % was used to compare the differences in execution in each implementation platform.	94
13	Performance data for the OpenPose algorithm for keypoint detection. Accuracy data set after the optimization and quantization step in 65 % was used to compare the differences in execution in each implementation platform.	97
14	Resource utilization for the YOLO v4 algorithm (total resources).	100

Contents

1	Introduction	19
1.1	Context and motivation	19
1.2	State of the art	20
1.2.1	License plate and keypoint identification methods	20
1.2.1.1	Energy consumption among different data processing units	21
1.2.1.2	License plate identification with YOLOv4	23
1.2.1.3	FPGA implementation	24
1.2.1.4	Hardware acceleration technologies	25
1.3	Objectives	26
1.3.1	Main objective	26
1.3.2	Specific objectives	26
1.4	Main contributions	27
1.5	Structure of the text	28
2	Literature review	29
2.1	Convolutional neural networks	29
2.1.1	Introduction – CNNs	29
2.1.2	CNN structure	30
2.1.2.1	Input layer	30
2.1.2.2	Implicit layer	31
2.1.2.3	Hidden layer	31
2.1.2.4	Convolutional layer parameters	33
2.1.2.5	Activation function	34
2.1.2.6	Inception module	35
2.1.3	Learning paradigm	36
2.1.3.1	Supervised learning	36
2.1.3.2	Regularization	37
2.1.3.3	Batch normalization, BN	38
2.1.3.4	Skip connection	38
2.2	YOLO architectures	40
2.2.1	YOLO v3	40
2.2.2	YOLO v4	41
2.2.2.1	Main changes	41
2.2.2.2	Structure – Backbone of backbone feature extraction network	41
2.2.2.3	Feature pyramid	43
2.2.2.4	YoloHead	44
2.2.2.5	Decodification of predicted results	44
2.2.2.6	Drawing the bounding box	45
2.2.2.7	Bag of freebies	45
2.2.2.8	Bag of specials	45
2.3	PoseNet	45
2.3.1	Connection of keypoints for subjects	47

2.3.2	Keypoint position score	48
2.4	OpenPose	49
2.5	License plate detection	50
2.5.1	License plate recognition with multiple stages	50
2.5.2	License plate recognition with a single stage	51
2.5.2.1	Single stage detector implementation with a Multiprocessor systems-on-chip FPGA	51
2.5.3	License plate detection using YOLO	52
3	Methodology	54
3.1	Synthesis of models and frameworks	55
3.1.1	Model conversion	57
3.1.2	Frozen graph	58
3.1.3	Confirming acceptable losses	59
3.2	Optimization	59
3.2.1	Optimization steps	61
3.2.1.1	Fine-tuning	61
3.3	Quantization	61
3.3.1	Quantization mechanism	62
3.3.1.1	First sweep stage	62
3.3.1.2	Second sweep stage	63
3.4	Compilation	64
3.4.1	HDL High Level Synthesis	65
3.4.1.1	HLS structure	66
3.4.2	Wrapper for custom models	67
3.4.3	Vitis AI inference engine	67
3.4.3.1	Conversion between darknet model to caffe	67
3.4.4	Framework based inference with XIR	68
3.4.4.1	Specifics for object detection algorithms	71
3.4.4.2	Specifics for keypoint detection algorithms	72
3.4.4.3	Resource estimation	72
3.4.4.3.1	HLS based architecture	72
3.4.4.3.2	Vitis AI based architecture	73
3.5	Performance evaluation for keypoint detection algorithms and CNN power consumption evaluation	75
3.6	Performance evaluation for object detection algorithms	75
3.6.1	Mean Average Precision (mAP)	75
3.6.2	Energy evaluation	76
3.6.3	Hardware model estimation	76
3.6.3.1	The Vivado Design Suite	76
3.6.3.2	Power consumption estimation	81
3.6.4	OS system estimation	81
3.7	Materials	82

3.7.0.1	10th Generation Intel® Core™ i7-10750H processor (test on CPU)	83
3.7.0.2	GEFORCE GTX 1660 Ti (Test on GPU)	84
3.7.0.3	ZYNQ Ultrascale + MPSoC (Test on FPGA device)	85
4	Results	86
4.1	YOLO v3	86
4.2	YOLO v4	90
4.3	PoseNet	93
4.4	OpenPose	96
4.5	Resource consumption	98
4.5.1	Power estimation	101
4.5.2	Synthesis, implementation and routing	101
4.5.2.1	Synthesis	101
4.5.2.2	Implementation and routing	105
5	Conclusion and future work	110
5.1	Discussion	110
	Bibliography	112
6	Appendix A – Bag of freebies	123
6.1	Random Erase	123
6.2	Cut out	123
6.3	Hide-and-seek	124
6.4	Grid mask	125
6.5	Drop out	126
6.6	Drop connect	126
6.7	Drop block	127
6.8	Mix up	128
6.9	Cut mix	129
6.10	General Adversarial Networks – GAN	129
7	Appendix B – Bag of specials	131
7.1	Bag of specials	131
7.1.1	SPPNet	131
7.1.2	ASPP	132
7.1.3	Receptive Field Block – RFB	132
7.1.4	SE	133
7.1.5	SAM	133
7.1.6	Modified SAM	134
7.1.7	Skip connection	135
7.1.8	Hyper column	135
7.1.9	FPN	136

8	Appendix C – Other methods of learning	139
8.0.1	Batch learning	139
8.0.2	Online learning	139
8.0.3	Reinforcement learning	140
8.0.4	Active learning	140
8.0.5	Direct push learning	141
8.0.6	Incremental learning and decremental learning	141
8.0.7	Transfer Learning	142
8.0.8	YOLO v1	143
8.0.8.1	Detection process	144
8.0.8.2	Training steps	148
8.0.9	YOLO v2	152
8.0.9.1	Batch normalization	152
8.0.9.2	High resolution classifier	152
8.0.9.3	Convolutional with anchor boxes	153
8.0.9.4	Dimension clusters	153
8.0.9.5	Direct location prediction	154
8.0.9.5.1	Fine-grained features	154
8.0.9.5.2	Multi-scale training	155
9	Appendix D – Characteristics of a CNN	156
9.1	Part I	156
9.1.1	Main characteristics of a CNN	156
9.1.1.1	Connection between layers	156
9.1.1.2	Representation learning	156
9.1.1.3	Biological similarity	157
9.2	Part II	157
9.2.1	Activation functions	157
9.2.1.1	What is the activation function?	157
9.2.1.2	Why use activation function?	158
9.2.1.3	Commonly used activation functions	158
9.2.1.3.1	Linear	159
9.2.1.3.2	Sigmoid	159
9.2.1.3.3	Tanh	159
9.2.1.3.4	ReLU	160
9.2.1.4	Loss function	161
9.2.1.4.5	Regression problem	161
9.2.1.4.6	Classification problem	161
9.2.1.5	Gradient descent	163
9.2.1.6	Backpropagation error	164
9.2.1.6.7	Neuron model and its learning process	164
9.2.1.6.8	Fault tolerance	164
9.2.1.7	Learning rate	165
9.2.1.7.9	Definition of the learning rate	165

9.2.1.8	Momentum	165
9.2.1.9	Weight decay	166
9.2.1.10	Other methods of learning	167
9.2.1.11	Pooling Layers and Flattening	167
9.2.1.11.10	Fully connected layers and flattening	167
9.2.1.11.11	Dropout regularization	169
9.2.1.12	Transfer learning	171
9.2.1.12.12	Definition	171
9.2.1.12.13	Method	171
9.2.1.13	Fine tuning	172
9.3	Part III	173
9.3.1	Non-maximum suppression (NMS)	173
9.3.1.1	Implementation procedure	174

1 Introduction

1.1 Context and motivation

Artificial intelligence technologies are becoming vital for the insertion of vehicles into a new era of driving and logistics. Furthermore, the inclusion of autonomous electric vehicles in the EU by companies such as Tesla and Volkswagen [23] prompts the development of new methods to deal with obstacles and orientation in open roads and cities. Convolutional Neural Networks (CNNs) is one of the essential algorithms to develop autonomous orientation techniques [24], [25].

Training CNNs for object detection in wild scenarios requires a large amount of image data from the targeted environments. For vehicular applications, images from urban roads, highways, and general car pathways must be obtained.

The European General Data Protection Regulation (GDPR) 2016/679 has imposed new challenges for data collection where personal information such as faces and license plates must be removed from the images. Implementing this task through embedded hardware immediately after acquisition proves to be quite efficient [26].

License plate and pedestrian detection is paramount in computer vision due to its application in fraud prevention, safety and security in reserved areas, for example, to identify approved vehicles in gateways. Another application is to find specific plates related to security investigations in a concise amount of time. Table 1 the main license plate and keypoint detection techniques in the literature.

Even though car manufacturers mostly use Graphics Processing Units (GPUs) to handle real-time data processing, reductions in complexity and performance are required to handle many objects. Embedded hardware like Field Programmable Gate Arrays (FPGAs) is more compact, faster and more power-efficient than general use GPUs.

In this dissertation, the aim was to create an easily implementable top-down design workflow from a top level IA training framework, such as in python, to a convenient Data/Deep Learning Processing Unit (DPU). The evaluated DPUs are Central Processing Units (CPUs), Graphics Processing Units (GPUs), and Field Programmable Gate arrays (FPGAs). The initial high level frameworks analyzed are Pytorch (written in Lua/python), Caffe (C++/C), and TensorFlow (Python), which are the most used structures for AI research and development. Out of these, the most used for deep learning customization and deployment is Tensorflow [27] – given that the framework is written in python and can easily be used for different purposes by clustering and wrapping different functions required by specific AI applications.

The conventional method to develop embedded AI applications throughout recent years was to write a custom *Hardware description language* (such as VHDL, Verilog, or SystemC)

code to replicate the behavior of high abstraction codes written in python or C++/C to run the same functions in reconfigurable hardware platforms such as FPGA devices. However, this approach takes a long development time, even for experienced programmers, since one has to accommodate complex operations in a limited fixed-point arithmetic fashion. Unlike the usual approach, the proposed method uses a custom interpreter produced by Xilinx, Inc. in order to accelerate the development process and build an easily reproducible workflow starting from Python projects.

In this work, the You Only Look Once (YOLO) v4 network – [28], is built on the TensorFlow framework and also built using specific optimized layers on VHDL. This algorithm is the state-of-the-art technology used for efficient image detection. However, it uses considerable amount of energy and hardware resources for real-time processing. This algorithm has a significant role in the automotive industry, most notably in autonomous vehicles. Said vehicles must be able to identify different features in the surrounding area in order to navigate according to traffic regulations.

In order to test the workflow using different algorithms, the YOLO v3 – [3] the predecessor of YOLO v4, is built using the same methodology.

The OpenPose [29] and PoseNet [5] algorithms are implemented in order to test the developed methodology for different kinds of CNNs related to recognition of patterns in the immediate vehicle environment. These CNNs identify varying amounts of keypoints on a civilian's body using images acquired from conventional cameras.

The development of embedded AI applications opens new possibilities for improvements and acceleration of autonomous driving in terms of both hardware requirements (price) and energy performance.

1.2 State of the art

1.2.1 License plate and keypoint identification methods

License plate detection is essential in computer vision because it can be used for vehicle tracking and object identification. In addition, fraud prevention and security in reserved areas also employ this technology to identify approved vehicles and open gates. Finally, this technology can also study the footage of millions of live video feeds to find specific plates that may be related to security investigations in a concise amount of time.

Five main methods are used for identifying license plates; they are summarized in table 1 alongside the studies and characteristics.

Method	Characteristics	References
Deep learning	Complex and resource heavy computing yields very efficient and accurate predictions even with environment changes	[30], [31], [32], [33], [34], [35], [36]
Statistical classifier	Resource heavy and accurate even with environment changes	[37], [38], [39], [40], [41], [42], [43]
Texture identification	Resource heavy and poor accuracy for different lightning conditions, efficient for boundary deformations	[44], [45], [46], [47], [48], [49], [50]
Colour identification	Accurate even with deformation and rotation, sensitive to light	[51], [52], [53], [54], [55]
Edge detection	Low computational complexity but innacurate for blurry or distorted images	[56], [57], [58]
DLCNN keypoint	Accurate predictions and high computational complexity	[59], [60], [29]

Table 1: Different methods of license plate and keypoint identification.

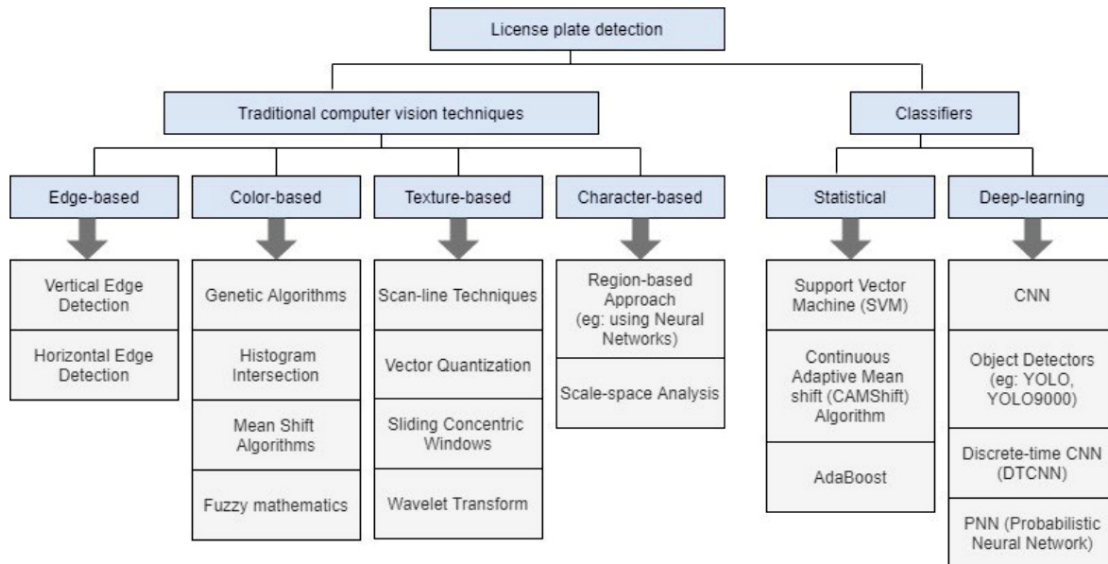


Figure 1: Different techniques used for license plate detection.

t

The studied algorithms' main difficulty was finding an accurate method to identify license plates in different lighting and environmental conditions. The most efficient observed method so far is through deep learning analysis, which consists of complex calculations and significant energy consumption – [61].

1.2.1.1 Energy consumption among different data processing units

Four different types of hardware serve as data processors in CNN.

The first is the CPU, which uses a fixed generic processor architecture to run the software. However, While running the operating system, many parallel operations and extra

calculations co-occur in the CPU, slowing down processing time and tangling its performance to the number of resources available.

The next is the GPU which has its performance based on the number of resource cores present in the packaged solution. On the other hand, graphics cards can deal with parallel operations using different tensor cores to increase the processing power of CNNs to 4-5 times the speed obtained for CPUs – [62].

The third alternative are the FPGA devices. These devices are re-programmable integrated circuits used to deploy individual applications that require the use of parallel computing, low power consumption, and scaled down electronics. The structure of logical gates inside the device can be modified akin to reconfiguring the factory settings of the CPU or GPU processors, optimizing the map of logical circuits to execute the desired applications using the least amount of resources and the fastest way. The algorithm's performance is not entirely based on the number of resources the board has but on the complexity of the code that was written to run the application. The study in [1] shows the performance of three of the alternatives when operating with vision kernels, including OpenCV, NVIDIA VisionWorks, and OpenCV implemented in the FPGA. The kernels used in the study include image arithmetic, filtering analysis, geometric transformation, and other composite kernels. Figure 2 demonstrates the energy consumption for different image kernel operations.

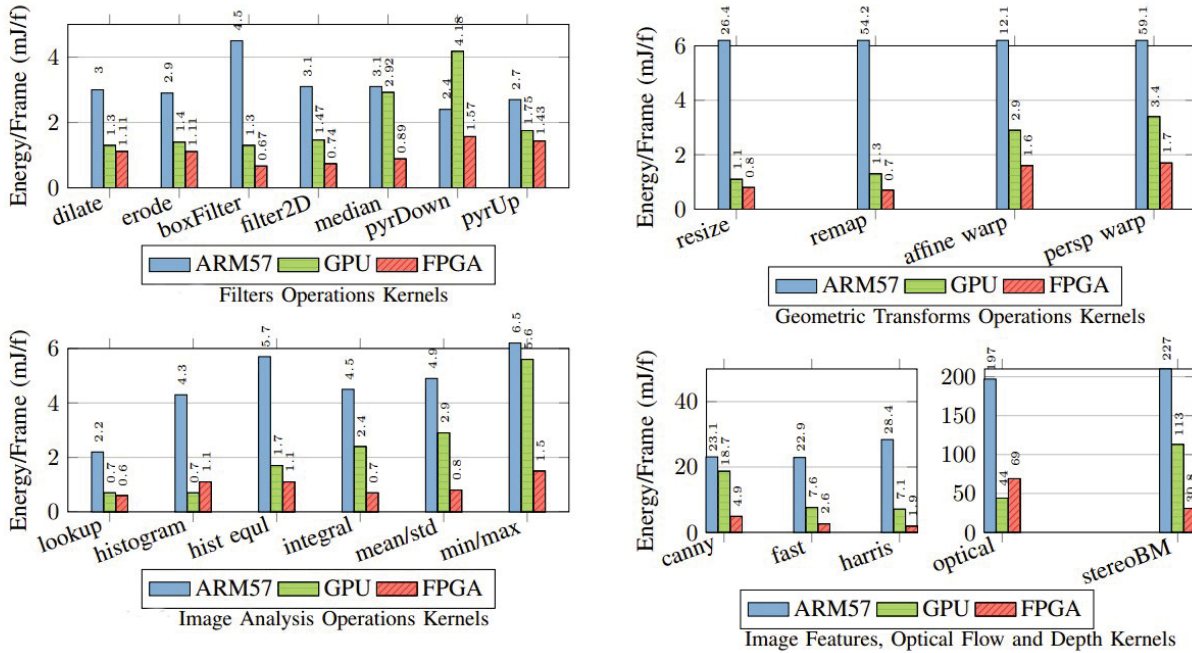


Figure 2: Energy consumption per frame of the main three data processing units for different image kernel operations – [1]

Figure 2 demonstrates that the energy per frame for the FPGA is much less for many

operations. Even considering that FPGA devices provide considerable gain in frame rate for CNNs in most cases, the power consumption is overall smaller than that observed of the GPU and justifies further development using this alternative.

At last, one may develop an ASIC (Application-specific Integrated Circuit) device. The ASIC device is a dedicated chip made only with the necessary logic gates to conduct the target operation. The transistor can be carefully dimensioned considering the target processing speed and quiescent current. Therefore, ASIC devices are the most efficient ones for any CNN application [63]. On the other hand, they are costly to produce, require an extensive design time and also have a fixed structure that cannot be modified for updates.

1.2.1.2 License plate identification with YOLOv4

The state-of-the-art algorithm for object identification regarding energy use and accuracy is YOLO v4. It is possible to identify license plates in images in real-time with GPU devices, like NVIDIA GeForce RTX 2080 Ti. Nevertheless, smaller commercial portable devices that operate using higher-level programming languages are still unsuited for real-time detection, as can be seen in table 2.

Detector	GeForce RTX 2080 Ti		NVIDIA Jetson TX2	
	Time (ms)	FPS	Time (ms)	FPS
SSD	51	19.6	74	13.5
SSD (TensorRT)	4.38	228.3	32.5	30.77
YOLOv4 (256)	9	111	105	9.5
YOLOv4 (608)	25	40	470	2.1

Table 2: Computation time of license plate detection for a dedicated graphics card (GeForce RTX 2080 Ti) and and embedded computing board (NVIDIA Jetson TX2) – [2]

Figure 3 shows that the chosen GPU and embedded computer are capable of processing both the entire plate and the characters of the license plate with wildly varying speeds.



Figure 3: Demonstration of the YOLOv4 algorithm detecting different individual details of a license plate alongside a license plate for multiple objects – [2].

1.2.1.3 FPGA implementation

The FPGA devices are being used to optimize the operation and training of various CNNs. Different from GPUs, these devices require exponentially smaller quantities of energy consumption to deliver much higher performance. This occurs mainly because the pertinent logical gates inside the device are uniquely routed to perform the target task only - differently from CPUs or GPUs, which have a static logical architecture.

FPGA devices provide a smaller and much more efficient acceleration alternative and can be implemented for any CNN and for custom-built interfaces between different sensors – [64].

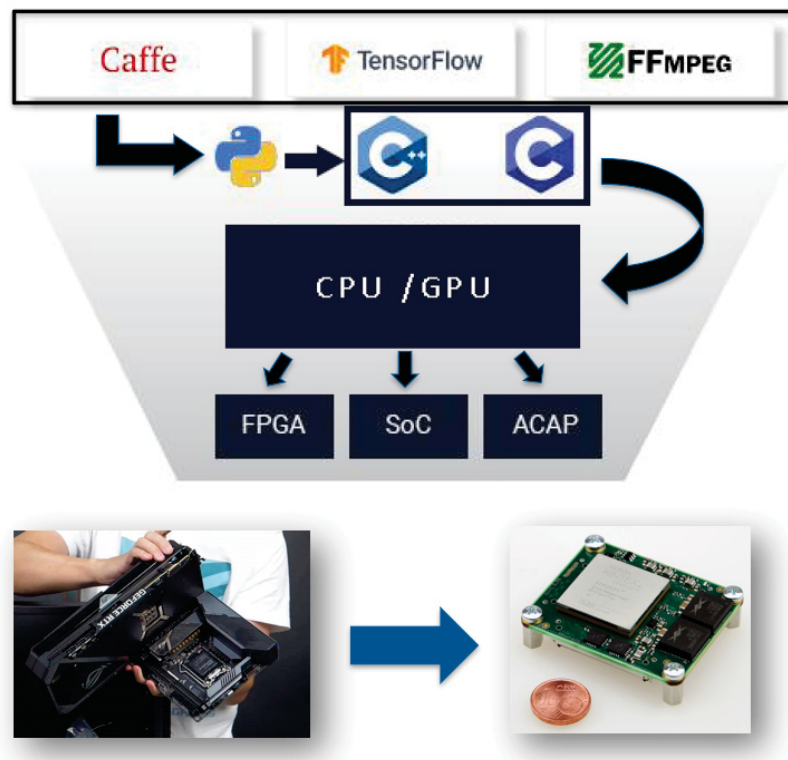


Figure 4: Representation of the acceleration function.

1.2.1.4 Hardware acceleration technologies

Hardware acceleration is already done when running conventional applications from a CPU to a GPU. However, further acceleration is done using FPGA devices in a new manner. The development of applications and codes using FPGAs requires knowledge of *hardware description languages* which are low-level and challenging to deploy. Xilinx Inc., which manufactures and sells FPGA devices, developed a way to use higher-level C/C++ and Python languages to develop applications and significantly reduce development times – [65]. This is done using precompiled and rarely modified logical functions called higher functions. This ensures that the final application still has a very efficient performance compared to what could be constructed using the VHDL or Verilog languages – [66].

The adopted structure is integrated to the main machine learning frameworks: TensorFlow – [27], Caffe – [67], [68].

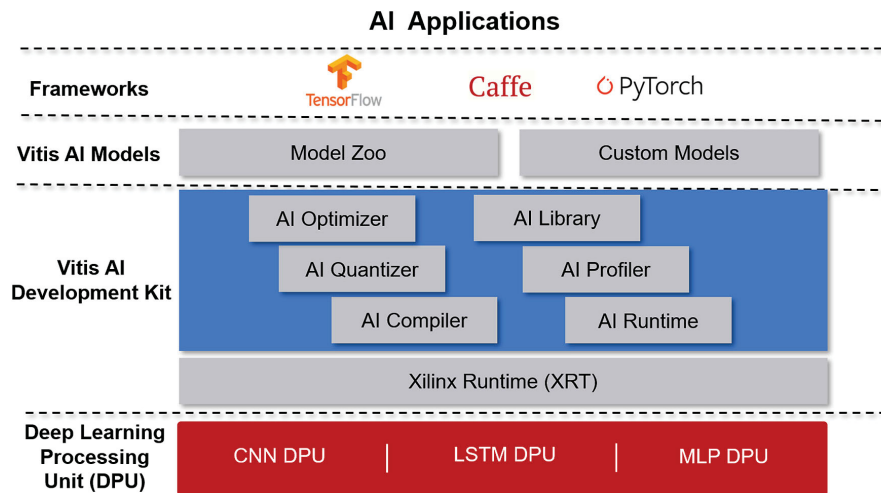


Figure 5: Structure of the FPGA high level language interpreter.

Figure 5 presents the Vitis workflow used to get algorithm from the main frameworks to the CNN, Long Short Term Memory (LSTM) and Multilayer Perceptron Deep Processing units.

Unlike previous digital integrated circuit design, the user does not have a deep understanding of the HDL languages to deploy plug-and-play applications with drivers that work directly with third devices such as cameras and sensors that have an available interface such as Universal Serial Bus (USB) or Recommended Standard-232 (RS-232).

1.3 Objectives

1.3.1 Main objective

The project's initial goals are to study the digital processing and CNN techniques used in the YOLO v3, YOLO v4, PoseNet and OpenPose algorithms to estimate which parts of the deep learning convolutional neural networks are synthesizable using VHDL/Verilog hardware description languages (HDLs) – VHDL and Verilog. Subsequently, parts of the evaluated code will be adapted to operate with the simulation platforms of Xilinx Inc. (Vivado and Vitis) to create a VHDL/bitstream code for a target FPGA (Field Programmable Gate Array) device.

1.3.2 Specific objectives

The following are the project's specific objectives:

- Synthesize logic blocks from higher computer vision frameworks using VHDL/Verilog hardware description languages (HDLs) and specific DPU generation bitstreams;
- Subsequently, parts of the evaluated code will be adapted to operate with the simulation platforms of Xilinx Inc. (Vivado and Vitis) in order to create a VHDL/bitstream code for a target FPGA (Field Programmable Gate Array) device;
- The implemented code will be validated using processors such as different CPUs and GPUs;
- After successful theoretical validation, the design code will be tested using a physical target FPGA device;
- The YOLO v4, PoseNet and Openpose algorithms should be analyzed and analyzed using the published material on theoretical and practical analysis – [69], [3] and [70].
- Each technique will then be evaluated and extracted from the YOLO v4 open source code made available by the author via *github*. The original algorithm is written in C, C++, and Python;
- The compatible code will be synthesized into the VHDL hardware description language using Vivado High-Level Synthesis (HLS) to test the written code through logic gates via the ModelSim/Quarta simulator;
- After confirming that the results produced by the ModelSim/Quarta simulator are identical or similar to those produced by the GPU unit, the VHDL code will be assigned to a target FPGA device that will be determined depending on all the feature requirements exposed in previous simulations.

1.4 Main contributions

The main contributions of this thesis are:

1. Development of a code interpreter tool that converts high-level programming languages to low-level HDL languages;
 - (a) Python, C++ and C high-level programming languages are converted into VHDL and Verilog HDLs;
 - (b) Direct bitstream generation from high-level programming languages.
2. Development of an optimization structure for the TensorFlow, Caffe and Pytorch DL-CNN frameworks;
3. Creation of a top-down workflow used to evaluate and implement the creation of optimized DLCNNs;
4. Comparison of object and keypoint detection DLCNNs between different processing platforms (CPU, GPU and FPGA).

1.5 Structure of the text

The remainder of this thesis is organized as follows: in section II, a literature review is presented, focusing on neural networks and their different types. Section III is about the methodology and materials. Section IV presents the results obtained with the developed methodology, emphasizing the performance obtained among different platforms. Finally, section V presents the final considerations with a conclusion of the thesis and suggestions for future works.

2 Literature review

The literature review is focused on the theoretical description of neural networks and their architecture alongside the structure of the data processing units and their technologies used for hardware acceleration.

2.1 Convolutional neural networks

2.1.1 Introduction – CNNs

The research on convolutional neural networks can be traced back to 1980 [71], where a neural network named "neocognitron" was designed after the human visual cortex. Neocognitron was one of the first proposed deep learning algorithms, with alternating S-layer and C-layer implicit layers [72]. The neocognitron's S-layer-C-layer combination can perform feature extraction and filtering: the S-layer unit extracts the image features in the receptive field, and the C-layer unit receives and responds to the same features returned from different receptive fields, partially implementing the convolution layer and pooling layer in convolutional neural networks [73].

The first convolutional neural network was the Time Delay Neural Network (TDNN) proposed by Alexander Waibel et al. in 1989 [74]. TDNN is a convolutional neural network that uses the Fast Fourier Transform (FFT) preprocessed speech signal as input to the speech recognition problem. Its hidden layer consists of two one-dimensional convolutional kernels to extract translation-invariant features in the frequency domain. TDNN was able to use the Back-Propagation BP framework for learning within the framework [75]. The TDNN outperformed the Hidden Markov Model (HMM) for speech recognition in the 1980s, under equivalent conditions.

In 1988, Wei Zhang [76] proposed the first two-dimensional convolutional neural network: the space invariant artificial neural networks (SIANN) and applied it to detect medical images. Independent of Zhang (1988), Yann LeCun also built a convolutional neural network for computer vision problems in 1989, the original version of LeNet. LeCun (1989) [77] used Stochastic Gradient Descent (SGD) [78] for learning after random initialization of the weights, a strategy that has been retained by subsequent deep learning studies. In addition, LeCun (1989) first used the term "convolution" when discussing the structure of his network, hence the name "convolutional neural network".

LeCun's (1989) work was co-developed by ATT Bell Laboratories in 1993 and deployed in the NCR (National Cash Register Corporation) check reading system [73]. However, in general, due to the limited numerical computing power, insufficient learning samples, and the rise of kernel learning methods represented by Support Vector Machine (SVM) [79] during the same period, convolutional neural networks designed for various image processing prob-

lems remained in the research stage during this period, with less diffusion at the application end.

Based on LeNet, a complete convolutional neural network, LeNet-5 [80], was built by Yann LeCun and his collaborators in 1998 and was successful in the problem of handwritten digit recognition. LeNet-5 and its subsequent variants define the basic structure of modern convolutional neural networks. The alternating convolutional-pooling layers in its construction are believed to extract translation-invariant features of the input image. Other applications based on convolutional neural networks have also been developed, including portrait recognition, gesture recognition, and so on.

After the deep learning theory was proposed in 2006 [81], the representational learning capability of convolutional neural networks gained attention and was developed with the upgrade of computing devices. Since AlexNet [82] in 2012, complex convolutional neural networks supported by GPU computing clusters have been the winning in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [83] several times, including ZFNet [84] in 2013, VGGNet [85], GoogLeNet [86] in 2014, and ResNet [87] in 2015. The most recent CNN structures will be further explained in the sequence.

2.1.2 CNN structure

2.1.2.1 Input layer

The input layer of a convolutional neural network can process multidimensional data; commonly, the input layer of a 1D convolutional neural network receives 1D or 2D arrays, where the 1D arrays are usually temporal or spectral samples; 2D arrays may contain multiple channels; the input layer of a 2D convolutional neural network receives 2D or 3D arrays, and the input layer of a 3D convolutional neural network receives 4D arrays [16]. Since convolutional neural networks are widely used in the field of computer vision, many studies have introduced their structures with the presupposition of three-dimensional input data, i.e., two-dimensional pixel points on a plane and RGB channels.

Like other neural network algorithms, the input features of convolutional neural networks need to be normalized due to the use of the gradient descent algorithm for learning. Specifically, the input data needs to be normalized in the channel or time/frequency dimension before inputting the learning data into the convolutional neural network. In the case of pixels, the original pixel values are distributed in the interval. Normalization of input features is beneficial to improve the learning efficiency and performance of convolutional neural networks.

2.1.2.2 Implicit layer

The implicit layer of a convolutional neural network consists of three common constructions: convolutional, pooling, and fully connected layers, and in some more modern algorithms, there may be complex constructions such as Inception modules and residual blocks. Among the standard constructions, the convolutional and pooling layers are specific to convolutional neural networks. The convolutional kernel in the convolutional layer contains weight coefficients. In contrast, the pooling layer does not contain weight coefficients, so the pooling layer may not be considered as a separate layer in the literature. In LeNet-5, for example, the order of the three classes of common constructs in the implicit layer is usually: input-convolutional layer-pooling layer-full-connected layer-output.

2.1.2.3 Hidden layer

The function of the convolutional layer is to extract features from the input data, and it contains several convolutional kernels, each element of which corresponds to a weight factor and a bias vector, similar to a neuron in a feed-forward neural network. Each neuron in the convolutional layer is connected to multiple neurons in a region close to the previous layer, and the size of the region depends on the size of the convolutional nucleus, which is referred to in the literature as the "receptive field," which is analogous to the receptive field of visual cortical cells. The convolutional kernel works by regularly sweeping through the input features, multiplying the input features by matrix elements, and superimposing the bias in the receptive field.

$$Z^{l+1}(i, j) = [Z^l \otimes w^{l+1}](i, j) + b = \sum_{k=1}^{K_l} \sum_{x=1}^f \sum_{y=1}^f [Z_{y=1}^f(s_0 i + x, s_1 j + y) w_k^{l+1}(x, y)] + b \quad (1)$$

$$(i, j) \in 0, 1, \dots, L_{l+1} | L_{l+1} = \frac{L_l + 2p - f}{s_0} + 1 \quad (2)$$

where Z represents the convolutional kernel (or pixel of the feature map), i and j are the indexes used to represent the bidimensional kernel values, k is the depth of the kernel values, l is the index used to denote the input and output of the layer, b denotes the bias values, w_k represents the array of weights, K represents the number of channels in the feature map, f , s_0 , s_1 , p are the convolution layer parameters (individual weights). L represents the feature map for a specific layer. x and y are the specific coordinates for each pixel.

The summation part of Eq.1 is equivalent to solving the cross-correlation once. It is the deviation quantity and denotes the convolutional input and output of the first layer, also known as the feature map, and is the size of the feature map, which is assumed to be the same length and width. is the number of channels in the feature map, and, and are the

convolutional layer parameters corresponding to the convolutional kernel size, convolutional stride, and padding layers.

The above equation uses a two-dimensional convolution kernel as an example, and a one-dimensional or three-dimensional convolution kernel works similarly. Theoretically, the convolution kernel could also be flipped 180 degrees and then solved for the cross-correlation. The result would be equivalent to a linear convolution satisfying the exchange law, but this does not facilitate solving for the parameters while increasing the solution steps, so the linear convolution kernel uses cross-correlation instead of convolution.

In particular, when the convolution kernel is a unit convolution kernel of size, step and does not contain padding, the computation of cross-correlation within the convolutional layers is equivalent to matrix multiplication. Thus a fully connected network is constructed between the convolutional layers.

$$Z^{l+1} = \sum_{k=1}^{K_l} \sum_{i=1}^L \sum_{j=1}^L (Z_{i,j,k}^l w_k^{l+1}) + b = w_{l+1}^T Z_{l+1} + b, L^{l+1} = L \quad (3)$$

A convolution layer consisting of unit convolution kernels is also known as Network-In-Network (NIN) [88] or multilayer perceptron convolution layer (mlpconv). The unit convolution kernel can reduce the number of channels in the graph while maintaining the feature graph size, thus reducing the computational effort of the convolution layer. A convolutional neural network built entirely from unit convolution kernels is a multilayer perceptron (MLP) with shared parameters.

Based on linear convolution, some convolutional neural networks use more complex convolution, including tiled convolution, deconvolution, and dilated convolution. In tiled convolution, the convolution kernel sweeps only a portion of the feature map, and the remaining portion is processed by other convolution kernels in the same layer, so the parameters are only partially shared among the convolution layers, which facilitates the neural network to capture the rotation-invariant features of the input image. Deconvolution or transposed convolution connects a single input excitation with multiple output excitations to amplify the input image. Convolutional neural networks consisting of deconvolution and up-pooling layer have applications in semantic segmentation of images and are also used to build Convolutional AutoEncoder (CAE) [89]. Dilated convolution introduces a dilation rate based on linear convolution to increase the perceptual field of the convolution kernel to obtain more information about the feature map, which helps capture the long-range dependency of the learning target when used for sequence-oriented data. Convolutional neural networks using dilated convolution are mainly used in Natural Language Processing (NLP) [90], such as machine translation, speech recognition, and others.

2.1.2.4 Convolutional layer parameters

Convolutional layer parameters include convolutional kernel size, step size, and padding, which determines the size of the output feature map of the convolutional layer and are the hyperparameters of the convolutional neural network. The convolutional kernel size can be specified as any value smaller than the input image size; the more significant the convolutional kernel, the more complex the input features can be extracted. The convolutional step size defines the distance between two adjacent sweeps of the convolutional kernel over the feature map; for a convolutional step size of 1, the kernel sweeps over the elements of the feature map one by one, and for a step size of n , $n-1$ pixels are skipped in the next sweep.

From the cross-correlation calculation of the convolution kernel, it is clear that the size of the feature map decreases gradually as the convolution layers are stacked, e.g., a 16×16 input image will output a 12×12 feature map after a unit-step, padding-free five $\times$ five convolution kernel. For this reason, padding is a way to artificially increase the size of the feature map before it passes through the convolution kernel to offset the effect of size shrinkage in the computation. Standard padding methods are padding by 0 and replication padding. Padding can be classified into four categories based on the number of layers and their purpose.

- Valid padding: i.e., no padding is used at all, and the convolution kernel allows access only to the locations in the feature map that contain the full sensory field. All pixels of the output is a function of the same number of pixels in the input. Convolution with valid padding is called "narrow convolution," and the size of the feature map output by narrow convolution is $(L - f)/s + 1$.
- Same/half padding: Only enough padding is applied to keep the output and input feature maps of the same size. The size of the feature map does not shrink with the same padding, but the part of the input pixels near the boundary has less impact on the feature map than the middle part, i.e., there is an under-representation of the boundary pixels. Convolution with equal padding is called "equal-width convolution."
- Full padding: padding enough so that each pixel is visited the same number of times in each direction. With a step size of 1, full padding outputs a feature map of size $L + f - 1$, which is larger than the input value. Convolution with full padding is called "wide convolution." Arbitrary padding: between effective padding and full padding, artificially set padding, rarely used.

Bringing in the previous example, if a 16×16 input image is padded with the same padding before passing through a 5×5 convolution kernel with a unit step, it will be padded horizontally and vertically with two layers, i.e., 2 pixels on each side ($p = 2$) to become a 20×20 size image. After passing through the convolution kernel, the output feature map size is 16×16 , keeping the original size.

2.1.2.5 Activation function

The convolutional layer contains excitation functions to assist in expressing complex features and is represented as follows.

$$A_{i,j,k}^l = f(Z_{i,j,k}^l) \quad (4)$$

A represents the complex features. Like other deep learning algorithms, convolutional neural networks usually use a linear rectification function (Rectified Linear Unit, ReLU). Other ReLU-like variants include ReLU with slope (Leaky ReLU, LReLU), Parametric ReLU (Parametric ReLU, PReLU), Randomized ReLU (Randomized ReLU, RReLU), Exponential Linear Unit (ELU), etc... Before the advent of ReLU, Sigmoid functions and hyperbolic tangent functions were also used.

The excitation function operation usually follows the convolutional kernel, and some algorithms using pre-activation techniques place the excitation function before the convolutional kernel. In some early studies of convolutional neural networks, such as LeNet-5, the excitation function comes after the pooling layer.

Pooling layer

After feature extraction in the convolution layer, the output feature map is passed to the pooling layer for feature selection and information filtering. The pooling layer contains a predefined pooling function, which replaces the result of a single point in the feature map with the feature map statistics of its neighboring regions. The pooling layer selects the pooled regions in the same steps as the convolutional kernel scan feature map, controlled by the pooling size, step size, and padding. Pooling strategies include:

- Lp pooling

Lp pooling is a class of pooling models inspired by the hierarchical structure within the visual cortex, which is generally represented as:

$$A_k^l(i, j) = \left[\sum_{x=1}^f \sum_{y=1}^f A_k^l(s_0i + x, s_0j + y)^p \right]^{\frac{1}{p}} \quad (5)$$

The new introduced step variable is p . The meanings of step and pixel are the same as those of the convolutional layer and are pre-specified parameters. When Lp pooling takes the average value in the pooling region, it is called average pooling; when Lp pooling takes the extreme value in the region, it is called max pooling. Average pooling and max pooling are long-used pooling methods in the design of convolutional neural networks, both of which preserve the background and texture information of the image at the expense of losing some information or size of the feature map. In addition, $L2$ pooling has been used in several works.

- Random/Mixed Pooling

Mixed pooling and stochastic pooling are extensions of the L_p pooling concept. Stochastic pooling randomly selects a value within its pooling region according to a specific probability distribution to ensure that a portion of the non-extreme excitation signal makes it to the subsequent construct. Hybrid pooling can be expressed as a linear combination of mean pooling and maximal pooling.

$$A_k^l = \lambda L_1(A_k^l) + L_\infty(A_k^l), \lambda \in [0, 1] \quad (6)$$

λ is a random value between 0 and 1. It has been shown that hybrid pooling and stochastic pooling have the function of regularization compared to mean and great pooling, which helps to avoid overfitting in convolutional neural networks.

- Spectral pooling

Spectral pooling is an FFT-based pooling method that can be used together with FFT convolution to build FFT-based convolutional neural networks. Given the feature map size and the output size of the pooling layer, spectral pooling performs a Discrete Fourier Transform (DFT) transform on each channel of the feature map separately and intercepts a sequence of size $n \times n$ from the center of the spectrum to perform an inverse DFT transform to obtain the pooling result. Spectral pooling has a filtering function that can resize the feature maps while preserving the low-frequency variation information of the input features. Based on the FFT algorithm, spectral pooling can be done with a small computational effort.

2.1.2.6 Inception module

An *Inception module* is an implicit layer construction obtained by stacking several convolutional and pooling layers. Specifically, an Inception module contains several different types of convolution and pooling operations simultaneously and uses the same padding to obtain feature maps of the same size for the above operations, and subsequently stacks the channels of these feature maps in an array and passes the excitation function. Since the above approach introduces multiple convolutional kernels in a single construct, the Inception module is usually designed with a bottleneck layer to simplify the computation by first reducing the number of channels of the feature map using unit convolutional kernels, i.e., the NIN structure, before performing other convolutional operations. The Inception module was first applied to GoogLeNet successfully in the ImageNet dataset and inspired (or generalized) a series of lightweight convolutional neural networks built on depthwise separable convolution, including Xception and MobileNet.

Fully-connect layer

The fully connected layer in a convolutional neural network is equivalent to the hidden layer in a traditional feedforward neural network. The fully connected layer is located in the

last part of the hidden layer of the convolutional neural network and transmits signals only to the other fully connected layers. The feature map loses its spatial topology in the fully connected layer and is expanded into vectors and passed through the excitation function.

According to the representational learning view, the convolutional and pooling layers in a convolutional neural network can perform feature extraction on the input data, while the role of the fully connected layers is to perform a nonlinear combination of the extracted features to obtain the output, i.e., the fully connected layers are not expected to have feature extraction capabilities per se, but rather try to accomplish the learning goal using the available higher-order features.

In some convolutional neural networks, the function of the fully connected layer can be replaced by global average pooling, which averages all the values of each channel of the feature map, i.e., if there is a $7 \times 7 \times 256$ feature map, global average pooling will return a 256 vector in which each element is 7×7 in steps of 7, with no padded average pooling.

Output layer

The upstream of the output layer in a convolutional neural network is usually a fully connected layer, so its structure and working principle are the same as the output layer in a traditional feedforward neural network. The output layer uses a logistic function or a normalized exponential function (softmax function) to output the classification labels for image classification problems. In the object detection problem, the output layer can be designed to output the center coordinates, size, and object classification. In image semantic segmentation, the output layer directly outputs the classification results for each pixel.

2.1.3 Learning paradigm

This section will review the main aspects of supervised learning, regularization, batch normalization and the skip connection feature – common to all the convolutional neural networks.

2.1.3.1 Supervised learning

Convolutional neural networks use a BP framework for learning in supervised learning, whose computational process was identified in LeCun (1989) [77] and was one of the first deep algorithms to learn in a BP framework. BP in convolutional neural networks is divided into three parts, i.e., the fully connected layer with the backward propagation of the convolutional kernel and the backward pass of the pooling layer. The BP computation in the fully connected layer is the same as that of a traditional feedforward neural network, and the backpropagation in the convolutional layer is a cross-correlation computation similar to forwarding propagation:

$$\left(\frac{\partial E}{\partial A}\right)_{i,j}^l = \sum_{k=1}^{K_l} \sum_{x=1}^f \sum_{y=1}^f \left[w_k^{l+1}(x, y) \left(\frac{\partial E}{\partial A}\right)_{s_0 i+x, s_0 j+y, k}^{l+1} \right] f'(A_{i,j}^l) \quad (7)$$

$$w^l = w^{l+1} - \alpha \left(\frac{\partial E}{\partial A}\right)_k = w^{l+1} - \alpha \left[A^{l+1} \left(\frac{\partial E}{\partial A}\right)_k^{l+1} \right] \quad (8)$$

Where E is the error calculated by the cost function, f' is the derivative of the excitation function, α is the learning rate. If the forward propagation of the convolutional kernel is calculated using convolution, the backward propagation also flips the convolutional kernel to perform convolutional operations. The error function of a convolutional neural network can have various choices, commonly including softmax loss, hinge loss, and triplet loss.

The pooling layer has no parameter updates in backpropagation, so it is only necessary to assign the errors to the appropriate locations in the feature map according to the pooling method; for extreme pooling, all errors are assigned to the location of the extreme value; for mean pooling, the errors are equally assigned to the entire pooling region.

Convolutional neural networks usually use Stochastic Gradient Descent (SGD) [78] within the BP framework and its variants, such as the Adam algorithm (Adaptive moment estimation). SGD computes gradients with randomly selected samples in each iteration, which facilitates information filtering in the case of sufficient learning samples. It facilitates information filtering in the case of sufficient learning samples, fast convergence at the beginning of the iteration, and less computational complexity.

2.1.3.2 Regularization

Various types of regularization methods in neural network algorithms can be used in convolutional neural networks to prevent overfitting. Common regularization methods include Lp-norm regularization, spatial dropout, and drop connect. Figure 6 represents the ReLU regularization.

L_p regularization incorporates implicit layer parameters in the definition of the loss function to constrain the complexity of the neural network.

$$L(X, Y, w) = L(X, Y, w) + \lambda \sum ||w||^p \quad (9)$$

where $L(X, Y, w)$ is the loss function and the summation term containing the Frobenius norm is called the regularization term, where λ is the regularization parameter that determines the binding force of the regularization term and p is the step. It can be shown that the regularization term is a convex function when; in particular, the L2 regularization becomes Tikhonov regularization when the L_p regularization at times favors the sparsity

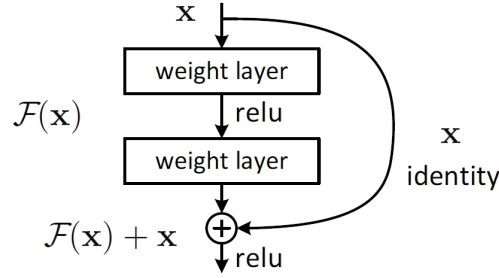


Figure 6: Representation of the ReLU functions for CNNs.

of the convolutional kernel weights, but the regularization is not convex at this time. The spatial dropout in convolutional neural networks is a generalization of the theory of random deactivation in feedforward neural networks. In the learning of fully connected networks, random deactivation randomly zeroes the output of a neuron, while spatial random deactivation randomly selects channels of the feature map to zero in an iteration. Further, random connection deactivation acts directly on the convolutional kernel, zeroing out some of the weights of the convolutional kernel in an iteration. It has been shown that spatial random deactivation and random connection deactivation enhance the generalization ability of convolutional neural networks, which is beneficial in improving learning performance when learning samples are insufficient.

2.1.3.3 Batch normalization, BN

Normalization of data is a common step in the preprocessing of the input pipeline of a neural network, but in deep networks, the mean and standard deviation of the input data change as they are passed from level to level within the hidden layer, giving rise to the phenomenon of covariate shift. Batch normalization (BN) partially solves this problem at the cost of introducing additional learning parameters by first normalizing the features in the hidden layer and then scaling up the normalized features using two linear parameters as new inputs, and the neural network updates its BN parameters. The BN parameters in a convolutional neural network have the same nature as the convolutional kernel parameters, i.e., pixels in the same channel in the feature map share a set of BN parameters. In addition when using BN the convolutional layer does not need a bias term and its function is replaced by the BN parameters.

2.1.3.4 Skip connection

Jump connections or shortcut connections are derived from jump connections and various gating algorithms in Recurrent Neural Networks (RNNs), and are techniques used to alleviate the gradient disappearance problem in deep structures. Jump connections in convolutional neural networks can span an arbitrary number of implicit layers and are illustrated

here by jumps between adjacent implicit layers.

$$A^l = f(Z^l + uZ^{l-1}) \quad (10)$$

Where u is the conversion factor of the feature map. When the sizes of and are different, the conversion factor converts the smaller size feature map, usually, to the size of the matrix element operation, ensuring that the matrix element operation holds. When the output value of is small and the output value of is large, the output of the convolutional layer approximates the equivalence function and has no negative impact on the feature transfer of the layer, thus setting the learning baseline of the layer so that the layer does not degrade at least during the iterations. Within the BP framework, part of the error can be directly applied to the layer by skipping the layer during backpropagation, which compensates for the gradient loss caused by its level-by-level propagation in the depth structure and therefore facilitates the error propagation in the depth structure. The combination of multiple convolutional layers containing jump connections is called the residual block and is the building block of some convolutional neural network algorithms, such as ResNet.

Convolutional neural networks can be accelerated using similar techniques to other deep learning algorithms to improve operational efficiency, including quantization, transfer learning, etc. Quantization, i.e., using low numerical precision in the computation to increase the computational speed, has been tried in some deep algorithms. For convolutional neural networks, an extreme example is XNOR-Net, i.e., a convolutional neural network built only from heterogeneous gates (XNOR). The general strategy of migration learning is to migrate unlabeled data to labeled data to improve the performance of the neural network, and migration learning in convolutional neural networks is usually to initialize a new convolutional neural network using the convolutional kernel weights that have completed learning under labeled data, migrate to unlabeled data, or apply to other labeled data to shorten the learning process.

FFT Convolution

Both convolution and pooling calculations of convolutional neural networks can be performed by FFT conversion to the frequency domain, when the FFT of the convolutional kernel weights and gradients in the BP algorithm can be reused and the inverse FFT is only used in the output results, reducing the computational complexity. In addition, as a widely used scientific and engineering numerical computation method, some numerical computation tools include FFT for GPU devices, which can provide further speedups. FFT convolution can reduce the memory overhead when dealing with small-sized convolution kernels using the Winograd algorithm.

Weight sparsification

Sparsification of weights in convolutional neural networks can reduce the redundancy of convolutional kernels and reduce the computational complexity, using a construction known

as Sparse Convolutional Neural Networks. In the learning of ImageNet data, a convolutional neural network sparsified at a ratio of 90 % runs two to ten times faster than a conventional convolutional neural network of the same structure, with only a 2% loss in classification accuracy of the output.

2.2 YOLO architectures

Information regarding the previous versions of the YOLO algorithm are listed on the appendices.

2.2.1 YOLO v3

This version has the introduction of **Feature Pyramid Networks** (FPN): 3 predictions are made from the input in different scales. The algorithm uses the **Darknet-53** structure, which has 53 convolutional layers used for prediction. The structure of the layer is presented in the fig. 7 below:

	Type	Filters	Size	Output
	Convolutional	32	3×3	256×256
	Convolutional	64	$3 \times 3 / 2$	128×128
1x	Convolutional	32	1×1	
	Convolutional	64	3×3	
	Residual			128×128
	Convolutional	128	$3 \times 3 / 2$	64×64
2x	Convolutional	64	1×1	
	Convolutional	128	3×3	
	Residual			64×64
	Convolutional	256	$3 \times 3 / 2$	32×32
8x	Convolutional	128	1×1	
	Convolutional	256	3×3	
	Residual			32×32
	Convolutional	512	$3 \times 3 / 2$	16×16
8x	Convolutional	256	1×1	
	Convolutional	512	3×3	
	Residual			16×16
	Convolutional	1024	$3 \times 3 / 2$	8×8
4x	Convolutional	512	1×1	
	Convolutional	1024	3×3	
	Residual			8×8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 7: Layers of the Darknet-53 structure – [3].

The backbone part evolved from Darknet-19 (YOLO v2) to Darknet-53, deepening the number of network layers and introducing cross-layer summation operations in Resnet. The original article lists the comparison of Darknet-53 with other networks.

2.2.2 YOLO v4

2.2.2.1 Main changes

Among the several programming libraries used for the development of software and solutions in computer vision, the YOLO v4 algorithm [28] – launched in April, 2020 – exhibits the greatest advancement in both accuracy, energy consumption and speed. The algorithm is the 4th version of the YOLO algorithm idea initially developed in 2016 [91]. The YOLO v4 version is considered the state of the art in image identification. It is a technology that presents greater precision when processing classification and location of objects in real time. The use of faster technologies such as embedded processing could elevate the performance levels to even more notable features. The study described in this text tests and compares the different characteristics of the YOLO v4 running on different processors.

The overall detection idea of YOLO v4 is not very different from YOLO v3, both use three feature layers for classification and regression prediction. New features included in this algorithm are:

1. Backbone feature extraction network: DarkNet53 => CSPDarkNet53;
2. Feature Pyramid: Spatial Pyramid Pooling (SPP), Pyramid Attention Network (PAN);
3. Classification regression layer: YOLO v3 (unchanged);
4. Training tips used: Mosaic data enhancement, Label Smoothing, CIOU, learning rate cosine annealing decay;
5. Activation function: use of the **Mish** activation function.

2.2.2.2 Structure – Backbone of backbone feature extraction network

YOLO v4 operates using a 416x416 pixel or a 608x608 pixel image, below the feature structure for the latter is presented:

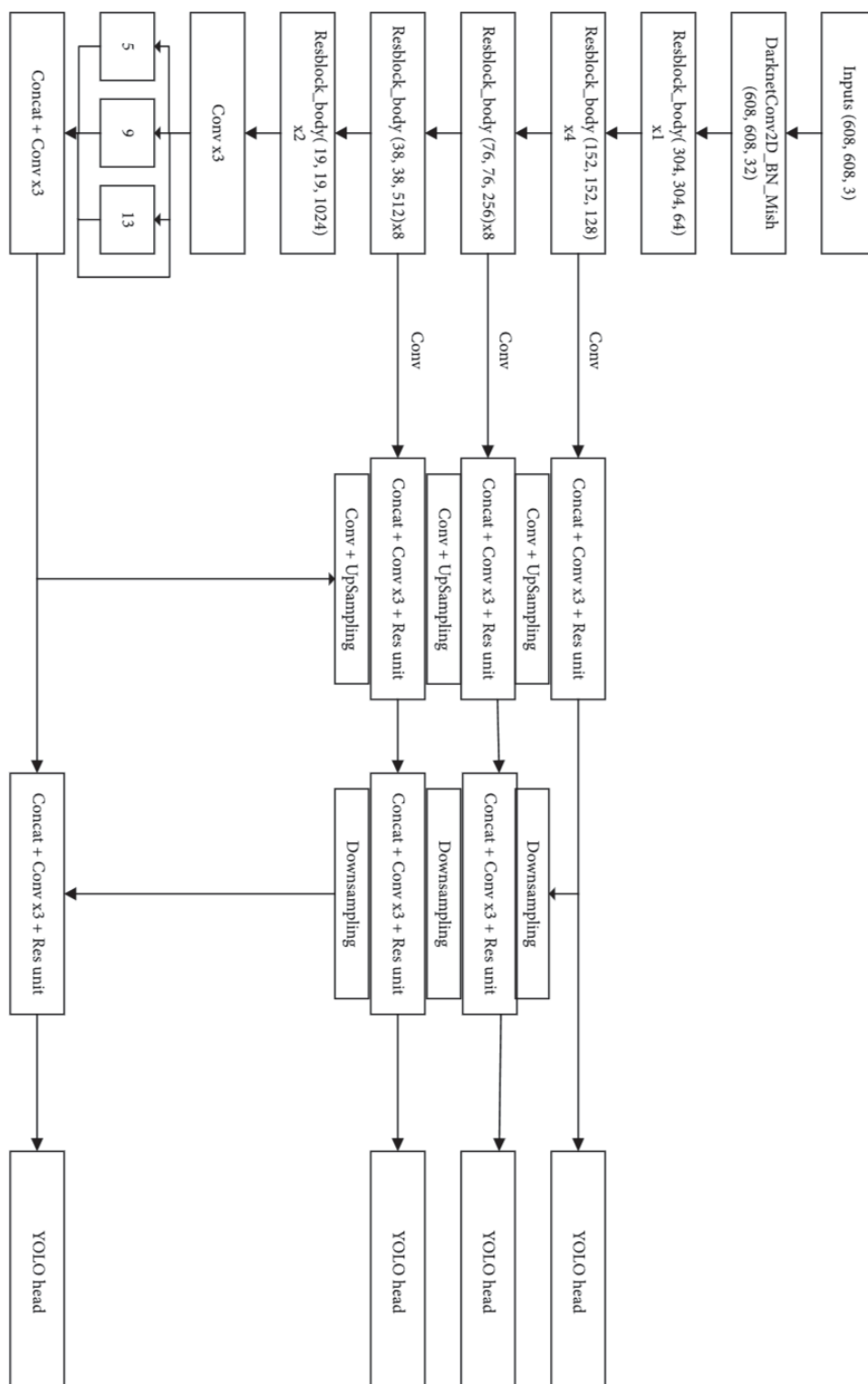


Figure 8: Feature structure for a 608x608 pixel input – [4].

There are two improvement points for the backbone feature extraction network.

- Trunk feature extraction network: DarkNet53 to CSPDarkNet53;
- Activation function: The structure of Darknet53, consists of a series of residual network structures. In Darknet53, there is a resblock_body module, which consists of a stack of down-sampling and multiple residual structures, and Darknet53 is a combination of resblock_body modules.

In YOLO v4, some modifications have been made to this part:

One of them is that the activation function of DarknetConv2D is modified from LeakyReLU to Mish, and the convolution block is changed from DarknetConv2D_BN_Leaky to DarknetConv2D_BN_Mish.

An analytical description of the Mish function is presented below:

$$\text{Mish}(x) = x \cdot \tanh(\text{softplus}(x)) \quad (11)$$

while $\text{softplus}(x)$ is denoted as:

$$\text{softplus}(x) = \frac{1}{\beta} \cdot \log(1 + \exp(\beta \cdot x)) \quad (12)$$

β is a constant value used for calibration.

The second main alteration is the modification of the structure of resblock_body and use the CSPnet structure. At this time, Darknet53 in YOLO v4 is modified to CSPDarknet53.

2.2.2.3 Feature pyramid

In the feature pyramid section, YOLO v4 combines two improvements:

- The SPP structure is used;
- The PANet structure is used.

The structure of the feature pyramid is removed except for the structure of CSPDarknet53 and Yolo Head: the SPP structure is involved in the convolution of the last feature layer of CSPdarknet53. After three DarknetConv2D_BN_Leaky convolutions of the last feature layer of CSPdarknet53, they are processed using four different scales of maximum pooling, and the sizes of pooling kernels of maximum pooling are 13x13, 9x9, 5x5, 1x1 (1x1 i.e. no processing).

2.2.2.4 YoloHead

YoloHead uses the acquired features to make predictions

- In the feature utilization part, YOLO v4 extracts multiple feature layers for target detection, a total of three feature layers are extracted, which are located in the middle layer, lower middle layer, and bottom layer, and the three feature layers have a shape of (76,76,256), (38,38,512), and (19,19,1024) respectively;
- The output layer's shape are (19,19,75), (38,38,75), (76,76,75), the last dimension is 75 because the graph is based on Visual Object Classes (VOC) dataset, it has 20 kinds of classes, YOLO v4 only for each feature layer exists 3 a priori boxes, so the last dimension is 3x25;
- If the COCO training set is used, the classes are 80, the final dimension should be 255 = 3x85, and the three feature layer shapes are (19,19,255), (38,38,255), (76,76,255).

2.2.2.5 Decodification of predicted results

From the second step, we can obtain the prediction results of the three feature layers, shaped as (N,19,19,255), (N,38,38,255), and (N,76,76,255) data, corresponding to the position of the three prediction boxes on the grid of each image divided into 19x19, 38x38, and 76x76.

However, this prediction result does not correspond to the final prediction box position on the image; it still needs to be decoded to complete.

Here we should talk about the prediction principle of YOLO v3. The three feature layers of YOLO v3 divide the whole picture into 19x19, 38x38 and 76x76 grids, and each network point is responsible for the detection of one region.

We know that the prediction results of the feature layers correspond to the positions of the three prediction boxes, which we first reshape a bit, and the results are (N,19,19,3,85), (N,38,38,3,85), and (N,76,76,3,85).

The 85 in the last dimension contains 4+1+80, which represent x_offset, y_offset, h and w, confidence level, and classification result, respectively.

The decoding process of YOLO v3 is to add each grid point with its corresponding x_offset and y_offset, and the result after the addition is the center of the prediction frame. Then the length and width of the prediction frame are calculated using the combination of the prior frame and h and w. This gives the position of the whole prediction frame.

The final prediction structure is of course obtained and then subjected to score ranking and non-maximal suppression screening.

This part is the part common to all target detection. But this project is handled differently from the others. It discriminates for each class.

It consists of two steps:

- taking out each class score greater than `self.obj_threshold` box and score;
- Using the position of the box and the score to perform non-extreme suppression.

2.2.2.6 Drawing the bounding box

With the fourth step, we can get the position of the prediction boxes on the original image and these prediction boxes are filtered. These filtered boxes can be drawn directly on the image and the results can be obtained.

2.2.2.7 Bag of freebies

Bag of freebies refers to a method that only increases training costs without increasing inference costs to improve detection accuracy, it usually refers to data enhancement. It is included in appendix A.

2.2.2.8 Bag of specials

The bag of specials is a composition of special packages used to improve target detection accuracy by increasing the inference cost and changing the network structure. It mainly consists of three mechanisms: increased receptive field, attention development, and feature fusion. The detailed description of these mechanisms are presented in appendix B.

2.3 PoseNet

PoseNet– [92] is a real-time stance recognition system that can be used to recognize human poses in both images and videos. It operates in single-mode (single human posture detection) and multi-mode (multi-position detection) modes (Multiple humans pose detection). PoseNet is a deep learning TensorFlow model that allows you to estimate human position by identifying body components such as elbows, hips, wrists, knees, and ankles, and then combining these points to construct a skeletal structure of your stance – [5].

For human recognition and posture estimation, a bottom-up technique that is box-free is proposed. It consists of two steps: the detection of K keypoints and the grouping of those keypoints into person instances. The ground truth keypoints consists of 14 face and

body components. The COCO dataset was used by the authors to train the network in a supervised manner.

This algorithm aims to detect all visible keypoints belonging to any individual in the image in an instance-agnostic manner. A hybrid classification and regression technique in a single-person environment is used to build the algorithms. To predict the keypoint locations, heatmaps (one channel per keypoint) and offsets (two channels per keypoint for horizontal and vertical displacements) are created. Figure 9 demonstrates the instance segmentation and pose estimation modules used to construct the algorithm.

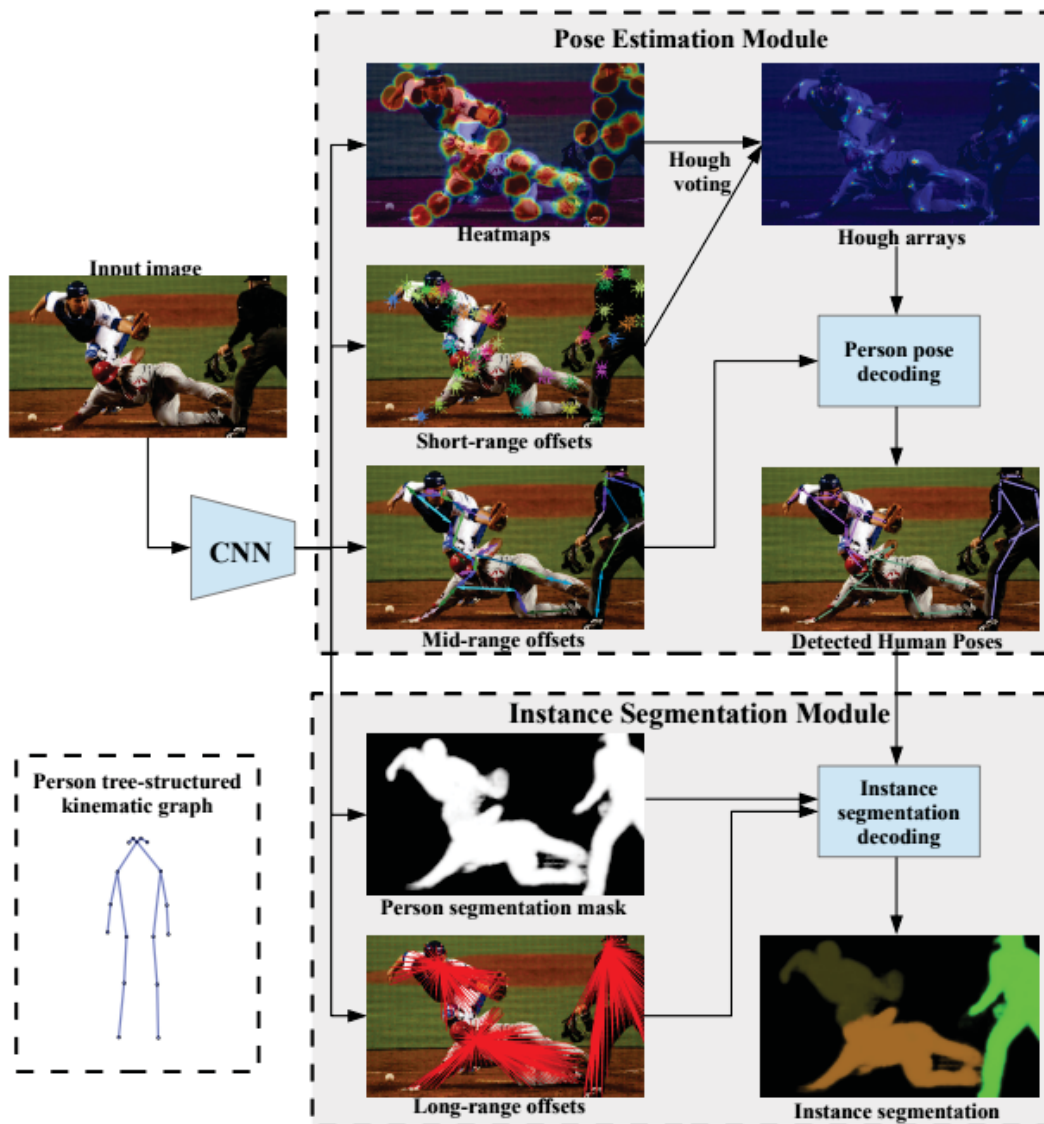


Figure 9: PoseNet modules – [5].

A hybrid classification and regression strategy is used, which is adapted to the multi-

person context. Heatmaps (one channel per keypoint) and offsets are created (two channels per keypoint for displacements in the horizontal and vertical directions). Let x_i denote the image's 2-D location, with $I = 1, \dots, N$ denoting the image's index and N denoting the number of pixels. Let $D_R(y) = \{x : \|x - y\| \leq R\}$ be a disk of radius R that is centered on y . Let $y_{j,k}$ be the 2-D location of the k -th keypoint of the j -th person instance, with $j = 1, \dots, M$, where M is the image's number of person instances. We built up a binary classification job as follows for each keypoint type $k = 1, \dots, K$. We forecast a heatmap $p_k(x)$ with $p_k(x) = 1$ for every person instance j if $x \in D_R(y_{j,k})$, otherwise $p_k(x) = 0$. As a result, we have K separate dense binary classification jobs, one for each type of keypoint. Each one entails forecasting a disk of radius R centered on a certain keypoint type of each individual in the image. For all of the tests in this work, the disk radius is fixed to $R = 32$ pixels and is independent of the person instance scale.

To ensure that all individual instances in the classification loss are equally weighed, we used a disk radius that does not scale with instance size. We calculate the heatmap loss as the average logistic loss along picture locations during training and back-propagate throughout the whole image, excluding only regions that include persons who have not been properly annotated with keypoints (person crowd areas and small scale person segments in the COCO dataset).

We additionally anticipate short-range offset vectors $S_k(x)$ to increase keypoint localization accuracy, in addition to the heatmaps. The short-range 2-D offset vector $S_k(x) = y_{j,k} - x$, k_x points from the image position x to the k -th keypoint of the nearest person instance j at each location x inside the keypoint disks and for each keypoint type k , as shown in fig. 1. We create K such vector fields by solving a 2-D regression problem individually for each picture location and keypoint. During training, we use the $L1$ loss to punish short-range offset prediction errors, averaging and back-propagating the errors solely at $x \in D_R(y_{j,k})$ in the keypoint disks. To normalize and make their dynamic range comparable with the heatmap classification loss, we split the errors in the short-range offsets (and all other regression tasks given in the research) by the radius $R = 32$ pixels.

Using distinct Hough accumulators for each keypoint type, we combine the heatmap and short-range offsets into 2-D Hough score maps $h_k(x)$, $k = 1, \dots, K$. Each picture position gives each keypoint channel k a vote with a weight equal to the chance of activation.

$$h_k(x) = \frac{1}{\pi R^2} \sum_{i=1:N} p_k(x_i) B(x_i + S_k(x_i) - x) \quad (13)$$

$B(\cdot)$ is the bilinear interpolation kernel.

2.3.1 Connection of keypoints for subjects

In order to make sure the detected keypoints stand to detect a human form, a minimum offset is defined to enable the formation of a uniform figure. With a minimum offset distance

defined, each keypoint can form discernible spaces for each of the body positions represented on the body. Pairs of keypoints are connected with the *mid-range* association offset.

The original authors have named a *recurrent offset refinement* valid for target members greatly distanced apart (this occurs typically for significant subjects). A bilinear interpolation to sample shorter ranges for the offset field was practical to implement the correction. This correction is also adapted into the software and hardware description languages implemented to conduct the research.

A fast greedy decoding technique was implemented to organize keypoints into recognized person instances. We start by creating a priority queue that is shared across all K keypoint types, into which we add the location and keypoint type of any local maxima in what the Hough score maps $h_k(x)$ (defined above) that have a score more significant than a specific threshold value (set to 0.01 in all reported experiments). These sites can be used to start a detection instance as candidate seeds. The components are subsequently removed from the queue in descending score order. In contrast to previous approaches that constantly employ the same keypoint type (e.g., Torso or Nose) as a seed for producing detections, our decoding algorithm does not regard any keypoint type preferentially. Although most detections in frontal facing person instances begin with the more easily localizable facial keypoints, the technique can also handle circumstances when a considerable section of the person is obscured.

The original authors have named a *recurrent offset refinement* valid for target members greatly distanced apart (this occurs typically for significant subjects). A bilinear interpolation to sample shorter ranges for the offset field was practical to implement the correction. This correction is also adapted into the software and hardware description languages implemented to conduct the research.

2.3.2 Keypoint position score

Various keypoints are detected at each frame, however, in order to ensure that what keypoint is the best at predicting the subject's members, a *keypoint position score* is defined. Specifically, consider a detected person instance j with keypoint coordinates $y_{j,k}$. j is the square root of the area of the bounding box tightly containing all detected keypoints of the j - th person instance. We define the Expected-OKS score for the k - th keypoint by

$$s_{j,k} = E\{OKS_{j,k}\} = p_k(y_{j,k}) \int_{x \in D_R(y_{j,k})} \tilde{h}_k(x) \exp\left(\frac{-(x - y_{j,k})^2}{2\lambda_j^2 \kappa_k^2}\right) dx \quad (14)$$

the new $\tilde{h}_k(x)$, is the Hough score normalized. Given the presence of the keypoint, the predicted OKS keypoint-level score is the product of our confidence that the keypoint is present multiplied by the OKS localization accuracy confidence.

As an instance-level score, we take the average of the keypoint scores, $s_j^h = \frac{1}{K} \sum Pksj,k$, followed by non-maximum suppression (NMS). We've tried both hard OKS-based NMS and a soft-NMS scheme suited for keypoints jobs, in which we utilize the sum of the scores of the keypoints that haven't been claimed by higher scoring instances as the final instance-level score, normalized by the total number of keypoints:

$$s_j = \frac{1}{K} \sum_{k=1:K} s_{j,k} [||y_{j,k} - y_{j',k}|| > \epsilon, \text{ for every } j' < j], \quad (15)$$

2.4 OpenPose

The OpenPose algorithm – [29] is the state of the art algorithm used for 2D keypoint detection. It is a real-time multi-person human pose recognition library that has successfully detected the human body, foot, hand, and facial keypoints on single photos for the first time. A total of 135 keypoints may be detected by OpenPose. The approach won the COCO 2016 Keypoints Challenge and is well-known for its good quality and adaptability to multi-person scenarios.

Using the first few layers, the OpenPose library extracts characteristics from a photo. The collected characteristics are then sent into two convolutional network layer divisions that run in parallel. The first segment forecasts a series of 18 confidence maps, each of which represents a different portion of the human posture skeleton. The following branch predicts 38 Part Affinity Fields (PAFs), which indicate the degree of affinity between components. The latter steps are used to clean up the branches' forecasts. Bipartite graphs are created between pairs of parts using confidence maps. Weaker linkages in bipartite networks are trimmed using PAF values.

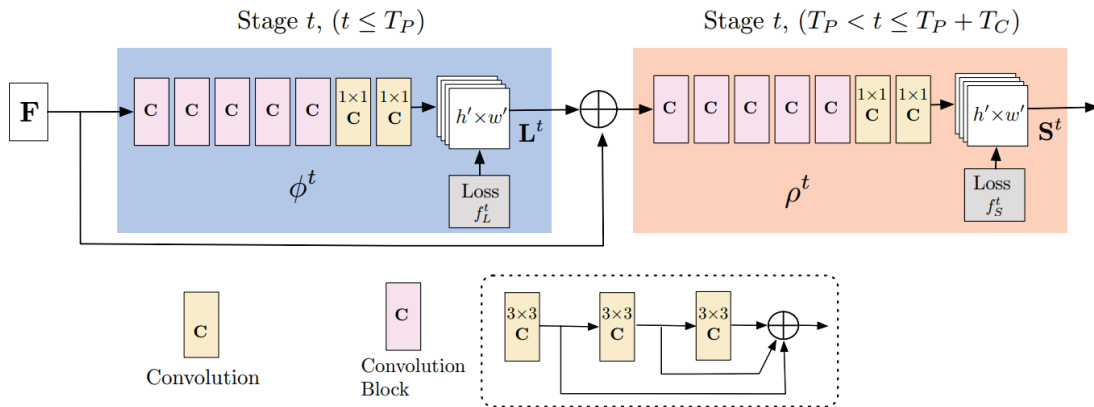


Figure 10: Architecture of the OpenPose CNN.

The architecture, illustrated in fig. 10, predicts affinity fields that encode part-to-part

connection (blue) and detection confidence maps (beige) repeatedly. Following, the iterative prediction architecture refines predictions through consecutive phases, $t \in \{1, \dots, T\}$, with intermediate supervision at each level. In comparison to PoseNet, the network depth has been enhanced. The network design of the initial technique includes numerous 7×7 convolutional layers. By replacing each 7×7 convolutional kernel with three sequential 3×3 kernels in the existing model, the receptive field is kept while the computation is lowered. While the number of operations for the former is $2 \times 7^2 = 92$, the later only has 51. In addition, the output of each of the three convolutional kernels is concatenated, similar to DenseNet's technique. The network's number of non-linearity layers is tripled, and it can maintain both lower and higher level characteristics.

2.5 License plate detection

License plate detection methods are listed in table 1 were reviewed to analyze what is the most efficient method compatible with hardware implementation.

2.5.1 License plate recognition with multiple stages

There are two types of ALPR systems now available: multistage – [93], [94] and single-stage approaches – [95]. The multi-stage strategy, which comprises three primary phases, has been considered in most existing ALPR solutions. The detection or extraction of license plates is the first stage. Existing systems employ classic computer vision techniques and deep learning approaches with object detection to find the license plate in a picture. Traditional computer vision approaches are mostly dependent on the license plate's shape, color, symmetry, and texture. The license plate is segmented in the second step, and the characters are retrieved using methods such as mathematical morphology, linked components, relaxation labeling, and vertical and horizontal projection.

However, since some segmentation-free algorithms skip this step, not every multi-stage ALPR system performs the character segmentation stage. Instead, pattern matching techniques or classifiers such as neural networks and fuzzy classifiers are used to recognize the characters in the final stage. Separating detection from recognition, on the other hand, has a negative influence on the whole recognition process' accuracy and efficiency. This occurs primarily due to defects in the detection process, such as bounding box prediction problems. For example, if the detection method misses a portion of a license plate, the overall accuracy of the identification process would suffer. As a result, achieving satisfactory outcomes at each level with a multi-stage method is critical.

2.5.2 License plate recognition with a single stage

While most previous license plate recognition research has focused on multi-stage systems, there have lately been many successful single-stage attempts. To the best of our knowledge, all of these approaches employ a single deep neural network trained to detect, localize, and recognize the license plate from beginning to end in a single forward pass. Object detection may be a special instance of license plate recognition. Like single-stage object detectors, these models may take advantage of the strong correlation between license plate detection and identification. This enables models to share parameters and needs fewer parameters than a traditional two-stage model. As a result, they may be more efficient and quicker than a two-stage process.

As a feature extractor, they employed VGG16 [85], which is a convolutional neural network model. Because the license plate occupies a smaller area in a typical photograph, they adjusted VGG16 to employ only two pooling layers instead of five. The feature extractors' output is then sent into a Region Proposal Network (RPN). They have made several tweaks, such as using two rectangular convolution filters instead of the standard three. This is to make use of the fact that license plates have a more significant aspect ratio than square filters. Therefore rectangle filters function better. After that, these filters' retrieved local characteristics were concatenated to maintain local and contextual information for the license plate categorization step.

2.5.2.1 Single stage detector implementation with a Multiprocessor systems-on-chip FPGA

Implementations of a single stage detector using individual features such as character segmentation [6] and image binarization and Region of interest related plate identification [96]. Figure 11 demonstrates the block diagram.

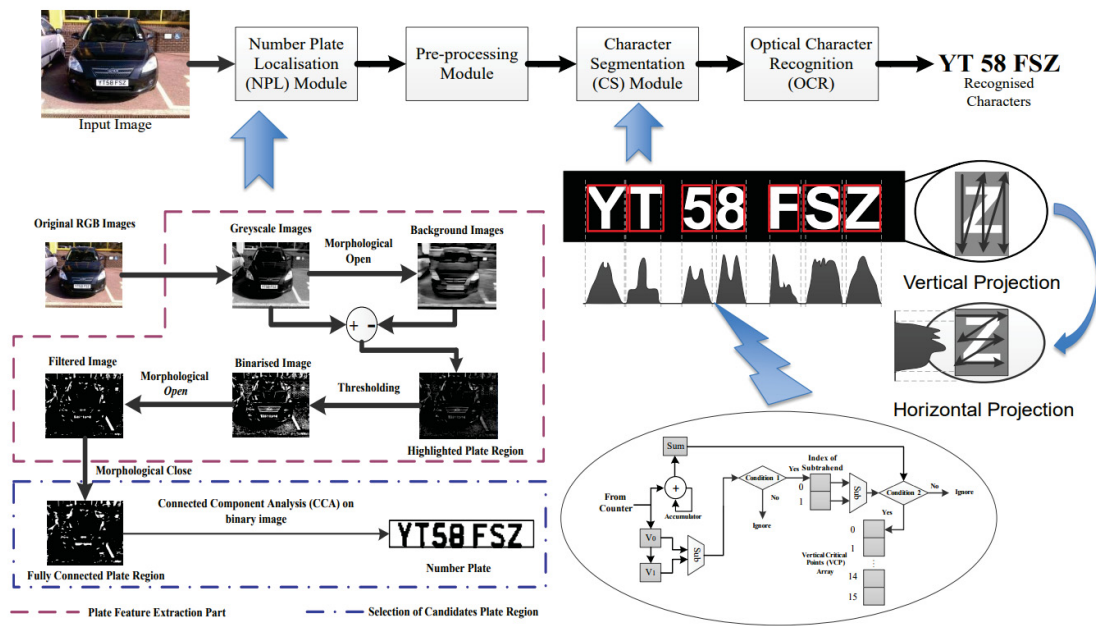


Figure 11: Block diagram used to represent the data flow for [6].

2.5.3 License plate detection using YOLO

The work presented in [7] uses the YOLO v3 algorithm in a multi stage configuration to detect the license plates in a similar manner used in the work presented on [97]. For both works, the first stage consists on identifying a higher area of the image in which a person or vehicle is present, afterwards this cropped section is used to identify the license plate. [7] initially used the original COCO weights to start the training process by appropriating its capabilities to detect cars and other vehicles.

Batch size	64
Image size	608 x 608 pixels
Momentum	0.949
Decay	0.0005
Learning rate	0.001
No. of anchor boxes	18
No. of epochs	30000

Table 3: Hyperparameters used to train the license plate detector with the YOLO v4 algorithm from [21].

Table 3 presents the hyperparameters used for training the YOLO v4 algorithm for license plate detection. An example of the the license plate detection can be verified in fig. 12.



Figure 12: License plate detection using the YOLO algorithm – [7].

3 Methodology

The conversion of high-level abstraction languages to lower hardware description languages is made using bash, C, C++, python programming, and tools from the Vitis software created by Xilinx, Inc. The Vitis software acts as an interpreter to group-specific pre-made logic blocks into the FPGA when it satisfies high-level function structures. Parts of these blocks and custom-made logic structures are created to implement the functional and logical architecture. The implemented CNNs where: YOLO v3, YOLO v4, SSD, OpenPose, and PoseNet CNNs. The high level object is then compared with the hardware description language implementation. Figure13 shows the workflow for the proposed method:

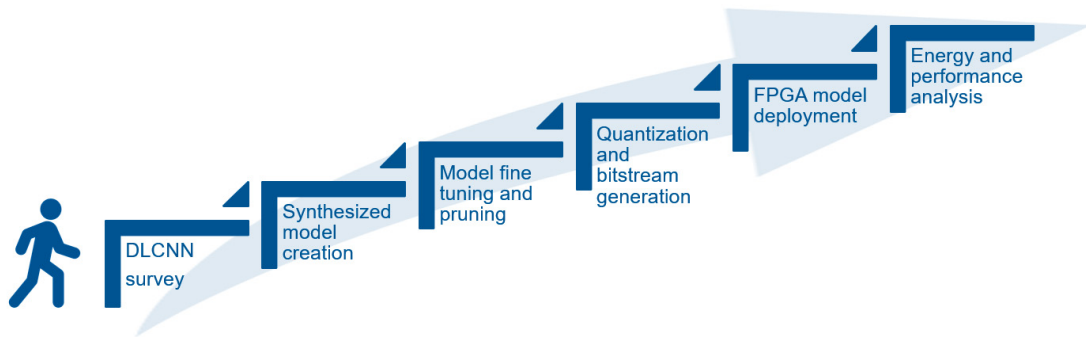


Figure 13: Workflow used for the development of the project.

The workflow was developed using a six-stage methodology:

1. Survey of the deep learning convolutional neural network;
2. Synthesis or model freezing;
3. Optimization;
4. Quantization;
5. Compilation;
6. Performance and energy evaluation.

Figure 14 demonstrates the main structure and flow of data within each stage. The progression in the workflow always goes from a higher level language to a lower level language.

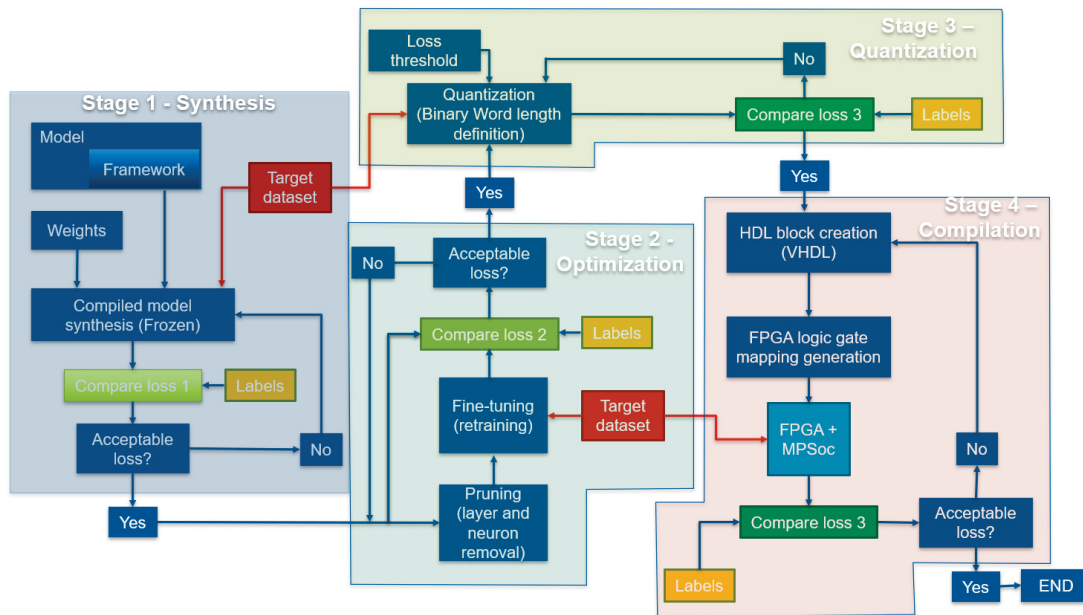


Figure 14: General structure of each stage.

3.1 Synthesis of models and frameworks

The synthesis stage consists of translating a specific model within a target framework into a "frozen" executable model which contains both the architecture and trained weights of a model. The trained weights of a model requires the use of a target dataset to establish parameters such as:

- Classes (object detection);
- Anchors (object detection);
- Number of epochs;
- Model configuration (dependant on the chosen algorithm for backbone, head, number of neurons definition);
- Cache of images.

The first stage models, frameworks and target datasets can be verified in figures 15 and 16 – respectively. The description of necessary steps different for each framework are presented in fig. 17.

Stage 1 – Models and frameworks

□ Identify target framework and model

- Scaled YOLO v4;
- Tensorflow framework;
- Weights trained for license plate detection

- Openpose
- Caffe framework
- 18 keypoints identification

- PoseNet
- Pytorch framework
- 14 keypoints detection

- SSD
- Tensorflow/Pytorch framework
- Person (full body) detection

 TensorFlow

 Caffe

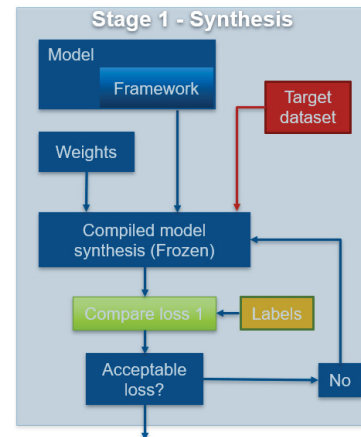
 PyTorch


Figure 15: Characteristics of the first stage.

Stage 1 – Target datasets

□ Target datasets

- THI license plate dataset – **YOLO V4**
 - 5779 images (80% training , 20% validation);
- HumanEval dataset – **PoseNet** (80% training, 20% validation)
 - 9000 images used for fine-tuning;
- COCO dataset – **Openpose and SSD** (80%, 20% validation)
 - 12000 images used for fine-tuning.
 - (Same images evaluating different characteristics.)

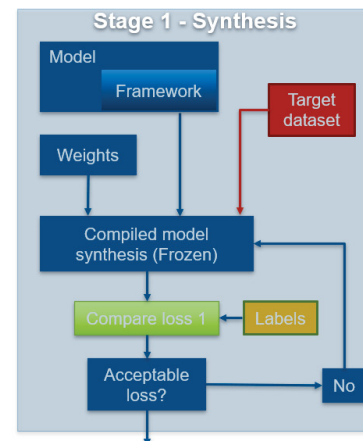


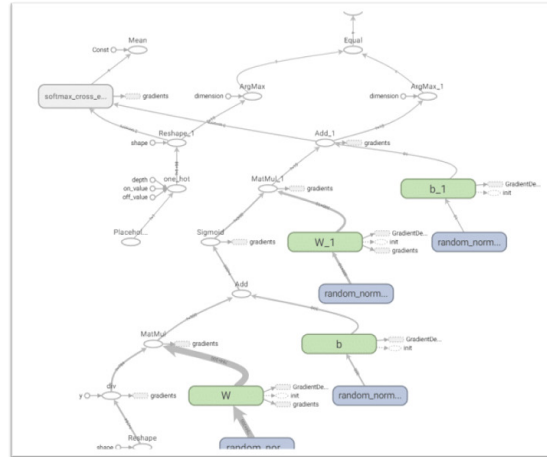
Figure 16: Datasets used for training each of the studied algorithms in the first stage.

Methodology

Stage 1 – Compiled model synthesis

- ❑ **Tensorflow** models requires conversion of the model and weights into a *frozen graph* file which contains both weights and layers into a single model.
- ❑ **Caffe** models use the layer definitions and weights separately for the creation of a compiled model, a *caffemodel file*.
- ❑ **Pytorch** models compile .pth layers and weight files into a *frozen graph*, which enters the next steps of the workflow in the same manner as tensorflow models.

Compiled model
synthesis (Frozen)



Representation of part of the layers of a compiled model created with the weight data.

Figure 17: Differences of the synthesized model for different frameworks.

3.1.1 Model conversion

Most of the models that are used in the workflow already have a working structure within the Caffe, Pytorch or TensorFlow frameworks. In order to utilize them along the Vitis software, data conversions are necessary.

Any model in TensorFlow can be implemented using Vitis. The required elements are: the frozen image of the model, the dataset and the bash activation functions.

- There are compatibility issues between the new versions of tensorflow (2.0>) and the creation of frozen graphs datafiles (necessary for further deployment in Vitis AI);
 - A workaroud is to get a .h5 file from a keras deployment and use a documented python script to convert it to the desired frozen graph;
- The desired test wrapper for YOLO v4 built for the DPU platform can be customized by editing a prebuilt YOLO v3 inside the Vitis AI library;
- All three elements (Frozen graph, dataset, activation wrapper) are necessary to do the Vitis compilation and quantization. After that, the YOLO V4 code will be ready for the board deployment. (No further work required for this stage);
- It is not possible to emulate the behavior of the YOLO v4 model using the Vitis AI software with the same performance obtained using the Vitis/Vivado HLS design tool. Designing requires the creation of a precise C/C++ description to implement a functional model of the YOLO v4 algorithm – described in section 3.4.1.

3.1.2 Frozen graph

Freezing of tensorflow, means to integrate and merge model files and weight files into one file. The main purpose is to facilitate deployment with parts of the Vitis software. The process has the subsequent steps:

1. Loads the model architecture;
2. Reads the weight data from the checkpoint file and initializes it to the weight variable in the model;
3. Converts the weight variables to weight constants (since the constants can be saved in the same file with the model and be later quantized to a fixed point constant);
4. Strips the Op node that is not used for output inference from the graph through the specified output node, and then saves it into the specified file.

In order to pass through the conversion step, the user must establish 11 parameters:

1. Input graph : (required) model file , which can be a binary pb file (tensorflow model) or a text meta file (other models). An "input binary" code is used to specify the distinction between models;
2. Input saver : (optional) Saver parser. When saving models and permissions, Saver can also serialize the save itself so that the appropriate version is applied on the loading step. Mainly used when versions are incompatible. Can be empty;
3. Input binary : (optional) used with the "Input graph" file, when true, "Input graph" is binary, when false, "Input graph" is a file;
4. Input checkpoint : (required) checkpoint data file. During training, it is used for the Saver to save variable values such as weights and biases. This is used for the model restoration of variable values;
5. Output node names : (required) the names of the output nodes, separated by commas if there are multiple. It is used to specify the output node, and other nodes that are not on the output line (which will be eliminated);
6. Restore op name : (Optional) The name of the node to restore from the model;
7. Filename tensor name : (optional) Name of the used tensor, can be any value;
8. Output graph : (required) to save the integrated model output file;
9. Clear devices : (optional) Specifies whether to clear the computing device specified by the node during training (such as CPU, GPU, TPU. CPU is the default);

10. Initializer nodes : (optional) After the permissions are loaded, this parameter is used to specify the nodes to be initialized and also to separate multiple node names with commas;
11. Variable names blacklist : (optional) Variable blacklist, which is used to specify variables that do not need to restore values. Separated by commas when using multiple variable names.

It should be noted that if the model file is in .meta format, that is to say, the meta model file generated with the saver, the established save method is modified by using "meta" functions in place of the frozen graph functions.

3.1.3 Confirming acceptable losses

The process of conversion is model dependent and often needs model architectural details to assure that losses happened on the process. As cited in the introduction, the process of comparing loss for the synthesized model requires the use of a target metric. For object detection, the mAP metric was used. For keypoint detection algorithms, 2D radial matching between the ideal position and the predicted position.

3.2 Optimization

The optimization is cutting parts of the architecture to make it less computationally complex. The end result should be implemented in CPUs, GPUs, TPUs and FPGAs. It starts by observing which parts of the model exercise less influence on the result to remove them progressively (fig. 18). The remaining parts of the network generally assume the functions of the less influential parts. The increase in speed comes at the expense of accuracy, retested here with the previously cited metrics.

Stage 2 – Optimization

- ❑ Optimization consists of pruning parts of the model;
- ❑ The approach eliminates layers and neurons;
- ❑ Fine-tuning uses the target dataset to verify which “prunable” elements can be eliminated without a significant impact.

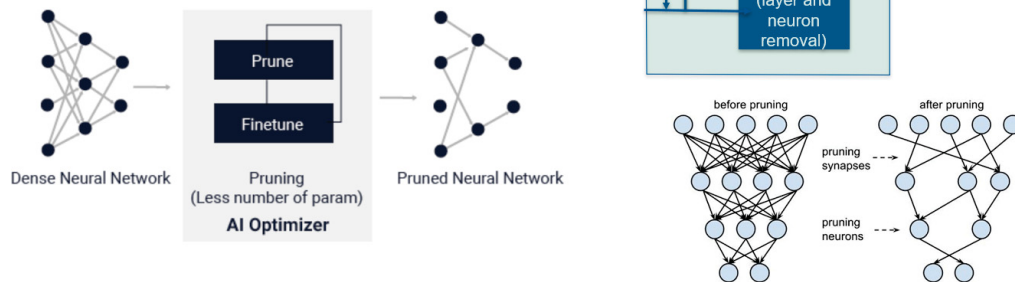


Figure 18: Overview of the optimization stage.

As indicated in the diagram of fig. 18, this is done in an iterative method. As observed in fig. 19 pruning reduces accuracy, but retraining restores it. One cycle consists of pruning and retraining. Typically, this technique is performed numerous times until the required sparse model is obtained, or the minimum accepted loss is reached, when it becomes difficult to rebuild the model.

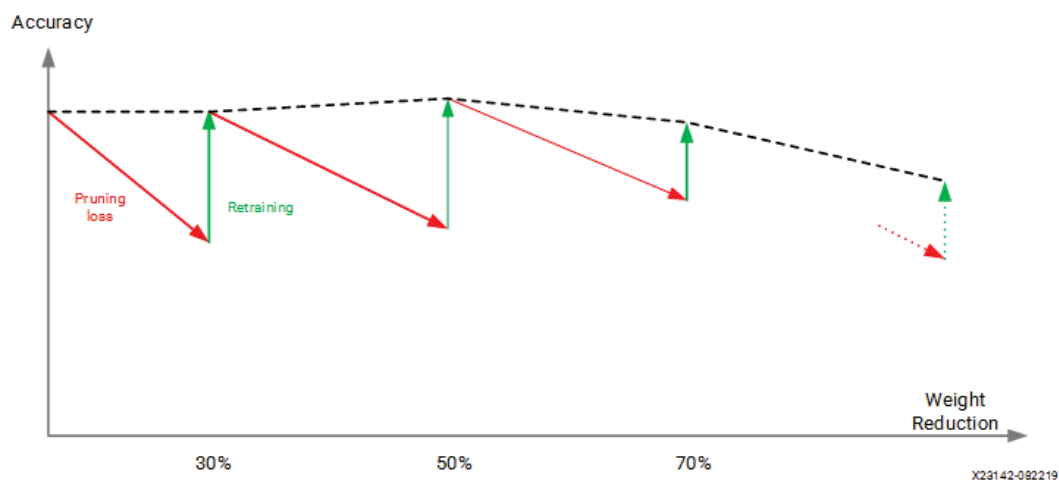


Figure 19: Relation between accuracy and model weight reduction (compression) – [8]

3.2.1 Optimization steps

The more to undertake a model analysis in this step, the better. At least half of the data from the initial training set should be used in the optimization to obtain beneficial results. In this work all of the data within the validation dataset was used for the tested models. The training process typically takes 70 h, applying an RTX and 2080 GPU. Parameters such as initial learning rate and the decline rate are necessary for the process. The best result for each iteration should be used as the input to the next round of pruning. If the precision does not increase after multiple fine-tuning attempts, the user should reduce the pruning rate and re-run pruning and fine-tuning.

3.2.1.1 Fine-tuning

The pruned model's performance has deteriorated to some extent, and you'll need to fine-tune it to enhance it. Except for the hype-parameters, such as the initial learning rate and the learning rate decay type, finetuning a pruned model is essentially the same as training a model from scratch. An iteration of pruning is completed when pruning and fine-tuning are accomplished. In general, the model must be pruned numerous times to attain a greater pruning rate without a noticeable loss of performance.

The hyperparameters used for retraining the model during the fine-tuning step are the same ones used when training the [7] weights for LP detection. These parameters can be seen in table 4.

Batch size	64
Image size	608 x 608 pixels
Momentum	0.949
Decay	0.0005
Learning rate	0.001
No. of anchor boxes	18
No. of epochs	30000

Table 4: Hyperparameters used to train the license plate detector with the YOLO v4 algorithm from [21].

3.3 Quantization

The quantizer transforms 32-bit floating-point weights and arithmetic to fixed-point variables like INT8 (fig. 20). Because the fixed-point network model uses less memory bandwidth, it is quicker and more energy efficient than the floating-point network model. The part is also fundamental to predict and prepare the model behavior for the FPGA device, which can only operate using fixed point arithmetic. The number of bits representation is reduced and supervised to not sacrifice prediction accuracy. Quantization allows for the use

of integer computing units and the representation of weights and activations using fewer bits, while pruning decreases the overall number of operations required. Again, the cited metrics are used for the word length evaluation, and precision loss.

Stage 3 – Quantization

- ❑ The quantization will run the algorithm over a defined number of iterations until a loss threshold is attained;
- ❑ The target dataset is used to verify which digits are most significant for each operation and layer

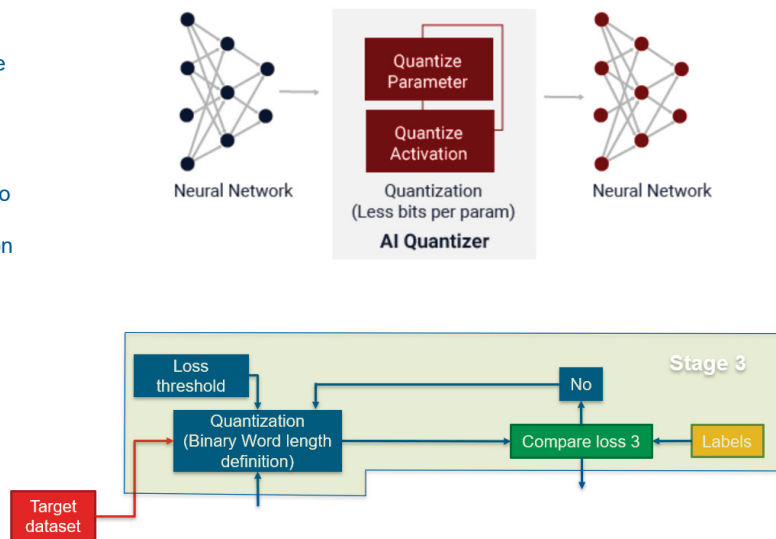


Figure 20: Overview of the quantization stage.

In the quantization process, only a small set of unlabeled images are required to analyze the distribution of activations, taking only a few seconds to several minutes. Generally, there is a little decline in accuracy after quantization. However, for some networks such as Mobilenets, the accuracy loss might be large. In this situation, quantize fine-tuning can be used to further improve the accuracy of quantized models. Quantize fine-tuning requires the original train dataset. According to experiments, several epochs of fine-tuning are needed and the finetune time varies.

After calibration, the quantized model is transformed into a DPU deployable model, which follows the data format of a DPU. This model can then be compiled by the Vitis AI compiler and deployed to the DPU.

3.3.1 Quantization mechanism

3.3.1.1 First sweep stage

The quantization process consists of running the optimized over the entire training and validation dataset and saving the mean accuracy for a reference word length. After the accuracy is calculated for a specific word length, the number of bits for both integer and

fractional word length are reduced. Figures 21 and 22 demonstrate how the sweep process is conducted for the first stage.

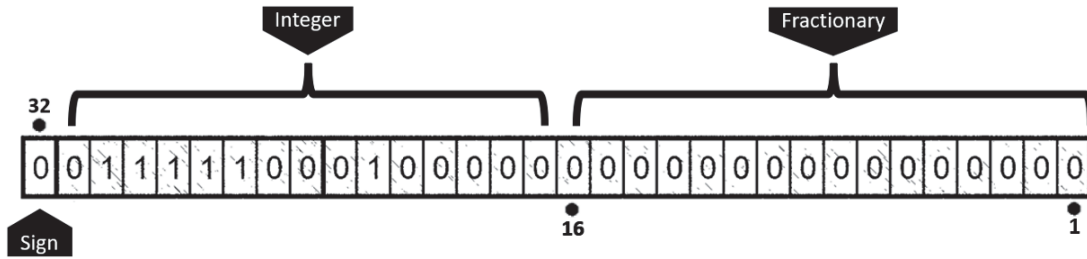


Figure 21: Demonstration of the initial 32 float variable used to calculate variables up to the optimization step.

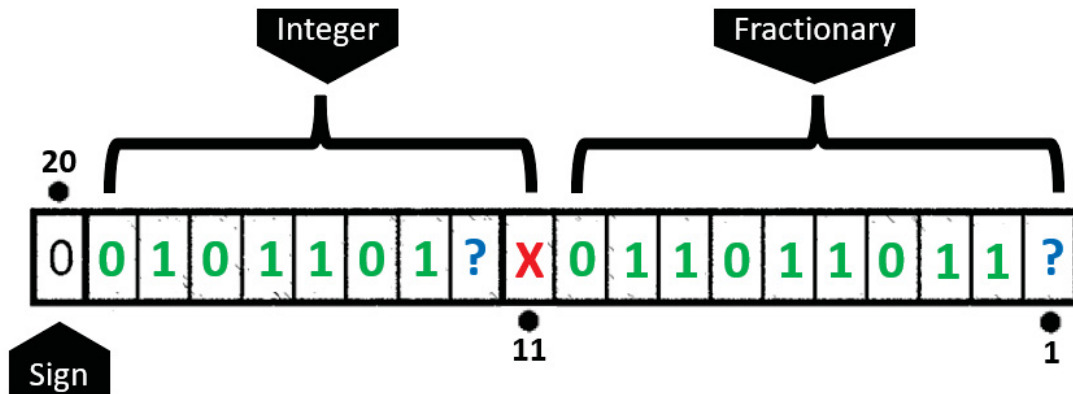


Figure 22: Example of the word length sweep on the uniform test for the word length. In the example diagram, the green numbers are the ones that have already been tested and found relevant to assert accuracy above the established threshold level. Blue characters are the ones that haven't been tested in the current iteration.

3.3.1.2 Second sweep stage

After the main uniform removal step is concluded, the second phase consists on randomly removing the bits that do not directly make the accuracy fall below the user defined level. Each combination is tested for the entire dataset. The second test is required to test whether the results obtained in the first step have a uniform accuracy decrease from the most significant to the least significant bit for both the fractional and integer parts.

The test used for both the Technische Hochschule Ingolstadt (THI) plate dataset and HumanEval dataset included 15 word lengths + signal bit to test. The 32 bit word length was impractical to calculate for the chosen datasets. Figure 23 demonstrates how the sweep process conducted for the second stage.

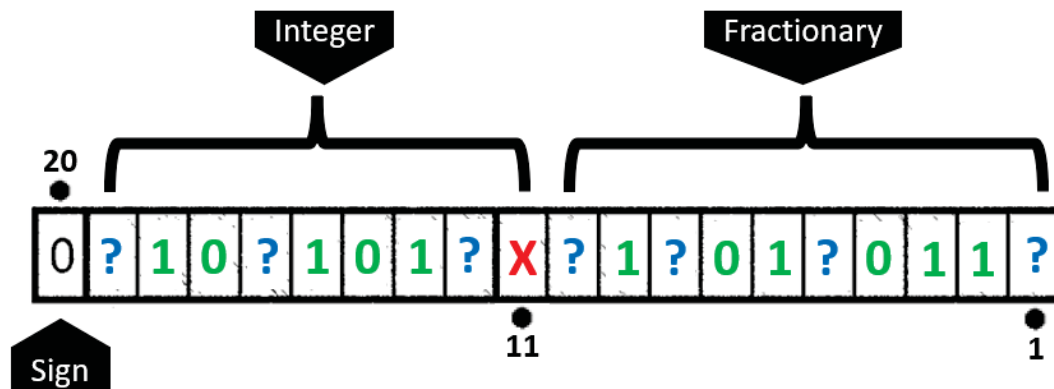


Figure 23: Example of the word length sweep on the random test for the word length. In the example diagram, the green numbers are the ones that have already been tested and found relevant to assert accuracy above the established threshold level. Blue characters are the ones that haven't been tested in the current iteration.

At the end of this stage, the aforementioned performance metrics are applied to the developed algorithms for the GPU and CPU platforms. The result of the quantization stage has the fixed-point arithmetic closest to the operation of FPGA devices. Further evaluation from the subsequent stage has HDL description parameters that the CPU and GPU cannot interpret.

3.4 Compilation

Stage 4 – Compilation

- The compilation step creates a Hardware Description Language (HDL) bitstream that will arrange the logic gates and DSPs into a specific logic circuit;
- If the target board is not large enough to accommodate the resources, the code is modified over the other stages or customized (VHDL)

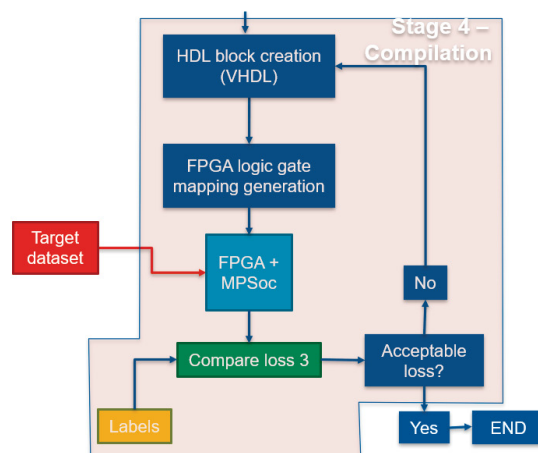


Figure 24: Overview of the compilation stage.

The compilation step represented in fig. 24 consists on converting the quantized model into the HDL language/bitstream which is used to build the logical circuit inside the FPGA, alongside look up tables to store weight data. The process has two non-exclusive sub-stages: HDL High Level Synthesis (HDLHLS) for edition and framework based inference for immediate deployment.

3.4.1 HDL High Level Synthesis

For the implementation of the V4 algorithm, the Vitis High Level Synthesis (HLS) design tool is used. The high level synthesis platform enables the user to visualize and edit each of the C/C++ and OpenGL architectural blocks. In order to visualize the C/C++ source, the user must extract the model created after the optimization stage to verify which layers are removed.

- The main need to use the High level synthesis (HLS) platform is to emulate the code behavior in a generic FPGA device structure for *industrial application deployment*. It will be possible to analyse the application speed requirement and deploy a cost effective solution. In this manner, solutions can make use of smaller programmable only FPGAs that don't rely on DPU systems;
- For comparison purposes, we will be able to see the differences of **emulation results** and **real deployment** by utilizing the ZCU104 board;
- Even though the pruned algorithm has to pass through another development step in Vitis HLS, after that step - it can be deployed easily in smaller FPGAs for benchmarking or cheaper applications.

The Vitis AI docker is a packaged Linux system with specific libraries created to run the processor + FPGA processing. The entire interaction can be reconstructed in Vitis HLS (software to hardware synthesis) + Vivado (hardware design). To correctly recreate the hardware device for the first simulation steps, it was fundamental to:

1. Verify which libraries are used in the Vitis AI framework and what are their requirements;
2. Integrate those libraries into the custom-built Petalinux system (Xilinx already supplies a base custom image for development);
3. The custom-built Linux image then is loaded with the previously used codes to synthesize a hardware image (emulation and validation is done in this step);
4. Execute the entire operation on the Vivado hardware analyzer to verify the power consumption.

Figure 25 shows the workflow for this step.

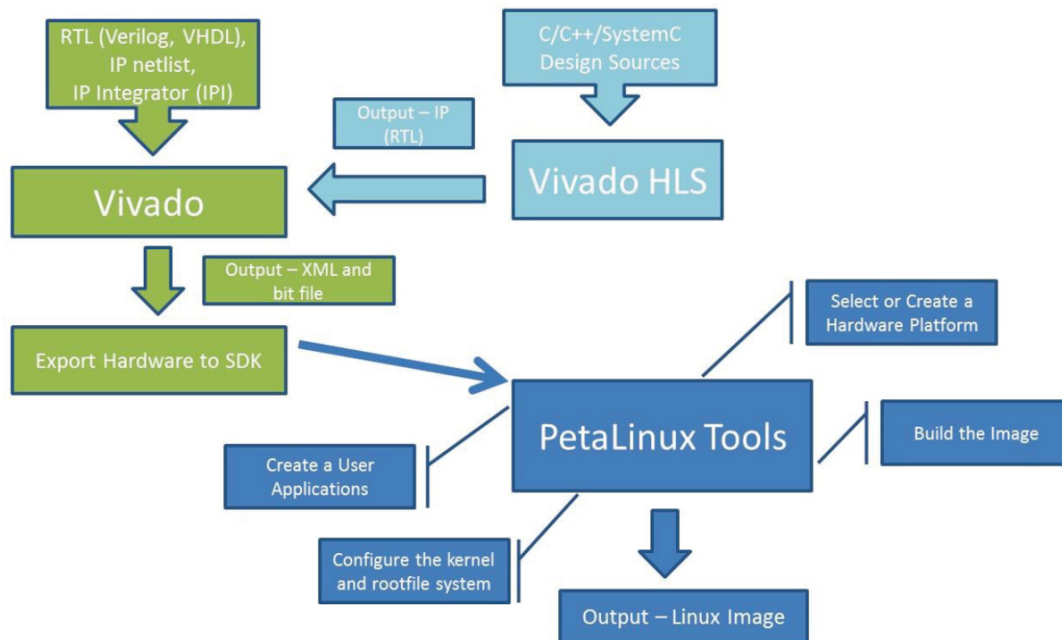


Figure 25: Hardware validation structure workflow.

3.4.1.1 HLS structure

The tool uses an Integrated Development Environment (IDE) interface to separate and organize how the target (ZCU104) board will execute each of the instructions written in C++ and how the kernels will be activated alongside the main operating system (OS) – custom Linux system. It should be noted that the process of "hardware replication" is complex and must be carefully projected by the user for each CNN on the acceleration process to optimize resource utilization and, therefore, can not be automated.

Similarly on how the validation was executed in the previous steps, an emulated simulation using three distinct steps is executed:

1. **Emulation SW:** The kernel and C++ peripherals are executed using the computer OS (Linux 20.04 LTS – Long Term Support).
2. **Emulation HW:** The kernel and C++ peripherals created emulated PetaLinux OS, which is responsible for emulating the interface between the FPGA device and the board processors; **Hardware:** If an interface is programmed to interact with the main OS, it is possible to use the hardware function to test the code directly using the IDE.

If the code were tested successfully in the first two steps, a kernel would be ready to be loaded alongside the full OS inside the MpSoC+FPGA board to operate using wrapper functions.



Figure 26: Steps executed inside the HLS interface to emulate and deploy the code.

3.4.2 Wrapper for custom models

A wrapper function written in python is used to call the functions of a built model already present in the Vitis AI library or to activate a custom-built function.

The built wrappers used to implement the YOLO v4 algorithm were based on the YOLO v3 wrappers displayed on the *AI Model Zoo* – [98]. These wrappers usually are bash and python commands built to work specifically with the VITIS AI system.

3.4.3 Vitis AI inference engine

Inference in machine learning is computation-intensive and requires high memory bandwidth to meet various applications' low-latency and high-throughput requirements.

3.4.3.1 Conversion between darknet model to caffe

The tool works with three main frameworks: Tensorflow, Caffe, and Pytorch. Conversion from a darknet model (YOLO framework) is possible using prepared conversion scripts

in the "scripts folder".

3.4.4 Framework based inference with XIR

The quantized model can be swiftly converted into a bitstream to program the FPGA device if it is based on Tensorflow, Caffe, Pytorch, Neptune, or Darknet frameworks and can be implemented by the subsequent operators:

- Convolution with a 1 - 16 kernel size;
- Convolution stride equal or lesser than 64;
- Padding for 0 to 15;
- Dilated convolution;
- Max/Avg pooling with kernel size from 1x1 to 256x256 size;
- Pooling stride from 1 to 8;
- Pooling padding to 0 to 255;
- ReLU;
- ReLU6;
- Leaky ReLU;
- Concatenation;
- Split operation;
- Elementwise;
- Mean scale;
- Deconvolution;
- Depthwise convolution;
- Batch normalization;
- Reorganize;
- Hard-sigmoid;
- Hard-swish;
- 3D convolution;
- 3D transposed convolution;

- 3D upsampling;
- 3D element wise sum.

The conversion process uses the Vitis AI compiler to generate the bitstream. The compiler creates an internal computation graph as an intermediate representation after parsing the topology of the optimized and quantized input model. As a result, a control flow and a data flow representation are required. It then implements a variety of optimizations, such as compute node fusion, such as when a batch norm is fused into a presiding convolution, efficient instruction scheduling by taking use of intrinsic parallelism, and data reuse. It should be noted that this method is limited to Data Processing Units with the available resources to accept the generated bitstream.

The Xilinx Intermediate Representation (XIR) is a graph-based intermediate representation of AI algorithms that is used to compile and deploy the DPU on the FPGA platform. By expanding the XIR to accommodate bespoke IPs in the Vitis tool flow, sophisticated users may apply entire application acceleration to allow the FPGA to be exploited to its full capacity. The Vitis AI quantizer, compiler, runtime, and other tools are all built on top of it. The subsequent fig. 27 was extracted from the Vitis manual to show how the XIR flow:

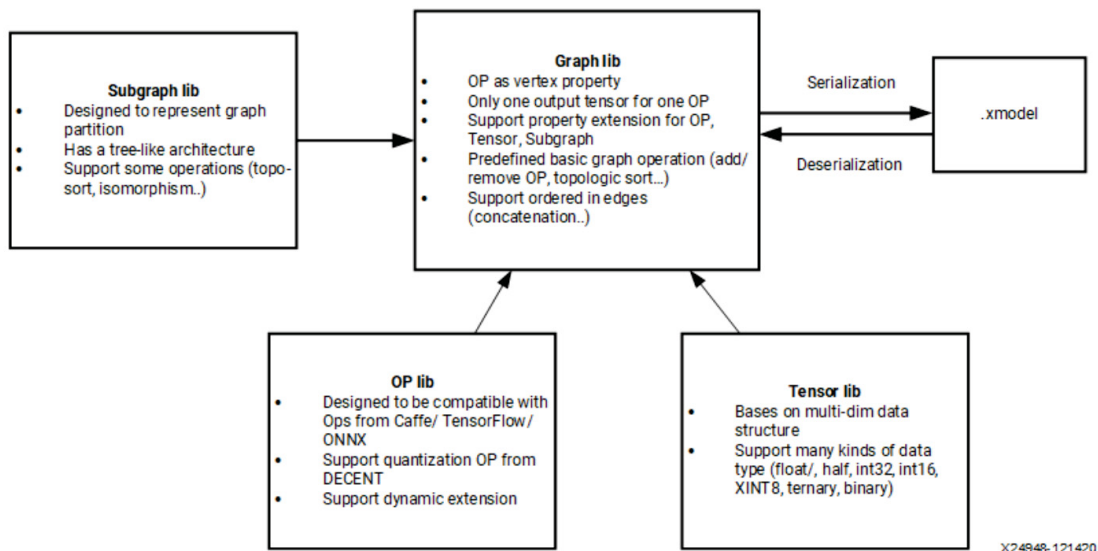


Figure 27: XIR workflow.

The components of the XIR workflow are:

- **Graph:** The XIR's primary component is the graph. It acquires two main functions:
 - Graph model serialization;
 - Graph model deserialization;

The Graph is similar to a container, with the Op as the vertex and the producer-consumer relationship as the edge;

- **Op:** In XIR, Op is an instance of an operator definition that is either in XIR or has been expanded from XIR. The Graph can only generate or add Op instances that are defined in the built-in/extended op definition library. The input parameters and intrinsic properties are mostly included in the Op definition. Aside from the intrinsic preset characteristics, the `xir::Op::set` attr Application Programming Interface (API) allows an Op instance to carry extra extrinsic attributes. Each Op instance can only get one output tensor, although several fanout operations can be obtained;
- **Tensor:** Tensor is another important class in XIR. Unlike other frameworks' tensor definition, XIR's Tensor is only a description of the data block it represents. The real data block is excluded from the Tensor. The key attributes for Tensor is the data type and shape;
- **Subgraph:** XIR's Subgraph is a tree-like hierarchy, which divides a set of ops into several non-overlapping sets. The Graph's entire op set can be seen as the root. The Subgraph can be nested but it must be non-overlapping. The nested insiders must be the children of the outer one.

The quantized TensorFlow or Caffe model is sent into the XIR-based compiler. It begins by converting the input models to the XIR format, which serves as the foundation for the subsequent procedures. In XIR, the majority of the differences across frameworks are erased and converted to a single representation. Then, depending on whether the operation can be performed on the DPU, it applies different optimizations to the graph and divides it into many subgraphs. As needed, architecture-aware optimizations are done to each subgraph. The compiler creates and attaches to the instruction stream for the DPU subgraph. Finally, the optimized graph is serialized into a constructed xmodel file with all of the essential information and instructions for Vitis AI Runtime (VART).

The DPUCZDX8G series on the Edge Zynq UltraScale+ MPSoC platforms, DPU-CADF8H on the Alveo platform, DPUCAHX8H on the Alveo HBM platform optimized for high-throughput applications, DPUCVDX8G on the Versal Edge platform, and DPUCVDX8H on the Versal Cloud platform can all be supported by the XIR-based compiler. The same steps apply to compiling Caffe or TensorFlow models using VAI C as they did for the prior DPUs. It's expected that you've installed the Vitis AI package, including VAI C, and that you've used the vai quantizer to compress your model.

The framework-independent XIR graph created by deep learning frameworks is used by the XIR-based compiler. The parser strips the CNN models of framework-specific properties and converts them into XIR-based compute graphs. The compiler separates the computational graph into subgraphs, takes use of heterogeneous optimizations, and outputs subgraph-specific optimized machine code as seen in fig. 28.

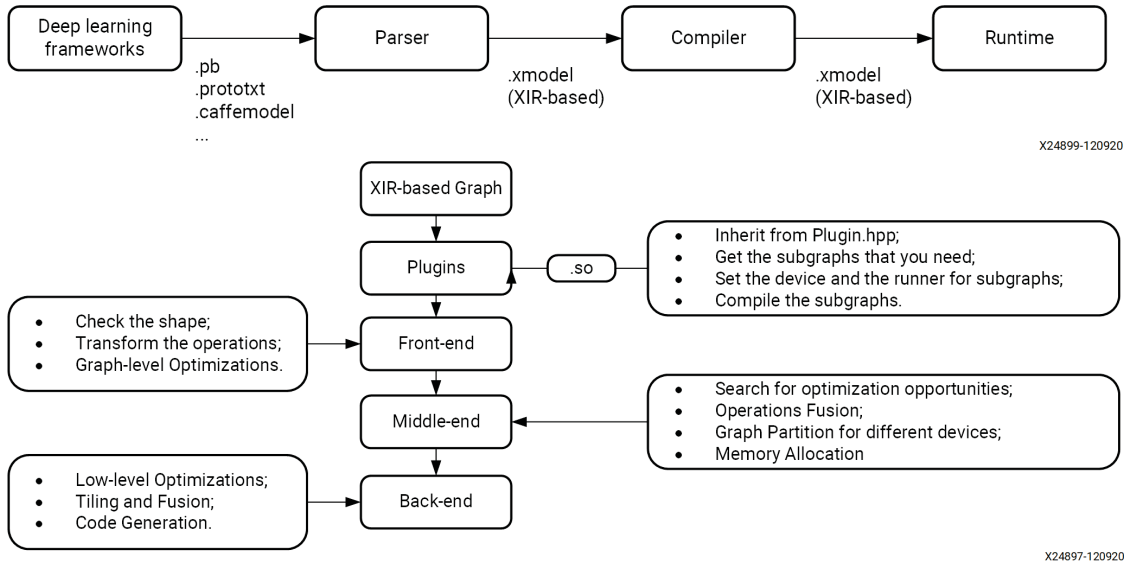


Figure 28: Compilation stage workflow.

Some subgraphs (operations) are formed and assigned to the CPU when the model contains operations that the DPU cannot handle and are therefore executed by the quad-core ARM Cortex-A53 applications processor. It's feasible to construct a custom IP to speed up such processes for better end-to-end performance. Use a pipeline called plugin to expand the XIR and compiler to enable customizable accelerated IPs with an XIR-based toolchain.

The interface class Plugin is specified in the plugin kernel. Before the compiler starts compiling the graph for the DPU, the plugins are performed in order. For each operator, a child subgraph is formed first, and the plugin selects the operators it can accelerate. It combines them into bigger subgraphs, maps them to the specific IP, and includes runtime metadata such as the subgraph instructions.

3.4.4.1 Specifics for object detection algorithms

The SSD and YOLO v4 algorithms tested for the COCO dataset (pedestrian identification) and THI plate dataset – respectively – can be implemented using all the original convolutional and pooling layers. YOLO v4 had the weights for car plate identification adjusted by the works executed in [7]. While SSD has the weights adjusted for the full-body person (pedestrian) class based on [99].

The entire model is condensed into a `.xmodel` file loaded in the FPGA device. Afterward, C++ wrappers are used to control the inputs and outputs of each specific DLCNN to be used. In YOLO v4 and SSD cases, only the convolutional and pooling layers are executed. In order to visually show the bounding boxes to the viewer, the OpenCV library was used.

However, OpenCV was executed using the four onboard CPUs on the MPSoC FPGA to reduce development time.

3.4.4.2 Specifics for keypoint detection algorithms

PoseNet and OpenPose differ in their ability to detect multiple people; for this reason, these networks differ in their complexity in terms of the number of convolutional and pooling layers. Similar to the object detection algorithms, a condensed .xmodel is created to test each developed algorithm.

3.4.4.3 Resource estimation

3.4.4.3.1 HLS based architecture

Designs using the high level synthesis can have their resources estimated by the Vivado tool, where the developed VHDL code is translated to specific Register-Transfer Level (RTL) circuit. The process follows the synthesis, which consists of transforming an RTL-specified design into a gate-level representation. Fig 29 represents a register with its buffers for one of the filter values used for the convolution process.

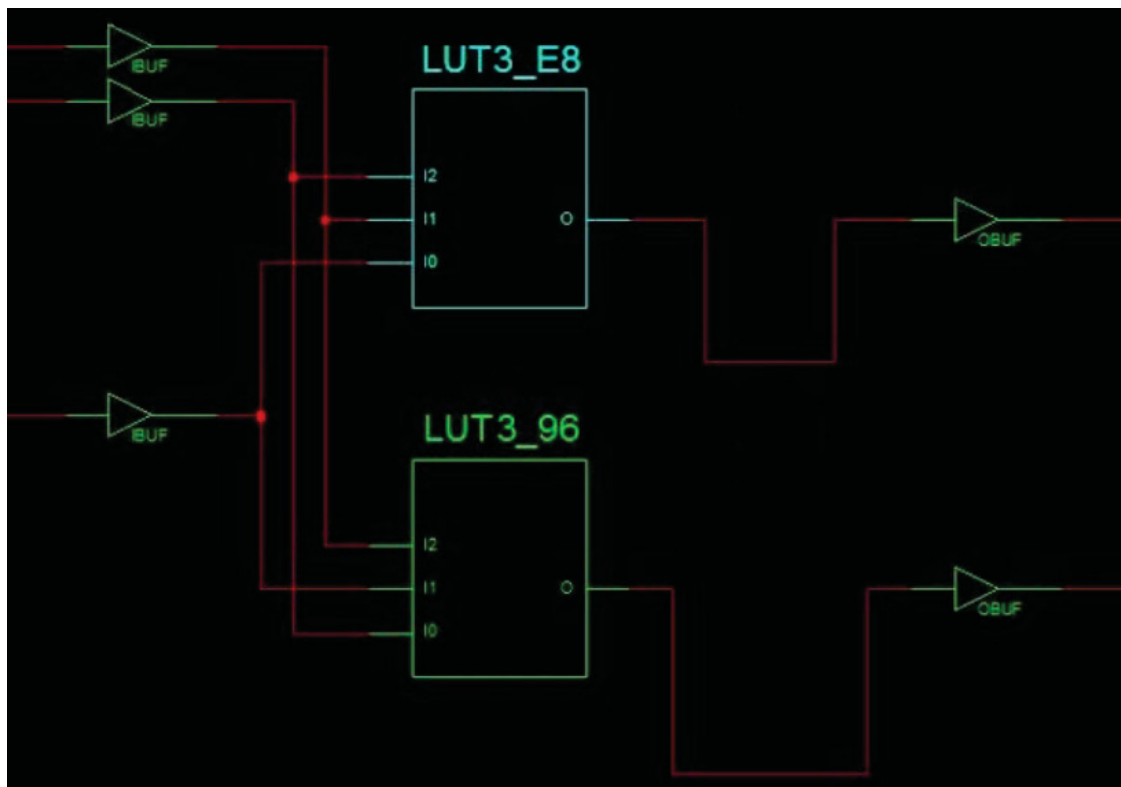


Figure 29: Detail representation of the RTL schematic for one of the convolution registers.

The synthesis process only structures the most efficient logical circuit with the instructions given by the HDL code. After synthesis, the implementation step organizes the synthesized logic circuit with the most efficient positioning using the available resources. This optimization is done automatically by the tool choosing the nearest elements to use routing with least resistance, least amount of resources – while ensuring no accuracy loss – and most minor (analytically expected) energy consumption.

3.4.4.3.2 Vitis AI based architecture

For Vitis AI-based algorithms, the user defines the total resource utilization based on the available architecture in the FPGA. The resource consumption can be modified by establishing the following user-defined system parameters:

- Architecture;
- Used Random Access Memory (URAM) Number;
- RAM Usage;
- Channel Augmentation;
- DepthwiseConv;
- AveragePool;
- ReLU Type;
- Discrete Signal Processor (DSP) Usage;
- Device;
- Softmax.

The FPGA used for development is the ZCU 104 model – which has the DPUCZDX8G integrated architecture. This architecture enables the user to access the highest resource **B4096** DPU – [22]. The following tables demonstrates the fixed number of resources used to build the DLCNNs.

DPU Architecture	Pixel (PP)	Parallelism	Input Channel Parallelism (ICP)	Output Channel Parallelism (OCP)	Peak Ops (operations/per clock)
B512	4		8	8	512
B800	4		10	10	800
B1024	8		8	8	1024
B1152	4		12	12	1150
B1600	8		10	10	1600
B2304	8		12	12	2304
B3136	8		14	14	3136
B4096	8		16	16	4096

Table 5: Parallelism for different FPGAs developed by Xilinx, Inc. – [22].

Table 5 demonstrates that each channel can achieve up to 4096 parallel operations per clock. The performance impact of different DPU architectures was not researched since the increase in parallelism makes DLCNNs more efficient by comparing the performance between CPUs and GPUs, which make lesser and greater use of parallelism – respectively.

DPU Architecture	LUT	Register	Block RAM	DSP
B512 (4x8x8)	27893	35435	73.5	78
B800 (4x10x10)	30468	42773	91.5	117
B1024 (8x8x8)	34471	50763	105.5	154
B1152 (4x12x12)	33238	49040	123	164
B1600 (8x10x10)	38716	63033	127.5	232
B2304 (8x12x12)	42842	73326	167	326
B3136 (8x14x14)	47667	85778	210	436
B4096 (8x16x16)	53540	105008	257	562

Table 6: Resource utilization for different FPGAs developed by Xilinx, Inc. – [22].

DPU Architecture	Extra Lookup tables (LUTs) with Channel Augmentation
B512(4x8x8)	3121
B800(4x10x10)	2624
B1024(8x8x8)	3133
B1152(4x12x12)	1744
B1600(8x10x10)	2476
B2304(8x12x12)	1710
B3136(8x14x14)	1946
B4096(8x16x16)	1701

Table 7: Extra LUTs of DPU with channel augmentation. – [22].

Differently from the resource utilization done by an HLS design, the deployment establishes a set number of LUTs, Registers, RAM blocks and DSPs to run the application. Maximum occupation for each parallel operation is used by establish the D4096 architecture. The used method maximizes the use of the available resources, however if the application is too small (YOLO v3) some resources may assume an idle status. Direct verification of these elements is not possible with the current tools using this method.

DPU Architecture	Low RAM Usage	High RAM Usage
B512 (4x8x8)	73.5	89.5
B800 (4x10x10)	91.5	109.5
B1024 (8x8x8)	105.5	137.5
B1152 (4x12x12)	123	145
B1600 (8x10x10)	127.5	163.5
B2304 (8x12x12)	167	211
B3136 (8x14x14)	210	262
B4096 (8x16x16)	257	317.5

Table 8: Number of blocks of random access memory of 36 kylobytes (BRAM36K) Blocks in Different Architectures for Each DPU Core – [22].

Similarly, as used for other resources, the RAM resources are fully allocated to accommodate each of the target algorithms. It is possible to configure the board to use up the different parts of the RAM blocks to different algorithms. However, selecting the RAM blocks will be executed in the same manner as done in conventional processing systems such as in a conventional platform.

3.5 Performance evaluation for keypoint detection algorithms and CNN power consumption evaluation

In order to compare the results predicted through the four stages of CNN implementation and conversion, the keypoint object detection algorithms such as PoseNet and OpenPose – were evaluated through the Object Keypoint Similarity (OKS) metric – [100]. The metric was used in COCO dataset challenges on the detection of keypoint detection and is the most divulged metric for keypoint identification in related work. The definition provided by the authors is "It is calculated from the distance between predicted points and ground truth points normalized by the scale of the person. Scale and Keypoint constant needed to equalize the importance of each keypoint: neck location more precise than hip location." – [100]. The metric can be analitically described as:

$$OKS = \exp\left(\frac{-d_i^2}{2s^2k_i^2}\right) \quad (16)$$

where d_i is the euclidean distance between the reference labelled keypoint and the keypoint predicted by the network. s is the scale, representing the square root of the object segment area; k_i is the per-keypoint constant that controls the *fall-off effect*.

This metric is used to analyse the performance for the OpenPose and PoseNet keypoint detection algorithms.

3.6 Performance evaluation for object detection algorithms

3.6.1 Mean Average Precision (mAP)

The mean Average Precision metric is used to measure the performance of target detection algorithms. In general, the mean Average Precision (mAP, also known as average class precision) is obtained by combining and weighting the average precision (AP) of all class detections. In general, the category-wide average correctness (mAP, also known as category-wide average accuracy) is obtained by combining the average correctness (AP) of all categories detected and weighting the average – [101]. The mAP value is defined analytically as:

$$mAP = \frac{1}{n} \sum_{k=1}^{k=n} AP_k \quad (17)$$

where AP_k is the average precision of the class k and n is the total number of classes. The average precision is calculated by executing the integral of the precision-recall curve formed when for a specific set of tested data:

$$AP = \int_0^1 P(R) dR \quad (18)$$

where $P(R)$ is the precision in function of R , the recall.

The precision is verified over the *true positive* results over the total amount of samples:

$$P = \frac{True\ Positives}{True\ Positives + False\ Positives} \quad (19)$$

while the recall R is defined as:

$$R = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (20)$$

The aforementioned detections are:

- TP = True Positives (Predicted as positive as was correct);
- FP = False Positives (Predicted as positive but was incorrect);
- FN = False Negatives (Failed to predict an object that was there).

The evaluation must consider a minimum threshold over the Intersection over Union (IoU) – [102] value in order to know what to consider as a positive detection - following the challenge in [101], the IoU threshold considers positives values above 50 % intersection between the predicted bounding box and the labelled (ground truth) bounding box.

3.6.2 Energy evaluation

Two methods were used to measure power consumption over the studied algorithms:

- Hardware model estimation;
- OS system estimation.

3.6.3 Hardware model estimation

In order to analyse the energy consumption of devices implemented over the HLS system enables one to verify the resource occupation and energy consumption in much more detail than what is observed for an OS system estimation. In this method, it is possible to verify the power consumption of each logic element used to build the application. Other parameters such as voltage and temperature are also simulated and estimated.

3.6.3.1 The Vivado Design Suite

The Vivado Design Suite is a Xilinx design environment that boosts design, integration, and deployment productivity on the UltraScale FPGA platforms. The growth of ever more

complicated System on Chip (SoC) devices, which bring a novel approach in the programming phase, necessitates the creation of a unique automated environment. When these new devices are handled poorly, they create multidimensional design issues, which have a significant impact on development timelines and productivity. The whole design process may be conducted in memory without the need to write or translate intermediate file formats, which would slow down the implementation, debugging, and deployment stages, as well as raise memory needs.

All of the Vivado Design Suite products are developed in Tool Command Language (Tcl), a native scripting language that can be used with a command interface. A Tcl script may cover the complete design and implementation process, including all necessary reports provided for design analysis at any stage. The biggest benefit is that you have total control over the whole operation.

The Vivado software package, which provides a new paradigm of hardware design, is used to establish the design workflow for the newest generation systems, ZCU 104. Such software can automate low-level detail management in order to fulfill the needs of the many supported processors. The user should only give instructions for defining the design structure and dealing with floor-planning in general. The partly reconfigurable design process is similar to the normal one, but it includes a few more phases. This design flow necessitates the use of various configurations, which results in the development of whole bitstreams for each configuration and partial bitstreams for each reconfigurable module. The number of settings required vary depending on the number of modules to be implemented. All configurations, however, use the same static and routed design, which is exported from the first configuration and imported into all future ones.

The construction of an infrastructure for the partial dynamic reconfiguration approach is the work's key contribution. For our implementation, we chose the ZCU 104 as the target device. To help our study, the design workflow was modified to take into account the common aspects and phases. Despite the fact that the implementation phases may be used regardless of the target device, owing to the Vivado automation procedures, the initial design step, which is the generation of a block diagram from a combination of IP cores, must be tailored to each target system. Some IP cores (for example, the PS IP Wrapper) are specialized to the target board and cannot be used in generic design procedures.

As a result, the first step required drawing a diagram in which the DSP components (video recording, filtering, and display) constituted the design's starting point. Then, to make the entire design tailored to the target device, the particular blocks of PS, clock, interrupt, and interconnections were included. As a result, device-dependent constraint files were developed and loaded into Vivado to connect the design signal line to the board's actual pins. The constraint files are often utilized in later design processes.

The design flow is defined by synthesis and implementation procedures, which are based on the static design flow, after selecting a reconfigurable target block. The primary distinction is that the outline of the region of the target chip that has to be reconfigurable (in our

example, the programmable logic of the ZCU 104) initiates the Vivado process for the final development of the partial configuration files.

The structure adjusted, including all peripherals necessary to run the Yolo V4 algorithm, could be accommodated using the same components shown in the Vivado tool IDE in fig. 30.

Figure 31 demonstrates the position of all logic circuits of the total FPGA resources that must be allocated to create the application.

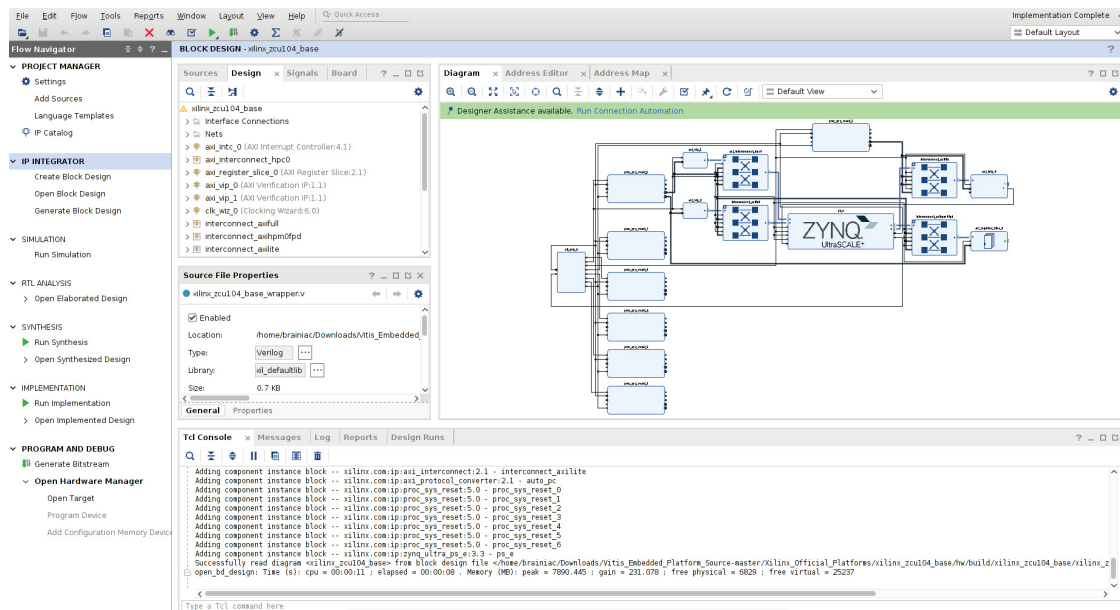


Figure 30: Hardware interface structure defined for the object detection algorithm.

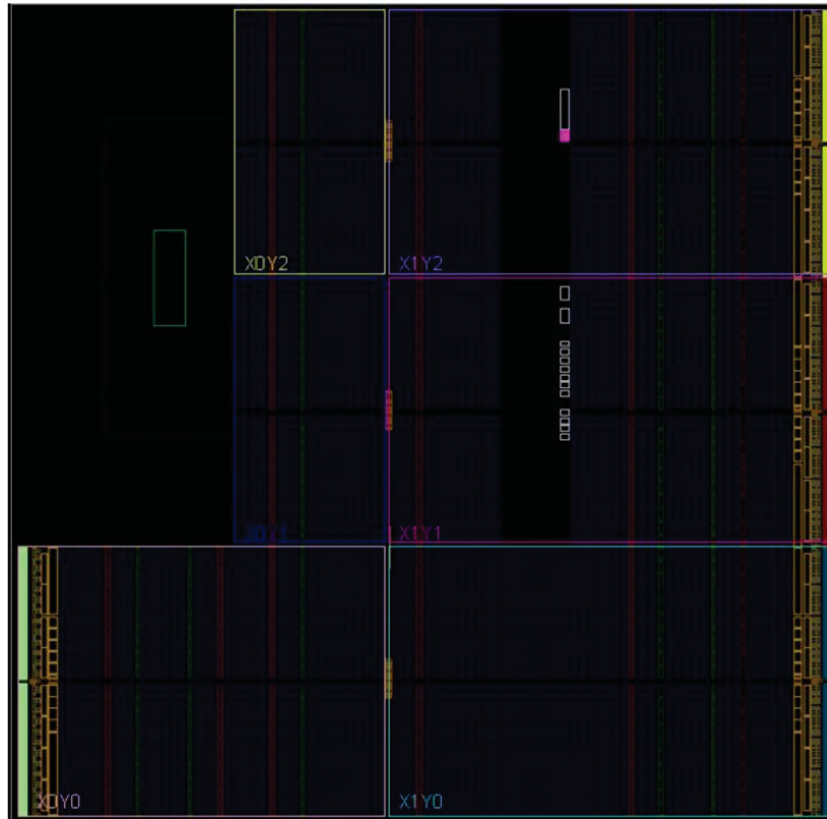


Figure 31: Logic circuit utilization without implementation. Each section is divided for specific functions such as convolution, pooling, sum and main OS interface – [9].

Figure 32 is a zoomed view of fig. 31 presenting the logic gates, DSPs and custom technology blocks inside the FPGA device.

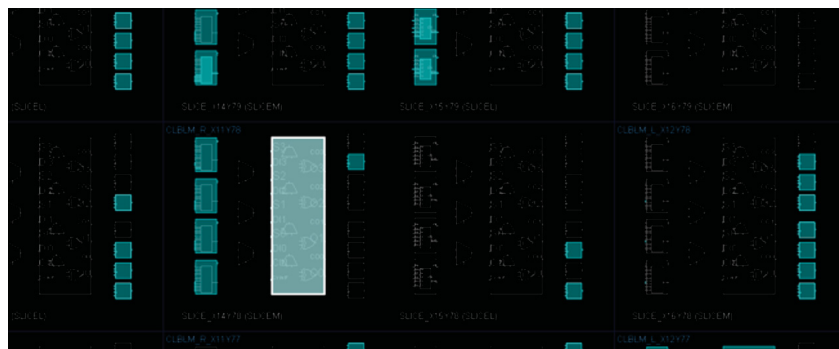


Figure 32: Representation of the logic elements inside the FPGA device with a group of logic gates used for convolution.

Each of the elements are have target ports that are going to be used for the routing step – which is automated by the Vivado tool. Figure 33 demonstrates the already defined

connection targets for a register used in the first convolution filter.

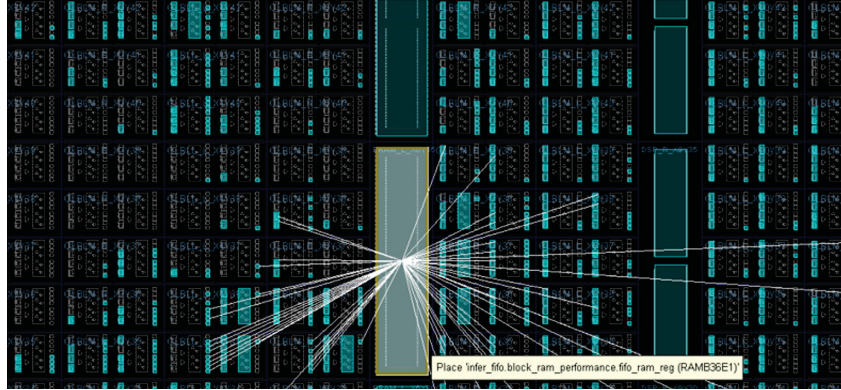


Figure 33: Representation of the routing targets for a convolution register.

After the routing is executed, the general routing structure for each block is done laterally and autonomously, as shown in fig. 34.

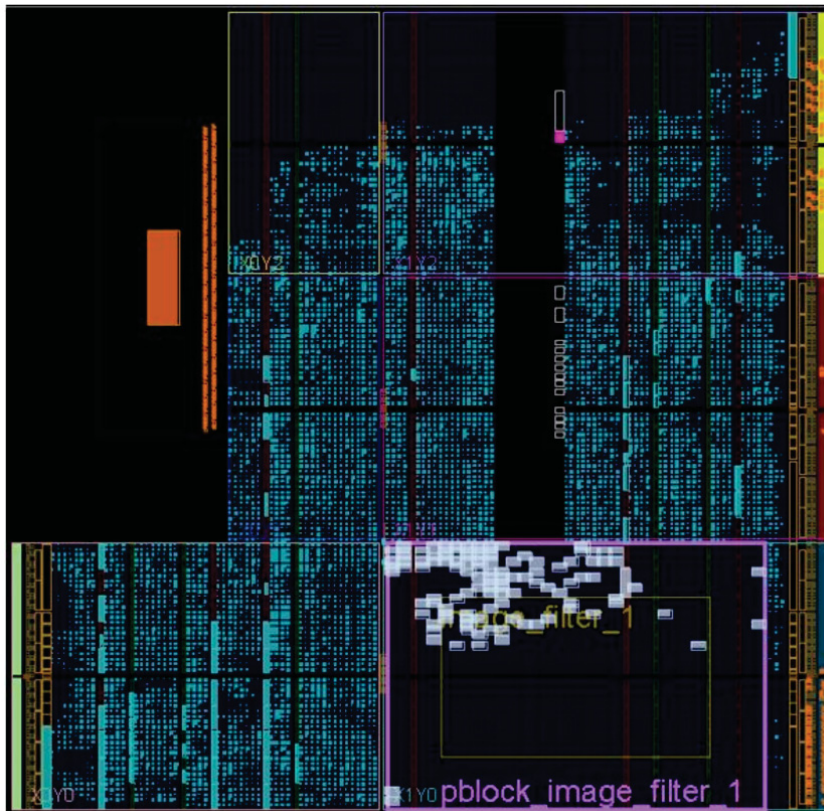


Figure 34: Representation of different parts of an image filter in [9] after implementation (routing and positioning).

Figure 34 represents the image filter application developed for [9].

3.6.3.2 Power consumption estimation

The method used to estimate power consumption in each element and globally was partially based on the work in [9]. Physical measurements may be understood and analyzed using an analytical framework. We will look at how and under what situations partial dynamic reconfiguration can save energy. The energy consumption of electronic devices is influenced by structural characteristics (such as the technology utilized, the semiconductor manufacturing process, and the operating frequency) and functional characteristics (i.e., DSP task and technique, coding language, requested throughput). Furthermore, programmable FPGAs add another layer of complexity. In reality, FPGAs may modify the DSP functionality dynamically while keeping the fundamental elements unchanged. As a result, the gap between the structural and functional power consumption is narrowing, as the same hardware resources may be used for several applications (modifying configuration and routing).

As a result, energy consumption is influenced by the structure and functionality and by a set of FPGA parameters intimately linked to each reconfigurable hardware design. Since the objective of this work is to compare functionality and performance parameters, the structural expenses are ignored. The reconfiguration time and working frequency are given to a specific module included in the FPGA. As a result, due to the large number of elements that must be examined simultaneously, developing an analytical model that thoroughly examines how each parameter influences energy consumption is quite challenging. Furthermore, all of the emulated factors can only be inferred by Xilinx IP models rather than measured objectively. The results section presents the power consumption in each simulated element for the application functionality.

3.6.4 OS system estimation

Estimating the energy usage for systems that do not rely on HLS must be shared for all studied platforms. The energy estimation on CPUs and GPUs does not have analytical models provided by the manufacturer, such as the Vivado tools. In order to estimate the power consumption in these devices, the Linux OS native *powertop* command is used.

The OS can determine the power used for each process. The user separates the processes to verify the distributed power for the analyzed algorithm. It is possible to verify that the console process is solely responsible for conducting the algorithm analysis as a single process. Overall data collection steps, ten iterations over a 5-minute continuous period, were analyzed while actively restricting other system applications to a bare minimum. Figure 35 represents the graphical user interface (GUI) used to verify power consumption data with the OS system estimation.

PowerTOP 2.7					
Overview		Idle stats	Frequency stats	Device stats	Tunables
Summary: 1541.8 wakeups/second, 42.9 GPU ops/seconds, 0.0 VFS ops/sec and 18.9% CPU use					
Power est.	Usage	Events/s	Category	Description	
4.45 W	0.0 pkts/s		Device	nic:virbr0	
1.45 W	88.7 ms/s	315.3	Process	/usr/bin/gnome-shell	
353 mW	54.7%		Device	Display backlight	
292 mW	36.7 ms/s	103.1	Process	/usr/libexec/Xorg vt4 -displayfd 3 -auth /run/user/100	
200 mW	0.0 pkts/s		Device	Network interface: wlp2s0 (iwlwifi)	
146 mW	7.4 ms/s	57.6	Process	/usr/libexec/gnome-terminal-server	
110 mW	4.9 pkts/s		Device	Network interface: enp3s0 (r8169)	
7.31 mW	1.3 ms/s	92.4	Process	/usr/libexec/at-spi2-registryd --use-gnome-session	
0 mW	8.7 ms/s	62.0	Process	/opt/google/chrome/chrome --type=renderer --enable-def	
0 mW	5.4 ms/s	385.4	Interrupt	PS/2 Touchpad / Keyboard / Mouse	
0 mW	4.9 ms/s	79.0	Process	/opt/google/chrome/chrome	
0 mW	4.4 ms/s	2.5	Process	/usr/bin/python /usr/bin/powerline-daemon -q	
0 mW	4.3 ms/s	163.0	Process	powertop	
0 mW	3.6 ms/s	18.6	Process	gnome-shell --mode=gdm --wayland --display-server	
0 mW	3.1 ms/s	5.6	Process	tmux	
0 mW	2.8 ms/s	3.0	Process	tmux	
0 mW	2.5 ms/s	0.00	Timer	hrtimer_wakeup	
0 mW	2.1 ms/s	4.1	Process	sh	
0 mW	1.8 ms/s	37.8	Timer	tick_sched_timer	
0 mW	1.3 ms/s	0.00	Interrupt	[1] timer(sofirq)	
0 mW	1.1 ms/s	0.8	Interrupt	[7] sched(sofirq)	
0 mW	1.0 ms/s	22.3	Interrupt	[46] 1915	
0 mW	0.9 ms/s	106.9	Process	[rcu_sched]	
0 mW	0.9 ms/s	0.00	Timer	intel_pstate_timer_func	
0 mW	0.8 ms/s	48.2	Process	[rcuos/0]	
0 mW	693.2 µs/s	2.1	Process	env	
0 mW	461.9 µs/s	0.00	Timer	process_timeout	

Figure 35: GUI used for power consumption estimation over various iterations

Each algorithm is analyzed over the same hardware for a CPU and GPU study. The multiple processor system-on-chip has the powertop command included over the construction of the custom OS used to operate alongside the FPGA. The method is used in the same manner to run the algorithm over the same time and number of iterations as done in the CPU while using the same console interruptions to ensure the involved hardware was focused on running the application.

3.7 Materials

Three different devices were used to evaluate the YOLO v4 algorithm on the COCO dataset – [103]. Each device is a data processing unit with different features and performances. Furthermore, each device is considerably more customized to the custom algorithm. The most diminutive specialized device is the CPU. The GPU possesses features customized for image processing for general use. The FPGA is structured so that the logic gates inside the device are specially arranged to execute a specific function. Therefore, it usually exhibits the most significant energy efficiency and performance compared to the previously noted devices.

3.7.0.1 10th Generation Intel® Core™ i7-10750H processor (test on CPU)

Processor

Name	Intel Core i7 10750H				
Code Name	Comet Lake	Max TDP	45 W		
Package	Socket 1440 FCBGA				
Technology	14 nm	Core Voltage	0.798 V		
Specification	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz				
Family	6	Model	5	Stepping	2
Ext. Family	6	Ext. Model	A5	Revision	
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T VT-x, AES, AVX, AVX2, FMA3				

Clocks (Core #0)

Core Speed	2293.26 MHz
Multiplier	x 23.0 (8 - 50)
Bus Speed	99.71 MHz
Rated FSB	

Cache

L1 Data	6 x 32 KBytes	8-way
L1 Inst.	6 x 32 KBytes	8-way
Level 2	6 x 256 KBytes	4-way
Level 3	12 MBytes	16-way

Selection

Processor #1

Cores

6

Threads

12

Figure 36: Parameters of the CPU used for testing.

The CPU is one of the tested processing units used to evaluate the performance of the YOLO v4 algorithm. Each core used for the conducted test has a frequency of 2.6 GHz. Using six cores of the i7 processor, the maximum frequency reaches 5 GHz. The test conducted uses the Vitis AI library to send the bitstream generated from the C/C++ language to the processor to perform the calculations. Parameters for this device are presented in fig. 36.

3.7.0.2 GEFORCE GTX 1660 Ti (Test on GPU)

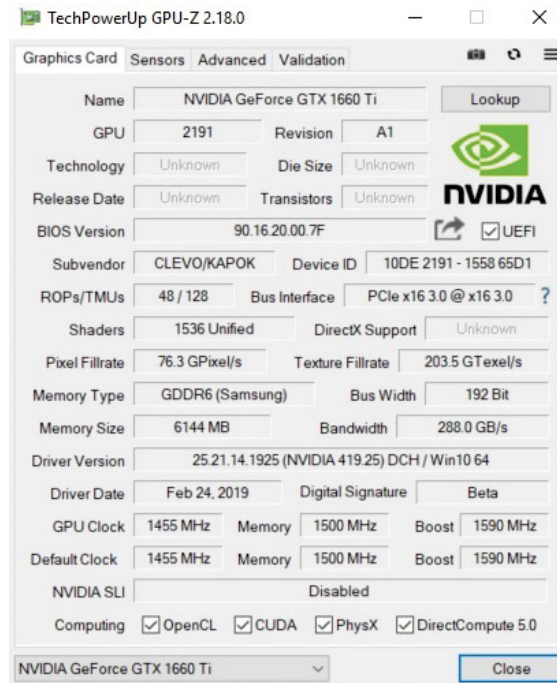


Figure 37: Parameters of the GPU used for testing.

The graphics card used for testing is the NVIDIA GEFORCE GTX 1660 Ti with 6 Gigabytes (GB) of memory. This device is set to operate at the maximum frequency of 1770 MHz for testing the YOLO v4 algorithm. The device requires using the CUDA libraries for NVIDIA hardware to operate. The CUDA libraries are written in C and C++. However, the VITIS AI software is projected to integrate each wrapper function into CUDA coding, so customization is required. The user is only prompted to specify which is the end device that is going to perform the calculations for the AI acceleration. Parameters for this device are presented in fig. 37.

3.7.0.3 ZYNQ Ultrascale + MPSoC (Test on FPGA device)



Figure 38: ZYNQ ZCU104 Ultrascale + MPSoC board used for testing.

The FPGA device is integrated into an evaluation board with an MPSoC (Multiprocessor system on a chip) feature with additional I/O ports. The board is fabricated and sold by Xilinx, Inc. The board has a plug-and-play functionality with the VITIS AI library, making the development process significantly shorter for the user – since the developer does not have to write hardware description code. The software setup procedure is similar to the one conducted for the previous processing units. The network is pruned, quantized, and deployed on the specified processing unit. The clock used for all tests is 250 MHz. A representation of the device is presented in fig. 38.

4 Results

This part presents the results comparing the difference in performance parameters after each stage in CPUs, GPUs, and the FPGA. For each algorithm, the following parameters are evaluated: frame rate, latency, accuracy, global power consumption, and frame specific consumption. Table 9 presents the metric used for each processing platform.

	CPU	GPU	FPGA
YOLO v3	mAP / PC (powertop))	mAP / PC (powertop))	mAP / DPU energy evaluation ((powertop)
YOLO v4	mAP / PC (powertop))	mAP / PC (powertop))	mAP / Vivado estimation
OpenPose	OXS / PC (powertop))	OXS / PC (powertop))	OXS / DPU energy evaluation (powertop)
PoseNet	OXS / PC (powertop))	OXS / PC (powertop))	OXS / DPU energy evaluation (powertop)

Table 9: Representation of the performance and energy consumption evaluation metric for each platform. PC represents power consumption.

4.1 YOLO v3

The results of the YOLO v3 algorithm were used to evaluate how each of the I-IV methodology stages responded to variations (from high level to low level programming) in the object class detection for license plates using a predefined wrapper and structure from Vitis AI. In addition, trained weights from the Darknet framework were converted into a Keras frozen graph and optimized using stages II and III for direct FPGA implementation.

Figure 39 presents the result with the object detection algorithm for plate detection observed.

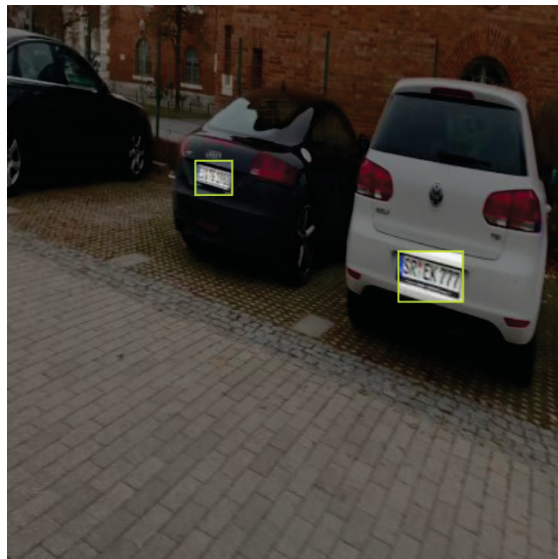


Figure 39: Plate detection seen by drawing a bounding box around the AI generated coordinates in a 608x608 pixel image.

The aforementioned 10th Generation Intel® Core™ i7-10750H mobile processor was used to test the YOLO v3 algorithm for license plate detection. The YOLO v3 algorithm is trained to identify license plates based on the Technische Hochschule Ingolstadt (THI) dataset. When applying the pruning, fig. 40 represents the mAP on the left Y axis and the frame rate of the right Y axis versus the compression level.

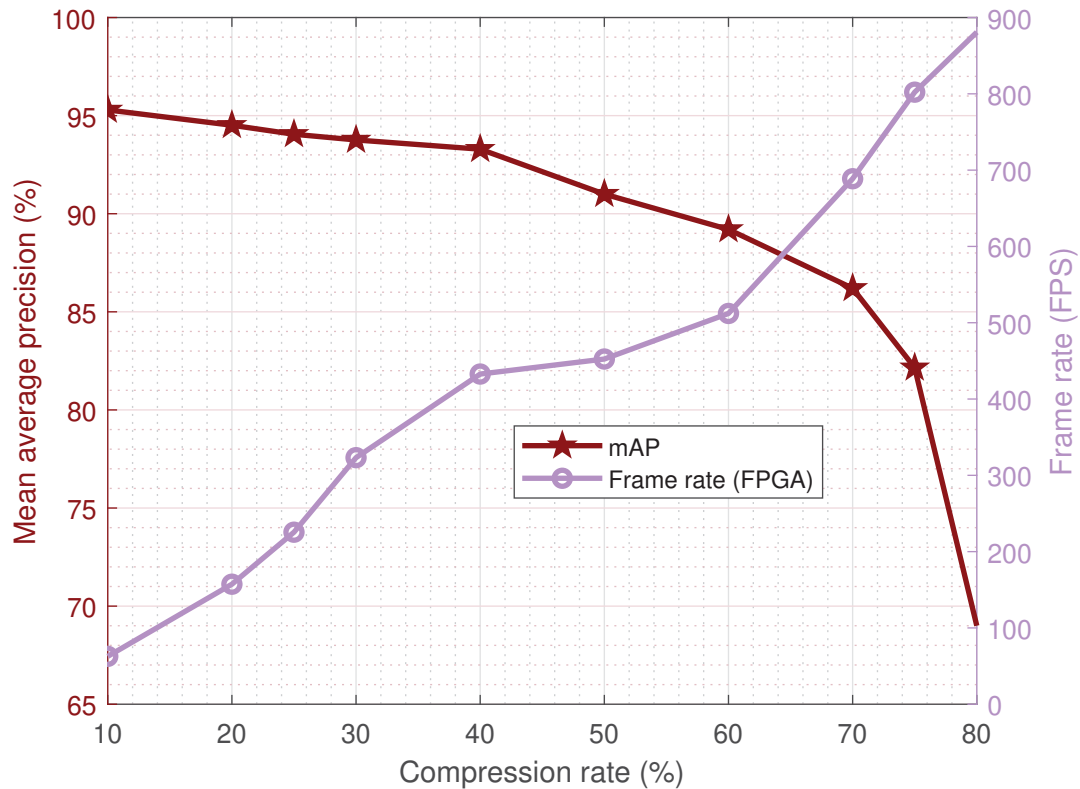


Figure 40: Accuracy decay and frame rate enhancement for different compression rates of the YOLO v3 algorithm.

The mean value for the distribution was considered the most trustworthy for the compression value in fig. 40.

The compression of more than 70 % yields an almost linear behavior in the mean average precision value when evaluated for the FPGA platform. This linear behavior is explained by the relative complexity of the algorithm being interpreted by the compilation engine's decrease and the low mAP results being processed as random data detection. The removal of layers in a linear fashion results in a linear decay for small processing complexity. Each compression point takes about 72 h to simulate in order to generate the cross-processor model which can be compared using a common baseline. The methods of optimization for the deployment are reviewed in the methodology section. Figure 41 presents the frame rate performance for the same accuracy. The different hardware implementations showed no

significant difference in the mAP per compression rate.

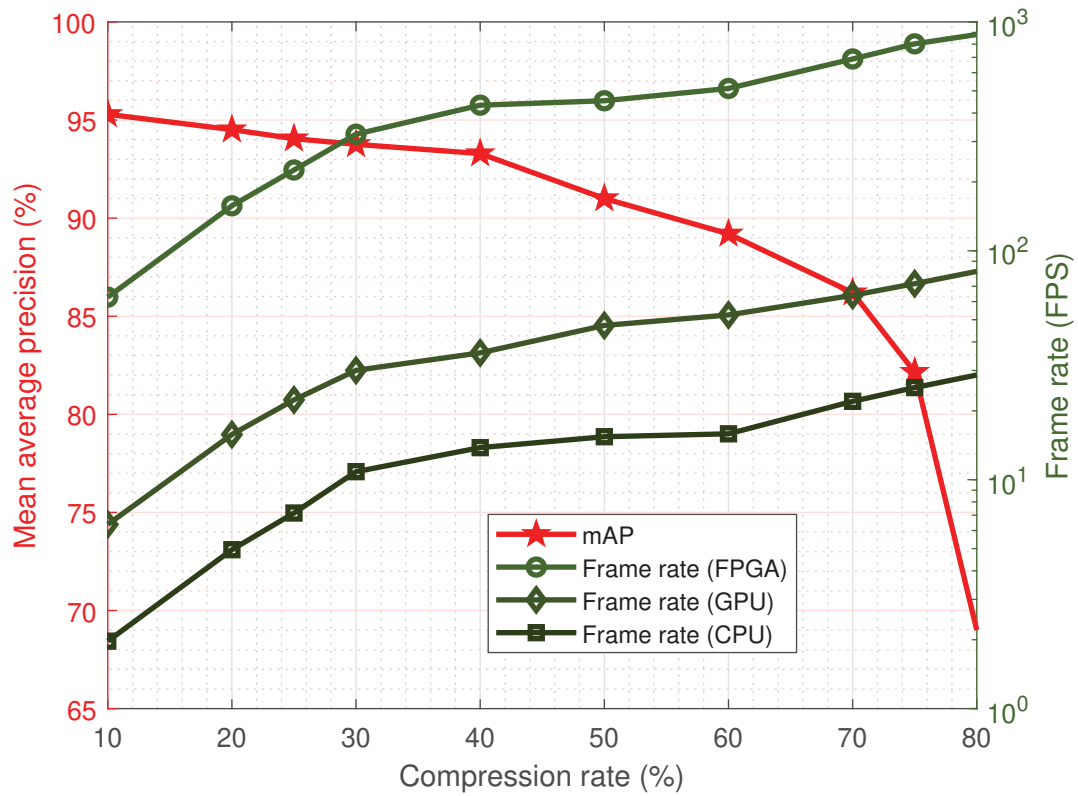


Figure 41: Performance comparison among different platforms for the YOLO v3 algorithm (optimisation normalized to prevent accuracy loss).

One can observe in fig. 41 that the FPGA is faster one order of magnitude, that is why the frame rate had to be shown in logarithmic scale. This makes less evident to highlight the difference between the GPU and CPU. Table 10 show the performance parameters for network at 50% compression rate and the different hardware.

Performance parameter	CPU	GPU	FPGA
Frame rate (FPS)	15.41	47.21	452.44
Average precision (%)	93.29	93.29	93.29
Energy consumption per frame (mJ/Frame)	3.43	1.82	0.87
Latency (ms)	94.56	41.43	5.40

Table 10: Performance data for the YOLO v3 algorithm for license plate detection.

Accuracy data set after the optimization and quantization steps in 93.29 % was used to compare the differences in execution in each implementation platform at 50 % compression.

After compression of approximately 80 %, the algorithm notably loses the ability to

process each image accurately or does not detect car plates in the test dataset. This marks a *compression saturation point*, in which the optimization step reduces the number of neurons and synapses that the algorithm is unable to detect images from a target dataset effectively while still being capable of predicting accurately inside the training dataset. The saturation point behaves similarly to over-trained CNNs.

4.2 YOLO v4

The YOLO v4 algorithm uses new computer vision techniques to improve YOLO v4 in terms of accuracy and speed. The network uses a considerably more significant amount of resources than its predecessor to attain these gains in performance. The DLCNN was tested on the same datasets for its training, validation, and evaluation as YOLO v3. However, the algorithm was developed using the HLS platform, converting the source C++ and darknet framework through the discussed stages I to IV.

The output parameters of the algorithm are the same as for its predecessor: coordinates for the bounding box and the class. After drawing the bounding box, the detection output is the same as the one observed in fig. 40.

Figure 42 demonstrates the decay in accuracy for different compression checkpoints.

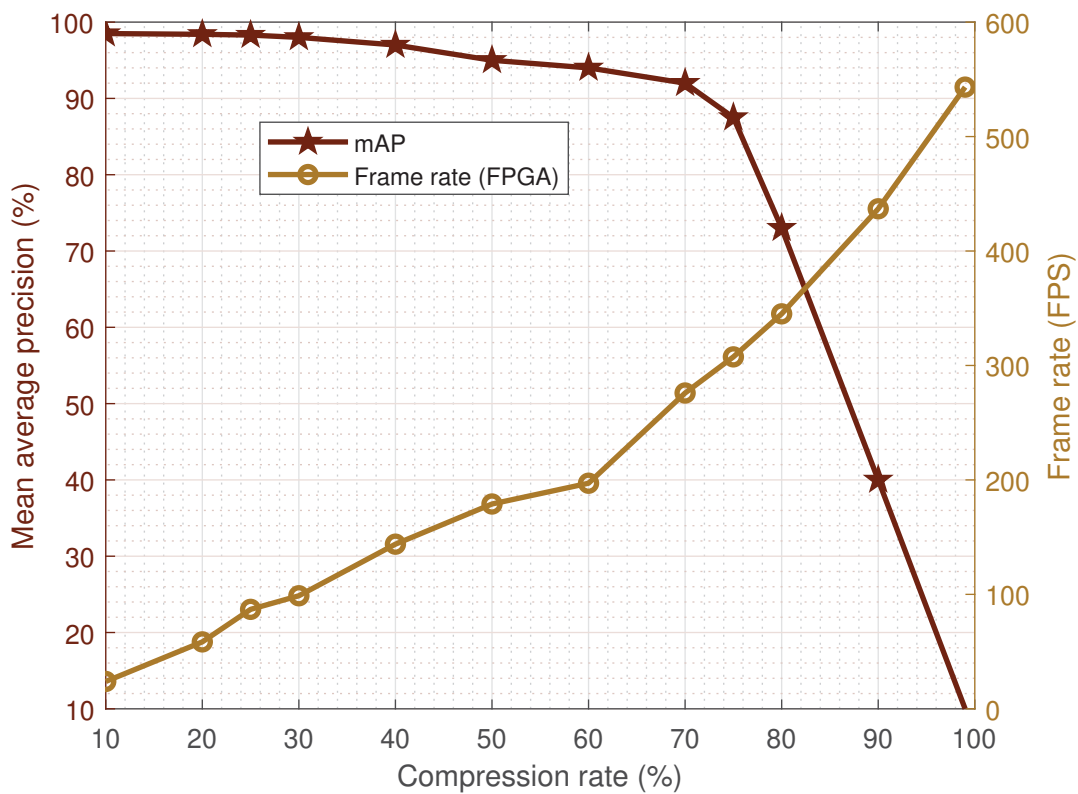


Figure 42: Accuracy decay and frame rate enhancement for different compression rates of the YOLO v4 algorithm.

The results obtained for the YOLO v4 algorithm are similar to those obtained for YOLO v3 in terms of accuracy decay. However, the network has a more complex structure, making it comparatively slower. Nevertheless, the accuracy gain on the network can be observed by

greater initial accuracy and an overall smaller frame rate.

A *compression saturation point* can be observed close to the 80 % compression point similarly to the previous version.

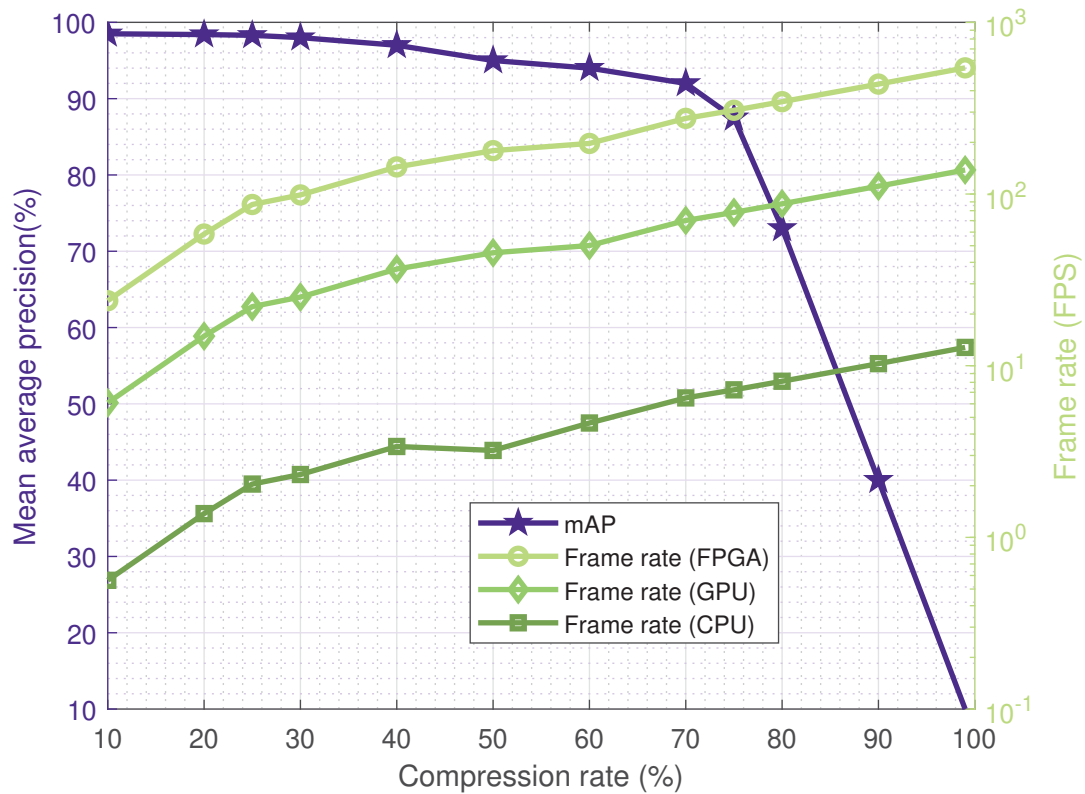


Figure 43: Performance comparison among different platforms for the YOLO v4 algorithm (optimisation normalized to prevent accuracy loss).

Again, the network is tested against the different hardware implementations. Similar behavior was observed between both algorithms. However, the difference between the performance of the FPGA and GPU was slightly reduced. It should be noted that some operations using the YOLO v4 algorithm cannot be fully parallelized, in which the FPGA normally takes advantage. The method used to design the logic circuit differs from the one used to implement the YOLO v3 algorithm, which uses a set number of resources and can divide the hardware requirements to use all those resources. Table 11 shows the performance parameters at 60% compression rate for the different hardware implementations:

Performance parameter	CPU	GPU	FPGA
Frame rate (FPS)	4.21	45.49	178.95
Average precision (%)	95	95	95
Energy consumption per frame (mJ/Frame)	2.423	1.763	1.192
Latency (ms)	346.95	43.46	12.47

Table 11: Performance data for the YOLO v4 algorithm for license plate detection. Accuracy data set after the optimization and quantization step in 95 % was used to compare the differences in execution in each implementation platform.

A dramatic increase in frame rate can be observed as the processing unit becomes more specialized. Even though the methodology differs in implementation using the high-level synthesis, the frame rate increase can be similarly observed to the YOLO v3 implementation. The optimized CPU code yields the slowest frame rate and the highest energy consumption even if all its resources are used to process the algorithm. The GPU uses the CUDA software to use the board's resources, including the tensor cores TPUs (Tensor Processing Unit) available on the RTX 2080 board.

4.3 PoseNet

The PoseNet algorithm was used to test the FPGA's capabilities to detect computer vision algorithms whose targets are not object detection. PoseNet is used to detect 14 keypoints in the human body. Figure 44 demonstrates the detected keypoints on the HumanEval dataset.

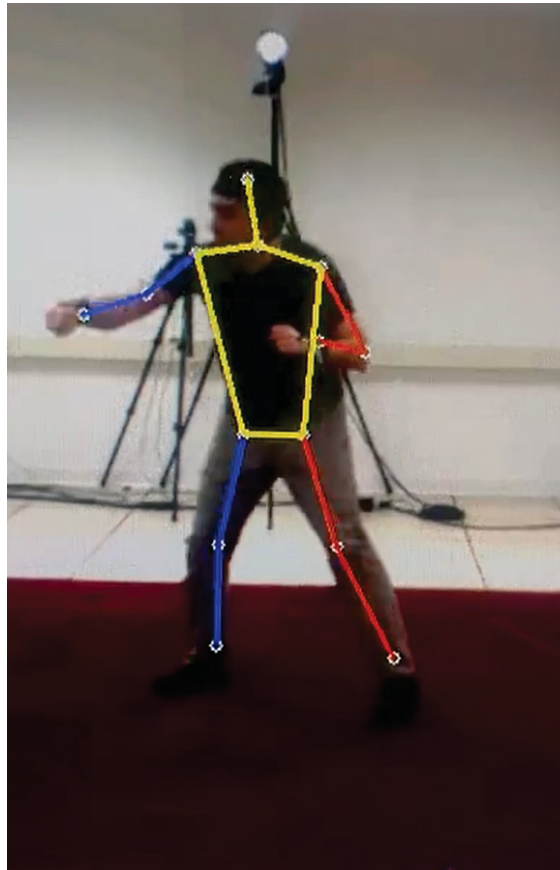


Figure 44: Demonstration of the target 14 keypoint detectors on a human body (HumanEval dataset).

Figure 45 demonstrates the OKS (left Y axis) and frame rate (right Y axis) versus compression rate for the PoseNet algorithm implemented in FPGA. Differently from the object detection algorithms, a break point for the frame rate is not observed for this specific keypoint detection algorithm. Compression equates an approximately linear behavior in performance.

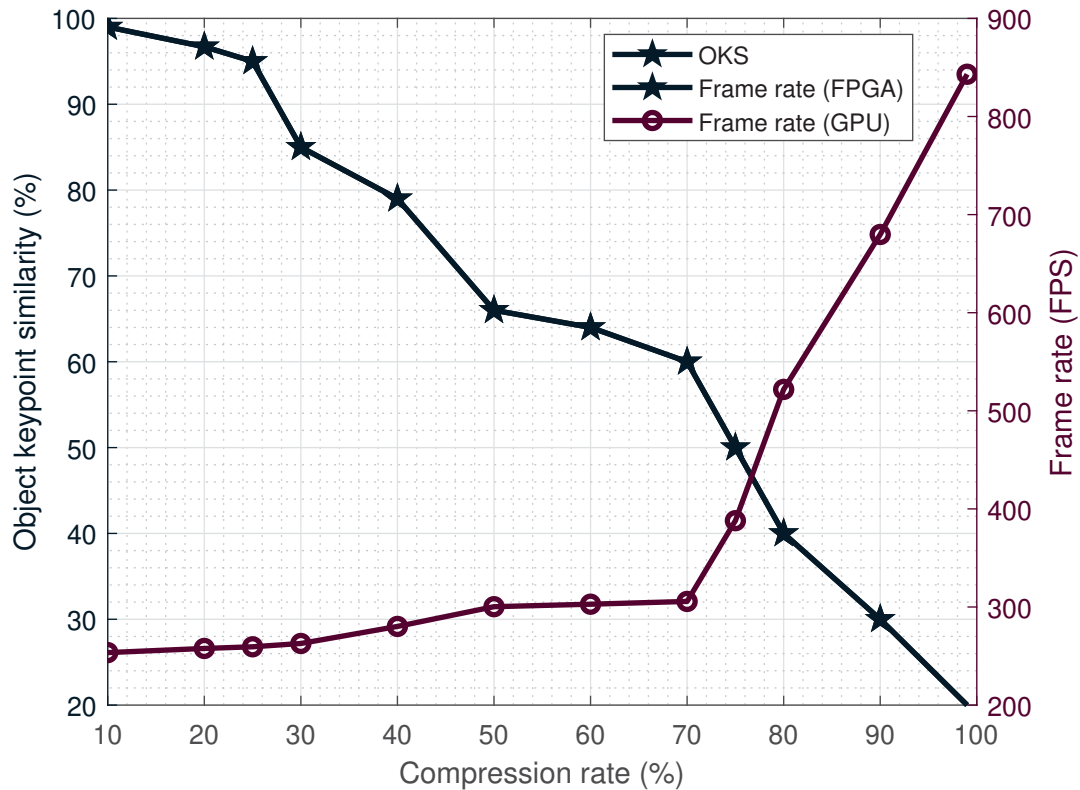


Figure 45: Accuracy decay and frame rate enhancement for different compression rates of the PoseNet algorithm.

Table 12 presents the performance parameters of the model at 60% compression rate for the different hardware implementations. The PoseNet algorithm is smaller and simpler than the object detection algorithms, therefore one can observe higher frame rates. From fig. 49 one can observe a very significant increase in frame rate for FPGAs can be seen for the implementation of this algorithm as well. Part of the algorithm's simplicity is reflected in the small consumption of energy per frame observed by changing platforms and comparing the obtained values to those obtained for object detection algorithms.

Performance parameter	CPU	GPU	FPGA
Frame rate (FPS)	38	128	300.313
Average precision (%)	66	66	66
Energy consumption per frame (mJ/Frame)	1.104	0.872	0.429

Table 12: Performance data for the PoseNet algorithm for keypoint detection. Accuracy data set after the optimization and quantization step in 66 % was used to compare the differences in execution in each implementation platform.

Figure 49 presents the algorithm's performance on different platforms with the same

methodology used for the evaluation of the object detection algorithms.

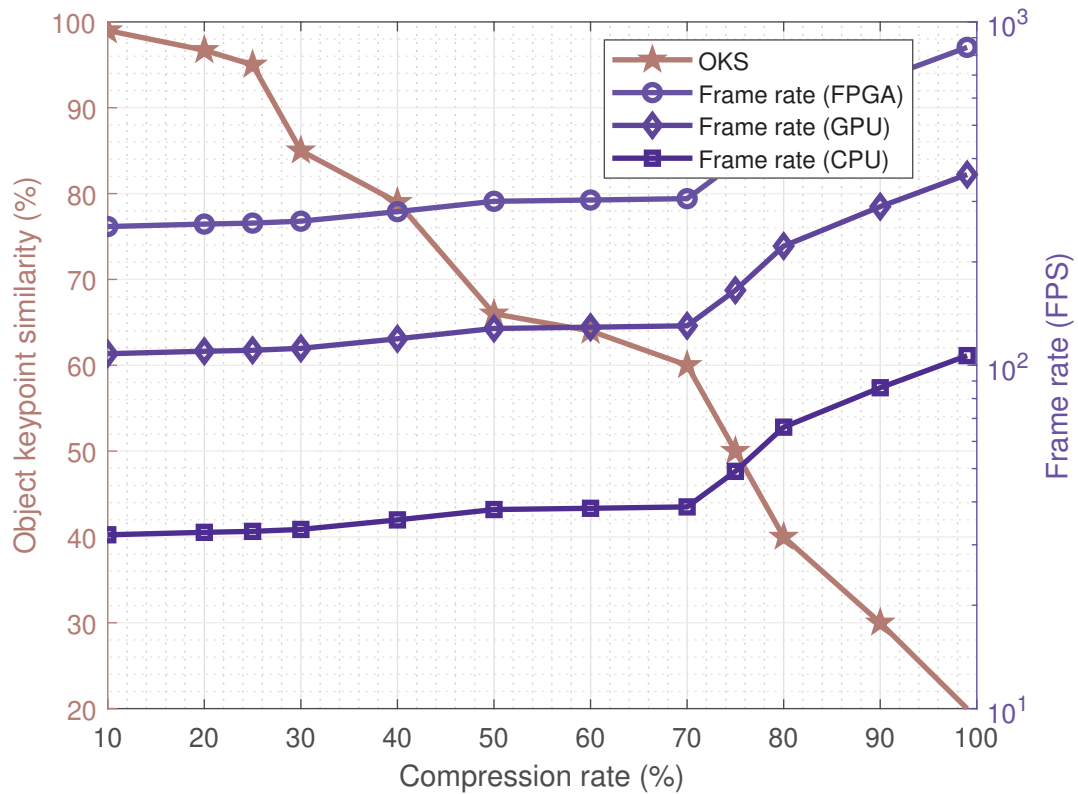


Figure 46: Performance comparison among different platforms for the PoseNet algorithm (optimisation normalized to prevent accuracy loss).

The behavior observed for PoseNet differs from the previously analyzed algorithms in terms of speed. After the 70 % compression mark, the algorithm quickly loses accuracy and rapidly increases the frame rate. This behavior is justified due to the slight complexity required to process a small network. However, as the network becomes incredibly simple, it becomes incapable of executing correct predictions and often poses incorrect values. The methodology used to measure accuracies below 50 % is the same as the previous algorithms.

4.4 OpenPose

The OpenPose algorithm is an enhanced keypoint detector used to test the functionality of keypoint detection for multiple targets and 24 keypoints. Figure 47 demonstrates how each of the 24 points are positioned for a humanoid shape.

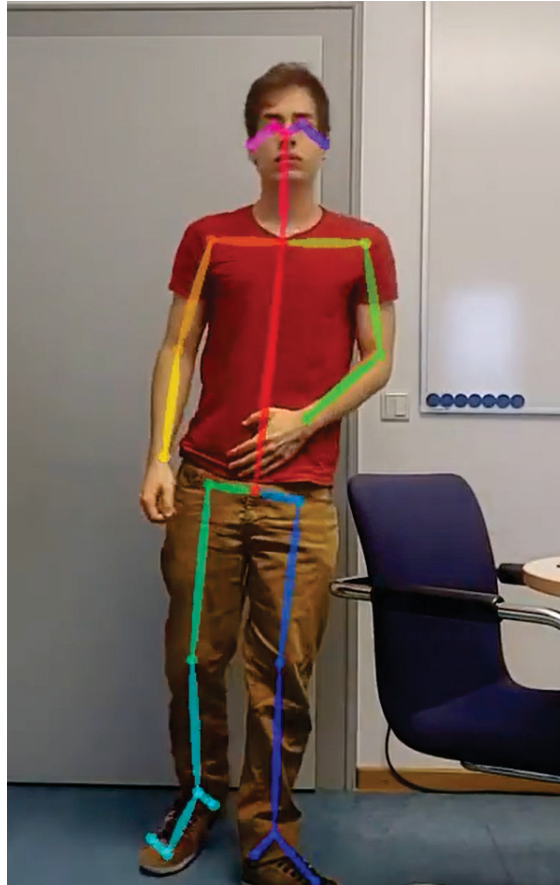


Figure 47: Demonstration of OpenPose 24-keypoint detection.

The OpenPose algorithm is the most complex DLCNN tested on this study. Figure 48 demonstrates the OKS (left Y axis) and frame rate (right Y axis) versus compression rate for the OpenPose algorithm implemented in FPGA. Table 13 presents the performance parameters of the model at 50% compression rate for the different hardware implementations.

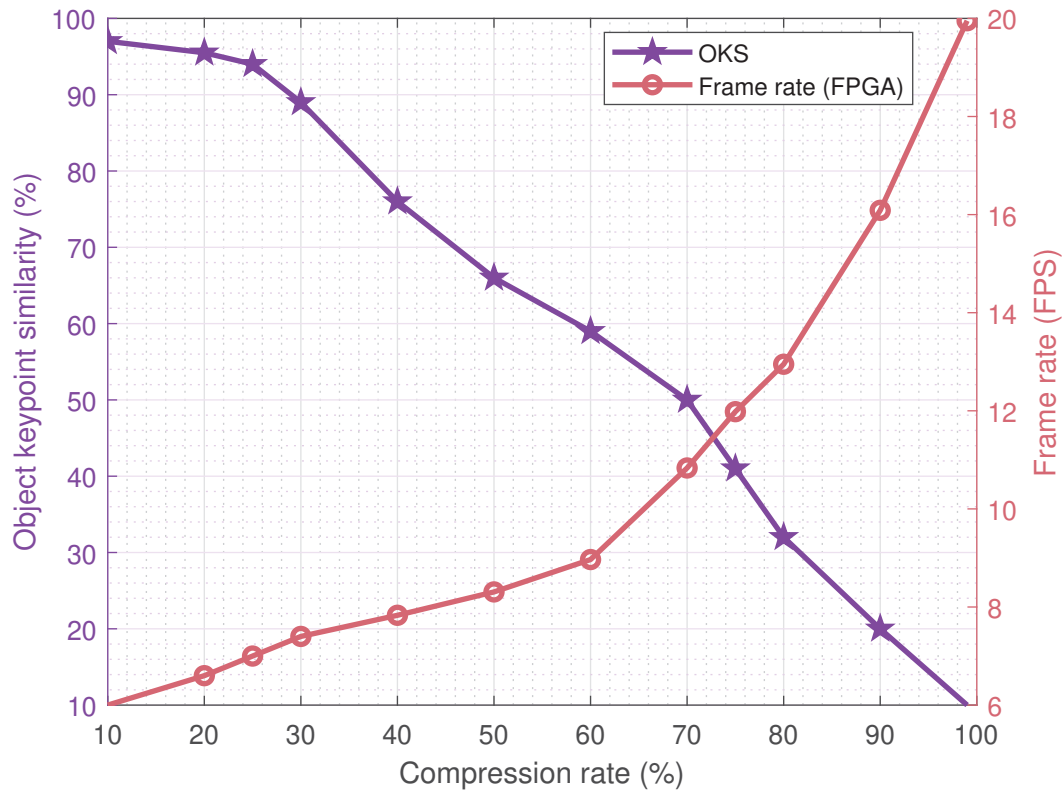


Figure 48: OKS and frame rate of the OpenPose algorithm for different compression rate.

Performance parameter	CPU	GPU	FPGA
Frame rate (FPS)	0.2	30	8.4
Average precision (%)	65	65	65
Energy consumption per frame (mJ/Frame)	6.532	2.721	2.268

Table 13: Performance data for the OpenPose algorithm for keypoint detection. Accuracy data set after the optimization and quantization step in 65 % was used to compare the differences in execution in each implementation platform.

Differently from the previous analysis, the performance of the FPGA for the OpenPose algorithm is not greater than the GPU. While the GPU can process the at 30 FPS for a 50 % compression, the FPGA reaches a much lower 8.4 FPS speed. This behavior is observed due to the inability of the FPGA to adequately accommodate all necessary intermediate instructions into a parallel structure – due to the insufficient number of hardware resources. The slower performance is explained by having to serialize different stages of operations into limited hardware. Each operation stage has to be processed into different serial stages to complete the processing of a single frame, greatly reducing the speed output for the same accuracy – when compared to other platforms.

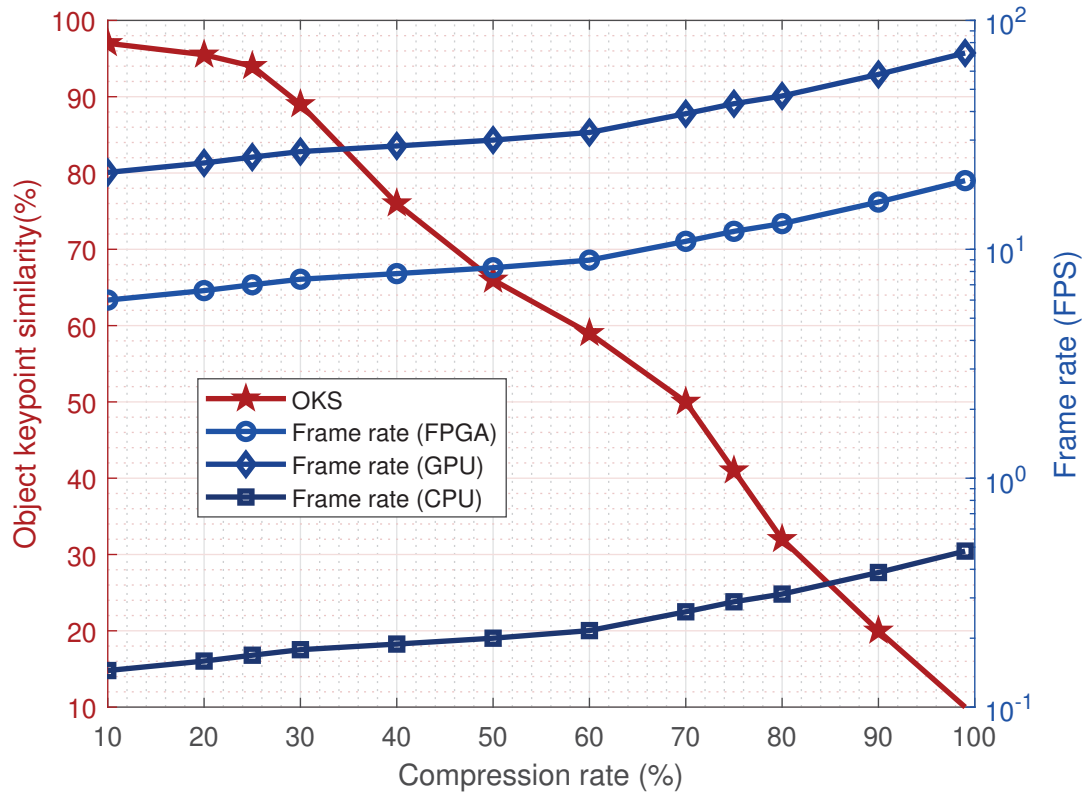


Figure 49: Performance comparison among different platforms for the OpenPose algorithm (optimisation normalized to prevent accuracy loss).

The OpenPose algorithm was the only one which presented a lower speed performance when compared to the GPU due to its saturated status. However, it is still faster than the genetic CPU unit, as seen in fig. 49. Since the network has been saturated over the limited hardware resources to work correctly, any reduction in the number of neurons and synapses directly affects the accuracy since fundamental parts of the network are being taken away. Therefore, the accuracy decay over an increasing compression rate is much more significant than what was observed for the object detection algorithms.

4.5 Resource consumption

Energy performance was measured based on the methods described in 3.6.3 and 3.6.4 for the aforementioned DLCNNs. However, it was possible to estimate resource consumption in terms of used LUTs in a more localized fashion by using the High Level Synthesis which sets the number of resources depending on the direct number of logic gates, LUTs, DSPs directly specified by the hardware description language.

Every studied algorithm has its number of resources set in the methodology section

with the exception of the YOLO v4. The resource occupation of the YOLO v4 algorithm over the used FPGA can be seen in fig. 50.

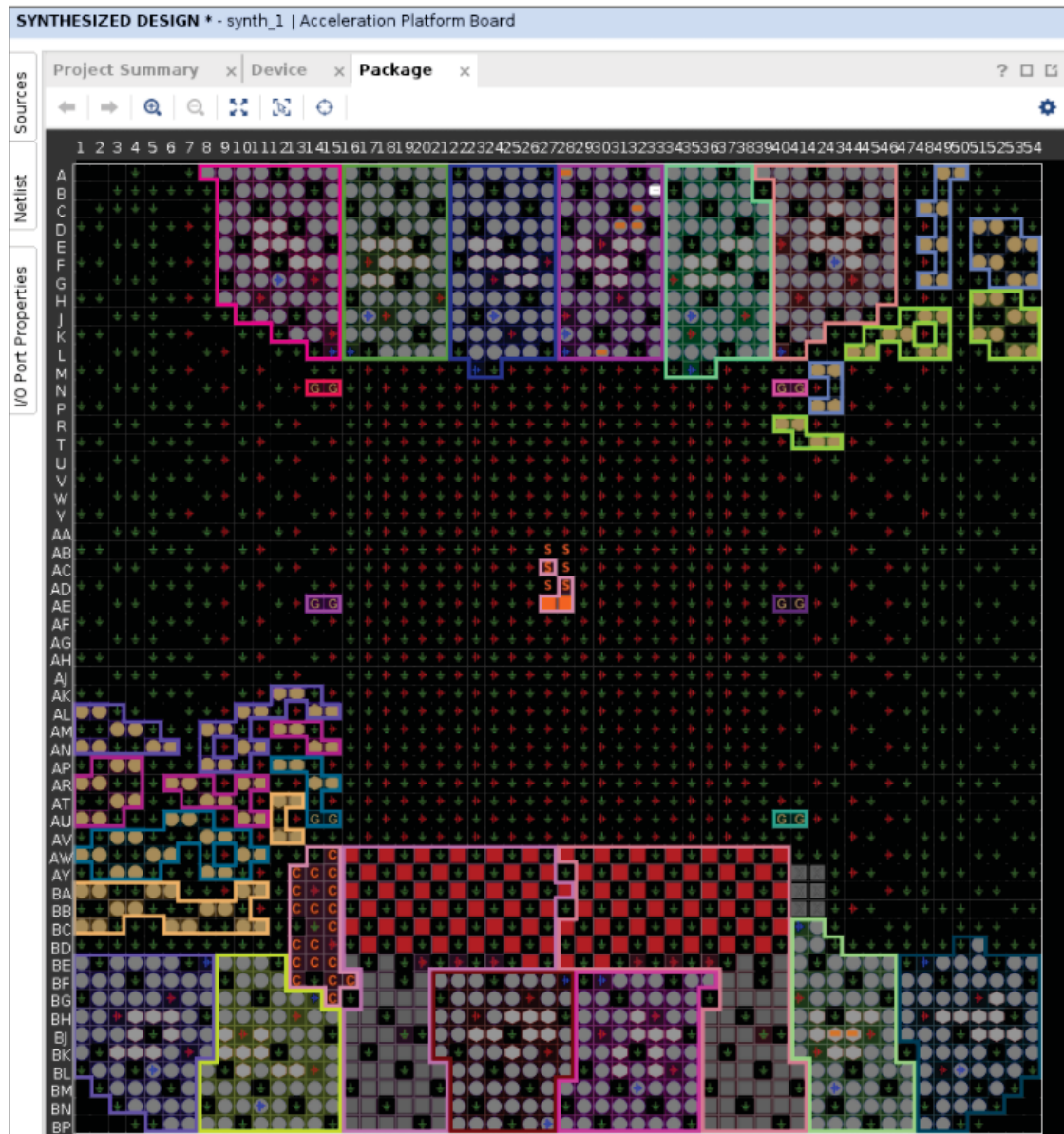


Figure 50: Global resource utilization in DPUCZDX8G FPGA.

The resource utilization for the YOLO v4 algorithm reaches 25.7 % of the total resource area. However, the total available resources vary for the number of elements available. It should be noted that the HLS design can use all the resources available on the evaluation board and not only the fixed set design for the DPU architecture used for the deployment method used to create the other researched algorithms. Figure 51 and 14 demonstrates the number of different resources used to implement the YOLO v4 algorithm.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	3	-	-	-
Expression	-	0	0	958	-
FIFO	-	-	-	-	-
Instance	49	149	27757	51281	0
Memory	129	-	0	0	0
Multiplexer	-	-	-	4688	-
Register	-	-	2549	-	-
Total	178	152	30306	56927	0
Available	624	1728	460800	230400	96
Utilization (%)	28	8	6	24	0

Table 14: Resource utilization for the YOLO v4 algorithm (total resources).

	BRAM	DSP	FF	LUT	Bits P0	Bits P1
YOL04_FPGA	178	152	30306	56927		
> I/O Ports(38)					1154	
> Instances(9)	49	149	27757	51281		
> Memories(65)	129		0	0	2064	
> Expressions(42)	0	0	0	958	719	708
> Registers(132)			2549		2577	
> Channels(0)	0		0	0	0	
> Multiplexers(374)	0		0	4688	3327	
> DSP(12)		3				

Figure 51: Resource utilization for the YOLO v4 algorithm.

This level of occupation enables the FPGA to be used for other applications in parallel, such as running most of the studied algorithms provided that enough space is available for them.

4.5.1 Power estimation

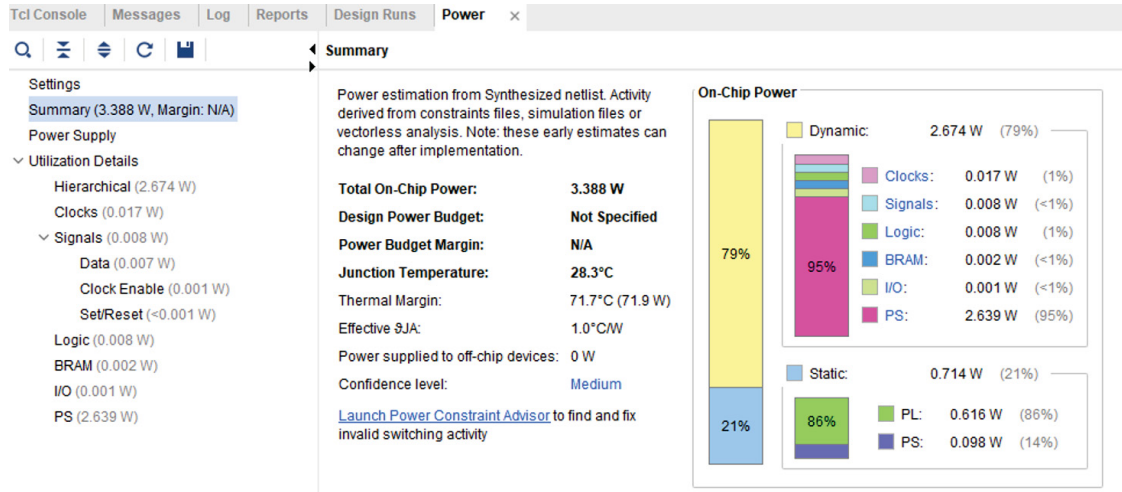


Figure 52: Power estimation for the YOLO v4 algorithm using analytical models.

Power utilization is ideally directed to the dynamic operation of the logic circuit. It should be noted that the implementation and deployment present a different result for the dynamic energy use than what is analytically presented by figure 52. The dynamic energy used for the YOLO v4 algorithm only measured by the powertop command is 0.213 W. Most of the energy is used to make the peripherals of implementation work.

4.5.2 Synthesis, implementation and routing

The process of deployment using the HLS design yields different sections for the RTL design. These sections are separated by the tool by the power consumption of each HDL section.

4.5.2.1 Synthesis

The synthesis process analysed the HDL description implemented using the high level synthesis. The representation of a multiplexer in the RTL circuit can be seen in fig. 53.

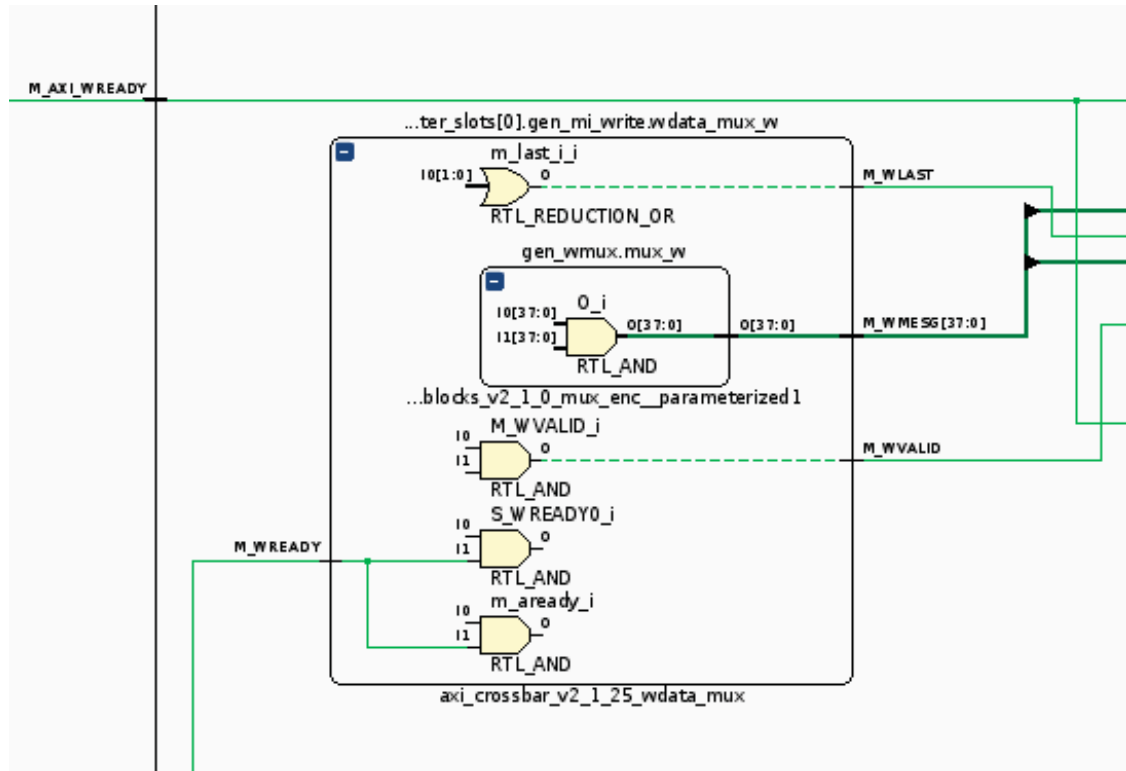


Figure 53: RTL representation of a multiplexer inside the implemented object detection design.

The interface of the 53-layer convolution backbone block used in YOLO v4 with the other processing blocks (proprietary Xilinx IP) is represented in fig. 54.



Figure 54: RTL representation of the 53-convolution backbone inside the implemented object detection design.

Figure 57 presents the aforementioned blocks.

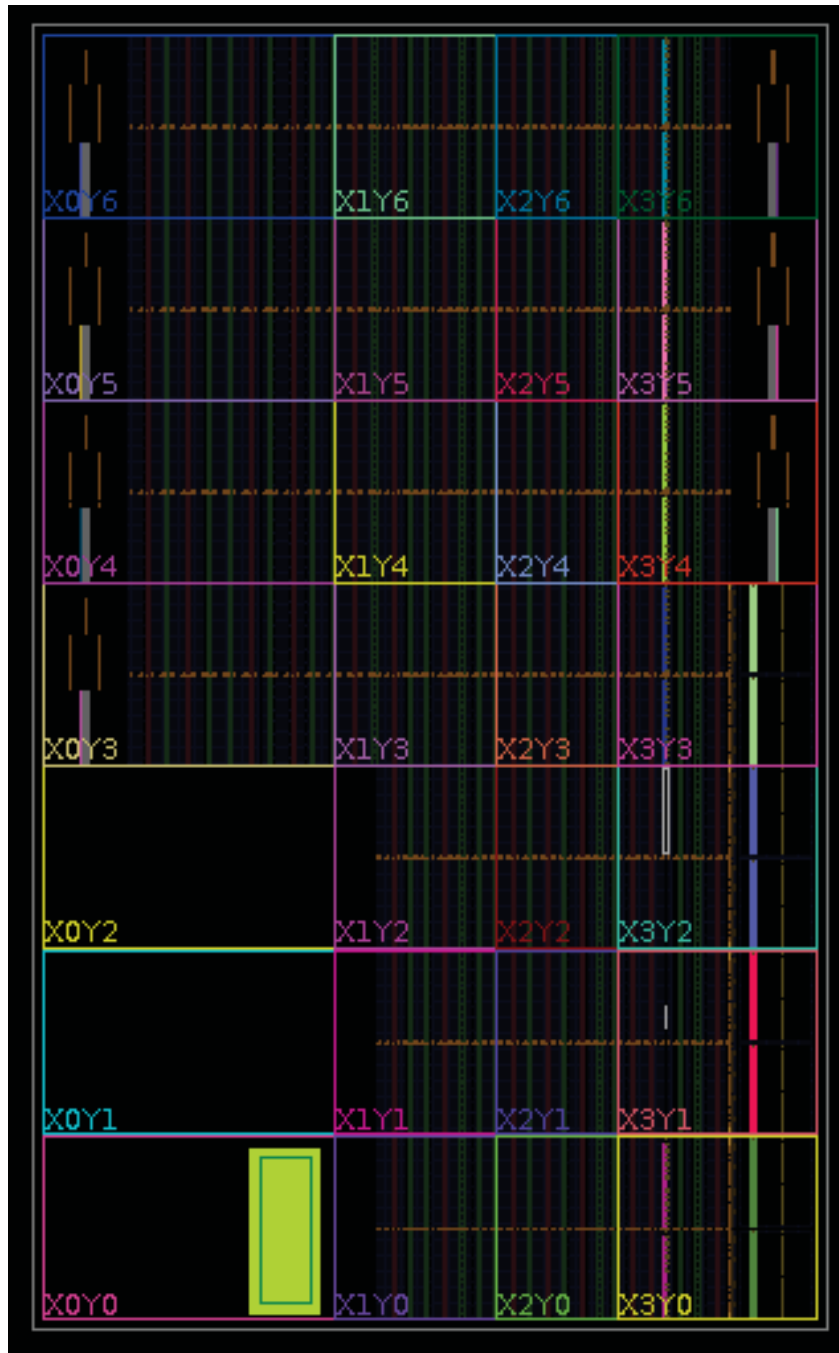


Figure 55: Representation of the different blocks used to synthesize the HDL design (Synthesis).

A representation of the entire synthesized design for the object detection algorithm represented in schematic form is shown in fig. 56.

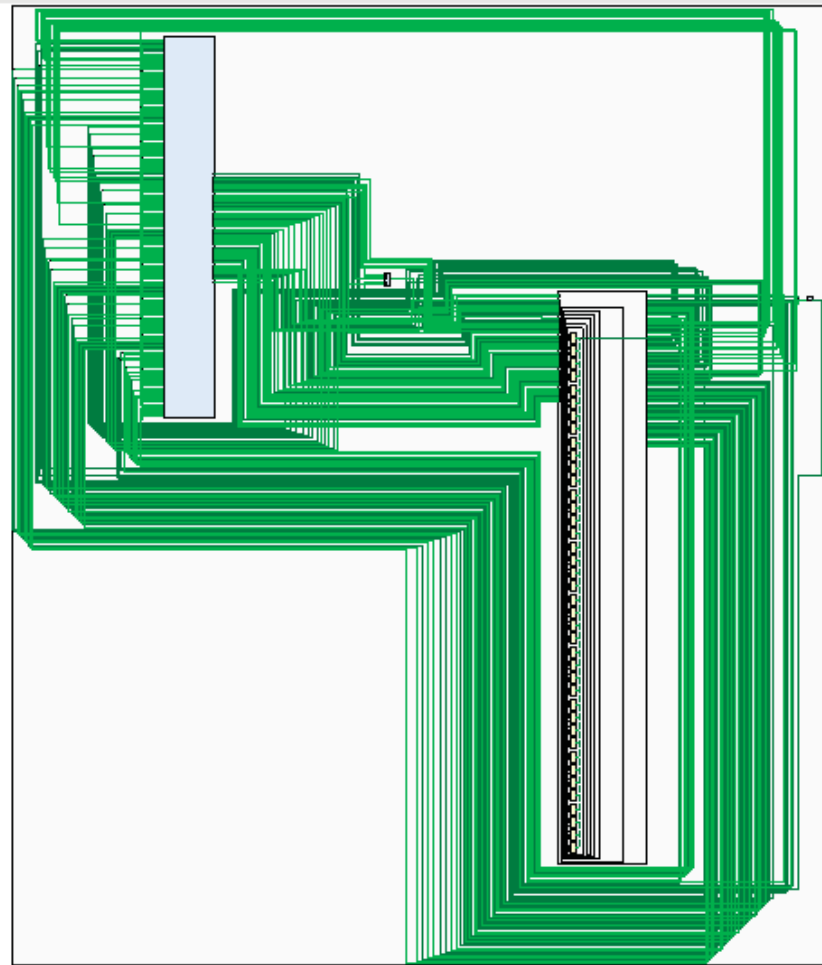


Figure 56: Schematic of the implemented design for the ZCU104 design.

4.5.2.2 Implementation and routing

The implemented design using the individual resource makeup for the ZCU104 FPGA is presented in fig. 56 after the synthesis shown in fig 57.

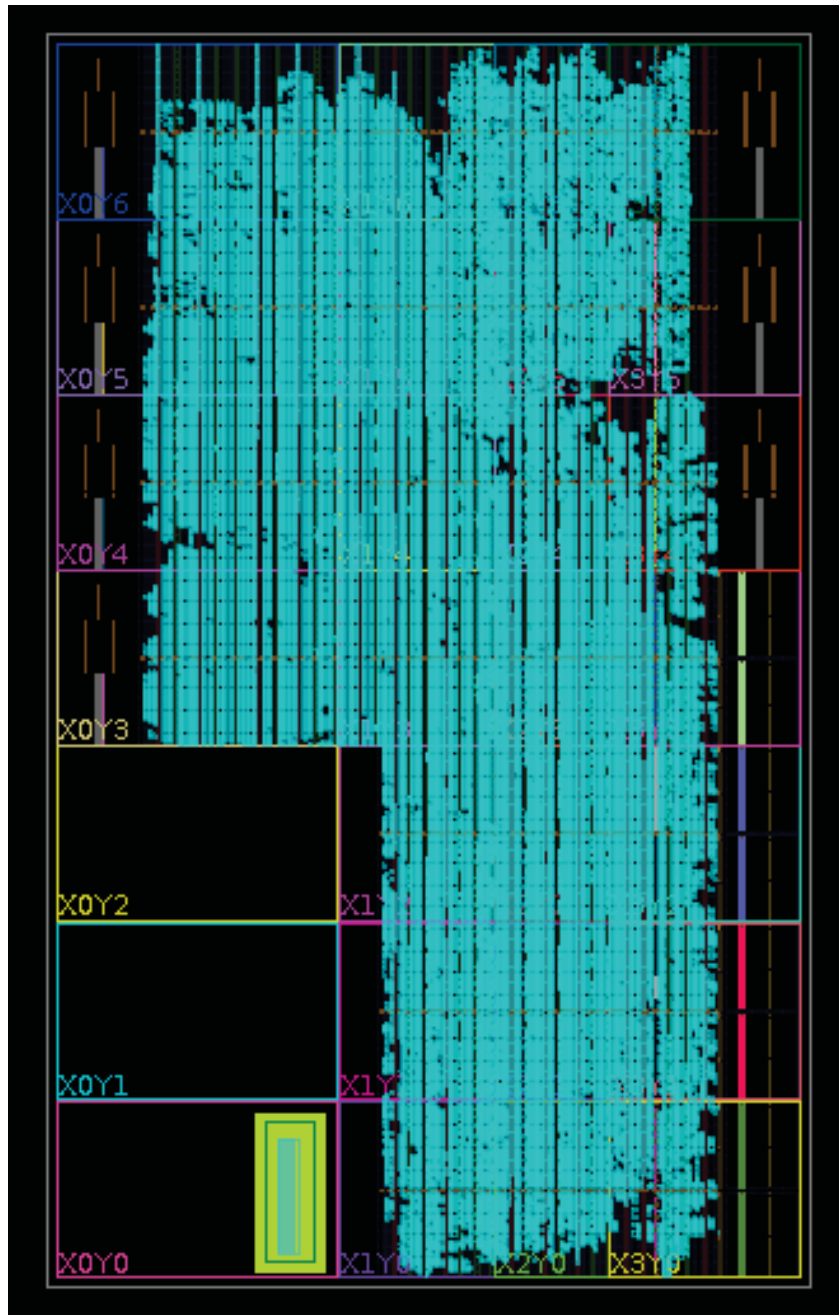


Figure 57: Implemented design visualization for the ZCU104 system.

A representation of the entire implemented design for the object detection algorithm represented in schematic form is shown in fig. 58.

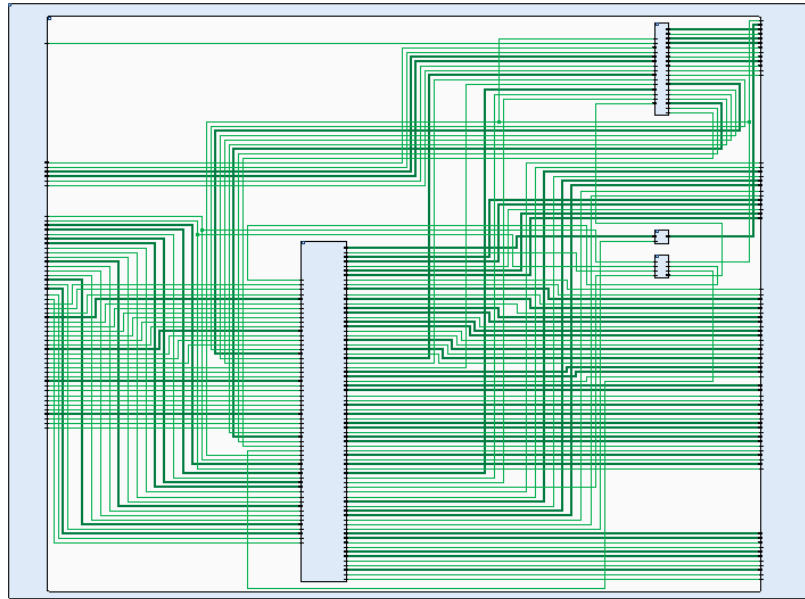


Figure 58: Schematic of the implemented design for the ZCU104 design.

Compared to the synthesized design observed in 56, the implementation process visibly reduces the complexity of the design by accommodating the VHDL design into the available resources on the target device. The resource utilization before and after implementation can be seen in fig. 59.

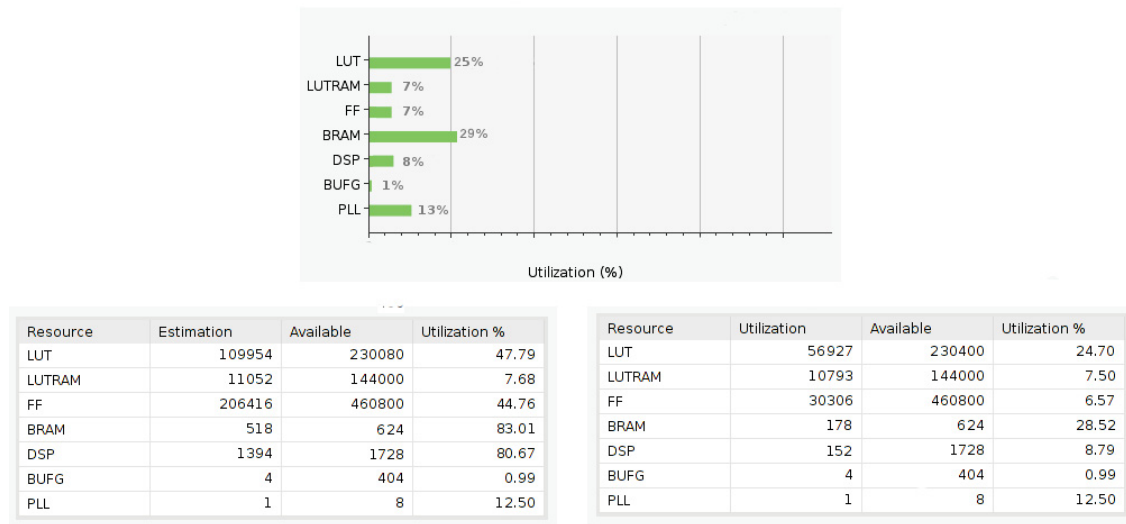


Figure 59: Resource utilization before and after implementation.

The number of resources used after implementation is reduced by allocating similar ports and devices to standard pipelines, buffers, and additional elements evaluated automatically

by the tool.

The implementation and routing of the global circuit cannot be represented accurately due to the immense quantity of individual elements. However, fig. 61 represents a small enough convolution block where the routing lines can be seen.

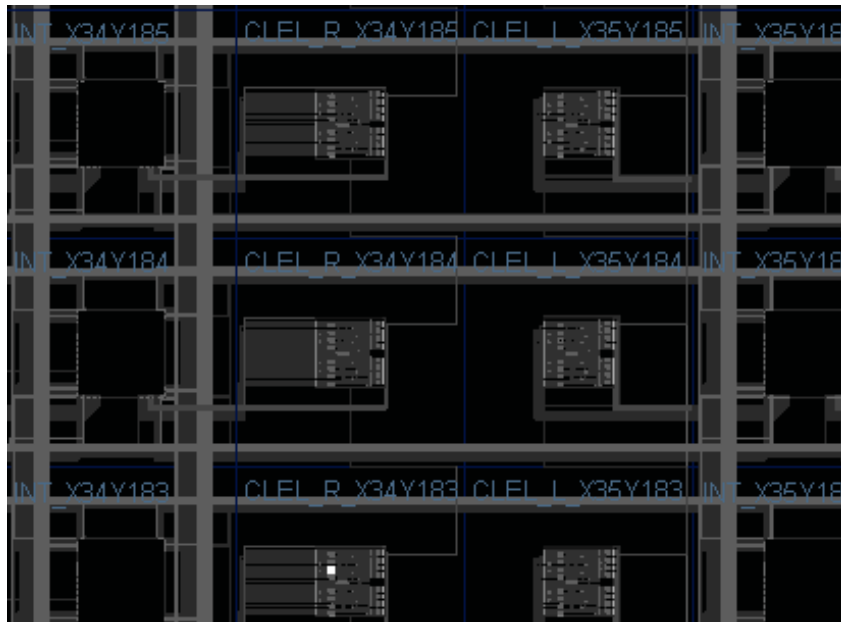


Figure 60: Routed elements of a convolution filter for the YOLO v4 algorithm (implementation step).

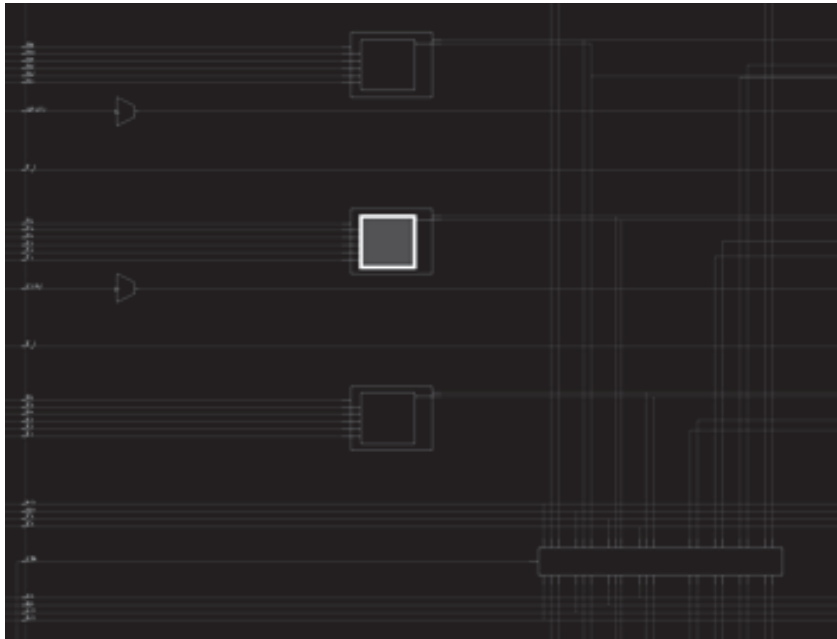


Figure 61: Routed elements of a convolution filter for the YOLO v4 algorithm (zoomed on individual elements).

The implementation step differs from the RTL design in the number of routed elements and the number of utilized elements depending on the positioning of each logical element required to deploy the application effectively. For example, fig 61 uses a significant number of resources with multi-functional logic capabilities – which differs from the RTL design that requires each logical gate to be distinctly separated.

5 Conclusion and future work

5.1 Discussion

The developed work highlights the advantages of implementing applications that rely on real-time CNN's processing into FPGA devices by designing a customized logic circuit, significantly increasing physical surface occupation, speed, and energy consumption. The reduction of resource usage also reduces the development cost for vehicular-related solutions. Future works will combine different CNN algorithms and camera drivers into the global logical circuit to reduce the latency of the object detection and blurring process, compacting the operations of the embedded optical sensor into a single FPGA.

The TensorFlow framework was used to implement the YOLOv4 model in Python. TensorFlow models store their weights as 32-bit floating-point numbers, resulting in sophisticated 32-bit floating-point calculations. Weights are quantized from 32-bit floating-point to 8-bit integers using the optimizer tool to decrease the complexity of these operations and allow a faster deployment. As a result, operations are done in 8 bits, which reduces the amount of space required. Another advantage is that weight sizes are lowered by a factor of four, resulting in lower bandwidth needs. In addition, the quantization scheme, used to create a correspondence between the bit-representation of values and their interpretation as real mathematical numbers, adds additional complexity to the hardware accelerator.

Network size and complexity reduction via compression are efficient ways to increase the algorithm's speed. However, excessive pruning can lead to a useless application since a minimum size is required to produce satisfactory defections. It should be noted that the inclusion of an FPGA in an embedded system can make use of minimal networks in order to consume little power. Therefore, the compression rate should be analyzed to reach a target speed for the target application.

Analyzing the accelerator's current bandwidth needs reveals a possible bottleneck that results in needlessly high Random Access Memory (RAM) accesses. The overall storage needs for both maps and filters for each layer and the equivalent DRAM accesses performed by the accelerator. The Accesses/Storage columns display the average number of accesses per storage element. Each RAM access retrieves 32 bits of data, implying that each DRAM access may fetch four maps or filters. However, this is difficult since the global system memory cannot retain all fetched data, requiring some data to be re-accessed – as shown for the OpenPose algorithm. Because the algorithm possesses too many hardware requirements, it cannot be fully parallelized by the available resources. Some of the operations are serialized in order to execute the whole algorithm, significantly increasing the time necessary to analyze each frame – making the processing speed lower than the GPUs and closer to the CPU performance.

Running the YOLOv4 model on the board's CPU (ARM Cortex A53) identifies the software application's computationally expensive operations. Convolutional layers consume

99.67 percent of the total execution time of all layers, according to used tool's built-in profiler. Furthermore, the fundamental computation of the convolutional approach, as seen in, consists of multiply and accumulates (MACs), which may be substantially parallelized. As a result, hardware acceleration of convolutional layers represents a success.

The Ethernet connection utilizing the UDP protocol is used to link the host PC to the ZCU board. The UDP protocol is implemented by the Xilinx Development Kit's LightWeight IP (lwIP) open source TCP/IP networking stack. The host PC may send photos to the ZCU104, and the board can respond with predictions. The interface is only utilized for demonstration reasons which means that the real execution time of the model is monitored on the OS between inference cycles.

The model performance using Vitis AI greatly depends on the target model and internal target detection parameters. The implementation of the YOLO v3 and YOLO v4 have demonstrated that the number of identifiable models.

Further optimization for the developed work can be implemented, focusing on moving the resource utilization to an Application Specific Integrated Circuit (ASIC). The ASIC device will have only the minimal elements necessary to compose the circuit for a specific DLCNN. The created HDL code can be adapted to operate with tools to map the physical and transistor-level structure (microelectronics) in a target technology. An ASIC will be developed using simulation and design tools from *Cadence Design Systems* in which each logical gate – using a specific smelter technology – will be tested for a physical, digital integrated circuit performance analysis. A physical circuit could be produced using a designated smelter to test it using a measurement laboratory to analyze the ASIC's performance and compare it to the results observed in this project.

It is not fully understood how the optimization differs for different convolutional algorithms. However, further analysis of which layers are more significant for each target DLCNN can help developers understand how to project highly efficient networks in detecting the environmental context for element-specific detection.

Further development using the same materials can use adaptive AI computer vision technology to change the structure of the FPGA in real-time for different targets. The target FPGA board must have at least 50 % of resource occupation to operate, changing its internal structure to accommodate a plastic mode operation.

Implementation of smaller, more efficient devices also opens up the development of both cloud and edge computing applications in the vehicle field and most fields that deal with computer vision and surveillance.

References

- [1] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICCESS)*, 2019, pp. 1–8.
- [2] J.-Y. Sung, S.-B. Yu, and S.-h. P. Korea, "Real-time automatic license plate recognition system using yolov4," in *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, 2020, pp. 1–3.
- [3] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.
- [4] W. Chen, S. Lu, B. Liu, G. Li, and T. Qian, "Detecting citrus in orchard environment by using improved yolov4," *Scientific Programming*, vol. 2020, pp. 1–13, 11 2020.
- [5] G. Papandreou, T. Zhu, L.-C. Chen, S. Gidaris, J. Tompson, and K. Murphy, "Personlab: Person pose estimation and instance segmentation with a bottom-up, part-based, geometric embedding model," 2018.
- [6] Z. Jeffrey, X. Zhai, F. Bensaali, R. Sotudeh, and A. Ariyaeinia, "Automatic number plate recognition system on an arm-dsp and fpga heterogeneous soc platforms," 08 2013, pp. 1–9.
- [7] L. Y. Chan, A. Zimmer, J. L. d. Silva, and T. Brandmeier, "European union dataset and annotation tool for real time automatic license plate detection and blurring," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, 2020, pp. 1–6.
- [8] X. Inc., "Vitis ai optimizer user guide." [Online]. Available: <https://docs.xilinx.com/r/1.3-English/ug1333-ai-optimizer/Iterative-Pruning?tocId=ZKMWOJRZgO8ntUtr5GPTtQ>
- [9] V. Conti, L. Rundo, G. Billeci, C. Militello, and S. Vitabile, "Energy efficiency evaluation of dynamic partial reconfiguration in field programmable gate arrays: An experimental case study," *Energies*, vol. 11, p. 739, 03 2018.
- [10] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, "Random erasing data augmentation," 2017.
- [11] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," 2017.
- [12] K. K. Singh, H. Yu, A. Sarmasi, G. Pradeep, and Y. J. Lee, "Hide-and-seek: A data augmentation technique for weakly-supervised localization and beyond," 2018.
- [13] P. Chen, S. Liu, H. Zhao, and J. Jia, "Gridmask data augmentation," 2020.

- [14] G. Ghiasi, T.-Y. Lin, and Q. V. Le, “Dropblock: A regularization method for convolutional networks,” 2018.
- [15] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, “mixup: Beyond empirical risk minimization,” 2018.
- [16] S. Yun, D. Han, S. Chun, S. J. Oh, Y. Yoo, and J. Choe, “Cutmix: Regularization strategy to train strong classifiers with localizable features,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 6022–6031.
- [17] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel, “Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness,” 2019.
- [18] S. Woo, J. Park, J.-Y. Lee, and I.-S. Kweon, “Cbam: Convolutional block attention module,” in *ECCV*, 2018.
- [19] B. Hariharan, P. Arbeláez, R. Girshick, and J. Malik, “Hypercolumns for object segmentation and fine-grained localization,” 2015.
- [20] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature pyramid networks for object detection,” 2017.
- [21] D. Vriesman, M. E. Pederiva, J. Mario De Martino, A. B. Junior, A. Zimmer, and T. Brandmeier, “A fusion approach for pre-crash scenarios based on lidar and camera sensors,” in *2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring)*, 2021, pp. 1–5.
- [22] X. Inc., “Dpu configuration manual - xilinx inc.” [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/dpu_config.html
- [23] M. Aeberhard, S. Rauch, M. Bahram, G. Tanzmeister, J. Thomas, Y. Pilat, F. Himm, W. Huber, and N. Kaempchen, “Experience, results and lessons learned from automated driving on germany’s highways,” *IEEE Intelligent Transportation Systems Magazine*, vol. 7, no. 1, pp. 42–57, 2015.
- [24] A. Agnihotri, P. Saraf, and K. R. Bapnad, “A convolutional neural network approach towards self-driving cars,” 2019. [Online]. Available: <https://arxiv.org/abs/1909.03854>
- [25] H. Haavaldsen, M. Aasbø, and F. Lindseth, “Autonomous vehicle control: End-to-end learning in simulated urban environments,” *CoRR*, vol. abs/1905.06712, 2019. [Online]. Available: <http://arxiv.org/abs/1905.06712>
- [26] W. Jiang, L. Yang, E. H. Sha, Q. Zhuge, S. Gu, Y. Shi, and J. Hu, “Hardware/software co-exploration of neural architectures,” *CoRR*, vol. abs/1907.04650, 2019. [Online]. Available: <http://arxiv.org/abs/1907.04650>
- [27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga,

- S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>
- [28] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.
- [29] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh, "Openpose: Realtime multi-person 2d pose estimation using part affinity fields," 2019.
- [30] R. Laroca, E. Severo, L. A. Zanolensi, L. S. Oliveira, G. R. Goncalves, W. R. Schwartz, and D. Menotti, "A robust real-time automatic license plate recognition based on the yolo detector," *2018 International Joint Conference on Neural Networks (IJCNN)*, Jul 2018. [Online]. Available: <http://dx.doi.org/10.1109/IJCNN.2018.8489629>
- [31] G. Hsu, A. Ambikapathi, S.-L. Chung, and C.-P. Su, "Robust license plate detection in the wild," *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–6, 2017.
- [32] L. Xie, T. Ahmad, L. Jin, Y. Liu, and S. Zhang, "A new cnn-based method for multi-directional car license plate detection," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, pp. 507–517, 2018.
- [33] S. Montazzolli and C. Jung, "Real-time brazilian license plate detection and recognition using deep convolutional neural networks," *2017 30th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, pp. 55–62, 2017.
- [34] L. Zou, M. Zhao, Z. Gao, M. Cao, H. Jia, and M. Pei, "License plate detection with shallow and deep cnns in complex environments," *Complexity*, vol. 2018, pp. 1–6, 12 2018.
- [35] Z. Selmi, M. B. Halima, and A. Alimi, "Deep learning system for automatic license plate detection and recognition," *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, vol. 01, pp. 1132–1138, 2017.
- [36] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.
- [37] X. Zhang, P. Shen, Y. Xiao, B. Li, Y. Hu, D. Qi, X. Xiao, and L. Zhang, "License plate-location using adaboost algorithm," 07 2010, pp. 2456 – 2461.
- [38] Q. Wu, H. Zhang, W. Jia, X. He, J. Yang, and T. Hintz, "Car plate detection using cascaded tree-style learner based on hybrid object features," in *2006 IEEE International Conference on Video and Signal Based Surveillance*, 2006, pp. 15–15.

- [39] H. Zhang, W. Jia, X. He, and Q. Wu, "Learning-based license plate detection using global and local features," vol. 2, 01 2006, pp. 1102–1105.
- [40] S.-Z. Wang and H.-J. Lee, "A cascade framework for a real-time statistical plate recognition system," *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 2, pp. 267–282, 2007.
- [41] K. Kim, K. Jung, and J. Kim, "Color texture-based object detection: An application to license plate localization," vol. 2388, 01 2002, pp. 293–309.
- [42] Y. Zhong and A. Jain, *Object localization using color, texture and shape*, 04 2006, vol. 33, pp. 277–294.
- [43] M. Mirmehdi and M. Petrou, "Segmentation of color textures," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 2, pp. 142–159, 2000.
- [44] H. ke Xu, F. hua Yu, J. hua Jiao, and H. sheng Song, "A new approach of the vehicle license plate location," in *Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'05)*, 2005, pp. 1055–1057.
- [45] R. Zunino and S. Rovetta, "Vector quantization for license-plate location and image coding," *IEEE Transactions on Industrial Electronics*, vol. 47, no. 1, pp. 159–167, 2000.
- [46] C. Anagnostopoulos, I. Anagnostopoulos, V. Loumos, and E. Kayafas, "A license plate-recognition algorithm for intelligent transportation system applications," *IEEE Transactions on Intelligent Transportation Systems*, vol. 7, no. 3, pp. 377–392, 2006.
- [47] D. Kaushik, H.-U. Chae, and K.-H. Jo, "Vehicle license plate detection method based on sliding concentric windows and histogram," *Journal of Computers*, vol. 4, 08 2009.
- [48] Y.-R. Wang, W.-H. Lin, and S.-J. Horng, "A sliding window technique for efficient license plate localization based on discrete wavelet transform," *Expert Syst. Appl.*, vol. 38, pp. 3142–3146, 04 2011.
- [49] H. Caner, H. S. Gecim, and A. Z. Alkar, "Efficient embedded neural-network-based license plate recognition system," *IEEE Transactions on Vehicular Technology*, vol. 57, no. 5, pp. 2675–2683, 2008.
- [50] C.-T. Hsieh, Y.-S. Juan, and K.-M. Hung, "Multiple license plate detection for complex background," in *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, vol. 2, 2005, pp. 389–392 vol.2.
- [51] S. Yohimori, Y. Mitsukura, M. Fukumi, N. Akamatsu, and N. Pedrycz, "License plate detection system by using threshold function and improved template matching method," in *IEEE Annual Meeting of the Fuzzy Information, 2004. Processing NAFIPS '04.*, vol. 1, 2004, pp. 357–362 Vol.1.
- [52] W. Jia, H. Zhang, X. He, and Q. Wu, "Gaussian weighted histogram intersection for license plate classification," vol. 3, 01 2006, pp. 574–577.

- [53] W. Jia, H. Zhang, X. He, and M. Piccardi, "Mean shift for accurate license plate localization," in *Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005.*, 2005, pp. 566–571.
- [54] W. Jia, H. Zhang, and X. He, "Region-based license plate detection," *Journal of Network and Computer Applications*, vol. 30, no. 4, pp. 1324–1333, 2007, special issue on Information technology. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1084804506000762>
- [55] F. Wang, L. Man, B. Wang, Y. Xiao, W. Pan, and X. Lu, "Fuzzy-based algorithm for color recognition of license plates," *Pattern Recognition Letters*, vol. 29, pp. 1007–1020, 05 2008.
- [56] G. Heo, M. Kim, I. Jung, D.-R. Lee, and I.-S. Oh, "Extraction of car license plate regions using line grouping and edge density methods," in *2007 International Symposium on Information Technology Convergence (ISITC 2007)*, 2007, pp. 37–42.
- [57] H.-J. Lee, S.-Y. Chen, and S.-Z. Wang, "Extraction and recognition of license plates of motorcycles and vehicles on highways," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 4, 2004, pp. 356–359 Vol.4.
- [58] T. Duan, D. Tran, P. Tran, and N. Hoang, "Building an automatic vehicle license-plate recognition system," 02 2005.
- [59] A. Toshev and C. Szegedy, "DeepPose: Human pose estimation via deep neural networks," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, jun 2014. [Online]. Available: <https://doi.org/10.1109%2Fcvpr.2014.214>
- [60] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," 2016. [Online]. Available: <https://arxiv.org/abs/1603.06937>
- [61] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "Neuralpower: Predict and deploy energy-efficient convolutional neural networks," 2017.
- [62] E. Buber and B. Diri, "Performance analysis and cpu vs gpu comparison for deep learning," 10 2018, pp. 1–6.
- [63] M.-H. Ho, Y.-Q. Ai, T. Chau, S. Yuen, C. Choy, P. Leong, and K.-P. Pun, "Architecture and design flow for a highly efficient structured asic," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, pp. 424–433, 03 2013.
- [64] K. Majumder and U. Bondhugula, "A flexible fpga accelerator for convolutional neural networks," 2019.
- [65] V. Sunder, "Development of state of the art ai vision algorithm on xilinx alveo u-200 fpga cloud and cpu+gpu platform," *International Journal of Engineering and Technical Research*, vol. V9, 07 2020.
- [66] L. Huang, G. Teng, D. Li, Z. Wang, and K. Wang, "A highly configurable high-level synthesis functional pattern library," *Electronics*, vol. 10, p. 532, 02 2021.

- [67] E. Cengil, A. Çınar, and E. Özbay, “Image classification with caffe deep learning framework,” in *2017 International Conference on Computer Science and Engineering (UBMK)*, 2017, pp. 440–444.
- [68] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” 2019.
- [69] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, “Yolov4: Optimal speed and accuracy of object detection,” 2020.
- [70] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun, “Yolox: Exceeding yolo series in 2021,” 2021.
- [71] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.
- [72] J. Schmidhuber, “Deep learning in neural networks: An overview,” *CoRR*, vol. abs/1404.7828, 2014. [Online]. Available: <http://arxiv.org/abs/1404.7828>
- [73] Y. Lecun, K. Kavukcuoglu, and C. Farabet, “Convolutional networks and applications in vision,” 05 2010, pp. 253–256.
- [74] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang, “Phoneme recognition using time-delay neural networks,” *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 37, pp. 328 – 339, 04 1989.
- [75] D. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.
- [76] W. Zhang, J. Tanida, K. Itoh, and Y. Ichioka, “Shift-invariant pattern recognition neural network and its optical architecture,” in *Proceedings of annual conference of the Japan Society of Applied Physics*, 1988, pp. 2147–2151.
- [77] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.
- [78] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010*. Springer, 2010, pp. 177–186.
- [79] C. J. Burges, “A tutorial on support vector machines for pattern recognition,” *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.
- [80] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

- [81] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, no. 5786, pp. 504–507, 2006.
- [82] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and L. Fei-Fei, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, 09 2014.
- [83] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and L. Fei-Fei, “Imagenet large scale visual recognition challenge,” *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: <http://arxiv.org/abs/1409.0575>
- [84] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” *CoRR*, vol. abs/1311.2901, 2013. [Online]. Available: <http://arxiv.org/abs/1311.2901>
- [85] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2014. [Online]. Available: <https://arxiv.org/abs/1409.1556>
- [86] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” 2014. [Online]. Available: <https://arxiv.org/abs/1409.4842>
- [87] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015. [Online]. Available: <https://arxiv.org/abs/1512.03385>
- [88] M. Lin, Q. Chen, and S. Yan, “Network in network,” 2013. [Online]. Available: <https://arxiv.org/abs/1312.4400>
- [89] Y. Zhang, “A better autoencoder for image: Convolutional autoencoder,” 2018.
- [90] K. Chowdhary, “Natural language processing,” *Fundamentals of artificial intelligence*, pp. 603–649, 2020.
- [91] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” 2016.
- [92] Tensorflow, “Tfjs-models/posenet at master · tensorflow/tfjs-models.” [Online]. Available: <https://github.com/tensorflow/tfjs-models/tree/master/posenet>
- [93] A. Menon and B. Omman, “Detection and recognition of multiple license plate from still images,” in *2018 International Conference on Circuits and Systems in Digital Enterprise Technology (ICCSDET)*, 2018, pp. 1–5.
- [94] I. Shafi, I. Hussain, J. Ahmad, P. W. Kim, G. S. Choi, I. Ashraf, and S. Din, “License plate identification and recognition in a non-standard environment using neural pattern matching,” *Complex & Intelligent Systems*, pp. 1–13, 2021.
- [95] K. D. Adedayo and A. O. Agunloye, “Real-time automated detection and recognition of nigerian license plates via deep learning single shot detection and optical character recognition,” *Computer and Information Science*, 2021.

- [96] N. Bellas, S. Chai, M. Dwyer, and D. Linzmeier, “Fpga implementation of a license plate recognition soc using automatically generated streaming accelerators.” 01 2006.
- [97] J. Mistry, A. K. Misraa, M. Agarwal, A. Vyas, V. M. Chudasama, and K. P. Upla, “An automatic detection of helmeted and non-helmeted motorcyclist with license plate extraction using convolutional neural network,” in *2017 Seventh International Conference on Image Processing Theory, Tools and Applications (IPTA)*, 2017, pp. 1–6.
- [98] “Ai model zoo –.”
- [99] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft coco: Common objects in context,” 2014, cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list. [Online]. Available: <http://arxiv.org/abs/1405.0312>
- [100] M. R. Ronchi and P. Perona, “Benchmarking and error diagnosis in multi-instance pose estimation,” 07 2017.
- [101] M. Everingham, L. Gool, C. K. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes (voc) challenge,” *Int. J. Comput. Vision*, vol. 88, no. 2, p. 303–338, jun 2010. [Online]. Available: <https://doi.org/10.1007/s11263-009-0275-4>
- [102] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, “Generalized intersection over union,” June 2019.
- [103]
- [104] F. Murtagh, “Multilayer perceptrons for classification and regression,” *Neurocomputing*, vol. 2, no. 5, pp. 183–197, 1991. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0925231291900235>
- [105] X. Wang and J. Song, “Iciou: Improved loss based on complete intersection over union for bounding box regression,” *IEEE Access*, vol. 9, pp. 105 686–105 695, 2021.
- [106] S. Yang, W. Xiao, M. Zhang, S. Guo, J. Zhao, and F. Shen, “Image data augmentation for deep learning: A survey,” 2022. [Online]. Available: <https://arxiv.org/abs/2204.08610>
- [107] D. Misra, “Mish: A self regularized non-monotonic activation function,” 2019. [Online]. Available: <https://arxiv.org/abs/1908.08681>
- [108] L. Zhuo, S. Tan, B. Li, and J. Huang, “Self-adversarial training incorporating forgery attention for image forgery localization,” *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 819–834, 2022. [Online]. Available: <https://doi.org/10.1109/TIFS.2022.3152362>
- [109] Z. Yao, Y. Cao, S. Zheng, G. Huang, and S. Lin, “Cross-iteration batch normalization,” 2020. [Online]. Available: <https://arxiv.org/abs/2002.05712>

- [110] C.-Y. Wang, H.-Y. M. Liao, I.-H. Yeh, Y.-H. Wu, P.-Y. Chen, and J.-W. Hsieh, “Cspnet: A new backbone that can enhance learning capability of cnn,” 2019. [Online]. Available: <https://arxiv.org/abs/1911.11929>
- [111] F. Shen and G. Zeng, “Weighted residuals for very deep networks,” 2016. [Online]. Available: <https://arxiv.org/abs/1605.08831>
- [112] L. B. Godfrey and M. S. Gashler, “A parameterized activation function for learning fuzzy logic operations in deep neural networks,” 2017. [Online]. Available: <https://arxiv.org/abs/1708.08557>
- [113] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
- [114] —, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
- [115] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression,” 2017.
- [116] N. Silberman, “Tf-slim: A lightweight library for defining, training and evaluating complex models in tensorflow,” 2017.
- [117] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgrency-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, “hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices,” 2021.
- [118] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, “Squeeze-and-excitation networks,” 2019.
- [119] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, “Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs,” 2017.
- [120] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, jan 2014.
- [121] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, “Regularization of neural networks using dropconnect,” in *30th International Conference on Machine Learning, ICML 2013*. International Machine Learning Society (IMLS), 2013, pp. 2095–2103, 30th International Conference on Machine Learning, ICML 2013 ; Conference date: 16-06-2013 Through 21-06-2013.

- [122] A. Shrivastava, A. K. Gupta, and R. B. Girshick, "Training region-based object detectors with online hard example mining," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 761–769, 2016.
- [123] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, "Distance-iou loss: Faster and better learning for bounding box regression," 2019.
- [124] M. A. Islam, S. Naha, M. Rochan, N. Bruce, and Y. Wang, "Label refinement network for coarse-to-fine semantic segmentation," 2017.
- [125] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," 2018.
- [126] K.-K. Sung and T. Poggio, "Example-based learning for view-based human face detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 1, p. 39–51, jan 1998. [Online]. Available: <https://doi.org/10.1109/34.655648>
- [127] "Vitis ai, <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>." [Online]. Available: <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>
- [128] "Vitis ai manual." [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_0/ug1414-vitis-ai.pdf
- [129] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," 2016.
- [130] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, p. 21–37, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2
- [131] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: <http://arxiv.org/abs/1506.02640>
- [132] A. L. Luiz, "Convolution in time and frequency domain (examples)," 2020. [Online]. Available: <https://colab.research.google.com/drive/1Jk6b8VpgKbXHqY4g5w8p5M6rVoHZGkUB?usp=sharing>
- [133] C. Wang, H. M. Liao, I. Yeh, Y. Wu, P. Chen, and J. Hsieh, "Cspnet: A new backbone that can enhance learning capability of CNN," *CoRR*, vol. abs/1911.11929, 2019. [Online]. Available: <http://arxiv.org/abs/1911.11929>
- [134] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *CoRR*, vol. abs/1406.4729, 2014. [Online]. Available: <http://arxiv.org/abs/1406.4729>
- [135] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," *CoRR*, vol. abs/1803.01534, 2018. [Online]. Available: <http://arxiv.org/abs/1803.01534>

- [136] C. Zheng, W. Wu, T. Yang, S. Zhu, C. Chen, R. Liu, J. Shen, N. Kehtarnavaz, and M. Shah, “Deep learning-based human pose estimation: A survey,” 2021.
- [137] N. Zhang, X. Wei, H. Chen, and W. Liu, “Fpga implementation for cnn-based optical remote sensing object detection,” *Electronics*, vol. 10, no. 3, 2021. [Online]. Available: <https://www.mdpi.com/2079-9292/10/3/282>
- [138] J. Matas and K. Zimmermann, “Unconstrained licence plate and text localization and recognition,” in *Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005.*, 2005, pp. 225–230.
- [139] S. Draghici, “A neural network based artificial vision system for licence plate recognition,” *International Journal of Neural Systems*, vol. 08, no. 01, pp. 113–126, 1997. [Online]. Available: <https://doi.org/10.1142/S0129065797000148>
- [140] B. Cho, S. Ryu, D. Shin, and J. Jung, “License plate extraction method for identification of vehicle violations at a railway level crossing,” *International Journal of Automotive Technology*, vol. 12, pp. 281–289, 04 2011.
- [141] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” 2019.

6 Appendix A – Bag of freebies

6.1 Random Erase

An area is randomly selected and covered with a random value. The selected area must be in the image. Part of the object can also be correctly recognized and classified, which forces the network to use local data for training and recognition, which increases the difficulty of training and improves the generalization ability of the model to a certain extent [10].

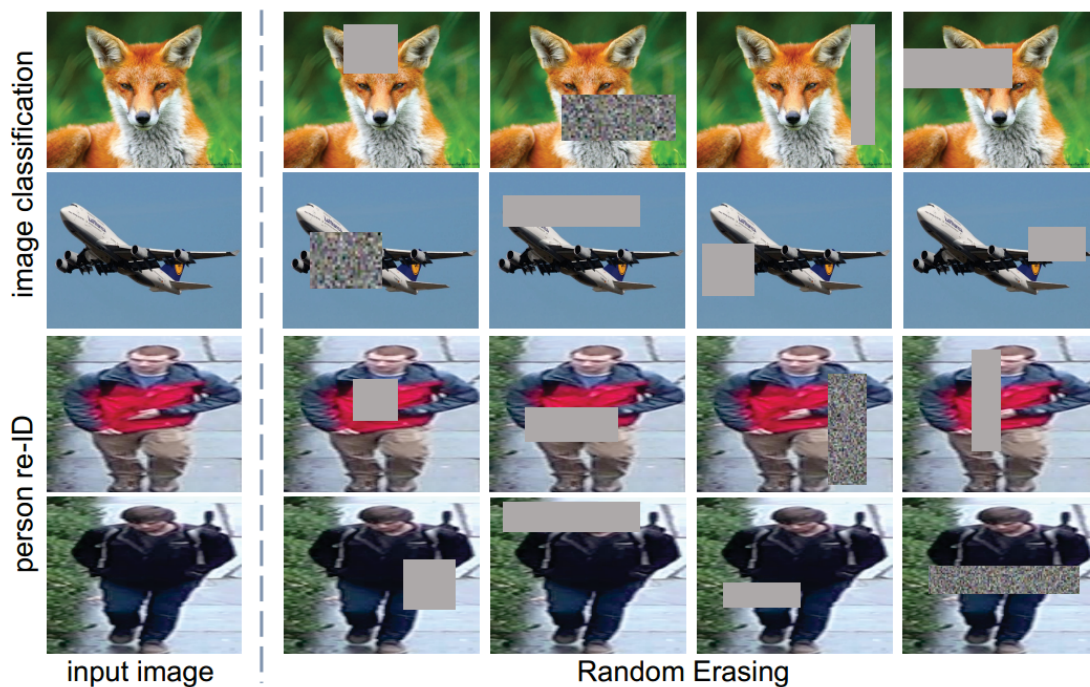


Figure 62: Example of random erase technique in 4 different images – [10].

6.2 Cut out

The idea is simpler than Random Erase. A fixed-size square area is randomly selected, and the pixel values are all filled with 0. However, in order to avoid the impact of 0 filling on the training data, the training data needs to be normalized, norm to 0, but It should be noted that the randomly selected area may not be in the image, that is to say, the square area may only be part of the processed image, thus simulating the erasing operation of any shape [11].

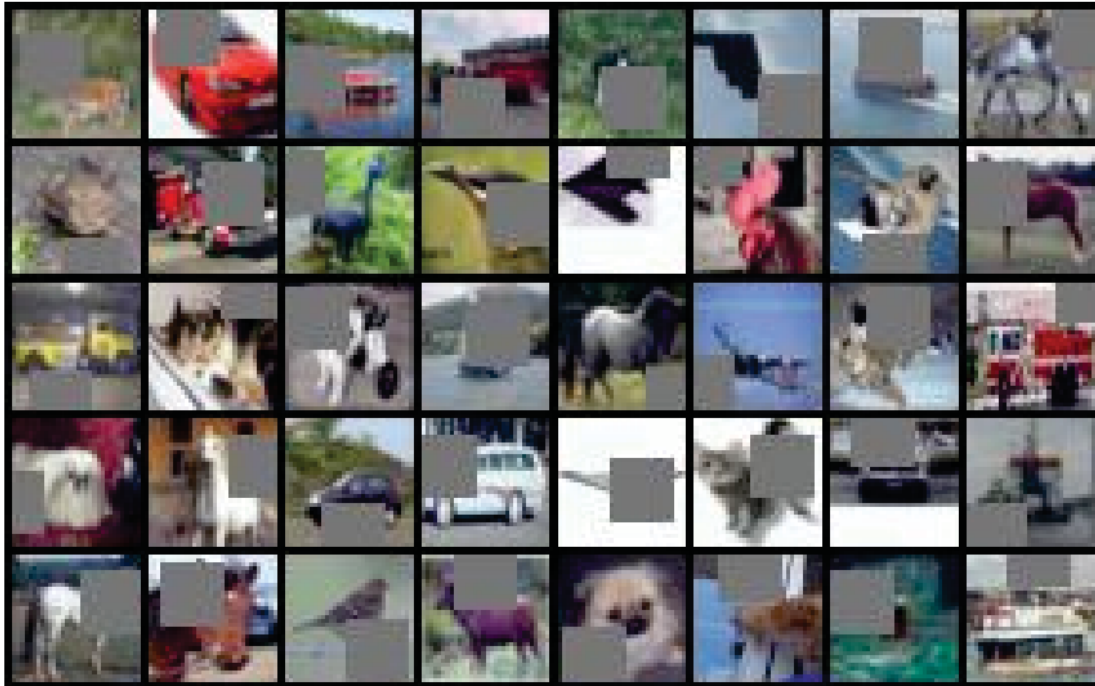


Figure 63: Example of the *cut out* technique used on the CIFAR-10 dataset – [11].

6.3 Hide-and-seek

The image is divided into $S \times S$ networks, and each network uses a certain probability for occlusion, thereby simulating the effects of Random Erase and CutOut. But why the occlusion value is set is more critical. If you fill in 0 or 255, it may change the data distribution of the training set. After calculation, the author uses the average of the entire data set to have the least impact.

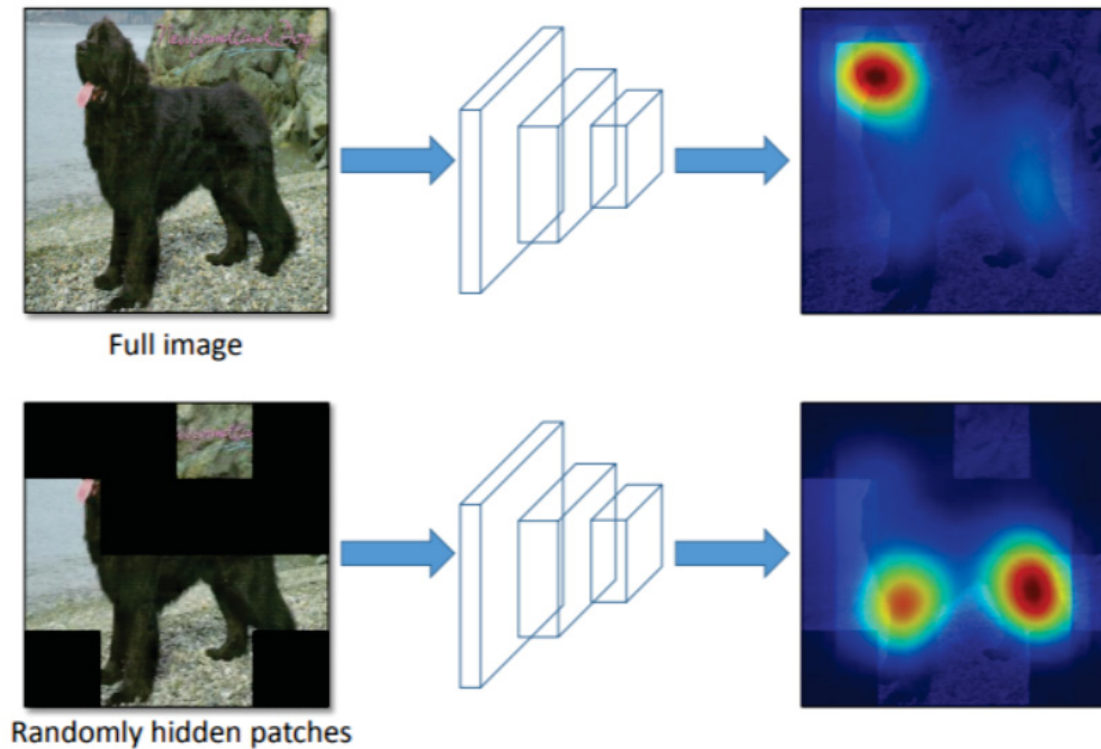


Figure 64: Example of the hide-and-seek technique on a dog image – [12].

6.4 Grid mask

This method is an improvement of the first three occlusion methods, because Random Erase, CutOut and Hide-and-Seek may all have a problem, which is to delete or keep all the discriminable areas of the image, introduce noise and have poor results.

How to solve the above problems and make a trade-off between deleting information and keeping information? Just structured drop operations, such as deleting square areas evenly, and controlling the density and size parameters to achieve a balance, the density is the number of squares, and the size is the size of the squares.

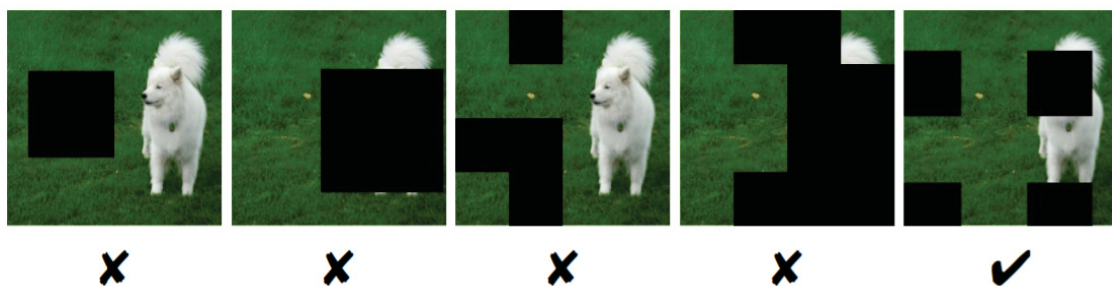


Figure 65: Examples of various iterations Cutout (first two on the left), Hide and Seek (subsequent two images) and the grid mask method (last) – [13].

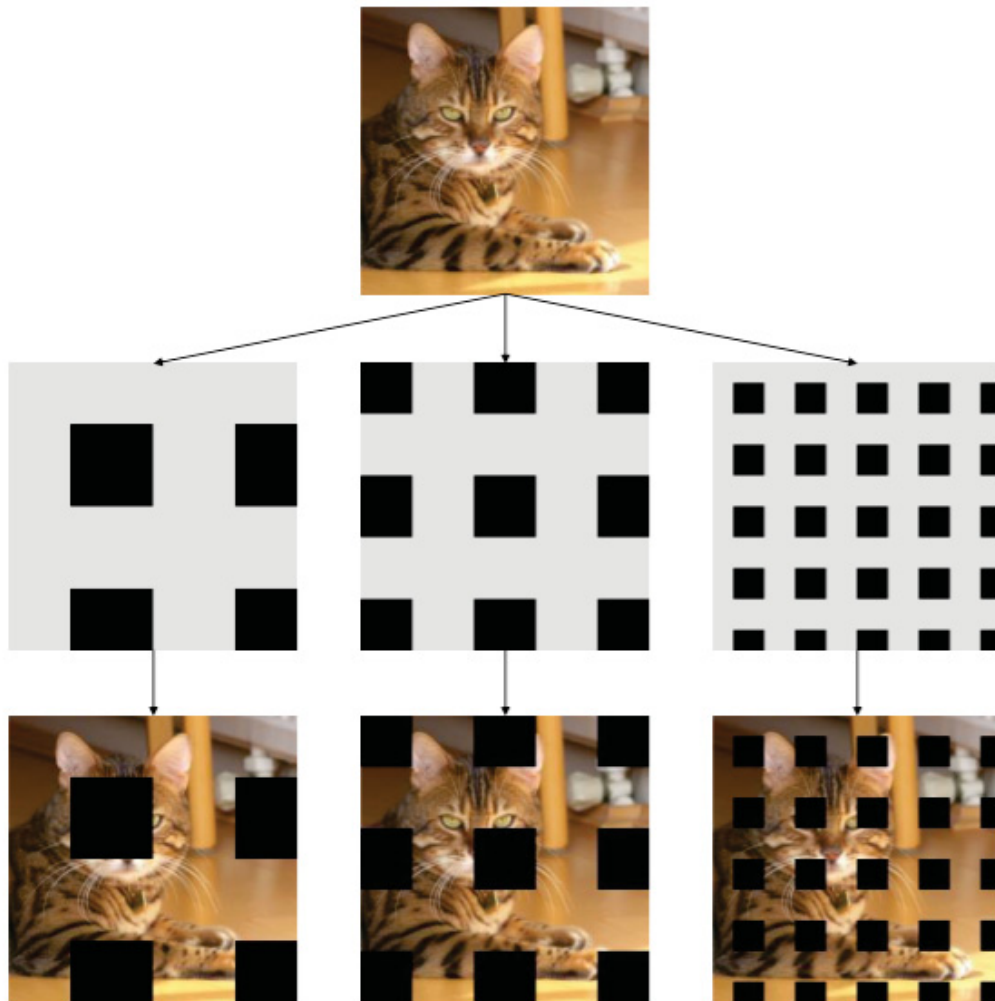


Figure 66: Application of the grid mask technique using different parameters for image erasure.

6.5 Drop out

Dropout will not explain much. During the training process, randomly discard some neurons according to the probability, and the output of the discarded neurons is set to 0 – [120].

6.6 Drop connect

DropConnect is a method proposed by Chinese people. The idea is similar to Dropout, but DropConnect sets the input weight of neurons to 0 during the training process. Because

DropConnect is more troublesome than Dropout, the effect is similar to Dropout, so it has not become a mainstream technology – [121].

6.7 Drop block

DropBlock is actually the promotion of Dropout on the convolutional layer. The author believes that Dropout is mostly used on the fully connected layer. It does not make much sense to use Dropout on the convolutional layer, because each feature map position has a receptive field range, convolution. The adjacent elements in the feature map of the layer share semantic information in space. Dropout of a certain pixel alone cannot reduce the feature range of feature map learning. In other words, the network can still learn the corresponding information through adjacent elements. Semantic information will not encourage the network to learn more robust features.

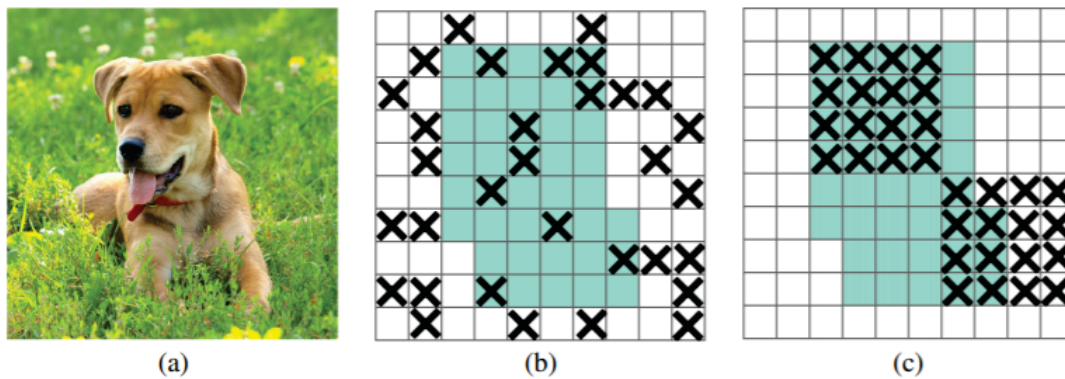


Figure 67: "(a) input image to a convolutional neural network. The green regions in (b) and (c) include the activation units which contain semantic information in the input image.

Dropping out activations at random is not effective in removing semantic information because nearby activations contain closely related information. Instead, dropping continuous regions can remove certain semantic information (e.g., head or feet) and consequently enforcing remaining units to learn features for classifying input image" – [14].

As shown in the fig. 67, the green part in the figure (b) is the semantic information. This figure simulates the practice of Dropout, randomly discarding some location features, but the author believes that this has no effect, because the network can still infer from adjacent location elements. Figure (c) is the method of DropBlock. Since discarding a single location is not enough, discard it piece by piece at will, forcing the network to learn more robust features.

6.8 Mix up

In the task of image classification, the core idea of *Mix Up* is to linearly mix one image with another in a certain ratio, and at the same time, mix the one-hot tags of the two images with the same ratio. The specific value of this ratio is produced by the beta distribution.

$$\tilde{x} = \lambda x_i + (1 - \lambda)x_j, \quad (21)$$

$$\tilde{y} = \lambda y_i + (1 - \lambda)y_j, \quad (22)$$

(x_i, y_i) and (x_j, y_j) are two random feature-target vectors chosen from the training data, and $\lambda \in [0, 1]$. The interpolation strength between feature-target pairings is controlled by the mixup hyper-parameter, which returns the Empirical risk minimization (ERM) principle as 0.

Mixup training is simple to implement and has a low computational burden. The few lines of code required to implement mixup training in PyTorch are shown in Figure 1a. Finally, we discuss other design options. First, preliminary studies show that convex combinations of three or more samples with weights taken from a Dirichlet distribution do not yield additional benefit, but instead raise the cost of mixup calculation. Second, our present method employs a single data loader to generate a single minibatch, which is then mixed after random shuffling. We discovered that this method worked just as effectively while lowering I/O needs. Third, interpolating simply across inputs with the same label did not result in the mixup performance enhancements.

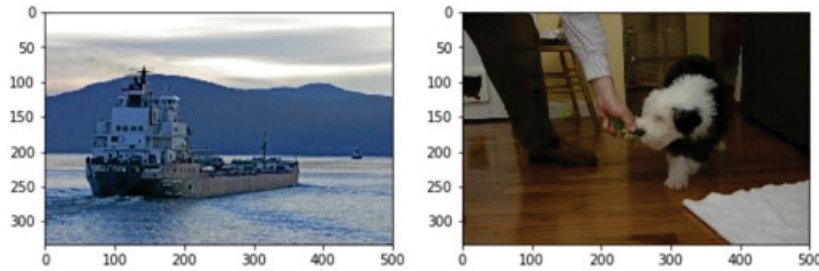


Figure 68: Original images to be used in the mix up process – [15].

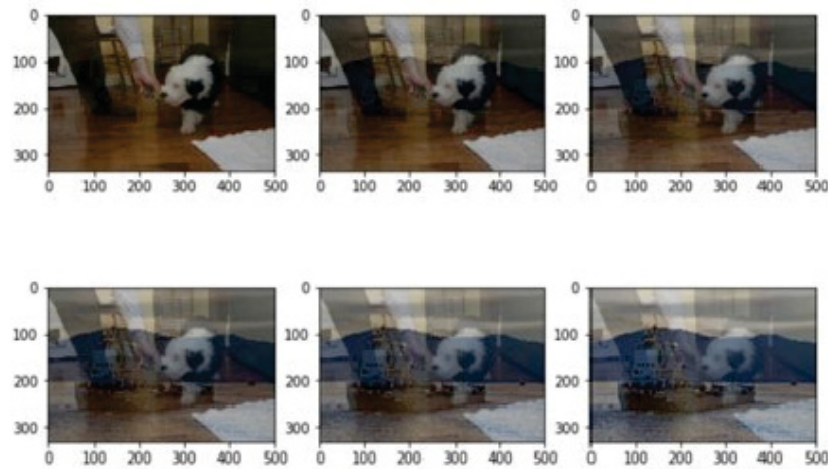


Figure 69: Mix up process of the two previous images – [15].

6.9 Cut mix

CutMix is a combination of CutOut and Mixup. It erases a part of the picture and randomly fills in the pixel values of other data areas in the training set.

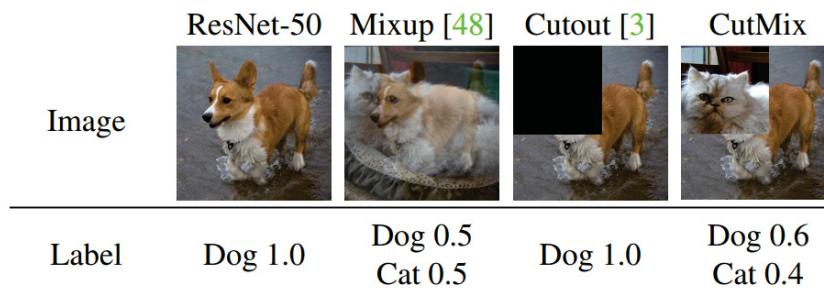


Figure 70: Demonstration of the cut mix process – [16].

6.10 General Adversarial Networks – GAN

The paper [17] found a conclusion: CNN training actually learned texture features (texture bias) rather than shape features, which is contrary to human cognition, so the stylized ImageNet data set was introduced through GAN to balance texture and shape bias. Improve the generalization ability of the model.

Therefore, the texture of the object can be randomly replaced by GAN, thereby enhancing the learning feature ability of the network.

The author used six pieces of data, normal cats, grayscale images, only contours, only

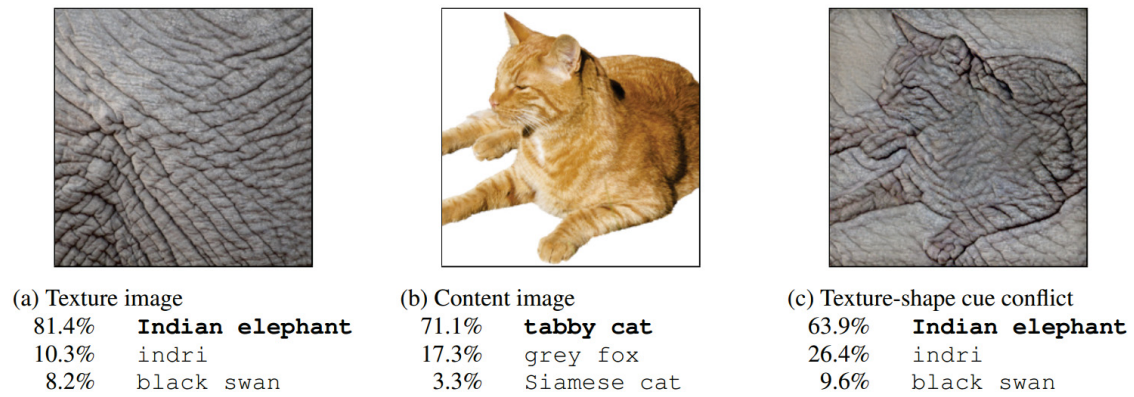


Figure 71: Classification of a texture and a cat using ResNet-50 and the resultant texture-shape cue conflict created by the style transference between the initial images – [17].

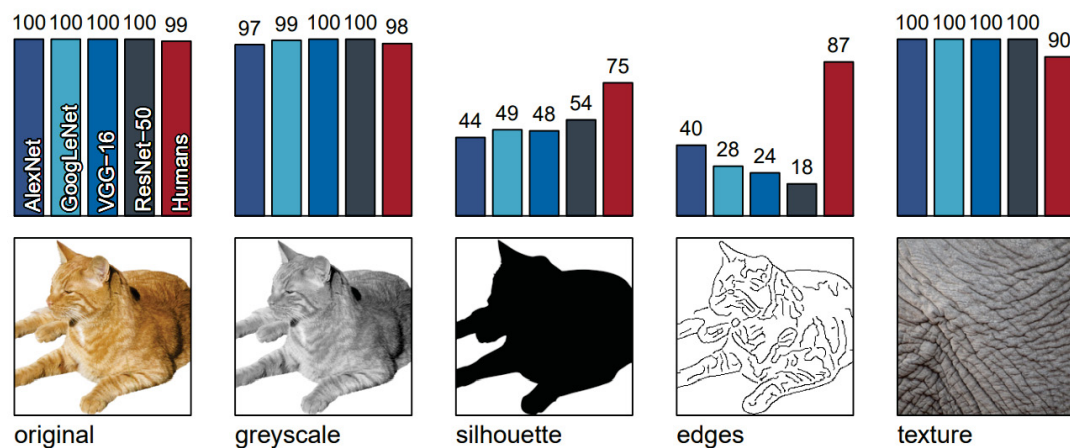


Figure 72: Demonstration of different accuracies for five different images without the cure conflict effect – [17].

edges, only textures without shapes, and cats with elephant-shaped textures. From the results, we can see that for regular pictures and grays, the network recognition rate is very high for degree graphs. For those that only contain contours and edges, the network recognition rate is low, and for those that only contain texture, the network recognition rate is also high. The network recognition rate is low for those containing contours and edges. For a picture with a texture of an elephant in the shape of a cat, the network has a higher probability of misrecognizing it as an elephant, so it can be inferred that what the network learns is a texture feature.

7 Appendix B – Bag of specials

7.1 Bag of specials

Increase of the receptive field: the accuracy of the target detection network is closely related to the size of the receptive field of the network. If the receptive field is too small, it may not recognize large-scale objects, resulting in a decrease in the accuracy of the target detection network. Attention development: Each position of the feature map and each feature map may have different degrees of importance to the detection results. The position on the feature map that contains the cat's outline, shape, and texture information is more important than the background information, so how to add it to the feature map? Weight information, so that the network can automatically learn to adjust the weight information, is the problem that the attention mechanism needs to solve. Feature fusion: A great feature of convolutional neural networks is that the high-level feature maps contain rich semantic information, but the location information will be significantly lost due to convolution and pooling operations. In contrast, the low-level feature maps have less semantic information. , because the number of operations of convolution and pooling is small and relatively complete position information is retained, target detection needs not only rich semantic information to distinguish the category of the object, but also needs accurate position information to locate the position of the object, so how to Fusion of low-level feature maps and high-level feature map information is the problem that features fusion needs to solve.

7.1.1 SPPNet

SPPNet is called a spatial pyramid pooling network. Its idea is actually very simple, which is to allow the network to adapt to image inputs of different scales and avoid cropping or scaling of the image, resulting in loss of position information. Since the convolutional layer will generate feature maps of different sizes when facing input images of different scales, it will bring great trouble to the training of the network. Therefore, SPPNet uses the pooling layer of the last layer of the convolutional layer. Replaced with a pyramid pooling layer and fixed the size of the input feature vector – [134].

7.1.2 ASPP

Before introducing Atrous Spatial Pyramid Pooling (ASPP), we need to introduce dilated convolution (hole convolution). As the name suggests, hole convolution is to increase the receptive field by filling holes in the convolution kernel.

- Ordinary convolution, 1-dilated convolution, the receptive field of the convolution kernel is 3×3 ;
- Hole convolution, 2-dilated convolution, the receptive field of the convolution kernel is 7×7 ;
- Hole convolution, 4-dilated convolution, the receptive field of the convolution kernel is 15×15 ;

The actual size of the atrous convolution is $RF = k + (k - 1) \times (rate - 1)$. Stacking atrous convolutions of different sizes can increase the size of the receptive field. $RF = RF1 + RF2 - 1$ Note that k = kernel size, $rate$ = dilated rate, the interval between adjacent weights is $rate - 1$. For example, a 2-dilated convolution is added after the 1-dilated convolution, and the receptive field can reach the size of 7×7 ($3 + 5 - 1$). If you use the ordinary 3×3 Convolution, i.e. requires a stack of 3 layers, $RF = (k - 1) \times layers + 1$.

ASPP is called the hole space pyramid pooling network. In fact, the feature map is sampled in parallel by hole convolution at different sampling rates, and the features extracted at each sampling rate are further processed and fused in different branches to produce the final result – [119].

7.1.3 Receptive Field Block – RFB

Different scales of conventional convolution + wormhole convolution are used on each branch of Receptive Field Block (RFB). Different receptive fields in pRFs are simulated by convolution kernel scales of conventional convolution. In fact, to simulate the ratio of pRF scale to eccentricity, to simulate the human visual perception mode. RFB uses 1×1 convolution to reduce the number of feature map channels and uses three regular convolutions of different sizes to extract the feature information of the feature map in parallel and then connects to the same dilated rate as the average convolution size. 3×3 hole convolution, and finally stitched together, using 1×1 convolution to restore the number of channels. (Similar to bottleneck structure, but with the addition of hole convolution) The difference between ASPP and the ASPP mentioned above is that when ASPP extracts features from the feature map, it uses a 3×3 regular convolution of the same size and then dilates. For hole convolutions with different rates, the extracted features are not highly discriminative. When RFB extracts features from the feature map, it uses three regular convolutions of different sizes, followed by hole convolutions, and the extracted features conform to human visual perception.

7.1.4 SE

The idea of SENet is actually very simple. It just wants to add weights to each feature map and let the network automatically learn to adjust the weight information.

The algorithm works by performing global average pooling on C feature maps of size $H \times W$ (the total dimension is $C \times H \times W$), which is the Global pooling operation in the above figure, and then the output is C features of size 1×1 Figure (total dimension is $C \times 1 \times 1$), following the fully connected layer, the fully connected layer is mainly to reduce the number of channels, and the size of r is related to the channel compression ratio. In the text, r is taken as 16, which is to reduce the number of channels to the original $1/16$, then perform ReLU activation, and then restore the number of channels to C through the fully connected layer. After Sigmoid activation (dimension is $C \times 1 \times 1$), it is directly multiplied with C feature maps of size $H \times W$, so that It can be understood that each feature map is added with weight information. Personally, I understand that the fully connected layer can be solved by 1×1 convolution, which is all to reduce the number of channels, and there is no difference – [118].

7.1.5 SAM

After everyone understands SE, it is estimated that they will inevitably think since weights can be added to the feature maps, can you add weights to the spatial positions inside the feature maps? The answer is obvious. Convolution Block Attention Module (CBAM) is an attention mechanism model proposed to solve the above problems. It is divided into CAM (Channel Attention Module) and the Spatial Attention Module (SAM). First, let's analyze the CAM module, which is almost the same as the SE module, except it adds a maximum global pooling. Then, the $C \times 1 \times 1$ feature vector and global average pooling are obtained by maximum global pooling. The obtained $C \times 1 \times 1$ feature vector passes through the MLP layer in turn (that is, the FC layer of SE, which is used to reduce the dimension first and then increase the dimension), and then add the output feature vector pixel by pixel, and then go through Sigmoid activation and feature Multiply graphs. (The only difference from SE is that an additional global maximum pooling is added) SAM is a module used to add an attention mechanism to the spatial position inside the feature map, assuming that the input feature map is still $C \times H \times W$ (that is, C A feature map with a size of $H \times W$), this time we perform maximum pooling with a channel number of C for each point (in $H \times W$) of the feature map, so that the size of the feature map output by the maximum pooling is $1 \times H \times W$, at the same time, average pooling with channel number C is also performed. The output feature map size is also $1 \times H \times W$. The feature map output by max-pooling and the feature map output by average pooling are spliced. A spliced feature map of $2 \times H \times W$ is formed, and then the channel dimension is reduced to a $1 \times H \times W$ output feature map through 1×1 convolution. Then the spatial attention weight is formed by sigmoid activation, and when combined with the original $C \times H$, The feature maps of $\times W$ are multiplied. This is equivalent to multiplying each $H \times W$ feature map by a $H \times W$ spatial weight to form a spatial

attention module.

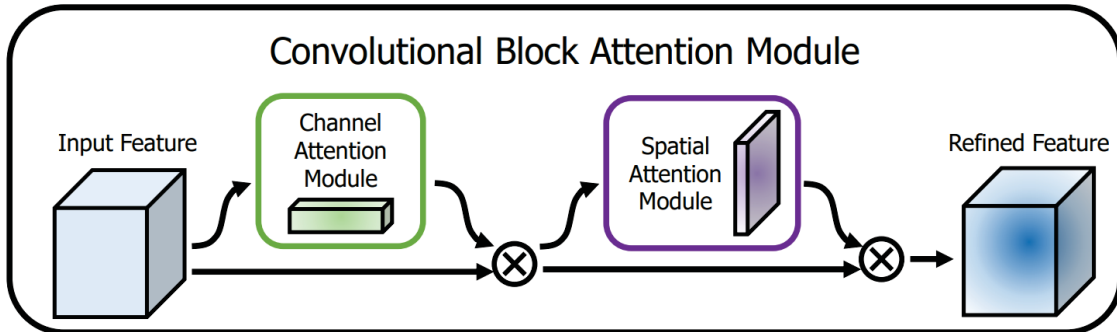


Figure 73: Two attention modules are used in the SAM technique – [18].

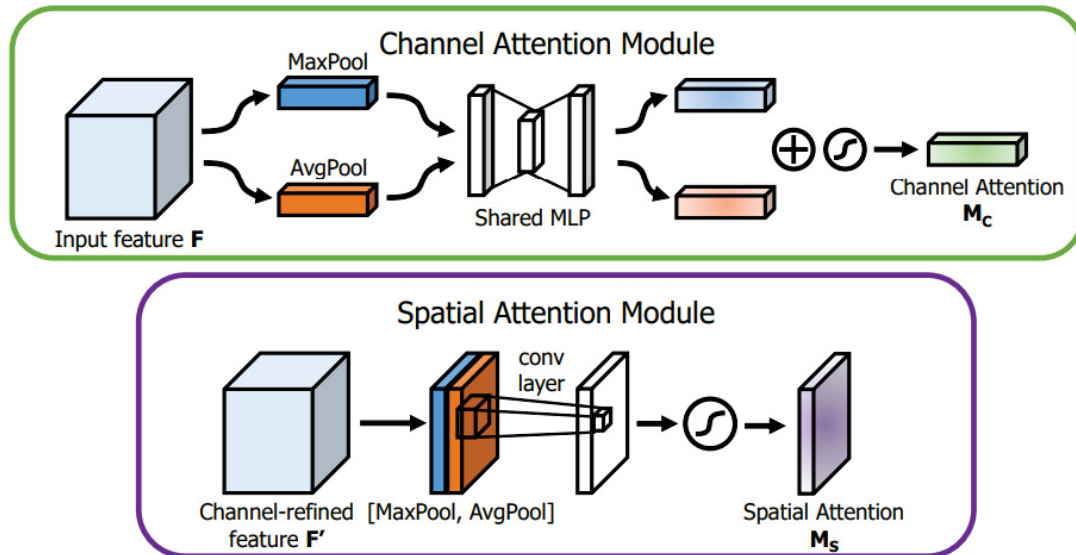


Figure 74: The input feature goes through pooling steps, the shared MLP and is then included into the channel attention module. The spatial module uses two outputs pooled together before transferring it to a convolution layer – [18].

7.1.6 Modified SAM

Modified SAM is an innovation of YOLO v4, called the pixel attention mechanism. Its idea is also very simple, that is to remove all the pooling layers of the SAM module and perform 1×1 convolution on the feature map of $C \times H \times W$ (There is neither down channel nor up channel), get the output feature map of $C \times H \times W$, then use sigmoid activation, and then multiply the pixels with the original $C \times H \times W$ – [18].

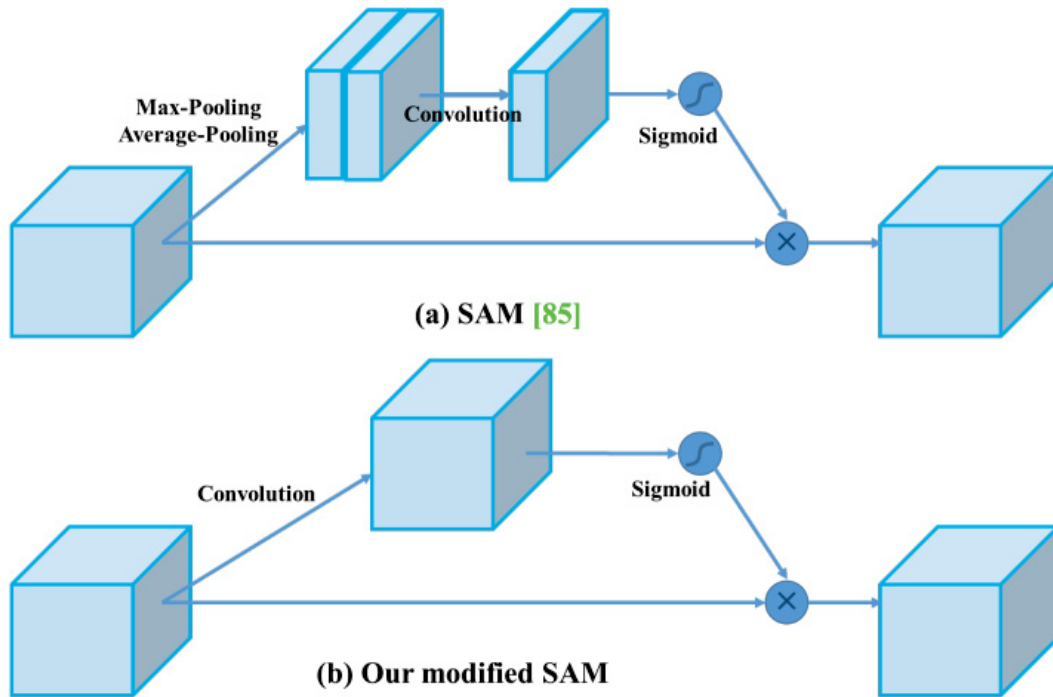


Figure 75: Structure of the modified SAM architecture – [18].

7.1.7 Skip connection

Similar to a differential amplifier, small gradient changes are amplified, resulting in larger adjustments to the weights. The following figure demonstrates the *skip connection* process.

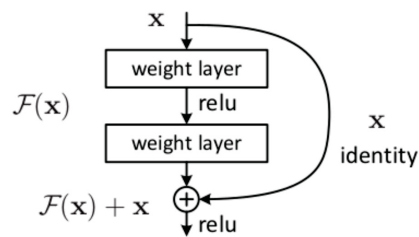


Figure 76: Skip connection.

7.1.8 Hyper column

The calculation of the super column is to obtain a detection frame (may be a square), resize it to the fixed input size of the classification network (the classification network uses a fully connected layer), resize the image into blocks and enter the classification network,

and bilinear the feature map of each layer. Interpolate, restore to the image block size, and obtain the hypercolumn (cascaded multi-channel activation value) of the corresponding position of each layer – [19].

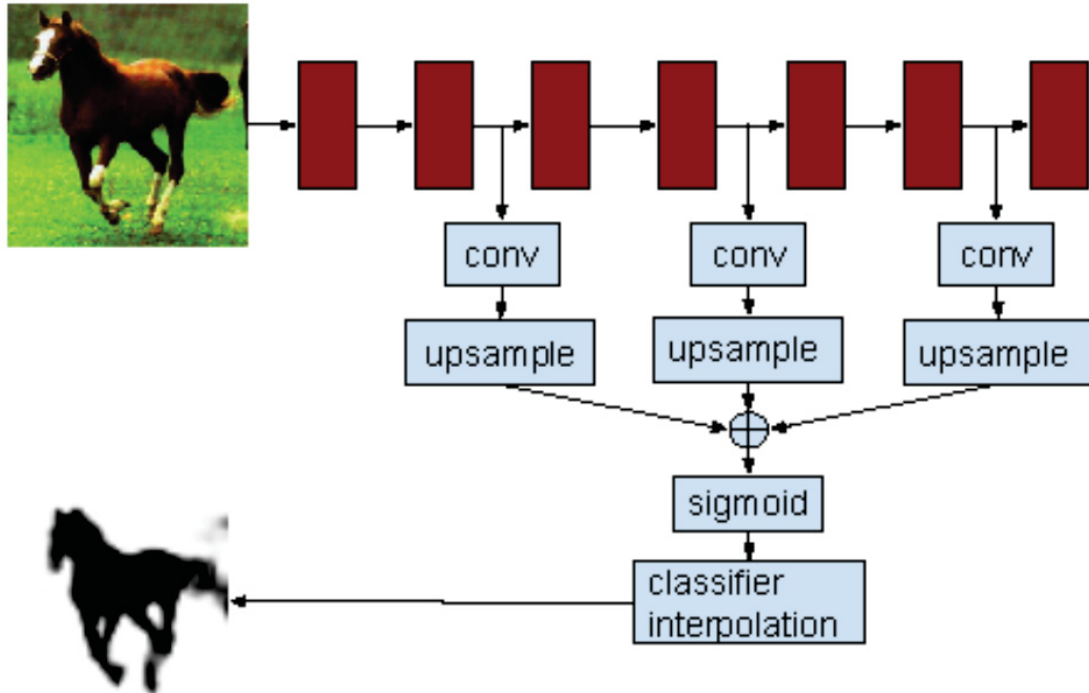


Figure 77: "Representing hypercolumn classifiers as a neural network. Layers of the original classification CNN are shown in red, and layers that we add are in blue." – [19].

7.1.9 FPN

FPN is a feature pyramid network, which combines low-resolution, strong semantic features with high-resolution, weak semantic features through top-down paths and horizontal connections to perform feature fusion – [20].

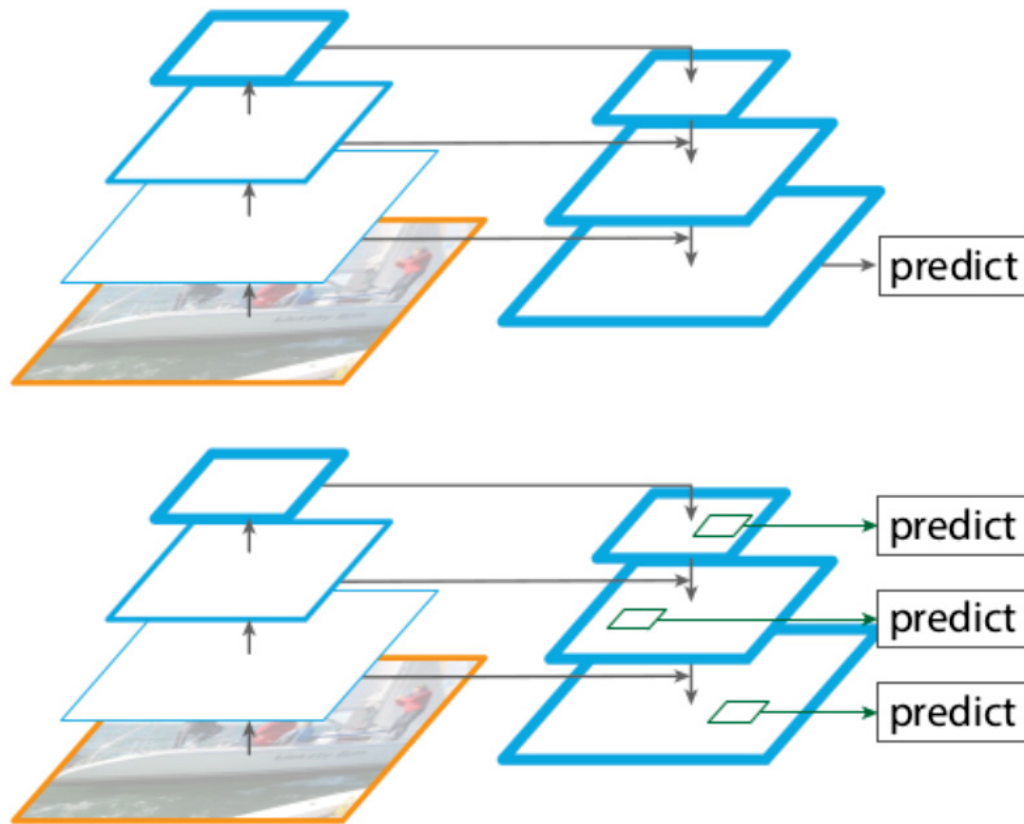


Figure 78: "The top-down architecture with skip connections, where predictions are made on the finest level. Bottom: our model that has a similar structure but leverages it as a *feature pyramid*, with predictions made independently at all levels." – [20].

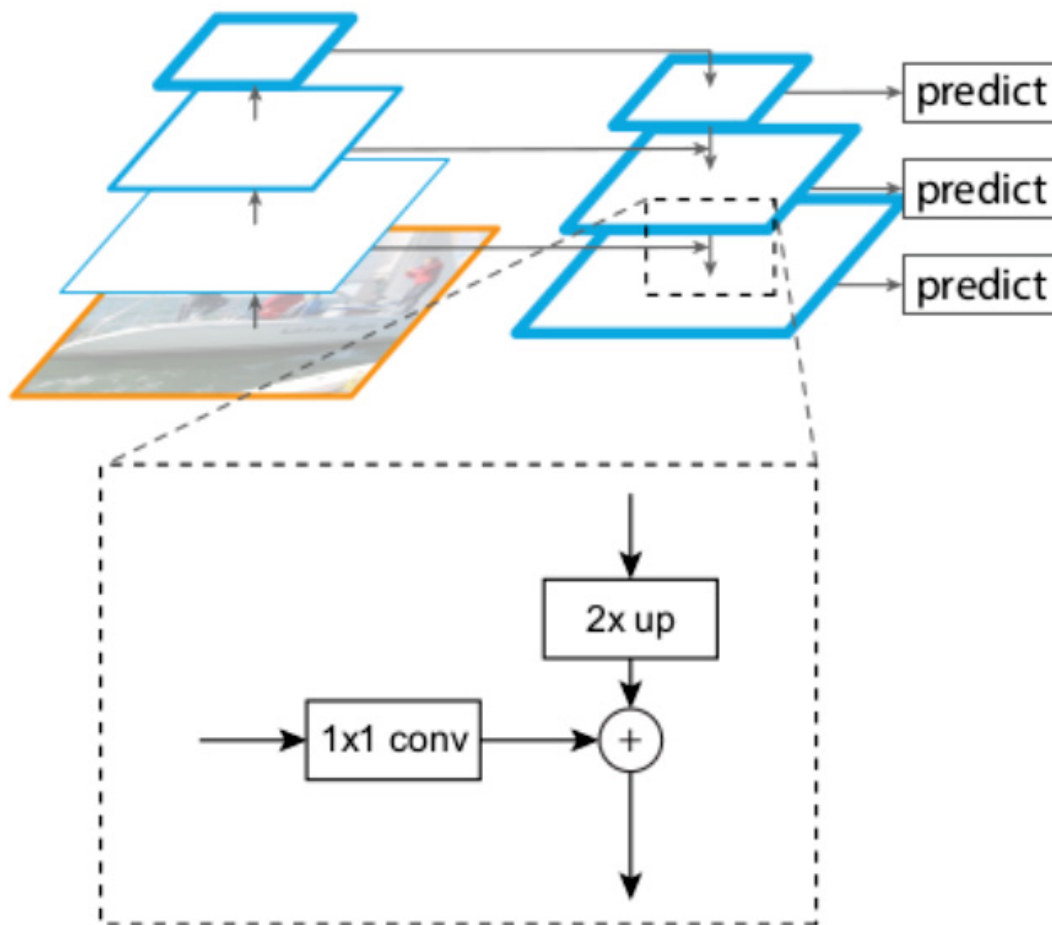


Figure 79: "A building block illustrating the lateral connection and the top-down pathway, merged by addition." – [20].

The idea is to double-upsample the high-level feature map and add it to the low-level feature map. Since the number of channels in the feature map does not match, it is necessary to use a 1×1 convolution to channel the low-level feature map. Number matching allows high-level semantic information to be transmitted to the underlying feature map so that high-level semantic information can be transmitted to the underlying feature map and fused with low-level location information to obtain correctly classified and accurate target detection objects.

- The pyramid of each layer is set with a single-scale anchor (the anchor corresponds to the original image);
- The anchor size corresponding to P2, P3, P4, P5, P6 is 32^2 , 64^2 , 128^2 , 256^2 , 512^2 , and the aspect ratio is 1:2, 1:1, 2:1.

8 Appendix C – Other methods of learning

8.0.1 Batch learning

In the batch approach of supervised learning, the adjustment of the salient weights of the multilayer perceptron is performed after all N examples of the training sample set are present, which constitutes one round of training. In other words, the cost function of batch learning is defined by the average error energy. The adjustment of the synaptic weights of the multilayer perceptron is on a round-round basis. Accordingly, one implementation of the learning curve is obtained by plotting the average error energy against the number of rounds, and for each round of training, the samples of the training sample set are randomly selected. The learning curve is computed by averaging over a sufficiently large number of such realizations overall, where each realization is done under a randomly selected different initial condition. This feature is consistent with cross-validation, where the experimental set, validation set, and test set are generally exemplary of batch processing.

Advantages

1. Eliminate the effect of sample order;
2. Accurate estimation of the gradient vector, thus, under simple conditions, guarantees the convergence of this method with the fastest descent to the local minima;
3. Parallelism of learning;

Disadvantages

Has storage requirements;

8.0.2 Online learning

Under the online method of supervised learning, the adjustment of synaptic weights for multilayer perceptrons is on a sample-sample basis, and the cost function used for minimization is the plenary instantaneous error energy. As with batch learning, the learning curve for online learning is computed by a sufficiently large number of overall averages over randomly selected initial conditions. For a given network structure, the learning curve obtained under online learning is very different from that obtained under batch learning.

Given that the training samples are presented to the network in a random manner, the use of online learning makes the search in the multidimensional weight space de facto random; it is for this reason that online learning methods are sometimes referred to as stochastic methods.

Advantages

1. Easy to implement;
2. It provides efficient solutions for large-scale and difficult pattern classification problems;
3. Randomness makes it less likely to fall into local extrema;
4. Much less storage;

8.0.3 Reinforcement learning

The so-called reinforcement learning is the learning of an intelligent system mapping from environment to behavior in order to maximize the value of the reward signal (reinforcement signal) function, reinforcement learning is different from the supervised learning in connectionist learning, mainly in the teacher signal, reinforcement learning in the reinforcement signal provided by the environment is an evaluation of the goodness of the resulting action (usually a scalar signal), rather than telling the reinforcement learning system RLS (reinforcement learning system) how to produce the correct action. Since the external environment provides little information, the RLS must learn by its own experience. In this way, the RLS acquires knowledge in an action-evaluation environment and improves the action plan to adapt to the environment.

In this learning model, the input data is used as feedback to the model, unlike supervised models where the input data is used only as a way to check whether the model is right or wrong, in reinforcement learning, the input data is fed directly to the model and the model must adjust to it immediately. If the learning agent (e.g., the quadruped robot, the chess AI program above) gets a better result after deciding on a step, then we give the agent some reward (e.g., a positive reward function result), and if it gets a worse result, then the reward function is negative. For example, for a quadruped robot, if he takes a step forward (close to the goal), then the payoff function is positive and backward is negative. If we can evaluate each step and get the corresponding payoff function, then it is good to know that we only need to find a path with the largest payoff value (the sum of the payoffs of each step is the largest) and consider it as the best path.

Common application scenarios include dynamic systems as well as robot control. Common algorithms include Q-Learning and Temporal difference learning.

8.0.4 Active learning

Active learning refers to a learning method in which sometimes data with class labels are scarce and data without class labels are abundant, but manual labeling of the data is

very expensive, so the learning algorithm can initiate some labeling requests and submit some filtered data to experts for labeling. This filtering process is the main research area of active learning, how to filter the data so that the number of requests for annotation is as small as possible and the final result is as good as possible. The process of active learning is roughly like this: there is a dataset K (initially empty) that has been labeled with a class label, and a dataset U that has not been labeled yet, and through the information in the K set, a subset C of U is identified, a request for labeling is made, and the expert adds the dataset C to the K set after it is labeled for the next iteration. According to the description on the wiki, active learning also belongs to the category of semi-supervised learning, but in fact it is not the same. As mentioned above, "active" in active learning refers to the active request for annotation, i.e., it still requires an external entity that can annotate the request (usually the relevant domain personnel), i.e., active learning is interactive.

Semi-supervised learning, on the other hand, specifically refers to learning algorithms that do not require human intervention and are based on their own use of unlabeled data.

8.0.5 Direct push learning

The difference is that direct push learning assumes that the unlabeled data are the data that will eventually be used for testing, and the goal of learning is to obtain the best generalization capability on these data. In contrast, semi-supervised learning does not know what the final test case will be at the time of learning. That is, direct push learning is actually similar to a subproblem of semi-supervised learning, or a specialization of semi-supervised learning, so some people also classify it as semi-supervised learning.

8.0.6 Incremental learning and decremental learning

Online learning includes incremental learning, and decremental learning means that a learning system can continuously learn new knowledge from new samples and preserve most of the previously learned knowledge. Incremental learning is very similar to the learning pattern of humans themselves. Because humans learn and receive new things every day as they grow, learning occurs gradually, and humans generally do not forget what they have already learned.

An incremental learning algorithm should also have the following characteristics.

1. The possibility of learning new knowledge from new data;
2. Data that has been previously processed does not need to be repeated;
3. Only one training observation sample is seen and learned at a time;
4. The ability to learn new knowledge while preserving most of the previously learned knowledge;

5. Once learning is complete the training observation sample is discarded;
6. the learning system has no prior knowledge about the entire training sample; incremental algorithm: it is not necessary to rebuild all the knowledge base whenever new data is added, but only the updates caused by the new data are done on the basis of the original knowledge base, which is more in line with the principle of human thinking;

In offline learning, the whole training data must be available at model training. Only when training is completed can the model be used. In contrast, online algorithms process data sequentially. They produce a model and put it in operation without having complete training. The model is continuously updated during operation as more training data arrives.

8.0.7 Transfer Learning

In the traditional machine learning framework, learning is to learn a classification model given sufficient training data; this learned model is then used to classify and predict the test documents. However, we see a crucial problem with machine learning algorithms in current Web mining research: large amounts of training data in some emerging domains are tough to obtain. First, we have seen the Web application domain evolve very rapidly. Many new domains are emerging, from traditional news to web pages, images, blogs, podcasts, etc. Traditional machine learning requires a large amount of training data for each domain, which is very labor-intensive. Many learning-related research and applications cannot be carried out without a large amount of labeled data. Second, traditional machine learning assumes that the training and test data obey the same data distribution. However, in many cases, this assumption of identical distribution is not satisfied. This can often happen when the training data is out of date. This often requires us to re-label a large amount of training data to meet our training needs, but labeling new data is costly and requires a lot of human and material resources.

On the other hand, if we have many training data under different distributions, it is also very wasteful to discard them altogether. How to use these data wisely is the main problem of migration learning. Transfer learning can transfer knowledge from existing data to help future learning. Transfer learning aims to use the knowledge learned from one environment to help learn tasks in a new environment. Therefore, transfer learning does not make the same distributional assumptions as traditional machine learning. As a general example, a person who knows how to play chess can learn to play Go; a person who knows a table can know a chair more easily. learn to play Go; a person who knows a table can know a chair more easily

Current work in transfer learning can be divided into three parts: instance-based transfer learning in homogeneous spaces, feature-based transfer learning in homogeneous spaces, and transfer learning in heterogeneous spaces. Instance-based transfer learning has a more vital ability to communicate knowledge, feature-based transfer learning has a broader ability

to share knowledge, and migration in heterogeneous areas has a general power to learn and extend.

Transfer learning, the influence of one type of learning on another, is widely present in teaching knowledge, skills, attitudes, and behavioral norms. Any teaching is influenced by the learner's existing knowledge and experience, skills, attitudes, etc. As long as there is learning, there is transfer. The transfer is the continuation and consolidation of education, and the condition for improving and deepening understanding and knowledge and transfer are inseparable. For the development path of AI, many people may be familiar with AI-based on big data, but there are also attempts and migrations based on small samples, which is also a path of AI.

8.0.8 YOLO v1

The YOLO (You Only Look Once) algorithm is a Convolutional Neural Network (CNN) that requires an image or video to be processed only once through the network to identify objects. The first version was published – [131] – by Joseph Redmon in 2016.

YOLO's previous object detection systems used classifiers to perform object detection tasks. To detect an object, these object detection systems use the object's classifier to evaluate the presence or absence of the object at different locations and on different sizes of bounding boxes in a test image. In the case of DPM systems, a sliding window is used to slide evenly over the entire image, and the classifier is used to assess the object's presence.

Other methods proposed after DPM, such as the R-CNN method, use a regional proposal to generate potential bounding boxes containing objects to be detected throughout the image and then use the classifier to evaluate these boxes. Boxes eliminate duplicate detection targets and re-score the boxes based on other objects in the whole scene. The whole process is slow to execute, and because these sessions are trained separately, detection performance is challenging to optimize.

The authors designed YOLO (you only look once) to treat the object detection tasks as a regression problem. The bounding box coordinates, the confidence level of the objects contained in the box, and the class probabilities are obtained directly from all pixels of the whole image. With YOLO, each image can be viewed at a glance to find out which objects are in the image and where they are located.

The following images explain the process of bounding box identification using different classes.

8.0.8.1 Detection process

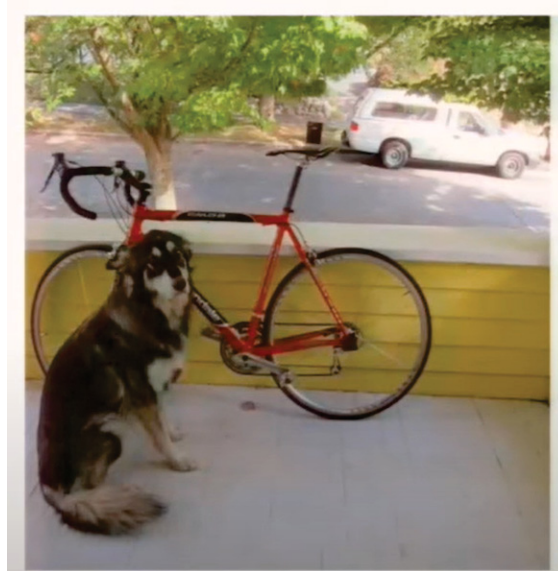


Figure 80: The original image displays three identifiable COCO classes: a dog, a bicycle and a car.

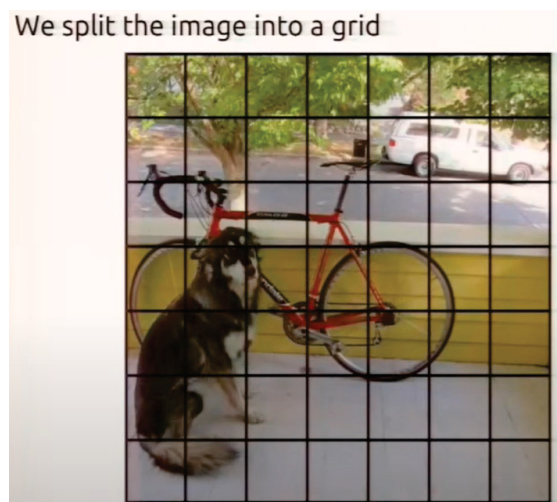


Figure 81: The image is divided into a 13x13 grid.

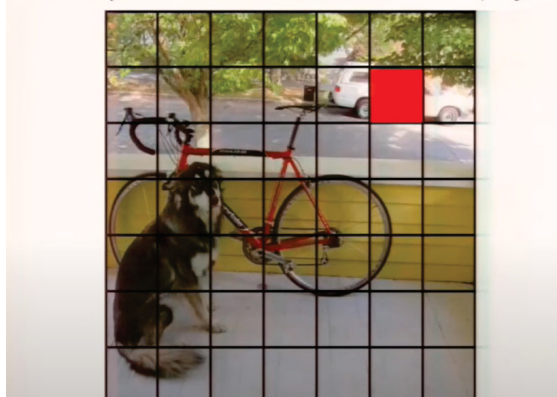
Each cell predicts boxes and confidences: $P(\text{Object})$ 

Figure 82: Each one of the divided cells in the 13x13 grid predicts different number of boxes and confidences. It is should be noted that bounding boxes can have different sizes and use multiple cells to constitute it.

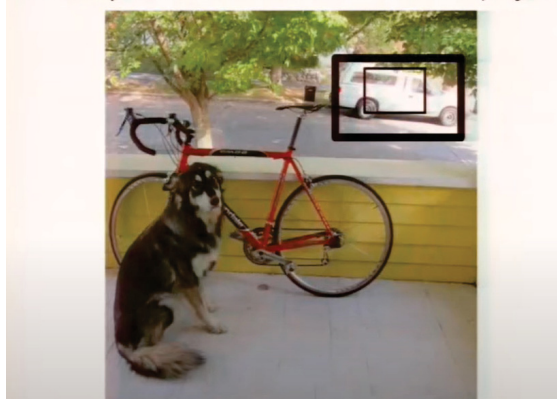
Each cell predicts boxes and confidences: $P(\text{Object})$ 

Figure 83: The algorithm predicts the box size with a level of confidence.

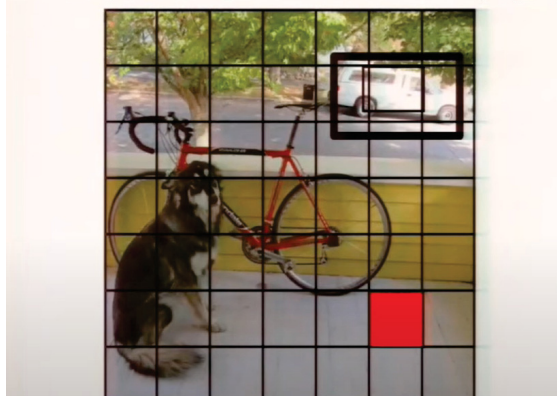
Each cell predicts boxes and confidences: $P(\text{Object})$ 

Figure 84: The algorithm selects a new bounding box region to analyse size and confidence.

Each cell predicts boxes and confidences: $P(\text{Object})$

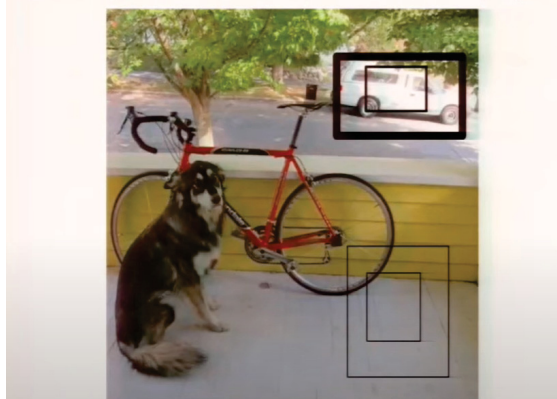


Figure 85: New bounding boxes are predicted by the algorithm

Each cell predicts boxes and confidences: $P(\text{Object})$



Figure 86: At the end of the process, many different bounding boxes are present in the picture – predicting different possible objects.

Each cell also predicts a class probability.

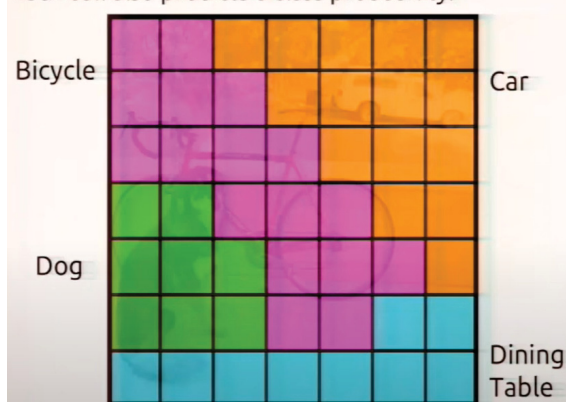


Figure 87: Simultaneously, each cell predicts different class probabilities.

Then we combine the box and class predictions.

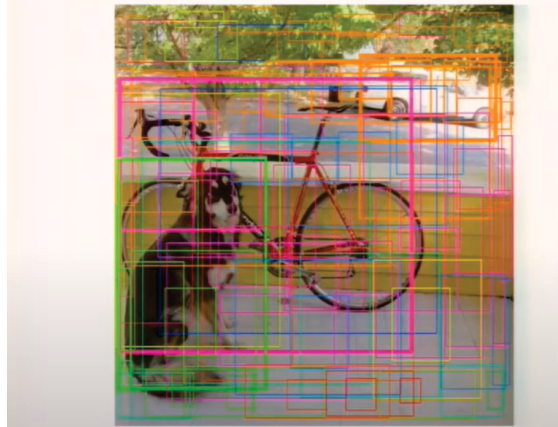


Figure 88: The combination of the box and class predictions exposes all the different predictions made by each cell.

Finally we do NMS and threshold detections

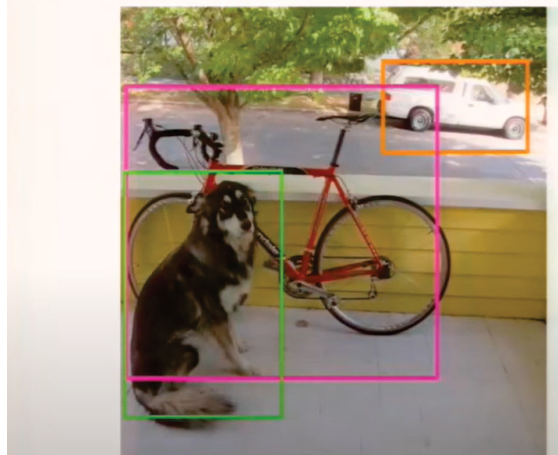


Figure 89: Using the non-maximum suppression technique (NMS) and a minimum confidence threshold, the process eliminates bounding boxes that have a low confidence level and are not compliant with the background context. The background context is partially defined by the level of confidence of each cell.

Figure 90 represents the architecture of the YOLO algorithm. The bounding box has 4 coordinates relative to the input image with a defined maximum dimension. Two coordinates that establish the center of the box and two values for the height and weight of the box. The last value defines the confidence value related to the class for the identified object.

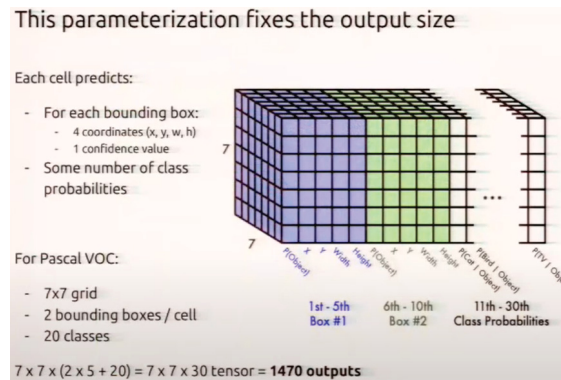


Figure 90: Representation of the YOLO darknet architecture.

The end result is an extremely fast detection pipeline which can combine the results obtained from the selection of different bounding boxes and the class confidence values for each cell.

8.0.8.2 Training steps

The training step first starts by analysing the labelled bounding box around the object. A specific class is then attributed to each of the 13x13 cells, in figure 92.

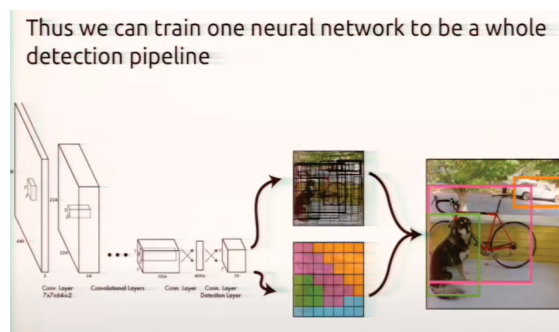


Figure 91: Representation of the bounding box with class.

During training, match example to the right cell



Figure 92: Representation of the attribution of the dog class to the specific cell using a centered x,y position. The box central position is going to define the class attributed to each cell on the grid.

During training, match example to the right cell

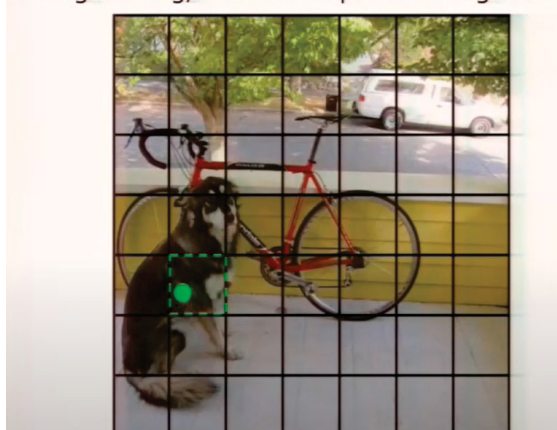


Figure 93: The entire cell is defined as belonging to a specific class after the center of the bounding box falls to that area.

Adjust that cell's class prediction Find the best one, adjust it, increase the confidence



Figure 94: Out of all the bounding boxes falling with the (x,y) coordinates to that specific cell, the best one is chosen and then adjusted – increasing the total confidence of the bounding box.



Figure 95: Some of the bounding boxes is a specific location doesn't have an object in that area, so the level of confidence in that region is reduced.

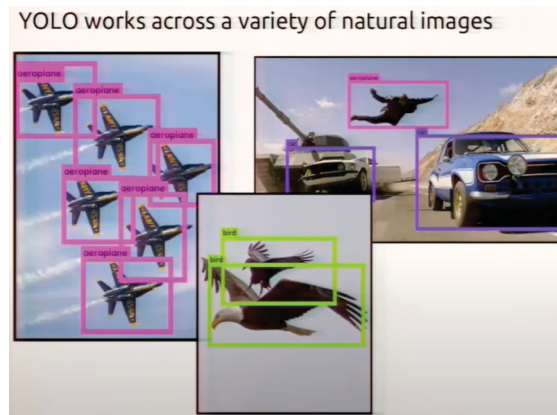


Figure 96: Trained on the COCO dataset, the YOLO algorithm can predict a great variety of different objects – including different simultaneous predictions.

The YOLO loss function for the algorithm described by [91] is represented by:

$$\begin{aligned}
 Loss = & \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
 & + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\
 & + \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\
 & + \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
 \end{aligned} \tag{23}$$

The main loss is described in parts:

1. The loss of the predicted center coordinates is described by:

$$Loss = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \ell_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \quad (24)$$

This equation calculates the value of the loss relative to the predicted bounding box location (x,y). λ_{loss} is a given constant. This function calculates the loss for each grid cell ($i = 0, \dots, S^2$) for each of the bounding box predictions ($j = 0, \dots, B$) summed. ℓ_{ij}^{obj} is defined as follows:

- 1, if the target exists in grid cell i , then the j_{th} bounding box prediction is valid for that prediction;
- 0, if there is no target in grid cell i ;

The other terms in the equation are the coordinates: (x,y) is the location of the predicted bounding box, and (x , $\hat{y}$) is the actual location obtained from the training data.

2. The loss of width and height of the predicted bounding box is described by:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \quad (25)$$

This is the loss associated with the predicted width/height (w and h associated variables) of the bounding box. The equation looks similar to the first one except for the square root. The error metric responds that the small deviation of the large box is smaller than that of the small box. To solve this problem step by step, we predict the square root of the width and height of the bounding box instead of predicting the width and height directly.

3. The loss on the predicted category is described by:

$$\sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{noobj} (C_i - \hat{C}_i)^2 \quad (26)$$

Except for the fact that ℓ_{ij}^{obj} term, the equation looks similar to the normal summation squared error for classification. This term is used because when there is no object (the conditional class probability discussed earlier), we do not penalize the classification error.

4. The loss of the confidence of the prediction:

$$\sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \quad (27)$$

Here we calculate the loss associated with the confidence score for each bounding box prediction. c is the confidence score, $\hat{C}$ is the confidence score for the predicted bounding box with the intersection of the underlying facts. When there are objects in a cell, ℓ_{ij}^{obj} is equal to 1, otherwise it takes the value of 0. The parameters appearing here and in the first part are used in different weighted parts of the loss function. This is very key to improve the stability of the model. The highest penalty is for the coordinate prediction ($coord = 5$) ($coord = 5$) ($coord = 5$), and there is a minimum confidence prediction penalty when no target is detected ($noobj = 0.5$) ($noobj = 0.5$) ($noobj = 0.5$).

Here, the author states that ℓ_{ij}^{obj} denotes if object appears in cell i , which is understood to be 1 whenever it contains a bounding box. The output calculate objects in a tensor format that illustrates this.

After the process of training is completed on the labeled target dataset, the algorithm can identify as many classes as trained. However, notable constraints are the inability to generalize objects in different aspect ratios and different configurations. Small objects in the picture are difficult to predict since the algorithm treats errors in large and small bounding boxes in the same manner. This version of the algorithm has greater performance when detecting small objects.

8.0.9 YOLO v2

The YOLO v2 algorithm was published as YOLO9000 – [129] and was optimized to predict over 9000 object categories in the Microsoft COCO dataset – [99].

Enhancements made in the architecture of the second version include using a hierarchical structure of object classification which allows the developer to merge two different kinds of datasets. This algorithm also allows one to train using labeled detection images alongside classification images to make the detection more accurate.

8.0.9.1 Batch normalization

In the network structure of the detection series, BN has gradually become a standard configuration. After adding BN to each convolutional layer of Yolo, mAP is increased by 2%, and Dropout is removed.

8.0.9.2 High resolution classifier

In YOLOv1, the backbone part of the network will be pre-trained on the ImageNet dataset, and the resolution of the network input image during training is 224×224 . In v2, the classification network is trained for 10 epochs on the ImageNet dataset with the input image

resolution of 448*448, and then the detection dataset (such as COCO) is used for fine-tuning. High-resolution pre-training improves mAP by approximately 4%.

8.0.9.3 Convolutional with anchor boxes

In the first Interpretation of v1, it is mentioned that each grid point predicts two rectangular boxes. When calculating the loss, only the box closest to the ground truth will generate the loss value, and the other box will not be corrected. After this stipulation, the author found that the two boxes gradually divided labor in the size, aspect ratio, and category of the object. In v2, the neural network does not predict the absolute value of the width and height of the predicted rectangular box, but predicts the deviation (offset) from the Anchor box, and each grid point specifies n Anchor boxes. During training, the box closest to the ground truth produces loss, and the other boxes do not produce loss. After the introduction of the Anchor Box operation, the mAP dropped from 69.5 to 69.2. The reason is that after the number of objects predicted by each grid point increases, the recall rate increases sharply, the accuracy rate decreases, and the overall mAP decreases slightly.

In v2, the last two fully connected layers of v1 are removed. The fully connected layer is computationally intensive and takes a long time. The article does not describe the replacement of the fully connected layer in detail. Here, I guess it is replaced by a 1*1 convolutional layer (corrections are welcome). The specific network structure is not mentioned in the original text, and the official code is also replaced by yolo v3. V2 is mainly the effect verification after the introduction of various tricks. It is recommended that you do not have to worry about the network structure of v2.

8.0.9.4 Dimension clusters

Here is an innovation of the author. The width and height of the nine Anchor Boxes in Faster R-CNN are preset ratios. A total of three rectangular boxes with area sizes are set. Each rectangular box has three aspect ratios: 1:1, 2:1, 1:2, a total of nine boxes. In v2, the width and height of the Anchor Box are not obtained artificially. Instead, all the rectangular boxes in the training data set are taken out, and k-means clustering is used to obtain the width and height of the prior box. For example, if 5 Anchor Boxes are used, then the number of category centers of k-means clustering is set to 5. After adding the clustering operation and introducing the Anchor Box, mAP increased.

It should be emphasized that clustering must define [formula] the distance function between clustering points (rectangular boxes). The following functions are used in the article:

$$d(box, centroid) = 1 - IOU(box, centroid) \quad (28)$$

Using the value of (1-IOU) as a function of the distance between two rectangular boxes, the application here is also very clever.

8.0.9.5 Direct location prediction

The location prediction method in YOLO is very clear, which is to predict the offset relative to the grid point coordinates in the upper left corner. The specific meaning of Direct here should be obtained after comparing with other algorithm frameworks. For example, other popular position prediction formulas are as follows:

$$x = (t_x * w_a) + x_a \quad (29)$$

$$y = (t_y * h_a) + y_a \quad (30)$$

Neural network prediction is donated as t_x, t_y , but these parameters need *a priori* value by multiplying the frame width which can be obtained as compared to the high x_a, y_a position of the offset value, in YOLO v2, the position prediction equation is as follows:

$$b_x = \sigma(t_x) + c_x \quad (31)$$

$$b_y = \sigma(t_y) + c_y \quad (32)$$

$$b_w = p_w e^{t_w} \quad (33)$$

$$b_h = p_h e^{t_h} \quad (34)$$

The natural advantage of this implementation is that the coordinates of the grid points can be used as reference points. At this time, t_x, t_y an activation function is used to directly generate the offset position value, which is independent of the width and height of the rectangular box, and becomes more direct.

8.0.9.5.1 Fine-grained features

After the 26*26 feature map is transformed into a 13*13 feature map through convolutional layers, it can be verified that a lot of fine-grained features are lost, resulting in poor recognition of small-sized objects, so pass through layer is added in this step:

As shown in the figure above, the passthrough layer is to turn the feature map of 26*26*1 into a feature map of 13*13*4, without losing fine-grained features in this operation. Some people say that the specific details of the passthrough layer divided into four are not cut into 4 pieces with two knives, but the upper left corner piece is selected on each 2*2 small area. In fact, it is the same no matter how to cut it here. In the subsequent 1*1 convolutional layer, it is just a matter of the commutative law of addition.

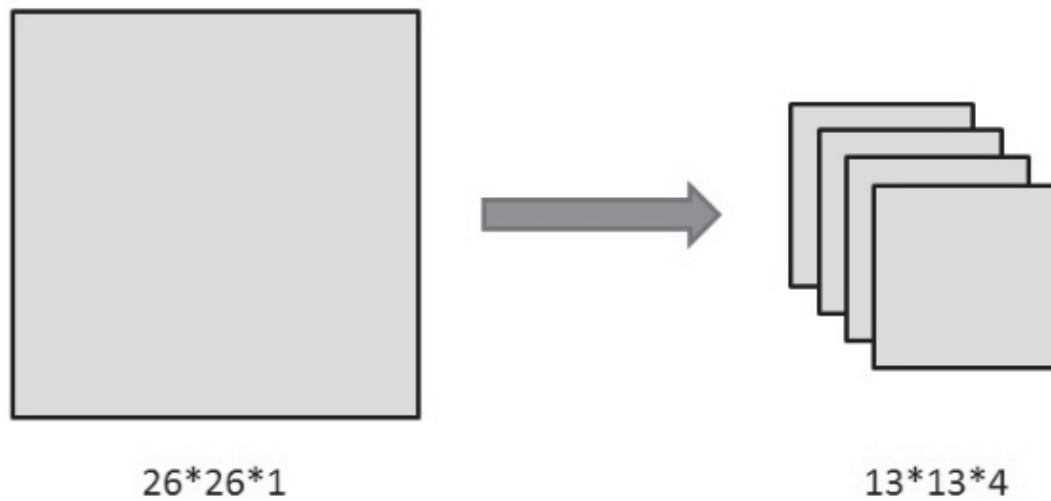


Figure 97: Representation of a pass through layer.

8.0.9.5.2 Multi-scale training

The key point is that there are only convolutional and pooling layers in Yolo v2, so there is no limit to the input size of the network. The downsampling multiple of the entire network is 32, as long as the input feature map size is a multiple of 32. If there is a fully connected layer in the network, this is not the case. So Yolo v2 can use input images of different sizes for training.

The training method used by the author is to resize the image into one of 320, 352, ..., 608 after every ten batches. Different input, the final number of grid points is different; for example, the input picture is 320×320 , then the output grid point is 10×10 , if the number of a priori boxes for each grid point is set to 5, then a total of 500 prediction results will be output; If the input picture size is 608×608 , the output grid point is 19×19 , and a total of 1805 prediction results.

After introducing the multi-size training method, the convolution kernel is forced to learn features of different scale sizes. When the input is set to 544×544 or even larger, the mAP of Yolo v2 has surpassed other object detection algorithms.

9 Appendix D – Characteristics of a CNN

9.1 Part I

9.1.1 Main characteristics of a CNN

9.1.1.1 Connection between layers

The connections between the convolutional layers in a convolutional neural network are called sparse connections, i.e., in contrast to the full connections in a feedforward neural network, the neurons in a convolutional layer are connected to only some, but not all, of the neurons in its neighboring layers. Specifically, any pixel (neuron) in the feature map of layer l of a convolutional neural network is only a linear combination of pixels within the receptive field defined by the convolutional kernel in layer $l - 1$. The sparse connections in convolutional neural networks have a regularizing effect, which improves the stability and generalization of the network structure and avoids overfitting, while the sparse connections reduce the total number of weight parameters, which facilitates fast learning of the neural network and reduces memory overhead during computation. All pixels in the same channel of the feature map in a convolutional neural network share a set of convolutional kernel weight coefficients, a property known as weight sharing. Weight sharing distinguishes convolutional neural networks from other neural networks that contain locally connected structures, which use sparse connections but have different weights for different connections. Weight sharing, like sparse connections, reduces the total number of parameters in a convolutional neural network and has a regularizing effect.

In the fully connected network perspective, the sparse connectivity and weight sharing of a convolutional neural network can be considered as two infinitely strong priors (priors), i.e., a hidden layer neuron has a constant weight coefficient of 0 outside its receptive field (but the receptive field can move in space); and all neurons have the same weight coefficient within a channel.

9.1.1.2 Representation learning

As a representative algorithm of deep learning, convolutional neural networks have representational learning capabilities, i.e., they are able to extract higher-order features from the input information. Specifically, the convolutional and pooling layers in a convolutional neural network are able to respond to the translational invariance of the input features, i.e., they are able to recognize similar features located at different locations in space. The ability to extract translation-invariant features is one of the reasons why convolutional neural networks are used in computer vision problems. The transfer of translation-invariant features within a convolutional neural network has a general pattern. In image processing problems,

the feature map at the front of the convolutional neural network usually extracts representative high and low frequency features of the image; the feature map, which is subsequently pooled, shows the edge features (aliasing artifacts) of the input image; and when the signal enters a deeper hidden layer, its more general and complete features are extracted. Deconvolution and anti-pooling (un-pooling) allow visualization of the hidden layer features of a convolutional neural network. In a successful convolutional neural network, the feature map passed to the fully connected layer will contain the same features as the learning target, such as the complete images of each category in image classification.

9.1.1.3 Biological similarity

The sparse connections in convolutional neural networks based on the receptive field setting have a clear counterpart to a neuroscientific process - the organization of visual space by the visual cortex in the visual nervous system. Visual cortex cells receive signals from photoreceptors in the retina, but individual visual cortex cells do not receive all signals from photoreceptors, but only from the area of stimulation they innervate, i.e., within the receptive field. Only stimuli within the receptive field can activate the neuron. Multiple visual cortical cells receive signals from the retina and establish visual space by systematically superimposing the receptive fields. In fact, the term "receptive field" in machine learning is derived from its biological counterpart. There is no clear evidence for the nature of weight sharing in convolutional neural networks in biology, but in studies of target-propagation (TP) and feedback alignment (FA) mechanisms, which are closely related to brain learning, weight sharing has been shown to enhance learning.

9.2 Part II

9.2.1 Activation functions

9.2.1.1 What is the activation function?

Activation functions are very important for artificial neural network models to learn and understand very complex and nonlinear functions. They introduce nonlinear properties into our networks. As in fig. 98, in the neuron, the input inputs are weighted, summed, and then a function is also acted upon, which is the activation function. The activation function is introduced to increase the non-linearity of the neural network model. Each layer without the activation function is equivalent to matrix multiplication. Even after you stack several layers, it is still a matrix multiplication.

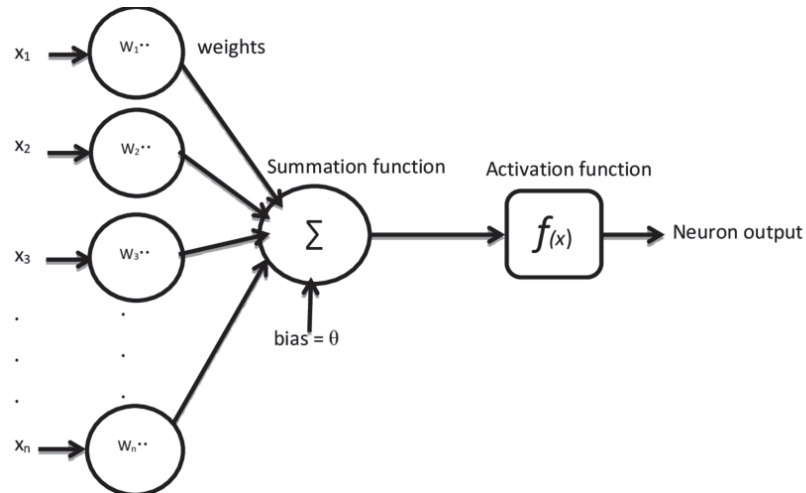


Figure 98: Structure of an artificial neuron.

9.2.1.2 Why use activation function?

If the activation function is not used, the output of each layer is a linear function of the input of the upper layer, and the output is a linear combination of the inputs no matter how many layers there are in the neural network; this situation is the most primitive perceptron (perceptron).

If used, the activation function introduces a nonlinear factor to the neurons, allowing the neural network to approximate any nonlinear function at will, so that the neural network can be applied to a multitude of nonlinear models.

9.2.1.3 Commonly used activation functions

The commonly used activation functions can be observed in the image below:

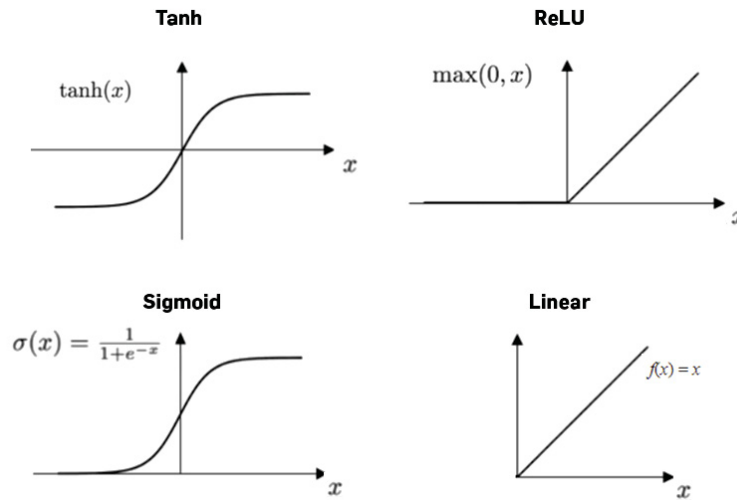


Figure 99: Types of activation functions.

Non-linear activation functions are often used for fuzzy logic applications – [112]. Applications with deep learning applications use the ReLU function for most applications.

9.2.1.3.1 Linear

It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input.

$$f(x) = cx \quad (35)$$

9.2.1.3.2 Sigmoid

The Sigmoid function is an S-shaped function commonly found in biology, also known as an S-shaped growth curve. In information science, the Sigmoid function is often used as a threshold function for neural networks, mapping variables to between 0,1 due to its single increasing as well as inverse function single increasing properties. The formula is as follows:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (36)$$

9.2.1.3.3 Tanh

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (37)$$

9.2.1.3.4 ReLU

The ReLU activation function (rectified linear unit) used as a hidden layer neuron output is as follows:

$$f(x) = \max(0, x) \quad (38)$$

The main reason why ReLU, although it is one of the best activation functions, was not widely used recently. The reason was that it was not differentiable at time 0. Researchers tended to use differentiable functions such as Sigmoid and Tanh. However, ReLU was found to be the best activation function for deep learning.

The ReLU activation function is differentiable at all points except 0. For values greater than 0, we consider only the maximum of the function.

For the backpropagation neural network calculation, the differentiation for the ReLU is relatively simple. The only assumption we will make is the derivative at point 0, which is also considered to be 0. This is usually not that big of a problem and works well for the most part. The derivative of the function is the value of the slope. The slope for negative values is 0.0 and the slope for positive values is 1.0.

The main advantages of the ReLU activation function are:

1. convolutional layers and deep learning: they are the most popular activation functions for training convolutional layers and deep learning models.
2. computational simplicity: the implementation of the rectifier function is trivial and requires only a `max()` function.
3. representative sparsity: an important advantage of the rectifier function is that it can output a true zero value.
4. linear behavior: A neural network is easier to optimize if its behavior is linear or nearly linear.

However, the main problem with rectified linearity is that all negative values immediately become zero, which reduces the model's ability to properly fit the data or train from it.

This means that any negative input to the ReLU activation function immediately converts the value to zero in the graph, which in turn affects the resulting graph by not assigning the negative values appropriately. However, this can be easily remedied by using the different variants of the ReLU activation function such as the Leaky ReLU and other functions discussed previously.

9.2.1.4 Loss function

The loss function or cost function is a function that maps a random event or its associated random variable to a non-negative real number to represent the "risk" or "loss" of the random event. . In applications, the loss function is often used as a learning criterion in connection with optimization problems, i.e., solving and evaluating models by minimizing the loss function. It is used, for example, in statistics and machine learning for parametric estimation of models, in macroeconomics for risk management and decision making, and in control theory for optimal control theory.

9.2.1.4.5 Regression problem

The loss functions corresponding to the regression problem are the L_2 loss function and the L_1 loss function, both of which measure the difference between the model estimates and the observed values.

$$L_2 \text{ loss} : L(y, y') = w(\theta)(y' - y) \quad (39)$$

$$L_1 \text{ loss} : L(y, y') = w(\theta)|y' - y| \quad (40)$$

where $w(\theta)$ is the weight of the true value, y' is the true value, and y is the output of the model. Various regression models, such as linear regression, Generalized Linear Model (GLM) and Artificial Neural Network (ANN) estimate their parameters by minimizing the L_2 or L_1 loss. The L_2 loss differs from the L_1 loss in that the L_2 loss is calculated by squaring amplifies the distance between the estimated and true values, and therefore gives a large penalty to the output that deviates from the observed values. In addition, L_2 loss is a smoothing function, which facilitates the calculation of error gradients when solving its optimization problem; L_1 loss takes the absolute value of the difference between the estimated and true values and is insensitive to the output that deviates from the true value, thus helping to keep the model stable in the presence of outliers in the observations.

9.2.1.4.6 Classification problem

The loss function corresponding to the classification problem is the 0-1 loss, which is a measure of classification accuracy, taking 0 for a correct estimate of the classification and 1 for the opposite.

$$L(y', y) = \begin{cases} 0, & y' = y \\ 1, & y' \neq y \end{cases}$$

The 0-1 loss function is a discontinuous segment function, which is not conducive to solving its minimization problem, so its surrogate loss can be constructed in the application.

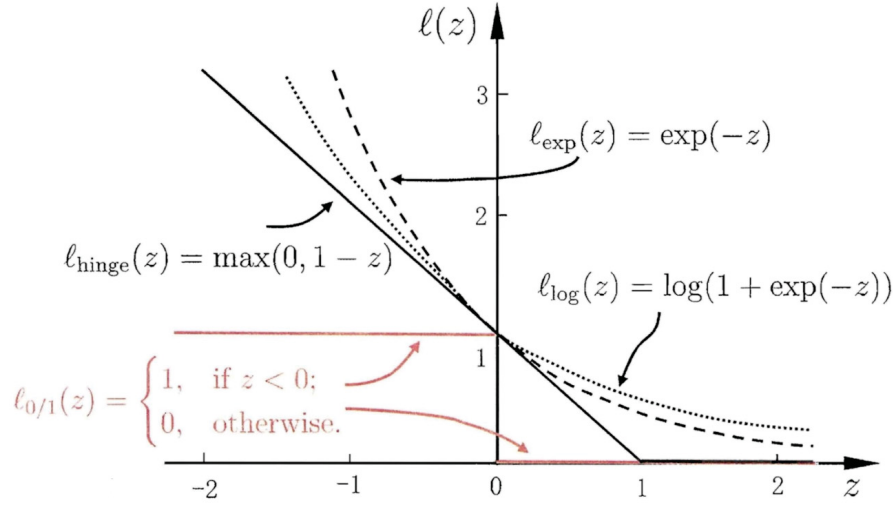


Figure 100: Representation of the loss functions.

A surrogate loss is a loss function that has consistency with the original loss function, and the model parameters obtained by minimizing the surrogate loss are also the solutions of minimizing the original loss function. When a function is continuously convex and is an upper bound of the 0-1 loss function at any value, it can be used as a proxy function for the 0-1 loss function.

The hinge loss function is defined as:

$$L(y', y) = \max(0, 1 - y'y) \quad (41)$$

The cross entropy loss function is:

- $L(y', y) = -y \log(y') - (1 - y) \log(1 - y')$

The exponential loss function is defined by:

$$L(y', y) = \exp(-y'y) \quad (42)$$

The functions are represented graphically in figure 100.

The hinge loss function is a segmented continuous function that takes 0 when the classifier classifies perfectly correctly. The classifier corresponding to the use of hinge loss is a Support Vector Machine (SVM). The nature of the hinge loss determines that the SVM is sparse, i.e., samples that are correctly classified but with probability less than 1 and incorrectly classified are identified as support vectors (support vector) are used to delineate the decision boundary, and the remaining samples that are perfectly classified are not involved in model solving.

The cross entropy loss function is a smoothing function, which is essentially an appli-

cation of the cross entropy in information theory to the classification problem. From the definition of cross entropy, it is clear that minimizing the cross entropy is equivalent to minimizing the relative entropy of the observations and estimates (relative entropy), i.e., the Kullback-Leibler scatter of the probability distributions of both; so it is a proxy loss that provides unbiased estimates. The cross-entropy loss function is the most widely used proxy loss in the table, corresponding to examples of classifiers including logistic regression, artificial neural networks, and support vector machines with probabilistic outputs. The exponential loss function is the loss function in the table that imposes the largest penalty for misclassification, and thus has the advantage of a large error gradient, corresponding to a very small value problem that is fast to solve when using the gradient algorithm. The classifier using exponential loss is usually Adaptive Boosting (AdaBoost). AdaBoost takes advantage of the ease of computation of exponential loss to build multiple "weak" classifier members that can be solved quickly, and assigns weights and iterations according to the performance of the members to combine to obtain A "strong" classifier is combined and the result is output.

9.2.1.5 Gradient descent

The objective of the training step is to reduce the loss function by changing the weight matrix. This modification is computed in an iterative manner – and this process is known as gradient descent. At each iteration a small change is made to the weight matrix. The change (ΔY) is added to the current values of the weight matrix W . The resultant gradient can then be defined as:

$$Y^{t+1} = Y^t + \Delta Y^t \quad (43)$$

where t represents the number of steps. The rate of change is defined by the learning rate $-\eta$ multiplied by the gradient of the loss function $\nabla L(Y)$ which can be summarized by:

$$\Delta Y^t = -\eta \nabla L(Y^t) \quad (44)$$

The global change to the loss function will then be expressed by:

$$\Delta L \approx \Delta Y^T \nabla L(Y) \quad (45)$$

It should be noted that the function L , which denotes the loss function varies depending on the user defined target. When the loss function becomes zero, the error of the network reaches a saturated minimum.

It is not possible to predict the saturated minimum while the training step is being calculated – it stabilizes in a target value after a determined number of iterations.

9.2.1.6 Backpropagation error

The back propagation algorithm, or BP for short, is a learning algorithm suitable for multilayer neuron networks, which is based on the gradient descent method. The input-output relationship of BP networks is essentially a mapping relationship: an n-input m-output BP neural network accomplishes the function of continuous mapping from an n-dimensional Euclidean space to a finite domain in an m-dimensional Euclidean space, which is highly nonlinear. Its information processing ability comes from the multiple compounding of simple nonlinear functions, so it has a strong function recurrence ability. This is the basis for the application of BP algorithm.

9.2.1.6.7 Neuron model and its learning process

The learning process of BP network is an error-corrected learning algorithm, which consists of forward propagation and backward propagation. In the forward propagation process, the input signal is propagated from the input layer through the action function. The state of neurons in each layer only affects the state of neurons in the next layer. If the desired output is not obtained at the output layer, it is transferred to the backward propagation, which returns the link signal along the original connection path. By modifying the connection weights of neurons in each layer. The connection structure and mapping process of the BP network is the same as that of the multilayer Perceptron, except that the unit excitation function of the latter step is replaced with a Sigmoid function. It is understood that the sample learning algorithm with modified weights (generalized delta rule) by the gradient method can be effectively applied on multilayer networks, enabling the solution of learning problems such as XOR that were previously impotent in the Perceptron model. Afterwards, three-layer networks that handle pattern recognition problems on convex regions were developed.

9.2.1.6.8 Fault tolerance

In addition to the strong characteristic of distributed memory for information, BP networks also have certain fault tolerance and interference resistance. Sun Debao and Gao Chao studied the fault tolerance and interference resistance of the three-layer BP homotopy and concluded that the fault tolerance of the three-layer BP network depends on the result of the product of the connection weight matrix from the input layer to the implicit layer and the connection weight matrix from the implicit layer to the jurisdictional output layer.

BP networks can better achieve signal identification and signal-to-noise separation in the case of wide bandwidth, small noise ratio, and few signal patterns.

9.2.1.7 Learning rate

9.2.1.7.9 Definition of the learning rate

The learning rate is one of the important hyperparameters for training a neural network, which represents the number of steps the gradient moves toward the optimal solution of the loss function in each iteration, usually denoted by η . Its magnitude determines how fast the network learns. During the network training process, the model gives predicted values from sample data, calculates the cost function and adjusts the parameters by back propagation. This process is repeated so that the model parameters gradually converge to the optimal solution to obtain the optimal model. In this process, the learning rate is responsible for controlling the step size of the parameter update at each step. The appropriate learning rate allows the cost function to converge to the minimum value at a suitable rate.

According to the above formula we can see that if the learning rate is large, then the parameters will be updated quickly and the convergence of the network can be accelerated, but if the learning rate is too large, it may cause the parameters to oscillate around the optimal solution, and the cost function will be difficult to converge, and it may even miss the optimal solution, resulting in the parameters being updated in the wrong direction, and the cost function may not only not converge but explode (as shown in fig. 101 – left representation).

$$\Delta Y^t = -\eta \nabla L(Y^t) \quad (46)$$

If the learning rate is small, the network may not miss the optimal point, but the network learning speed will become slower. At the same time, if the learning rate is too small, it is likely to fall into local optimal points (as shown in fig. 101).

Therefore, only by finding a suitable learning rate can we ensure that the cost function approximates the global optimal solution at a faster rate.

9.2.1.8 Momentum

A way to increase the learning rate without creating an oscillating effect using the backpropagation method is to use the momentum strategy. The momentum defines the past changes to the calculated weights. Using this method, the momentum considers the states of past changes in the direction of the "movement" the weights are heading to. The equation for the global weight change will then be:

$$\Delta Y^t = (1 - \alpha) \eta \nabla L(Y^t) + \alpha \Delta Y^{t-1} \quad (47)$$

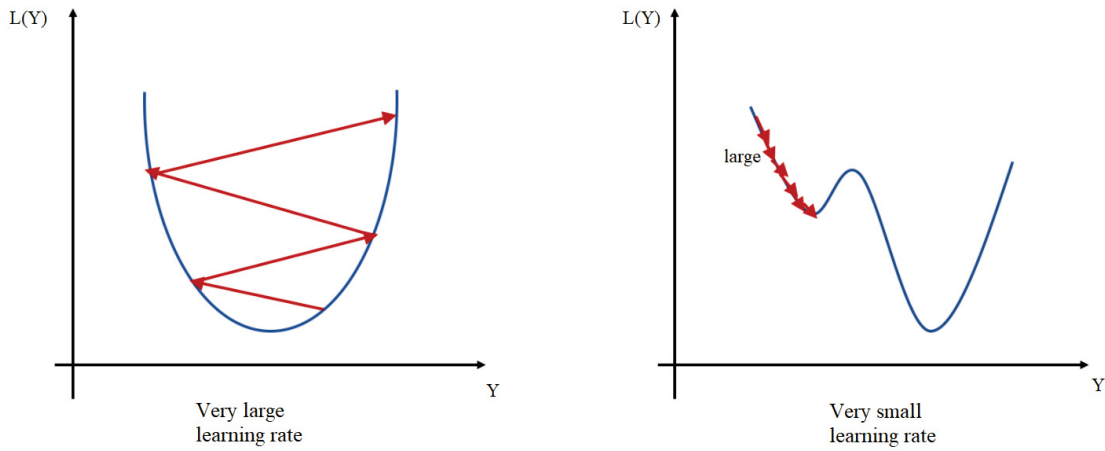


Figure 101: Different scales of learning rate.

The factor *alpha*, which denotes momentum can vary between 0 and 1. If the momentum factor is 0, the weight calculation becomes dependent on the learning rate and the previous weight adjustments.

CNNs used for computer vision have a very high momentum, which helps stabilize the convergence of the CNN.

9.2.1.9 Weight decay

The weight decay is an additional factor that can be added to the 47 to further adjust the weight adjustment. The equation then becomes:

$$\Delta \tilde{Y} = \Delta Y + \frac{\lambda}{2} W^2 \quad (48)$$

the parameter λ denotes the decay parameter. With the new loss function, the weight adjust matrix becomes:

$$\Delta Y^t = (1 - \alpha)\eta \nabla L(Y^t) + \alpha \Delta Y^{t-1} - \eta \lambda Y^t \quad (49)$$

The factor $\eta \lambda Y^t$ will counteract the momentum factor to adjust the weight matrix. It is important to note that weights with higher values will have a greater effect on the change – compelling the network to find solutions with smaller weights. This strategy requires the use of more neurons with lower weights and not few neurons with larger weights.

The characteristic of increasing the number of neurons acts as a way to solve more complex problems that require the need to generalize more varied datasets.

9.2.1.10 Other methods of learning

Other methods of learning are presented in appendix C.

9.2.1.11 Pooling Layers and Flattening

The second layer in a CNN is the pooling layer. The pooling layers are a form of down-sampling. This technique reduces the spatial size of the resulting convolutional matrices, thus reducing the parameters to be learned in the network contributing to control overfitting.

Typically a pooling layer is like a filter of predetermined size and pitch that computes the local maximum of a given region of the attribute map. Figure 102 shows a pooling layer of size 2x2 with a step of 2.

The pooling operation does not require any parameters to be passed, we assume that H divides H^l where H^l is the height of the current layer matrix and W divides W^l , where W^l is the width of the current layer matrix, i.e. the output of the convolution layer is the input to the pooling layer, the pooling output will result in a tensor of order 3 of size $H^{l+1} \times W^l \times D^l + 1$ being:

$$H^{l+1} = \frac{H^l}{H}, W^{l+1} = \frac{W^l}{W}, D^{l+1} = \frac{D^l}{D} \quad (50)$$

There are two types of pooling operations widely used: max-pooling and mean-pooling (also known as average pooling). The max-pooling function starts from the analysis of the largest value in a region (WU, 2017), because the maximum values in the feature map represent where features are located, and by applying the pooling layer a percentage of nonfeature information is discarded. The resulting matrix (image) can be called a pooled feature map. The average-pooling function in turn is responsible for computing the average value of a given region bounded by the filter.

CNN architecture models usually have numerous convolution stages, nonlinearity and convolution, nonlinearity and pooling layers followed by more convolution layers and fully convolution and fully connected layers.

9.2.1.11.10 Fully connected layers and flattening

Initially, before reaching the fully connected layers, it is necessary to convert the characteristic matrices resulting from the pooling layer into a column vector layer into a column vector that will serve as input to the classical ANN.

Fully connected layers are typically used as final layers, where unlike convolutional

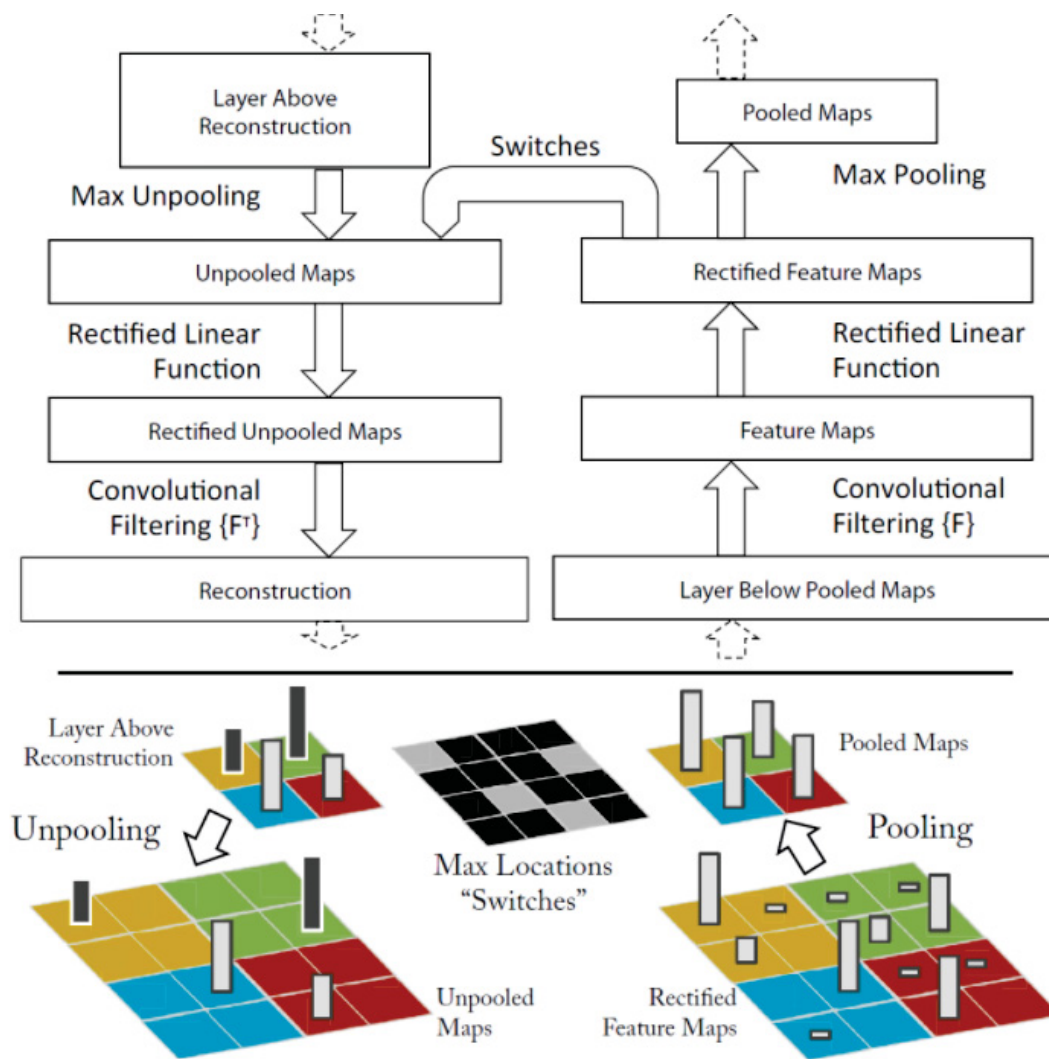


Figure 102: Pooling layers represented by a 2x2 kernel.

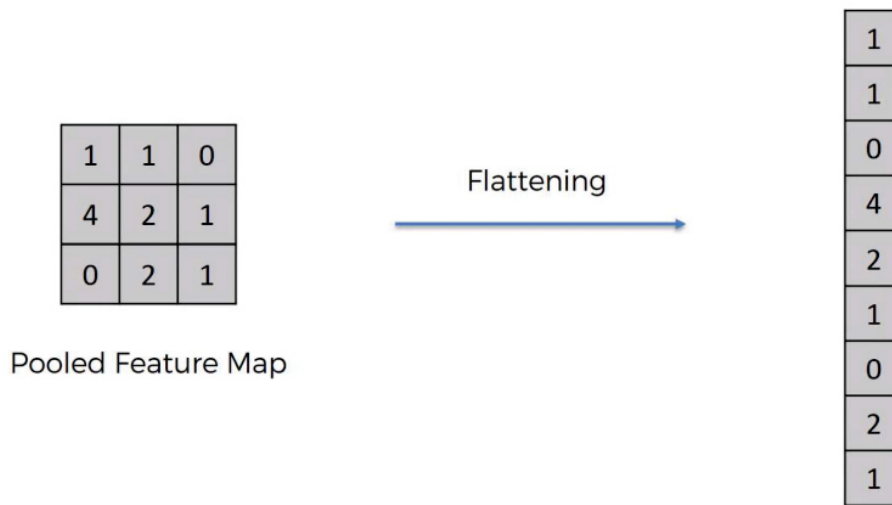


Figure 103: Representation of the flattening operation.

layers, where unlike convolutional layers, all weights are fully connected to the previous layer.

Here, an entire classical artificial neural network, that is, an entire Artificial Neural Network (ANN) is added to the end of the CNN, in which each element of the column vector created by the layer behaves as an input to a classical ANN, and then be connected to the fully connected hidden layers up to the final layer, as shown in fig. 104 layer. The purpose of an ANN after a flattening operation is to combine the features into more attributes, to improve the prediction performance of the classes. The error is calculated and propagated back as in an ordinary ANN and some parameters such as: the ANN weights (synapses) and filters (feature detectors) are adjusted to help optimize performance. Initially, the filters search for certain features, but these features may not be correct, then the filters are adjusted to look for new features.

Finally, fig. 105 shows a complete Convolutional Neural Network from its input image Neural Network, from its input image to its output class, which in the example example has only 2 classes.

9.2.1.11.11 Dropout regularization

The capacity of a neural network is based on the level of complexity of the function it needs to learn. Small neural networks, that is, networks with a relatively small number of parameters have a low learning capacity, a low complexity of resolution, and therefore will be more prone to underfitting. A problem in which the network is not able to learn the structure of more complex structure of more complex data, thus resulting in poor performance.

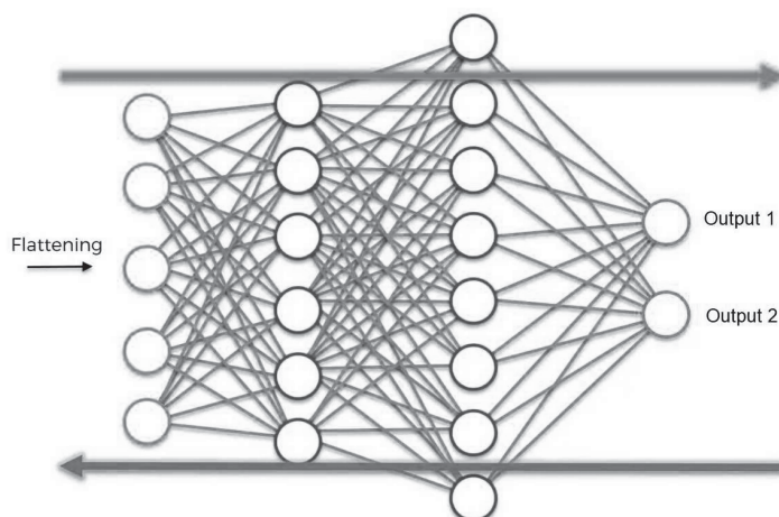


Figure 104: Totally connected layer – a classical ANN.

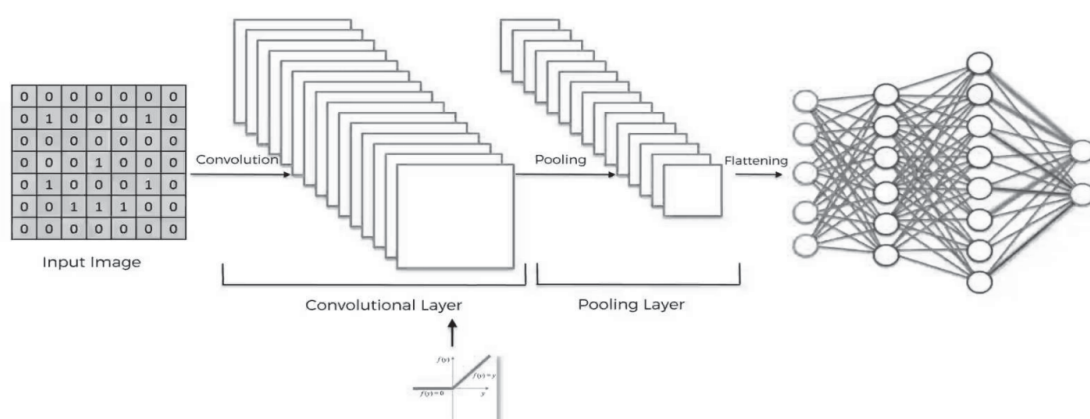


Figure 105: Simplified architecture of a CNN.

On the other hand, large networks with a high number of parameters may suffer from the net will memorize the training data, resulting in extremely good performance during the training phase, but will perform poorly against the more complex data. will perform poorly on the test set.

One way to address these problems is to build a relatively large net, a little larger than necessary, and to network, slightly larger than necessary to achieve good performance in the training phase. Then, in order to prevent overfitting one can apply one or several regularization schemes to achieve good generalization performance on new data.

In recent years, a new regularization technique has been prominent, called Dropout. Dropout is a technique in which randomly selected neurons are ignored during training. They are randomly discarded. This means that their contribution to the activation neuron is temporarily removed in the forward phase and any weight updates are not applied to the updates are not applied to the neuron in the backward phase.

9.2.1.12 Transfer learning

9.2.1.12.12 Definition

Migration learning is a machine learning approach that refers to a pre-trained model being reused in another task. Transfer learning is related to the problems of multi-task learning and concept drift, and it is not a specialized field of machine learning. However, migration learning is very popular in certain deep learning problems, such as when there are a large number of resources needed to train a deep model or a large dataset to pre-train the model. Migration learning works only when the deep model features in the first task are generalized features. This kind of migration in deep learning is called inductive migration. It is the process of narrowing the search for possible models in a favorable way by using a model that is applicable to a different but related task.

9.2.1.12.13 Method

Methodology for developing the model:

1. Select the source task. You must select a relevant predictive modeling problem with rich data, where there is some relationship between the input data of the original task and the target task, the output data, and the concepts learned from the mapping between the input and output data.
2. Develop the source model. Then, you must develop a refined model for the first task. This model must be better than a normal model to ensure that some feature learning can be performed.

3. Reuse the model. The model applicable to the source task can then be used as a starting point for learning the target task. This may involve using the first model in whole or in part, depending on the modeling technique used.
4. Adapt the model. The model can be selectively fine-tuned on the input-output pairs in the target dataset in order to adapt it to the target task.

Pre-trained model approach:

1. Selecting a source model. A pre-trained source model is selected from the available models. Many research institutions have published models based on very large datasets, and these can be used as alternatives to the source model.
2. Reuse models. The selected pre-trained model can be used as a learning starting point for the model used for the second task. This may involve using all or part of the same model as the training model, depending on the model training technique used.
3. Tuning the model. The model can be selectively fine-tuned on the input-output pairs in the target dataset to allow it to adapt to the target task.

The second type of migration learning is more commonly used in the field of deep learning.

9.2.1.13 Fine tuning

In addition to training the network from scratch, a more efficient and better way is to fine-tune the pre-trained network model.

1. Since the network converges on the original data, a small learning rate should be set to fine-tune on the target data set, which should be below the negative fourth power of 10.

2. Shallow layers of convolutional neural networks have more generalized features (e.g., edges, textures, etc.), while in-depth features correspond to higher-level semantics more abstractly. Therefore, the generalized features may be updated to a lesser extent when fine-tuning on new data, and the high-level semantic features may be updated to a greater area so that the learning rate can be set for different layers according to the layer depth: the learning rate for the deeper layers of the network can be slightly more significant than the shallow learning rate. This way, the overall training speed of the model may be faster.

3. The fine-tuning strategy is not used because of the degree of similarity between the target task data and the original data. When the target data is small, and the target data is very similar to the original data, only the last few layers of the network near the target function can be fine-tuned. When the target data is less, and the original data has a significant difference, this situation is more problematic, the success of fine-tuning specific

problems, but still can try to first fine-tune the network after a few layers in the fine-tuning of the entire network.

4. In addition, the third point mentioned in the target data is tiny. Still, with the original information having a significant difference, an effective way is to use part of the original data and the target data co-training. Since the shallow network features of the pre-trained model are more generalizable, the nearest neighbors of the target data can be selected in the external feature space as the original data itself, after which the fine-tuning stage is transformed into a multi-objective learning task, one by basing the target task on a subset of the original data, and the other by establishing the target task on all the target data.

Fine-tuning neural networks is currently an essential technique in scene classification competitions on kaggle. However, nowadays, the datasets given by big data competitions are small, so it is not only long to design a new neural network for training, but also the effect is not particularly good.

9.3 Part III

9.3.1 Non-maximum suppression (NMS)

Non-Maximum Suppression (NMS), as the name implies, is the suppression of elements that are not maximal, which can be understood as a top local search. This local represents a neighborhood, and the area has two parameters variable, one is the dimension of the area, and the other is the size of the neighborhood.

Non-maximum suppression is an edge refinement technique applied to "thinning" edges. After using the gradient calculation, the boundaries extracted from the gradient values are still very blurred. Regarding criterion 3, there should be an accurate response to the edges only. Therefore, non-maximum suppression can help suppress all gradient values except local maxima (by setting them to 0), which indicates the location with the most significant intensity value change. The algorithm for each pixel in the gradient image is.

1. Compare the edge intensity of the current pixel with the edge intensities of pixels in the positive gradient direction and the negative gradient direction.
2. If the edge intensity of the current pixel is maximum compared to other pixels in the mask with the same orientation (i.e., pixels pointing in the y-direction are compared to pixels above and below it, on the vertical axis), the value will be retained. Otherwise, the value will be suppressed.

9.3.1.1 Implementation procedure

In some implementations, the algorithm classifies the continuous gradient directions into a small set of discrete directions. It then shifts the 3×3 filters over the output of the previous step (i.e., edge intensity and gradient direction). Each pixel suppresses the edge intensity of the central pixel (by setting its value to 0) if its magnitude is not greater than the magnitude of its two neighbors in the gradient direction. For example Suppose the circular gradient angle is 0° (i.e., the edge is north-south). In that case, the point will be considered to be on the edge if its gradient magnitude is greater than the magnitude at the pixel in the east-west direction. If the circular gradient angle is 90° (i.e., the edge is in the east-west direction), the point will be considered to be on the edge if its gradient is greater than the magnitude at the north-south pixel.

Suppose the circular angle of gradient is 135° (i.e., the edge is located in the northeast-southwest direction). In that case, the point will be considered to be on the edge if its gradient is greater than the magnitude at the northwest and southeast direction pixels.

If the circular gradient angle is 45° (i.e., the edge lies in the northwest-southeast direction), the point will be considered to be on the edge if its gradient magnitude is greater than the magnitude of the northeast and southwest pixels.

In a more precise implementation, linear interpolation is used between two neighboring pixels that span the gradient direction. For example, suppose the gradient angle is between 45° and 90° . In that case, the interpolation between the gradients of the north and northeast pixels will give one interpolation, and the interpolation between the south and west-west pixels will give the other (using the last paragraph). The gradient magnitude at the center pixel must be greater than the gradient magnitude of both to mark it as an edge. Note that the sign of the direction is irrelevant, i.e., north-south is the same as north-south, and so on.

```

1 procedure NMS(B,c)
2    $B_{nms} \leftarrow 0$  (initialize empty set)
3   for  $b_i \in B$  do (Iterate over all boxes)
4     discard  $\leftarrow$  False (Discard or keep)
5     for  $b_j \in B$  do ( $b(i)$  loop comparison)
6       if  $\text{same}(b_i, b_j) > \lambda_{nms}$  then (if both have the same IOU)
7         if  $\text{score}(c, b_j) > \text{score}(c, b_i)$  then
8           discard  $\leftarrow$  True
9     if not discard then
10       $B_{nms} \leftarrow B_{nms} \cup b_i$ 
11  return  $B_{nms}$ 

```

In a few steps, it can be summarized as:

1. Select the proposal with the highest confidence score, remove it from B and add it to the final proposal list D. (Initially, D is empty).

2. Now compare this proposal with all the proposals — calculate the IOU (Intersection over Union) of this proposal with every other proposal. If the IOU is greater than the threshold N , remove that proposal from B .
3. Again, take the proposal with the highest confidence from the remaining proposals in B and remove it from B and add it to D .
4. Once again, calculate the IOU of this proposal with all the proposals in B and eliminate the boxes which have high IOU than the threshold.
5. This process is repeated until there are no more proposals left in B .

Below is a representation of the Intersection Over Union (IOU):

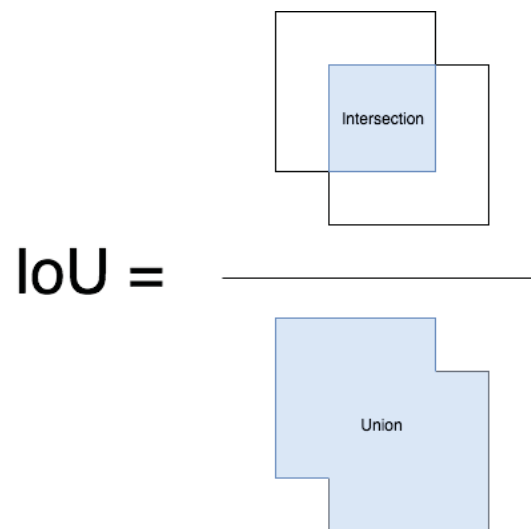


Figure 106: Representation of the Intersection Over Union (IOU) between two lanes.
Process used when selecting the best bounding box detection.

The effects of non-max suppression are represented in fig. 107.

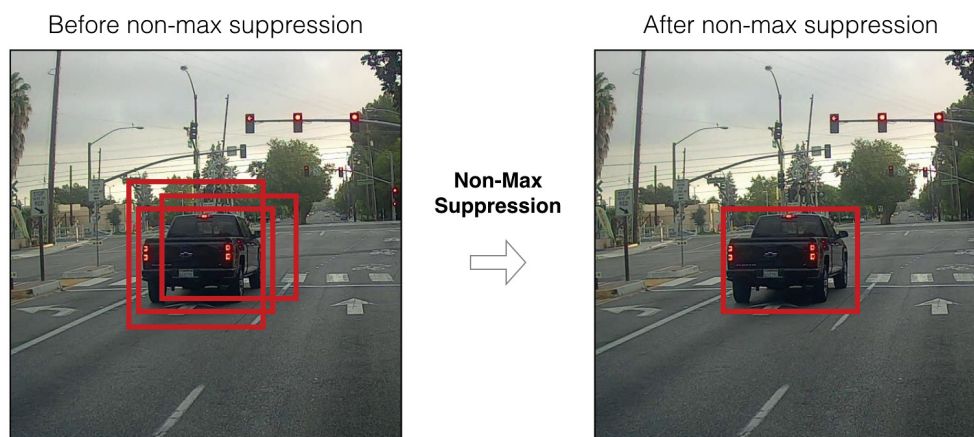


Figure 107: Representation of the non-max suppression.